APRENDA Qt4 DESDE HOY MISMO

Anónimo

Octubre de 2010 (versión de distribución libre)

2

APRENDA QT4 DESDE HOY MISMO

Este pretende ser el primer libro en español del mejor Framework de desarrollo multiplataforma de todos los tiempos, y que cada día nos brinda nuevas posibilidades. Es una lástima pero a día de hoy no hay ninguna traducción, ni libro dedicado seriamente a Qt 4 en nuestro idioma, por lo que me dispuse a cambiar esta situación.

El enfoque que vamos a usar es por supuesto desde C++, y por lo tanto es importante poseer de antemano unos conocimientos previos de este lenguaje. El nivel necesario para comprender este libro lo completa perfectamente el libro de *Bjarne Stroustrup* , "El lenguaje de programación C++" publicado en español por Addison-Wesley en 2001 (ISBN: 84-7829-046-X).

C++ es un lenguaje que ha evolucionado mucho desde que fue creado en 1983 por *Bjarne Stroustrup* en los laboratorios AT&T, donde también nació su predecesor C, unos diez años antes (1972), de la mano de Dennis Ritchie. El lenguaje C, que había nacido para sustituir al ensamblador en el desarrollo de sistemas operativos, y así poder portarlos más fácilmente a otras plataformas físicas, y así se reescribió completamente UNIX de los Laboratorios Bell creado en 1969. C++ surgió como una necesidad de añadir nuevos paradigmas a los ya incorporados por C, pero sin perder la potencia del mismo, al poder acceder al hardware de manera sencilla. Así C++ añadió el lenguaje orientado a objetos, por ello se le conoce como C con clases. Sin embargo cuando alguien que viene de C, comienza a estudiar C++, debe de evitar pensar que es una extensión del lenguaje C lo que está estudiando, ya que C++ es mucho más que eso. C++, se ha desvinculado de muchas prácticas habituales en C que eran motivo de múltiples fallos en la programación, como el casting incontrolado, el uso de la memoria, de los punteros, el paso de parámetros por referencia, incorporación de elementos muy útiles como las Colecciones o Contenedores y la creación de una librería de incalculable valor para el programador en C++, las STL (Standard Template Libraries). Aunque UNIX y Linux fue creado desde C, otros operativos como Windows, nacieron de C++. Los sistemas con entorno gráfico, eran mucho más sencillos de desarrollar usando el paradigma del lenguaje orientado a objetos, que la mera programación estructurada de C.

En 1991, dos programadores noruegos (Eirik Eng y Haavard Nord de Quasar Technologies, más tarde Trolltech) crearon un Framework en C++ que permitiera usar el mismo código en el desarrollo de una GUI (interfaz gráfica) tanto para Windows como para UNIX/X11. Inicialmente era código abierto, pero la licencia no era libre. Entre 1995 y 1998 se desarrolló con este Framework un escritorio para GNU/Linux que tuvo mucho éxito, llamado KDE que hoy sigue siendo uno de los escritorios más usados de Linux. GNU miraba con recelo que el escritorio más usado en Linux no fuera libre, así que hubo una reunión entre los desarrolladores de KDE y Trolltech, para asegurar el futuro de Qt en una liberación bajo licencia BSD. Entre los años 2000 y 2005 se produjo la liberación bajo licencia GPL de las versiones de Qt para Windows, Linux y Macintosh. En Junio de 2008, Nokia se aseguró el desarrollo para sus móviles tanto Symbian como Maemo (proyecto que al final se ha fusionado con Moblin de Intel, para crear MeeGo, y competir con Android de Google) con la adquisición de Qt.

Actualmente a finales de 2010, Qt acaba de salir en la versión 4.7 donde se añaden nuevas características extraordinarias para la creación de diseños de front-ends para dispositivos móviles y electrodomésticos, con la entrada de QML (Qt Declarative) un lenguaje declarativo para crear interfaces gráficos muy vistosos. Qt ya ha sido portado y soportado por Nokia para las siguientes plataformas: Linux/X11 (y otros UNIX, como Solaris, AIX, HP-UX, IRIX), Windows, Macintosh, Embedded Linux, Windows CE/Mobile, Symbian y MeeGo. Y con desarrollo externo se ha portado o se está portando a plataformas como: Open Solaris, Haiku(BeOS libre), OS/2(Plataforma eCS de IBM), iPhone y Android. Como podemos apreciar, Qt es un Framework que no para de crecer, y el secreto de este éxito, es no sólo su potencia y velocidad frente a otras alternativas como Java (donde es entre 3 y 6 veces más rápido), si no porque es atractivo para el programador y rápido para desarrollar, desde la salida de QtCreator 2.0, el entorno IDE para Qt.

Qt incorpora un estilo de programación a C++, que le hace menos vulnerable a los fallos de programación, sobre todo en el manejo de memoria (memory leaks), y que lo hacen muy agradable de cara al desarrollador.

¿Qué vamos a aprender en este libro?. Este libro pretende ser el primer escalón a subir para alguien de habla hispana que quiera aprender a desarrollar aplicaciones multiplataforma con Qt4. Su contenido ha sido escogido, para poder cubrir el desarrollo de cualquier aplicación de propósito general. Cualquiera que complete el contenido de este libro con sus ejercicios y ejemplos, podrá desarrollar este tipo de aplicaciones sin grandes problemas. Sin embargo, Qt crece todos los días, añadiendo nuevas posibilidades, nuevos módulos y nuevas clases, y a día de hoy es tan extenso que ningún libro en ningún idioma contiene una descripción completa del mismo. Por lo que la única fuente donde podemos recoger esa información, es la misma ayuda que el SDK del Framework nos proporciona con su asistente (Qt Assistant), pero esta documentación está escrita en inglés únicamente. Es por tanto necesario poder leer y ser capaz de desenvolverse en este idioma, para usar toda la capacidad que Qt nos pueda ofrecer.

Por lo tanto, este libro consigue sentar unas bases claras en el conocimiento de Qt4, pero para ir más allá es necesario saber usar la documentación en inglés que acompaña a toda versión del Framework. Los temas están ordenados de manera progresiva, para que nadie se pierda. El lenguaje usado es sencillo y directo, por lo que el contenido de ningún tema excede las 8 páginas de extensión, y por tanto puede cubrirse en un solo día. Acompañando a estos temas, está el código de los ejemplos, y los ejercicios resueltos, que nos complementan lo aprendido. No pase de un tema al siguiente, sin leer todo este código, comprenderlo, e intentar hacer los ejercicios que se proponen. Este libro propone una base fundamental para el conocimiento de Qt orientado al desarrollo de aplicaciones de propósito general, y sin dominar esta base sólida, no podrá seguir avanzando conforme a sus necesidades futuras en nuevos aspectos del Framework. Trataremos por tanto todos los aspectos fundamentales del desarrollo de aplicaciones GUI con todos sus componentes posibles, capacidades como el dibujo, impresión de documentos, acceso a bases de datos, acceso a la red, la programación concurrente, acceso al sistema de ficheros y el uso de los principales tipos de recursos.

Y lo mejor de todo, es que este libro es libre, y se podrá distribuir libremente y sin coste alguno. Tratamos de beneficiarnos con la creación de una comunidad de programadores en español, nutrida y prolífica. Por todo ello, bienvenido a esta maravillosa aventura que es aprender C++/Qt4.

Índice de Contenidos

TEMA 1

INSTALACION DEL QT SDK

A programar, se aprende programando, por lo que antes de comenzar a recibir una ingente cantidad de información teórica, hay que estar en la disposición de poder probar todo lo que se recibe de una manera práctica, para que así el ciclo didáctico se culmine por completo. Por ello, antes de nada, vamos a prepararnos un entorno de desarrollo lo suficientemente cómodo para poder acudir a él cada vez que lo necesitemos, mientras vamos aprendiendo nuevos conceptos.

Qt es un Framework completo multiplataforma, por lo que vamos a ver como se instala en los 3 tipos de sistemas más extendidos hoy día, a saber: Windows, Linux y MacOSX.

Bajarse el SDK de Internet

Qt se puede empezar a conocer e incluso a usar únicamente en su versión Open Source (GPL o LGPL), la cual podemos bajar en su última versión de la web http://qt.nokia.com/downloads .

De todos los archivos que allí se nos ofrece, recomendamos pulsar en la sección LGPL y bajarse las versiones "Complete Development Environment", que además son las más extensas, ya que no sólo incluyen el SDK, si no también el Qt Creator, que es el IDE de desarrollo, además de muchas otras herramientas de desarrollo, ejemplos y documentación, eso sí, en perfecto inglés. Por lo que es de gran ayuda al menos ser capaz de leer y comprender un poco el inglés informático.

Instalación en Windows 32 bits

Una vez nos hemos bajado el fichero qt-sdk-win-opensource-20xx.xx.exe (donde las xx.xx son números referentes a año y versión del empaquetado). Esta instalación lleva incluído el paquete MinGW que comprende el compilado y herramientas gcc , g++ y make.

Solamente hay que ejecutarlo, y seguir las instrucciones por defecto. Al final obtenemos un icono en el Escritorio del Qt Creator. En el próximo tema, veremos la estructura de lo que nos ofrece dicho IDE. No obstante puedes ver que en el menú de Aplicaciones, aparece el grupo "Qt SDK by Nokia …", y dentro de él, podemos ver herramientas como Qt Demo, donde se nos presenta una serie de aplicaciones de ejemplo, donde podremos ejecutarlas pues ya vienen compiladas para nuestra plataforma, y en la subcarpeta Tools, tenemos el Qt Assistant que será nuestro asistente y ayuda documental para buscar la información que necesitemos del SDK. El Qt Designer que nos servirá para diseñar GUIs, y el Qt Linguistic que nos ayudará a implementar traducciones a otros idiomas de nuestra aplicación.

El sistema ya está listo y preparado para ser usado en nuestro curso. Ya puedes ir directo al siguiente tema.

Instalación en Linux 32 y 64 bits

Aunque algunas distribuciones Linux pueden bajarse de sus repositorios oficiales una versión de Qt e incluso del Qt Designer, nosotros vamos a optar por instalar, la última versión completa en nuestra distro.

Para ello nos bajamos el fichero qt-sdk-linux-x86-opensource-20xx.xx.bin (donde las xx.xx son números referentes a año y versión del empaquetado), para Linux 32 bits o el fichero qt-sdk-linux-x86_64-opensource-20xx.xx.bin para Linux 64 bits. Antes de instalarlo, es necesario tener instalados GCC, G++ y Make, que en las versiones derivadas de Debian se hace con un mero "apt-get install build-essential" y en las basadas en Red Hat se hace con una "yum install gcc gcc-c++ make".

Un vez bajados los ficheros, desde root le damos permiso de ejecución (chmod +x), y lo ejecutamos desde la consola. Se abrirá una ventana de diálogo y podemos instalarlo en el directorio que queramos (por ejemplo /opt/qtsdk-20xx.xx/). Una vez se acaba la instalación, vamos a preparar el PATH para nuestra sesión de usuario que podrá acceder al IDE. Así por ejemplo en el directorio de usuario, en .bash_profile, añadiremos al PATH actual lo siguiente:

PATH=$PATH:/opt/qtsdk-2010.05/qt/bin: /opt/qtsdk-2010.05/bin

Volvemos a reiniciar la sesión, y ya podremos acceder desde la consola a las diferentes herramientas, como al Qt Creator, tecleando qtcreator, al Qt Demo, tecleando qtdemo, al Qt Assistant, tecleando assistant, al Qt Linguistic, tecleando linguistic.

En el Qt Demo, donde se nos presenta una serie de aplicaciones de ejemplo, donde podremos ejecutarlas pues ya vienen compiladas para nuestra plataforma. El Qt Assistant que será nuestro asistente y ayuda documental para buscar la información que necesitemos del SDK. El Qt Designer que nos servirá para diseñar GUIs, y el Qt Linguistic que nos ayudará a implementar traducciones a otros idiomas de nuestra aplicación.

Puedes hacerte unos disparadores (accesos directos) en tu escritorio, buscando el fichero en /opt/qtsdk-20xx.xx/qt/bin, y el icono en /opt/qtsdk-20xx.xx/bin (Nokia-QtCreator-128.png).

El sistema ya está listo y preparado para ser usado en nuestro curso. Ya puedes ir directo al siguiente tema.

Instalación en MacOSX

Bajado el fichero qt-sdk-mac-opensource-20xx.xx.dmg y previamente instalado XCode del disco de MacOSX correspondiente, sólo tenemos que instalarlo siguiendo las opciones por defecto del instalador.

Una vez terminada la instalación, podemos añadir accesos directos a nuestra barra a las herramientas que podremos encontrarlas en /Developer/Applications/Qt. Los ejemplos se han instalado en /Developer/Examples/Qt.

El sistema ya está listo y preparado para ser usado en nuestro curso. Ya puedes ir directo al siguiente tema.

Otras plataformas

La instalación en otras plataformas, no está explicada en este libro, ya que no se trata de un tema de carácter introductorio ni de propósito general. No obstante puedes obtener más información al respecto en http://doc.trolltech.com/4.6/supported-platforms.html y en otras secciones de la web de Qt Nokia.

TEMA 2

COMPILACIÓN

Una vez ya tenemos nuestros SDK y herramientas de desarrollo perfectamente instaladas, vamos a configurar el Qt Creator, y a explicar someramente su uso, compilando un ejemplo.

Antes de comenzar a programar, sería interesante destinar un directorio o carpeta del sistema local para guardar los ejemplos y ejercicios que vamos a hacer. Puede ser en cualquier carpeta o directorio, y dentro de esta se irán guardando en carpetas cada uno de los proyectos que vayamos usando. En esa misma carpeta puedes también descomprimir todos los ejemplos que van acompañando a este libro, de manera que si destinamos la carpeta /qtsrc para este menester, los ejemplos estarán en carpetas como /qtsrc/temaXX (donde XX es el número del tema).

Vamos por tanto ya a abrir el Qt Creator, ejecutándolo. Se nos abrirá el IDE con una ventana sobre él como ésta:

Podemos pulsar sobre Create Project... para crear un nuevo proyecto, u Open Project... para abrir uno preexistente. Si activamos el tab Develop, podremos ver las sesiones recientes que hemos abierto y así acceder a ellas directamente.

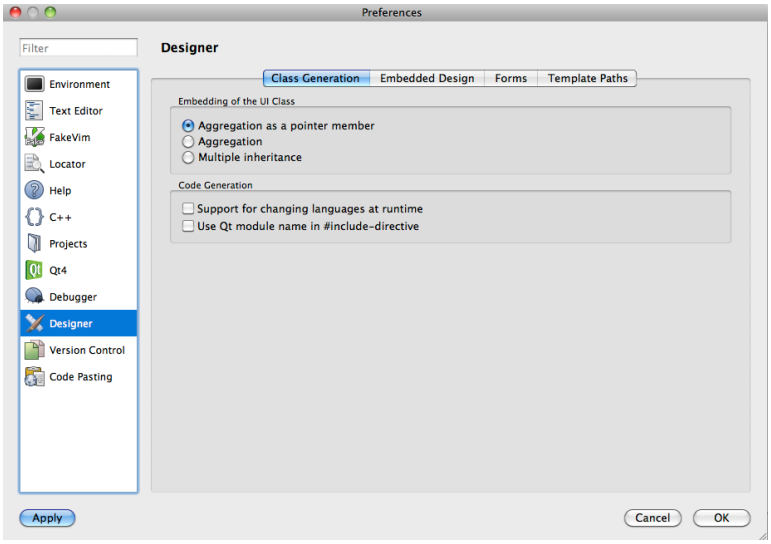
Bien vamos a pulsar en Open Project... y vamos ir a la carpeta tema02 y dentro de ella, vamos a entrar en la carpeta application, vamos a seleccionar el fichero application.pro, y vamos a abrirlo.

Al abrirse, podemos ver la disposición del Qt Creator. A la izquierda, hay una barra de herramientas, donde en la parte superior de la misma, se puede elegir la herramienta de desarrollo a usar. De arriba a abajo: Welcome (Bienvenida), Edit (editor, que es su forma habitual para introducir código), Design (acceso al Qt Designer, para crear la GUI editando ficheros tipo *.ui), Debug (para debugear un ejecutable), Projects (que nos permite configurar opciones del proyecto) y Help (para acceder al asistente). En la zona inferior de esta misma barra, encontramos un icono de una pantalla de ordenador, donde en la parte superior, indica el nombre del proyecto actual por defecto, y si el modo Build es Debug o Release (por defecto lo tendremos en Debug, hasta que vayamos a generar una aplicación para ser distribuida). Debajo están las herramientas de Run (ejecutar), Start Debug (comenzar debug) y Build All (construirlo todo).

A la derecha de esta barra de herramientas, pero aún en el lado izquierdo, nos queda en la zona superior, el árbol lógico de los ficheros que componen el proyecto. Podemos seleccionar dentro de él, cualquier fichero, y haciendo doble clic sobre él, abrirlo para edición. En la parte inferior aparecerán el listado de todos los ficheros que tengamos abiertos, de manera que podemos acceder directamente a ellos, pulsando sobre sus nombres, o cerrarlos, pulsando sobre la X que aparece si dejamos un tiempo el cursor sobre un nombre de fichero.

La parte derecha superior, es donde podremos ver el código de los ficheros, y en la inferior, los mensajes del compilador, debugger, etc.

Vamos ahora a configurar el entorno para trabajar con los ejemplos de nuestro libro. Para ellos entramos a las preferencias, que en Windows y Linux, está en Tools->Options, y en MacOSX, está en Qt Creator->Preferences. Nos aparecerá una ventana con opciones, las cuales vamos a dejar todas por defecto excepto en Projects, donde vamos a fijar el directorio para nuestros proyectos, y en Designer, nos vamos a cerciorar de tener la opción “Aggregation as a pointer member” y nada más (como puedes ver en el gráfico inferior).



Pulsamos en OK y nos vamos a Projects en la herramientas de la izquierda. Allí seleccionamos Editor Settings, y ponemos como Default File Encoding “UTF-8”.

Pulsamos ahora en Edit, y nos vamos abajo para pulsar en Run. De esta manera se compilará el proyecto actual, y se ejecutará una vez compilado. Nos aparecerá una ventana de un sencillo editor de texto con su barra de herramientas, que corresponde a la aplicación que acabamos de compilar y ejecutar. Si en la barra de la izquierda sobre la zona inferior, hubiera aparecido una barra en rojo con un número (el de errores), esto nos indicaría que hay errores en la compilación o en el linkado, por lo que habría que revisar que la instalación sea correcta, y la aplicación que se ha cargado.

En los siguientes temas, iremos profundizando en el uso del Qt Creator como IDE de desarrollo del SDK de Qt.

Tenga en cuenta que hemos preparado el entorno para desarrollar nuestras aplicaciones, pero aunque haya conseguido compilarlas en su versión Release (no Debug), no podrá portarlas tal cual a otra máquina para que funcionen tal cual, y habrá que tomar ciertas cuestiones en consideración para que se puedan implementar e instalar sin problemas en un sistema final de producción. Todo ello lo veremos con detenimiento en el tema 28. De momento, vamos a aprender a desarrollar, pero si se lleva los ejecutables a otra máquina sin el SDK instalado, sepa que no van a funcionar de antemano.

TEMA 3

ESTRUCTURA DE UNA APLICACIÓN QT4

Antes de comenzar a ver la primera aplicación en Qt4, vamos a ver cual es la estructura de una aplicación en Qt, de esta manera comenzaremos a ver el cuadro desde lejos, antes de empezar a adentrarnos en los detalles. Esta es la forma más deseable de comenzar un estudio que pretende ser los más aprovechable posible. Qt, posee una forma de ordenar el código, que desde el principio es fundamental conocerlo, para crear código sencillo y legible desde el primer momento. No importa lo sencillo o complejo que pueda ser un proyecto, si el código se estructura bien, siempre podrá ser mantenido de una manera más sencilla y eficaz.

Estructura de un proyecto

Todo proyecto con interfaz visual (GUI) en Qt, tiene los mismos elementos que son:

1.- *Ficheros de formularios (forms)*.- Con la extensión *.ui, cada fichero describe en lenguaje XML el interfaz gráfico de formularios o ventana de la aplicación. Por lo tanto cada fichero *.ui es una ventana, pero no todas las ventanas de la aplicación tienen que estar en un fichero *.ui, ya que se pueden implementar con código C++. Estos ficheros son diseñados gráficamente en el Qt Designer, el cual genera automáticamente el código XML de dichos ficheros.

2.- *Ficheros Cabecera (headers)*.- Con la extensión *.h . Lo más recomendable es que la declaración de cada clase vaya en un fichero cabecera por separado, y el nombre del fichero coincida con el nombre de la clase que se declara. Así por ejemplo, si vamos a desarrollar una clase llamada calculator, el fichero cabecera que la declara por completo la llamaríamos "calculator.h".

3.- *Ficheros Fuente (sources)*.- Con la extensión *.cpp . Lo más recomendable es que la implementación o desarrollo de cada miembro de una clase, esté contenido por completo en un fichero con el mismo nombre que la clase que implementa. Así por ejemplo, la implementación de la clase calculator, estaría en el fichero de nombre "calculator.cpp". Dentro de los fichero fuente, siempre hay uno que contiene la función principal (main) y cuyo nombre será "main.cpp".

4.- *Fichero de proyecto (project)*.- Con la extensión *.pro . Es un fichero que puede generarse y rellenarse automáticamente si hemos creado el código con el Qt Creator, o que puede no existir, y deba antes ser creador con el comando "qmake –project" antes de la compilación definitiva. El nombre del mismo coincidirá con el de la carpeta de proyecto creada. Si nuestro proyecto se llama calculator, el fichero proyecto tomará el nombre "calculator.pro".

Todos los ficheros de un proyecto Qt, es recomendable que se encuentren en un mismo directorio raíz, cuyo nombre coincida con el nombre completo del proyecto. Veamos ahora la estructura interna de cada uno de los tipos de ficheros mencionados.

Fichero formulario

Se trata de un fichero escrito en código XML con la descripción de todos los elementos del interfaz visual de uno de los formularios del proyecto. No es necesario conocer la estructura exacta del mismo, y de hecho, si en el Qt Creator, haces doble click sobre un fichero *.ui, se te abrirá directamente el Qt Designer, de manera que se mostrará la interfaz gráfica. En la barra de herramientas de la izquierda, tendrás que pasar del modo "Design" al modo "Edit" para poder ver el código XML que siempre tendrá la siguiente estructura.

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<ui version="4.0">
……

……

</ui>
```

Fichero cabecera (classname.h)

Vamos a partir de la idea de solamente declarar una clase en un solo fichero.

```
#ifndef CLASSNAME_H
#define CLASSNAME_H

#include <QClasePadre>
// podemos declarar aquí otras clases del proyecto que
// sean usadas en la implementación de esta clase
class Claseincluida;

class Classname : public QClasePadre
{
        // el contructor con un solo parámetro lo declararemos
        // siempre como explicit, y al menos con el puntero a la
        // clase padre inicializado a NULL
        explicit Classname (QClasePadre *parent = 0);

       // Cuerpo con la declaración de todos los miembros de la clase
      ……
      ……..
};
#endif
```

Como puedes observar, es muy recomendable el poner en la primera, segunda y última líneas del mismo, las condicionales al preprocesador para evitar la reinclusión de este código más de una vez. Después, incluímos con #include las clases Qt usadas en la implementación de esta clase, y la declaración de clases nuestras incluídas en este proyecto que son usadas en la implementación de esta clase. Finalmente tenemos la declaración de la clase misma con todos los elementos miembro.

Fichero fuente (classname.cpp)

```
#include "classname.h"

Classname:: Classname (QClasePadre *parent) :
 QClasePadre (parent)
{
        // implementación del contructor de la clase
}

// implementación del resto de miembros de la clase classname
……..
……
```

Primero incluímos la cabecera de nuestra clase. Luego implementamos el constructor, e inicializamos el constructor de la clase padre siempre lo primero. Después implementamos el resto de los miembros de la clase.

Fichero fuente principal (main.cpp)

```
#include <QModulo>
#include "claseinterfaz.h"

int main(int argc, char **argv)
{
    QApplication a(argc, argv);

    claseinterfaz w;
    w.show();
    // Mucho más código que sea necesario

    return a.exec();
}
```

Primeramente incluímos el módulo necesario para el código que vamos a incluir en la funcion main(). Lo más lógico es que vaya incluída la clase interfaz, que crea la ventana principal. Dicha ventana podrá ser un widget (QWidget), una ventana de diálogo (QDialog) o una ventana principal (QMainWindow). Primeramente creamos una instancia de QApplication que puede recibir argumentos de la linea de comandos. Luego instanciamos la clase interfaz, mostramos dicho interfaz, y finalmente con QApplication::exec() entramos en el bucle de eventos para interactuar con el interfaz mostrado.

La primera y última líneas de código de la funcion main(), sólo son necesarias si se trata de una aplicación con interfaz gráfico (GUI). Si se trata de una aplicación de consola, no se usarán, y en puesto de instanciar un clase interfaz, se haría uso de código no gráfico para consola. Haremos uso de programas para consola, para estudiar y reforzar clases de Qt que no son gráficas, para simplificar el código y facilitar el aprendizaje de los conceptos.

TEMA 4

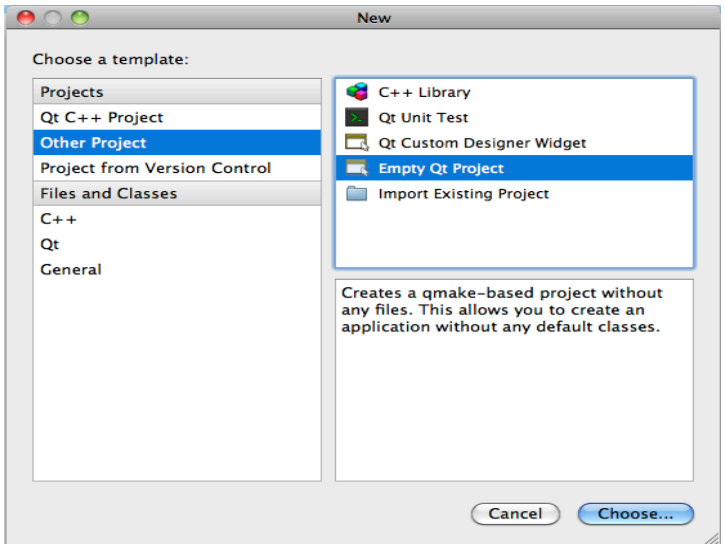
LA PRIMERA APLICACIÓN Qt

La mayoría de libros que enseñan un lenguaje de programación, siempre empiezan por demostrar la sencillez del lenguaje con la típica aplicación Hola Mundo. Y es una tradición que se ha perpetuado por los tiempos de los tiempos. Sinceramente me parece que es una trivialidad, ya que no enseña absolutamente nada sobre el lenguaje más allá de imprimir un mensaje en pantalla, pero en un mundo donde todos vamos corriendo y necesitamos ver los resultados antes de recorrer el camino, ha terminado imponiéndose. Bien es cierto que Qt no es un lenguaje, si no un Framework que en este libro en particular lo hemos dedicado al desarrollo de Qt en C++. Así que el lenguaje C++, es un conocimiento previo que se presupone antes de leer la primera línea de este libro, por lo que un Hola Mundo queda aún más descolocado, pero en fin, sucumbiremos a la tendencia generalizada.

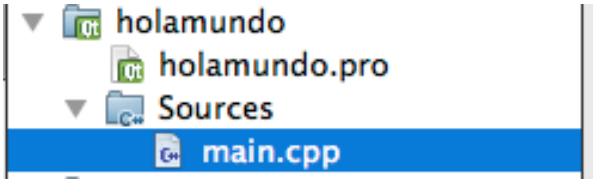
Para desarrollar dicha aplicación, vamos a usar el Qt Creator, y puesto que Qt se puede usar no sólo para crear interfaces gráficas, si no también aplicaciones de consola, vamos a ver las dos posibles aplicaciones Hola Mundo.

Hola mundo con GUI

Abrimos Qt Creator, o si ya lo tenemos abierto, abrimos un nuevo proyecto pulsando en File->New File or Project. Aparecerá una ventana como la de abajo.



Elegimos Other Project -> Empy Qt Project (un proyecto vacío). De esta forma no se nos creará ningún código automáticamente, todo lo haremos nosotros manualmente. Pulsamos el botón Choose… y aparece una nueva ventana, donde pondremos el nombre del proyecto que será "holamundo" y abajo en Create in, escogeremos un directorio donde tendremos todos nuestros proyectos. Pulsamos Continue y luego Done. A la izquierda en la columna de proyectos (Projects) nos aparece, la carpeta de nuestro nuevo proyecto, y sólo el fichero holamundo.pro que Qt Creator se encargará de ir rellenando por nosotros. Hacemos clic con el botón derecho sobre el proyecto y del menu emergente escogemos la penúltima opción (Add New…). Escogemos C++ en Files and Classes, y C++ Source File en la columna de la derecha. Pulsamos Choose… y le ponemos el nombre al fichero "main.cpp" en Name. Pulsamos Continue y Done. Ahora aparece una nueva carpeta llamada Sources con un fichero main.cpp dentro de ella.



Haciendo doble clic sobre main.cpp, podremos ver a la derecha el lienzo en blanco preparado para empezar a escribir el código en él. Vamos a escribir el siguiente código:

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

int main(int argc, char **argv){
    QApplication a(argc,argv);

    QLabel l("Hola mundo");
    l.show();

    return a.exec();
}
```

Como se puede ver hemos usado una clase QLabel para crear el letrero donde pondrá el mensaje "Hola mundo". Puesto que usamos dos clases en este fichero de Qt (QApplication y QLabel), hemos tenido que incluir sus cabeceras.

Hola mundo sin GUI

```
#include <QDebug>

int main(int argc, char **argv)
{
    qDebug() << "Hola mundo\n";

    return 0;
}
```

La clase QDebug es perfecta para sacar mensajes por consola, y tiene sobrecargado el operador << para que actue de manera similar a como lo hacía std::cout en la STL de C++.

Sigue los ejercicios de este tema donde se compila y se ejecutan estos 2 simples ejemplos.

TEMA 5

QObject, metaobjetos y propiedades

Llegó el momento de meternos en harina, y para ello mejor es empezar a conocer las particularidades de Qt que la hacen diferente a las STL. Para ello lo mejor es comenzar por la clase principal, padre de la mayoría de las clases que componen Qt4, esa clase es QObject. Luego veremos la estructura global de clases que componen el Framework y trabajaremos con una de las cualidades más interesantes de Qt, la *introspección*.

La clase QObject

Gran parte de la magia desplegada en el Framework Qt, deriva de la clase QObject, que es la clase principal. Todos los widgets usados para crear la GUI derivan de esta clase. De hecho Qt despliega gran parte de lo que Qt es: señales y slots, propiedades y manejo de memoria sencillo. Qt extiende a C++ con nuevas características, de las cuales dos son introducidas por esta misma clase como son: el modelo de introspección, y las conexiones entre objetos mediante señales y slots. Si revisas la documentación de QObject en el asistente (Qt Assistant), verás entre sus miembros todas estas características.

QObject, por tanto, es la clase base de la mayoría de las clases del framework Qt, sin embargo, no de todas. Las clases que no dependen de QObject son:

- Clases de primitivas gráficas que requieren tener poco peso, siendo más sencillas como: QGradient, QPainter, QScreen, QPalette…
- Clases contenedoras de datos como: QChar, QString, QList, QMap, QHash….
- Clases que necesiten ser copiables, ya que las clases derivadas de QObject no son copiables.

La primera características importante que tiene QObject, es que no es copiable, toda instancia de la misma es única. Todo objecto QObject es único e individual. Porque los objetos tienen nombre (propiedad *objectName*), y este es único para cada objeto, y esto es fundamental para implementar la introspección. Cada objeto, esta situado en algún lugar específico de la jerarquía de clases QObject, y eso no requiere ser copiable. Finalmente cada objeto, tiene conexiones con otros objetos mediante conexiones señal-slot que veremos más adelante, y eso no puede ser copiado. Por todas estas razones, los objetos de QObject, son únicos e incopiables en tiempo de ejecución como por ejemplo sí lo es un QString (una cadena).

C++ es extendido en Qt para dotarlo de características fundamentales para el framework. Una de ellas, son los ***metadatos*** .

Los metadatos, llevan información sobre el objeto instanciado, tales como el nombre del objeto, el nombre de su clase, sus objetos hijos, sus propiedades, sus señales y slots, toda una serie de informaciones sobre la clase del mismo. Esto nos permite una de las más importantes características de Qt, que es la introspección, que facilita mucho integrar Qt con otros lenguajes de scripting y entornos dinámicos (Ruby, Python, PHP, etc). Instanciado un QObject, podríamos obtener de él información sobre el mismo a través de los metadatos.

Una forma, sería mediante el método miembro QObject::inherits(const char* className), que nos permitiría hacer un casting dinámico sin usar el RTTI (lo cual puede evitar muchos errores del programador):

```
if(object->inherits("QAbstractItemView")){
        QAbstractItemView *view = static_cast<QAbstractItemView *>(widget);
        ............
    }
```

Uno de los miembros más importantes para acceder a los metadatos de una clase, y por tanto para implementar la introspección es ***metaObject()*** . Devuelve un objeto de la clase QMetaObject, que si miramos en el asistente todos sus miembros, podremos observar toda la información que podemos conseguir en *run-time* de un objeto cualquiera.

```
object->metaObject()->className(); // devuelve un char* con el nombre de la clase
object->metaObject()->indexOfProperty("width");
```

Los metadatos, son recabados por el moc (meta-object compiler) del propio código declarativo de las clases en los archivos cabecera. Lo primero que busca el moc es que la clase se derive de QObject de manera directa o indirecta, para buscar en ella todos los metadatos y generar el código necesario para ellos. Luego busca MACROS del tipo Q_ , como: Q_OBJECT, Q_CLASSINFO, etc; que añaden información de la clase. Finalmente busca palabras clave añadidas por Qt como slots y signals, que declaran los métodos miembros de la clase que se encargarán de la conectividad de la misma con objetos de otras clases. El moc con todo esto genera automáticamente código que guarda en ficheros que comienzan por moc_*.cpp .

Otra función miembro introspectiva muy importante que incorpora QObject es ***findchildren***. La firma de la misma es:

QList<T> QObject::findChildren (const QString &*name* = QString()) const

que nos devuelve una colección de objetos hijos de la clase cuyo nombre se pase como argumento. QList es una clase template que veremos más adelante, pero similar a una lista o vector de STL.

Por ejemplo, si queremos obtener los widgets que son hijos de un widget en particular a través de su nombre (widgetname), podemos usar un código como este:

```
QList<QWidget *> widgets = parentWidget.findChildren<QWidget *>("widgetname");
```

Por ejemplo, para obtener una lista con todos los botones del tipo QPushButton, contenidos como hijos de parentWidget, podemos usar un código como el que sigue:

```
QList<QPushButton *> allPushButtons = parentWidget.findChildren<QPushButton *>();
```

Propiedades de un objeto

Todo objeto, necesita guardar su estado en algún sitio, donde se defina por completo el estado del mismo y todas sus particularidades, esto es lo que llamamos ***propiedades*** de un objeto. En el Qt Designer, a abrir un fichero form (*.ui), podemos pulsar sobre cualquiera de los elementos que hay en el form, y a la derecha en la parte inferior, ver todas las propiedades que dicho objeto tiene, y los valores que esas propiedades toma para ese objeto en particular. Qt guarda esas propiedades como metadatos, y hace uso de una macro propia para ello:

```
Q_PROPERTY(type name
           READ getFunction
          [WRITE setFunction]
          [RESET resetFunction]
          [NOTIFY notifySignal]
          [DESIGNABLE bool]
          [SCRIPTABLE bool]
          [STORED bool]
          [USER bool]
          [CONSTANT]
)
```

Esta macro se pone en la declaración de la clase, en el fichero cabecera de la misma, después de la macro Q_OBJECT. Las partes entre corchetes ([]) no son obligatorias. Vamos a detallar cada parte de esta declaración:

type.- es el tipo de la variable que guardará el estado de dicha propiedad del objeto (QString, int…).

name.- es el nombre propio de la propiedad en cuestion (width, height….)

getFunction.- nombre de la función getter o que lee el valor de la propiedad (lo veremos más adelante)

setFunction.- nombre de la función setter o que establece o cambia el valor de la propiedad (lo veremos más adelante)

resetFunction.- nombre de la función que pone el valor por defecto a la propiedad.

notifySignal.- nombre de la señal (signal) que se producirá inmediatamente después de que el valor de la propiedad sea cambiado.

Designable.- nos dice si dicha propiedad es accesible desde el entorno de diseño, Qt Designer.

Scriptable.- nos dice si dicha propiedad es accesible desde el entorno de script de Qt.

Stored.- la mayoría de propiedades son guardadas, y establecen el estado de un objeto en un momento concreto, otras toman valores de otras propiedades (por ej. width es parte de size) y no son guardadas.

User.- es una propiedad modificable por el usuario de la aplicación (por ej. isChecked() en una QCheckBox).

Constant.- las propiedades declaradas constant, no pueden ser cambiadas en la instanciación de la clase, si no entre instanciaciones de clases.

La gran mayoría del tiempo usaremos meramente las secciones **READ** y **WRITE** para nuestras propiedades, y si nuestra clase es parte de un nuevo widget visual diseñado por nosotros, añadiremos **DESIGNABLE** true, y puede que **USER** true, si esta puede ser cambiada por el usuario que use la aplicación (no el programador).

La declaración de una nueva propiedad, implica una serie de añadidos en los miembros de la clase a la que pertenece, y estos son:

- El constructor debe de ponerle un valor inicial, la cual también podría ser un parámetro del mismo, y así poder inicializar su valor por parte del programador en su instanciación.
- Debe de haber una función miembro que se encargará de la recogida del valor actual de esa propiedad. Es lo que llamamos un getter. Y esta función debe estar en la parte pública para poder ser usada desde fuera de la clase misma, a la que llamamos setter de la propiedad. Sería bueno que se llamara getPropiedad() o Propiedad(), e isEstado_Propiedad() si devuelve un bool.
- Puede haber una función miembro que se encargue de establecer o cambiar el valor de propiedad. Sería bueno que se llamara setPropiedad(valor_de_la_propiedad).
- Debe de haber una variable privada, que contenga el estado de la propiedad en cada momento, de manera que esta no pueda ser directamente accesible fuera de la clase.

En una palabra, los getter y setter de una propiedad, son el interfaz de la propiedad de dicha clase con el resto de objetos, y sólo a través de ellos, el resto de la aplicación podrá acceder a dicha propiedad.

Pongamos un ejemplo sencillo de propiedad en una clase. Primero comenzaremos con la declaración de los miembros de la clase en su fichero cabecera.

```
// Fichero angleobject.h
class AngleObject : public QObject
{
        Q_OBJECT
        QPROPERTY(qreal angle READ angle WRITE setAngle)

   public:
        AngleObject(qreal angle, QObject *parent=0);

        qreal angle() const;
        void setAngle(qreal);

    private:
        qreal m_angle;
}
```

Si observas, verás que angle es una propiedad que se va a guardar en una variable del tipo qreal (que es un typedef de double en Qt, un número real de doble precisión). La variable la ponemos como privada bajo el nombre de m_angle (notación especial para distinguir variables miembro). La función getter es angle() que como vemos al no cambiar el estado del objeto podemos declararla como const. La función setter va a ser setAngle() y en la instanciación al crear un objeto nuevo de esta clase, tendremos que poner al menos como parámetro el valor inicial de dicha propiedad. Otra forma hubiera sido que dentro de la implementación del constructor, dicha propiedad hubiera tomado un valor fijo, que luego el programador hubiera podido cambiar usando la función setter correspondiente a dicha propiedad.

Para entrar un poco más en materia vamos a ver la implementación de esta clase:

```
AngleObject::AngleObject(qreal angle, QObject *parent) :
        QObject(parent), m_angle(angle)
{
}

qreal AngleObject::angle() const
{
        return m_angle;
}

void AngleObject::setAngle(qreal angle)
{
        m_angle = angle;
        hazAlgo();
}
```

Vemos como en la implementación del constructor, se inicializa llamando al constructor de QObject, la clase padre y la varible privada m_angle con el valor pasado como parámetro en la instanciación. Quizás no estés acostumbrado a este tipo de notación y lo veas mejor así:

```
AngleObject::AngleObject(qreal angle, QObject *parent)
{
  QObject(parent);
  m_angle = angle;
}
```

Pero es exactamente lo mismo. La implementación de la función getter es tan simple como devolver el valor de la propiedad que se almacena en la variable interna propia, y la implementación de la función setter, suele hacer dos cosas, primero cambiar el valor de la variable interna asociado a la propiedad en cuestión, y luego poner un código que se encargue de la notificación de dicho cambio, y la toma de decisiones al respecto (lo hemos representado todo eso como un mero hazAlgo()).

El usar un setter para cambiar el valor de una propiedad, y no acceder directamente a la variable privada del mismo, permite que un código pueda evaluar dicho valor previamente antes de asignarlo a dicha variable, y así controlar dicho valor dentro de un rango admisible dentro de la aplicación, y de esta manera hacer honor a la propiedad de encapsulación C++ como lenguaje orientado a objetos.

El tipo de una función podría ser un enum, algo así como:

```
public:
    enum AngleMode {Radianes, Grados};
```

En ese caso, tendremos que notificar previamente al moc de que, dicho tipo es un enum, usando la macro Q_ENUM de la siguiente forma:

```
Q_OBJECT
Q_ENUM(AngleMode)
Q_PROPERTY(AngleMode angleMode READ angleMode WRITE setAngleMode)
```

También es posible que ese enum, defina Flags numéricos que puedan combinarse entre sí con OR (|). Algo así como:

```
    Q_OBJECT
    Q_FLAGS(LoadHint LoadHints)
    Q_PROPERTY(LoadHint hint READ hint WRITE setHint)


public:
    enum LoadHint {
             ResolveAllSymbolsHint    = 0x01, // 00000001
             ExportExternalSymbolsHint= 0x02, // 00000010
             LoadArchiveMemberHint    = 0x04  // 00000100
         };
     Q_DECLARE_FLAGS(LoadHints, LoadHint)
```

Notación especial del código

Aunque C++ y Qt por tanto admita libremente la elección de los nombres de las clases, sus miembros y la organización de las mismas. Es adecuado para crear código fácil de leer, y por tanto fácil de mantener, si se siguen unas reglas que se han ido añadiendo al estilo de programación de Qt.

Todas las clases deben de estar definidas en ficheros aparte con el nombre de la clase en minúsculas, la declaración de la clase y sus miembros en un fichero *.h y la implementación de la misma en un fichero *.cpp .

Los nombres de las clases deben comenzar en mayúscula, y su nombre debe ser descriptivo de lo que representan, o del trabajo que van a realizar. Si es un nombre compuesto, es importante que cada parte del mismo se destaque con una mayúscula, por ejemplo VendedoresDeRopa. A veces si derivamos nuestra clase de otra preexistente de Qt, puesto que estas siempre comienzan por Q, puede ser de ayuda el llamarlas de la misma forma pero sin la Q inicial, claro esta, como:

```
class MainWindow : public QMainWindow
```

Todas las funciones comienzan en minúscula, y si su nombre es compuesto, el resto de partes las comenzamos con mayúsculas. Su nombre debe dejar claro, cual es su cometido dentro de la clase a la que pertenecen. Por ejemplo, setWidth(), isDigit(), dockWidgetArea().

Todas las variables miembro que están asociadas a propiedades que definen el estado del objeto, empiezan por m_. Por ejemplo m_height (para señalar la altura del objeto).

Como ya hemos señalado anteriormente, no es obligatoria este tipo de notación del código Qt, pero es muy aconsejable si quieres que todo el mundo entienda tu código de manera sencilla, incluido tú mismo unos meses o años después de haberlo escrito. Hacer anotaciones en el código es interesante siempre que se expliquen cosas que no salten a la vista en el mismo código. De hecho, comentar en exceso el código es la mejor forma de perder la visión global del mismo, por lo que una forma de evitar tener que comentar, es que las variables, clases y funciones tengan sigan una notación clara y tomen un nombre altamente descriptivo, aunque no muy largo.

Finalizados estos rudimentos, vamos a ver en el próximo capítulo, la parte más importante de una buena programación en C++, y es el manejo de la memoria. Una buena programación en este sentido nos evitará de muchos de los problemas propios de la memoria, como el tan temido “memory leak” o “fuga de memoria”. Es muy importante que tomes en consideración, y dediques el máximo esfuerzo al contenido del próximo tema.

TEMA 6

EL MANEJO DE LA MEMORIA

El mal manejo de la memoria, junto con el mal uso de punteros, es la fuente de la mayoría de los errores y problemas de aquellos que programan en C o en C++. Un programador de Java, nos dirá que esa es la gran desventaja de programar en C, y que ellos con su magnífico recolector de basura se evitan esos problemas. Sin duda es una buena solución para ellos, pero para aquel que ama C++, no es razón suficiente como para abandonar el desarrollo en este lenguaje tan potente y versátil a la vez. De hecho todo programador de sistemas que se precie, siempre antes o temprano tendrá que hacer uso de C, y la alternativa que quiero plantearte es usar el estilo Qt junto con su Framework para evitar éste y otros problemas típicos.

El manejo de la memoria en Qt

Antes de nada vamos a recordar a unos y a explicar a otros, algo que quizás en su momento no comprendieron. Cada vez que el sistema operativo carga en memoria un programa para ejecutarlo, el sistema reserva unas determinadas zonas de memoria para el mismo:

- La zona del código o segmento de código.- En ella se carga el código binario del programa que se va a ejecutar con todas sus instrucciones máquina tal cual el compilador las creó.
- La pila o stack. Esta zona es donde se van apilando variables como en una pila de platos, usando el método LIFO (last in, first out), el último que entra es el primero que sale. La pila es usada cada vez que el código llama a una función, y entra en ella. En la pila se guardan: los parámetros que se pasan a la función, las variables locales declaradas dentro del ámbito de la misma, y el valor de retorno de la función. Todas estas variables son destruidas una vez se vuelve de la llamada a la función. Lo que es lo mismo, fuera de las llaves que cierran el ámbito de la función, dichas variables ya no existen, son destruidas inexorablemente.
- El montón o heap. Esta zona se reserva con peticiones malloc() en C, y new en C++, y permanecen todo el tiempo que se quiera. El programador tiene la obligación de liberar esos espacios de memoria usando free() en C, o delete en C++. Si dicho espacio no es liberado, entonces el programa tiene una fuga de memoria (memory leak) y le puede conducir a un violento crash.

En la memoria, y principalmente en el heap, es donde va a estar el origen de gran parte de los desaguisados de nuestra programación en C++.

Veamos antes cuales son los problemas típicos en la gestión de la memoria:

a) Problemas en el stack:
- Leer o escribir fuera de los límites de una array estático. Esto se puede solucionar usando los contenedores que suministra Qt para manejar colecciones y usar iteradores para recorrerlos o bucles controlados con foreach, una nueva palabra clave de Qt.
- Corrupción de un puntero a función. Esto nos puede llevar a no poder alcanzar correctamente la función. La solución es no usar punteros a funciones, de hecho con el mecanismo de conexiones entre objetos mediante funciones señal y funciones slots, nos evitamos el uso de callbacks.

b) Problemas en el heap:
- Intento de liberación de memoria ya liberada.
- Liberación de memoria no asignada.
- Intento de escritura en memoria ya liberada.
- Intento de escritura en memoria no asignada
- Error en la asignación de memoria

- Lectura-escritura de la memoria fuera de los límites de un array dinámico. Este se puede resolver de igual manera al array estático en el stack.

Para resolver los cinco problemas señalados en el heap, aparece un estilo de programación que incorporaremos a Qt, y ese será el propósito del próximo parágrafo.

Herencia encadenada

La mejor forma de no tener que poner código para liberar memoria con delete, es que todos los objetos que componen la aplicación tengan un padre que o bien esté en el stack, o sean a su vez padre de otro objeto. De esta forma, ningún objeto que esté en el heap será huérfano.

Vamos a poner un ejemplo con los objetos que componen una ventana de diálogo.

```
QDialog *parent = new QDialog();
QGroupBox *box = new QGroupBox(parent);
QPushButton *button = new QPushButton(parent);
QRadioButton *option1 = new QRadioButton(box);
QRadioButton *option2 = new QRadioButton(box);
```

Como se puede ver, es un árbol jerárquico del que penden box y button de parent, y de box penden option1 y option2. El único objeto que no tiene padre, es parent, que sería la ventana de diálogo. Luego el truco para que no haya que añadir ningún código que libere su espacio, es colocar su creación en el stack.

Para ello vamos a tener que crear una clase derivada de QDialog, a la que llamaremos Dialog, y vamos a crear una instancia de la misma en el stack de esta manera:

```
#include <QApplication>
#include "dialog.h"

int main(int argc, char **argv){
    QApplication a(argc,argv);

    Dialog dialog;
    dialog.show();

    return a.exec();
}
```

Al derivar Dialog de QDialog, tenemos que declararlo en un fichero que llamaremos dialog.h, con el siguiente contenido:

```
#ifndef DIALOG_H
#define DIALOG_H

#include <QDialog>

class Dialog : public QDialog
{
  public:
    explicit Dialog(QWidget *parent = 0);
};

#endif // DIALOG_H
```

Es el mínimo código, ya que no vamos a redefinir ningún método, ni a añadir nada. Luego en la implementación, deberemos crear los objetos que dependen del objeto dialog en su constructor con new, de esa manera serán creados inmediatamente al ser creado dialog en el stack, y permanecerán en la memoria una vez acabado el ámbito del constructor, hasta que la aplicación se acabe, y al borrarse dialog del stack, por dependencia heredada serán liberados los espacios de todos sus hijos. De esta manera, hemos evitado el uso de código liberador, que nos

podría haber llevado a problemas con un código más grande y complejo. Veamos pues el código de implementación que pondríamos en el fichero dialog.cpp:

```
#include "dialog.h"

Dialog::Dialog(QWidget *parent) :
    QDialog(parent)
{
    QGroupBox *box = new QGroupBox(this);
    QPushButton *button = new QPushButton(this);
    QRadioButton *option1 = new QRadioButton(box);
    QRadioButton *option2 = new QRadioButton(box);
}
```

Es importante por lo tanto que te quedes con esta filosofía de programación, donde siempre, todo objeto que sea huérfano ha de crearse en el stack que requiera de acuerdo al tiempo que deba de vivir (en main, viviría todo el tiempo de la aplicación). Para ello sería necesario derivar dicho objeto de una clase derivada de una de Qt, generando así dos ficheros para implementar dicha nueva clase, y en el constructor de dicha clase, es donde crearemos en el heap los objetos que dependan jerárquicamente de éste. En una GUI es por tanto muy habitual el jerarquizar conforme al contenido. Así, si un objeto de una clase está contenido gráficamente dentro de los límites de un objeto de otra clase, ésta será la clase base de aquél.

Para ello, las clases de Qt, implementan varios constructores, donde al menos uno lleva sólo el parámetro del padre inicializado a NULL pointer por defecto, de manera que todas las versiones del constructor tiene como último parámetro el NULL pointer por defecto de la clase padre de la que deriva.

En el siguiente tema vamos a ver como en Qt se han evitado el uso de callbacks para conectar funciones de dos o más objetos. Es otra de las características fundamentales de QObject.

TEMA 7

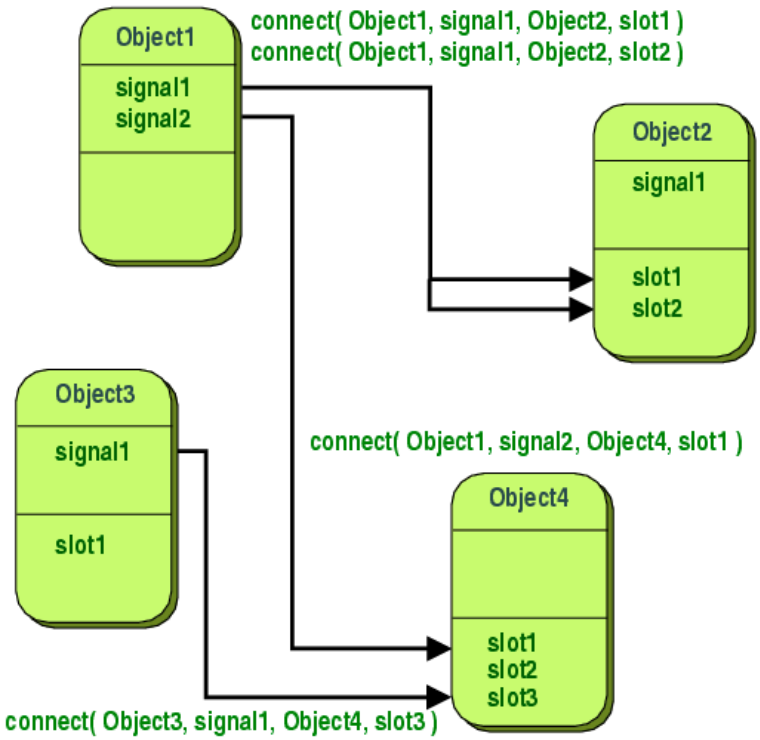
SEÑALES Y SLOTS

El mecanismo de señales y slots de Qt, crea una forma dinámica de comunicar eventos con los cambios de estado que estos provocan y las reacciones de los mismos. Lo excepcional de este mecanismo, es que los objetos que se comunican de esta forma, no requieren conocerse mutuamente. Para ello QObject la clase base de gran parte de las clases que conforman Qt, incorpora un método llamado ***connect***, que se encarga de establecer esa comunicación entre los dos objetos.

Todo objeto derivado de QObject puede poseer dos tipos de funciones propias especiales:

signals.- funciones que permiten emitir una señal cuando hay algún cambio de estado en el objeto al que pertenecen.

slots.- funciones que son el final de la conexión, y que ejecutan una serie de acciones una vez reciben el mensaje de la señal.



Como se puede ver en el gráfico superior, una señal puede conectarse a más de un slot, para llevar a cabo diferentes actividades. También varios señales diferentes pueden conectarse con un mismo slot, y tendríamos una actividad que puede ser desatada de varias formas diferentes.

Veamos como es la firma del método connect:

```
bool QObject::connect ( const QObject *sender, SIGNAL(*signal), const QObject *receiver, SLOT(*slot) )
```

Usamos las macros SIGNAL y SLOT para envolver a las funciones signal y slot con los tipos de sus parámetros.

Pongamos un ejemplo, claro. La clase QApplication, tiene un slot llamado quit() que simplemente hace que la aplicación se cierre y termine. Por otro lado la clase QPushButton, que sirve para implementar botones en el interfaz gráfico, posee una signal llamada clicked(), que es emitida cuando el estado del botón cambia al ser pulsado por el usuario. Así que podríamos crear una conexión entre el botón y la aplicación que conecten la señal clicked() del botón, con el slot quit() de la aplicación, y en consecuencia una vez establecida tal conexión, al pulsar el botón, eso haría que la aplicación se cerrará y terminará.

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

int main(int argc, char **argv){
    QApplication a(argc,argv);

    QPushButton button("Salir");
    button.show();
    QObject::connect(&button,SIGNAL(clicked()),&a,SLOT(quit()));

    return a.exec();
}
```

Bien, las propiedades de una función slot son las siguientes:

- Es implementada como una función ordinaria.
- Puede ser llamada como una función ordinaria.
- Un slot puede ser declarado en la zona private, protected o public de una clase, pero eso sólo le afectará si es llamada como función, no como slot. Como slot siempre podrá ser conectada con objetos de otra clase independientemente de la zona donde se haya definido en su clase. Y se declara en la seccion con la palabra slots.
- Como función puede devolver un valor de retorno, pero nunca en conexiones, sólo cuando es llamada como función.
- Cualquier número de señales puede ser conectadas a un mismo slot.

Las propiedades de una señal son las siguientes:

- Una señal es declarada en la sección con la palabra signals.
- Una señal es una función que siempre retorna void.
- Una señal nunca puede ser implementada, el moc se encarga de hacerlo automáticamente.
- Una señal puede ser conectada a cualquier número de slots a la vez.
- Es como una llamada directa, y por tanto es segura tanto en llamadas entre threads (hilos) como entre sockets (red).
- Los slots son activados por las señales en orden arbitrario.
- Una señal es emitida desde cualquier parte del código de la clase por la palabra emit (emit nombre_signal(params);).

Es importante tener en cuenta las firmas de las señales y los slots que son conectados, ya que como regla Qt no permite crear, ni convertir valores entre los parámetros, por lo que no se puede conectar cualquier señal con cualquier slot, si no entre compatibles entre sí, conforme a la citada regla.

Pondremos ejemplos de compatibilidad:

rangeChanged(int, int) ----------> setRange(int, int)
rangeChanged(int, int) ----------> setValue(int)
rangeChanged(int, int) ----------> updateDialog()

Y ahora de incompatibilidad:

clicked()----------> setValue(int)

textChanged(QString) --------> setValue(int)

Vamos a ver un ejemplo de implementación de una clase con signals y slots:

```
// Fichero angleobject.h
class AngleObject : public QObject
{
  Q_OBJECT
  Q_PROPERTY(real angle READ angle WRITE setAngle NOTIFY angleChanged)

public:
  AngleObject(qreal angle, QObject *parent=0);
  qreal angle() const;

public slots:
  void setAngle(qreal); // los setters son perfectos slots

signals:
  void angleChanged(qreal); // declarado en NOTIFY Q_PROPERTY arriba

private:
  qreal m_angle;
}


// Parte del fichero angleobject.cpp
void AngleObject::setAngle(qreal angle)
{
    if(m_angle == angle) return; // nunca olvides esta parte

    m_angle = angle;
    emit angleChanged(m_angle);
}
```

Como puedes ver en la implementación del setter, también slot, es donde tenemos que, primeramente para evitar que se creen bucles infinitos al conectar signal y slot, revisando si el valor que entra de angle, es nuevo o no, y si no es así, salir directamente. Si, el valor es nuevo, entonces primeramente actualizaremos el estado del mismo y finalmente emitiremos la señal de que ha habido un cambio en dicho estado. Esta es la forma en como se implementan señales y slots en las clases derivadas de QObject.

Es muy típico que dos widgets estén conectados entre sí, de manera que la señal de uno este conectado con el slot del otro. Vamos a ver un ejemplo completo de un convertidor de temperaturas de grados Celsius a Fahrenheit y viceversa.

Vamos a aprovechar este desarrollo para usar todo lo aprendido hasta el momento, y de esta manera ir avanzando en el desarrollo de aplicaciones Qt con interfaz gráfico.

Conversor de temperaturas

Vamos a abrir un nuevo proyecto en el Qt Creator, de nuevo un Empty Project (un proyecto vacio), y le vamos a llamar “tempconverter”. Con ello ya tenemos la carpeta tempconverter creada, conteniendo el fichero del proyecto que el Creator ha generado él solo.

Ahora vamos a añadir un fichero fuente main.cpp, haciendo clic derecho sobre la carpeta, eligiendo Add New… , luego C++ -> C++ Source File en la ventana emergente. Ponemos main.cpp en Name.

Ahora creamos Nuevo, y esta vez elegimos C++ -> C++ Class. Como class name le ponemos "TempConverter", y como Base class ponemos QObject. Le damos a continue, y done y se nos han creado los ficheros tempconverter.h y tempconverter.cpp con parte de código generado automáticamente.

Vamos a ver en primer lugar el código que describe la clase y sus miembros en el fichero cabecera correspondiente.

```
#ifndef TEMPCONVERTER_H
#define TEMPCONVERTER_H

#include <QObject>

class TempConverter : public QObject
{
    Q_OBJECT
  public:
    TempConverter(int tempCelsius, QObject *parent = 0);

    // funciones getter
    int tempCelsius() const;
    int tempFahrenheit() const;

  signals:
    void tempCelsiusChanged(int);
    void tempFahrenheitChanged(int);

  public slots: // funciones setter y slots
    void setTempCelsius(int);
    void setTempFahrenheit(int);

  private:
    int m_tempCelsius;
};
#endif // TEMPCONVERTER_H
```

Es suficiente con usar una sola variable como descriptora del estado de la temperatura, ya que la temperatura es una, se escriba en un sistema o en otro, sólo cambiar el valor de número que la representa. Tomamos por tanto la escala Celsius como la principal. Sin embargo vamos a usar de dos funciones setter, para que sea una de ellas la que haga el cambio. Por ejemplo setTempFahrenheit, será la que haga el cambio a Celsius, para guardar.

Puesto que ambos diales están interconectados entre sí, es decir, si cambia uno, inmediatamente debe cambiar el otro y viceversa, para evitar caer en un bucle infinito, tenemos que cuidar en el setter poner antes de nada el código que revise el cambio de estado. Como el estado lo hemos definido en Celsius, vamos a hacer esto en el slot setTempCelsius.

El código fuente de la implementación de esta clase quedará de esta forma.

```
#include "tempconverter.h"

TempConverter::TempConverter(QObject *parent) :
    QObject(parent)
{
    m_tempCelsius = 0;
}

int TempConverter::tempCelsius()
{
    return m_tempCelsius;
}

int TempConverter::tempFahrenheit()
{
    int tempFahrenheit = (9.0/5.0)*m_tempCelsius+32;
    return tempFahrenheit;
}

void TempConverter::setTempCelsius(int tempCelsius)
{
    if(m_tempCelsius == tempCelsius) return; // cortamos el bucle
```

```
    m_tempCelsius = tempCelsius; // actualizamos el nuevo estado
    // emitimos las dos se_ales del cambio ocurrido

    emit tempCelsiusChanged(m_tempCelsius);
    emit tempFahrenheitChanged(tempFahrenheit());
}
void TempConverter::setTempFahrenheit(int tempFahrenheit)
{
    int tempCelsius = (5.0/9.0)*(tempFahrenheit-32);
    setTempCelsius(tempCelsius);
}
```

La clase TempConverter que acabamos de definir va a ser la encargada de convertir los valores de Celsius a Fahrenheit y viceversa. Pero vamos a definir ahora el interfaz con los widgets que van a permitir cambiar la temperatura mediante unos diales, y mostrarla mediante unos LCD numbers. Como aún no hemos estudiado las ventanas de diálogo con widgets y layouts, vamos simplemente a definir el código de las conexiones implicadas, y cuando lleguemos al tema 14, entonces aprovecharemos para hacer el interfaz gráfico de este ejemplo.

Bien, queremos que cuando giremos los diales, automáticamente se cambie la temperatura de ambos LCD cada uno con sus valores de acuerdo a su escala de temperaturas, y que ambos diales giren el uno por acción del otro.

Bueno pues vamos a ver, cuales son las señales y slots que vamos a usar en cada elemento gráfico:

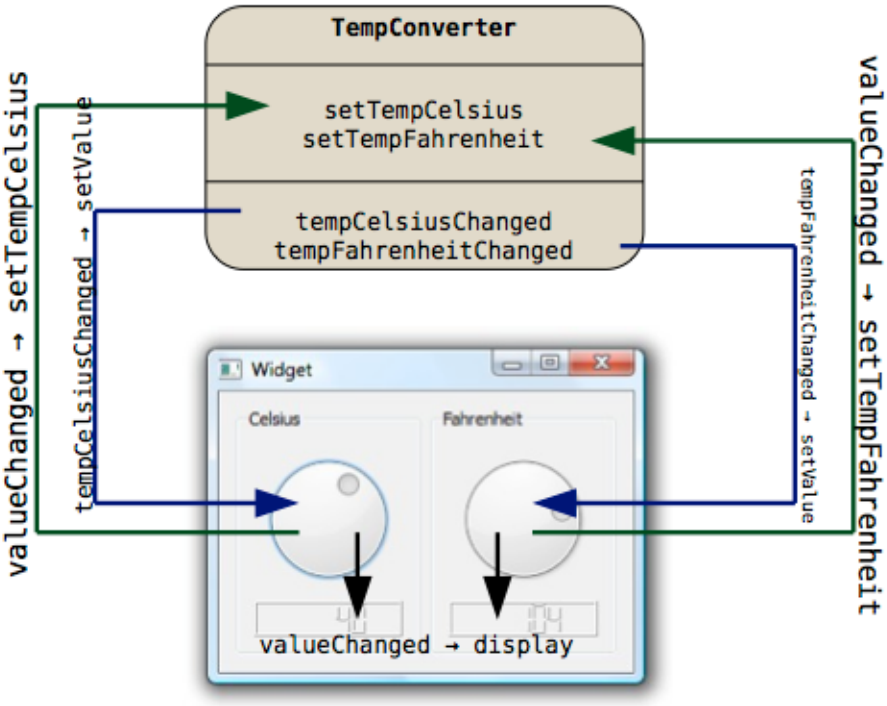
El dial Celsius (celsiusDial) => Señal -> valueChanged(int) ; Slot -> setValue(int)

El dial Fahrenheit (fahrenheitDial) => Señal -> valueChanged(int) ; Slot -> setValue(int)

El LCD Celsius (celsiusLcd) => Slot -> display(int)

El LCD Fahrenheit (fahrenheitLcd) => Slot -> display(int)

La forma en como se conectarían estos 4 widgets gráfico con la clase conversora TempConverter, sería como vemos en el gráfico.



La acción comenzaría al girar uno de los diales, supongamos por ejemplo, el de los grados Celsius. Esto dispararía la señal de celsiusDial, valueChanged(int), que iría a 2 sitios. Al celsiusLCD directamente mediante su slot display(int), y a TempConverter, a su slot setTempCelsius(int). Como podemos ver en el código, éste ultimo slot emite las 2 señales tempCelsiusChanged(int) y tempFahrenheitChanged(int), cada una conectada con los slots correspondientes de celsiusDial y fahrenheitDial, setValue(int) en ambos casos. De esa manera, tenemos todo perfectamente interconectado. Vamos por tanto a ver el código que implementaría esas conexiones.

```
connect(celsiusDial,SIGNAL(valueChanged(int)),tempConverter,SLOT(setTempCelsius(int));
connect(celsiusDial, SIGNAL(valueChanged(int)), celsiusLcd, SLOT(display(int));
connect(tempConverter,SIGNAL(tempCelsiusChanged(int)),celsiusDial,SLOT(setValue(int));

connect(fahrenheitDial,SIGNAL(valueChanged(int)),tempConverter,SLOT(setTempFahrenheit(
int));
connect(fahrenheitDial, SIGNAL(valueChanged(int)), fahrenheitLcd, SLOT(display(int));
connect(tempConverter,SIGNAL(tempFahrenheitChanged(int)),fahrenheitDial,SLOT(setValue(
int));
```

TEMA 8

QSignalMapper

Una vez hemos visto como funciona el sistema de conexiones entre objetos en Qt mediante señales y slots, vamos ahora a plantearnos un problema más complejo de afrontar mediante este esquema de trabajo.

Es muy normal, que en algunos casos, nos lleguemos a plantear el enviar un valor junto con la señal, sin embargo esto no es posible. Por ejemplo, supongamos que tenemos un conjunto de teclas representando los números del 0 al 9, como un marcador telefónico, o los numerales de una calculadora. Nos gustaría sin duda el poder enviar junto con la señal del clicked() el valor de la tecla pulsada.

Claro una forma de implementar este caso, sería crear 10 slots para cada tecla, pero eso supone repetir casi el mismo código diez veces, y si alguna vez nos planteamos hacer algún cambio, tendremos que estar atentos en hacerlo en 10 sitios diferentes.

Bien, para estos casos existe una clase muy útil llamada QSignalMapper. Si buscas información de la misma en el asistente, verás que es bastante sencilla en cuanto a miembros, slots y signals, sin embargo, como intermediaria de un sistema de transmisión de señales donde necesitamos enviar de alguna manera también un valor junto con la señal, es muy interesante.

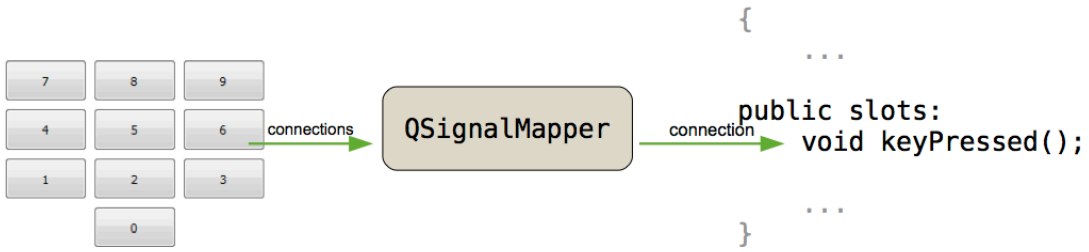
```
QSignalMapper *m = QSignalMapper(this);
QPushButton *b;

b=new QPushButton("1");
connect(b, SIGNAL(clicked()), m, SLOT(map());
m->setMapping(b,1);

b=new QPushButton("2");
connect(b, SIGNAL(clicked()), m, SLOT(map());
m->setMapping(b,2);
.................

connect(m, SIGNAL(mapped(int)), this, SLOT(keyPressed(int)));
```

Vamos conectando cada botón mediante la señal clicked() al slot map() del QSignalMapper. Mediante el método setMapping lo que hacemos es vincular al objeto del que recibe la señal el QSignalMapper, y que una vez, reciba la señal en el slot map(), éste generará una señal mapped(int) que lleva el valor entero asignado con setMapping a dicho objeto. Como vemos en el último connect, conectamos dicha señal mapped(int) con el slot keyPressed(int) que podemos ya redefinirlo para llevar a cabo las acciones que queramos, de acuerdo al botón pulsado. El gráfico inferior, representaría las relaciones implicadas en este método de conectar elementos.



TEMA 9

MANEJO DE CADENAS EN Qt

Antes de seguir con la programación gráfica (GUI), veo muy importante el conocer toda una serie de elementos fundamentales que usaremos en los programas con entorno gráfico, y que nos permitirán hacer aplicaciones útiles. Muchos libros de Qt, enfocan siempre sólo el aspecto gráfico, e incorporan estos elementos junto con aquellos, de manera que a veces pueden agobiar al que empieza, con multitud de conceptos, por ello aquí he preferido separar ambos aspectos completamente, y por tanto incluso los ejemplos que vamos a usar, van a ir desprovistos de GUI, y su salida será en la consola.

Ya hemos visto en el tema 4, el ejemplo de Hola Mundo con salida a consola, por lo que usaremos qDebug(), para nuestros ejemplos.

Qt es un Framework, cuya principal motivación es la de que usando el mismo código, podamos compilarlo en diferentes plataformas sin tener que hacer cambios. Por ello, ha tenido que abstraer muchos de los tipos que ya existían en C++/STL para hacerlo completamente multiplataforma. Así ocurre con el tratamiento de cadenas, en el que std::string ha sido mejorado, añadiéndole capacidad de representar caracteres unicote (chino, hebreo, árabe, etc), y de esta manera internacionalizar el ámbito aplicable del tratamiento de cadenas. Esto se ha hecho, creando una clase llamada **QString**. Si la buscas en el Qt Assistant podrás ver todas las posibilidades que esta clase nos brinda para el manejo de cadenas que en C, siempre fue complicado, frente a otros lenguajes de programación. Así, QString soporta las conversiones a otros sistemas (QString::toAscii, QString::toLatin1 …), permite la representación en unicode de prácticamente la totalidad de sistemas de escritura actuales y también permite métodos de inspección y manipulación de cadenas de texto.

Construcción de cadenas

Hay tres métodos para construir cadenas por concatenación:

1.- Mediante el operador +

```
QString res = "Hola " + "amigos ";
```

2.- Mediante el operador % de QStringBuilder

```
#include <QStringBuilder>
..............
QString str = "Hola " % "amigos";
```

3.- Mediante el método arg de QString

```
QString str = QString("Hola %1, tengo %2 años").arg("amigos").arg(42);
```

El método 1, es mejor que el 2 si se va a componer la cadena en muchas veces, con muchas asignaciones. El método 3 permite formatear la cadena, y dejar los huecos que luego serán rellenados por orden, por el método arg que permite introducir tanto cadenas como números.

Subcadenas

Vamos a ver una serie de métodos para tratar subcadenas dentro de cadenas más grandes.

Funciones left, right, mid, replace:

```
QString str = "Hola mundo";
str.left(4); // "Hola"
str.right(5); // "mundo"
str.mid(3,5); // "a mun"
str.replace("mundo","tronco"); // Hola tronco
```

qDebug() y QString (salidas por consola)

qDebug se usa para sacar mensajes por consola. Incluído en la clase QDebug, tiene diferentes formas de trabajar.

Como printf() en C pero añadiendo un retorno de carro "\n". Para imprimir un QString antes debe ser pasado por la funcion qPrintable incluida en QtGlobal (no requiere por tanto incluir ninguna cabecera para su uso):

```
QString str = "Antonio";
qDebug("Hola me llamo %s, y tengo %d años",qPrintable(str),42);
```

También podemos usarlo como un stream, como std::cout de C++/STL:

```
qDebug() << "Hola me llamo " << str << ", y tengo " << 42 << " años";
```

Números y cadenas

Convertir un número en una cadena:

```
QString years = QString::number(42); // "42"
```

Convertir de cadena a número:

```
bool ok;
years.toInt(&ok); // ok = true si lo convirtió bien
```

También existe toDouble(), toFloat(), etc.

STL y QString

Si usamos una librería creada con las STL, y queremos hacer un interfaz con ella, para recibir cadenas de ella, o enviarle cadenas hacia ella, entonces tendremos que adaptar la cadena para que todo funcione bien. Para ello existen unas funciones que vamos a ver.

```
std::string str = "Hola mundo";
QString qstr = QString::fromStdString(str);
```

```
QString qstr = "Hola mundo";
std::string str = QString::fromStdString(qstr);
```

Se supone que str está en ASCII.

Cadenas vacías y cadenas nulas

Decimos que una cadena es nula, si no está apuntando a ninguna cadena, y decimos que está vacía si no contiene ningún carácter.

Quedará aclarado con este ejemplo:

```
 QString str = QString; // str no esta apuntando a nada
str.isNull(); // true
str.isEmpty(); // true

QString str = ""; // str esta apuntando a un espacio vacío
str.isNull(); // false
str.isEmpty(); // true
```

Romper y unir cadenas

Métodos split y join.

```
QString str = "Madrid - Barcelona - Bilbao - Sevilla";
QStringList ciudades = str.split(" - "); /* ciudades = {Madrid, Barcelona, Bilbao,
Sevilla} */
QString new_str = ciudades.join(", "); // "Madrid, Barcelona, Bilbao, Sevilla"
```

QStringList es un contenedor, similar a una array de cadenas, que vamos a ver ahora.

QStringList

Es una clase especializada en contener QStrings en un contener tipo lista (QList). En realidad es una QList<QString>.

Esta lista puede ser rellenada usando el operador << como en un stream.

```
QStringList verbos;
verbos = "correr" << "comer" << "coser"; // {correr, comer, coser}
```

QStringList posee métodos para tratar, todas las cadenas que lleva consigo, que son muy interesantes:

```
verbos.replaceInStrings("er","iendo"); // {corriendo, comiendo, cosiendo}
verbos.sort(); // {comer, correr, coser}
verbos.filter("m"); // {comer}
verbos << "comer"; // {comer, comer}
verbos.removeDuplicates(); // {comer}
```

Iterando sobre QStringList

Podemos movernos por una QStringList usando el operador [] con ayuda de lenght() para no sobrepasar sus dimensiones, o con la propiedad de solo lectura at().

```
QStringList verbos;
verbos = "correr" << "comer" << "coser"; // {correr, comer, coser}
for(int i; i < verbos.length(); i++){
    qDebug() << verbos[i]; // verbos.at(i);
}
```

Pero la forma más cómoda y segura de recorrer una QStringList es con foreach, incluído en QtGlobal por defecto.

```
foreach(const QString &verbo, verbos){
        qDebug() << verbo;
    }
```

Usamos una referencia const para optimizar el rendimiento del bucle, pasando solo una referencia.

Foreach siempre tiene la misma forma, a la derecha como segundo parámetro esta una colección de elementos de un determinado tipo, y a la izquierda como primer parámetro, ponemos una variable del tipo de los elementos, que es la variable que va a ir recogiendo los valores recorridos de la colección, desde el primero hasta el último.

Bien, en el próximo tema vamos a ver las colecciones en Qt. Es conveniente que pruebes el código que hemos ido viendo en este tema, incluyéndolo en el esqueleto de una aplicación de consola usando qDebug().

TEMA 10

COLECCIONES EN Qt

Las colecciones en Qt están fundamentadas sobre las mismas de STL, pero llevadas a una implementación multiplataforma. Vamos a ir viéndolas una a una.

QList<T>

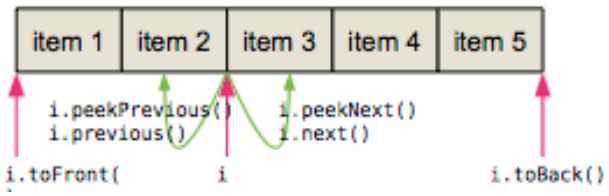
Es una lista, como las de STL, y la forma de rellenarla es igual. Vamos a ver unos ejemplos sencillos.

```
QList<int> pares;
pares << 2 << 4 << 6;
pares.prepend(0); // 0,2,4,6
pares.append(10); // 0,2,4,6,10
pares.insert(4,8); // 0,2,4,6,8,10
pares.removeFirst(); // 2,4,6,8,10
pares.removeLast(); // 2,4,6,8
pares.removeAt(2); // 2,4,8
pares.removeAll(); // nada
```

Con takeAt(), takeFirst(), takeLast(), nos devuelve el elemento en sí y además lo borra de la lista, de manera que si tenemos una lista de objetos, pues nos devolvería la referencia a dicho objeto.

Podemos acceder a sus elementos mediante el operador [], y la función value().

Para iterar una lista usamos **QListIterator<T>**. Que tiene funciones como hasNext() para asegurarse que no hemos llegado al final.




```
QListIterator<int> i(pares);
while (i.hasNext()){
    qDebug("Numero par: %d",i.next());
}
```

También podemos usar ***foreach*** para iterar toda la lista completa:

```
foreach(const int &valor, pares){
        qDebug("Numero par: %d",valor);
    }
```

Otras alternativas similares a QList, son QLinkedList que implementa una lista vinculada, y QVector que es una colección vector como las de STL. Los métodos usados son los mismos que QList. Dependiendo del uso que les vayamos a hacer, usaremos una lista, una lista vinculada o un vector.

Pilas con QStack

```
QStack<int> pila;
pila.push(1);
pila.push(2);
pila.push(3);
qDebug("Top: %d",pila.top()); // 3
qDebug("Top: %d",pila.pop()); // 3
qDebug("Top: %d",pila.pop()); // 2
qDebug("Top: %d",pila.pop()); // 1
pila.isEmpty(); // true
```

Colas con QQueue

```
QQueue<int> cola;
cola.enqueue(1);
cola.enqueue (2);
cola.enqueue (3);
qDebug("Top: %d",cola.head());    // 1
qDebug("Top: %d",cola.dequeue()); // 1
qDebug("Top: %d",cola.dequeue()); // 2
qDebug("Top: %d",cola.dequeue()); // 3
pila.isEmpty(); // true
```

Conjuntos con QSet

Los conjuntos o sets, también podemos asimilarlos a los Set de STL. Vamos a ver un poco de código para asimilarlos.

```
QSet<int> conjunto;
conjunto << 1 << 3 << 7 << 9;
conjunto.contains(8); // false
conjunto.contains(3); // true
conjunto.remove(8); // false y no borra nada
conjunto.remove(3); // true y borra el 3
```

Podemos también convertir una lista en un conjunto y viceversa:

```
QList<int> pares;
pares << 2 << 2 << 4 << 4 << 8; // 2,2,4,4,8
QSet<int> conjunto = pares.toSet(); // 2,4,8
QList<int> pares_no_repes = conjunto.toList();
```

Si hubiéramos querido conservar los elementos repetidos en el conjunto, hubiéramos tenido que usar un QMultiSet.

QMap<keyT,T> y QHash<keyT,T>

Se trata de pares key => value, como en las STL (o las matrices de PHP), donde una key se corresponde con un único value. Para introducir elementos se hace de la siguiente manera:

```
QMap<QString,int> edades;
edades["Antonio"] = 42;
edades["Maria"] = 42;
edades["Pablo"] = 8;
```

La función key() recoge el valor del primer elemento, keys() devuelve una QList<keyT> con todas las keys, y value(key) devuelve el valor asociado con un key en particular. Veamos ejemplos:

```
foreach(const QString &key, edades.keys()){
    qDebug("Nombre: %s",qPrintable(key));
}
```

Con la función contains(key), podemos ver si una determinada key esta en la colección.

```
if(edades.contains("Pablo")){
    qDebug("Pablo tiene %d", edades.value("Pablo"));
}
```

QHash es lo mismo que QMap, solo que la key se pasa por una función hash, para convertirla en un valor uint (unsigned int), la cual permite mayor rendimiento a la colección. Dicha función hash, ha de escogerse de manera que no se repita ningún valor uint que dificulte su rendimiento. Para ello el tipo de key, debe de proveer de 2 métodos: uint qHash (tipoKey), y bool operator== (key1, key2). Si así se hace de manera óptima, QHash<keyT,T> dará mejor rendimiento en todas las operaciones que QMap<keyT,T>. Veamos un ejemplo de implementación de estas funciones, como miembros de una clase hipotética Persona, que tiene dos miembros básicos llamados edad() y nombre().

```
uint Persona::qHash (const Persona &p)
{
    return p.edad() + qHash(p.nombre());
}
bool Persona::operator== (const Persona &p1, const Persona &p2)
{
    return (p1.edad() == p2.nombre()) && (p1.nombre() == p2.nombre());
}
```

Con esto ya podríamos crear un QHash<Person,int>, que trabaje de forma óptima.

Para finalizar, existen las versiones ***QMultiMap*** y ***QMultiHash***, para cuando una misma key, puede tener asociados varios valores a la vez. En estos casos, no se puede usar el operador [] para introducir nuevos datos, y se utiliza insert(). Si se quiere recoger el valor key una sola vez, se usa la función uniqueKeys(). Veamos un ejemplo para ilustrarlo.

```
QMultiMap<QString,QString> lang;
lang.insert("Antonio", "C/C++");
lang.insert("Antonio", "Java");
lang.insert("Antonio", "PHP");
lang.insert("Antonio", "Erlang");

foreach(const QString &nombre, lang.uniqueKeys()){
    QStringList program_lang = lang.values(nombre);
    // {C/C++, Java, PHP, Erlang} solo pasa una vez el bucle
}
```

TEMA 11

TIPOS Qt

Como en el tema anterior hemos visto, debido a que el propósito de Qt es poder ser multiplataforma sin tener que modificar el código, otra cosa que hubo que modificar son los tipos como los enteros, que dependen del sistema operativo, y de la CPU.

Así los tipos quedan fijados de esta manera en QtGlobal:

qint8 ------> 8 bits entero

qint16 ------> 16 bits entero

qint32 ------> 32 bits entero

qint64 ------> 64 bits entero

quint8 ------> 8 bits entero sin signo

quint16 ------> 16 bits entero sin signo

quint32 ------> 32 bits entero sin signo

quint64 ------> 64 bits entero sin signo

qreal ------> double

QVariant

Es un tipo que puede guardar un tipo cualquiera. Si necesitamos ser capaces de devolver cualquier tipo, a través de un interface QVariant nos puede ayudar, y finamente podemos convertirlo a su tipo final mediante las funciones del tipo toTipo() (ej. toInt(), toFloat()). Veamos un ejemplo.

```
QVariant var(5);
int i;
i=var.toInt(); // 5
```

Casting dinámico

Para hacer casting dinámico entre clases, pero sin requerir el uso del RTTI del compilador, en Qt existe el operador en QtGlobal llamado ***qobject_casting***.

Si no tenemos claro, si el casting se va a resolver bien, entonces usaremos este tipo de casting, ya que de no resolverse bien, devolverá un puntero nulo.

```
QObject *obj = new QTimer;          // QTimer hereda de QObject

QTimer *timer = qobject_cast<QTimer *>(obj);
// timer == (QObject *)obj

QAbstractButton *button = qobject_cast<QAbstractButton *>(obj);
// button == 0
```

TEMA 12

SISTEMAS DE FICHEROS

Otro de los sitios donde está muy justificado el uso de un sistema independiente de la plataforma es en los accesos a sistemas de ficheros, ya que en cada operativo, esto es diferente. De esta manera surge una serie de clases en Qt que nos ayudan a abstraernos de dichas particularidades.

QDir

Esta clase nos provee del manejo del sistema de ficheros de manera transparente. Los directorios más típicos están definidos como métodos estáticos.

```
QDir dir = QDir::current(); // directorio actual
QDir dir = QDir::home(); // /home
QDir dir = QDir::temp(); // directorio temporal
QDir dir = QDir::root();  // directorio raiz
QDir dir = QDir(QApplication::applicationDirPath());
// directorio de la aplicacion
```

QFileInfo es una clase que puede contener toda la información referente a un fichero (nombre, path, permisos, etc). QFileInfoList es una QList<QFileInfo>. Una función muy imporante de QDir es entryInfoList() que devuelve toda lista de un directorio (es como un dir o un ls). Veamos un ejemplo de cómo listar un directorio.

```
QFileInfoList infos = QDir::root().entryInfoList();
foreach(const QFileInfo &info, infos){
    qDebug("%s",qPrintable(info.fileName()));
}
```

Se podría hacer un filtrado de dicho listado usando los siguientes filtros, puestos y combinados con OR (|) como primer parámetro de entryInfoList(filter,sort):

QDir::Dirs	directorios
QDir:: Files	ficheros
QDir::NoSymLinks	no vínculos
QDir::Readable	con premiso de lectura
QDir::Writeable	con permiso de escritura
QDir::Executable	con permiso de ejecución
QDir::Hidden	ficheros ocultos
QDir::System	ficheros de sistema

El orden de listado se puede enviar como segundo parámetro y estos son los filtros de ordenación:

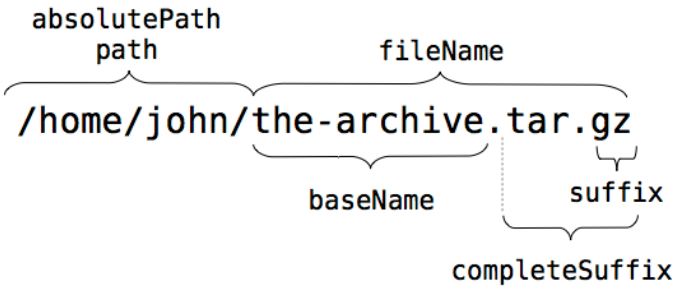
QDir::Name	por nombre
QDir::Type	por tipo (extensión)
QDir::Size	por tamaño

QDir::Time	por fecha de creación
QDir::DirsFirst	directorios primero
QDir::DirsLast	directorios lo último
QDir::Reversed	en orden inverso

EntryInfoList también tiene un constructor donde delante de estos parámetros vistos antes, se le puede añadir un parámetro que iría antes de que estos con una QStringList con filtros wildcard (del tipo *.h, *.cpp, etc). Veamos un ejemplo:

```
QFileInfoList infos = QDir::root().entryInfoList(QStringList << "*.h" <<
"*.cpp",QDir::Files,QDir::Name);
foreach(const QFileInfo &info, infos){
     qDebug("%s",qPrintable(info.fileName()));
}
```

Las distintas partes del path de un fichero están gráficamente relatadas en el siguiente gráfico.



TEMA 13

ESCRITURA Y LECTURA DE FICHEROS

Para la lectura y escritura en ficheros, tenemos que al menos implicar 3 clases diferentes, tanto en el caso de que sea un fichero binario, como si es de texto. Bien vamos a ver cada una de esas clases, y la manera en que se relacionan en el proceso de lectura y escritura.

La clase ***QFile*** que deriva de ***QIODevice***, es la que se usa para la operación de apertura, escritura, lectura y cierre del archivo. Los flags que representan los diferentes modos de acceso al fichero están definidos como un enum en QIODevice, y son los siguientes:

```
QIODevice::ReadOnly          abierto para lectura
QIODevice::WriteOnly         abierto para escritura
QIODevice::ReadWrite         abierto para lectura y escritura
QIODevice::Append            abierto para añadir al final del fichero
QIODevice::Text              cuando lee los enter los pasa a “\n” y cuando escribe,
                             los enter los pasa al sistema local
```

De la clase QIODevice vamos a usar meramente los modos de acceso a los ficheros, y de la clase QFile, vamos a usar los métodos open() para la apertura del fichero, close() para el cierre, write() para la escritura en el mismo, y read() y similares para la lectura. También vamos a usar métodos como atEnd() para reconocer el EOF (final del fichero), o seek() para situarnos en una posición concreta del fichero. Veamos un ejemplo de lectura:

```
QFile f("fichero.txt");

if(!f.open(QIODevice::ReadOnly))
    qFatal("No puedo abrir el fichero para lectura.");

while(!f.atEnd()){
    QByteArray data = f.read(1024);
    // Aqui puedes usar los datos leidos
}

f.close();
```

Esto mismo puede ser hecho de una manera más intuitiva usando streams. Las clases interface, que se encargan de crear un stream con los datos que proceden o van al fichero en nuestro caso, aunque también se puede usar con cualquier otro dispositivo. Estas clases son ***QTextStream*** para flujos de texto, y ***QDataStream*** para flujos de datos binarios. De estas clases vamos a usar solamente el constructor, indicando como parámetro el dispositivo de entrada/salida.

Veamos un ejemplo de lectura de fichero de texto usando un stream:

```
QFile f("fichero.txt");
QTextStream in(&f);

if(!f.open(QIODevice::ReadOnly)
    qFatal("No puedo abrir el fichero para lectura.");
while(!in.atEnd()){
    qDebug("%s",qPrintable(in.readLine()));
}
f.close();
```

Veamos ahora un ejemplo de escritura de fichero de texto:

```
QFile f("fichero.txt");
QTextStream out(&f);

if(!f.open(QIODevice::WriteOnly | QIODevice::Append)
    qFatal("No puedo abrir el fichero para escritura.");
out << "Esto es un fichero de texto" << str << endl;
f.close();
```

En el caso de leer o escribir datos, debemos decir al compilador de Qt la versión que estamos usando del mismo, ya que es posible que en diferentes versiones, la implementación del datastream sea diferente. Veamos un ejemplo de escritura.

```
QFile f("fichero.bin");
QDataStream out(&f);

out.setVersion(QDataStream::Qt_4_6);
if(!f.open(QIODevice::WriteOnly)
    qFatal("No puedo abrir el fichero para escritura");
out << 0xFFFC;
f.close();
```

Streams y Tipos

Se puede serializar a un fichero una clase cualquier creada por nosotros, simplemente si en la misma implementamos los operadores << y >>. Vamos a ver un ejemplo de nuevo con la clase Persona y sus miembros edad() y nombre().

```
QDataStream Persona::&operator<< (QDataStream &out, const Persona &p)
{
    out << p.nombre();
    out << p.edad();
    return out;
}

QDataStream Persona::&operator>> (QDataStream &in, const Persona &p)
{
    QString nombre;
    int edad;
    in >> nombre;
    in >> edad;
    p = Persona(nombre, edad);
    return in;
}
```

TEMA 14

WIDGETS Y LAYOUTS

Los primeros temas, del 1 al 13, contenían los principios básicos que diferencian a la programación C++/STL de la programación C++/Qt. Es fundamental que tenga todos esos conceptos claros, y que haya hecho todos los ejemplos, y ejercicios posibles para afianzarlos bien. Si esto no es así, es el momento de que se pare, y vuelva sobre los 13 temas que acabamos de pasar.

Desde el tema actual, hasta el tema 20 incluido, vamos a ver los conceptos básicos y todas las herramientas necesarias para desarrollar un interfaz gráfico completo. Es fundamental por tanto que trate este bloque de temas como un todo, y que se ejercite en ellos antes de seguir adelante. Desde el tema 21 en adelante, ya se tratan particularidades y ampliaciones, que con una base sólida en los 2 bloques anteriores (1 al 13 y 14 al 20), podrá avanzar por ellos de manera segura y fiable.

Un interfaz gráfico muestra al usuario ventanas con elementos gráficos con los que puede interactuar, seleccionando opciones, introduciendo texto, pulsando en botones, etc. Toda aplicación tiene una ventana principal, que es la que se abre al principio y se mantiene hasta el final, cuando el usuario decide cerrar la aplicación. A parte, pueden haber ventanas de Diálogo, que se superpongan a la principal en el momento en que piden alguna acción por parte del usuario. Cualquiera de estas ventanas, esta compuesta a su vez por unos elementos gráficos básicos que dividiremos en 2 tipos: los widgets y los layouts.

Los widgets son elementos gráficos interactivos o no, que se dibujan dentro de una ventana ocupando un área rectangular, como un botón, una etiqueta, un cuadro de selección, etc. Un Layout es un componente que se usa para disponer espacialmente los widgets dentro de la ventana, es decir, para organizar su posición en ella. Layouts y widgets negocian el espacio a ocupar. En Qt se usan unos widgets llamados spacers (vertical y horizontal) que actúan como muelles, empujando los widgets hacia una zona de la ventana, con todo esto se consigue un interfaz elástico, que se adapta a las dimensiones que se quiera dar a la ventana que los contiene, y es muy eficaz en el caso de querer internacionalizar un interfaz, traduciéndolo a diferentes idiomas, donde los textos, miden diferente en cada idioma, y donde no queremos que se trunquen ni textos, ni gráficos.

Los widgets

En el Qt Designer nos podemos encontrar más de 45 widgets para usar, y cerca de 60 son derivados de la clase principal ***QWidget***. Los widgets están organizados en categorías, y una categoría importante son los widgets contenedores, que pueden contener en su interior otros widgets, de manera que sirven para organizarlos, y además pueden actuar en su interactividad, como por ejemplo, cuando metemos varios radio-buttons (QRadioButton) dentro de group-box (QGroupBox), conseguimos que al seleccionar uno de ellos, los demás se deseleccionen automáticamente, por lo que se puede decir que los widgets contenedores pueden afectar y modificar la interactividad de un widget contenido en él.

Otra propiedad importante de los widgets, es que son capaces de recibir eventos desde los dispositivos de entrada (por ejemplo el ratón). Cuando por causa de un evento, o por otra causa, el estado del widget es cambiado, emite una señal notificando tal cambio. Los widgets por tanto pueden conectarse con otros widgets y objetos, mediante el mecanismo de signal-slot que vimos en los QObjects, ya que QWidget deriva de aquél.

Los Layouts

Los objetos layout derivan de la clase QLayout. Hay 3 tipos diferentes de layouts: QHBoxLayout (que organiza el espacio que ocupan los widgets horizontalmente), QVBoxLayout (que organiza el espacio que ocupan los widgets verticamente) y QGridLayout (que organiza el espacio que ocupan los widgets, sobre un grid, dividiendo el espacio en filas y columnas, y situando a cada widget en la celda que le corresponda). Los spacers, rellenan los espacios vacíos, y de esta manera, layouts y widgets, negocian su tamaño y el espacio para crear un interfaz elástico adaptable a todas las situaciones.

Un Layout, puede contener en su interior otros Layouts y otros Widgets. Así QLayout posee unas funciones miembro especiales para añadir estos contenidos a su interior, y cuando estas funciones son usadas, inmediatamente dicho contenido pasa a ser hijo de la clase contenedora. Este mecanismo facilita entonces la transmisión de eventos y propiedades a lo largo de la jerarquía familiar, dándole más consistencia al interfaz gráfico creado por objetos continentes y objetos contenidos. También facilita el mecanismo de liberación de memoria que vimos en el tema 6, evitando fugas de memoria. Veamos las 3 funciones básicas para añadir elementos a un Layout:

void QBoxLayout::addStretch (int *stretch*= 0)

Añade un spacer que ocupará por defecto como mínimo 0 pixeles, hasta el límite de su contenedor, empujando al widget en esa dirección (la del contenedor). Sólo BoxLayouts verticales y horizontales (no vale para GridLayouts).

void QLayout::addWidget (QWidget **w*)

Añade un widget dentro del Layout del tipo que sea éste. Desde ese momento dicho widget pasa a ser hijo del Layout que lo contiene.

void QLayout::addLayout (QLayout **layout*, int *stretch* = 0)

Añade un Layout dentro de otro (cajas pequeñas dentro de cajas más grandes). Desde ese momento dicho layout pasará a ser hijo de su contenedor, y por herencia a su vez de las cosas que aquél contenga.

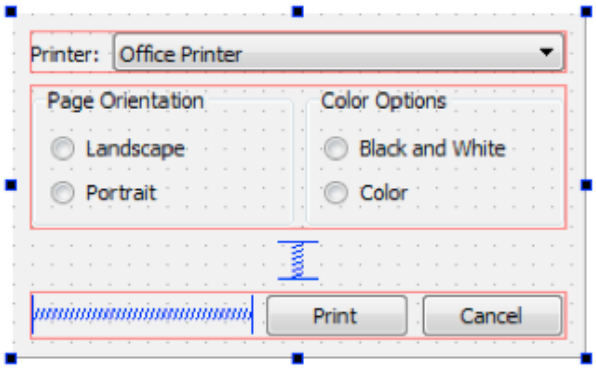
De hecho, en toda ventana gráfica, se puede dibujar un árbol jerárquico que recree las relaciones parentales de todos sus objetos. Por lo tanto, todo el contenido de una ventana debe estar sometido jerárquicamente al objeto que representa la ventana completa. Así cuando dicha ventana sea borrada de la memoria, con ella se borrarán todos sus elementos, evitando así problemas de memory leaks. Es muy habitual por tanto que los elementos gráficos de una ventana sean creados mediante new dentro del constructor del objeto que hará de ventana.

Puesto que hay muchos widgets, y sus propiedades son muchas, y sus métodos son muchos, no es propósito de este libro el verlos todos. Por tanto, es conveniente que conforme los vayas necesitando, los vayas revisando en el Qt Assistant, y aprendiendo a usar esta documentación como guía a la programación. El autorrellenado del Qt Creator, facilita también la labor de encontrar fácilmente los elementos de la programación que requerimos.

Vamos por tanto a continuación a plantearnos la construcción de un cuadro de diálogo, primero en código puro, y luego en el Qt Designer. De esta manera dejaremos concretada la enseñanza que pretendíamos para este tema, que es el diseño de ventanas usando Layouts y Widgets.

Construcción de una ventana de Diálogo con código

Vamos a construir una ventana de diálogo que tenga el aspecto de la foto que mostramos a continuación.



Podemos observar que primeramente hay 3 cajas horizontales (rectángulos rojos), que son contenedores Layout que en su interior tienen widgets o spacers (stretch). Vamos a suponer que la construcción de todo ello es hecho en el constructor de la ventana de diálogo, por lo que los contendores superiores, que son 3, tomarán como padre a la ventana y por tanto usaremos el puntero this como argumento de los constructores de los mismos.

```
QHBoxLayout *topLayout = new QHBoxLayout(this);
QHBoxLayout *groupLayout = new QHBoxLayout(this);
QHBoxLayout *buttonsLayout = new QHBoxLayout(this);
```

Empecemos por el topLayout donde hay un QLabel con el texto Printer y un QComboBox, al que luego queremos añadirle ítems de donde seleccionar opciones. Por lo que añadimos debajo de la primera línea del código de antes:

```
QComboBox *c;
QHBoxLayout *topLayout = new QHBoxLayout(this);
topLayout->addWidget(new QLabel("Printer:"));
topLayout->addWidget(c=new QComboBox());
```

Date cuenta de que la Label sólo la hemos instanciado, pero no la hemos asignado a ninguna variable, ya que no requerimos hacerle nada en el código más adelante, no requerimos acceder a ninguna variable que la maneje, mientras que la ComboBox sí la hemos asignado a la variable c, para más adelante poder añadirle ítems de selección mediante dicha variable.

Ahora vamos a por el buttonsLayout, donde tenemos que colocar un spacer primero, y luego los 2 botones.

```
QHBoxLayout *buttonsLayout = new QHBoxLayout(this);
buttonsLayout->addStretch();
buttonsLayout->addWidget(new QPushButton("Print"));
buttonsLayout->addWidget(new QPushButton("Cancel"));
```

Vamos ahora a por la caja central que es la más compleja de este ejemplo. Primeramente vamos a añadirle a groupLayout los 2 GroupBoxes.

```
QHBoxLayout *groupLayout = new QHBoxLayout(this);
QGroupBox *orientationGroup = new QGroupBox();
groupLayout->addWidget(orientationGroup);
QGroupBox *colorGroup = new QGroupBox();
groupLayout->addWidget(colorGroup);
```

Fíjate que aquí hemos instanciado primero los GroupBoxes y luego los hemos añadido al Layout. Bien, ahora vamos a usar un Layout vertical dentro de cada GrupoBox para organizar el espacio de los widgets que van a ir dentro.

```
QGroupBox *orientationGroup = new QGroupBox();
QVBoxLayout *orientationLayout = new QVBoxLayout(orientationGroup); // nueva
groupLayout->addWidget(orientationGroup);
QGroupBox *colorGroup = new QGroupBox();
QVBoxLayout *colorLayout = new QVBoxLayout(colorGroup); // nueva
groupLayout->addWidget(colorGroup);
```

Ahora vamos a añadirle a cada VertialLayout los widgets que le corresponden. Por lo que el código completo referente al Layout del medio se queda así.

```
QHBoxLayout *groupLayout = new QHBoxLayout(this);
QGroupBox *orientationGroup = new QGroupBox();
QVBoxLayout *orientationLayout = new QVBoxLayout(orientationGroup);
orientationLayout->addWidget(new QRadioButton("Landscape"));
orientationLayout->addWidget(new QRadioButton("Portrait"));
groupLayout->addWidget(orientationGroup);
QGroupBox *colorGroup = new QGroupBox();
QVBoxLayout *colorLayout = new QVBoxLayout(colorGroup);
colorLayout->addWidget(new QRadioButton("Black and White"));
colorLayout->addWidget(new QRadioButton("Color"));
groupLayout->addWidget(colorGroup);
```

Si te has dado cuenta, no hemos puesto aún el spacer vertical que hay entre el Layout central y el inferior (el de los botones). Para hacer eso usando la función addStretch, tenemos que referirnos a una VerticalLayout, por lo que vamos a tener que contenerlo todo dentro de un Layout más grande. Como este QVBoxLayout, contenedor de todo, sería el padre de todo, entonces tendríamos que referir los 3 Layouts iniciales a éste, y éste pasaría a ser el más cercano en jerarquía a la ventana de Diálogo. Y así quedaría todo:

```
QStringList options;
options << "Office Printer" << "HP LaserJet" << "Canon OfficeJet";
QComboBox *c;

QVBoxLayout *outerLayout = new QVBoxLayout(this);
        // Layout superior
        QHBoxLayout *topLayout = new QHBoxLayout();
        topLayout->addWidget(new QLabel("Printer:"));
        topLayout->addWidget(c=new QComboBox());
        // Fin Layout superior
outerLayout->addLayout(topLayout);
        // Layout Central
        QHBoxLayout *groupLayout = new QHBoxLayout();
         QGroupBox *orientationGroup = new QGroupBox("Page Orientation");
        QVBoxLayout *orientationLayout = new QVBoxLayout(orientationGroup);
        orientationLayout->addWidget(new QRadioButton("Landscape"));
        orientationLayout->addWidget(new QRadioButton("Portrait"));
        groupLayout->addWidget(orientationGroup);
        QGroupBox *colorGroup = new QGroupBox(("Color options");
        QVBoxLayout *colorLayout = new QVBoxLayout(colorGroup);
        colorLayout->addWidget(new QRadioButton("Black and White"));
        colorLayout->addWidget(new QRadioButton("Color"));
        groupLayout->addWidget(colorGroup);
        // Fin Layout Central
outerLayout->addLayout(groupLayout);
outerLayout->addStretch();
        // Layout Inferior
        QHBoxLayout *buttonsLayout = new QHBoxLayout();
        buttonsLayout->addStretch();
        buttonsLayout->addWidget(new QPushButton("Print"));
        buttonsLayout->addWidget(new QPushButton("Cancel"));
        // Fin Layout Inferior
outerLayout->addLayout(buttonsLayout);


c->addItems(options);
```

Hemos sangrado los bloques que ya teníamos construidos para dejar claro el código que hemos añadido, y el sentido del mismo con respecto a los bloques que contiene, y el spacer vertical. También nos hemos permitido el lujo de añadir los ítems del ComboBox que harán de opciones, mediante una StringList.

Construcción de una ventana de Diálogo con Qt Designer

Algunos pensarán que soy un masoquista, diseñando la ventana a tecla, pero es fundamental aprender antes el código que hay detrás de todo, antes que abandonarse al placer del mero y puro diseño, que nos separa de la programación y por tanto del entendimiento de las bases fundamentales.

Vamos a crear un nuevo proyecto, donde elegiremos Qt C++ Project -> Qt GUI Application. Le podemos llamar al proyecto "dialog". La clase base se llamará "Dialog" y la clase base va a ser QWidget.

Una vez se han creado todos los ficheros, abrimos "dialog.ui" que se nos abrirá en el Qt Designer. Vamos a ir dibujando en él, el interface del diseño que estamos siguiendo. Iremos colocando primero en la parte superior, lo mejor que podamos, un Label y un ComboBox, arrastrándolo de las herramientas al formulario y soltándolo. Hacemos doble clic sobre el texto del Label y ponemos "Printer:". Luego en el bloque central vamos a situar primero los 2 GroupBox uno al lado del otro. Les cambiamos los textos, por los que les corresponden. Luego añadimos en la parte inferior los 2 PushButtons, y les cambiamos su texto.

Ahora vamos a poner los RadioButtons en sus respectivos GroupBoxes, y les ponemos el texto que les corresponden. Ahora situamos el spacer horizontal y el vertical, conforme al dibujo del diseño que estamos siguiendo.

Si te das cuenta, por mucho que lo intentes, no va a quedar todo lo alineado que queremos, por eso, es el momento de comenzar a crear los Layouts que organicen todo este espacio. Esto se hace siempre de dentro hacia fuera, empezaremos por agrupar los RadioButton, seleccionando los del grupo izquierdo (ambos manteniendo la tecla Ctrl pulsada, Cmd en los Mac), y los agrupamos pulsando el botón Lay Out Vertically. Aparece una caja roja alrededor de ellos que los alinea perfectamente entre sí. Vamos a hacer lo mismo con los de la derecha. Luego vamos a seleccionar el GroupBox izquierdo, y pulsamos Lay Out Vertically, que lo organiza con su contenido, los radiobuttons. Hacemos los mismo con el GroupBox de la derecha.

Luego seleccionamos rodeando con el ratón todo el bloque central y pulsamos en Lay Out Horizontally. Luego hacemos lo mismo con el grupo superior (label y combobox) y con el grupo inferior (spacer horizontal y los 2 botones).

Ahora nos quedan los 3 grupos y el spacer vertical. Pulsamos sobre una zona vacía del formulario, para seleccionarlo a él por completo, y terminamos pulsando en Lay Out Vertically. Ya se nos ha organizado todos los widgets con los layouts. De hecho puedes ver como el diseño se adapta flexiblemente al tamaño libre de la ventana si arrastras uno de los bordes del formulario, cambiando su tamaño, verás como todo se organiza en tamaño y posición perfectamente.

Con esto damos por terminado el tema referente a los elementos de una ventana (widgets y layouts), en el próximo tema vamos a pasar a ver las ventanas de diálogo, elemento indispensable de una aplicación gráfica.

Si por curiosidad quieres añadir los ítems del ComboBox, abre el código de "dialog.cpp" y déjalo de esta forma:

```
QStringList options;
options << "Office Printer" << "HP LaserJet" << "Canon OfficeJet";
ui->setupUi(this);
ui->comboBox->addItems(options);
```

Fíjate que hemos accedido desde el objeto ui que representa el interfaz diseñado en el fichero *.ui, a la combobox, por su nombre (objectName), y entonces ya podemos llamar a sus métodos libremente. Ya veremos esto con más detalle más adelante.

TEMA 15

VENTANAS DE DIALOGO

Un widget , que no tenga padre se le llama top-level widget. Un top-level puede usar el método show() para mostrarse en pantalla, y puede por tanto constituirse como ventana. Así hay 3 tipos de top-level Windows (ventanas):

QWidget.- Cualquier widget sin padre es constituido automáticamente como una ventana, una ventana plana, como la que hemos creado en el tema anterior, o como la que se generaba simplemente mostrando un QLabel en el ejemplo de HolaMundo.

QDialog.- Es un widget especial, que conforma una ventana que además devuelve un valor siempre tras cerrarse (OK, Cancel, etc…)

QMainWindow.- Es otro widget especial, que conforma la ventana más completa y por tanto la principal de una aplicación gráfica, con menús, herramientas, status bar, dock, etc.

Puesto que ya hemos visto a los widgets funcionando como ventanas, y las MainWindows las trataremos a fondo en el tema 19, en este tema vamos a tratar de cerca las ventanas de diálogo, descendientes de QDialog.

Toda ventana de Diálogo lleva una serie de widgets que la componen, pero lo fundamental es que lleva botones para aceptar y/o rechazar una determinada acción, y que esa decisión es devuelta al programa mediante un valor.

En este capítulo vamos a ver el código que conforma a una ventana Diálogo para su uso dentro de una aplicación mayor, pero no vamos a desarrollar un ejemplo completo, ya que nos faltan elementos que no debemos aún tocar aquí y que una vez conocidos podremos unirlos todos en el tema 20.

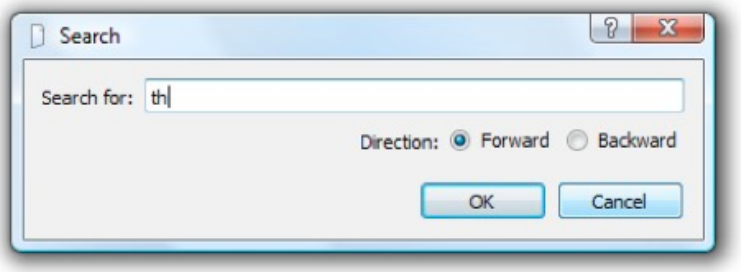
Una ventana dependiendo, de la interactividad que bloquea, puede ser:

No Modal.- Si no bloquea ninguna ventana que le acompañe, y permite estando ella abierta, interactuar con otras ventanas también abiertas.

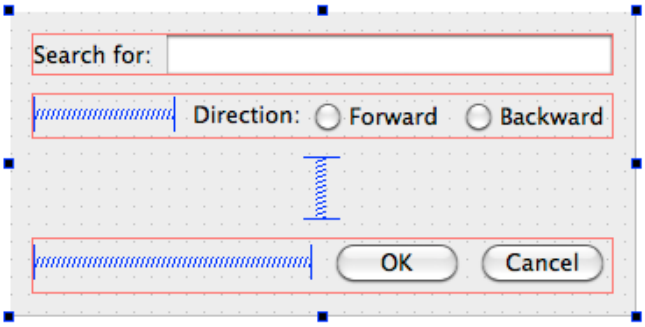
WindowModal.- Sólo bloquea la ventana padre de la misma, no pudiendo interacturar con ella, hasta que ésta es cerrada, aunque permite interactuar con el resto.

ApplicationModal.- Bloquea a todas las ventanas de la aplicación, sólo pudiendo interactuar con ella mientras está abierta.

Toda ventana de Diálogo, hereda de QDialog, y debe llevar botones para seleccionar opciones y pushbuttons para aceptar o rechazar la acción. Vamos a crear una ventana de diálogo con el Qt Designer, tal como hemos aprendido en el tema pasado, con esta forma:



Así que esta vez el proyecto lo basaremos en la clase QDialog y le llamaremos SearchDialog. Una vez abierto el fichero *.ui en el Qt Designer, selecciona el formulario, y a la derecha, cambia la propiedad WindowTitle en QWidget, por "Search" para que salga el título que vemos en el dibujo. Construye algo similar a esto:



Y pulsando en el cuadro azul inferior derecho de la selección del formulario, haz el tamaño lo mínimo que se pueda de alto, pero hazlo al menos el doble de ancho que de alto, de manera que se parezca al dibujo original. Cambia el nombre del QLineEdit por "searchText" y el de los radiobuttons por "directionForward" y "directionBackward" respectivamente.

Una vez hecho el diseño, vamos a ver el código que lo integrará en la funcionalidad de una aplicación más compleja.

En el archivo searchdialog.h en el constructor pondremos un código como éste:

```
class SearchDialog : public QDialog
{
    Q_OBJECT
public:
    explicit SearchDialog(const QString &initialText, bool isBackward, QWidget *parent
= 0);
    ~SearchDialog();

    // los getters que recogerán las opciones
    bool isBackward() const;
    const QString &searchText() const;

private:
    Ui::SearchDialog *ui;
};
```

Hemos puesto argumentos para inicializar el texto de búsqueda, por ejemplo, y así que se conserve entre las búsquedas. También se inicializa de igual manera la opción de dirección de la búsqueda. Usaremos 2 getters para recoger el estado de las opciones. Vamos a ver la implementación de este código en searchdialog.cpp .

```
SearchDialog::SearchDialog(const QString &initialText, bool isBackward, QWidget
*parent) : QDialog(parent), ui(new Ui::SearchDialog)
{
    ui->setupUi(this);
    ui->searchText->setText(initialText);
    if(isBackward)
        ui->directionBackward->setChecked(true);
    else
        ui->directionForward->setChecked(true);
}

bool SearchDialog::isBackward() const
{
    return ui->directionBackward->isChecked();
}

const QString &SearchDialog::searchText() const
{
    return ui->searchText->text();
}
```

Ahora vamos a ver como se integraría el código de mostrar la ventana en el resto de la aplicación.

```
SearchDialog
dlg(settings.value("searchText").toString(),settings.value("searchBackward").toBool(),
this);

if(dlg.exec() == QDialog::Accepted){
        QString text = dlg.searchText();
        bool backwards = dlg.isBackward();
        ......................
}
```

Por defecto la ventana es modal, pero puede ser cambiado mediante el método setWindowModality(). Estábamos acostumbrados a mostrar la ventana con show() (no modal por defecto), pero ahora al hacerlo con exec() (modal por defecto), lo que se devolverá una vez cerrada, será un valor entero, que es un DialogCode.

Constant	Value
QDialog::Accepted	1
QDialog::Rejected	0

Ventanas de Diálogo Standard

Además de poder diseñar ventanas a nuestro gusto y necesidades, Qt incorpora una serie de ventanas que son muy típicas de las GUIs de todas las plataformas. En concreto son 6 tipos más los subtipos de cada una de ellas. Vamos a verlas desde la más sencilla a las más avanzada.

1.- QInputDialog.- Es una ventana de diálogo que provee automáticamente de un Label, un LineEdit o un ComboBox, y los botones OK y Cancel. Puede recoger: un Int, un Double, un texto o un Item de un combo box, y lo hace con las funciones correspondientes getInt(), getDouble(), getText() y getItem(). Todas estas funciones devuelven como último parámetro un bool que dice si se eligió el botón OK (true), o el Cancel (false). Vamos a ver el código ejemplo de las 4 posibilidades:

Ejemplo con getInt:

```
bool ok;
int i = QInputDialog::getInt(this,"Ejemplo con getInt","Porcentaje:", 25, 0, 100, 1,
&ok);
```

Ejemplo con getDouble:

```
double d = QInputDialog::getDouble(this," Ejemplo con getDouble",
"Cantidad:", 37.56, -10000, 10000, 2, &ok);
```

Ejemplo con getItem:

```
QStringList items;
items << "Primavera" << "Verano" << "Otoño" << "Invierno";

QString item = QInputDialog::getItem(this, " Ejemplo con getItem",
 "Estación:", items, 0, false, &ok);
if (ok && !item.isEmpty()) itemLabel->setText(item);
```

Ejemplo con getText:

```
QString text = QInputDialog::getText(this, "Ejemplo con getText",
 "Usuario:", QLineEdit::Normal, QDir::home().dirName(), &ok);
if (ok && !text.isEmpty()) textLabel->setText(text);
```

2.-QErrorMessage.- Es una ventana de error que provee de un TextLabel y un CheckBox.

```
errorMessageDialog = new QErrorMessage(this);
errorMessageDialog->showMessage("Esta ventana muestra un error. "
                "Si el checkbox se deja, volverá a mostrarse, "
                "si no se deja, ya no volverá a mostrarse. ");
```

Con showMessage() ponemos el texto del error, automáticamente aparecerá el checkbox diciendo si quiere que se muestre el error de nuevo, si decide que no, ya no volverá a salir, aunque sea llamada en el código la función showMessage() de nuevo.

3.-QMessageBox.- Es una ventana modal que pregunta al usuario una pregunta, y recibe una respuesta. Las hay de 4 tipos:

warning.-

```
QMessageBox msgBox(QMessageBox::Warning,"warning",MESSAGE, 0, this);

msgBox.addButton("Save &Again", QMessageBox::AcceptRole);
msgBox.addButton("&Continue", QMessageBox::RejectRole);
if (msgBox.exec() == QMessageBox::AcceptRole)
      // codigo de aceptacion
else
      // codigo de rechazo
```

question.-

```
QMessageBox::StandardButton reply;

reply = QMessageBox::question(this,"question",MESSAGE , QMessageBox::Yes |
QMessageBox::No | QMessageBox::Cancel);

if (reply == QMessageBox::Yes)
         // codigo de Yes
else if (reply == QMessageBox::No)
         // codigo de No
else
         // codigo de Cancel
```

information.-

```
QMessageBox::StandardButton reply;
reply = QMessageBox::information(this," information", MESSAGE);
if (reply == QMessageBox::Ok)
         // codigo de OK
else
         // codigo de Escape (tecla escape) no todos los sistemas lo admiten
```

critical.-

```
QMessageBox::StandardButton reply;

reply = QMessageBox::critical(this,"critical",MESSAGE, QMessageBox::Abort |
QMessageBox::Retry | QMessageBox::Ignore);

if (reply == QMessageBox::Abort)
         // codigo de Abort
else if (reply == QMessageBox::Retry)
         // codigo de Retry
else
         // codigo de Ignore
```

4.-QColorDialog.- Muestra el cuadro de selección de color, y devuelve el código del mismo.

```
QColor color = QColorDialog::getColor(Qt::green, this); // color inicial verde

if (color.isValid()) {
          // codigo de uso de color
}
```

5.- QFontDialog.- Para elegir la fuente.

```
bool ok;

QFont font = QFontDialog::getFont(&ok, this);

if (ok) {
        // codigo de uso de la fuente
}
```

6.-QFileDialog.- Para hacer operaciones de ficheros o directorios. Hay de 4 tipos:

getOpenFileName.-

```
QString selectedFilter;
QString fileName = QFileDialog::getOpenFileName(this,
                                "getOpenFileName",
                                defaultFileName,
                                "All Files (*);;Text Files (*.txt)",
                                &selectedFilter,
                                options);
if (!fileName.isEmpty()) // codigo de apertura del fichero
```

getSaveFileName.-

```
QString selectedFilter;
QString fileName = QFileDialog::getSaveFileName(this,
                                "getSaveFileName",
                                defaultFileName,
                                "All Files (*);;Text Files (*.txt)",
                                &selectedFilter,
                                options);
if (!fileName.isEmpty())// codigo de guardado del fichero
```

getOpenFileNames.-

```
QString selectedFilter;
QStringList files = QFileDialog::getOpenFileNames(this,
                                "getOpenFileNames",
                                openFilesPath,
                                "All Files (*);;Text Files (*.txt)",
                                &selectedFilter,
                                options);
if (files.count()) {
        openFilesPath = files[0];
        // codigo de apertura de ficheros
}
```

getExistingDirectory.-

```
QFileDialog::Options options = QFileDialog::DontResolveSymlinks |
                               QFileDialog::ShowDirsOnly;

QString directory = QFileDialog::getExistingDirectory(this,
                                "getExistingDirectory",
                                defaultDirName,
                                options);
if (!directory.isEmpty())// codigo de apertura de directorio
```

TEMA 16

MODELO, VISTA Y CONTROLADOR, INTRODUCCION

El primer lenguaje que por primera vez introdujo el paradigma MVC (Modelo, Vista, Controlador) fue Smalltalk en 1979 en el laboratorio de Xerox. Actualmente muchos lenguajes y frameworks lo han incorporado como una ventaja al desarrollo inestimable, y por supuesto Qt no iba a ser menos.

En la gran mayoría de aplicaciones, se accede a fuentes de datos, que pueden ser, un fichero, un sistema de archivos, una base de datos, un stream de datos, etc. Luego esos datos se muestran al usuario de diversas formas, y el usuario interactúa con ellos, modificándolos, leyéndolos o tratándolos de alguna manera y la fuente de los mismos es actualizada con dicha modificación o tratamiento. El paradigma MVC separa en 3 aspectos diferentes de esta actividad, por un lado esta el Modelo, que está en contacto directo con la fuente de datos, de manera que es quien recoge los datos de la misma y los actualiza. Esos datos son enviados por el modelo a la Vista (view), que se encarga de visualizarlos de una manera específica. Un mismo modelo puede alimentar varias vistas diferentes y ambas sincronizar su contenido a la vez. Sobre la vista el usuario interactúa, y la misma puede proveer de métodos para enviar modificaciones de los datos al Modelo (model), y este a la fuente de datos, o existen también objetos especiales, especializados en la interacción del usuario con la vista, que aquí no se llama Controlador sino Delegado (delegate).

La verdad es que se trata de conceptos muy abstractos, y creo que lo mejor es ponerlos a funcionar para ver de lo que estamos hablando. Suponga que quiere presentar en una ventana en forma de árbol el contenido completo del directorio de su ordenador, con ficheros y directorios. Sin duda sería algo difícil de hacer meramente con QTreeWidget y QDir (QFileInfo, etc), y si al final lo lograra, seguro que ocuparía bastantes líneas de código. Pues bien, el paradigma MVC viene a nuestro rescate, para hacer de esta tarea algo sencillo y con poco código, de hecho se podría hacer con solamente 4 líneas de código, si no fuera porque nos gusta cambiar unas determinadas propiedades de la vista, pero en cualquier caso no va a pasar de las 10 líneas. ¿Cómo es eso posible?.

Qt provee de una serie de vistas suficientes para lo que cualquiera suele programar, y también de los modelos para lo mismo, pero si esto no fuera así, sería mucho más sencillo crearlos para poder ponerlos a trabajar juntos, a hacer un programa específico con un código complejo y poco reutilizable. Cuando un programador se pone a escribir mucho código, es porque sabe que lo va a rentabilizar en el tiempo, de lo contrario, se está quieto.

Bien, pues para llevar a cabo la aplicación que hemos comentado, vamos a usar como Vista, la clase QTreeView, que nos provee de un magnífico gráfico en árbol muy avanzado y parametrizable, y como Modelo, vamos a usar la clase QDirModel, que se relaciona directamente con el sistema de archivos como su fuente de datos. Aunque parezca mentira, simplemente hay que combinar ambas clases, Modelo y Vista para que surja la chispa mágica, y eso se hace con el método *setModel(modelname)* , que proveen todas las vistas. De esta manera la vista es cargada con datos que consigue el modelo para ella. Vamos a por el código.

Esta vez vamos a crear un proyecto vacío (Empty Project), al que vamos a llamar “mvc”. Luego vamos a añadirle un fichero fuente “main.cpp” y vamos a empezar a escribir directamente en él.

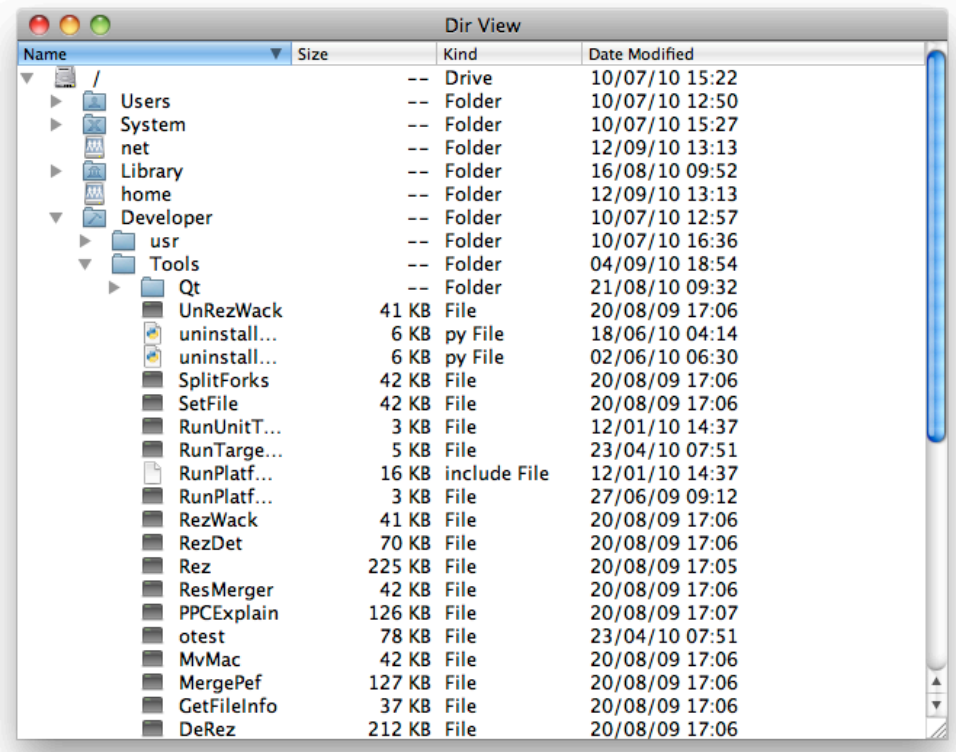
```
#include <QtGui>
#include <QApplication>
int main(int argc, char **argv)
{
    QApplication app(argc,argv);

    QDirModel model;
    QTreeView view;
    view.setModel(&model);
```

```
    // Vamos a cambiar el aspecto de la vista
    view.setAnimated(false);
    view.setIndentation(20);
    view.setSortingEnabled(true);
    view.setWindowTitle("Dir View");
    view.resize(640,480);
    // vamos a mostrar la vista
    view.show();

    return app.exec();
}
```

Eso es todo el código, vamos a ver cual es el resultado final de esas pocas líneas.



Efectivamente es impresionante el resultado obtenido con tan escueto código. Con esto queda justificado sobradamente el uso del paradigma MVC siempre que sea posible.

Ahora que he conseguido tu atención, es hora de que te presente las vistas y los modelos de que vas a disponer en Qt, que como ya apuntamos van a ser suficientes para la gran mayoría de aplicaciones que vas a acometer. Si esto no es así, en un curso más avanzado podrás aprender a crear tus propios modelos, o tus propias vistas, pero eso ya se sale del propósito de este primer libro.

Las Vistas

Las vistas que proporciona Qt son todas derivadas de la clase ***QAbstractItemView*** . Las vistas que de ella se derivan son tres tipos principalmente: QListView, QTableView y QTreeView.

QListView, proporciona una vista unidimensional, donde hay sólo una columna (column) y muchas filas (row). Dispone de una serie de métodos para cambiar la presentación de los ítems de la misma.

QTableView, es la más usada, sobre todo en la presentación de tablas de bases de datos. Tiene forma bidimensional con varias columnas y varias filas, además posee cabecera tanto para las columnas como para las filas, para poner sus títulos en ellas.

QTreeView, es usada sobre todo en acceso a directorios, como hemos visto en el ejemplo, y dispone la información en forma de árbol. Además también dispone de cabecera horizontal (header), para poner títulos a las propiedades de los ítems.

Los modelos

Los modelos que proporciona Qt para acceder a los diferentes tipos de fuentes de datos, derivan de ***QAbstramItemModel***, y entre los que proporciona están: QStringListModel, QStandardItemModel, QFileSystemModel y QSqlRelationalTableModel.

QStringListModel, es usado para guardar una lista de QStrings.

QStandardItemModel, se usa para el manejo de estructuras más complicadas en forma de árbol, que pueden guardar cualquier tipo de datos.

QFileSystemModel, provee de la información que obtiene del sistema de ficheros, como *QDirModel*, pero más avanzado.

QSqlRelationalTableModel, se usa para tratar datos provenientes de una base de datos relacional. De ella deriva *QSqlTableModel* que se usa para guardar los datos de una tabla concreta de una base de datos. De ella deriva *QSqlQueryModel* que guarda los datos de la respuesta a una query a una base de datos.

Observaciones finales

Como puedes imaginar, combinando una vista o varias vistas con un modelo, puedo obtener el interfaz gráfico que necesito para interactuar con los datos que quiero. El caso más típico es con una base de datos relacional, como MySQL u Oracle. Se trataría por tanto de combinar una QTableView con una QSqlTableModel o una QSqlQueryModel. Veremos casos concretos en los temas que correspondan a bases de datos.

TEMA 17

QLISTWIDGET Y QTABLEWIDGET

Derivadas de QListView y QTableView, tenemos 2 clases que representan vistas en forma de lista y tablas, respectivamente, con un modelo predefinido. Ambos son QWidgets proporcionados por la barra de herramientas del Qt Designer, y suelen ser usados para aplicaciones sencillas donde no se usan modelos combinados con ellas. Están formados por ítems de las clases QListWidgetItem y QTableWidgetItem, respectivamente.

QListWidget

Para añadir elementos a esta Lista hay 2 formas:

1.- Construyendo los ítems con esta clase como padre.

```
new QIstWidgetItem ("Texto elemento 1",listWidget);
```

2.- Construyendo el item sin padre y añadiéndolo a la lista más tarde.

```
QListWidgetViewItem *newItem = new QListWidgetViewItem;
newItem->setText("Texto Elemento 1");
listWidget->insertItem(row,newItem);
```

En la segunda opción también se puede usar el método *insetIntems(QStringList)*, y también *addItem(QString)*, que añade al final del anterior, y no es una posición exacta.

El modo de selección de los elementos en la lista se puede leer con la función *selectMode()*, y se puede establecer con la función *setSelectionMode(selmode)*. Los diferentes modos de selección son: SingleSelection (un solo elemento), MultiSelection (varios elementos a la vez), ContiguousSelections (varios elementos contiguos a la vez) y NoSelection (no se puede seleccionar nada).

La razón de usar QListWidgetItem en vez de QStrings en la inserción de elementos, es porque, éste permite no sólo insertar texto en las listas, si no también iconos. Así tambien hay dos Modos de vista: IconMode (drag & drop desabilitado por defecto) y ListMode (drand & drop habilitado por defecto). También se puede cambiar el tamaño del icono con *setIconSize(QSize)*, y el espaciado con *setSpacing(value)*.

Para borrar ítems, se puede usar *takeItem(row)*, y los Items pueden aparecer en la lista marcados con *QListViewItem::setCheckState(Qt::CheckState)*.

La única señal que usamos en esta clase es *currentItemChanged(current,previous)*, que se produce al cambiar el foco de item. La función *currentItem()*, nos devuelve el item actual, y *currentRow()*, nos devuelve el row (fila) del item actual.

QTableWidget

Se puede dimensionar en la creación en el constructor (*new QTableWidget (rows,cols,parent)*), y luego redimensionarla con *setRowCount(rows)* y *setColumnCount(cols)*.

Podemos añadir ítems con *setItem(row,col,newItem)*, donde newItem es un QTableWidgetItem. Se pueden borrar ítems con *takeItem(row,col)*. Podemos poner etiquetas en las cabeceras horizontales con *setHorizontalHeaderLabels(QStringList)* y en las cabeceras

verticales con *setVerticalHeaderLabels(QStringList)*. También ordenar por una columna usando *sortItems(col, Qt:SortOrder)*.

Con *rowCount*() podemos contar las filas, con *columnCount*() las columnas, y con *clear*() borrar todo el contenido de las celdas.

QTableWidgetItem, tiene funciones interesantes como *setIcon*(), *setFont*() y *setText*().

Las señales más usadas son: *cellClicked(int row,int col), cellChanged(int row,int col), itemClicked(QTableWidgetItem item), itemChanged(QTableWidgetItem item)*.

Veamos ahora un ejemplo de tabla con las cosas que hemos visto:

```
#include <QtGui>

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

    QTableWidget tabla(12,2);
    QStringList headers;

    headers << "Meses" << "Dias";
    tabla.setHorizontalHeaderLabels(headers);

    QStringList meses, dias;
    meses << "Enero" << "Febrero" << "Marzo" << "Abril" << "Mayo" << "Junio" <<
"Julio" << "Agosto" << "Septiembre" << "Octubre" << "Noviembre" << "Diciembre";
    dias << "31" << "28" << "31" << "30" << "31" << "30" << "31" << "31" << "30" <<
"31" << "30" << "31";

    for(int mes=0; mes < 12; mes++){
        tabla.setItem(mes,0,new QTableWidgetItem(meses[mes]));
    }

    for(int mes=0; mes < 12; mes++){
        tabla.setItem(mes,1,new QTableWidgetItem(dias[mes]));
    }

    tabla.resize(240,400);
    tabla.show();

    return app.exec();
}
```

TEMA 18

Qt CREATOR

Queremos dedicar un poco de tiempo al entorno de desarrollo integrado de Qt, su IDE llamado Qt Creator. En un principio Qt no poseía un IDE, todo el código tenía que ser escrito en un editor de texto avanzado, o integrado en otro IDE como Eclipse. Y aún se puede seguir haciendo así, pero desde la versión 4.0 del SDK, Trolltech incluyó Qt Creator como un entorno de desarrollo completo, desde donde se puede acceder al Qt Assistant, al Qt Designer, se puede compilar, debugear desde él. Pero una de las características más interesantes para el que pica código, es el autorrellenado, y las teclas de acceso rápido. Por ello, queremos detenernos un tiempo para ver como podemos sacar ventaja en nuestro desarrollo de este IDE avanzado.

El autorellenado

Cuando abrimos un fichero de código *.h , *.cpp, y nos situamos en una línea para empezar a teclear código, y queremos que el sistema nos ayude a rellenar, desde un fichero cabecera, una clase, una variable ya definida, sólo tenemos que pulsar la tecla Ctrl+espacio, para que aparezca toda una serie de alternativas factibles, de acuerdo a lo que ya hay escrito y declarado en el fichero. Tecleando las primeras letras de lo que queremos poner, nos irán apareciendo las opciones cada vez más específicas de manera que muchas veces no hace falta acordarse del nombre exacto de una clase, un fichero o una variable. Si entre las opciones que nos dan, no aparece una clase o variable específica, es que no es accesible desde esa parte del documento, por cualquier razón, que puede ser desde que no está incluido su fichero cabecera, a que dicha variable no es accesible en dicha zona.

También nos ayuda a acceder a los miembros de las clases, y nos da los argumentos de las funciones cuando escribimos tras ellas, los paréntesis.

Cuando nos situamos sobre un elemento del código, se nos recuadra este, y todos los mismos elementos dentro de la misma zona de ámbito. De esta forma podemos ver los ámbitos, y hacerle un seguimiento a una variable.

Si necesitamos ayuda del Qt Assistant sobre algún elemento del código, solo hay que situarse sobre él y pulsar F1, entonces se nos abrirá a la derecha del código la ventana de ayuda correspondiente.

Conforme se escribe, si hay algún error sintáctico, aparece subrayado en rojo, y si nos situamos sobre él, nos aparece un mensaje emergente, describiendo el error.

Atajos de teclado (keyboard shortcuts)

Ctrl + cursor derecho .- Te lleva al final de la línea actual.

Ctrl + cursor izquierdo .- Te lleva al principio de la línea actual.

Shift + cursores .- Selección de texto.

Ctrl + Up/Down.- Pasa páginas de código arriba y abajo.

Ctrl + u (Cmd + U en Mac).- Seleccionas bloques, pulsando varias veces aumentas el bloque de selección.

Ctrl + i (Cmd + i en Mac).- Hace que el código seleccionado tome la sangría requerida.

Ctrl + Shift + Up/Down (Cmd + Shift + Up/Down en Mac).- Permite mover líneas completas hacia arriba o hacia abajo. Funciona también con selecciones completas, moviendo el texto seleccionado.

Ctrl + / (Cmd + / en Mac).- Comenta y descomenta todo un bloque seleccionado.

Shift + Del .- Borra toda una línea completa.

F4 .- Pasa de fichero cabecera a fichero de código y viceversa.

F2 .- Pasa de la declaración a la implementación de una función o método.

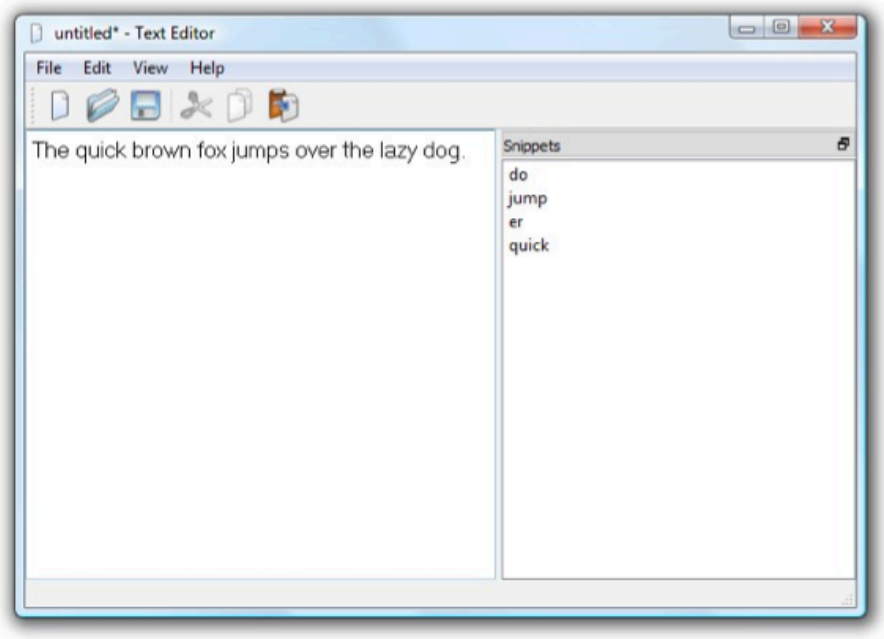
Ctrl + K .- Abre el localizador, para poder navegar por ayudas y documentos rápidamente.

Ctrl + Shift + R (Cmd + Shift + R en Mac).- Permite cambiar el nombre de una variable a la vez en todas las partes donde tiene ámbito.

TEMA 19

MAIN WINDOWS

Este es el tema más importante de todos los referentes al desarrollo gráfico, ya que en él se encuentra la base fundamental de todas las aplicaciones gráficas. La ventana principal (QMainWindow), es el componente fundamental, sobre la cual pueden aparecer toda una serie de widgets: QMenuBar (barra de menu), QToolBar (barras de herramientas), QStatusBar (barra de estado), QDockWidget (widgets acoplables), QMenu (menus en general, incluidos los contextuales) y otros que ya hemos visto o que veremos.



El interfaz principal de una aplicación, la Main Window (ventana principal), posee los elementos que vemos en la foto de arriba: un menú principal en la parte superior, una o varias barras de herramientas con iconos, justo debajo, que proporciona las opciones del menú más típicas en el uso cotidiano, mejorando la rapidez de uso de la aplicación, luego esta el widget principal (central widget), que en este caso está en la zona central izquierda, y es un QPlainTextEdit, a la derecha vemos una ventana adosable (dockable window), que se diferencia por el icono de la esquina superior derecha, que nos indica que pulsando sobre la barra superior y arrastrando, dicha ventana se separaría de la ventana principal, proporcionando otra ventana, finalmente en la parte inferior, se encuentra la barra de estado (status bar) donde se suele poner mensajes informativos al usuario al respecto de lo que esta aconteciendo en la aplicación en cada momento. También cada elemento de la ventana principal, puede tener un menú contextual asociado, que al pulsar el botón derecho del ratón nos permita llevar a cabo acciones propias de dicho elemento gráfico. A parte de estos elementos, una aplicación puede añadir ventanas de diálogo, derivadas de QDialog, que podrán aparecer como consecuencia de acciones del usuario, como pulsar sobre una opción del menú, una atajo del teclado, un menús contextual, o algún evento cualquiera (información, aviso, error, etc).

QAction

La principal misión de una QMainWindow, es centralizar todo el interfaz que comunica al usuario, con la actividad que la aplicación lleva a cabo, por ello todos los elementos que la componen están enfocados a proporcionar dicha interfaz. Sin embargo, una aplicación puede recibir órdenes por parte del usuario para llevar a cabo una misma acción desde diferentes

componentes, como por ejemplo, pegar una selección previa en el widget principal, podría hacerse seleccionado la opción Editar->Pegar en el menú principal, o pulsando sobre el icono correspondiente en la barra de herramientas, o seleccionado Pegar, del menú contextual del widget principal, o simplemente pulsando una combinación de teclas (shortcut) que lleve a cabo dicha acción. Debido a que una misma acción podría desencadenarla diferentes elementos, es por lo que todas las acciones se han centralizado entorno a una clase que no tiene representación gráfica física, llamada QAction. Cada opción del menú principal, de las toolbars o de los context menus, se corresponden con un objeto QAction (una acción), y una misma acción puede ser alcanzada desde varios de estos elementos, lo cual va a repercutir en como ha de definirse una acción. Vamos a ver la definición de una acción:

```
QAction *actionNew = new QAction("&New",this);
actionNew->setIcon(QIcon(":/images/new.png"));
actionNew->setShortcut("Ctrl+N");
actionNew->setStatusTip("Crear un nuevo documento");
actionNew->setToolTip("Crear un nuevo documento");
actionNew->setCheckable(false); // set
connect(actionNew, SIGNAL(triggered()), this, SLOT(newFile()));
```

Podemos ver que primero creamos un objeto acción que tendrá el texto asociado “&New” (donde el & va delante de la letra que queremos vaya subrayada indicando la combinación de teclas al usuario que nos llevará también a la misma acción). Luego mediante una serie de métodos de QAction, vamos definiendo la acción, que puede tener asociado un icono (ya veremos más adelante esto con más detalle), establecemos el shortcut de teclado, establecemos el texto que aparecerá en el status bar, al ejecutar esta acción, tambien el tooltip o texto emergente con la descripción de dicha acción si nos situamos sobre ella con el puntero del ratón, decidiremos si es una opción checkable, es decir que pueda llevar una marca de activada o desactivada, finalmente establecemos la conexión entre la señal triggered(), que se produce al pulsar sobre esa acción, y el slot que en este caso puede ser una función que nosotros mismos vamos a definir lo que va a hacer.

Menú principal

Cada opción principal del menú principal, valga la redundancia, es decir, las que se ven en la cabecera de la ventana sin pulsar sobre ellas, son objetos QMenuBar que son creados por la QMainWindow, cada vez que se llama a su función miembro ***menuBar()***, y con ello pasan a ser hechas hijas de esta ventana. Cada opción principal, por tanto requiere llamar la primera vez a dicha función para crearse, y luego va añadiendo acciones (subopciones de la misma) y separadores.

```
QMenu *fileMenu;
fileMenu = menuBar()->addMenu("&File");
fileMenu->addAction(actionNew);
fileMenu->addAction(openAction);
fileMenu->addSeparator();
fileMenu->addAction(exitAction);
```

QToolBar (barra de herramientas)

```
QToolBar *toolbar =addToolBar("&File");
toolbar->addAction(actionNew);
toolbar->addAction(openAction);
```

La función miembro de MainWindow, addToolBar, añade una barra con una seria de acciones, que llevan su propio icono.

Menu contextual (context menu)

Se asocia directamente el widget que lo va a mostrar, y sólo hay que añadirle las acciones que va a mostrar, y finalmente cambiar de dicho widget la propiedad ContextMenuPolicy para que aparezca al pulsar clic derecho.

```
QPlainTextEdit *editor;
editor->addAction(copyAction);
editor->addAction(pasteAction);
editor->setContextMenuPolicy(Qt::ActionsContextMenu);
```

Barra de estado (status bar)

La barra de estado normalmente contendrá widgets del tipo QLabel, para mostrar texto con avisos al usuario. Para ello, primeramente recogemos el puntero a la barra de estado con la la función de QMainWindow, ***statusBar()***. Luego añadimos los widgets, usando la función **addWidget()**, por lo que como se ve, una statusbar es un contenedor.

```
QLabel *label = new QLabel();
label->setAlignment(Qt::AlignHCenter);
label->setMinimumSize(label->sizeHint());
statusBar()->addWidget(label);
```

La ventana principal (QMainWindow)

La ventana principal al ser declarada e implementada, debe de contener todos los elementos que hemos descrito antes, toda una serie de slots entre los cuales estarán las funciones que ejecutan las acciones, como newFile(), que hemos visto en el ejemplo de QAction. También es interesante el redefinir la función virtual ***closeEvent(QCloseEvent *event)***, para por ejemplo interceptar la señal de cierre de la aplicación con alguna ventana de diálogo que pueda preguntar, si está seguro de cerrar. Otra cosa esencial que debe de tener en la implementación de su constructor, es la designación como widget central, esto se hace con su función miembro ***setCentralWidget(QWidget *)***.

Ficheros de recursos (*.qrc)

Los recursos que usa una aplicación, como pueden ser iconos, bitmaps y otros, van descritos en un fichero de recursos multiplataforma que lleva la extensión qrc. Qt soporta muchos tipos de imágenes e iconos tales como: BMP, GIF, JPG, PNG, PNM, XMB y XPM. El fichero qrc forma parte del proyecto antes de ser compilado e incluido en el ejecutable. Se trata de un fichero escrito en XML con el siguiente formato:

```
<!DOCTYPE RCC>
<RCC version="1.0">
        <qresource>
                <file>images/icon.png</file>
                •••
                <file>images/gotocell.png</file>
        </qresource>

</RCC>
```

En el fichero de proyecto (*.pro) se añade una linea que pone:

RESOURCES = recursos.qrc

En el código C++ ya se puede acceder a esos ficheros directamente mediante la URL ":/images/icon.png", por ejemplo:

```
action->setIcon(QIcon(":/images/new.png"));
```

Donde vemos que el fichero del icono se encuentra en la carpeta del proyecto en "images/new.png".

QSettings (ajustes)

Es muy útil y agradable para el usuario, que el aspecto que el interface tenía al cierre, vuelva a recuperarse al reabrir la aplicación al día siguiente. Por lo que, es muy usual que las aplicaciones modernas guarden una serie de parámetros que determinan todo esto que llamamos "settings", y lo recuperen la próxima vez que sea ejecutada.

Cada sistema operativo o plataforma, gestiona esto de una manera distinta, y lo guarda en sitios diferentes, por eso Qt, provee de una clase y métodos para tratar este tema de manera uniforme e independiente de la plataforma. Dicha clase es QSettings, y mediante sus métodos setValue() y value(), el setter y getter de la misma, gestiona el almacenamiento y recuperación de dichos ajustes (settings).

QSettings::QSettings (const QString &*organization*, const QString &*application*= QString(),QObject **parent* = 0)

Ejemplo de creación de settings:

```
QSettings settings("Qt Software, Inc", "TextEditor");
```

void QSettings::setValue (const QString &*key*, const QVariant &*value*)

Ejemplo de guardar ciertos settings:

```
settings.setValue("geometry", geometry());
settings.setValue("showGrid",showGridAction->isCheched());
```

QVariant QSettings::value (const QString &*key*, const QVariant &*defaultValue* = QVariant()) const

Ejemplo de recuperar ciertos settings:

```
QRect rect = settings.value("geometry",QRect(200,200,400,400)).toRect();
bool showGrid = settings.value("showGrid",true).toBool();
```

Splash screens

Son las ventanas que aparecen mientras se cargan los elementos de un programa, por diversos motivos como, indicar al usuario que se están cargando y darle información durante el tiempo de espera o por pura autopromoción. En cualquier caso es un elemento más de una aplicación que muy a menudo se usa.

Lo más habitual es que el código que muestra el splash screen, se encuentre en main() antes de que se llame a QApplication::exec().

La clase que la describe es ***QSplashScreen***, y usa funciones de QWidget como show() para mostrarse y setPixmap(QPixmap) para establecer el fondo a mostrar, finalmente el método showMessage() permite sobreimpresionar un mensaje sobre el fondo de dicha pantalla.

void QSplashScreen::setPixmap (const QPixmap &*pixmap*)

void QSplashScreen::showMessage (const QString &*message*, int *alignment* = Qt::AlignLeft, const QColor &*color* = Qt::black)

Veamos un ejemplo sencillo:

```
#include <QtGui/QApplication>
#include "mainwindow.h"

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

    QSplashScreen *splash = new QSplashScreen;
    splash->setPixmap(QPixmap(":/images/splash.png"));
    splash->show();
    splash->showMessage("Iniciand ventana principal ...", Qt::AlignRight |
Qt::AlignTop, Qt::white);

    MainWindow w;

    splash->showMessage("Cargando modulos ...", topRight, Qt::white);

    loadModules();

    splash->showMessage("Estableciendo conexiones ...", Qt::AlignRight | Qt::AlignTop,
Qt::white);

    establishConnections();

    w.show();

    splash->finish(&w);
    delete splash;

    return a.exec();
}
```

Date cuenta de que, se muestra el splash screen con su gráfico, luego se muestra el primer mensaje sobre el Pixmap en la zona superior derecha en texto blanco, luego se lleva a cabo la acción que indica el mensaje, luego el siguiente mensaje, luego la acción que indica el mismo, y así hasta que una vez todo esta cargado e iniciado, entonces muestra la ventana principal. La función finish(const QWidget&), lo que hace es esperar a que se cargue completamente el Widget, en este caso la MainWindow, antes de hacer un close() sobre él mismo.

En el próximo tema, vamos a desarrollar desde cero una aplicación completa con todo lo que hemos aprendido hasta el día de hoy, y lo haremos primero a código puro, sin diseño visual, para comprender y repasar todos los conceptos bien, y finalmente veremos como eso mismo se puede hacer de manera visual en el Qt Designer. Por lo tanto, es muy importante que todos los conceptos queden perfectamente aclarados con este ejemplo, ya que hasta aquí hemos completado el bloque principal y básico para construir aplicaciones en Qt, y sobre esta base vamos a apoyar el resto de la enseñanza que viene tras el tema 20, donde abundaremos en conceptos ya presentados, e introduciremos nuevos conceptos, pero todos fundamentados en esto primeros 20 temas, que conforman el nivel Básico.

TEMA 20

DESARROLLO DE UNA APLICACIÓN COMPLETA

En este tema vamos a desarrollar un editor de texto, con capacidad para cut/paste, con menú principal, barra de herramientas y status bar. Introduciremos algunos conceptos nuevos, pero pocos, como hemos hecho en los temas anteriores, para seguir siendo fieles al estilo de enseñanza gradual que estamos usando en este libro. El proyecto puedes abrirlo directamente en el Qt Creator, pero te recomiendo que sigas las explicaciones que se van a dar en este tema, y que luego por tu cuenta, intentes desarrollar lo mismo, o algo similar desde cero.

Vamos a crear un nuevo proyecto "Qt Gui Application", al que llamaremos "application", la clase se llamará MainWindow, su clase base será QMainWindow, y desactivaremos la opción "Generate form" para así crear nosotros mismos el interfaz gráfico en código puro. Lo siguiente que vamos a hacer es copiar los iconos que vamos a usar (directorio images) en nuestro proyecto, también en una carpeta llamada images, dentro de nuestro proyecto.

De nuevo hacemos clic derecho sobre le nuevo proyecto y añadimos un nuevo Qt->Qt Resource File, al que llamaremos también "application". Al abrirse el fichero qrc en el Qt Creator, aparece una ventana superior vacía, y abajo un botón Add. Pulsamos sobre él y luego sobre Add Prefix. En el campo de texto Prefix, vamos a poner "/". Volvemos a pulsar en Add-> Add Files, y navegamos hasta el directorio images donde hemos copiado los iconos que vamos a usar, seleccionamos los 6 iconos y los abrimos. Ya aparecen los 6 iconos adscritos al prefijo "/", así que ya podemos acceder a ellos usando una URL del tipo ":/images/icono.png", donde ":" hace referencia a los recursos internos.

Fichero main.cpp

```
#include <QApplication>
#include "mainwindow.h"
 int main(int argc, char *argv[])
 {
    Q_INIT_RESOURCE(application);

    QApplication app(argc, argv);
    app.setOrganizationName("Trolltech");
    app.setApplicationName("Application Ejemplo");

    MainWindow w;
    w.show();

    return app.exec();
 }
```

Usamos la macro Q_INIT_RESOURCE(nombre_base_del_qrc), aunque en la mayoría de los casos no sea necesario y los recursos se carguen automáticamente, pero en algunas plataformas, los recursos se guardan en una librería estática. Todo lo demás ya lo hemos visto anteriormente, a parte de los dos miembros de QApplication que se encargan de guardar en la aplicación el nombre de la empresa que hizo la aplicación y el nombre de la misma. Como vemos, vamos a usar una MainWindow que hemos derivado de QMainWindow y que esta implementada en los ficheros mainwindow.h y .cpp .

La clase MainWindow

Gran parte del código y de la estructura del mismo de esta clase, podrán ser usados en otras aplicaciones. Vamos a ver primeramente el widget que vamos a designar como widget central o principal de la Ventana Principal (MainWindow).

Al ser una aplicación editora de texto, vamos a usar QPlainTextEdit como el widget principal. Si consultas dicha clase en el asistente (Qt Assitant), podrás ver entre sus miembros, los slots que usaremos en los menús, como cut(), copy(), paste(). También podrás ver la función más importante, que es document(), que devuelve un puntero al texto del mismo del tipo QTextDocument. De entre los miembros de éste último, usaremos isModified() para saber si el documento ha sido modificado en algún momento.

Luego tendremos que mirar los slots que vamos a necesitar definir en la nueva clase. Puesto que cada entrada del menú requiere de una conexión entre la señal triggered() de la acción, y un slot que ejecuta la acción. Por lo que tendremos que dar un repaso a las opciones de menú que vamos a poner, para saber los slots que tendremos que declarar en esta clase.

Tendremos un menú File con los siguientes submenús (Nuevo, Abrir, Guardar, Guardar como y Salir). A excepción de la opción Salir, que la ejecutaremos con el slot close() de QWidget, el resto serán slots a declarar e implementar.

Tenemos también el menú Edit con los siguientes submenús (Cortar, Copiar y Pegar). Puesto que todos ellos pueden ejecutarse desde los slots predefinidos en QPlainTextEdit, respectivamente cut(), copy() y paste(), no es necesario implementar nuevos slots para este menú.

Finalmente tenemos el menú Ayuda con los submenús (Acerca de y Acerca de Qt), de los cuales, acerca de Qt viene ya preimplementado como el slot aboutQt() de QApplication.

Cuando el documento es modificado, debe de haber una acción que cambie la propiedad del widget modificado, windowModified, que lo hace el setter setWindowModified. Por tanto vamos a crear un slot que ejecute dicho cambio al que llamaremos documentWasModified().

Así los slots que vamos a declarar finalmente son:

```
private slots:
     void newFile();
     void open();
     bool save();
     bool saveAs();
     void about();
     void documentWasModified();
```

Lo siguiente que vamos a declarar son la funciones privadas, que se encargarán de llevar a cabo las tareas principales asignadas a los slots, y tareas secundarias propias del tipo de aplicación. Todas las tareas privadas que vamos a definir, son propias de una aplicación GUI genérica:

```
private:
     void createActions();
     void createMenus();
     void createToolBars();
     void createStatusBar();

     void readSettings();
     void writeSettings();

     bool maybeSave();
     void loadFile(const QString &fileName);
     bool saveFile(const QString &fileName);

     void setCurrentFile(const QString &fileName);
     QString strippedName(const QString &fullFileName);
```

Vamos a verlas una por una, ya que son fundamentales para el desarrollo de cualquier aplicación GUI, aunque puedan ser declaradas e implementadas a gusto del programador, pero es conveniente el estructurar tus aplicaciones de esta manera para hacer más legible y mantenible tu código.

createActions.- Aquí pondremos el código que crea todas las acciones de todos los menús, tal cual vimos en el tema anterior con QAction, y las conectaremos con los slots que declaramos antes. También podemos usar esta sección para deshabilitar las acciones que inicialmente lo deban de estar.

createMenus.- Aquí es donde vamos a poner el código que crea los menús, en nuestro caso Fichero, Editar y Ayuda, con sus separadores y todo.

createToolBars.- Aquí es donde pondremos el código que crea las barras de herramientas, en nuestro caso van a ser 2, una para opciones de Fichero y la otra para opciones de Editar.

createStatusBar.- Aquí es donde pondremos el código para crear la barra de estado, con todos sus elementos, y podremos mostrar un mensaje inicial si lo vemos oportuno.

readSettings.- Aquí pondremos el código que se va a encargar de leer los settings como vimos en el tema anterior, y los va a aplicar.

writeSettings.- Aquí podremos el código que se va a encargar de guardar los settings antes de salir de la aplicación.

setCurrentFile.- Es el setter de la propiedad CurrentFile que se encargará de llevar el nombre del fichero actual al que se hace alusión. Podrá hacer una serie de acciones relacionadas con el manejo de ficheros y notificación del cambio en el contenido de dicho fichero.

loadFile.- Este bloque se encargará de abrir el fichero y leer su contenido cargándolo en QPlainTextEdit con setPlainText(). Además hará todas las notificaciones que requiera sobre la carga del fichero, como establecer el nuevo CurrentFile o notificar su acción en el status bar.

saveFile.- Este bloque se encargará de abrir el fichero, escribir su contenido y notificar igualmente sobre el mismo.

maybeSave.- Este bloque puede estar integrado en los 2 anteriores, o puede ser separado para dejar más claro el código. Su función sería, si el documento ha sido modificado, muestra una ventana de diálogo warning, para que el usuario decida si guarda o no guarda los cambios. Es muy típico en las aplicaciones modernas.

strippedName.- Es una bloque simple que extrae el nombre del fichero, extrayéndolo del path completo.

Las variables privadas que vamos a usar corresponden con: la propiedad currentFile descrita antes, el widget central que es un QPlainTextEdit, los 3 menus QMenu, las 2 barras de herramientas QToolBar y las acciones propias a los menús QAction. Estas son las variables que vamos a usar:

```
QString curFile;
QPlainTextEdit *textEdit;

QMenu *fileMenu;
QMenu *editMenu;
QMenu *helpMenu;

QToolBar *fileToolBar;
QToolBar *editToolBar;

QAction *newAct;
QAction *openAct;
QAction *saveAct;
QAction *saveAsAct;
```

```
    QAction *exitAct;
    QAction *cutAct;
    QAction *copyAct;
    QAction *pasteAct;
    QAction *aboutAct;
    QAction *aboutQtAct;
```

Para finalizar la declaración de la clase MainWindow, vamos también a reimplementar una función virtual de QWidget llamada closeEvent(QCloseEvent) que nos permitirá interceptar el cierre de la aplicación, para notificar al usuario si el documento tiene cambios sin guardar, una característica muy usual también en aplicaciones modernas. La vamos a declarar en la zona protected para que pueda ser reimplementada en hijos que puedan derivarse de la clase MainWindow en futuras aplicaciones.

Implementación de MainWindow

A continuación vamos a ver la implementación de todos estos bloques y miembros que hemos visto anteriormente.

Comenzamos con el constructor que define el interfaz inicial de la aplicación:

```
MainWindow::MainWindow()
 {
    textEdit = new QPlainTextEdit;
    setCentralWidget(textEdit);

    createActions();
    createMenus();
    createToolBars();
    createStatusBar();

    readSettings();

    connect(textEdit->document(), SIGNAL(contentsChanged()),
            this, SLOT(documentWasModified()));

    setCurrentFile("");
 }
```

Quedan muy claras todas las acciones que lleva a cabo: establecer al QPlainText como el widget central, luego crear las acciones para los menús, luego crear los menus, las barras de herramientas, la barra de estado, leer los settings de el último cierre, conectar la señal de cambios en el documento con el slot que hemos creado para ello, y establecer como fichero actual (CurrentFile) a ninguno.

```
void MainWindow::closeEvent(QCloseEvent *event)
 {
    if (maybeSave()) {
        writeSettings();
        event->accept();
    } else {
        event->ignore();
    }
 }
```

La función closeEvent(), es ejecutada automáticamente cada vez que un top-level widget es cerrado. Si el evento es aceptado (accept()) entonces sale sin problemas, si es ignorado (ignore()) entonces no es cerrado. Aquí será maybeSave quien muestra la ventana de elección al usuario, y devolverá un bool de acuerdo al botón pulsado:

```
bool MainWindow::maybeSave()
 {
    if (textEdit->document()->isModified()) {
        QMessageBox::StandardButton ret; // es un int con un codigo
        ret = QMessageBox::warning(this, tr("Application"),
                     tr("El documento ha sido modificado.\n"
                        "_Quiere guardar los cambios?"),
                     QMessageBox::Save | QMessageBox::Discard | QMessageBox::Cancel);
        if (ret == QMessageBox::Save)
```

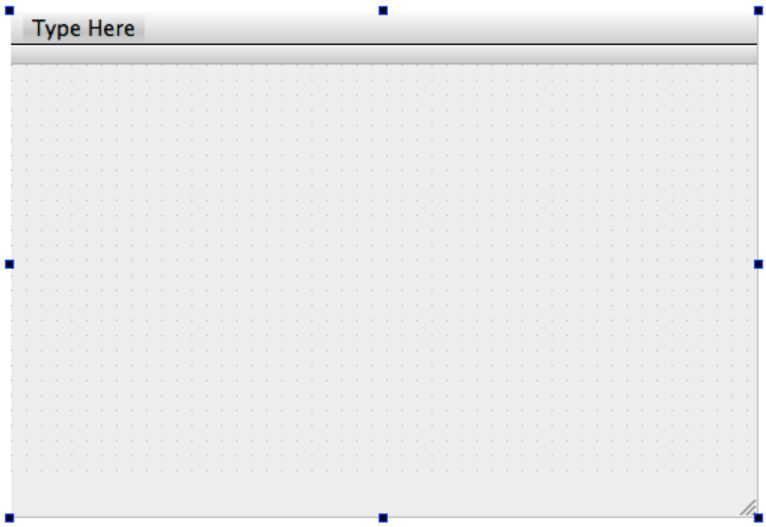
```
            return save();
        else if (ret == QMessageBox::Cancel)
            return false;
    }
    return true;
}
```

El resto del código puede ser entendido perfectamente con lo que hemos estudiado hasta ahora. Sólo haremos la introducción a un concepto nuevo que es la función QObject::tr(QString) que devuelve la traducción del argumento pasado, y es usado para internacionalizar una aplicación.

Ahora llegó el momento de hacer la misma aplicación pero usando el Qt Designer para diseñar el interfaz gráfico.

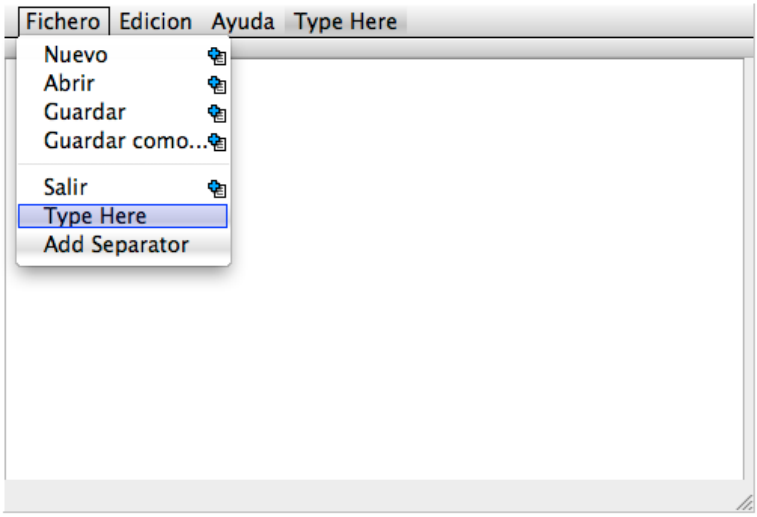
Crear la aplicación gráficamente

Hacer uso de Qt Designer para hacer una aplicación, facilita mucho las cosas, no solo para dibujar el interfaz si esto es necesario, si no para crear los menus, las acciones del mismo, y las conexiones con los slots. Pues bien vamos a abrir un nuevo proyecto, y esta vez si vamos a crear el fichero *.ui , le pondremos de nombre application2. Al abrir el ui veremos que aparece un formulario como éste:

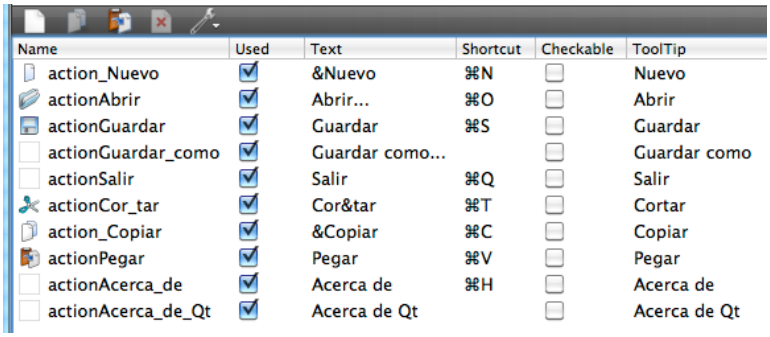


Así que le añadimos un PlainTextEdit y ocupamos con él todo el área que nos permite, veremos que en el inspector de objetos de la derecha superior, lo ha puesto como centralWidget.

Vamos a ir rellenado los menús haciendo doble clic sobre “Type Here” y pulsamos sobre Add Separator para añadir un separador de menú, por ejemplo antes del Salir. Lo que tendremos al final es algo como esto:

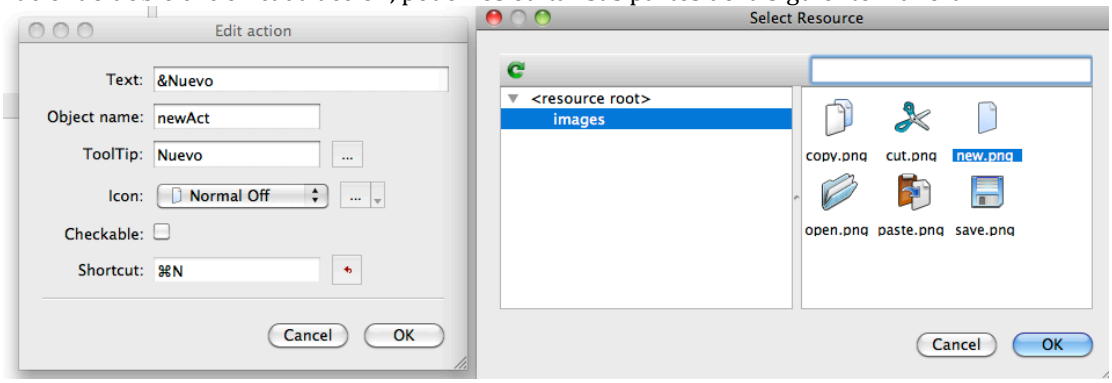


Así iremos rellenando los menús conforme hemos descrito al principio de este tema, opción a opción. Verás que en la parte inferior, en el Action Editor, empieza a rellenarse el sólo como esto:



Name	Used	Text	Shortcut	Checkable	ToolTip
action_Nuevo	☑	&Nuevo	⌘N	☐	Nuevo
actionAbrir	☑	Abrir...	⌘O	☐	Abrir
actionGuardar	☑	Guardar	⌘S	☐	Guardar
actionGuardar_como	☑	Guardar como...		☐	Guardar como
actionSalir	☑	Salir	⌘Q	☐	Salir
actionCor_tar	☑	Cor&tar	⌘T	☐	Cortar
action_Copiar	☑	&Copiar	⌘C	☐	Copiar
actionPegar	☑	Pegar	⌘V	☐	Pegar
actionAcerca_de	☑	Acerca de	⌘H	☐	Acerca de
actionAcerca_de_Qt	☑	Acerca de Qt		☐	Acerca de Qt

Haciendo doble clic en cada acción, podemos editar sus partes de la siguiente manera:



Por supuesto, se ha tener ya creado y cargado el qrc tal como lo vimos en el ejemplo anterior, para que aparezcan los iconos. En el inspector de propiedades ponemos el statusTip a mano. Vamos a hacer lo mismo con todos los submenús, y dejaremos para después la parte referente a las conexiones.

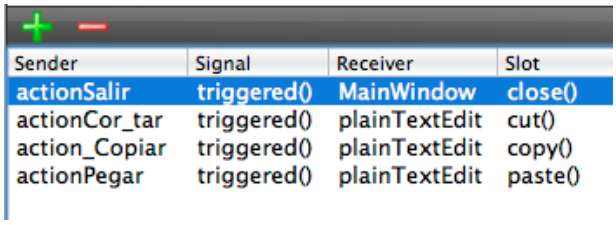
Ahora vamos a crear los slots que van a recibir las acciones, excepto los que ya vienen de herencia. Para ello, hacemos clic derecho sobre accion_Nuevo, y elegimos "go to slot", y pulsamos sobre la señal triggered(), que es la que vamos a conectar. Inmediatamente nos lleva al código del slot que se acaba de crear, *on_action_Nuevo_triggered()*, que se ha creado automáticamente. Todos los slots creados en el Qt Designer, llevan esta forma, on_ActionName_SignalName(). Así vamos a crear slots también para abrir, guardar, guardar como, acerca de. El resto de acciones las vamos a llevar a slots predefinidos en las clases correspondientes de Qt como hemos visto ya.

Los slots que se han creado automáticamente de esta manera son:

```
private slots:
    void on_actionAcerca_de_triggered();
    void on_actionGuardar_como_triggered();
    void on_actionGuardar_triggered();
    void on_actionAbrir_triggered();
    void on_action_Nuevo_triggered();
```

y además se ha establecido la conexión automáticamente entre la señal triggered() de dicha acción con dichos slots, por lo que no hace falta establecerlas en el código fuente, ya están en el *.ui.

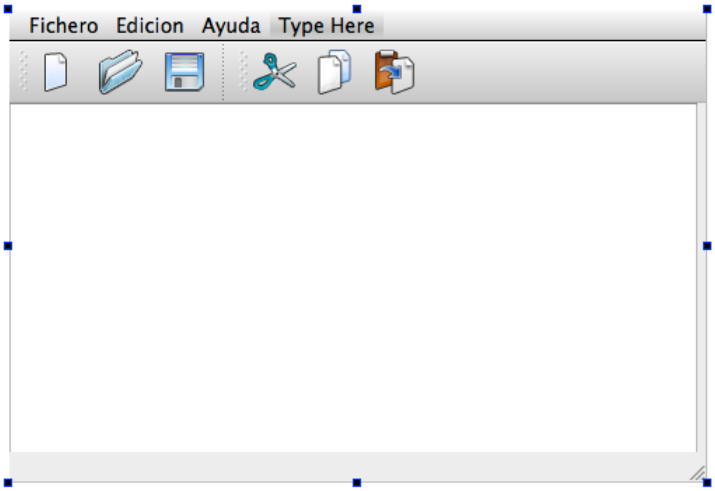
Ahora, nos vamos al Signal & Slots editor, donde esta todo vacío. Pulsamos sobre el botón verde +, y lo hacemos justo el número de veces que conexiones necesitamos para las acciones de los menús que van a slots predefinidos en Qt (son 4). Ahora vamos a ir rellenando todas las conexiones de cada acción, con los slots que les corresponden, con lo que una vez lleno quedaría de esta manera:



Sender	Signal	Receiver	Slot
actionSalir	triggered()	MainWindow	close()
actionCor_tar	triggered()	plainTextEdit	cut()
action_Copiar	triggered()	plainTextEdit	copy()
actionPegar	triggered()	plainTextEdit	paste()

Como puedes observar no hemos podido poner la conexión del aboutQtAct debido a que no aparece el receiver qApp, por lo que es parte tendremos que ponerla en el constructor de MainWindow a mano.

Volvamos a activar el Action Editor, y ahora hacemos clic derecho en la clase MainWindow en el inspector de objetos y elegimos “add toolbar”. Ya tenemos 2 toolbars, la primera la llamamos fileToolBar y la segunda editToolBar, cambiando su objectName en el inspector de propiedades. Arrastrando los iconos del Action editor de Nuevo, Abierto y Guardar sobre la primera ToolBar, creamos los 3 botones de dicha barra de herramientas. Hacemos lo mismo con Cortar, Copiar y Pegar en la segunda barra de herramientas. El resultado final es algo así:



Ya tenemos toda la parte gráfica construida, y ya sólo nos queda ir rellenando el código correspondiente de la clase MainWindow. Ahora eligiendo las acciones de Cortar y Pegar, vamos a desactivarlas por defecto en el panel de propiedades, deshabilitando el cuadro “enabled”. Verás que los iconos se vuelven más apagados. Y esto es todo lo que se puede hacer con el Qt Designer, el resto hay que teclearlo a mano.

Si revisamos el código de mainwindow.h, veremos que hay algo nuevo que hace referencia al contenido que hay en el fichero *.ui, y es el código:

```
namespace Ui {
    class MainWindow;
}

private:
    Ui::MainWindow *ui;
```

Lo que designa ese código es un puntero al interface que hay en ui, y por lo tanto para acceder a los objetos dentro del mismo, tendremos que hacerlo desde ese puntero, lo cual es tan sencillo como esto:

```
ui->actionGuardar->setEnabled(false);
```

que desactivaría directamente la acción Guardar, y que si pruebas a escribirlo en cualquier parte de la implementación de MainWindow, verás que el autorellenado te va dando pistas de todos los objetos y miembros accesibles desde ui.

Dejamos por tanto como ejercicio de repaso para el lector, la cumplimentación del código necesario para que esta aplicación funcione igual que su antecesora hecha completamente en código. No obstante te obsequiamos con el código del constructor de la clase MainWindow para que te sirva como orientación:

```
MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    setCentralWidget(ui->plainTextEdit);

    connect(ui->actionAcerca_de_Qt, SIGNAL(triggered()),
            qApp, SLOT(aboutQt()));
    connect(ui->plainTextEdit, SIGNAL(copyAvailable(bool)),
            ui->actionCor_tar, SLOT(setEnabled(bool)));
    connect(ui->plainTextEdit, SIGNAL(copyAvailable(bool)),
            ui->action_Copiar, SLOT(setEnabled(bool)));

    ui->statusBar->showMessage(tr("Preparado"));
    readSettings();

    connect(ui->plainTextEdit->document(), SIGNAL(contentsChanged()),
            this, SLOT(documentWasModified()));

    setCurrentFile("");
}
```

Resumiremos por tanto los 7 pasos a seguir para crear una aplicación con ayuda del Qt Designer (forma visual):

1.- Situar el centerWidget debajo del toolbar y encima del status bar.
2.- Escribir todas las opciones del menú principal.
3.- Editar las acciones que se crean con el menú principal cumplimentado para añadirles iconos, status tips, su estado inicial, etc.
4.- Crear las toolbar que se requieran arrastrando los iconos de las acciones sobre aquellas.
5.- Crear los slots que se requieran de las acciones con botón derecho-> go to slot, luego darlos de alta en el Qt Designer como miembros de MainWindow.
6.- Hacemos todas las conexiones entre las acciones y los slots en el signal-slot editor.
7.- Ya podemos añadir el código que haga falta, con ayuda del nuevo puntero *ui para designar el diseño.

TEMA 21

LAYOUT A FONDO

En el tema 12 ya tocamos el tema del layout como el medio mediante el cual, los widgets negociaban el espacio de la ventana que los contiene, creando una interface flexible a múltiples circunstancias como cambio de idiomas sin truncado de palabras, o adaptación del GUI a otras plataformas. En este capítulo vamos a añadir nuevas clases a este estudio para disponer el espacio de una ventana viendo QSplitter, que crea barras que dividen una ventana en varios espacios, QScrollArea, que permite que una zona pueda verse en parte y el resto pueda ser accedido mediante una barra de scroll, QWorkSpace o QMdiArea que nos introducirá en las aplicaciones MDI (multi-documento), donde se podrán abrir más de un documento a la vez en la misma ventana.

Como dijimos entonces, todo widget ocupa un espacio rectangular, como una caja, y las coordenadas de las vértices superior izquierdo e inferior derecho se guardan en la propiedad "geometry", que puede ser accedida mediante el getter geometry() como vimos en el ejemplo del tema anterior para los settings, y puede ser cambiado mediante el setter setGeometry(). Sin embargo, nadie usa estos métodos para posicionar de manera absoluta ningún widget dentro de la ventana, si no que usa de layouts para ello.

QStackedLayout

Esta clase crea un conjunto de widgets hijos que presentan páginas apiladas, y sólo puede mostrar una a la vez, ocultando el resto al usuario. QStackedWidget, es una widget que provee Qt, y que lleva dentro un QStackedLayout. No se provee de ningún control para que el usuario pueda navegar por las páginas, así que esto sólo puede hacer mediante el uso de código. Las páginas están numeradas del 0 en adelante, y podemos hacer una visible mediante el setter setCurrentIndex(page). Para averiguar el index de un widget contenido en este Layout se usa la función indexOf(QWidget), que devuelve su entero.

Bien, una de tantas formas de hacer funcionar este Layout, es vincular las opciones de un widget con las diferentes páginas del stacked layout. Esto se puede hacer con un ComboBox, o una QListWidget. Vamos a ver un ejemplo:

```
PreferenceDialog::PreferenceDialog(QWidget *parent) : QDialog(parent)
{
    //appearancePage, webBrowserPage, mailAndNewsPage, advancedPage
    listWidget = new QListWidget;
    listWidget->addItem(tr("Apariencia"));
    listWidget->addItem(tr("Navegador"));
    listWidget->addItem(tr("Correo y foros"));
    listWidget->addItem(tr("Avanzado"));

    stackedLayout = new QStackedLayout;
    stackedLayout->addWidget(appearancePage);
    stackedLayout->addWidget(webBrowserPage);
    stackedLayout->addWidget(mailAndNewsPage);
    stackedLayout->addWidget(advancedPage);

    connect(listWidget, SIGNAL(currentRowChanged(int)),
            stackedLayout, SLOT(setCurrentIndex(int)));
    ......
    listWidget->setCurrentRow(0);
}
```

Aquí previamente, appearancePage, webBrowserPage, mailAndNewsPage y advancedPage, son widgets, por ejemplo layouts con toda una serie de widgets dentro, que conforman una ventana de diálogo completa. Son añadidos como widgets a su vez del stacked layout, y lo que hacemos es vincular una listWidget con él mediante la señal currentRowChanged(int) que vimos en el tema

17, con el setter y además slot setCurrentIndex(int), ya que ambos pasan un índice igual (desde 0 a N). De esta manera podemos disponer espacialmente a ambos widgets contenendores, dentro de una ventana de diálogo de selección de preferencias. Fíjate en el detalle, de que fijamos la posición inicial del ambos con setCurrentRow(0).

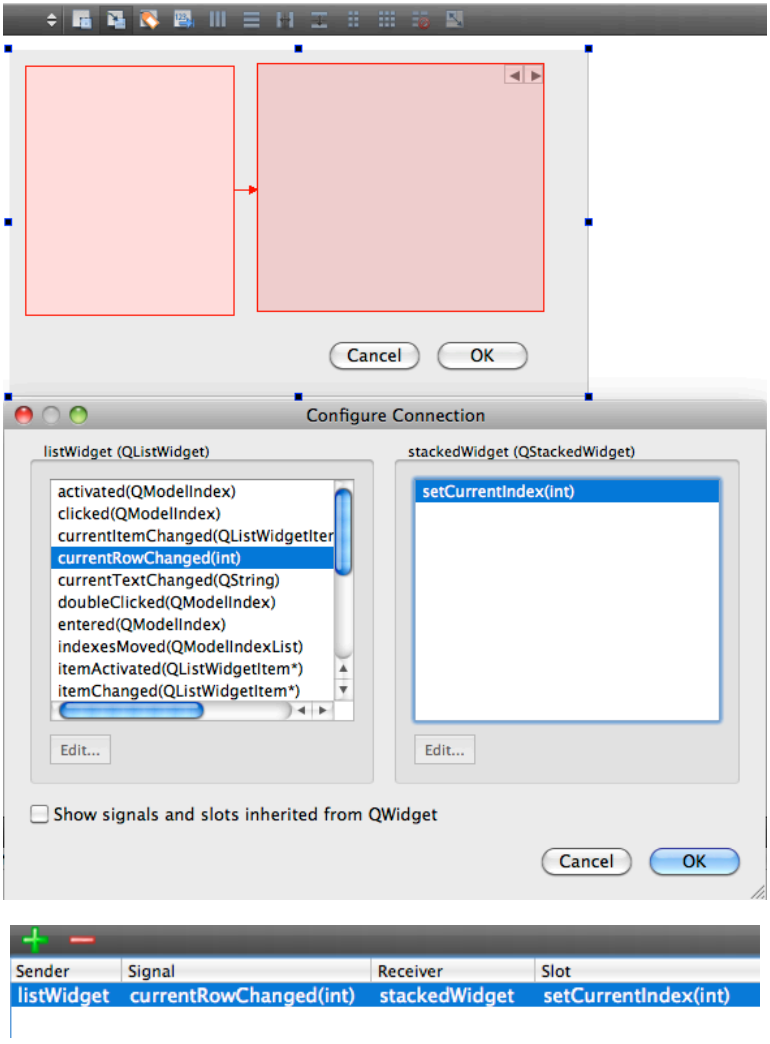
¿Cómo se haría esto mismo en el Qt Designer?. Pues siguiendo estos sencillos pasos:

1.- Crear un formulario nuevo basado en QDialog o en QWidget.

2.- Añadir un QListWidget y un QStackedWidget al mismo.

3.- Rellenar cada página con widgets y layouts conforme sea necesario. (haciendo clic derecho sobre el layout, y escogiendo "Insert Page" para añadir nuevas páginas, y para navegar por ellas, usando las flechitas en la zona superior derecha).

4.- Conectar ambos widgets, pulsando en el botón "Edit signal" de arriba del Designer, y pulsamos sobre el ListWidget y arrastramos hacia el StackedWidget, de manera que una flecha los conectará. Al soltar el ratón aparecerá la ventana para elegir la señal y el slot que estarán conectados.



5.- Añadimos los ítems de las opciones en el ListWidget con clic derecho->edit ítems. Luego ponemos la propiedad currentRow del listWidget a 0.

Así de fácil sería diseñarlo, luego habría que añadir los botones de la ventana de diálogo típicos, OK y Cancel (QButtonBox), y de igual manera conectarlos con la ventana de diálogo, y conectar las señales, accepted() de parte de los botones, con accept() del diálogo, y rejected() de parte de

los botones, con reject() del diálogo. Ya podríamos usar este diálogo en la aplicación completa, usando dialog.exec() para ejecutarlo en modo modal y que nos devuelva un valor con nuestra selección, QDialog::Accepted ó QDialog::Rejected.

QSplitter

Es un widget que contiene otros widgets, y estos están separados por separadores. Los usuarios pueden cambiar el tamaño de los widgets separados, desplazando los separadores. Los widgets separados, se van poniendo uno al lado del otro conforme se van creando y según la disposición.

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

    QTextEdit *editor1 = new QTextEdit;
    QTextEdit *editor2 = new QTextEdit;
    QTextEdit *editor3 = new QTextEdit;

    QSplitter splitter(Qt::Horizontal);
    splitter.addWidget(editor1);
    splitter.addWidget(editor2);
    splitter.addWidget(editor3);

    splitter.show();

    return app.exec();
}
```

Los splitters pueden anidarse horizontales con verticales y crear disposiciones más complejas. Los setting pueden guardar la posición de los mismos, usando grupos en la grabación de los mismos.

```
void MailClient::writeSettings()
{
    QSettings settings("Software Inc.", "Mail Client");

    settings.beginGroup("mainWindow");
    settings.setValue("size", size());
    settings.setValue("mainSplitter", mainSplitter->saveState());
    settings.setValue("rightSplitter", rightSplitter->saveState());
    settings.endGroup();
}

void MailClient::readSettings() {
    QSettings settings("Software Inc.", "Mail Client");

    settings.beginGroup("mainWindow");
    resize(settings.value("size", QSize(480, 360)).toSize());
    mainSplitter->restoreState(settings.value("mainSplitter").toByteArray());
    rightSplitter->restoreState(settings.value("rightSplitter").toByteArray());
    settings.endGroup();
}
```

¿Cómo diseñar con splitters en el Qt Designer?. Es similar a como disponíamos los widgets con layouts verticales y horizontales. Pones 2 o más widgets juntos, los seleccionas, y en la barra de herramientas superior podemos elegir, “Layout Horizontally in Splitter” o “Layout Vertically in Splitter”.

QScrollArea

Provee una zona visible (viewport) y 2 barras de scroll (vertical y horizontal). Para añadir barras de scrolling a un widget, solo hay que crear un objeto QScrollArea y añadirle dicho

objeto con setWidget(QWidget *). Podemos acceder al widget y sus miembros mediante QScrollArea ::viewport() que devuelve un puntero del mismo.

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

    IconEditor *iconEditor = new IconEditor;
    iconEditor->setIconImage(QImage(":/images/mouse.png"));

    QScrollArea scrollArea;
    scrollArea.setWidget(iconEditor);
    scrollArea.viewport()->setBackgroundRole(QPalette::Dark);
    scrollArea.viewport()->setAutoFillBackground(true);
    scrollArea.setWindowTitle(QObject::tr("Icon Editor"));
    scrollArea.show();

    return app.exec();
}
```

QTextEdit y QAbstractItemView (y derivados), al estar derivados de QAbstractScrollArea, no requieren esto para poseer barras de desplazamiento.

QDockWidget y QMainWindow

Un dock widget es un widget que puede aparcarse en un lado de una MainWindow, o puede quedarse flotando como una ventana aparte. Las área de enganche de las dock windows, en MainWindow son, una arriba, otra abajo, y una a cada lado del central Widget.

Estas ventanas tiene su propio título incluso enganchadas a la principal, y el usuario puede moverlas solo cogiendo del título y arrastrando de ellas. Pueden ser cerradas como una ventana normal. Con la función setWidget(QWidget), se añade un widget a un dock window.

QMainWindow, tiene una función para añadir docks widgets, que se llama addDockWidget(zona, widget).

```
QDockWidget *shapesDockWidget = new QDockWidget(tr("Shapes"));
shapesDockWidget->setWidget(treeWidget);
shapesDockWidget->setAllowedAreas(Qt::LeftDockWidgetArea | Qt::RightDockWidgetArea);
addDockWidget(Qt::RightDockWidgetArea, shapesDockWidget);
```

MDI

Toda aplicación que dentro de su ventana central pueda abrir varios documentos a la vez, es llamada una aplicación MDI. Se hace haciendo que la clase QWorkspace sea el centralWidget, y entonces cada documento que se abra, que sea hijo de QWorkspace.

Suele ser muy típico que una aplicación MDI, provea de un menu Window, que permite manejar los distintos documentos, y activarlos o desactivarlos, y esa opción suele ser checkable indicando en cada momento la ventana del documento que está activa.

```
MainWindow::MainWindow()
{
    workspace = new QWorkspace;
    setCentralWidget(workspace);
    connect(workspace, SIGNAL(windowActivated(QWidget *)),
            this, SLOT(updateMenus()));

    createActions();
    createMenus();
    createToolBars();
    createStatusBar();

    setWindowTitle(tr("MDI Editor"));
    setWindowIcon(QPixmap(":/images/icon.png"));
}
```

Este es un constructor típico de una aplicación MDI. Conectamos la señal windowActivated() con un slot nuestro que programaremos que actulizará el contenido de los menús (por ejemplo, haciendo toggle en el menú Window).

Cada vez que se pulsa en el menú Fichero->Nuevo, no se borrará el documento actual, si no que se creará uno nuevo devolviendo un puntero al mismo.

El constructor de los widgets documentos que se abren, debe de activar el atributo de borrado al cerrar, para evitar los memory leaks.

```
setAttribute(Qt::WA_DeleteOnClose);
```

Para poder manejar cada documento, es necesario poder recuperar un puntero al mismo, lo cual sería imposible sin el uso de un casting dinámico.

```
Editor *MainWindow::activeEditor()
{
    return qobject_cast<Editor *>(workspace->activeWindow());
}
```

Aquí, la función activeEditor de MainWindow, puede recuperar el puntero a dicho documento (en este caso es un Editor), haciendo un casting al valor devuelto por QWorkspace::activeWindow(), que es un QWidget *. Si no hay activa ninguna ventana entonces devolverá un NULL pointer, por lo que es importante que cuando se llame a este método desde cualquier parte, se observe si devuelve un valor nulo.

En este tema vamos a hacer que nuestra aplicación editora de texto del tema 20, pueda ser MDI, pero lo haremos usando QMdiArea y QMdiSubWindow, como nuevo enfoque a las aplicaciones MDI.

TEMA 22

BASES DE DATOS

La gran mayoría de aplicaciones de gestión, suelen guardar toda la ingente cantidad de datos que maneja en un motor de base de datos. Qt en su versión privativa incluye los drivers de la mayoría de motores de datos. Qt en su versión Open Source, no incluye aquellos que tienen licencia. Para aquellos usuarios acostumbrados a usar la sintaxis SQL, Qt provee de la clase QSqlQuery. Para usuarios que prefieren evitar esto, y desean trabajar a un nivel más alto, hay 2 modelos incluidos en las clases QSqlTableModel y QSqlRelationalTableModel, las cuales ya mencionamos en el tema 16.

Conectar e interrogar con SQL

Vamos a empezar tratando la primera forma de trabajar con bases de datos, donde conectaremos con una base de datos MySQL, y la interrogaremos mediante la sintaxis SQL.

```
bool createConnection()
{
    QSqlDatabase db = QSqlDatabase::addDatabase("QMYSQL");
    db.setHostName("localhost");
    db.setDatabaseName("megafonia");
    db.setUserName("root");
    db.setPassword("admin");
    if (!db.open()) {
        QMessageBox::critical(0, QObject::tr("Database Error"),db.lastError().text());
        return false;
    }
    return true;
}
```

Esta es una conexión a la base de datos MySQL, en el host local, con nombre megafonia, con el usuario y password del usuario que tenga acceso a la misma, de acuerdo a las queries SQL que vamos a enviarle.

Lo siguiente será enviarle una query, y esto se hace así:

```
QSqlQuery query;
query.exec("SELECT almacen, ntiendas FROM almacenes");
while (query.next()) {
    QString almacen = query.value(0).toString();
    int ntiendas = query.value(1).toInt();
    qDebug("Nombre: %s - Tiendas: %d",qPrintable(almacen),ntiendas);
}
```

Como puedes ver la función exec() ejecuta el query, next() recorre el puntero por la tabla respuesta, y value es una lista QVariant, donde los campos tienen índices del 0 en adelante, de esta forma value(0) contendría el campo almacen que es el primero en la respuesta, y en value(1) estaría el campo ntiendas.

Para que este código compile bien, es necesario que se añada el módulo sql al proyecto, es decir al fichero *.pro con la linea:

QT += sql

MVC aplicado a Bases de Datos

Como ya vimos en el tema 16, podemos combinar una vista y un modelo que acceda a la base de datos como su fuente. Aquí la vista más típica para ver una tabla de datos es QTableView, y el modelo puede ser QSqlTableModel, si vamos a ver una tabla en concreto, sin más.

Las funciones más típicas de este modelo, son:

setTable(QString tableName) -> que fija la tabla de la base de datos a fijar

setEditStrategy(strategy) -> Describe la estrategia a usar para actualizar los datos de la base de datos, que puede ser: OnFieldChange (todo cambio en la vista es inmediato en la base de datos), OnRowChange(se guardan los datos al cambiar de fila o registro), OnManualSubmit (se guarda en una cache hasta que el programa ejecuta un submitAll() o un revertAll()).

select() -> Rellena el modelo con los datos de la tabla.

setHeaderData(int section, Qt::orientation, QVariant header) -> Marca los nombres de las cabeceras.

clear() -> borra el modelo de todos los datos que previamente tuviera cargados en su caché (buffer).

Para los siguientes ejemplos vamos a usar una base de datos grabada en SQLite en un fichero que llamaremos database.db, en los ejercicios veremos como es esta base de datos.

```
#include <QtGui>
#include <QtSql>

static bool createConnection()
{
    QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");
    db.setDatabaseName("database.db");
    if (!db.open()) {
        QMessageBox::critical(0, "No se puede abrir la base de datos",
                    "Imposible establecer una conexi_n con la base de datos.\n"
                    "Pulsar Cancel para salir.", QMessageBox::Cancel);
        return false;
    }
return true;
}

void initializeModel(QSqlTableModel *model)
{
     model->setTable("person");
     model->setEditStrategy(QSqlTableModel::OnManualSubmit);
     model->select();
     model->setHeaderData(0, Qt::Horizontal, "ID");
     model->setHeaderData(1, Qt::Horizontal, "First name");
     model->setHeaderData(2, Qt::Horizontal, "Last name");
}

QTableView *createView(const QString &title, QSqlTableModel *model)
{
     QTableView *view = new QTableView;
     view->setModel(model);
     view->setWindowTitle(title);
     return view;
}

int main(int argc, char *argv[])
{
     QApplication app(argc, argv);
     if (!createConnection()) return 1;
     QSqlTableModel model;
     initializeModel(&model);
     QTableView *view1 = createView("Table Model (Vista 1)", &model);
     QTableView *view2 = createView("Table Model (Vista 2)", &model);
     view1->show();
     view2->move(view1->x() + view1->width() + 20, view1->y()); view2->show();
     return app.exec();
 }
```

Pero lo más normal es que tengamos varias tablas, relacionadas entre ellas mediante un identificador numérico (en SQLite mediante el rowid), y lo que queremos es presentar una tabla donde aparece la relación entre ellas. Para ello usaremos el modelo QSqlRelationalTableModel. Veamos un ejemplo similar con la misma base de datos y las 3 tablas.

```
#include <QtGui>
#include <QtSql>

static bool createConnection()
{
    QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");
    db.setDatabaseName("database.db");
    if (!db.open()) {
        QMessageBox::critical(0, "No se puede abrir la base de datos",
                    "Imposible establecer una conexi_n con la base de datos.\n"
                    "Pulsar Cancel para salir.", QMessageBox::Cancel);
        return false;
    }
return true;
}

void initializeModel(QSqlRelationalTableModel *model)
 {
     model->setTable("employee");
     model->setEditStrategy(QSqlTableModel::OnManualSubmit);
     model->setRelation(2, QSqlRelation("city", "id", "name"));
     model->setRelation(3, QSqlRelation("country", "id", "name"));
     model->setHeaderData(0, Qt::Horizontal, "ID");
     model->setHeaderData(1, Qt::Horizontal, "Name");
     model->setHeaderData(2, Qt::Horizontal, "City");
     model->setHeaderData(3, Qt::Horizontal, "Country");
     model->select();
 }

QTableView *createView(const QString &title, QSqlTableModel *model)
 {
     QTableView *view = new QTableView;
     view->setModel(model);
     view->setItemDelegate(new QSqlRelationalDelegate(view));
     view->setWindowTitle(title);
     return view;
 }

void createRelationalTables()
 {
     QSqlQuery query;
     query.exec("create table employee(id int primary key, name varchar(20), city int,
country int)");
     query.exec("insert into employee values(1, 'Espen', 5000, 47)");
     query.exec("insert into employee values(2, 'Harald', 80000, 49)");
     query.exec("insert into employee values(3, 'Sam', 100, 1)");
     query.exec("create table city(id int, name varchar(20))");
     query.exec("insert into city values(100, 'San Jose')");
     query.exec("insert into city values(5000, 'Oslo')");
     query.exec("insert into city values(80000, 'Munich')");
     query.exec("create table country(id int, name varchar(20))");
     query.exec("insert into country values(1, 'USA')");
     query.exec("insert into country values(47, 'Norway')");
     query.exec("insert into country values(49, 'Germany')");
 }

int main(int argc, char *argv[])
 {
     QApplication app(argc, argv);
     if (!createConnection()) return 1;
     createRelationalTables();
     QSqlRelationalTableModel model;
     initializeModel(&model);
     QTableView *view = createView("Relational Table Model", &model);
     view->show();
     return app.exec();
 }
```

TEMA 23

REDES TCP/IP

Cliente TCP (QTcpSocket)

Explicar los protocolos TCP/IP no es el objetivo de este libro, por lo que se da por supuesto el previo conocimiento y diferenciación entre protocolos TCP y UDP. Así que comenzaremos con la implementación de un cliente TCP.

Se trata de un servidor que una sirve por un puerto TCP, frases de la fortuna, como las de los rollitos de primavera. Así que cuando el cliente se conecta, el servidor le sirve una frase al azar sobre su fortuna.

Vamos a usar QDataStream para enviar los datos desde el servidor, y también para recogerlas en el cliente. El flujo de datos es muy sencillo, tan solo lleva como cabecera un quint16 con la longitud del mensaje que le sigue, luego viene una cadena con la frase.

Bien, en programación de sockets hay 2 aproximaciones distintas en cuanto a si la conexión es bloqueante o no-bloqueante. Una conexión bloqueante, es aquella que espera y no hace nada hasta que el servidor ha respondido y ha sido establecida por completo. Suele usarse en aplicaciones multi-hilo o de consola donde se hará uso de funciones como QTcpSocket::waitForConnected. En nuestro caso, vamos a usar la aproximación no-bloqueante, donde llamaremos a la función connectToHost(), y seguiremos a lo nuestro, y mediante una conexión de Qt, una vez el socket se establezca QTcpSocket emitirá la señal connected().

Vamos a derivar la clase Cliente de QDialog, y le vamos a añadir los slots necesarios y las propiedades privadas necesarias.

```
class Client : public QDialog
{
    Q_OBJECT

public:
    Client(QWidget *parent = 0);

private slots:
    void requestNewFortune();
    void readFortune();
    void displayError(QAbstractSocket::SocketError socketError);
    void enableGetFortuneButton();

private:
    QLabel *hostLabel;
    QLabel *portLabel;
    QLineEdit *hostLineEdit;
    QLineEdit *portLineEdit;
    QLabel *statusLabel;
    QPushButton *getFortuneButton;
    QPushButton *quitButton;
    QDialogButtonBox *buttonBox;

    QTcpSocket *tcpSocket;
    QString currentFortune;
    quint16 blockSize;
};
```

Aparte de los componentes visuales, vamos a usar la cadena currentFortune para guardar la frase actual, y su tamaño en blockSize. Los slots que vamos a usar son:

requestNewFortune -> que va a ser ejecutado cuando se haga clic en el botón de “Ver mi fortuna”.

readFortune -> que será disparado cuando el socket TCP mediante el QDataStream, envie la señal QIODevice::readyRead(), diciendo que nuevos datos están disponibles en el dispositivo de entrada/salida.

displayError -> va a estar conectado con la señal error() que se dispara si hay error en la conexión.

enableGetFortuneButton -> este es un mero slot de GUI para que se habilite un botón.

Vamos ahora a por la implementación del constructor que es donde se inicia el socket:

```
Client::Client(QWidget *parent)
   : QDialog(parent)
{
   hostLabel = new QLabel("&IP Servidor:");
   portLabel = new QLabel("&Puerto:");
   // Busca todas las direcciones del host actual
   QString ipAddress;
   QList<QHostAddress> ipAddressesList = QNetworkInterface::allAddresses();
   // va a usar la primera IP no localhost (127.0.0.1) tipo IPv4
   for (int i = 0; i < ipAddressesList.size(); ++i) {
      if (ipAddressesList.at(i) != QHostAddress::LocalHost &&
         ipAddressesList.at(i).toIPv4Address()) {
         ipAddress = ipAddressesList.at(i).toString();
         break;
      }
   }
   // si no encuentra ninguna usará la localhost (127.0.0.1)
   if (ipAddress.isEmpty())
      ipAddress = QHostAddress(QHostAddress::LocalHost).toString();
   hostLineEdit = new QLineEdit(ipAddress);
   portLineEdit = new QLineEdit;
   portLineEdit->setValidator(new QIntValidator(1, 65535, this));
   hostLabel->setBuddy(hostLineEdit);
   portLabel->setBuddy(portLineEdit);
   statusLabel = new QLabel(("Este ejemplo requiere que usted ejecute el Fortune server"));
   getFortuneButton = new QPushButton("Ver mi Fortuna");
   getFortuneButton->setDefault(true);
   getFortuneButton->setEnabled(false);
   quitButton = new QPushButton("Salir");
   buttonBox = new QDialogButtonBox;
   buttonBox->addButton(getFortuneButton, QDialogButtonBox::ActionRole);
   buttonBox->addButton(quitButton, QDialogButtonBox::RejectRole);
   tcpSocket = new QTcpSocket(this);

   connect(hostLineEdit, SIGNAL(textChanged(QString)),this, SLOT(enableGetFortuneButton()));
   connect(portLineEdit, SIGNAL(textChanged(QString)),this, SLOT(enableGetFortuneButton()));
   connect(getFortuneButton, SIGNAL(clicked()),this, SLOT(requestNewFortune()));
   connect(quitButton, SIGNAL(clicked()), this, SLOT(close()));
   connect(tcpSocket, SIGNAL(readyRead()), this, SLOT(readFortune()));
   connect(tcpSocket, SIGNAL(error(QAbstractSocket::SocketError)),this, SLOT(displayError(QAbstractSocket::SocketError)));

   QGridLayout *mainLayout = new QGridLayout;
   mainLayout->addWidget(hostLabel, 0, 0);
   mainLayout->addWidget(hostLineEdit, 0, 1);
   mainLayout->addWidget(portLabel, 1, 0);
   mainLayout->addWidget(portLineEdit, 1, 1);
   mainLayout->addWidget(statusLabel, 2, 0, 1, 2);
   mainLayout->addWidget(buttonBox, 3, 0, 1, 2);
   setLayout(mainLayout);
   setWindowTitle("Cliente TCP"));
   portLineEdit->setFocus();
}
```

Aquí a parte de los widgets visuales, inicializamos la IP del ordenador donde estamos en la LineEdit, también le añadimos un validador al LineEdit del puerto para que no admita puertos fuera del rango de 1-65535. Se inicializa el socket y se hacen las conexiones Qt entre slots y señales comentadas antes.

Cuando se pulsa el botón de Ver Mi Fortuna, se dispara el slot correspondiente que establece la conexión con el servidor de esta manera:

```
void Client::requestNewFortune()
{
    getFortuneButton->setEnabled(false);
    blockSize = 0;
    tcpSocket->abort();
    tcpSocket->connectToHost(hostLineEdit->text(),portLineEdit->text().toInt());
}
```

Inicializamos la propiedad blockSize con el tamaño a 0, luego aborta la conexión actual si la hay, reseteando el socket, luego conecta con el host dando su dirección IP y puerto.

Cuando el socket ya haya sido conectado correctamente, y haya datos nuevos en el dispositivo, entonces se lanzará el slot correspondiente:

```
void Client::readFortune()
{
    QDataStream in(tcpSocket);
    in.setVersion(QDataStream::Qt_4_6);

    if (blockSize == 0) {
        if (tcpSocket->bytesAvailable() < (int)sizeof(quint16))
            return;
        in >> blockSize;
    }

    if (tcpSocket->bytesAvailable() < blockSize)
        return;

    QString nextFortune;
    in >> nextFortune;

    if (nextFortune == currentFortune) {
        QTimer::singleShot(0, this, SLOT(requestNewFortune()));
        return;
    }

    currentFortune = nextFortune;
    statusLabel->setText(currentFortune);
    getFortuneButton->setEnabled(true);
}
```

Los datos, no tienen por qué venir todos de golpe, y por tanto requiera de varias ejecuciones de este slot, para tenerlos todos. Por ello, si es la primera lectura (blockSize == 0), y si los bytes que están preparados para ser leídos son menos que 2 bytes, el tamaño del header que indica el tamaño de la frase, entonces salimos del slot y esperamos a que nos llamen con más datos. Si hay más datos, pues leemos primero el blockSize. Luego igualmente salimos del slot, mientras no estén todos los datos disponibles. Finalmente leemos el mensaje que recogemos en una cadena QString. Si el mensaje es el mismo que el anterior que nos dio el servidor, entonces usamos un Timer de único disparo, para que dispare de nuevo el slot requestNewFortune que ya describimos antes, pidiendo un mensaje nuevo.

Finalmente vamos a ver el código que va a tratar los diferentes errores de conexión:

```
void Client::displayError(QAbstractSocket::SocketError socketError)
{
    switch (socketError) {
    case QAbstractSocket::RemoteHostClosedError:
        break;
    case QAbstractSocket::HostNotFoundError:
        QMessageBox::information(this, "Cliente TCP","No se encontró el servidor.");
        break;
    case QAbstractSocket::ConnectionRefusedError:
        QMessageBox::information(this,"Cliente TCP","Conexión rechazada por el
cliente. Servidor apagado.");
        break;
    default:
        QMessageBox::information(this, "Cliente TCP",
                                 QString("Error: %1.")
                                 .arg(tcpSocket->errorString()));
    }
    getFortuneButton->setEnabled(true);
}
```

No hay comentarios que añadir a esto.

Servidor TCP (QTcpSocket)

El servidor, es también derivado de QDialog. Su único slot es sendFortune, que está conectado a la señal newConnection().

```
class Server : public QDialog
{
    Q_OBJECT

public:
    Server(QWidget *parent = 0);

private slots:
    void sendFortune();

private:
    QLabel *statusLabel;
    QPushButton *quitButton;
    QTcpServer *tcpServer;
    QStringList fortunes;
};
```

La implementación del constructor es muy simple:

```
Server::Server(QWidget *parent)
    : QDialog(parent)
{
    statusLabel = new QLabel;
    quitButton = new QPushButton("Salir");
    quitButton->setAutoDefault(false);
    tcpServer = new QTcpServer(this);
    if (!tcpServer->listen()) {
        QMessageBox::critical(this, "Servidor TCP",
                              QString("Imposible iniciar servidor: %1.")
                              .arg(tcpServer->errorString()));
        close();
        return;
    }
    QString ipAddress;
    QList<QHostAddress> ipAddressesList = QNetworkInterface::allAddresses();
    for (int i = 0; i < ipAddressesList.size(); ++i) {
        if (ipAddressesList.at(i) != QHostAddress::LocalHost &&
            ipAddressesList.at(i).toIPv4Address()) {
            ipAddress = ipAddressesList.at(i).toString();
            break;
        }
    }
    if (ipAddress.isEmpty())
        ipAddress = QHostAddress(QHostAddress::LocalHost).toString();
    statusLabel->setText(tr("Servidor corriendo en \n\nIP: %1\npuerto: %2\n\n"
                            "Ejecute el cliente TCP ahora.")
                         .arg(ipAddress).arg(tcpServer->serverPort()));
    fortunes << tr("Has llevado una vida de perros. Bajate del sofa.")
             << tr("Tienes que pensar en el ma_ana.")
             << tr("Te ha sorprendido una sonido fuerte.")
             << tr("Tendr_s hambre dentro de una hora.")
             << tr("Tienes nuevo correo.");
    connect(quitButton, SIGNAL(clicked()), this, SLOT(close()));
    connect(tcpServer, SIGNAL(newConnection()), this, SLOT(sendFortune()));
    QHBoxLayout *buttonLayout = new QHBoxLayout;
    buttonLayout->addStretch(1);
    buttonLayout->addWidget(quitButton);
    buttonLayout->addStretch(1);
    QVBoxLayout *mainLayout = new QVBoxLayout;
    mainLayout->addWidget(statusLabel);
    mainLayout->addLayout(buttonLayout);
    setLayout(mainLayout);
    setWindowTitle("Servidor TCP");
}
```

Aparte de inicializar la GUI y de buscar la IP local como el cliente, solamente inicia la escucha de clientes con listen() y conecta la señal newConnection() con el slot que va a enviar las frases:

```
void Server::sendFortune()
{
    QByteArray block;
    QDataStream out(&block, QIODevice::WriteOnly);
    out.setVersion(QDataStream::Qt_4_6);

    out << (quint16)0;
    out << fortunes.at(qrand() % fortunes.size());
    out.device()->seek(0);
    out << (quint16)(block.size() - sizeof(quint16));

    QTcpSocket *clientConnection = tcpServer->nextPendingConnection();
    connect(clientConnection, SIGNAL(disconnected()),
            clientConnection, SLOT(deleteLater()));

    clientConnection->write(block);
    clientConnection->disconnectFromHost();
}
```

Una vez iniciado el DataStream como un ByteArray, primero ponemos en el buffer del mismo la cabecera a 0, luego mediante una función random, se elije una frase al azar, y se envía al buffer, luego rebobinamos a la posición 0 "**`out.device()->seek(0)`**" , le ponemos el tamaño de sólo la frase (por eso restamos al tamaño del bloque, el tamaño de la cabecera – 2 bytes).

Llamando a nextPendingConnection(), se devuelve el socket que ha sido conectado. Conectamos la señal de disconnected al slot QObject::deleteLater(), que hace que si hay desconexión del socket, el objeto que representa dicha conexión sea borrado por Qt.

Ya podemos enviar el bloque completo, por dicho socket conectado, y una vez hecho, ya podemos desconectar al cliente, y se borrará el objeto del socket como acabamos de ver.

Emisor UDP (QUdpSocket)

El emisor de datagramas UDP va a usar un QTimer para cada segundo enviar paquetes por un puerto. Veamos la implementación completa:

```
Sender::Sender(QWidget *parent)
    : QDialog(parent)
{
    statusLabel = new QLabel("Preparado para enviar datagramas por el puerto 45454");
    startButton = new QPushButton("Enviar");
    quitButton = new QPushButton("Salir");
    buttonBox = new QDialogButtonBox;
    buttonBox->addButton(startButton, QDialogButtonBox::ActionRole);
    buttonBox->addButton(quitButton, QDialogButtonBox::RejectRole);

    timer = new QTimer(this);
    udpSocket = new QUdpSocket(this);
    messageNo = 1;

    connect(startButton, SIGNAL(clicked()), this, SLOT(startBroadcasting()));
    connect(quitButton, SIGNAL(clicked()), this, SLOT(close()));
    connect(timer, SIGNAL(timeout()), this, SLOT(broadcastDatagram()));

    QVBoxLayout *mainLayout = new QVBoxLayout;
    mainLayout->addWidget(statusLabel);
    mainLayout->addWidget(buttonBox);
    setLayout(mainLayout);
    setWindowTitle("Emisor UDP");
}



void Sender::startBroadcasting()
{
    startButton->setEnabled(false);
    timer->start(1000);
}
```

```
void Sender::broadcastDatagram()
{
    statusLabel->setText(QString("Enviando datagrama %1").arg(messageNo));
    QByteArray datagram = "Mensaje enviado " + QByteArray::number(messageNo);
    udpSocket->writeDatagram(datagram.data(), datagram.size(),
                             QHostAddress::Broadcast, 45454);
    ++messageNo;
}
```

Iniciamos el timer con start(1000) una vez apretamos el botón de Enviar, como tenemos conectada la señal del timer, timeout() con broadcastDatagram, ésta se encarga de escribir el datagrama a enviar con writeDatagram(datos, tamaño_datos, IP_receptor, puerto). Muy sencillo como vemos.

Receptor UDP (QUdpSocket)

El receptor, meramente vincula el puerto y la dirección IP del interfaz de red por donde va a escuchar con bind(). Aquí de nuevo conectamos QIODevice::readyRead(), que nos indica que hay nuevos datos esperando ser leídos, con nuestro slot de lectura processPendingDatagrams.

Esto lo hace mediante un bucle usando hasPendingDatagrams() para saber si aún queda algo por procesar. La funcion pendingDatagramSize() nos devuelve los bytes que quedan pendientes de lectura. Con readDatagram(datos, tamaño) leemos los datos que se pondrán en QByteArray::data().

Veamos la implementación:

```
Receiver::Receiver(QWidget *parent)
    : QDialog(parent)
{
    statusLabel = new QLabel("Escuchando mensajes enviados");
    quitButton = new QPushButton("Salir");

    udpSocket = new QUdpSocket(this);
    udpSocket->bind(45454, QUdpSocket::ShareAddress);

    connect(udpSocket, SIGNAL(readyRead()),this, SLOT(processPendingDatagrams()));
    connect(quitButton, SIGNAL(clicked()), this, SLOT(close()));

    QHBoxLayout *buttonLayout = new QHBoxLayout;
    buttonLayout->addStretch(1);
    buttonLayout->addWidget(quitButton);
    buttonLayout->addStretch(1);
    QVBoxLayout *mainLayout = new QVBoxLayout;
    mainLayout->addWidget(statusLabel);
    mainLayout->addLayout(buttonLayout);
    setLayout(mainLayout);
    setWindowTitle("Receptor UDP");
}

void Receiver::processPendingDatagrams()
{
    while (udpSocket->hasPendingDatagrams()) {
        QByteArray datagram;
        datagram.resize(udpSocket->pendingDatagramSize());
        udpSocket->readDatagram(datagram.data(), datagram.size());
        statusLabel->setText(QString("Datagrama
recibido:\"%1\"").arg(datagram.data()));
    }
}
```

Con la ayuda del tema siguiente, que trata de la programación concurrente o multi-hilo, vamos a implementar el ejemplo del servidor TCP, pero esta vez con múltiples hilos para poder manejar a muchos clientes a la vez.

TEMA 24

PROGRAMACIÓN CONCURRENTE

Una aplicación normalmente usa un solo hilo de ejecución, donde lleva a cabo una sola tarea a la vez. Pero existe la posibilidad de que la tarea principal, genere nuevos hilos de ejecución independientes, y que estos se comuniquen con la tarea principal, o que trabajen juntos en una misma cosa, teniendo que sincronizar su trabajo. No es nuestra idea, explicar el trasfondo de la programación multi-hilo en este libro, ya que eso pertenece a un bagaje que se supone ya adquirido, lo que sí vamos es a explicar de qué manera Qt implementa las aplicaciones más típicas multi-hilo.

QThread

Esta es la clase que provee de hilos de manera multiplataforma. Cada objeto de esta clase representa un hilo de ejecución diferente a los demás y diferente al principal que se inicia y se termina con main(). Un hilo comparte datos con otros hilos, pero se ejecuta de manera independiente de los demás. En vez de comenzar en main() como el hilo principal, los hilos QThreads comienzan y terminan en su función run(). Así que run() ejecutaría un hilo e iniciaría su propio bucle de eventos con exec(), como hace main().

Para crear tus propios hilos, sólo hay que derivar una clase de QThread y reimplementar la función run(), con las tareas que este hilo vaya a ejecutar.

```
class MyThread : public QThread
 {
 public:
     void run();
 };

void MyThread::run()
 {
     QTcpSocket socket;
     // conecta las señales de QTcpSocket
     ...
     socket.connectToHost(hostName, portNumber);
     exec();
 }
```

Esto sería un hilo que conecta un socket con un host y luego inicia su propio bucle de eventos. Para comenzar la ejecución de un hilo, una vez se ha instanciado en el hilo principal, es usando la función start(). Recuerda instanciar cada hilo, poniendo como argumento padre al hilo del que nace, de esa manera, cuando el padre finalice, todos los hilos creados de él se borrarán también, evitando memory leaks. La ejecución de un hilo acaba cuando se sale de run(), igual que un programa cuando termina main().

QThread notifica con señales cuando un hilo, comienza (started()) o finaliza (finished()). También se pueden usar los getters isFinished() o isRunning() para saber si ha finalizado o aún esta corriendo.

No se pueden usar widgets en un hilo. Sí se pueden usar QTimers y sockets, y se pueden conectar señales y slots entre diferentes hilos.

Para ver un uso real de esta concurrencia de hilos, vamos a ver como quedaría el código del servidor TCP del tema anterior, de manera que pueda admitir múltiples conexiones de clientes, cada una en un hilo independiente.

Como los widgets, hemos dicho que no pueden correr en un hilo, han de correr en la tarea principal que inicia main(), ha de separarse el código que genera toda la interfaz gráfica y que inicia la escucha con listen().

```
FortuneThread::FortuneThread(int socketDescriptor, const QString &fortune, QObject
*parent)
    : QThread(parent), socketDescriptor(socketDescriptor), text(fortune)
{
}

void FortuneThread::run()
{
    QTcpSocket tcpSocket;
    if (!tcpSocket.setSocketDescriptor(socketDescriptor)) {
        emit error(tcpSocket.error());
        return;
    }

    QByteArray block;
    QDataStream out(&block, QIODevice::WriteOnly);
    out.setVersion(QDataStream::Qt_4_0);
    out << (quint16)0;
    out << text;
    out.device()->seek(0);
    out << (quint16)(block.size() - sizeof(quint16));

    tcpSocket.write(block);
    tcpSocket.disconnectFromHost();
    tcpSocket.waitForDisconnected();
}
```

Ahora, hemos creado una clase hilo del servidor que envía las frases y en run() es donde se encarga de hacer todo lo que hace, desde que es despertado para conectarse con el cliente, hasta que envía la frase, se desconecta y espera la desconexión (modo bloqueo). Creamos un QTcpSocket y lo asociamos al descriptor de socket que nos va a pasar el servidor (socketDescriptor), que identifica a cual de los clientes nos vamos a conectar. Bien, esto es lo que hace el hilo que despacha a un cliente.

```
FortuneServer::FortuneServer(QObject *parent)
    : QTcpServer(parent)
{
fortunes << tr("Has llevado una vida de perros. Bajate del sofa.")
             << tr("Tienes que pensar en el ma_ana.")
             << tr("Te ha sorprendido una sonido fuerte.")
             << tr("Tendr_s hambre dentro de una hora.")
             << tr("Tienes nuevo correo.");
}

void FortuneServer::incomingConnection(int socketDescriptor)
{
    QString fortune = fortunes.at(qrand() % fortunes.size());

    FortuneThread *thread = new FortuneThread(socketDescriptor, fortune, this);
    connect(thread, SIGNAL(finished()), thread, SLOT(deleteLater()));
    thread->start();
}
```

Y aquí está la parte principal del servidor, y esta vez se deriva de QTcpServer, así el código que pondríamos en el hilo principal, crearía un servidor FortuneServer al que ponemos a escuchar con listen(), cuando un cliente conectara su socket con él, automáticamente se enviaría la ejecución a incommingConnection, que pasa como parámetro el entero que representa el descriptor del socket conectado. Y en esta función creamos un hilo de ejecución para despachar al cliente, como hemos visto arriba, que pasaría el descriptor del socket TCP para que no se confunda de cliente, y la frase a enviarle. También conectamos la señal finished() de dicho hilo con su borrado, para que cuando acabe su ejecución sea borrado, y finalmente iniciamos dicho hilo con start().

Sincronización de hilos

Ya hemos visto el ejemplo más sencillo de hilos, donde un programa principal, genera hilos que hacen cosas diferentes, y donde no hay interacción entre los hilos. Pero, hay casos donde varios hilos pueden compartir datos o código de manera que pueda esto afectar al resultado.

Vamos a poner 2 supuestos resueltos de manera diferente. Supongamos que el código de varios hilos accede a la vez al código que aumenta un contador común. Podría ocurrir que un hilo acceda al valor antes de incrementarlo, mientras otro hilo ya lo esta incrementando, de manera que ambos dejarían el mismo resultado en el contador, cuando debieran contarse el paso de los 2 hilos de manera independiente. La única forma de resolver esto, es usar un Mutex, es decir una zona bloqueada, a donde sólo un hilo puede entrar a la vez, de esta manera el hilo que entra, bloquea esa zona, sólo él recoge el valor del contador ahora, y lo aumenta, luego sale de la zona, permitiendo que otro hilo entre, y así, se evitaría el problema inicial.

En Qt esto es hecho con la clase QMutex, que se declararía como privada dentro del QThread, y llamando a la función lock() se produce el bloqueo, y llamando a la función unlock() el desbloqueo de la zona.

```
void Thread::run()
{
    mutex.lock();
    counter++;
    mutex.unlock();
}
```

Para más comodidad se creo QMutexLocker(QMutex *) que el objeto que crea, al nacer bloquea el mutex, y al borrarse, desbloquea el mutex. De esta manera el mismo código quedaría así de simple:

```
void Thread::run()
{
    QMutexLocker locker(&mutex);
    counter++;
}
```

El segundo supuesto, se trata del típico problema del productor-consumidor, donde 2 hilos comparten un mismo buffer de memoria, donde un escribe y el otro lee de la siguiente manera. El hilo productor, escribe en dicho buffer hasta que llega a su final, y una vez allí vuelve de nuevo al principio para seguir escribiendo. El hilo consumidor, lee lo que el otro escribe conforme es producido y lo saca por la salida estándar de error (cerr).

Si usáramos solamente un mutex para bloquear el acceso simultáneo al buffer, bajaría la productividad del mismo, ya que no permitiríamos acceder al buffer a uno de los hilos, mientras el otro, está trabajando en él, cuando realmente no habría problema alguno, si cada uno trabajara en zonas diferentes en todo momento.

Para resolverlo vamos a usar 2 QWaitCondition y 1 QMutex. Veamos antes qué es QWaitCondition.

QWaitCondition, provee una variable condición para sincronizar hilos. Los hilos se quedan en espera cuando llegan a la función wait(&mutex). Wait lo que hace es que un mutex previamente bloqueado, es desbloqueado y el hilo, se queda en espera, hasta que otro hilo lo despierte, cuando es despertado, el mutex vuelve a bloquearse y el hilo sigue la ejecución después de wait(). Para despertar a los hilos que esperan, un hilo externo (por ejemplo, el principal), lo hace usando wakeAll() para despertarlos a todos o wakeOne() para despertar sólo a uno de ellos, el orden en que esto ocurre es aleatorio y no se puede controlar.

Vamos entonces ahora a resolver el problema del productor y consumidor, para ello vamos a definir las siguientes variables globales que van a compartir todos los hilos, incluido el principal:

```
const int DataSize = 100000;
const int BufferSize = 8192;
char buffer[BufferSize];

QWaitCondition bufferNotEmpty;
QWaitCondition bufferNotFull;
QMutex mutex;
int numUsedBytes = 0;
```

Datasize en la cantidad de bytes que va a escribir el consumidor en buffer. El tamaño del buffer es inferior a esa cantidad, por lo que el consumidor tendrá que dar varias vueltas, lo que hace con un código como éste:

```
buffer[i % BufferSize] = "ACGT"[(int)qrand() % 4];
```

Lo que escribe son aleatoriamente las letras ACGT (bases del código genético). De esta manera no se sale del buffer y lo escribe cíclicamente.

La condición de espera bufferNotEmpty, será señalada por el productor cuando haya escrito algún dato, despertando al consumidor para que lo lea. La condición de espera bufferNotFull, la señalará el consumidor cuando haya leído algún dato, despertando al consumidor para que siga escribiendo. numUsedBytes, va a ser el número de bytes en el buffer que contiene datos escritos por el productor, y que no han sido leídos por el consumidor. El mutex, se encargará de bloquear el acceso a la variable común numUsedBytes, para que sea leída y cambiada por un solo hilo en cada momento. numUsedBytes no podrá exceder el tamaño del buffer nunca, o el consumidor estará sobrescribiendo datos aún no leídos por el consumidor.

Por lo tanto el código que gobierna el hilo productor es:

```
class Producer : public QThread
{
public:
    void run();
};
void Producer::run()
{
    qsrand(QTime(0,0,0).secsTo(QTime::currentTime()));

    for (int i = 0; i < DataSize; ++i) {
        mutex.lock();
        if (numUsedBytes == BufferSize)
            bufferNotFull.wait(&mutex);
        mutex.unlock();

        buffer[i % BufferSize] = "ACGT"[(int)qrand() % 4];

        mutex.lock();
        ++numUsedBytes;
        bufferNotEmpty.wakeAll();
        mutex.unlock();
    }
}
```

Inicializa la semilla random con el timer. Antes de acceder a numUsedBytes bloquea con el mutex su acceso, si se ha llenado el buffer completo de datos, entonces se queda en espera de la condición bufferNotFull y libera el mutex. Si no ha llenado el buffer, libera el mutex, escribe en la posición siguiente y de nuevo antes de incrementar numUsedBytes (el contador común del buffer), bloque esta sección, incrementa su valor, y despierta al hilo consumidor, luego libera el mutex, y sigue el bucle a su ritmo.

El código del hilo consumidor sería complementario a éste:

```
class Consumer : public QThread
{
public:
    void run();
};

void Consumer::run()
{
    for (int i = 0; i < DataSize; ++i) {
        mutex.lock();
        if (numUsedBytes == 0) bufferNotEmpty.wait(&mutex);
        mutex.unlock();

        fprintf(stderr, "%c", buffer[i % BufferSize]);

        mutex.lock();
        --numUsedBytes;
        bufferNotFull.wakeAll();
        mutex.unlock();
    }
    fprintf(stderr, "\n");
}
```

El hilo consumidor también tiene su bucle para leer todos los datos que el consumidor va dejando en el buffer, y sacándolos por la salida estándar de error. Antes de revisar el valor de numUsedBytes, bloquea la sección con un mutex, si no hay datos que leer nuevos, entonces se queda esperando que se cumpla la condición de espera bufferNotEmpty y libera el mutex. Si no es así, también libera el mutex, lee el siguiente dato del buffer, y luego bloquea de nuevo con el mutex el cambio de numUsedBytes, que es decrementado, indicando que hay un dato menos por leer, luego despierta al hilo productor con su señal bufferNotFull, y libera el mutex. Y sigue en su bucle también a su ritmo.

De esta manera, ambos hilos se autosincronizan, sin pisarse el uno al otro. Sin embargo, existe una segunda forma de hacer esto de manera también óptima, usando semáforos. En Qt esto se hace con la clase QSemaphore. Por tanto veamos antes como funciona esta clase.

QSemaphore, provee de un contador general con semáforo. Mientras un mutex, sólo puede ser bloqueado una vez, hasta que es desbloqueado, un semáforo puede ser bloqueado múltiples veces. Para ello el semáforo inicia el contador con un número que define los recursos disponibles, y cada vez que se llama al semáforo con la función acquire() se piden recursos, si lo hay, el hilo sigue la ejecución, pero si no los hay para adquirirlos, entonces se bloquea la ejecución del hilo en ese punto hasta que hayan recursos suficientes para ser adquiridos, por lo que realmente funciona como un mutex especial. Veamos un ejemplo sencillo:

```
QSemaphore bytes(6);
    bytes.acquire(4);
    bytes.acquire(3);
```

Aquí, se crea un semáforo bytes con el contador a 6, cuando el hilo llega a la primera adquisición de 4 bytes, no hay problema ya que aún quedan 2, por lo que la ejecución sigue, pero en la siguiente al intentar adquirir 3, ya no puede, así que el hilo se quedaría ahí, hasta que haya bytes disponibles para seguir. Para devolver los bytes habría que llamar a la función release(int n). Por defecto si no se pone un número en acquire o en release, se presupone que es 1.

De esta manera podemos crear 2 semáforos en nuestro caso, con los 2 casos extremos que pararían la ejecución de alguno de los hilos. El semáforo freeBytes por ejemplo podría indicar el número de bytes libres para ser escritos, y que el productor podría adquirir antes de ver si puede seguir escribiendo, y el consumidor podría liberar con release(), una vez lo haya leído. Lo inicializaríamos con un valor igual al número total de bytes del buffer. Por otro lado otro semáforo llamado usedBytes podría indicar el número de bytes disponibles para lectura, el hilo consumidor adquiriría antes uno, para poder leer, y si no lo hay, entonces quedaría bloqueado, hasta que lo hubiera, y el productor, liberaría bytes en este semáforo una vez haya escrito en el buffer. Dicho semáforo se inicializaría con el valor a 0, o lo que es lo mismo instanciado por

defecto. De esta manera, cada hilo tiene su propio semáforo, y ambos hilos indican al otro las nuevas situaciones, liberando del semáforo contrario, y adquiriendo del suyo propio. Así verás que el código queda mucho más sencillo que usando el mutex y las 2 condiciones de espera.

```
const int DataSize = 100000;
const int BufferSize = 8192;
char buffer[BufferSize];

QSemaphore freeBytes(BufferSize);
QSemaphore usedBytes;

class Producer : public QThread
{
public:
    void run();
};

void Producer::run()
{
    qsrand(QTime(0,0,0).secsTo(QTime::currentTime()));
    for (int i = 0; i < DataSize; ++i) {
        freeBytes.acquire();
        buffer[i % BufferSize] = "ACGT"[(int)qrand() % 4];
        usedBytes.release();
    }
}

class Consumer : public QThread
{
public:
    void run();
};

void Consumer::run()
{
    for (int i = 0; i < DataSize; ++i) {
        usedBytes.acquire();
        fprintf(stderr, "%c", buffer[i % BufferSize]);
        freeBytes.release();
    }
    fprintf(stderr, "\n");
}
```

Comprendido el mecanismo que hemos explicado de los semáforos, su iniciación, adquisición y liberación, no hay mucho que comentar de este código.

Bueno y hemos dejado para el final, el hilo principal, que en ambos casos sería el mismo. Desde main(), se iniciarían ambos hilos con start(), y luego se llamaría en ambos a wait(), para que no acabe la función main() y mueran los hilos antes de que terminen su trabajo. Lo que hace esa función (wait) es esperar a que dicho hilo acabe, y una vez acaba, devuelve ***true***.

```
int main(int argc, char *argv[])
{
    QCoreApplication app(argc, argv);
    Producer producer;
    Consumer consumer;
    producer.start();
    consumer.start();
    producer.wait();
    consumer.wait();
    return 0;
}
```

Ambos wait() se ejecutan uno detrás del otro, ya que son hilos diferentes, y se espera a que estos acaben, para terminar la ejecución de main().

TEMA 25

GRÁFICOS E IMPRESIÓN

Los gráficos 2D en Qt están basados en QPainter, que puede dibujar desde formas geométricas como, puntos, lineas, rectángulos, etc, hasta mapas de bits, imágenes y textos. Puede aplicar efectos como antialiasing, alpha-blending, rellenado con gradientes, etc. Puede aplicar transformaciones y más cosas. Puede hacer todo esto sobre un QWidget, un QPixmap o un QImage. Puede ser usado en conjunción con QPrinter para imprimir o para generar PDFs, así que el mismo código que usamos para dibujar nos puede permitir imprimir. Una alternativa a QPainter es el módulo QtOpenGL, que accede a la librería 2D y 3D OpenGL.

La forma más típica de usar QPainter en un Widget es por ejemplo redefiniendole la función paintEvent() que se encarga de dibujarlo. Esto es un ejemplo sencillo:

```
void SimpleExampleWidget::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    painter.setPen(Qt::blue);
    painter.setFont(QFont("Arial", 30));
    painter.drawText(rect(), Qt::AlignCenter, "Qt");
}
```

Todas las funciones draw…(), son las encargadas de dibujar con QPainter. Previamente a dibujar hay que definir las características del pintado con las funciones setter setPen(), setBrush() y setFont(), que definen el lápiz de trazado de dibujo, el rodillo de rellenado de formas, y la fuente para el texto, respectivamente. Esas herramientas pueden ser cambiadas en cualquier momento.

Las funciones de dibujado de formas más importantes son: drawPoint(), drawLine(), drawPolyline(), drawPoints(), drawLines(), drawPolygon, drawRect(), drawRoundRect(), drawEllipse(), drawArc(), drawChord(), drawPie(), drawText(), drawPixmap() y drawPath().

QPainter sitúa el origen de coordenadas por defecto en la esquina superior izquierda (0,0). Sin embargo sería más cómodo tener una ventana con coordenadas lógicas y que el sistema las pasara a físicas sin que el programador tenga que hacer las conversión. Pues bién, llamamos viewport al rectángulo con coordenadas físicas, y Windows al rectángulo con coordenadas lógicas. Esto se hace con QPainter::setWindow().

Luego esta la matriz mundo, que es una matriz para aplicar transformaciones afines como rotaciones y translaciones. QMatrix es quien implementa dicha matriz, y usando sus funciones rotate() y translate() se llevan a cabo las transformaciones, y finalmente se aplican al QPainter mediante su función setMatrix(QMatrix).

Si requerimos de más calidad, aunque esto redunda en menos velocidad, podemos usar QImage para dibujar. Para ello creamos un objeto QImage() con un tamaño y formato de color, y luego instanciamos con QPainter(QImage *), para ir dibujando con él, sería su dispositivo de pintado.

QPrinter

Imprimir en Qt es similar a dibujar con QPainter y se trata de seguir los siguientes pasos:

1.- Crear QPrinter para que sirva como el dispositivo de pintado (painting device).

2.- Abrir una ventana QPrintDialog, que permita al usuario ajustar la impresora y demás ajustes.

3.- Crear un QPainter que opere con QPrinter.

4.- Pintar una página usando QPainter.

5.- Llamar a QPrinter::newPage() para avanzar a la página siguiente.

6.- Repetir los pasos 4 y 5 hasta que se imprima todo.

Se puede también imprimir a un PDF llamando a setOutputFormat(QPrinter::PdfFormat).

```
QPrinter printer(QPrinter::HighResolution);
QPrintDialog *dialog = new QPrintDialog(&printer, this);
dialog->setWindowTitle("Imprimir Documento");
if (dialog->exec() != QDialog::Accepted)
    return;

QPainter painter;
painter.begin(&printer);

for (int page = 0; page < numberOfPages; ++page) {
    // Usar painter para imprimir la pagina.

    if (page != lastPage)
        printer.newPage();
}

painter.end();
```

El tamaño de la zona imprimible del papel viene determinado por QPrinter:pageRect(), que devuelve el QRect de la zona imprimible, con ello se pueden hacer los cálculos para determinar la variable numberOfPages y lastPage.

TEMA 26

RECURSOS EXTERNOS

Acabando ya los temas que resumen todo lo que es necesario saber para desarrollar una aplicación de propósito general, hemos querido meter en un mismo tema con el nombre de recursos externos, una serie de prácticas que nos permiten hacer colaborar a Qt con su exterior. Comenzando por la posibilidad de cargar en tiempo de ejecución un diseño de ventana de diálogo en formato *.ui con las uitools, pasando por la posibilidad de ejecutar procesos o aplicaciones externas mediante la clase QProcess, y acabando por la posibilidad de añadir la capacidad de uso de librerías externas a Qt.

Diálogos dinámicos (QUiLoader)

Llamamos diálogos dinámicos a aquellos que han sido creados en el Qt Designer como ficheros *.ui, pero que no han sido compilados junto con la aplicación, por lo que el uic (ui compiler), no ha podido pasarlos a C++, y luego ser compilados, si no que van a ser cargados en tiempo de ejecución por la aplicación principal como .ui tal cual. Para ello existe una clase capaz de tal manejo, que es QUILoader, que mediante su función load(QFile), puede cargar dicho fichero con toda su interfaz.

```
QUiLoader uiLoader;
QFile file("dialog.ui");

QWidget *dialog = uiLoader.load(&file);

if(dialog){
    // código para recoger los widgets
    QComboBox *combo1 = dialog->findChild<QComboBox *>("combo1");
    // si combo1 no existe devuelve un puntero nulo
    if (combo1).........

}
```

Date cuenta que aquí cargamos el fichero .ui con el UiLoader, y recuperamos su estructura en un puntero QWidget (dialog). Usando de la introspección de Qt, podemos mediante findChild, ir recuperando punteros a sus componentes, conocidos sus nombres (objectName). Podemos de esta manera ya usar dialog->show() para mostrarlo de manera no-modal o dialog->exec() para hacerlo de manera modal.

Para usar QUiLoader tenemos que añadir a *.pro el módulo uitools de la siguiente manera:

```
CONFIG          += uitools
```

Si vas a usar el compilar MSVC 6, es necesario que en vez de usar findChild<T>() de QWidget, uses qFindChild<T> de QtGlobal, ya que el primero no es soportado por dicho compilador.

Programas externos (QProcess)

Una de las utilidades de desarrollar una GUI, es facilitar el manejo de una aplicación de consola que previamente existía, y cuya línea de comandos era muy larga e incómoda de usar. Nos podemos plantear el construirle una GUI para poder usarla de una manera más gráfica y cómoda por lo tanto. Qt, nos permite eso mediante la clase QProcess.

Para simplemente ejecutar una aplicación externa, usaremos la función execute() de manera similar a ésta:

```
QString cmd = "cp a.txt b.txt";
QProcess::execute(cmd);
```

Si lo que queremos es establecer una comunicación entre el programa externo y la aplicación, si aquél lo permite, en ese caso usaremos la función start(), y usaremos write() para enviarle datos, y readAll() para leer del mismo. Usaremos close() para cerrar el canal de comunicación. Un ejemplo de este tipo de comunicación es posible con programas como el bash de Linux, o Telnet, o cmd.exe de Windows. Veamos un ejemplo:

```
QProcess bash;
bash.start("bash");
if(!bash.waitForStarted()){
    qDebug() << "NO RULA";
    return -1;
}
bash.write("ls");
bash.closeWriteChannel();
if(!bash.waitForFinished()){
    qDebug() << "NO RULA";
    return -1;
}
QByteArray response = bash.readAll();
qDebug() << response.data();
bash.close();
```

Uso de librerías externas

Cuando queremos usar librerías para linkarlas nuestro programa, simplemente hay que añadir a nuestro fichero *.pro una línea como esta:

LIBS += -lnombre_libreria

Siembre que compilemos con el GNU Compiler. Así la librería matemática sería –lm, la librería curl sería –lcurl, y así sucesivamente.

Pero hay veces que querremos llamar en tiempo de ejecución a funciones de una librería dinámica, que en Windows tiene extensión .dll, en MacOSX suele ser .dylib, y en Linux es .so . Para tal caso, Qt provee de un sistema multiplataforma a través de la clase QLibrary.

Toda librería tiene sus funciones exportadas como símbolos. Supongamos que una librería llamada “mylib” con la extensión correspondiente del operativo donde esté instalada, tiene una función que en C tiene como prototipo “int avg(int, int)” que lo que hace es realizar la media aritmética de los 2 parámetros que se le pasan, y devuelve esa media en formato entero. Si queremos llamar a dicha función desde Qt, habría que usar el siguiente código:

```
QLibrary library("mylib");
typedef int (*AvgFunction)(int, int);

AvgFunction avg = (AvgFunction) library->resolve("avg");
if (avg)
    return avg(5, 8);
else
    return -1;
```

TEMA 27

BÚSCATE LA VIDA

Sí, has leído bien, este capítulo se llama "búscate la vida", esto no quiere decir que ya me haya cansado de escribir este libro, ni de explicar nuevas cosas. Qt es un Framework que ya tiene casi la veintena de años, por lo que a día de hoy ha crecido enormemente, tanto que ningún libro que lo trate puede aspirar a tocar absolutamente todas sus clases. También es cierto que hemos tocado las clases que más vas a usar siempre que desarrolles aplicaciones de propósito general, pero es posible que alguna vez te plantees hacer otra cosa, y esto no esté cubierto en ningún libro. Hay que pensar que Qt, seguirá creciendo en el futuro, por lo que la única forma de mantenerse al día, es usando el Qt Assistant, es decir, buscándose uno mismo la vida.

Por tanto, lo que pretendemos con este capítulo es cubrir esa parte que viene después de cuando acabes este libro, y tengas que empezar a escribir código, y en algunas ocasiones te tropieces con que tienes que usar cosas, que no has estudiado antes, y nunca las has usado, e incluso que no puedas encontrar en los foros a nadie que lo haya usado antes. Qt está muy bien documentado, pero en inglés, por lo que es fundamental entender al menos el idioma de Shakespeare a nivel de lectura técnica.

Pues bien, para practicar, se me ha ocurrido plantear unas clases que no hemos visto, y que curiosamente eches de menos, si has programado en otros entornos. Se trata de la hora, la fecha e incluso del uso de timers, aunque en el tema 23 hemos visto un ejemplo de éste último. Por ello vamos a practicar en este tema a buscarnos la vida en el asistente. Realmente podría haber centrado la búsqueda en cualquier otro tema más complejo, pero simplemente lo sería por tener más funciones, propiedades, señales y slots, o por que el texto de la descripción es mucho más extensa. Así que si seguimos bien estas 3 clases, no habría por qué haber ningún problema en cualquier otra.

Lo primero sería saber qué clases podrían encargarse de la hora, de la fecha, o de un proporcionar un timer. Debido a que un poco de inglés sí que sabemos, y más si hemos trabajado con librerías similares en C, C++ u otro lenguaje, sabemos que la hora es Time, que la fecha es Date y Timer está claro lo que es, así que no sería muy descabellado pensar que dichas clases se llamen QTime, QDate y QTimer. Así que nos vamos al asistente, y buscamos las 3 clases, veremos que hemos acertado en todo. ¡ Caramba, que suerte tenemos !. De todas formas si no tuviéramos claro el concepto de nuestra clase, siempre puede uno darse una paseo por todas, para detectar dentro de su nombre un atisbo de luz. Por ejemplo, ¿ de qué cree que tratará la clase QMimeData o QFtp ?.

QTime

Es evidente que el constructor de esta clase, debe de permitir iniciar una instancia con una hora determinada. Vemos entre las public functions, siempre en primer lugar los distintos constructores, uno sin parámetros, por lo que es una mera instanciación, y debe de existir por tanto una función que más tarde pueda poner la hora, que por su prototipado, debe de ser setHMS().

Otra de las cosas importantes que hay que mirar en una clase, es quedarse con las propiedades o variables privadas, ya que asociadas a ellas vamos a encontrar una serie de funciones setter y getter importantes para su manejo. Así nos encontramos con hour(), minute() y second() en los getter, y el ya mencionado setHMS() como el único setter.

Luego nos encontraremos con una serie de funciones que llevan a cabo una labor especial, que podemos leer en su descripción de qué se trata. Busca por tanto, la labor de start(), restart() y ellapsed(). Están muy relacionadas entre sí, y así aparece en sus descripciones. También es muy importante aquella que recoge la hora exacta actual, que de seguro es

currentTime(), entre los static public. Dentro de los operadores, vemos que se pueden comparar 2 objetos QTime entre sí, lo cual puede ser muy útil. Y aún más útil una serie de funciones que permiten añadir a la hora, intervalos de tiempo variables, en segundos (addSecs).

Te animo a que hagas pruebas con ellas en un código sencillo de consola.

QDate

Se trata de una clase de similar disposición a la anterior. Por ello te animamos a que la describas tú mismo, y luego compruebes su funcionalidad en un código también sencillo.

QTimer

Esta clase, era fundamental explicarla, ya que es muy útil siempre el uso de timers. El dejarla para un tema autosuficiente como éste, nos va a permitir ser más autónomos de aquí en adelante a la hora de manejar el Framework de Qt, porque otra posibilidad que nos puede ocurrir, es que se nos olvide la funcionalidad de algunas de las clases que ya hemos explicado, y no tengas cerca este libro para repasarlas, mientras que teniendo el asistente, siempre podrás sacar el trabajo adelante.

Lo primero que nos dice el asistente es que QTimer provee de timers repetitivos y de un solo disparo. Los timers repetitivos son aquellos que cada intervalo, se dispara el evento del timer, y los de un solo disparo, sólo se programar y se disparan una sola vez. Es de suponer que para indicar que ha llegado al final de su contador, dispare una señal. Y así es, de hecho sólo podemos ver la señal timeout().

Veamos sus propiedades, que son: active, que es un bool que estará a true cuando esté activo, y false cuando no lo esté; interval, que es un entero, que de seguro contiene el intervalo programado; y singleShot que es un bool que estará true si es de único disparo, o a false, si es repetitivo. En torno a estas propiedades, deben hacer getters y setters. Como setters tenemos setInterval() y setSingleShot(), y como getters tenemos, isActive() e isSingleShot().

Como public functions, tenemos las más importantes, que son start() para poner en marcha el timer, y stop() para pararlo en cualquier momento. Hay otra función llamada singleShot() que nos permite, programar el intervalo y el slot a ejecutar de un determinado objeto, cuando dicho intervalo haya pasado.

Puesto que QTimer provee de una señal, timeout() es también muy lógico el usarla para conectarla a slots de otros objetos que hagan algo una vez pasado el tiempo.

Te animamos a que escribas un programa sencillo, que cada segundo, aumente el valor de un contador en un QLCDNumber.

Quizás este tema te haya parecido muy trivial, si es así, es que ya estas preparado para nadar tú sólo en el universo Qt. Pero aún no te vayas, porque el último tema es el siguiente, y trata de algo que muy pocos suelen tratar, esto es, de cómo preparar un instalador para nuestra aplicación para los 3 operativos más importantes: Linux, MacOSX y Windows.

TEMA 28

INSTALADORES

Al final del tema 2, cuando preparamos el entorno de desarrollo, para poder desarrollar todos los ejemplos del libro sin problemas, avisábamos de que las aplicaciones desarrolladas por defecto con estas instalaciones por defecto del SDK de Qt, no podían ser implementadas tal cual en sistemas de producción donde el SDK no va a estar instalado.

Llegó el momento de tomar en consideración qué archivos debemos de llevarnos con nuestras aplicaciones fuera del ordenador donde se efectuó desarrollo, para que todo funcione correctamente.

El entorno de desarrollo instalado por defecto, funciona con librerías de enlace dinámico, por lo que vamos a compilar ejecutables que son poco pesados, pero que enlazan en tiempo de ejecución con una serie de librerías dinámicas, por lo que nos tendremos que llevar junto con el ejecutable, toda una serie de librerías y plugins, sobre la que el mismo se va a apoyar para poder funcionar. Por ello es muy interesante disponer, no sólo de un ordenador operativo para el desarrollo, si no también, disponer de otro (máquina real o virtual) donde esté instalado el operativo por defecto sin más, donde vamos a probar que entregamos todo lo necesario para que nuestra aplicación funcione. Una vez tengamos en una carpeta a parte, todos los elementos necesarios para que la aplicación funcione autónomamente, podremos recurrir a crear un sistema de instalación que se encargue de copiar dichos elementos en la manera debida al sistema en producción.

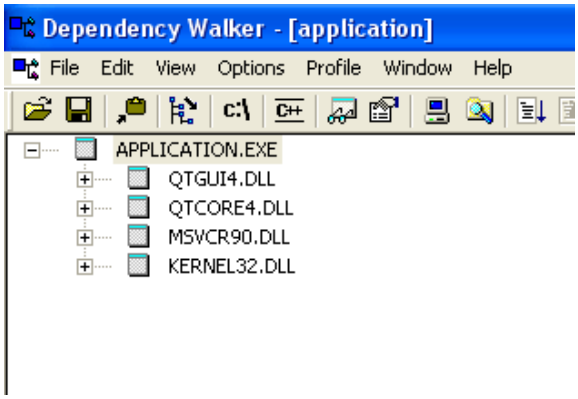
Vamos por tanto a seguir un ejemplo, en los 3 sistemas operativos de siempre (Windows, Linux y Mac OSX), y lo vamos a hacer con la aplicación que desarrollamos en el tema 20 que es un editor de textos donde están presentes los elementos típicos de una aplicación de propósito general. Vamos pues a usarla para crear el paquete de instalación correspondiente a cada uno de los sistemas operativos objetivo.

Windows

Si no lo has hecho anteriormente, abre el archivo de proyecto (.pro) en tu entorno de desarrollo (Qt Creator), y pulsando en el modo Projects (iconos de la izquierda), selecciona en "Edit Build configuration" la opción Release. Luego, vamos a reconstruir la aplicación con Rebuild Project. Podemos incluso ejecutarla. No debe de haber ningún problema, ya que quedó completamente creada y probada en el tema 20. Ya tenemos en el directorio donde guardemos nuestros fuentes, una carpeta que se llamará application-build-desktop, en cuyo interior encontraremos en la carpeta Release, el ejecutable final (application.exe). Si usas Visual Studio 2008 como compilador, y en el sistema está registrada la run-time de Visual C++ 9.0 "msvcp90.dll", podrás ejecutar sin problemas la aplicación. Si usas el compilador MinGW que va en el paquete de Qt4, las liberías dinámicas correspondientes no estarán registradas en el sistema, y la ejecución de la aplicación desde su carpeta, te dará error indicándote la primera librería dinámica que intenta enlazar pero que no encuentra.

Sea como fuere, debemos prepararle al ejecutable, todo lo que requiere en su propia carpeta, así que vamos a ello.

Para saber de qué depende nuestra aplicación, en Windows vamos a tener que usar una herramienta que no viene incluida en el SDK de Qt, llamada Dependency Walker. Puedes bajarla de http://www.dependencywalker.com/. En nuestro caso hemos bajado la versión de 32 bits (for x86). Al ejecutarla, vamos a File -> Open, y abrimos nuestro ejecutable (application.exe). Se nos despliega en la ventana superior izquierda todo un árbol de dependencias, pero lo que nos interesa, está en las ramas principales de dichos árboles. Si colapsamos dichas ramas, en un sistema con VS2008 tendríamos algo así:



Como podemos observar, nuestra aplicación compilada con VisualC++ 2008 (9.0) depende de 4 dlls, pero una de ellas, ya viene provista por el sistema operativo Windows (kernel32.dll), por lo que realmente vamos a requerir de las Dll's QtGui4.dll, QtCore4.dll y MSVCR90.dll .

De hecho estas 3 dependencias, las podíamos preveer, ya que si miramos el contenido del fichero de proyecto (application.pro) veremos que pide añadir los modulos "core" y "gui":

QT += core gui

Por lo que, es de suponer que requerirá ambas librerías dinámicas. La tercera dependencia es típica de quien usa el compilador VS2008. Quien use MinGW se encontrará con 2 dependencias, las de los ficheros mingwm10.dll y libgcc_s_dw2-1.dll .

¿Dónde podemos encontrar esas librerías?. Pues las librerías de Qt se encuentran en el directorio /bin de la instalación del Qt SDK. La librería de Visual C++ está en el directorio de instalación de VS2008 (\Archivos de Programas\Microsoft Visual Studio 9.0\VC\ce\dll), donde podrás elegir la arquitectura de CPU, en nuestro caso en la carpeta x86 (para PC).

Así que trasladamos esas 3 dependencias al mismo sitio donde esta nuestro ejecutable. Si llevamos esos 4 ficheros en una carpeta a cualquier PC con Windows XP, Vista o 7, verás que se puede ejecutar sin problema alguno.

Habrán aplicaciones que hacen uso de plugins, como puedan ser drivers de bases de datos, drivers de formatos gráficos, etc. Veremos luego en un apartado qué hacer en ese caso.

Linux

Si no lo has hecho anteriormente, abre el archivo de proyecto (.pro) en tu entorno de desarrollo (Qt Creator), y pulsando en el modo Projects (iconos de la izquierda), selecciona en "Edit Build configuration" la opción Release. Luego, vamos a reconstruir la aplicación con Rebuild Project. Podemos incluso ejecutarla. No debe de haber ningún problema, ya que quedó completamente creada y probada en el tema 20. La aplicación quedará en su directorio de manera que puede ejecutarse directamente desde él (./application). Para comprobar sus dependencias, hay una herramienta que ya viene con las herramientas GNU, que es ***ldd***.

ldd ./application

Nos responderá con una serie de librerías dinámicas del tipo .so . La gran mayoría de esas librerías suelen estar instaladas en un sistema limpio con gestor de ventanas Gnome o KDE. Pero es conveniente que antes te cerciores de ello. Y si es así, es posible que dicha librería se instale mediante algún paquete que se pueda bajar de los repositorios oficiales de la distribución de Linux en cuestión. Finalmente dichas librerías siempre acabarán en /lib, /usr/lib ó /usr/local/lib. De esta manera son siempre accesibles a la aplicación. Pero a parte de estas librerías veremos otras que no están en esos directorios que de seguro son:

libQtGui.so.4 y libQtCore.so.4 . Estas las vamos a tener que poner al alcance de nuestra aplicación en su propio directorio. Puesto que se tratan de links a ficheros de otro nombre lo hacemos de la manera:

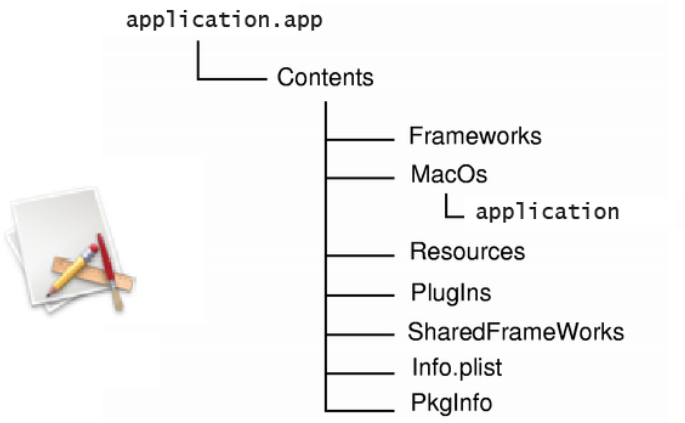
cp –R qtsdk/qt/lib/libQtGui.so.4* application_folder

cp –R qtsdk/qt/lib/libQtCore.so.4* application_folder

Habrán aplicaciones que hacen uso de plugins, como puedan ser drivers de bases de datos, drivers de formatos gráficos, etc. Veremos luego en un apartado qué hacer en ese caso.

Mac OSX

Una aplicación para Mac, lleva la extensión .app, y en sí mismo es una carpeta en cuyo interior lleva el verdadero ejecutable y todos los recursos, plugins, frameworks, etc, que necesita para funcionar. En realidad un .app es un bundle y tiene la forma que definimos en el gráfico siguiente:



Si no lo has hecho anteriormente, abre el archivo de proyecto (.pro) en tu entorno de desarrollo (Qt Creator), y pulsando en el modo Projects (iconos de la izquierda), selecciona en "Edit Build configuration" la opción Release. Luego, vamos a reconstruir la aplicación con Rebuild Project. Podemos incluso ejecutarla. No debe de haber ningún problema, ya que quedó completamente creada y probada en el tema 20. Inicialmente application.app contiene:

Info.plist MacOS PkgInfo Resources

Para ver las dependencias del ejecutable en sí (no del bundle), usamos una herramienta que lleva el XCode, que se llama ***otool***:

otool -L application.app/Contents/MacOs/application

En nuestro caso responde así en Mac OSX 10.6.4:

application.app/Contents/MacOs/application:

QtGui.framework/Versions/4/QtGui (compatibility version 4.7.0, current version 4.7.0)

QtCore.framework/Versions/4/QtCore (compatibility version 4.7.0, current version 4.7.0)

/usr/lib/libstdc++.6.dylib (compatibility version 7.0.0, current version 7.9.0)

/usr/lib/libgcc_s.1.dylib (compatibility version 1.0.0, current version 625.0.0)

/usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version 125.2.0)

Las 3 librerías inferiores (.dylib) son propias de todo sistema por defecto. Las 2 superiores, son librerías que se han instalado en forma de Framework, que es como se instalan por defecto las librerías del Qt4 SDK en Mac. De hecho puedes verlas a todas en /Library/Frameworks.

Cómo crear a mano el bundle con todo lo que necesita, sería largo y tedioso, Nokia nos ofrece una herramienta con el SDK que lo hace por nosotros de manera sencilla y cómoda. Se llama "macdeployqt", que pide los siguientes requisitos para la construcción del bundle final:

1.- Las versiones debug de los plugins no son añadidas al bundle.

2.- Los plugins designer no son añadidos al bundle.

3.- Los plugins para formato de imágenes sí que son añadidos al bundle.

4.- SQL drivers, Scripts plugins, Phonon back-end plugins, SVG Icon plugins, siempre son añadidos al bundle si la aplicación usa los módulos correspondientes QtSql, QtScript, Phonon y QtSvg.

5.- Si quieres añadir una librería no-Qt al bundle, tendrás que antes añadirla al proyecto .pro como una librería explícita. En tal caso, se añadiría sin problemas a bundle.

Bien, como en nuestro caso cumplimos sin problemas todas estas premisas, vamos a crear el bundle de una vez y por todas haciendo:

macdeployqt application.app

Tras lo cual, el contenido del bundle habrá cambiado a:

Frameworks Info.plist MacOS PkgInfo PlugIns Resources

Podemos ver que en Frameworks se han añadido los que hacían falta, y alguno más. Podemos borrar a mano los que sobran si queremos ahorrar espacio.

Ya podemos distribuir la aplicación sin problemas, y para instalarla en otro equipo el usuario solamente tendría que arrastrar el icono correspondiente al .app a la carpeta Aplicaciones.

Habrán aplicaciones que hagan uso de plugins, como puedan ser drivers de bases de datos, drivers de formatos gráficos, etc. Veremos en el siguiente apartado como hacer esto, aunque nuestra herramienta ya se haya encargado también de colocar los plugins que considera necesarios en sus respectivas carpetas dentro del bundle.

Plugins

Los plugins en Qt, permiten añadir nuevas funcionalidades a las aplicaciones. Así hay plugins que permiten añadir nuevos formatos de imágenes legibles en Qt, o nuevos tipos de bases de datos, etc. Los plugins incluídos en el SDK, están en el directorio /qt/plugins en su carpeta de instalación. Los nombres de esas carpetas son fijos, y si se trasladan tal cual a la carpeta del ejecutable de la aplicación, la misma no tendrá problemas en encontrar dicho plugin si lo requiere, ya que por defecto es ahí donde los buscará. Si queremos ponerlos en otro sitio diferente, entonces en la función main() de la aplicación tendremos que decirle donde encontrarla usando QCoreApplication::addLibraryPath() .

Los plugins para bases de datos que el SDK provee son muy limitados (ODBC y SQLite). Nosotros hemos añadido el de PostGresql y el de MySQL en nuestros ejemplos, para cualquiera de las 3 plataformas que hemos explicado. Puedes compilar tú mismo cualquier plugin bajando los fuentes del SDK (qt-everyone-src), y compilando lo que necesites conforme a los README que le acompañan. El último parágrafo de este tema explica el procesos de compilación del driver para MySQL.

Instaladores en Windows y Linux

Sabiendo ya el paquete que debemos copiar, podríamos nosotros mismos crear un instalador mediante una aplicación, script o similar. Vamos a proponerte una opción más sencilla para que puedas tener en cuenta a la hora de entregar tus aplicaciones. Y vamos a empezar por Windows y Linux. Aunque hay muchas aplicaciones que están especializadas en hacer instaladores de un paquete, nosotros hemos escogido una en particular desarrollada en Tk/Tcl que permite crear instaladores para Windows, Linux y otros UNIX. Su nombre es InstallJammer, que es gratuito, de código abierto y multiplataforma, como a nosotros nos gusta.

Crear un instalador con esta herramienta es muy sencillo e intuitivo a pesar de que esté en inglés. Vamos a seguir con nuestra aplicación para hacer el ejemplo de creación del instalador.

Al arrancar InstallJammer, pulsamos en New Project, y se nos abre un wizard para crear el proyecto del instalador. Rellenamos los datos que se nos van pidiendo. Contestadas todas las preguntas, pulsamos en Finish, y se nos abre el Install Designer con una serie de ítems que podemos ir cambiando a nuestro gusto, o dejando su valor por defecto. Dejamos al usuario el husmear por toda una ingente cantidad de parámetros. Vamos a construir con los valores por defecto para Windows o para Linux conforme necesitemos. Pulsamos sobre Build Install, y vemos en la ventana como se nos crea Application-1.0-Setup.exe (por ejemplo para Windows). Si pulsamos sobre su link, se nos abre la carpeta donde se ha creado el ejecutable instalador, listo para ser usado.

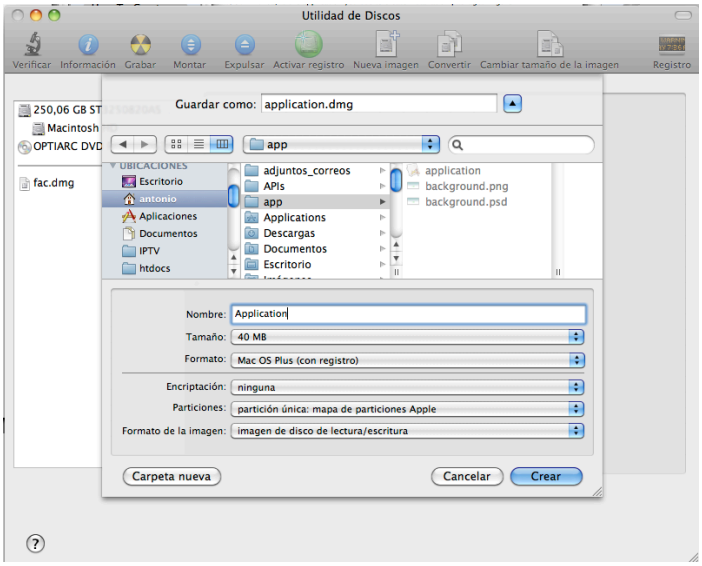
Así de sencillo es crear un instalador con este sencillo programa.

Instaladores en Mac OSX

Para crear instaladores en Mac, no hay ninguna aplicación simple y libre como la que acabamos de ver, pero sí podemos usar herramientas que vienen en nuestro sistema por defecto, una vez hemos instalado XCode, y que nos permitirán crear un instalador con aspecto profesional. Vamos a hacerlo de 2 maneras diferentes: una mediante una imagen .dmg donde se arrastra el .app sobre la carpeta Aplicación, y otra creando una paquete instalable mediante la herramienta PackageMaker de XCode.

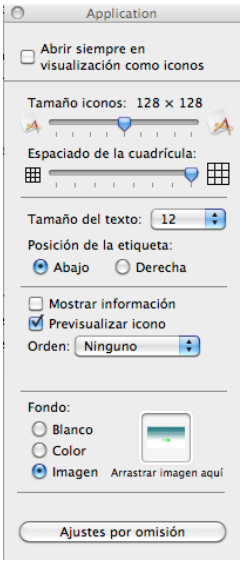
Seguimos usando nuestro ejemplo ya empaquetado en un bundle completo, tal como lo dejamos, preparado para llevar a otro equipo (application.app).

Vamos a crear una imagen DMG con la utilidad de discos. Pulsaremos sobre Nueva Imagen, y vamos a poner los ajustes de la imagen de a continuación:



La creamos del tamaño suficiente para que quepan .app y una fondo en PNG que crearemos de 700x400, similar a éste:

Montamos con un doble clic la imagen DMG vacía recién creada, y arrastramos a ella el bundle .app. Luego nos vamos al disco raiz, y creamos un alias de la carpeta Aplicación, la arrastramos al DMG y le cambiamos el nombre a Aplicación. Con el DMG montado y abierto en el finder, pulsamos Cmd+J. Se abren las opciones de carpeta, y hacemos algo como esto:

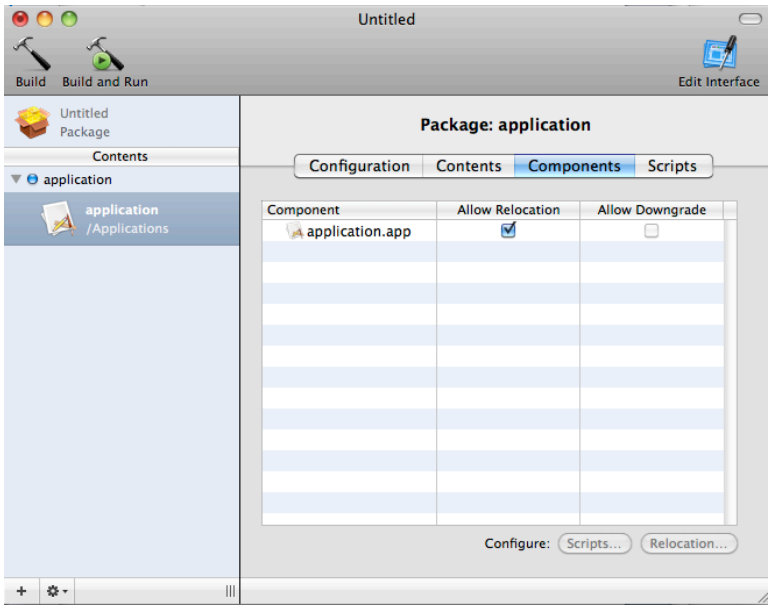


El fondo lo elegimos haciendo doble click en el cuadro de arrastrar la imagen, y seleccionamos el PNG. Ya se verá el DMG montado con la imagen de fondo, y podremos situar los iconos, y redimensionar la ventana para ajustarla al tamaño del fondo. Expulsamos el DMG montado, y nos vamos a la utilidad de discos, y eligiendo la imagen DMG creada, pulsamos en convertir, y la guardamos esta vez comprimida. Se crea el DMG final preparado para ser distribuido. Al montarlo quedará algo como esto:

Donde queda clara la intención de que se arrastre el icono de application.app sobre la carpeta Applicaciones, para que se produzca la instalación.

Pasemos a la segunda opción, que consiste en crear un paquete de instalación del tipo *.pkg con el PackageMaker del XCode. Dicha aplicación la puedes encontrar en /Developer/Applications/Utilities/PackageMaker. Al arrancarla, lo primero que nos pide es la organización (tipo com.nombre_organizacion), y el operativo objetivo mínimo. Pulsando sobre el

+ en la esquina inferior izquierda podemos añadir los componentes a instalar, que en nuestro caso es solamente uno (application.app).



Si navegas por esta herramienta, verás que también es muy intuitiva y potente, y está en inglés. En la zona de contenidos, se pueden poner varios elementos (aplicaciones, librerías, frameworks, ayudas, etc). En Configuration, podemos elegir qué vamos a instalar y donde, si vamos a permitir que el usuario reubique la instalación y si se va a requerir autentificación por parte del administrador para instalarla, de manera que podremos también copiar ficheros en zonas del sistema. En Contents podemos elegir el usuario y grupo al que pertenecerá este contenido y sus permisos. Si seleccionamos el paquete de distribución (encima de Contents), podremos editar las opciones del paquete completo, poniéndole un nombre, los tipos de selección accesibles al usuario, el destino de instalación, podremos programar requisitos para instalarlo, como espacio libre en disco duro, CPU, versión del sistema operativo y muchos más, también se podrán programar acciones a llevar a cabo antes de proceder a la instalación, y una vez ésta ha finalizado. En la zona superior derecha, tenemos el icono "Edit Interface", que nos permite editar el interfaz de instalación, cambiando su fondo, y el texto de cada uno de los pasos de la instalación.

Una vez, ya tenemos todas las opciones rellenadas conforme a nuestra necesidad, podemos pulsar en el icono de la zona superior izquierda "Build", que construirá el paquete de instalación. Dicho paquete puede entregarse como tal, o puede meterse en una DMG tal como aprendimos anteriormente.

Compilación de un driver SQL

El último de los escollos a sortear para distribuir nuestra aplicación es que tengamos el driver del tipo de base de datos que vamos a usar en nuestra aplicación compilado. La versión Open Source de Qt4, por cuestiones de licencias no lleva compilados los drivers más que de ODBC y SQLite (que va integrado en Qt4), y quizás con esto haya suficiente para muchos, pero una gran cantidad de gente usa otras bases de datos, y aunque podrían usar el driver ODBC para ello, siempre es recomendable usar el driver nativo sin más añadidos.

Vamos por tanto a explicar, como compilar el driver para una de las bases de datos más usadas, que es sin duda MySQL. Para poder hacerlo, además de tener instalado el sistema de desarrollo completo, como ya tenemos, para poder ejecutar qmake, g++ y make desde la linea de comandos, tendremos que bajarnos los fuentes de MySQL y de Qt4.

Los fuentes de MySQL, podemos bajarlos de http://www.mysql.com/downloads/mysql/ en la version essentials. A parte de la instalación completa, tendremos dos carpetas fundamentales que son /lib y /include, donde tendremos respectivamente la librería necesaria para nuestro cliente, y los fuentes de MySQL.

Los fuentes de Qt4, están en la web de Nokia http://qt.nokia.com/ , y se llama qt-everywhere-opensource-src-4.x.x . En este paquete encontraremos los fuentes que necesitamos de nuestro driver en la carpeta $QT_SOURCES/src/plugins/sqldrivers/mysql. Ahí encontraremos un fichero main.cpp con el código del driver y el fichero de proyecto mysql.pro .

Situados en el directorio de los fuentes del driver para Qt4, ejecutaremos los siguientes comandos en línea:

1.- Windows (VS2008):

Presuponemos %PATH_MYSQL% como el raiz de los fuentes de MySQL.

```
qmake "INCLUDEPATH+=%PATH_MYSQL%\include" "LIBS+=%PATH_MYSQL%\ lib\opt\libmysql.lib" mysql.pro
nmake debug
nmake release
```

Tendremos al final las versiones debug y release de los drivers, estática y dinámica (qsqlmysql4.dll / qsql_mysql.lib).

2.- Linux:

Presuponemos $PATH_MYSQL como la raiz de los fuentes de MySQL

```
qmake "INCLUDEPATH+=$PATH_MYSQL/include" "LIBS+=-L$PATH_MYSQL/lib -lmysqlclient_r" mysql.pro
make
```

Se creará la versión release dinámica libqsqlmysql.so .

3.- Mac OSX:

Presuponemos $PATH_MYSQL como la raiz de los fuentes de MySQL

```
qmake -spec macx-g++ "INCLUDEPATH+=PATH_MYSQL/include" "LIBS+=-LPATH_MYSQL/lib -lmysqlclient_r" mysql.pro
make debug
make release
```

Se crearán la versión release y debug dinámicas libqsqlmysql.dylib y libqsqlmysql_debug.dylib .

Con esto damos por finalizado el contenido de este libro, en el que hemos intentado dotarlo de todo el contenido práctico necesario para el desarrollo eficaz de aplicaciones de propósito general en Qt4, pero aún queda mucho camino por andar, pero sin duda ya tenemos los fundamentos para que este viaje sea más agradable y sencillo.

BIBLIOGRAFIA

An introduction to design patterns in C++ with Qt4 - Prentice Hall - Alan & Paul Ezust

The book of Qt4. The art of building Qt applications - No Starch Press - Daniel Molkentin

C++ GUI Programming with Qt4 - Prentice Hall - Jasmin Blanchette & Mark Summerfeld

Qt Assistant – QT NOKIA SDK

The C++ Programming Language – Addison Wesley – Bjarne Stroustrup

Finalizado el 20 de Octubre de 2010 – anónimo