



UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Escuela Técnica Superior de Ingenieros
de Telecomunicación

Manual de diseño e implementación de interfaces

*Diseño e implementación de Interfaces
ETSI de Telecomunicación*

Himar Alonso Díaz

MANUAL DE DISEÑO E IMPLEMENTACIÓN DE INTERFACES

Índice

1. Introducción	4
1.1. Sobre este documento	4
1.2. El lenguaje C++	5
1.3. La biblioteca gráfica Qt-3	12
1.4. El lenguaje SQL	12
2. Programación orientada a objetos	14
2.1. Funciones de una clase	16
2.2. Clases y objetos. Miembros <i>estáticos</i>	16
2.3. Encapsulación	18
2.4. Constructor y destructor	19
2.5. Herencia	21
2.6. Polimorfismo	23
3. Programación de la interfaz gráfica del usuario (GUI)	26
3.1. Primera aplicación gráfica: un <i>Hello World!</i>	27
3.2. Entorno de desarrollo gráfico: <i>Qt Designer</i>	28
3.3. Elementos gráficos	30
3.4. Propiedades y eventos. Conexiones	32
4. Bases de datos	36
4.1. Estudio inicial y diseño	36
4.2. Formas normales	36
4.2.1. Primera forma normal (1FN)	37
4.2.2. Segunda forma normal (2FN)	37
4.2.3. Tercera forma normal (3FN)	37
4.2.4. Forma normal de Boyce Codd (BCFN)	38
4.2.5. Cuarta forma normal (4FN)	38
4.3. Bases de datos con MySQL	39
4.3.1. Definición de datos	39
4.3.2. Manipulación de datos	42
5. Acceso a bases de datos con C++	44
Referencias	47

1 Introducción

1.1. Sobre este documento

Este documento pretende ser una guía de consulta que permita al lector iniciarse en los tres grandes temas que se abarcan en el mismo:

- Desarrollo de aplicaciones mediante el uso de *objetos*, un concepto avanzado de estructura de datos con capacidad de intervenir activamente en el programa.
- Desarrollo de aplicaciones con interfaz *gráfica*.
- Diseño y optimización de *bases de datos*, y acceso a las mismas mediante *consultas*.

Cada uno de los temas tratados incluye pequeños ejemplos ilustrativos, con el código de las estructuras que se explican, o bien a través de imágenes del entorno de desarrollo. Todo el código utilizado se encontrará adjunto en formato electrónico. La primera línea de cada fragmento de código indicará entre corchetes “[...]” la *ruta* en la que se encuentra el fichero. Por ejemplo:

```
[ cpp/estructuras/if.cpp ]
if (x == 1)
{
    // Sentencia True;
} else {
    // Sentencia False;
}
```

Este código se encontrará en el fichero `if.cpp`, dentro de la carpeta `cpp/estructuras`. La mayoría de los ficheros *no* están preparados para ser compilados, ya que muchas veces sólo contienen fragmentos de código. He pretendido que los ejemplos utilizados sirvan como *iniciación* a cada uno de los aspectos tratados.

El lenguaje elegido para ilustrar los ejemplos que contienen este documento es el C++. Para compilar he utilizado G++, un compilador que forma parte del proyecto GCC de GNU¹ bajo el sistema operativo GNU/Linux. Esto no debería suponer ningún inconveniente para los usuarios de otras plataformas ya que G++ se encuentra disponible para varios sistemas operativos, y además se puede utilizar cualquier otro compilador.

En este punto conviene aclarar que este documento *no* es un manual para aprender a programar, ya que de hecho se requieren unos conocimientos mínimos de programación, pero tampoco es una referencia especializada en cada uno de los temas que se explican, sino una guía de introducción a los mismos.

A continuación se disponen tres subapartados introductorios sobre las estructuras básicas del lenguaje C++, la biblioteca Qt-3 para crear aplicaciones gráficas, y el lenguaje SQL para bases de datos.

¹Puede obtener más información en la web del proyecto: <http://gcc.gnu.org/>

1.2. El lenguaje C++

C++ es un lenguaje de programación muy potente, diseñado como extensión del lenguaje C y que además incluye soporte de *plantillas –templates–*, o también *programación genérica*, y soporte para *programación orientada a objetos*, imprescindible para poder abordar el Capítulo 2.

C++ está considerado por muchos programadores como el lenguaje más versátil, debido a que permite trabajar tanto a alto como a bajo nivel, sin embargo es a su vez uno de los que menos automatismos incorpora. Posee una serie de propiedades difíciles de encontrar en otros lenguajes de alto nivel, como son:

Sobrecarga de funciones: Permite utilizar el mismo nombre para dos o más funciones diferentes. El compilador usará una u otra dependiendo de los parámetros usado. Es posible incluso redefinir los operadores de C++.

Identificación de tipos en tiempo de ejecución: Es decir, que mientras se está ejecutando el programa, es posible saber de qué tipo es una entidad que ha sido almacenada en memoria.

En los siguientes ejemplos se explican las estructuras básicas de programación con C++. Para compilar los programas con G++ hay que escribir la siguiente sentencia en la consola: “g++ -o programa archivo.cpp” donde *programa* será el nombre del *ejecutable* una vez esté compilado, y *archivo.cpp* es el fichero que contiene el código fuente.

::: Primera toma de contacto, un Hello World :::

En este primer ejemplo tenemos mucha información sobre algunos aspectos importantes de C++.

```
[ cpp/helloworld/hello.cpp ]
/* Esto es un HelloWorld
   escrito en lenguaje C++ */
#include <iostream>
using namespace std;
int main () {
    cout << "Hola mundo!" << endl; //Imprimimos por pantalla
}
```

Por ejemplo, a la vista del código del *Hello World* podemos sacar algunas conclusiones:

- La biblioteca que nos permite sacar un texto por pantalla se llama *iostream*, y la forma de utilizarla es mediante la palabra *#include*.
- La función principal en un programa escrito en C++ se llama *main*. Al ver su forma podemos intuir cómo será la estructura de las demás funciones.
- La sentencia para sacar un texto por pantalla es *cout*.
- Cada sentencia acaba con punto y coma “;”
- Para comentar una línea se utiliza “//”. Si el comentario ocupa más de una línea lo ponemos entre “/*” y “*/”.

::: Tipos de datos. Declaración de variables :::

Los tipos de datos primitivos de C++ son los siguientes. La declaración de variables puede realizarse dentro de una función si queremos que sólo sean accesibles desde esa función, o bien fuera de ella, en cuyo caso hablaremos de variables *globales*. El modificador `unsigned` puede ponerse delante de los tipos *enteros* para indicar que sólo contendrá valores positivos.

```
[ cpp/estructuras/tipos.cpp ]
//Tipos enteros
int a;           // Número entero
unsigned int b; // Número positivo
long int c;
long long d;
short e;
char f;         // Carácter ASCII
char g;
wchar_t h;
//Tipos flotantes
float i;        // Número real
double j, k;
long double l = 1.0;
```

::: Funciones :::

En la función `main` del *Hello World* ya se puede averiguar la estructura de las funciones en C++. Todas las funciones devuelven algún tipo de datos, así que la declaración de las mismas comienza siempre indicando el *tipo*. Luego se indica el *nombre* de la función, y los *parámetros*, que irán entre paréntesis “()”. Las *instrucciones* irán entre llaves “{}”. El siguiente programa realiza el producto de dos variables y lo almacena en la variable global `resultado`; luego hace la suma y la almacena en la variable `z` de la función `main`.

```
[ cpp/estructuras/funciones.cpp ]
int resultado; // Variable global, accesible desde cualquier función

int suma (int a, int b) {
    return a + b; /* Los parámetros a y b se utilizan como variables de la función
suma. Utilizamos la palabra return para devolver el resultado. */
}

void producto (int a, int b) {
    resultado = a * b;
}

int main () {
    int x, y, z; // x,y,z son variables locales de la función main
    x = 2;
    y = 3;
    producto(x, y);
    z = suma(x, y);
}
```

Nota: Obsérvese la función `producto`. Según lo que hemos explicado, esta función devuelve un dato de tipo “`void`”. Lo que en realidad significa esta *palabra clave* es que la función `producto` *no devuelve ningún dato*. En otros lenguajes de programación, este tipo de funciones recibe el nombre de *procedimientos*, sin embargo C++ sólo utiliza funciones.

::: Control de flujo. Condición :::

```
[ cpp/estructuras/if.cpp ]
if (x == 1)
{
    // Sentencia True;
} else {
    // Sentencia False;
}
```

::: Bucles while y do...while :::

Estos dos tipos de bucles son muy similares. La única diferencia es que el *while* puede no ejecutarse nunca si no se cumple la condición, mientras que el *do...while* se ejecutará al menos una vez:

```
[ cpp/estructuras/whiledowhile.cpp ]
// Bucle while
// Incrementar x mientras sea menor que 100:
while (x < 100) {
    x++;
}
// Bucle do...while
// Hace la misma función:
do {
    x++;
} while (x<100);
```

::: Bucle for :::

El bucle *for* se diferencia del *while* y el *do...while* en que conocemos de antemano el número de iteraciones que deben producirse. Entre paréntesis indicaremos el *valor inicial* del contador, luego la *condición de salida* del bucle, y finalmente el *incremento* del contador en cada iteración. El siguiente programa imprime por pantalla 100 líneas de texto, y además se indica el número de iteración:

```
[ cpp/estructuras/for.cpp ]
#include <iostream>
using namespace std;
int main() {
    int i; // Utilizaremos esta variable como contador para el bucle
    for (i=1; i<101; i++) {
        cout << "Iteracion numero " << i << endl;
    }
}
```

::: Registros de datos :::

Las *estructuras* o *registros de datos* surgen de la necesidad de agrupar una serie de variables que aun pudiendo ser de distinto tipo, están relacionadas entre sí de algún modo. C++, a diferencia de C, permite incluir no sólo variables sino también *funciones* dentro de los registros de datos. Sin embargo esta es una característica propia, no de los registros, sino de un elemento de la programación denominado *clase*, que estudiaremos en el Capítulo 2.

En el siguiente ejemplo se pide al usuario los datos de una ficha personal y luego se muestra por pantalla. Cada variable de tipo `ficha` contendrá tres campos: Nombre, edad y estatura. Para acceder al contenido de cada una de las variables del registro se utiliza un *punto*, de la siguiente manera: “`mificha.nombre`”.

[`cpp/estructuras/struct.cpp`]

```
#include <iostream>
using namespace std;

struct ficha {
    char *nombre;
    int edad;
    float estatura;
};

int main() {
    ficha mificha;
    // Pedimos los datos al usuario:
    cout << "Introduzca nombre: ";
    cin >> mificha.nombre;
    cout << "Introduzca edad: ";
    cin >> mificha.edad;
    cout << "Introduzca estatura: ";
    cin >> mificha.estatura;
    // Imprimimos el registro en pantalla
    cout << "Nombre:  " << mificha.nombre << endl;
    cout << "Edad:    " << mificha.edad << endl;
    cout << "Estatura: " << mificha.estatura << endl;
}
```

A continuación se explican dos novedades adicionales que presenta este ejemplo:

- Se utiliza “`cin`” para recoger el texto que el usuario introducirá a través del teclado y almacenarlo en una variable. Su uso es análogo al de `cout`.
- En C++ no suele utilizarse el tipo de datos `string` para definir una cadena de caracteres. En lugar de ello se utilizan, bien *arrays* de caracteres, o bien un *puntero* al primer carácter de la cadena, que es lo que hemos utilizado para el campo `nombre` al añadir un asterisco en su definición: “`char *nombre`”. Ahora explicaremos mejor cómo funcionan los *arrays* y los *punteros* en C++.

::: Arrays :::

Los *arrays* son agrupaciones de variables de un mismo tipo en posiciones sucesivas de memoria, con tamaño fijo², y que pueden tener más de una dimensión. Por eso se utiliza este elemento para operar con *vectores* o *matrices*.

Los *arrays* se declaran, al igual que una variable, comenzando por el *tipo* de datos que se quiere almacenar en cada posición; luego entre corchetes “[...]” se indica el número de posiciones que tendrá cada dimensión. Si por ejemplo queremos declarar una matriz de números reales de dimensiones 2×3 lo haremos así: “float mimatriz[2][3]”. Para acceder a la posición $A_{(1,2)}$ y guardar el valor “3,424” lo haremos de la siguiente manera: “mimatriz[1][2] = 3.424”.

En el siguiente programa vamos a crear una matriz de caracteres, de dos dimensiones, definida por el usuario. Luego pediremos cada uno de los términos, que deberán ser introducidos por teclado, y finalmente imprimimos la matriz por pantalla.

```
[ cpp/estructuras/array.cpp ]
```

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {
    int fila, nfilas, columna, ncolumnas;
    cout << "Introduzca el numero de filas: ";
    cin >> nfilas;
    cout << "Introduzca el numero de columnas: ";
    cin >> ncolumnas;
    /* Creamos una matriz de caracteres, de dos dimensiones, con el tamaño que indique
    el usuario para cada dimensión */
    char matriz[nfilas][ncolumnas];
    // Recorremos la matriz para recoger los datos:
    for (fila=0; fila<nfilas; fila++) {
        for (columna=0; columna<ncolumnas; columna++) {
            cout << "Introduzca el termino A("<< fila+1 << columna+1 <<
            "): ";
            cin >> matriz[fila][columna];
        }
    }
    // Imprimimos la matriz de forma ordenada:
    for (fila=0; fila<nfilas; fila++) {
        for (columna=0; columna<ncolumnas; columna++) {
            cout << matriz[fila][columna] << " ";
        }
        cout << endl; // Salto a la siguiente fila
    }
}
```

²A diferencia de otros lenguajes, C++ permite definir el tamaño de los *arrays* en medio de una función—no en un preámbulo, o en otra función— pero una vez se define el tamaño, ya no se puede variar, por eso decimos que el tamaño es *fijo*.

Nota: En este programa hemos utilizado dos bucles *for* encadenados para recorrer la matriz en sus dos dimensiones. En general se deben utilizar “tantos bucles *for* encadenados como dimensiones tenga el *recorrido* que queremos hacer”. Si por ejemplo queremos recorrer la diagonal principal de una matriz cuadrada de dos dimensiones sólo necesitaremos *un* bucle *for*, porque a pesar que la matriz sea de dos dimensiones, el recorrido es de una sola; al saber qué relación existe entre filas y columnas, sólo necesitamos un *contador* para el recorrido.

Este es el resultado cuando compilamos y ejecutamos el programa:

```
> g++ -o array array.cpp
> ./array
Introduzca el numero de filas: 3
Introduzca el numero de columnas: 4
Introduzca el termino A(11): 2
Introduzca el termino A(12): a
Introduzca el termino A(13): g
Introduzca el termino A(14): 4
Introduzca el termino A(21): f
Introduzca el termino A(22): 6
Introduzca el termino A(23): e
Introduzca el termino A(24): 3
Introduzca el termino A(31): 7
Introduzca el termino A(32): h
Introduzca el termino A(33): u
Introduzca el termino A(34): 8
2 a g 4
f 6 e 3
7 h u 8
```

::: **Punteros. Operadores de referencia, indirección y acceso** :::

Un puntero es un dato de tipo *entero* que contiene una *dirección de memoria*, a la cual se dice que “está apuntando”. Para que un puntero pueda comportarse como tal, es necesario por tanto que pueda direccionar todas las posiciones de memoria. Es por ello que en casi todos los lenguajes de programación los datos de tipo entero tienen el tamaño del bus de datos de la arquitectura para la cual se compila un programa³.

Para trabajar con punteros se recurre al uso de los operadores de *referencia* (&) e *indirección* (*), dos operadores muy sencillos que definimos brevemente:

Referencia (&) devuelve la posición de memoria en la que se encuentra el operando. Normalmente el operando es una variable cuya posición en memoria queremos conocer.

Indirección (*) considera su operando como una dirección, y devuelve su contenido.

Junto con el operador de *referencia* es imprescindible para trabajar con punteros.

Veamos cómo se usan estos operadores mediante un ejemplo en el que trabajamos con punteros. Observe que si bien el puntero no es un dato de tipo *float*, cuando apunta a un dato de este tipo, lo podemos tratar como si realmente lo fuera:

³Por ejemplo, en una arquitectura de 32 bits, un entero *-int-* tiene un tamaño de 32 bits, de modo que los punteros –que son de tipo entero– puedan direccionar todas las posiciones de memoria.

```
[ cpp/estructuras/refind.cpp ]
#include <iostream>
using namespace std;

float a;          // Una variable con un numero real
float *puntero;  // Puntero a un dato de tipo real

int main() {

    a = 3.14159; // Asignamos un valor a la variable a
    puntero = &a; // El puntero apunta a la dirección de a
    cout << "cout puntero: " << puntero << " (Dirección)" << endl;
    cout << "cout *puntero: " << *puntero << " (Dato)" << endl;

    *puntero = 2.3534; // Equivale a escribir: a = 2.3534
    cout << "cout a: " << a << " (Dato)" << endl;
}

```

Este programa está preparado para compilar y ejecutar. Lo que vemos en la pantalla nos ayuda a comprender mejor el manejo de los punteros.

Cuando trabajamos con punteros a *registros de datos* (también con punteros a *clases*, como veremos en el Capítulo 2) cambia la forma de *acceder* a los campos de la estructura. Si bien con un dato de tipo `struct` accedemos a sus campos con un *punto* “.” (por ejemplo: `estructura.campo = valor`), cuando utilizamos punteros accederemos a los campos con “->” (por ejemplo: `punteroaestructura->campo = valor`).

En el siguiente ejemplo se muestra el uso de los operadores de acceso. En primer lugar accedemos al registro de la forma tradicional e imprimimos el nombre por pantalla. Luego hacemos lo mismo, pero con un puntero.

```
[ cpp/estructuras/structacceso.cpp ]
#include <iostream>
using namespace std;

struct ficha {
    char *nombre;
    int edad;
    float estatura;
};

int main() {
    ficha mificha;
    ficha *tuficha;
    // Rellenamos la ficha de la forma tradicional:
    mificha.nombre = "Marta";
    mificha.edad = 23;
    mificha.estatura = 1.62;
    cout << "Nombre: " << mificha.nombre << endl;
}

```

```

// Ahora modificamos la ficha haciendo uso del puntero:
tuficha = &mificha;
tuficha->nombre = "Himar";
tuficha->edad = 21;
tuficha->estatura = 1.75;
cout << "Nombre: " << tuficha->nombre << endl;
}

```

Si ejecutamos este programa veremos dos líneas en la salida. En primer lugar imprimirá el nombre de “Marta” y luego el de “Himar”. Hemos modificado la misma variable directamente, y luego haciendo uso de un puntero.

∴ *Conclusión* ∴

Con esta introducción se ha pretendido que el usuario con conocimientos de programación –en cualquier otro lenguaje– pueda manejar las estructuras básicas de C++, imprescindibles para comprender los conceptos de este manual, ya que éstos se apoyan fuertemente en el código y en programas de ejemplo.

1.3. La biblioteca gráfica Qt-3

Qt-3 es una biblioteca⁴ multiplataforma que utiliza C++, y que permite desarrollar interfaces gráficas de usuario (en adelante GUI, del inglés: *Graphic User Interface*). Esta biblioteca fue creada por la compañía *Trolltech* y en la actualidad es utilizada, entre otras cosas, para el desarrollo del entorno de escritorio KDE, para GNU/Linux.

La biblioteca Qt-3 está diseñada para ofrecer una programación sencilla e intuitiva, y de hecho es posible desarrollar aplicaciones Qt escribiendo el código en C++. Sin embargo existen entornos de desarrollo que facilitan aún más esta tarea. El entorno de desarrollo por excelencia para Qt-3 es el *Qt Designer* (Figura 1), también creado por *Trolltech*. Este entorno abre un abanico de posibilidades a los programadores, permitiéndoles crear y modificar formularios de forma gráfica, existiendo también la posibilidad de acceder al código para modificarlo directamente[3].

En el Capítulo 3 comprobaremos claramente las ventajas que nos aporta el *Qt Designer*.

1.4. El lenguaje SQL

En los primeros programas informáticos en los que se precisaba manejar bases de datos era necesario crear procedimientos y funciones específicos para acceder, procesar y gestionar debidamente la información, que estaría guardada en un lugar conocido de un dispositivo de almacenamiento. De este modo, cada programador y cada programa contaba con un criterio propio para la gestión de sus datos. Sin embargo a medida que los programas se hacen más complejos resulta muy aparatoso tener que programar una aplicación que además de cumplir con un objetivo determinado, deba encargarse también de la gestión de datos.

⁴Una biblioteca es un conjunto de procedimientos y funciones agrupadas en uno o varios archivos[1]. En el caso de la biblioteca Qt-3, estos procedimientos y funciones se utilizan para automatizar la creación de objetos gráficos.

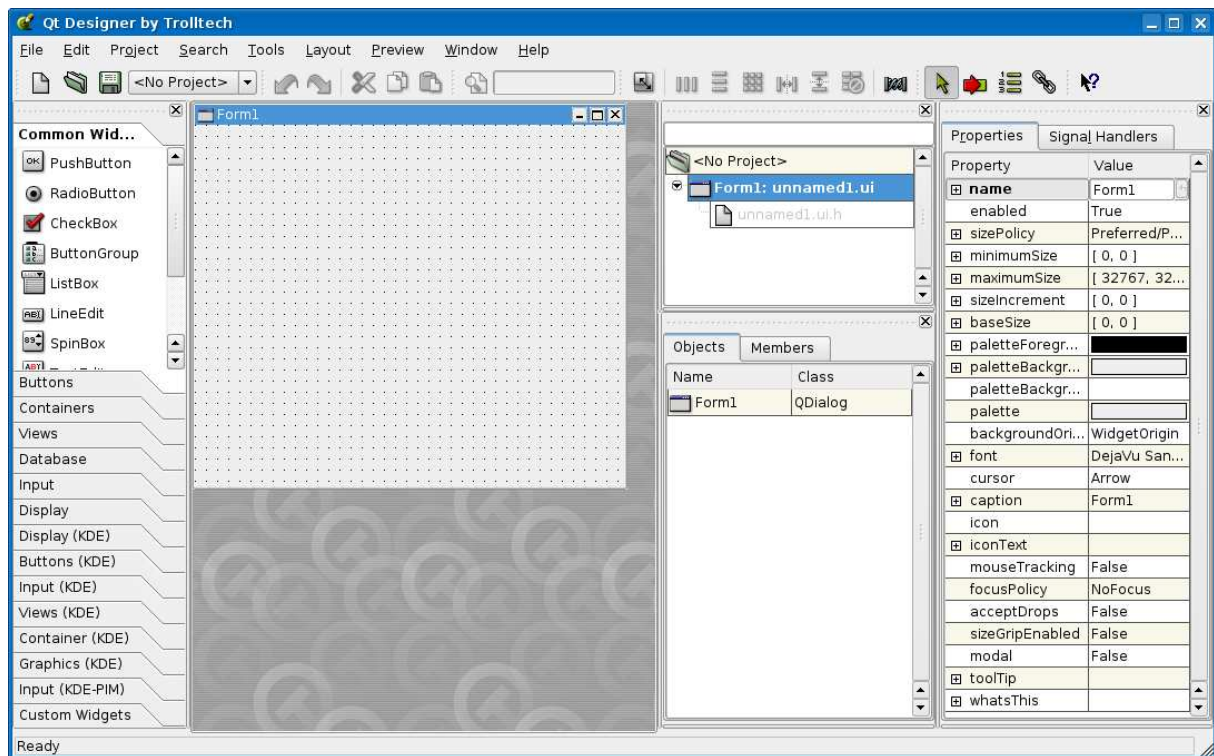


Figura 1: Captura del entorno de desarrollo *Qt Designer*

La solución a este problema nace con los *servidores de bases de datos*, también llamados *sistemas de gestión de bases de datos*, que son aplicaciones específicas para el manejo de información. De esta manera cada vez que queramos introducir un registro en una base de datos no tendremos que preocuparnos por guardarla en el disco duro; bastará con indicarle al servidor la información del registro que deseamos introducir. Será éste quien se encargue de guardarla en el disco duro y de tenerla localizada para cuando esta información sea solicitada.

Al dividir las tareas de los programas con el fin de que cada uno de ellos desarrolle sus funciones de manera óptima, surge el problema de la *comunicación* entre ambos programas. Para que esta comunicación sea posible, los dos programas deben *utilizar el mismo lenguaje*. Se han definido varios lenguajes para el acceso a bases de datos, siendo el más importante el SQL, ya que es el que emplean los sistemas de gestión más utilizados en la actualidad (DB2, Oracle, SQL Server, Sybase ASE, MySQL, PostgreSQL o Firebird).

El lenguaje SQL (*Structured Query Language*, Lenguaje de consulta estructurado) es un lenguaje que permite acceder a *bases de datos relacionales*⁵ y realizar diversos tipos de operaciones sobre las mismas [1].

Con una sintaxis sencilla mediante una línea de comandos se pueden realizar consultas para recuperar información de manera ordenada, así como para introducir nuevos registros o bien modificar la estructura de la base de datos. No es objetivo de esta sección hacer un análisis tan pormenorizado como el que hemos hecho con C++, ya que en el Capítulo 4 tendremos ocasión de conocer la sintaxis y las instrucciones SQL. No obstante haremos un brevísimo adelanto. En el siguiente ejemplo se muestra la orden que debemos introducir

⁵En el Capítulo 4 se abordará el tema del diseño y la estructura de las bases de datos.

cuando queremos consultar los *registros* de una *tabla* llamada “clientes” donde sólo nos interesa ver los *campos* “Nombre” y “Apellido”:

```
[ sql/ejemplos/mostrar.sql ]
```

```
SELECT Nombre,Apellidos FROM cliente
```

Como puede apreciarse, la estructura de las consultas SQL es muy sencilla, aunque quizá cuando queramos introducir filtros pueda parecer algo más engorroso. Sin embargo, como veremos en el Capítulo 5, los procedimientos para acceder a las bases de datos pueden automatizarse bastante cuando en lugar de realizar las consultas por línea de comandos, disponemos de un programa (que podemos hacer en C++ por ejemplo) que:

- Mediante un entorno gráfico y sencillo de utilizar, permita crear filtros de búsqueda, seleccionar criterios de ordenación, etc.
- Genere el código SQL correspondiente y realice la consulta al servidor.
- Recoja los datos proporcionados por el servidor y los muestre por pantalla.

La biblioteca Qt-3, de la que hablamos anteriormente, cuenta con métodos específicos para acceder a bases de datos mediante SQL, así que resultará muy sencillo desarrollar aplicaciones gráficas que realicen estas tres funciones que acabamos de mencionar.

2 Programación orientada a objetos

Una de las principales características que vimos en la introducción sobre el lenguaje C++ es que tiene soporte para la *programación orientada a objetos* (en adelante POO). Este concepto de programación es algo más abstracto ya que permite atribuir *interactividad* a determinados elementos de un programa, a los que llamaremos *objetos*. Es decir, que estos objetos podrán tener una participación *activa* dentro del programa.

Al empezar a hablar de POO se hace necesario introducir el concepto de *clase*. Para ello partiremos de su “antecesor”, el *registro de datos*, que en C y C++ se denomina **struct**. Veamos un ejemplo:

```
[ cpp/estructuras/structobj.cpp ]
```

```
struct rectangulo {
    int ladoa;
    int ladob;
    int area;
};
int main() {
    rectangulo r;
    r.ladoa = 2;
    r.ladob = 3;
    r.area = r.ladoa * r.ladob;
}
```

En primer lugar definimos un registro al que hemos llamado `rectangulo`, que contiene las tres variables que se muestran. Luego declaramos una variable `r`, que contendrá un registro de tipo `rectangulo`, con el que podremos operar de la forma que se indica.

Obsérvese que a todos los efectos, el registro `rectangulo` define un elemento accesible al programador y completamente “pasivo”. Esto supone que cualquier tipo de operación deberá realizarse a nivel externo, de manera que el registro solamente se utiliza para almacenar los resultados. . . ¿Por qué no crear un nuevo elemento que además de almacenar variables, tenga capacidad para ejecutar sus propias funciones? La POO hace que esto sea posible si en lugar de utilizar un registro de datos utilizamos una *clase*⁶. En el ejemplo del rectángulo, la siguiente clase permite calcular el área –a nivel interno– con tan solo llamar a la función `calcularArea`:

```
[ cpp/poo/clase.cpp ]
class rectangulo {
public:
    int ladoa;
    int ladob;
    int area;
    void calcularArea () {
        area = ladoa * ladob;
    }
};
int main() {
    rectangulo r;
    r.ladoa = 2;
    r.ladob = 3;
    r.calcularArea();
}
```

Aunque puede deducirse observando la estructura de la clase `rectangulo`, la forma de definir una clase es la siguiente:

```
[ cpp/poo/esquemaclase.cpp ]
class identificador {
    // Contenido de la clase: Variables y funciones.
};
```

Este primer ejemplo puede dar una idea del potencial del uso de este elemento en la programación. A lo largo de este capítulo trataremos de matizar bien los nuevos conceptos que introduce la POO, y estudiaremos las propiedades de *encapsulación*, *herencia* y *polimorfismo*, así como el uso de las funciones *constructor* y *destructor*.

⁶Aunque los conceptos de *registro de datos* (`struct`) y *clase* (`class`) son diferentes, C++ permite definir una *clase* utilizando indistintamente cualquiera de las dos palabras clave: `struct` o `class`.

2.1. Funciones de una clase

No es necesario que la definición de las funciones de una clase se realice en el *entorno* de ésta (entre llaves). Si en el ejemplo anterior hubiésemos querido definir la función `calcularArea` fuera del entorno de la clase `rectangulo` lo podríamos haber hecho, de la siguiente manera:

```
[ cpp/poo/claseyfuncion.cpp ]
class rectangulo {
public:
    int ladoa;
    int ladob;
    int area;
    void calcularArea ();
};
void rectangulo::calcularArea () {
    area = ladoa * ladob;
}

int main() {
    rectangulo r;
    r.ladoa = 2;
    r.ladob = 3;
    r.calcularArea();
}
```

Observe cómo debemos declarar una función si queremos que ésta pertenezca a una clase determinada, aun no estando dentro de su entorno: “`tipo clase::funcion`”. No obstante, será *necesario* incluir una *cabecera* de la función `calcularArea` en el entorno de la clase. Para el compilador, declarar una función fuera o dentro del entorno de la clase es absolutamente *indiferente*, así que hacerlo de una manera u otra es simplemente una cuestión de *organización*.

2.2. Clases y objetos. Miembros *estáticos*

Cuando se habla de POO con frecuencia se suelen confundir los conceptos de *clase* y *objeto*, pese a que la diferencia entre ambos es bien sencilla. En el siguiente ejemplo se calcula el área de un rectángulo y se asigna el resultado a una variable de tipo entero. Servirá para aclarar cualquier confusión:

```
[ cpp/clasesyobjetos/clasesyobjetos.cpp ]
// Definimos una clase rectangulo:
class rectangulo {
public:
    int ladoa;
    int ladob;
    int area() {
        return ladoa * ladob;
    }
};
```

```

// Declaramos las variables que utilizaremos:
int a;
rectangulo r;

// Ejecucion del programa principal:
int main() {
    r.ladoa = 2;
    r.ladob = 3;
    a = r.area();
}

```

En primer lugar definimos una *clase* con el nombre `rectangulo`, en la que se declaran sus variables y operaciones (también llamadas *métodos*). Con esta definición de la clase `rectangulo` podemos crear tantas variables de tipo `rectangulo` como necesitemos en nuestro programa. Diremos que cada una de esas variables son *objetos* de la clase `rectangulo`. Es decir:

- La *clase* es una plantilla que define una estructura, con sus propiedades, variables y métodos. Podemos utilizar el símil de “los planos de una casa”.
- Por el contrario un *objeto* es una instanciación de una clase, una ocurrencia de ésta, que tiene los atributos definidos por la clase, y sobre la que se puede ejecutar las operaciones definidas en ella. Del mismo modo que para hacer dos casas iguales podemos utilizar los mismos planos, para crear dos objetos de la misma clase, sólo tenemos que definir la clase una vez [1].

Dos objetos de una misma clase pueden considerarse como dos *variables* del mismo *tipo*, pero que pueden tener valores diferentes. Sin embargo podemos definir –dentro de la clase– un miembro⁷ que sea *común* a todos los objetos de una misma clase. Nos referimos entonces a un *miembro estático*. Para ello utilizamos la palabra clave “`static`”. Los miembros estáticos deberán ser inicializados *fuera* de la clase. Veamos un ejemplo en el que utilizamos una variable estática para contar el número de objetos de una clase que han sido declarados:

```

[ cpp/poo/static.cpp ]
#include <iostream>
using namespace std;
class rectangulo {
public:
    /* En la variable estatica almacenaremos el numero de objetos
       declarados de la clase rectangulo */
    static int numrectangulos;
    int area;
};
// Inicializamos la variable estatica a 0:
int rectangulo::numrectangulos = 0;
int main() {

```

⁷Podemos decir que las variables y funciones son *miembros* de la clase a la que pertenecen [2].

```

// Creamos dos rectangulos:
rectangulo r,s;
r.numrectangulos++;
r.area = 10;
s.numrectangulos++;
s.area = 20;
cout << r.numrectangulos << " = " << s.numrectangulos << endl;
}

```

Si consultamos el valor de un miembro estático desde cualquier objeto de esa clase obtendremos siempre el mismo resultado [2].

2.3. Encapsulación

La *encapsulación* es una propiedad que permite especificar qué elementos de una clase serán accesibles al programador que utilice objetos de esa clase, y cuáles no.

Recordemos que en el `struct` todas las variables eran *visibles*. En las clases, por defecto, esto no es así. Distinguiremos tres tipos de *miembros*, según sea el acceso [2]:

Público: Cualquier miembro público de una clase es accesible desde cualquier parte donde sea accesible el propio objeto. Se utiliza la palabra “`public:`”. Nótese que en los ejemplos de clases vistos hasta ahora todos los miembros son *públicos*.

Privado: Los miembros privados de una clase sólo son accesibles por los propios miembros de la clase y en general por objetos de la misma clase, pero no desde funciones externas o desde funciones de *clases derivadas*⁸. Se utiliza la palabra “`private:`”

Protegido: Con respecto a las funciones externas, es equivalente al acceso privado, pero con respecto a las clases derivadas se comporta como público. Se utiliza la palabra “`protected:`”

Estas palabras *clave* se utilizan como se muestra en el siguiente ejemplo (en este ejemplo sólo usaremos `public` y `private`):

[`cpp/poo/encapsulacion.cpp`]

```
#include <iostream>
```

```
using namespace std;
```

```
class rectangulo {
```

```
public:
```

```
    void definirRectangulo(int ladoa, int ladob) {
```

```
        a = ladoa;
```

```
        b = ladob;
```

```
        area = a * b;
```

```
    }
```

```
    void mostrarArea() {
```

```
        cout << "El area es: " << area << endl;
```

```
    }
```

⁸Estudiaremos el concepto de *clase derivada* cuando abordemos la propiedad de la *herencia*.

```

private:
    int a;
    int b;
    int area;
};

int main() {
    int tmpa, tmpb; // Variables temporales
    cout << "Introduzca el valor del lado a: ";
    cin >> tmpa;
    cout << "Introduzca el valor del lado b: ";
    cin >> tmpb;
    // Creamos un rectangulo con esas dimensiones:
    rectangulo r;
    r.definirRectangulo(tmpa, tmpb);
    // Mostramos el area:
    r.mostrarArea();
}

```

Como puede apreciarse, desde la función `main` sólo puede accederse a las funciones *públicas* que son `definirRectangulo` y `mostrarArea`. En la parte privada quedan las variables `a`, `b` y `area`, que sólo pueden ser utilizadas por las funciones internas de la clase. De este modo no permitimos que el programador que haga uso de esta clase establezca de modo arbitrario los valores para los lados y el área, ya que de ser así podrían no concordar el área con el producto de los lados. Muchas veces se utiliza la encapsulación precisamente para *proteger* al programa de los errores de programación, o de posibles *incoherencias*.

2.4. Constructor y destructor

En este punto vamos a explicar dos funciones muy importantes –aunque de uso opcional– del ámbito de la POO, que son las funciones *constructor* y *destructor*:

- El *constructor* es una función que se ejecuta en el momento de declarar un objeto, antes de que poder ser utilizado. Los constructores tienen el mismo nombre que la clase, no devuelven ningún valor y no pueden ser heredados. Deben ser funciones públicas, accesibles para el momento de la declaración. Al igual que cualquier otra función, el constructor puede realizar cualquier acción, pero el uso más común que suele tener es el de *inicializar las variables del objeto*.
- El *destructor* es una función que ejecuta un objeto justo antes de ser eliminado⁹ de la memoria. Debe tener el mismo nombre que la clase, precedido del símbolo “~”. Al igual que el constructor, el destructor tendrá que ser público, y no podrá ser heredado. El uso más común que suele tener es el de *liberar* la memoria que haya sido utilizada (por ejemplo para borrar listas encadenadas, o cuando la clase tenga datos de tipo puntero) o bien para *actualización de variables globales*.

⁹Un objeto puede ser eliminado en dos situaciones: bien porque termina el ámbito en el que fue definido, bien porque fue creado dinámicamente (con el operador `new`) y ahora se desea eliminar utilizando `delete`.

Puede haber *sobrecarga* de constructores o destructores. Como se explicó en la Sección 1.2, la sobrecarga de funciones nos permitirá crear en este caso más de un constructor o más de un destructor para cada clase.

Cuando explicamos el uso de los miembros estáticos, vimos un ejemplo donde se utilizaba una variable para contar el número de objetos. Con las funciones *constructor* y *destructor* podemos llevar esta cuenta de forma automática, es decir, que al declarar un objeto se incremente la variable `numrectangulos`, y que al eliminarlo se decremente. Para este ejemplo declararemos los objetos mediante *punteros*, para así poderlos borrar con la sentencia “delete”:

```
[ cpp/poo/constructor.cpp ]
```

```
#include <iostream>
```

```
using namespace std;
```

```
class rectangulo {
```

```
public:
```

```
    // Constructor:
```

```
    rectangulo(int ladoa, int ladob) {
```

```
        a = ladoa;
```

```
        b = ladob;
```

```
        area = a * b;
```

```
        numrectangulos++;
```

```
    }
```

```
    // Destructor:
```

```
    ~rectangulo() {
```

```
        numrectangulos--;
```

```
    }
```

```
    // Esta funcion muestra el numero de objetos declarados:
```

```
    void mostrarNumeroObjetos() {
```

```
        cout << "Numero de rectangulos: " << numrectangulos << endl;
```

```
    }
```

```
    // Esta funcion muestra el area del rectangulo:
```

```
    void mostrarArea() {
```

```
        cout << "Area del rectangulo: " << area << endl;
```

```
    }
```

```
private:
```

```
    int a;
```

```
    int b;
```

```
    int area;
```

```
    static int numrectangulos;
```

```
};
```

```
// Inicializamos la variable estatica:
```

```
int rectangulo::numrectangulos = 0;
```

```
int main() {
```

```
    // Creamos dos punteros a objetos de tipo rectangulo:
```

```
    rectangulo *r, *s;
```

```

// Al declarar un objeto rectangulo tendremos que inicializarlo:
r = new rectangulo(10,20);
r->mostrarArea();
r->mostrarNumeroObjetos();
s = new rectangulo(12,16);
s->mostrarArea();
s->mostrarNumeroObjetos();
/* Borramos uno de los dos rectangulos, y comprobamos que el
   destructor ha hecho su trabajo (decrementar el numero de objetos) */
cout << "(Borramos un rectangulo)"<< endl;
delete r;
s->mostrarNumeroObjetos();
}

```

Con el uso de las funciones *constructor* y *destructor* hemos conseguido simplificar bastante el código, ya que al declarar las variables, éstas quedan inicializadas automáticamente. Además, la cuenta del número de objetos se actualiza cada vez que se crea o se elimina un nuevo objeto. En definitiva son funciones con un evidente potencial de *simplificación y automatización*.

2.5. Herencia

La *herencia* es una propiedad que nos permite crear nuevas clases a partir de clases existentes, conservando las propiedades de la clase original y añadiendo otras nuevas.

Llamaremos *clase derivada* a la clase resultante, y *clase base* a la clase desde la cual derivó. A su vez, la *clase derivada* puede servir como *clase base* para sucesivas derivaciones, en cuyo caso estaremos hablando de *derivación múltiple*. Esta propiedad se denomina *herencia* porque la clase derivada *hereda* los miembros (variables y funciones) de la clase base [2].

La derivación de clases es análoga al orden natural del *pensamiento* en los humanos: en primer lugar se define un concepto general y luego se va concretando en distintos aspectos. Esto nos permite crear estructuras tan complejas como queramos.

Cuando queremos crear una clase a partir de otra contamos con dos posibilidades [2]:

- “class claseDerivada : public claseBase” nos permite crear una clase derivada, que hereda los miembros de la clase base como miembros *públicos* en la clase derivada.
- “class claseDerivada : private claseBase” nos permite crear una clase derivada, que hereda los miembros de la clase base como miembros *privados* en la clase derivada.

Vamos a empezar con un ejemplo sencillo. Supongamos que queremos tener una ficha de los alumnos y los profesores de un centro de enseñanza. Cualquier persona, sea profesor o alumno tendrá un *nombre* y un *DNI*, de modo que creamos una clase *persona* con estos dos campos. A continuación nos fijaremos en las características *distintivas* de alumnos y profesores (en la Figura 2 se muestra esta clasificación de manera gráfica):

- De los alumnos nos interesa saber en qué *carrera* están matriculados, el número de *créditos* aprobados, y si son becarios o no¹⁰.
- De los profesores queremos saber a qué *departamento* pertenecen y el número de *horas de clase* que tienen asignadas.

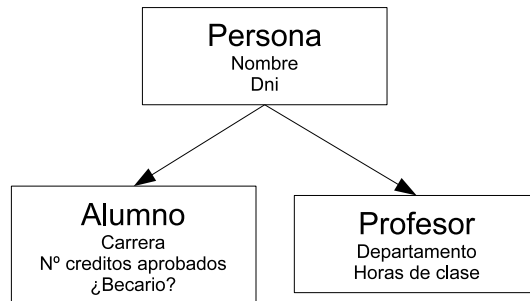


Figura 2: “Persona” es una *clase base*. “Alumno” y “Profesor” son *clases derivadas* que heredan los miembros de la clase “Persona”

Observe que los constructores de las clases derivadas `alumno` y `profesor` pueden utilizar las variables “Nombre” y “DNI”, ya que han sido *heredadas* de la clase `persona`.

Nota: Para copiar cadenas de caracteres no podemos hacer “`*cadena1 = *cadena2`” ya que de esta manera *no* estaríamos copiando el contenido de las cadenas, sino los *punteros*. Utilizaremos la función “`strcpy(cadena1,cadena2)`” de la biblioteca “`string.h`”, que lo que hace es copiar la cadena apuntada por `cadena2` en la cadena apuntada por `cadena1`.

[`cpp/poo/herencia.cpp`]

```

#include <iostream>
#include <string.h>
using namespace std;
  
```

```

class persona {
public:
    char *nombre;
    char *dni;
};
  
```

```

class alumno : public persona {
public:
    char *carrera;
    float creditos;
    int becario;
    alumno(char *nom, char *dn, char *carr, float cre, int bec) {
        strcpy(nombre,nom);
        strcpy(dni,dn);
        strcpy(carrera,carr);
    }
};
  
```

¹⁰En C++ no existe de forma primitiva el tipo de datos *booleano*. En su lugar utilizaremos un dato *entero* (`int`), al que sólo asignaremos valores lógicos— 0 ó 1.

```

        creditos = cre;
        becario = bec;
    }
};

class profesor : public persona {
public:
    char *departamento;
    float horas;
    profesor(char *nom, char *dn, char *dep, float hor) {
        strcpy(nombre,nom);
        strcpy(dni,dn);
        strcpy(departamento,dep);
        horas = hor;
    }
};

int main() {
    alumno A("Juan", "23748384", "Telecomunicaciones", 61.5, 0);
    profesor P("Javier", "47582934", "DIT", 10.0);
}

```

Podemos implementar una mejora en este programa, considerando lo siguiente: En el centro que queremos gestionar hay exclusivamente o *alumnos* o *profesores*. Sin embargo con el programa que acabamos de hacer, además de alumnos y profesores, sería posible crear un objeto de tipo *persona* y asignarle un *nombre* y un *DNI*. Para evitarlo, modificamos la *encapsulación* de los miembros *nombre* y *dni*, mediante la palabra clave *protected*:

```

[ cpp/poo/herprot.cpp ]
class persona {
protected:
    char *nombre;
    char *dni;
};

```

2.6. Polimorfismo

El *polimorfismo* es posiblemente el concepto más importante en relación con la POO. Si bien para definir la *herencia* partíamos de las *clases*, para el *polimorfismo* el punto de partida está en las *funciones*. En muchos ejemplos vistos hasta ahora hemos utilizado una función para calcular el *área*... de un rectángulo. Podríamos plantearnos hacer un programa que también calcule el área de otras figuras geométricas. Si intentamos elaborar esta idea tan simple llegaremos siguiente problema:

- Queremos un programa para calcular el área de varias *figuras*. Pongamos las siguientes: *rectángulo*, *cuadrado* y *triángulo*.

- Estas tres figuras geométricas definen un *área*, así que en una clase *base* a la que llamaremos *figura* tendremos un miembro común: el área.
- Sin embargo el área *se calcula de manera diferente* dependiendo de cada figura.
- Si definimos una función para calcular el área en las clases derivadas (*rectángulo*, *cuadrado* o *triángulo*), pero no en la clase *figura*, sólo podremos calcular el área de un rectángulo, refiriéndonos a éste como un *rectángulo* (puntero a clase derivada), pero no como una *figura* (puntero a clase base). Haciendo uso del polimorfismo podremos saber el área de una figura *sin importarnos de qué figura se trate*.

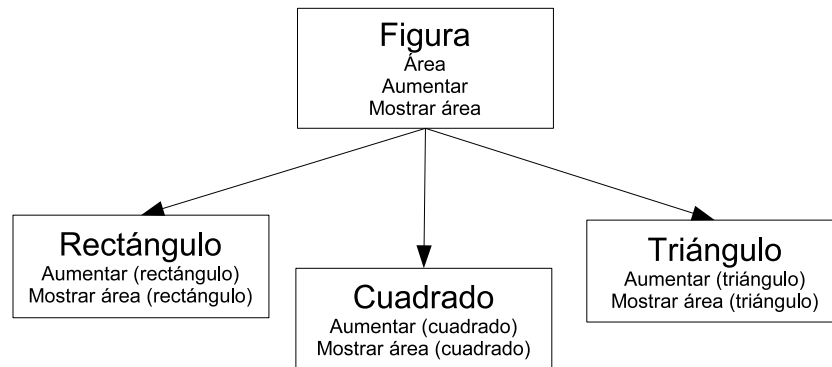


Figura 3: Todas las clases derivadas tienen las mismas funciones, pero en cada una se opera de forma diferente

Para dar solución al problema planteado, en el polimorfismo contamos con las llamadas *funciones virtuales*. Éstas son funciones que, por ser comunes a todas las figuras, estarán en la clase base (*figura*) pero como su ejecución depende del objeto concreto al que se refiere, *se definen* en cada una de las clases derivadas (*rectángulo*, *cuadrado* y *triángulo*).

La forma de hacer lo que acabamos de describir consiste en anteponer la palabra clave “*virtual*” en la declaración de la función, *en la clase base*. Luego se define la función en cada una de las clases derivadas, tal y como hemos hecho hasta ahora con todas las funciones. En el siguiente ejemplo puede apreciarse todo este procedimiento.

Observe que en la función principal se definen tres punteros *a la clase base*, y sin embargo haciendo uso de éstos, creamos un objeto de cada una de *las clases derivadas*. Cuando llamamos a las funciones virtuales *calcularArea()* o *aumentar()* no se ejecutan las funciones de la clase base, sino las de la clase derivada:

```
[ cpp/poo/polimorfismo.cpp ]
```

```
#include <iostream>
using namespace std;
```

```
class figura {
public:
    virtual float calcularArea() = 0;
    virtual void aumentar() = 0;
protected:
    float area;
};
```

```

// Clase rectangulo: ladoa y ladob
class rectangulo : public figura {
public:
    int ladoa;
    int ladob;
    float calcularArea() {
        area = ladoa * ladob;
        cout << "El area del rectangulo es: " << area << endl;
    }
    void aumentar() {
        ladoa++;
        ladob++;
    }
    rectangulo(int a, int b) {
        ladoa = a;
        ladob = b;
    }
};

```

```

// Clase cuadrado: lado
class cuadrado : public figura {
public:
    int lado;
    float calcularArea() {
        area = lado * lado;
        cout << "El area del cuadrado es: " << area << endl;
    }
    void aumentar() {
        lado++;
    }
    cuadrado(int l) {
        lado = l;
    }
};

```

```

// Clase triangulo: base y altura
class triangulo : public figura {
public:
    int base;
    int altura;
    float calcularArea() {
        area = base * altura / 2;
        cout << "El area del triangulo es: " << area << endl;
    }
    void aumentar() {
        base++;
        altura++;
    }
};

```

```

    }
    triangulo(int a, int b) {
        base = a;
        altura = b;
    }
};

int main() {
    // Declaramos tres punteros a figura:
    figura *r, *c, *t;
    // Creamos tres objetos, derivados de figura:
    r = new rectangulo(10,20);
    c = new cuadrado(10);
    t = new triangulo(10,19);
    /* Llamando a calcularArea, se ejecuta una funcion diferente
       en cada caso, segun el tipo de objeto */
    r->calcularArea();
    c->calcularArea();
    t->calcularArea();
    cout << "(Aumentamos el triangulo)"<< endl;
    t->aumentar();
    t->calcularArea();
}

```

3 Programación de la interfaz gráfica del usuario (GUI)

En los primeros ordenadores se utilizaba un terminal con una *interfaz de línea de comandos* (CLI, del inglés: *Command Line Interface*) como principal dispositivo de salida. De hecho hasta ahora hemos utilizado una *emulación de terminal* o *consola* para comprobar los programas de ejemplo que hemos visto.

Sin embargo fue la aparición de las *interfaces gráficas de usuario* (GUI) lo que potenció enormemente el desarrollo de la industria del *software*, y ha hecho posible la expansión de su uso, no sólo a nivel industrial y empresarial, sino también a nivel doméstico.

Las GUI sustituyen la interfaz de línea de comandos por una interfaz basada en *mapas de bits*, con objetos *gráficos* (ventanas, botones, iconos, ...) que crean un entorno *amigable*, facilitando así la interacción del usuario con la máquina. Entre las GUI más conocidas están *Microsoft Windows*, *Mac OS* y *X-Window-System* (para *GNU/Linux*).

Para programar una aplicación gráfica con C++ necesitamos utilizar una *biblioteca gráfica*. Como dijimos en el capítulo introductorio, una biblioteca es un conjunto de procedimientos y funciones agrupadas en uno o varios archivos. En el caso de las bibliotecas gráficas, estos procedimientos y funciones sirven para crear una interfaz gráfica de usuario, con distintos tipos de objetos (ventanas, botones, campos de texto, barras de desplazamiento, etiquetas, ...).

Hay entornos de programación integrados, como *Borland Builder C++* o *Microsoft Visual C++*, que incluyen sus propias bibliotecas gráficas, e incluso su propio compilador. El problema principal que presentan estos entornos es que *sólo* funcionan bajo *Microsoft Windows*. También contamos con otras bibliotecas gráficas como GTK+ o Qt-3, que al ser multiplataforma podemos utilizarlas para programar bajo *Windows*, *Mac OS/X*, *GNU/Linux*. . . Además estas bibliotecas son *software libre* y *open source*, por lo que es posible acceder al código fuente y modificarlo [1].

De todas estas opciones hemos elegido la biblioteca gráfica Qt-3 para explicar cómo desarrollar aplicaciones gráficas. Aparte de las ventajas citadas anteriormente, cabe destacar que Qt-3 utiliza C++ de forma nativa, que cuenta con un entorno de desarrollo gráfico muy potente (el *Qt Designer*, cuyo uso explicaremos en este capítulo), y que incluye métodos específicos para acceder a bases de datos, lo cual será muy útil en el Capítulo 5.

Cuando trabajamos con Qt-3, cada *proyecto* debe situarse en una carpeta de trabajo. Antes de compilar los programas utilizaremos una herramienta que viene incluida con la biblioteca Qt-3, llamada “*qmake*”, que preparará nuestro proyecto para ser compilado. Debemos ejecutar la siguiente secuencia de instrucciones:

“*qmake -project*” Crea un fichero con extensión *.pro*, con información sobre el *proyecto* (el proyecto estará formado por todos los ficheros *.cpp* que estén dentro de la carpeta de trabajo, y también los ficheros con elementos gráficos).

“*qmake -makefile*” Crea un archivo llamado *Makefile* que recoge todas las *dependencias* existentes entre las bibliotecas utilizadas.

“*make*” Compila el programa.

3.1. Primera aplicación gráfica: un *Hello World!*

Hay dos maneras de programar una aplicación gráfica:

1. La más primitiva consiste en definir todos los objetos gráficos, su comportamiento, etc. en un fichero de texto plano, como hemos hecho hasta ahora para los programas de línea de comandos.
2. Con la complejidad que puede llegar a tener el código cuando queremos añadir muchos objetos gráficos a una *ventana*, acabaremos optando por utilizar un *entorno de desarrollo*, una aplicación que nos permitirá diseñar la interfaz de forma *visual*. Dicho de otro modo: no tendremos que programar los objetos gráficos mediante código, pero sí tendremos que escribir código para definir su *comportamiento*.

Para comprobar la diferencia entre estas dos formas de programar, haremos una primera aplicación gráfica *Hello World!* que consistirá en una ventana con una etiqueta con el texto “*Hello World!*”. Primero con código, y luego con *Qt Designer*.

En una aplicación gráfica no incluiremos la biblioteca *iostream*, ya que ésta sirve para utilizar los dispositivos de entrada y salida estándar (teclado y pantalla) en línea de comandos. En su lugar incluiremos la biblioteca principal de cualquier aplicación Qt-3, que es *qapplication.h*, y en concreto para nuestro programa queremos una *etiqueta* para escribir el texto citado, así que incluiremos también *qlabel.h*. Observe que todos los elementos gráficos son *objetos*; de ahí la enorme importancia de comprender los conceptos de POO del Capítulo 2.

Observe los comentarios del código del *Hello World!*. Al compilar y ejecutar (podemos ejecutar desde la consola, como hemos hecho hasta ahora, o bien haciendo doble clic sobre el icono de la aplicación) podemos ver una ventana como la que se muestra en la Figura 4:

```
[ cpp/helloqt/helloqt.cpp ]
#include <qapplication.h>
#include <qlabel.h>

int main(int argc, char *argv[]) {
    // Creamos un objeto QApplication (aplicacion grafica Qt-3):
    QApplication app(argc, argv);
    // Creamos un objeto "etiqueta" y definimos el texto que tendra:
    QLabel *label = new QLabel("Hello World!", 0);
    /* Establecemos que el objeto principal de la aplicacion
       sea la etiqueta con el texto "Hello World!": */
    app.setMainWidget(label);
    // Mostramos la etiqueta:
    label->show();
    return app.exec();
}
```

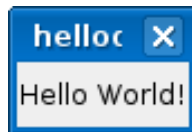


Figura 4: Captura del *Hello World!* creado con Qt-3 mediante código

3.2. Entorno de desarrollo gráfico: *Qt Designer*

En la Figura 1 (Capítulo 1) vimos la apariencia del entorno de desarrollo *Qt Designer*. Vamos a hacer una aplicación *Hello World!*, pero creando la interfaz de manera visual, mediante este entorno de desarrollo. Abrimos *Qt Designer*. Si no nos aparece el diálogo *New File/Project* vamos al menú *File|New...* y seleccionamos la opción *Dialog*. Nos aparece entonces un formulario vacío con el título *Form1*. Reduciremos el tamaño del formulario, y añadiremos una etiqueta (*TextLabel*). Vea la Figura 5.



Figura 5: Diseño del formulario con *Qt Designer*

Para ajustar el tamaño del formulario automáticamente también podemos usar una herramienta de *Qt Designer*, en el menú `Layout|Adjust Size`. Cuando hayamos terminado el diseño de la interfaz gráfica¹¹ guardamos el formulario con el nombre `form1` en una carpeta de trabajo `-helloqtdesigner-` (donde guardaremos todos los ficheros pertenecientes al proyecto). El formato del formulario no es código C++, sino un formato específico denominado *User Interface*, con extensión `“.ui”`, que podemos volver a abrir con *Qt Designer* para modificarlo, etc.

Tenemos en nuestra carpeta de trabajo el formulario guardado. Ahora crearemos un fichero C++ con la función principal del programa, que se encargará de llamar al formulario que acabamos de diseñar. El `qmake` utilizará el fichero `form1.ui` para generar la cabecera del C++ (con extensión `“.h”`) y el fichero principal C++ (con extensión `“.cpp”`):

```
[ cpp/helloqtdesigner/helloqtdesigner.cpp ]
#include <qapplication.h>
// Incluimos la cabecera del formulario form1:
#include <form1.h>

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    // Creamos un objeto Form1, de nombre "miventana":
    Form1 *miventana = new Form1;
    app.setMainWidget(miventana);
    miventana->show();
    return app.exec();
}
```

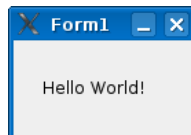


Figura 6: Captura del *Hello World!* creado con *Qt Designer*

En principio podría parecer que es incluso más difícil utilizar *Qt Designer* que programar el código manualmente. Esto es porque la aplicación que hemos creado hasta ahora sólo tiene *un* elemento gráfico: una etiqueta de texto. A medida que añadimos más elementos veremos las siguientes ventajas:

- No es necesario saber la sintaxis de cada uno de los objetos gráficos que queramos incluir (por ejemplo `“QLabel *label = new QLabel(“Hello World!”, 0)”`).
- Además no tenemos que saber en qué biblioteca se hallan (`“<qlabel.h>”`), por tanto sólo tendremos que hacer `“#include”` de los formularios que creamos. Las dependencias¹² se resuelven automáticamente con `qmake`.

¹¹En `Preview|Preview Form` podemos previsualizar el formulario, como si estuviera ejecutándose. Esto es muy útil para realizar comprobaciones.

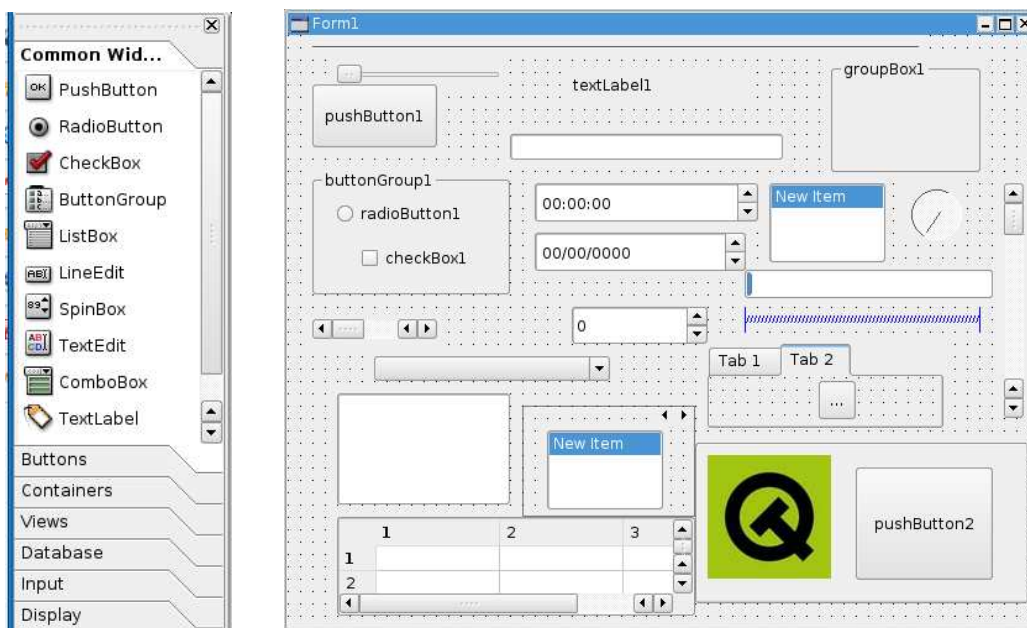
¹²Si en nuestro formulario utilizamos una *etiqueta*, decimos que éste depende con la biblioteca `qlabel.h`. Lo mismo sucede con los demás elementos gráficos.

3.3. Elementos gráficos

En los dos ejemplos gráficos vistos hasta ahora sólo hemos utilizado un elemento gráfico: una etiqueta de texto. Existen muchos otros elementos gráficos que podemos utilizar.

::: *Controles básicos* :::

En la barra lateral izquierda del *Qt Designer* encontraremos una lista (Figura 7(a)) con todos los elementos gráficos que podemos añadir a un formulario. Como primer ejercicio cuando utilizamos un entorno de desarrollo se suele recomendar al usuario practicar y probar, añadir todo tipo de controles, aunque no tengan relación unos con otros. Esta primera experiencia, similar a cuando ofrecemos una caja de lápices a un niño pequeño para que juegue, es casi *necesaria* para conocer el entorno de desarrollo y ver qué posibilidades nos ofrece.



(a) Barra lateral con elementos gráficos

(b) Formulario con varios elementos gráficos

Figura 7: Elementos gráficos en *Qt Designer*

Hemos creado un formulario con diversos elementos, como puede ver en la Figura 7(b). En este apartado sólo nos interesa saber de qué elementos disponemos. Más adelante veremos cómo *personalizar* estos elementos mediante sus *propiedades*, y cómo controlar el comportamiento y los *eventos*.

Para generar el ejecutable procedemos de igual manera que en el caso del *Hello World!*. Si hasta el momento no ha comprendido la ventaja de utilizar el entorno de desarrollo, lo hará ahora mismo:

- Primero guardamos el formulario en una carpeta de trabajo –*controles*– con el nombre `form1.ui`.
- Luego creamos un fichero C++ al que llamaremos `controles.cpp`, y que contendrá el siguiente código.

```
[ cpp/controles/controles.cpp ]
#include <qapplication.h>
#include <form1.h>

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    // Creamos un objeto Form1, de nombre "miventana":
    Form1 *miventana = new Form1;
    app.setMainWidget(miventana);
    miventana->show();
    return app.exec();
}
```

Como habrá intuido, es *exactamente el mismo* código que hemos utilizado para el *Hello World!*, y sin embargo hemos creado una aplicación con muchísimos más elementos gráficos, que si hubiésemos tratado de programar por completo en C++ habría ocupado *cientos* de líneas de código. En adelante no compilaremos todos los programas, para centrarnos más en el desarrollo de la interfaz gráfica.

::: *Estilos predefinidos* :::

En *Qt Designer* contamos con varios estilos de formularios predefinidos para determinados tipos de ventanas que aparecen con frecuencia en los entornos gráficos más conocidos. Cuando creamos un nuevo proyecto (*File|New...*) podemos elegir uno de estos formularios predefinidos. Elegiremos uno u otro en función de lo que queramos hacer. Destacaremos los siguientes:

Dialog Para crear una ventana genérica vacía.

Wizard Para crear una aplicación con formato de *asistente*, similar a los programas de instalación o los asistentes de configuración.

Main Window Es un asistente para crear ventanas con barra de *menú*, de *herramientas* y de *estado*. Es una de las opciones más completas y rápidas que nos aporta *Qt Designer*.

Configuration Dialog Es una ventana de configuración como las que utilizan los programas, con varias pestañas.

Dialog with Buttons Es la clásica ventana de diálogo que tiene algún objeto sencillo y dos botones (Normalmente *Aceptar* y *Cancelar*). Tenemos dos modalidades: los botones en la parte inferior (*Bottom*) o a la derecha (*Right*).

Tab Dialog Ventana con varias pestañas, para uso diverso. También incluye botones de *Aceptar* y *Cancelar* en la parte inferior.

En la Figura 8 se recoge una previsualización de cada una de estas plantillas predefinidas. Antes de crear un formulario conviene ver si hay alguno de estos estilos que nos interese como punto de partida.

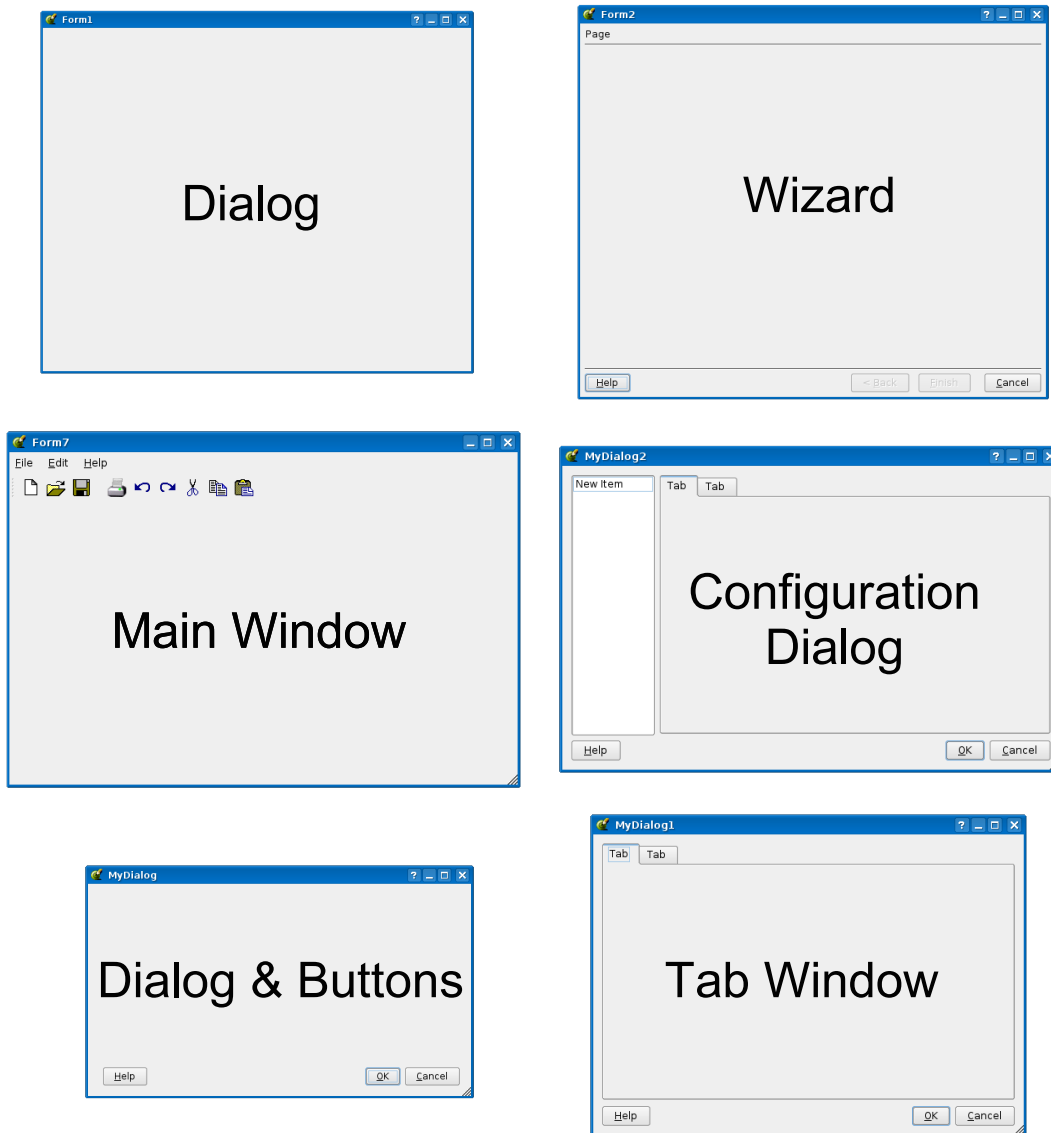


Figura 8: Plantillas de uso común en programación gráfica

3.4. Propiedades y eventos. Conexiones

Ya sabemos crear interfaces *genéricas* y *sencillas* para aplicaciones gráficas. En este apartado trataremos de *personalizar* las aplicaciones. Veremos qué *propiedades* tienen los elementos, y cómo modificarlas para adaptarlos a nuestras necesidades. Nos encargaremos de controlar los *eventos* que se producen, y establecer *conexiones* para que cuando se produzca un determinado evento, se ejecute un fragmento de código.

Entendemos por *evento* el acaecimiento de un *suceso*; algo que cuando ocurre activa la ejecución de un fragmento de código. El evento por excelencia de las aplicaciones gráficas es el *clic* del ratón. Estamos ante un tipo de programación diferente, a la que llamamos *orientada a eventos*, y que difiere sustancialmente de la *programación secuencial* —que utilizábamos en línea de comandos—, en la que los programas ejecutan cada línea de código desde el principio hasta el final. Veremos también en este apartado cómo *conectar* los eventos con el fragmento de código que queremos ejecutar.

Vamos a crear una aplicación gráfica [4] que muestre por pantalla –mediante una etiqueta– un anuncio de texto que habremos introducido previamente. Este programa tan sencillo servirá para comprender el manejo de propiedades, eventos y conexiones. Creamos un nuevo proyecto: File|New...|Dialog. Nos aparece un formulario vacío. En *Qt Designer* tenemos a la derecha una ventana con una lista de *propiedades* (*Properties* – Figura 9) relativas al formulario. Las propiedades nos permiten cambiar todos los atributos del formulario: el nombre, el tamaño, el título, los botones de la barra de título, la fuente, el color de fondo, etc. Modificaremos las siguientes:

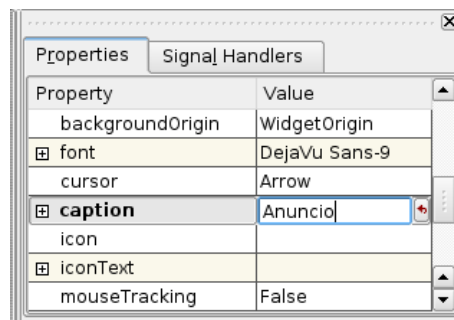
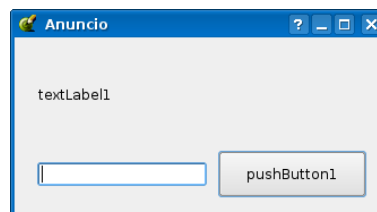


Figura 9: Ventana de propiedades

- **name:** “frmAnuncio”, **caption:** “Anuncio”.

Añadiremos una caja de texto (*LineEdit*), una etiqueta (*TextLabel*) y un botón (*PushButton*), y ajustaremos el tamaño del formulario como se muestra a continuación:



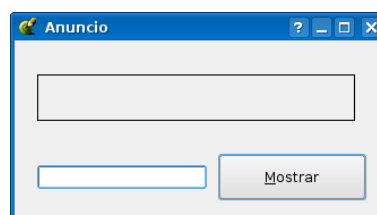
Y modificamos las propiedades, haciendo clic sobre cada elemento:

TextLabel: **name:** “lblAnuncio”, **font:** “(tamaño 14pt)”, **text:** “” (vacío), **hAlign:** “AlignHCenter”, **vAlign:** “AlignVCenter”, **frameShape:** “Box”.

LineEdit: **name:** “txtEntrada”, **text:** “” (vacío).

PushButton: **name:** “btnMostrar”, **text:** “&Mostrar”¹³.

Hemos modificado las *propiedades* de los distintos elementos para adaptarlos a nuestro programa. El resultado es el siguiente:



¹³El símbolo “&” hará que la siguiente letra (la “M”) resulte *subrayada*. Esto automatiza la asignación de teclas rápidas: Alt + M para pulsar el botón, sin necesidad de hacer clic sobre él.

Ahora falta la parte más importante: *hacerlo funcionar*, ya que de momento no hay ninguna acción asociada al evento de *pulsar el botón*. En `Edit|Connections...` crearemos una conexión entre el evento *pulsar el botón* y una función a la que llamaremos `anunciar()`.

La ventana para gestionar conexiones es bastante sencilla. Ajustamos las opciones que se muestran en la Figura 10.

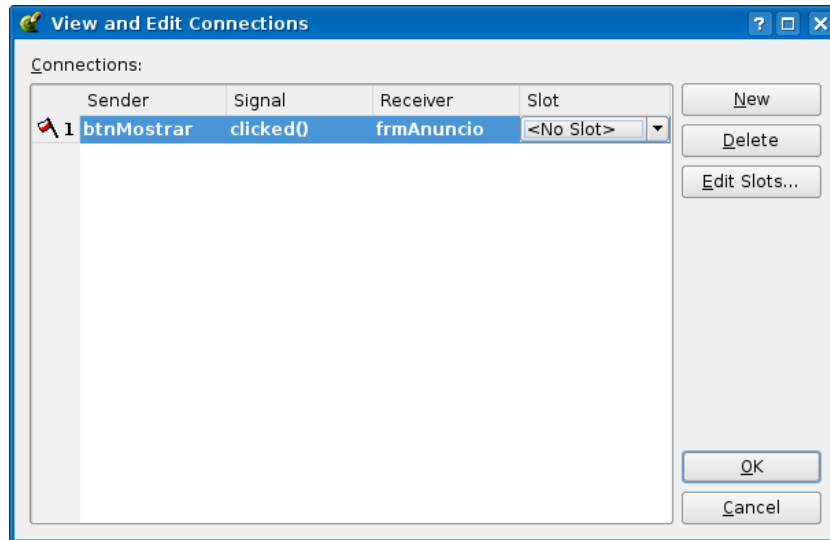


Figura 10: Ventana de conexiones

Observará que no existe la función `anunciar()` en la lista desplegable del campo *Slot*, ya que es un nombre que nos hemos inventado. En estos casos lo que debemos hacer es crear una nueva función en `Edit Slots...`. Le damos al botón *New Function* (Figura 11) y modificamos el nombre. Cuando volvamos a la ventana anterior (Figura 10) aparecerá la función `anunciar()` en la lista del campo *Slot*; la seleccionamos. Una vez creada la *conexión* sólo falta programar el código de la función `anunciar()`.

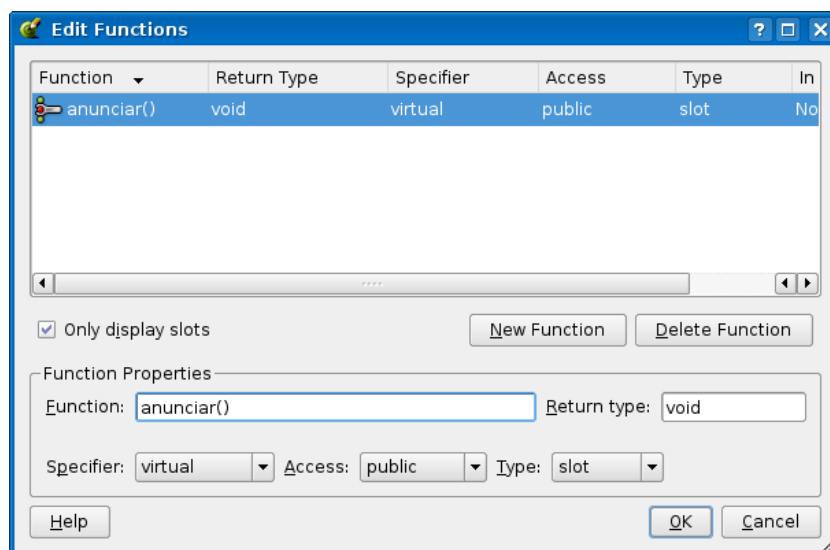


Figura 11: Ventana de funciones

Para ello vamos a la pestaña de los manejadores de eventos (*Signal handlers* - Figura 12) y –habiendo pulsado antes el botón sobre el cual se producirá el evento– hacemos clic en la función `anunciar()`.

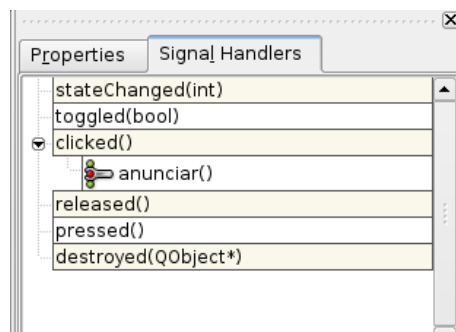


Figura 12: Ventana de eventos

Nos aparecerá una ventana de código como la que se muestra en la Figura 13. Escribamos el siguiente código, que sirve para “Poner en la etiqueta `lblAnuncio` el texto que se encuentre en la caja de texto `txtEntrada`”, como puede intuirse al leerlo:

```
lblAnuncio->setText(txtEntrada->text());
```

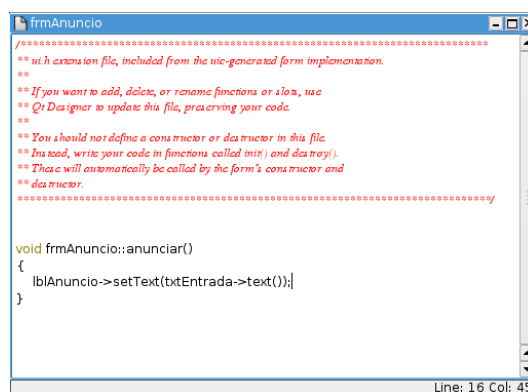


Figura 13: Ventana de edición de código

Compilamos y ejecutamos, y comprobamos que el programa funciona como queríamos (Figura 14).

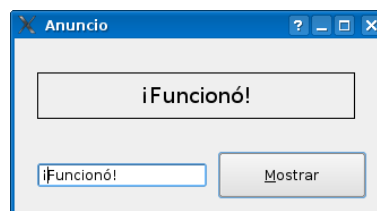


Figura 14: Programa compilado y funcionando

Lo que ha aprendido con estos ejemplos le permitirá crear aplicaciones gráficas sencillas, ajustar las propiedades de los elementos, y controlar el comportamiento.

4 Bases de datos

En este capítulo trataremos de explicar cómo diseñar una *base de datos*, cómo *optimizarla*, y cómo almacenarla en un *sistema de gestión de bases de datos (SGBD)*, donde también quedarán guardaremos los datos que introduzcamos. Utilizaremos *MySQL Administrator* para crear las tablas y gestionar el diseño informático (a esta tarea se le conoce con el nombre de *definición de datos*). Sin embargo para introducir registros o realizar consultas (*manipulación de datos*) utilizaremos sentencias SQL mediante la línea de comandos de MySQL.

4.1. Estudio inicial y diseño

Una de las tareas más importantes a la hora de crear una base de datos es la fase de *diseño* y la *optimización*. La base de datos debe estar bien definida y optimizada “*en el papel*”, antes de proceder a su informatización, ya que de no ser así podemos encontrarnos con problemas graves, que haga que nuestra base de datos sea inoperante.

Las *tablas* en las que almacenaremos los datos deben cumplir unos requisitos en el diseño para que su uso sea efectivo [1]:

- Cada columna debe tener un nombre *único*.
- No puede haber más de dos filas iguales. No se permitirán duplicados.
- Todos los datos en una columna deben ser del mismo tipo.

Teniendo en cuenta estas premisas, analizaremos el problema que se nos presenta y realizaremos un primer esbozo de las tablas que necesitemos. Finalmente, antes de introducir el diseño en el SGBD, *normalizaremos* la base de datos para que éste sea óptimo.

4.2. Formas normales

La manera de *optimizar* una base de datos es mediante las llamadas *formas normales*. Se trata de unas *reglas* de optimización que se aplican *de forma sucesiva*. Cada una de las formas normales está basada en la anterior. Salvo en casos excepcionales debemos siempre utilizar la implementación *óptima*.

Hemos utilizado un ejemplo [5] para observar cómo se aplica el proceso de normalización. Supongamos que inicialmente tenemos toda la información de una *empresa* en una sola tabla con el nombre EMPLEADOS, cuyos campos son los siguientes:

TABLA DE EMPLEADOS

Nombre	Edad	Alojamiento	Propietario	Dirección	Oficio 1	Oficio 2	Oficio 3
--------	------	-------------	-------------	-----------	----------	----------	----------

Quien tuviera que introducir los datos en esta tabla acabaría encontrando inconvenientes, ya que hay campos que quedarían vacíos si el empleado tiene únicamente dos oficios. O peor aún, que tuviera más de tres, en cuyo caso sería imposible almacenar esa información. Además no podríamos contar con información *independiente* sobre cada uno de los campos: para saber los distintos *oficios* posibles tendríamos que observar los tres campos de *oficios* existentes en cada registro. Veamos cómo se van solucionando éste y otros problemas mediante la aplicación de las *formas normales*.

4.2.1. Primera forma normal (1FN)

La primera forma normal (1FN) establece que debemos situar los *grupos repetitivos* en tablas separadas, de modo que exista un sólo campo de cada tipo, y tratando que cada uno de los campos sólo tenga un dato, no una lista de datos. Por tanto no podremos utilizar los tres campos que tenemos para designar *oficios*.

Cada una de las tablas resultantes debe tener una *clave primaria*. La clave primaria es un campo que identificará *de manera única* los registros y servirá para obtener una fila de información. Suponiendo que los *nombres* de los trabajadores son únicos¹⁴, haremos que el campo *nombre* sea la clave primaria. Como en la nueva tabla OFICIO un trabajador puede tener varias filas (si tiene varios oficios) utilizaremos los campos *nombre* y *oficio* de forma conjunta como clave primaria. El resultado en 1FN es el siguiente:

TABLA DE EMPLEADO

Nombre	Edad	Alojamiento	Propietario	Dirección
---------------	------	-------------	-------------	-----------

TABLA DE OFICIO

Nombre	Oficio	Descripción	Calificación
---------------	---------------	-------------	--------------

4.2.2. Segunda forma normal (2FN)

Una base de datos está en segunda forma normal (2FN) si está en 1FN y además sus atributos no principales dependen de forma completa de la clave primaria. Para ello aislaremos los datos que sólo dependen de una parte de la clave. Los campos *oficio* y *descripción* están relacionados entre sí pero no tienen relación directa con los demás campos de la tabla OFICIO, así que los separaremos en una tercera tabla. ¿Qué conseguimos con esto? independizar los conceptos no relacionados, para que si eliminamos todos los registros de empleados que realicen un oficio determinado, dicho oficio –y su descripción– no desaparezca de la base de datos. El resultado en 2FN es el siguiente:

TABLA DE EMPLEADO

Nombre	Edad	Alojamiento	Propietario	Dirección
---------------	------	-------------	-------------	-----------

TABLA DE OFICIO–EMPLEADO

Nombre	Oficio	Calificación
---------------	---------------	--------------

TABLA DE OFICIO

Oficio	Descripción
---------------	--------------------

4.2.3. Tercera forma normal (3FN)

La tercera forma normal (3FN) se da en una base de datos que está en 2FN y elimina las *dependencias transitivas* dentro de una misma tabla. Separaremos todos los elementos que no dependan exclusivamente de la clave primaria. Por ejemplo, a través del *nombre* de un empleado podemos saber dónde se *aloja*, y a través del *alojamiento* sabremos quién es el *propietario* del mismo. Por la propiedad transitiva, podremos deducir el nombre del *propietario* partiendo del *nombre* del empleado. Esta situación *no* debe darse dentro de una misma tabla, por eso separamos los campos en tablas diferentes, hasta eliminar todas las dependencias transitivas. El resultado en 3FN es el siguiente:

¹⁴Cuando en nuestra tabla no tenemos un campo *único* suele emplearse un código de referencia, o un identificador numérico que se autoincrementa al introducir un registro nuevo.

TABLA DE EMPLEADO

Nombre	Edad	Alojamiento
---------------	------	-------------

TABLA DE OFICIO-EMPLEADO

Nombre	Oficio	Calificación
---------------	---------------	--------------

TABLA DE OFICIO

Oficio	Descripción
---------------	--------------------

TABLA DE VIVIENDA

Alojamiento	NombreCompleto	Propietario	Dirección
--------------------	----------------	-------------	-----------

Observe que el campo *alojamiento* ha pasado a ser la clave primaria de la nueva tabla *Vivienda*, y lo hemos utilizado como *identificador* en la tabla EMPLEADO.

Podemos considerar que una normalización 3FN es bastante óptima. Sólo nos queda mencionar dos grados más, que en la mayoría de las ocasiones no añaden ningún cambio a la 3FN, ya que se da en situaciones particulares.

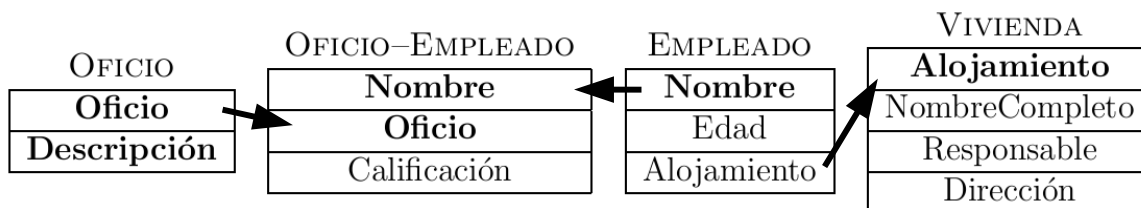
4.2.4. Forma normal de Boyce Codd (BCFN)

Una base de datos estará en forma normal de Boyce Codd (BCFN) si está en 3FN y además no contiene *dependencias realimentadas*. Si en el diagrama de dependencias no hubiese realimentación¹⁵, la BCFN es equivalente a la 3FN, como es el caso del ejemplo que hemos utilizado.

4.2.5. Cuarta forma normal (4FN)

La cuarta forma normal (4FN) resuelve el problema de las *dependencias multivaluadas*. En nuestro ejemplo, a través de un *nombre* podemos averiguar si esa persona tiene *uno o más* oficios... pero no al revés, es decir: a través de una relación de *oficios* no podemos determinar –unívocamente– un *nombre*. Nuestro objetivo por lo tanto es tener relaciones del tipo “*uno a uno*” o “*uno a varios*” pero nunca de “*varios a uno*”. De manera similar que en casos anteriores, diremos que una base de datos está en 4FN si es BCFN y no contiene dependencias multivaluadas.

En el ejemplo que estamos utilizando no hay dependencias realimentadas ni multivaluadas, así que nuestra base de datos, tal y como está, es BCFN y 4FN. A continuación se muestra el *diagrama de relaciones*:



¹⁵La realimentación en el diagrama de dependencias se da cuando un campo depende de otro, y éste a su vez depende del primero. No suele darse este caso.

4.3. Bases de datos con MySQL

Nuestra base de datos está optimizada, y por tanto preparada para ser informatizada. El primer paso es crear un *usuario*. Utilizaremos el administrador de MySQL (*MySQL Administrator*) en la pestaña *User Administration* (Figura 15). Click en *New User*, asignamos un nombre de usuario y una contraseña, y pulsamos *Apply Changes*. Salimos de *MySQL Administrator* y volvemos a entrar con esta identificación.

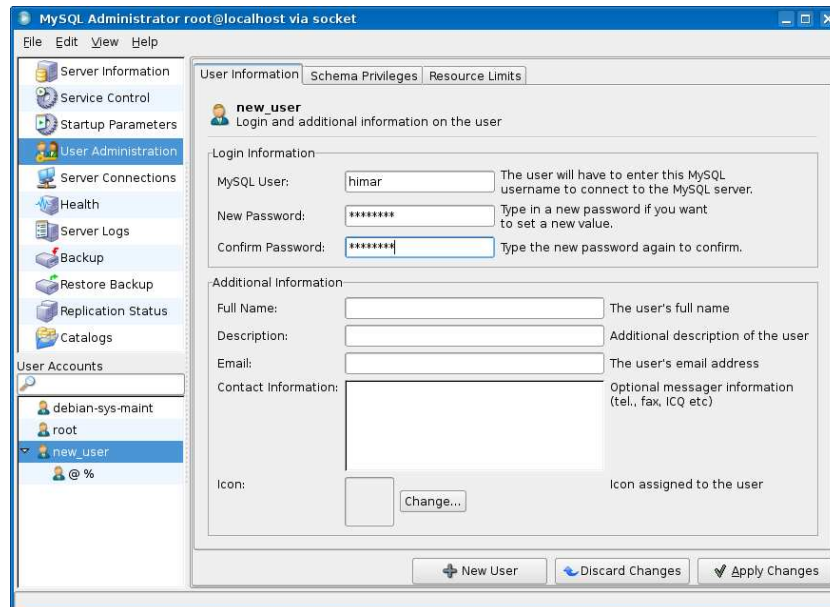


Figura 15: Ventana de gestión de usuarios de *MySQL Administrator*

Con nuestro nuevo usuario vamos a crear una base de datos, a la que llamaremos *empresa*. Para ello vamos a la pestaña *Catalogs*, y en la parte inferior izquierda de la ventana hacemos clic con el botón derecho del ratón: seleccionamos *Create Schema* (Figura 16). Nos aparece un cuadro de diálogo para introducir el nombre: ponemos *empresa*. Ya está creada la base de datos, podemos observar que aparece un nuevo icono con el nombre *empresa* en el cuadro *Schema*.

4.3.1. Definición de datos

La *definición de datos* abarca todos los procesos de crear, eliminar o modificar la *estructura* de la base de datos; no se refiere a su *contenido*. El lenguaje SQL posee unas sentencias para poder llevar a cabo esta tarea (**CREATE** –para crear tablas–, **ALTER** –para modificarlas–, **DROP** –para eliminarlas–). Sin embargo es mucho más sencillo utilizar *MySQL Administrator*, que es precisamente lo que vamos a hacer:

Ejecutamos *MySQL Administrator*, y entramos con el nombre de usuario y contraseña de antes, en el cuadro *Schema* de la pestaña *Catalogs*. Pulsamos el botón *Create Table* y nos aparece una ventana (Figura 17) donde tendremos que escribir los nombres de todos los *campos* de cada una de las tablas. Además tendremos que decidir el *tipo* de datos, y si es o no una *clave primaria*. Cuando acabemos de introducir todos los campos pulsamos *Apply Changes* y nos saldrá un cuadro para pedirnos confirmación. En él podremos ver el código SQL que se va a ejecutar para crear la tabla.

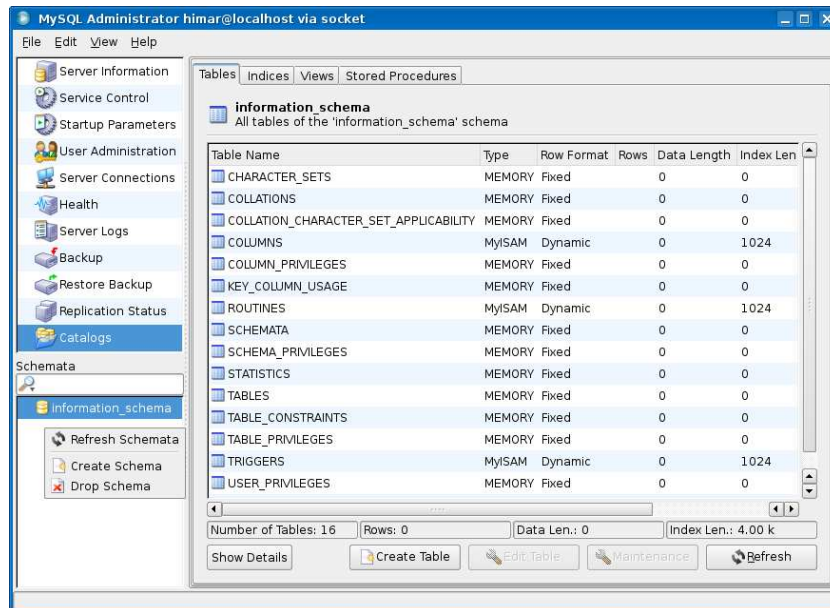


Figura 16: Crear una nueva base de datos

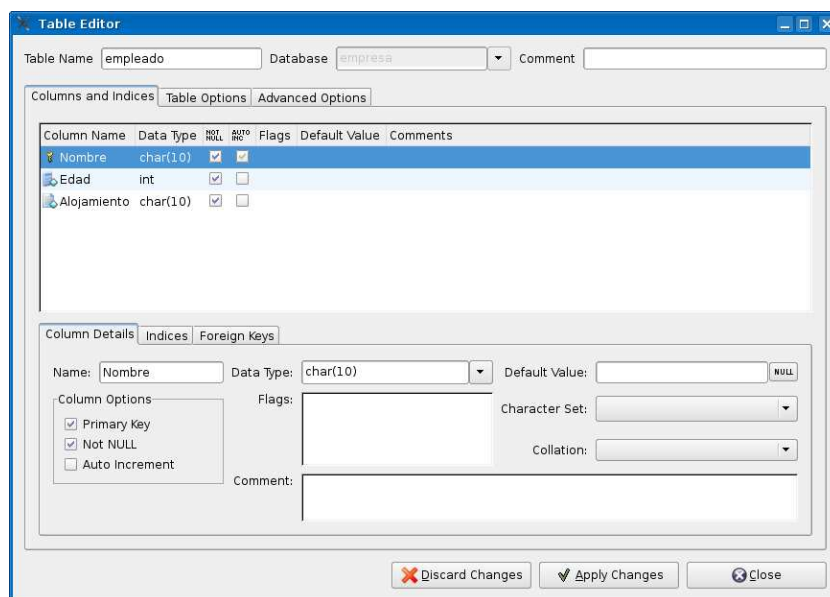


Figura 17: Crear una tabla

Si hubiésemos decidido escribir manualmente el código SQL de las cuatro tablas, nos habría quedado de la siguiente manera (utilice este código para ver el tipo de cada campo):

[sql/empresa/tablas.sql]

```
CREATE TABLE 'empresa'.'empleado' (
  'Nombre' char(30) NOT NULL DEFAULT "",
  'Edad' int NOT NULL DEFAULT 0,
  'Alojamiento' char(10) NOT NULL DEFAULT "",
  PRIMARY KEY('Nombre')
);
```

```
CREATE TABLE 'empresa'.'vivienda' (
  'Alojamiento' char(10) NOT NULL DEFAULT "",
  'NombreCompleto' char(30) NOT NULL DEFAULT "",
  'Responsable' char(10) NOT NULL DEFAULT "",
  'Direccion' char(30) NOT NULL DEFAULT "",
  PRIMARY KEY('Alojamiento')
);
```

```
CREATE TABLE 'empresa'.'oficio-empleado' (
  'Nombre' char(30) NOT NULL DEFAULT "",
  'Oficio' char(10) NOT NULL DEFAULT "",
  'Calificacion' int NOT NULL DEFAULT 0,
  PRIMARY KEY('Nombre', 'Oficio')
);
```

```
CREATE TABLE 'empresa'.'oficio' (
  'Oficio' char(10) NOT NULL DEFAULT "",
  'Descripcion' char(30) NOT NULL DEFAULT "",
  PRIMARY KEY('Oficio', 'Descripcion'),
);
```

Vamos ahora a la línea de comandos, para ver cómo tendríamos que introducir las sentencias SQL necesarias para crear las *relaciones* entre tablas. En una consola escribimos la sentencia “mysql -u *usuario* -A empresa -p”, donde *usuario* es el nombre de usuario configurado. La opción “-p” es para que nos pida la contraseña (puede haber una base de datos *pública* que no necesite contraseña para entrar, pero no es nuestro caso), y la opción “-A” para seleccionar la base de datos a la que queremos acceder. Una vez validados nos aparece el *prompt* de MySQL. Veamos si se han creado las tablas:

```
mysql> show tables;
+-----+
| Tables_in_empresa |
+-----+
| empleado          |
| oficio-empleado   |
| oficio             |
| vivienda          |
+-----+
4 rows in set (0.00 sec)
mysql>
```

Salta a la vista que la interfaz no es tan *sofisticada* como las que hemos creado con Qt-3, pero nos ha permitido ver las tablas de la base de datos, así que nos sirve igualmente. En esta consola es donde introduciremos las consultas que queramos realizar en adelante.

Vamos a empezar por crear las relaciones entre tablas. Recordemos que eran tres relaciones entre los campos principales de las cuatro tablas: *nombre*, *oficio* y *alojamiento*.

A continuación se muestra el código (Utilizamos ALTER porque estamos modificando las tablas que ya están creadas):

```
[ sql/empresa/relaciones.sql ]
ALTER TABLE 'empresa'.'oficio'
  ADD FOREIGN KEY ('Oficio')
  REFERENCES 'oficio-empleado' ('Oficio');
ALTER TABLE 'empresa'.'empleado'
  ADD FOREIGN KEY ('Nombre')
  REFERENCES 'oficio-empleado' ('Nombre');
ALTER TABLE 'empresa'.'empleado'
  ADD FOREIGN KEY ('Alojamiento')
  REFERENCES 'vivienda' ('Alojamiento');
```

Hemos finalizado así el diseño de nuestra base de datos. Por tediosa que pueda parecer esta metodología, es preferible entretenerse en hacer un buen diseño antes que tener que corregirlo cuando haya miles de registros introducidos, con el riesgo de perder información valiosa.

4.3.2. Manipulación de datos

En este apartado estudiaremos cómo utilizar una base de datos ya creada: la *manipulación de datos* consiste en *insertar* registros, *eliminarlos*, y sobre todo *recuperar* información de manera ordenada. Para ello utilizaremos la consola de MySQL.

::: Introduciendo registros :::

Ya que no hemos introducido datos, empecemos por explicar cómo se hace:

```
[ sql/ejemplos/introducir.sql ]
INSERT INTO 'tabla' VALUES ("valor1","valor2","valor3");
```

Estos son los datos que introduciremos en las tablas:

```
[ sql/empresa/introdatos.sql ]
INSERT INTO 'empleado' VALUES ("Himar","21","LPA");
INSERT INTO 'empleado' VALUES ("Marta","23","LPA");
INSERT INTO 'empleado' VALUES ("Juan","21","TAF");
INSERT INTO 'empleado' VALUES ("Luciano","25","TAF");

INSERT INTO 'vivienda' VALUES ("LPA","Residencia Las Palmas",
  "Eladio","C/ León y Castillo");
INSERT INTO 'vivienda' VALUES ("TAF","Residencia Tafira",
  "Francisca","Campus de Tafira");

INSERT INTO 'oficio-empleado' VALUES ("Himar","Inegniero","7");
INSERT INTO 'oficio-empleado' VALUES ("Marta","Arquitecto","7");
INSERT INTO 'oficio-empleado' VALUES ("Juan","Ingeniero","8");
INSERT INTO 'oficio-empleado' VALUES ("Luciano","Profesor","6");

INSERT INTO 'oficio' VALUES ("Ingeniero","Diseño de proyectos");
INSERT INTO 'oficio' VALUES ("Arquitecto","Diseño de viviendas");
INSERT INTO 'oficio' VALUES ("Profesor","Docencia");
```

::: *Modificando datos* :::

Supongamos que hemos cometido un error al introducir datos, o que se da una situación que requiere modificar un registro de la base de datos. Lo haremos de la siguiente manera:

[sql/ejemplos/actualizar.sql]

```
UPDATE 'tabla' SET Campo="valor"WHERE condición
```

En nuestra base de datos hemos puesto que *Juan* es *Ingeniero* y tiene una *calificación* de 8. Sin embargo es *Profesor*, y su *calificación* es de 9. Procedamos a cambiarlo:

[sql/empresa/juanprofe.sql]

```
UPDATE 'oficio-empleado' SET Oficio="Profesor",
    Calificacion="9"WHERE Nombre="Juan";
```

::: *Buscando...* :::

Buscar y mostrar información es la tarea por excelencia en la que podemos sacar todo el potencial del lenguaje SQL [1]. Las consultas pueden hacerse tan complejas como precisas. En general, para buscar utilizamos la siguiente sintaxis:

[sql/ejemplos/buscar1.sql]

```
SELECT Campos FROM 'tabla' WHERE condición ORDER BY Campos
```

Veamos dos ejemplos:

1. Incluso algo tan sencillo como *mostrar el contenido de una tabla* puede interpretarse en términos de *búsqueda*: “Buscar todos los campos de una tabla” Queremos ordenar por nombre, en orden alfabético inverso:

[sql/empresa/mostrartabla.sql]

```
SELECT * FROM empleado ORDER BY Nombre DESC;
```

2. Se desea mostrar el nombre y la profesión de los empleados que tengan una calificación igual o superior a 7. Además la lista debe estar ordenada alfabéticamente por nombres:

[sql/empresa/califica.sql]

```
SELECT Nombre,Oficio FROM 'oficio-empleado'
    WHERE calificacion>=7 ORDER BY Nombre;
```

No tiene sentido seguir añadiendo ejemplos, ya que los criterios de búsqueda suelen ser muy particulares. Cada consulta tiene su propia estructura.

::: *Eliminando registros* :::

Para buscar el registro (o los registros) que queramos eliminar se utiliza también la palabra clave “WHERE”, así que el filtro de búsqueda se aplica igual que cuando utilizamos “SELECT”. La sintaxis es:

[sql/ejemplos/borrar.sql]

```
DELETE FROM 'tabla' WHERE condición
```

Juan ha tenido que irse de la empresa, así que vamos a borrar su ficha:

[sql/empresa/juanseva.sql]

```
DELETE FROM 'empleado' WHERE Nombre="Juan";
```

::: Resumen de sentencias SQL :::

Para finalizar con el apartado de SQL disponemos de una tabla-resumen de las sentencias SQL más importantes:

- Definición de datos:

CREATE Crear una tabla.

DROP Eliminar una tabla.

ALTER Modificar una tabla, o alguna de sus propiedades.

- Manipulación de datos:

INSERT Añadir un registro a una tabla.

UPDATE Modificar un registro de una tabla.

SELECT Buscar información en una tabla.

DELETE Eliminar un registro de una tabla.

5 Acceso a bases de datos con C++

En este capítulo toca mezclar lo que hemos aprendido de programación gráfica y de bases de datos. Lógicamente es mucho más cómodo trabajar con una base de datos desde una *interfaz gráfica*, que utilizando la línea de comandos de MySQL.

Vamos a crear una nueva aplicación gráfica con Qt-3: File|New...|Dialog. En la pestaña de controles con el nombre *Database* tenemos tres elementos que nos permiten acceder a bases de datos, y nos muestran el resultado de forma gráfica. También nos permiten *añadir*, *borrar*, o incluso *filtrar* los registros, introduciendo una consulta SQL en la propiedad *Filter* de estos elementos. Veamos las particularidades de cada uno de ellos:

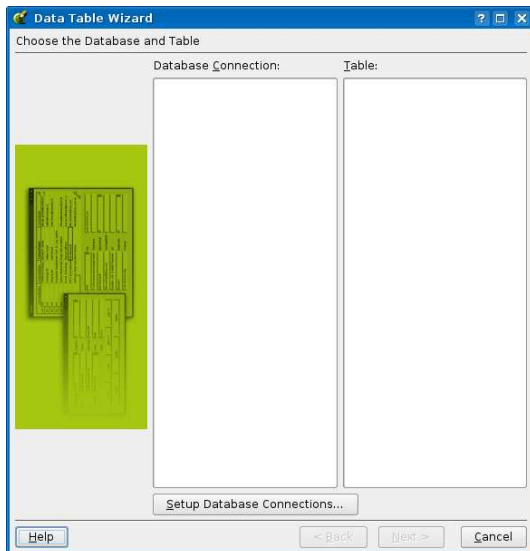
Data Table Muestra una tabla (la que indiquemos en tiempo de diseño –mediante el asistente–, o bien en tiempo de ejecución –modificando la propiedad *database*–). Utilizando el botón derecho del ratón podemos insertar un nuevo registro.

Data Browser Muestra sólo un registro de la tabla, pero incluye botones de *navegación* para movernos entre registros.

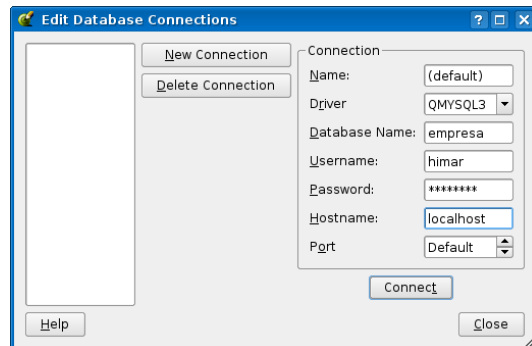
Data View Muestra un solo registro de la tabla. Este control se suele utilizar para modificar un campo en concreto, o bien para mostrar el resultado de una búsqueda.

::: Configuración de los elementos mediante el asistente :::

En nuestro ejemplo añadiremos tres controles, uno de cada tipo. Nada más incluirlos en el formulario, nos aparecerá un *asistente* para configurar la conexión entre el elemento y la base de datos. Ajustaremos las opciones que se muestran en la Figura 18(b). Se pueden utilizar varios *drivers* para acceder al SGBD, en función del servidor que utilicemos; en el caso de MySQL utilizaremos *QMYSQL3*.



(a) Conexión con la base de datos



(b) Configurando la conexión

Figura 18: Asistente de conexión con la base de datos

Cada uno de estos controles nos permite visualizar una tabla. Nosotros seleccionamos la tabla *empleado*. En la siguiente ventana (*Next*), podemos elegir los campos de la tabla que queremos mostrar (Figura 19).

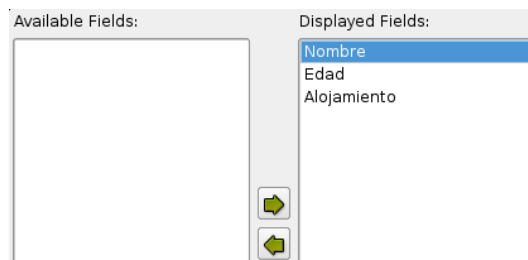


Figura 19: Mostraremos todos los campos de la tabla

Luego se nos pregunta qué acciones deberá confirmar el usuario antes de realizar, por ejemplo para evitar errores (Figura 20). También podemos elegir si queremos que la tabla sea de sólo lectura (*Read-only*) para nuestro control.

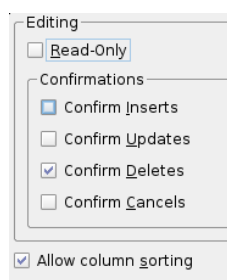


Figura 20: Seleccionamos para qué acciones queremos pedir confirmación al usuario

Finalmente se nos da la opción de elegir un filtro de búsqueda. En este campo podemos utilizar una sentencia SQL, del estilo “WHERE”. No es necesario introducirla en este momento, ya que más adelante podemos modificar la propiedad *Filter* mediante el código.

Llegados a este punto, el control está listo para ser utilizado. No es necesario compilar el programa para comprobar el funcionamiento. Basta con *previsualizar* el formulario (Preview|Preview Form). En la Figura 21 se puede ver el resultado.

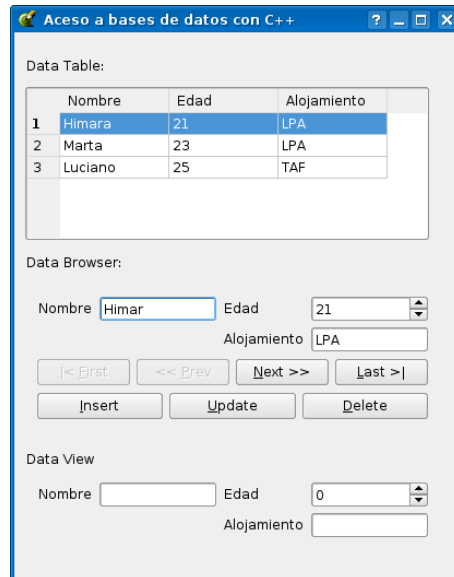


Figura 21: Acceso a bases de datos con C++

El acceso a bases de datos desde C++ no solo permite disponer de una interfaz gráfica amigable. De hecho no es esa la finalidad que se persigue con el uso de los controles que hemos visto en este capítulo. El objetivo fundamental es combinar la funcionalidad de *cualquier* tipo de programa con el acceso a bases de datos.

Referencias

- [1] Wikipedia (Enciclopedia Libre):
<http://es.wikipedia.org/wiki/Portada>
- [2] Curso de C++:
<http://c.conclase.net/>
- [3] *C++ GUI Programming with Qt-3*
Jasmin Blanchette, Mark Summerfield
Prentice-Hall, 2004
- [4] *Programación en C++ con Qt bajo entorno GNU/Linux*
Martín Sande
<http://www.gulca.com.ar>
- [5] *MySQL para Windows y Linux*
César Pérez
Ra-Ma Editorial, 2004