

## Borland C++ Builder . EL ENTORNO DE TRABAJO.

**C++Builder** es un entorno para el desarrollo rápido de aplicaciones (RAD, Rapid Application Development) para Windows. Mediante esta herramienta de programación podremos realizar aplicaciones Win32 de consola (tipo DOS), de interfaz gráfica de usuario (GUI, Graphical User Interface) o cliente-servidor con una gran rapidez.

El entorno de desarrollo de C++Builder (**IDE, Integrated Development Environment**) se invoca localizando en el escritorio de Windows o a través del menú Inicio el icono correspondiente denominado C++Builder y haciendo click en él. Este entorno consta de ventanas de distinto tamaño y posición. A diferencia de otras aplicaciones Windows, estas ventanas no ocupan todo el escritorio, sino sólo el espacio necesario, dejando libre el resto. Cada una de estas ventanas posee un propósito en particular dentro del programa y lo más habitual es tener abiertas sólo las que se precise en cada momento.

El IDE consta de las herramientas necesarias para diseñar, implementar, ejecutar y depurar una aplicación, mediante una serie de elementos que operan de forma integrada y complementaria: un Editor de Código, un Depurador de Errores, una Barra de Herramientas, Herramientas de Bases de Datos, etc.

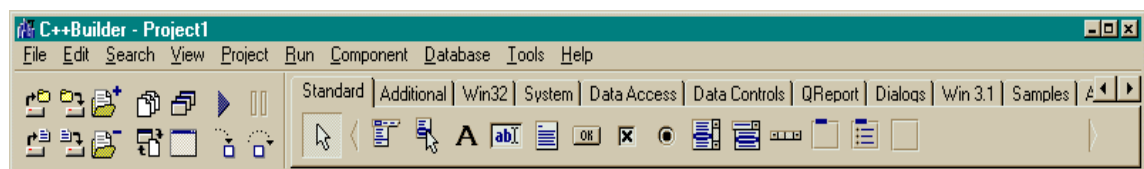
El IDE consta de.

La **Ventana Principal**, con el Menú, la Barra de Herramientas de acceso inmediato y la Paleta de Componentes

El **área de trabajo**, donde introduciremos el código fuente de nuestra aplicación, además de desarrollar de forma visual su interfaz de usuario mediante fichas (formularios) y el Inspector de Objetos, con el que modificaremos las propiedades y los sucesos de los objetos de nuestra aplicación.

### Ventana principal

En la parte superior de la pantalla, aparece una ventana que contiene el nombre del proyecto, los botones de acción rápida, la Paleta de Componentes y el menú principal. A esta ventana se le denomina **ventana principal**. Cerrándola estaremos saliendo de C++ Builder. La ventana principal se divide en:



#### - El menú principal.

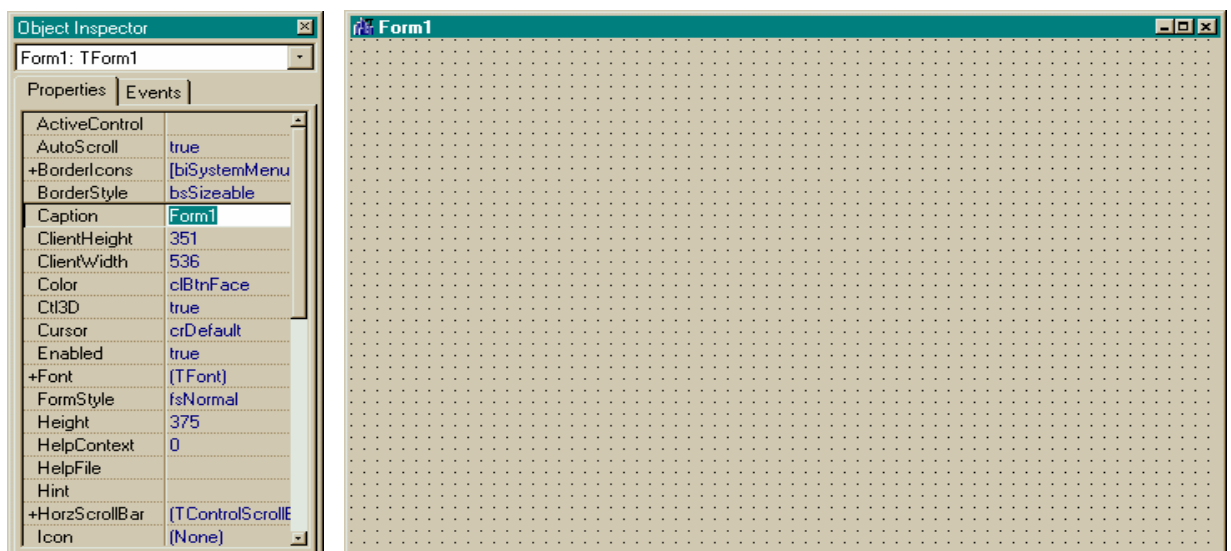
- **File.** Permite crear, abrir, salvar, cerrar e imprimir los elementos que forman una aplicación, además de abandonar el entorno.
- **Edit.** Agrupa las funciones típicas de edición como deshacer, rehacer, copiar, pegar y borrar. Además existen funciones de ayuda en la colocación de los elementos de la interfaz, como alinear con la cuadrícula, traer al frente, enviar al fondo, ...
- **Search.** Comprende todas las funciones típicas de búsqueda de texto en las ventanas del área de trabajo.
- **View.** Permite seleccionar las ventanas que desean visualizarse.
- **Project.** Gestiona los elementos del proyecto o aplicación y genera su programa asociado

(compilación y linkado).

- **Run.** Contiene las funciones que nos permiten tanto ejecutar como depurar la aplicación.
  - **Component.** Contiene una serie de utilidades que nos facilitan la gestión de los componentes VCL (Visual Component Library), así como la configuración de la Paleta de componentes de la Ventana Principal.
  - **Database.** Contiene comandos que permiten crear, modificar y visualizar Bases de Datos.
  - **Tools.** Para visualizar y cambiar las opciones de entorno, modificar la lista de programas, crear y editar imágenes, etc. Permite, por tanto, añadir y configurar herramientas auxiliares.
  - **Help.** Contiene información sobre C++Builder, guías de programación, información sobre componentes VCL, etc.
- Los **botones de acción rápida.** Son iconos que realizan de forma directa, sin necesidad de acceder a los menús desplegables, las funciones más utilizadas durante el desarrollo de una aplicación. Es totalmente configurable (pulsando con el botón derecho y accediendo a Properties) y, por defecto, presenta las acciones más comunes para crear y depurar un programa, como abrir, salvar, ejecutar, detener (pausa), ejecutar línea a línea, etc.
  - La **Paleta de Componentes.** Está compuesta por múltiples páginas a las que accederemos mediante las pestañas que hay en la parte superior. Cada una de las páginas contiene componentes que son los elementos básicos con los que construiremos los programas. Esta paleta la usaremos cuando desarrollemos programas para Windows.

### El área de trabajo

En el área de trabajo nos encontramos con dos tipos de ventanas: las de elaboración de interfaz en caso de desarrollo de una aplicación Windows (Ficha o Formulario e Inspector de Objetos) y las de implementación de la aplicación en código fuente C++ mediante un editor, para todo tipo de aplicaciones (Editor de Código). Ahora, solamente vamos a realizar pequeñas aplicaciones en **modo consola**, por tanto, las dos primeras ventanas que aparecen a continuación, no serán usadas, aunque se explica su propósito:

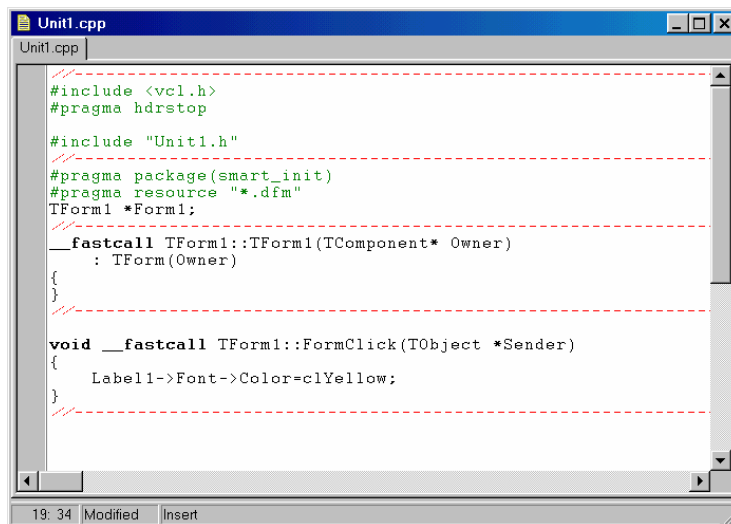


- La **ficha** o “formulario” es la ventana que aparece aproximadamente en el centro de la pantalla y que tiene

por título `Form1`. Una ficha es una ventana que en una aplicación C++ Builder actúa como contenedor, en la cual se irán insertando componentes que serán necesarios para solicitar datos, mostrar información o realizar cualquier tarea. Durante la ejecución una ficha se comportará como cualquier otra ventana Windows, es decir, se mostrará con botones de minimizar, maximizar o cerrar, un título y un área de trabajo con los controles con los que el usuario interactuará. En realidad todos estos elementos serán configurables.

- El **Inspector de Objetos** es la ventana que inicialmente aparece en la parte izquierda de la pantalla. Consta de 3 elementos básicos: la lista de componentes de que consta la aplicación, la página de propiedades y la página de eventos. La lista desplegable de componentes contiene el nombre de cada componente que existe en la ficha y el tipo de éste. Por ejemplo `Form1:TForm1`. Los componentes son elementos genéricos que sirven para cualquier aplicación por lo que permiten su personalización. La página *Properties* muestra las propiedades del componente que tengamos seleccionado, así como los valores que actualmente contienen. Alterando el valor de una de estas propiedades podremos, por ejemplo, modificar el título de la ficha, conseguir que no aparezcan los botones de minimizar, maximizar, etc. La página *Events* es la que contiene los nombres de los eventos de que dispone el componente, así como los procedimientos que se ejecutarán al generarse esos eventos. Por ejemplo, mediante un evento podremos indicar la acción a realizar cuando se haga clic en el ratón sobre un botón.

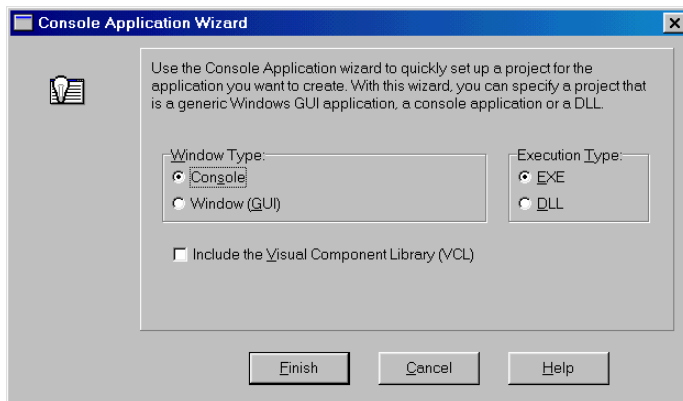
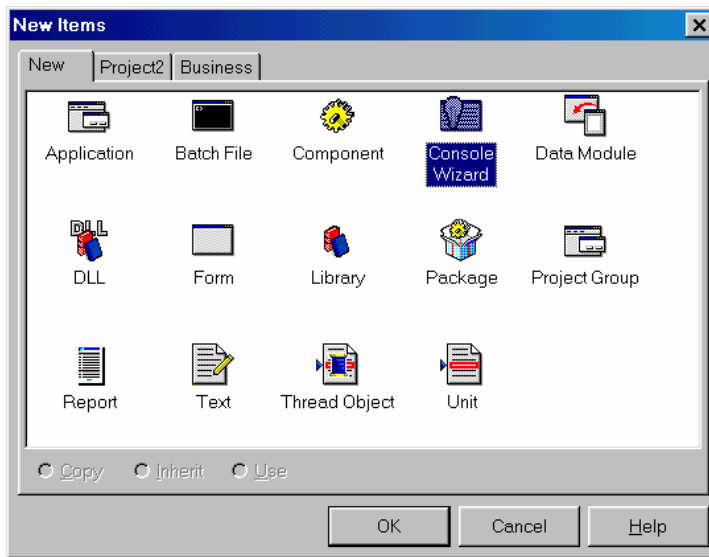
La ventana que sí usaremos siempre para introducir (editar) el programa fuente es la que aparece a continuación:



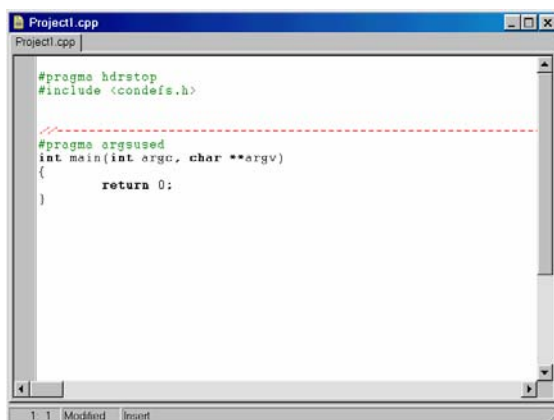
- Es el **Editor de Código**, que permite introducir el código fuente que definirá el funcionamiento de nuestra aplicación. Posee unas características que facilitan la labor de introducción del código fuente, como son el marcado de sintaxis a color, la capacidad de deshacer a diferentes niveles, la integración con el depurador, etc.

## Creación de un programa en modo consola.

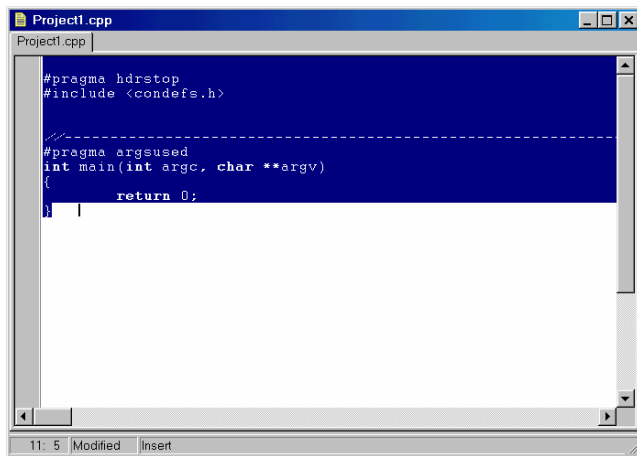
Para crear un programa en modo consola debemos acceder al menú **File/New**, que presentará un cuadro de diálogo donde elegiremos el tipo de programa que queremos generar: **Console Wizard**:



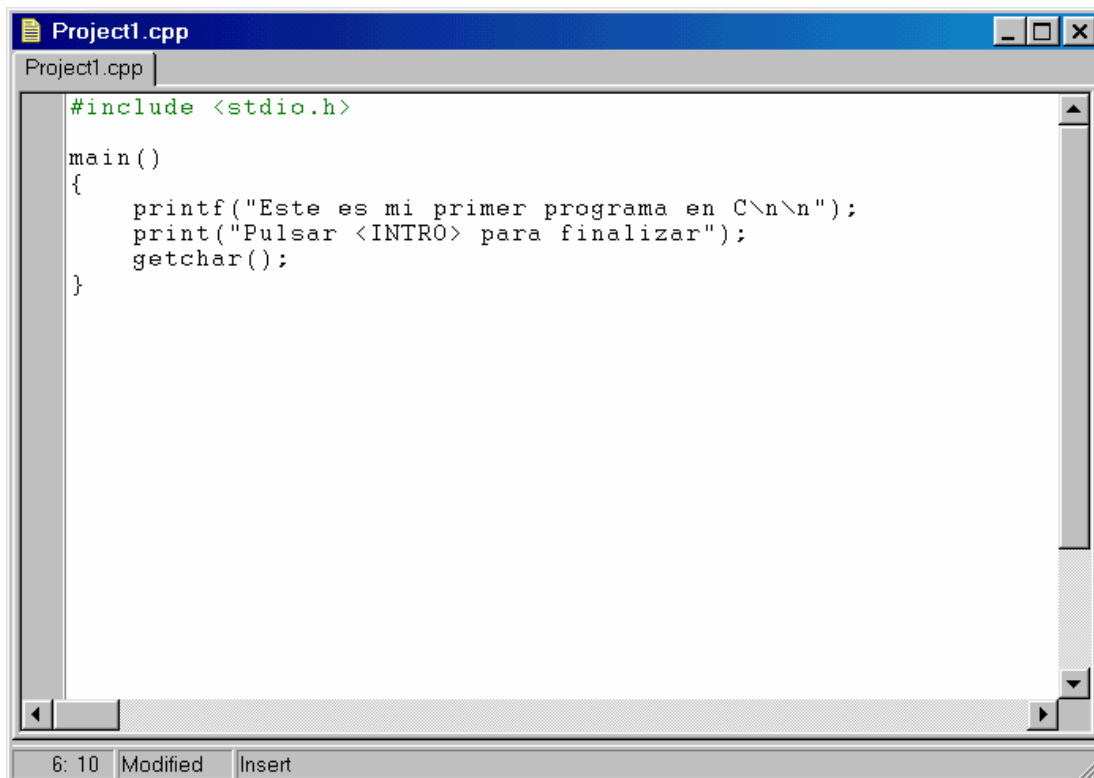
A continuación aparecerá una ventana donde escribiremos el código de nuestro programa fuente. En este momento es posible cerrar la ventana correspondiente al Inspector de Objetos (Object Inspector), ya que nuestro programa no va a contar con elementos visuales (componentes).



Por defecto, aparece ya el código básico asociado a un programa en modo consola. Este texto podemos eliminarlo y partir de cero para crear nuestro primer programa, seleccionándolo (con el teclado o el ratón) y pulsando la tecla de suprimir.



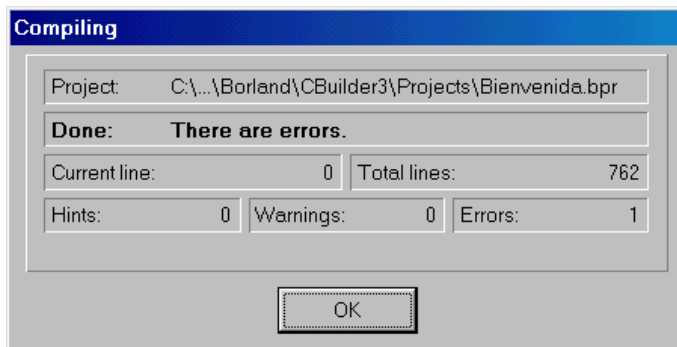
En el editor de código habrá que introducir lo siguiente:



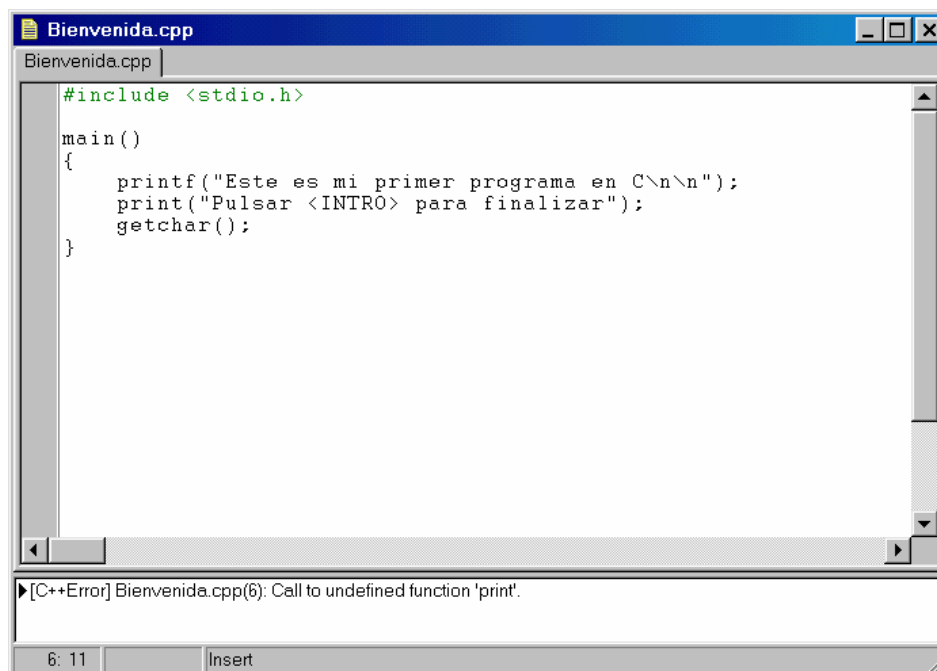
El nombre de nuestro programa, por defecto, es **Project1.cpp**. La extensión **cpp** indica que es un fichero con código fuente en Borland C++ Builder. Para cambiar el nombre de nuestro programa bastará acceder a la opción **Save** del menú **File**, donde nos solicitan el nombre que queremos dar a nuestro programa. Lo llamaremos **Bienvenida.cpp** y lo almacenaremos en la carpeta *Archivos de programa\Borland\CBUILDER3\Projects*.

Una vez escrito nuestro programa, podemos compilarlo para comprobar que no tiene errores de compilación. En el ejemplo, la función **printf()** se ha escrito como **print()**, por lo que el compilador no la reconocerá. Para compilar un programa hay que pulsar en el menú **Project** la opción **Make** <nombre de programa> o pulsar directamente **Ctrl+F9**.

El resultado de esta opción es una ventana informativa que indica el resultado del proceso de compilación, avisando de los errores y cuáles de ellos son simples avisos o consejos (**Hints**-señales o insinuaciones de que algo no es del todo correcto-, **Warnings** – avisos-) y cuántos son verdaderamente errores fatales (**Errors**). Sin la depuración de estos últimos no podremos probar la ejecución de nuestro programa, puesto que un proceso de compilación que detecte errores no genera código objeto.



En caso de error, aparecerá señalada la línea donde se ha producido el error y, justo debajo de la ventana de edición de código, una ventana indicando el tipo de error producido.




El error deberá ser solventado modificando el código fuente. Una vez que los errores de compilación están solucionados y que el proceso de compilación nos indica que el proceso no ha tenido problemas (Done: Make) podemos ejecutar nuestro programa.

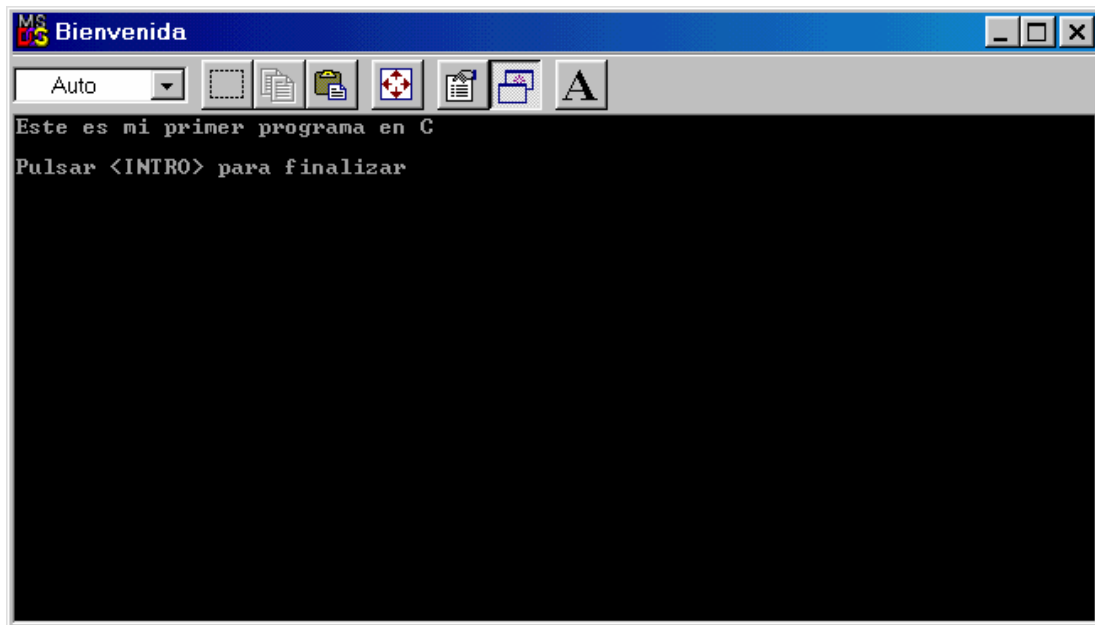
Todo el proceso anterior genera, en la carpeta donde tengamos guardado el programa fuente, el siguiente conjunto de ficheros:

 Bienvenida.~bp	4 KB	Archivo ~BP
 Bienvenida	4 KB	C++Builder 3 Project File
 Bienvenida.cpp	1 KB	Archivo CPP
 Bienvenida	55 KB	Aplicación
 Bienvenida.obj	5 KB	Archivo OBJ
 Bienvenida	128 KB	Archivo TDS

donde el fichero que aparece con el icono de MS-DOS corresponde al programa **.exe** que hemos generado.

Para ejecutar el programa bastará con pulsar sobre éste fuera del entorno de desarrollo o, dentro de este entorno, pulsar la opción de menú **Run/Run**, pulsar **F9** o pulsar sobre el icono  de los botones de acción rápida.

La ejecución de nuestro programa dará paso a una ventana en modo MS-DOS con la siguiente apariencia:



## Creación de la interfaz de un programa bajo Windows.

La creación de una aplicación con C++ Builder se inicia, generalmente, con el diseño de su interfaz, para lo que será necesario manipular la ficha o fichas con que cuenta el proyecto, insertando y manipulando componentes, propiedades y eventos, así como escribiendo el código que sea necesario. Los pasos que seguiremos generalmente para construir un programa son los siguientes:

### 1) Configuración de la Ficha (Ventana del programa)

Toda aplicación Windows de tipo estándar cuenta con una ventana en la que se visualiza o solicita información. En un programa Windows desarrollado con C++ Builder las ventanas de la aplicación son las fichas, cuyas características estableceremos mientras estamos construyendo el programa, es decir, en la denominada **fase de diseño**.

El aspecto de una ficha durante el diseño no tiene por qué ser precisamente el aspecto que tendrá la ventana cuando el programa se ejecute. Por ejemplo, durante el diseño se muestra en el interior de la ficha una matriz o **rejilla de puntos** cuya finalidad es facilitar la alineación de los componentes en su interior. Esta matriz no será visible durante la ejecución. Su aspecto es totalmente configurable desde la opción **Tools** del menú principal en el apartado **Environment Options-Preferences**, donde

Display grid	Activa/Desactiva la rejilla de puntos
Grid size X	Distancia horizontal entre dos puntos consecutivos
Grid size Y	Distancia vertical entre dos puntos consecutivos
Snap to grid	Si se desea que al mover un componente esta distancia sea la unidad mínima de trabajo o no

Inicialmente, las dimensiones de la ventana serán las mismas que hayamos dado a la ficha durante el diseño, aunque este aspecto depende del valor que asignemos a ciertas propiedades. También con el ratón podemos modificar estas propiedades de tamaño y situación inicial.

### 2) Inserción y manipulación de componentes

Una parte muy importante del desarrollo de una aplicación con C++ Builder consiste en insertar en la ficha los componentes apropiados, situándolos adecuadamente y estableciendo las propiedades que correspondan. Todos estos pasos se completarán con la escritura de código asociado a eventos.

Para insertar un componente lo primero que hay que hacer es seleccionarlo en la paleta de componentes, dentro de la página a la que pertenezca. La selección del componente se efectúa simplemente marcando con el ratón el icono que lo representa, posicionando el ratón en el lugar que queramos de la ficha y pulsando el botón izquierdo de éste.

El nuevo componente tomará unas dimensiones por defecto que, posteriormente, podremos modificar desplazándolo o modificando su forma con el ratón o mediante sus propiedades.

### 3) Modificación de propiedades

Algunas propiedades se modifican de forma indirecta cuando manipulamos un control en la ficha, alterando, por ejemplo, su posición. Sin embargo, la mayor parte de las propiedades habrá que editarlas directamente sirviéndonos del **Inspector de Objetos**.

Cada vez que en la ficha se selecciona un componente, automáticamente el Inspector de Objetos muestra su nombre y sus propiedades y eventos.



Existen varios tipos de propiedades: a las que se les da directamente un valor (p.e. `Caption`), las que permiten seleccionar una lista desplegable (p.e. `Enabled`) y las que permiten usar una ventana específica para su edición (p.e. `Icon`). Por último otras propiedades son las que se denominan de conjunto y cuentan con un carácter '+' a su izquierda. Pulsando este signo es posible desplegar un conjunto de subpropiedades (p.e. `Font`).

#### 4) Uso de los eventos y Edición de código

El código de un programa Windows no se ejecuta de forma secuencial sino que lo hace según los mensajes que recibe del propio sistema. Estos mensajes se generan, por ejemplo, al pulsar un botón, introducir un carácter, etc. Estos mensajes son gestionados en C++ Builder automáticamente por cada componente y traducidos en eventos. Nosotros podemos asociar un método a un evento, de tal forma que cuando dicho evento se produzca se ejecute el código de ese método.

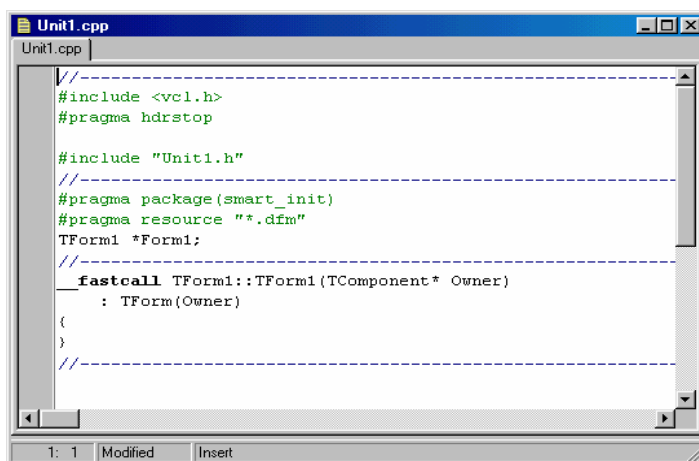
En la página `Events` del Inspector de Objetos veremos los nombres de los eventos con los que cuenta el componente que tengamos seleccionado en ese momento en la ficha. A la derecha de cada uno de los eventos puede existir el nombre de un método, que será el que se ejecute en el momento en el que se genere el evento. Haciendo doble clic sobre la columna derecha de un evento abriremos automáticamente la ventana de código situando el cursor en el método adecuado.

La mayor parte de un programa C++ Builder estará siempre asociado a algún evento, y será el mismo C++ Builder el que se encargará de establecer el nombre de los métodos, determinar los parámetros necesarios y sus tipos, de forma que nosotros solamente deberemos introducir las sentencias que deseemos ejecutar.

El código de un programa se estructura en lo que se denominan módulos, que son ficheros de texto conteniendo código. Cada una de las fichas de un proyecto tiene asociado un módulo, pudiendo además existir módulos independientes que nosotros añadamos para definir funciones, procedimientos u objetos.

La edición se realiza en la ventana del **Editor de Código**, que aparece normalmente bajo la ficha en la que se está trabajando. Para pasar de la ficha al editor de código o viceversa se pulsa <F12> o se pincha con el ratón.

El editor de código es multiarchivo, lo que quiere decir que es capaz de mantener abiertos múltiples módulos de código de forma simultánea.



#### 5) Ejecución.

El fin último de todo el proceso que se está describiendo es conseguir un programa que sea posible compilar y ejecutar. Para ello pulsaremos <F9> o accederemos a **Run** desde el menú principal o desde los botones de acceso rápido.

## Borland C++ Builder . BIBLIOTECA DE COMPONENTES VISUALES (VCL).

### Introducción

La **Biblioteca de Componentes Visuales** o **VCL** (Visual Component Library) está formada por objetos prefabricados, también denominados componentes. Cualquier componente que se incluya en una aplicación pasa a formar parte de su ejecutable. El uso de componentes visuales permite ahorrar tiempo en la codificación y en las pruebas de una aplicación, además de ofrecer la posibilidad de darle un aspecto visual.

Un componente es, desde nuestro punto de vista, un elemento de nuestro programa que posee características (propiedades) y acciones (métodos) asociadas y que reaccionará o no ante una situación producida en el entorno (evento). Los distintos **tipos** de componentes (botón, formulario, cuadro de diálogo, menú, etiqueta, etc...) son facilitados al programador mediante la utilización de la VCL. Cada tipo de componente ofrecerá un conjunto de propiedades, métodos y eventos predefinidos.

Un componente, por tanto, está asociado a **métodos** (código de programa correspondiente a un procedimiento al que se puede acceder desde el programa), y en muchos casos a **propiedades** y **eventos**.

Las **propiedades** representan los datos contenidos en el objeto.

Los **métodos** son las acciones que el objeto puede realizar.

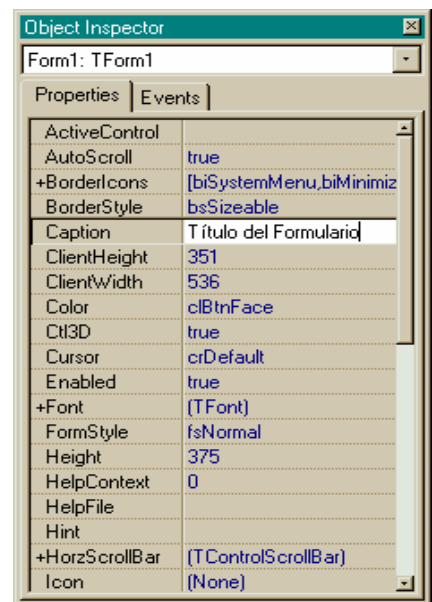
Los **eventos** son condicionantes ante los cuales el objeto puede reaccionar.

Un componente posee siempre un nombre (Name) que lo diferencia del resto de componentes de una aplicación. Por defecto, el C++ Builder asigna un valor que tiene relación con el tipo de componente y un número que los diferencie del resto de componentes.

Si comprobamos el Inspector de Objetos de un formulario o ficha del programa veremos que tiene asignado el nombre `Form1` puesto que su tipo es `TForm1` y es el primer formulario que usamos en nuestro programa. Este nombre, **que puede ser modificado (aunque no es aconsejable)**, nos indica qué tipo de componente es y lleva asociado un número correspondiente al orden en el que se insertó el componente tras todos aquellos del mismo tipo. Por ejemplo, si en una aplicación contamos con un formulario, dos botones y una etiqueta, los nombres que reciben estos componentes serán `Form1`, `Button1`, `Button2` y `Label1`.

El nombre del componente se usará en la edición de código para acceder a sus propiedades, métodos y eventos. Para ello se utiliza el operador `->` (operador de cualificación) de la forma.

Nombre del componente `->` Propiedad | Método | Evento



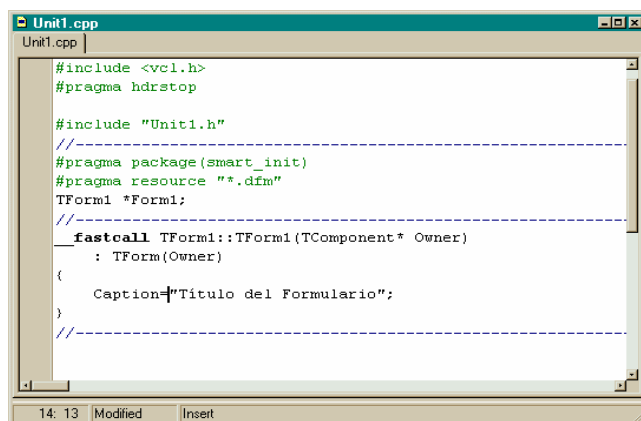
A través de la ayuda podremos conocer todas las propiedades, eventos y métodos asociados a cada uno de los componentes. Pulsando la tecla de función `<F1>` accederemos a la Ayuda de Borland C++ Builder. Si previamente a esta acción tenemos activo uno de los componentes con los que estamos trabajando, se accederá a la página de la ayuda correspondiente a ese componente.

## Generalidades comunes a la mayoría de los componentes VCL

### 1. Propiedades.

Una propiedad es como una variable ligada al componente que posee un valor que condicionará el funcionamiento de éste. Algunas propiedades están disponibles en **fase de diseño** y otras no, es decir, que podremos modificar su contenido mediante el Inspector de Objetos o lo deberemos hacer en **fase de ejecución**.

Por ejemplo, para modificar el título de un formulario se puede asignar el valor directamente a la propiedad `Caption` desde el Inspector de Objetos o modificarla desde código cuando ocurra un evento o cuando se inicie la aplicación. El código asociado se incluiría en el código del programa como muestra la figura:



Donde aparece, la línea de código

```
Caption="Titulo del formulario";
```

Que modificará el contenido de la propiedad en tiempo de ejecución.

Si tuviéramos que hacer referencia a una propiedad de un componente desde un procedimiento no asociado a ese componente deberíamos referenciar la propiedad con el operador `->` de la forma:

```
Form1->Caption="Titulo del formulario";
```

En la tabla siguiente se muestran algunas propiedades comunes a la mayor parte de los componentes visuales.

Para alterar en el componente ...	La propiedad es ...	Y almacena ...
Posición	Left	Un número entero (columna donde aparecerá el componente)
	Top	Un número entero (fila donde aparecerá el componente)
Dimensiones	Width	Un número entero (ancho del componente)
	Height	Un número entero (alto del componente)
Título	Caption	Una cadena de caracteres (título del componente)
Color de fondo	Color	Un valor predeterminado dentro de un conjunto de valores (color de fondo del componente)
Aspecto del Texto	Font	Valores numéricos o predeterminados para las subpropiedades (color, altura, tipo de fuente, tipo de espaciado, tamaño en puntos y estilo del texto).  Para controles que cuentan con <code>Caption</code> o <code>Text</code> .
Estado visual	Visible	True/false (si se quiere que el componente sea visible o no)
Estado de Acceso	Enabled	True/false (si el control estará accesible para el usuario del programa o no)

Acceso al control	TabOrder	Un número entero entre 0 y el número de controles de ese tipo que haya en el formulario menos uno.  (El número indicará el orden en el que se accederá a ese control cuando se pulse la tecla <Tab>, comenzando por el que tenga el valor 0 en esta propiedad).
	TabStop	True/false (Si su valor es Falso excluimos al control del acceso mencionado anteriormente).

## 2. Eventos.

Un evento es una situación que se produce durante la ejecución de un programa. Los eventos pueden provenir del sistema, del ratón, desde teclado, etc....

Un componente podrá reaccionar ante un evento si el tipo de componente lo tiene así predefinido, dependiendo del objetivo para el que está diseñado. Por ejemplo, un componente `TForm` (formulario o ficha) tiene, por supuesto, un evento asociado a la acción de cerrar (Sistema), en cambio, el componente `TButton` (botón) no lo tendrá.

Cuando ocurre un suceso durante la ejecución de nuestro programa, los componentes sobre los que se produzca reaccionarán o no a este suceso si disponen de un evento que lo detecte. Cuando queramos incluir un conjunto de sentencias que se ejecuten cuando se produzca un determinado suceso deberemos asociar un conjunto de instrucciones al evento correspondiente del componente. Este grupo de acciones se expresará en el código del programa.

C++Builder generará automáticamente la base principal del código que se va asociar al evento, pudiendo escribir como bloque de instrucciones las sentencias que deseemos. Por ejemplo, si pulsamos en el Inspector de Objetos del componente formulario (`Form1`) sobre el evento `OnClose` (que se produce cuando se cierra el formulario) aparecerá el editor de código con las líneas:

```
void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction &Action)
{
}
```

```
//-----
```

Donde podemos insertar, por ejemplo, la instrucción

```
ShowMessage("Adiós");
```

Que hará que al cerrar el formulario (evento `OnClose`) aparezca en pantalla un cuadro de mensaje con el texto "Adiós" y el botón de OK.

Los eventos más comunes son:

### **OnClick.**

Se produce al realizarse una pulsación, que puede venir del ratón o del teclado, es decir:

1. Cuando se pulsa el botón izquierdo del ratón.
2. Cuando se elige una opción de menú.

3. Cuando se pulsa la barra espaciadora.
4. Cuando se pulsa la tecla <Intro> sobre un botón.

#### Eventos de ratón.

Cada vez que se mueve el ratón o se pulsa o libera uno de sus botones se produce un evento.

Evento	Cuándo afecta al control
OnMouseMove	Cuando el puntero del ratón se está desplazando sobre él
OnMouseDown	Cuando se pulsa un botón cualquiera del ratón
OnMouseUp	Cuando se libera un botón cualquiera del ratón
OnDblClick	Cuando se hace doble clic con el botón izquierdo

#### Eventos de teclado.

Algunos controles tienen la capacidad de recibir eventos desde teclado, como un formulario (TForm) o un control de edición TEdit.

Cada vez que se pulsa una tecla que corresponde directamente con un carácter (letra, número, signo de puntuación, <Intro>, etc.) se genera un evento OnKeyPress. La función que asociemos a este evento recibirá un parámetro de tipo char & llamado Key, conteniendo el carácter pulsado.

Existen muchas teclas que no reciben un evento OnKeyPress puesto que no corresponden a un carácter (por ejemplo, <Insert> o <Supr>). Estas teclas, al igual que las anteriores, siempre generan un evento OnKeyDown al ser pulsadas y otro OnKeyUp al ser liberadas. El uso de estos eventos se hará más adelante.

Existen muchos métodos más, como los asociados a operaciones de arrastrar y soltar, por ejemplo, además de los específicos de cada componente. Estos métodos se irán estudiando mediante ejemplos.

#### Otros eventos.

Otros eventos son generados por el sistema y se producirán, por ejemplo, para informarnos del estado de la impresora, de la hora, de una situación de error, etc.

### 3. Métodos.

Un método es una acción que actúa sobre un determinado componente. Básicamente un método es una función que recibirá datos o no y devolverá resultados o no. Los más comunes, existentes en prácticamente todos los componentes, son:

Método	Qué acción realiza
Show()	Muestra el control (componente)
Hide()	Oculto el control
CanFocus()	Conoce si el control puede tomar el foco o no (Devolverá true o false)

SetFocus()	Hace activo el control que deseemos (si puede tomar el foco)
------------	--

## Proyectos en C++ Builder

Un proyecto en C++ Builder consta de varios módulos relacionados. Cada módulo se almacena en un fichero distinto. Los ficheros más comunes que aparecen en un proyecto son:

<b>Configuración del Proyecto</b>	<Nombre_del_Proyecto>.BPR
<b>Código Fuente del Proyecto</b>	<Nombre_del_Proyecto>.CPP
<b>Código Fuente de las unidades</b>	<Nombre_de_la_Unidad>.CPP
<b>Ficheros de Cabecera</b>	<Nombre_de_la_Unidad>.H
<b>Formularios</b>	<Nombre_del_Proyecto>.DFM
<b>Recursos</b>	<Nombre_del_Proyecto>.RES
<b>Código Objeto</b>	<Nombre_de_la_Unidad>.OBJ
<b>Código Ejecutable</b>	<Nombre_del_Proyecto>.EXE

Los ficheros con extensión .CPP y .H son los que normalmente podremos modificar puesto que estarán compuestos de líneas de código en lenguaje C++. Los demás serán creados y mantenidos por el propio C++ Builder.

A continuación se muestra un ejemplo del contenido de los ficheros fuente PROYECT1.CPP, UNIT1.CPP y UNIT1.H de un programa que solamente presenta un formulario en pantalla y presenta, al cierre del formulario un mensaje de despedida.

```

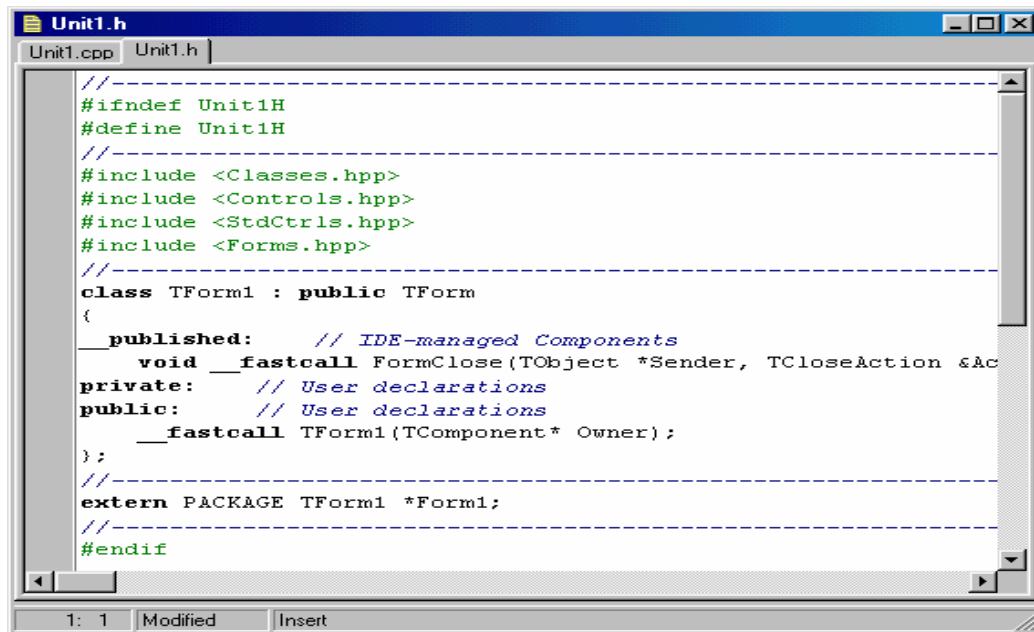
Unit1.cpp
Unit1.cpp
//-----
#include <vcl.h>
#pragma hdrstop

#include "Unit1.h"
//-----
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//-----
__fastcall TForm1::TForm1(TComponent* Owner)
: TForm(Owner)
{
}
//-----

void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction &Action)
{
    ShowMessage("¡ Hasta otra !");
}
//-----
19: 33 Modified Insert

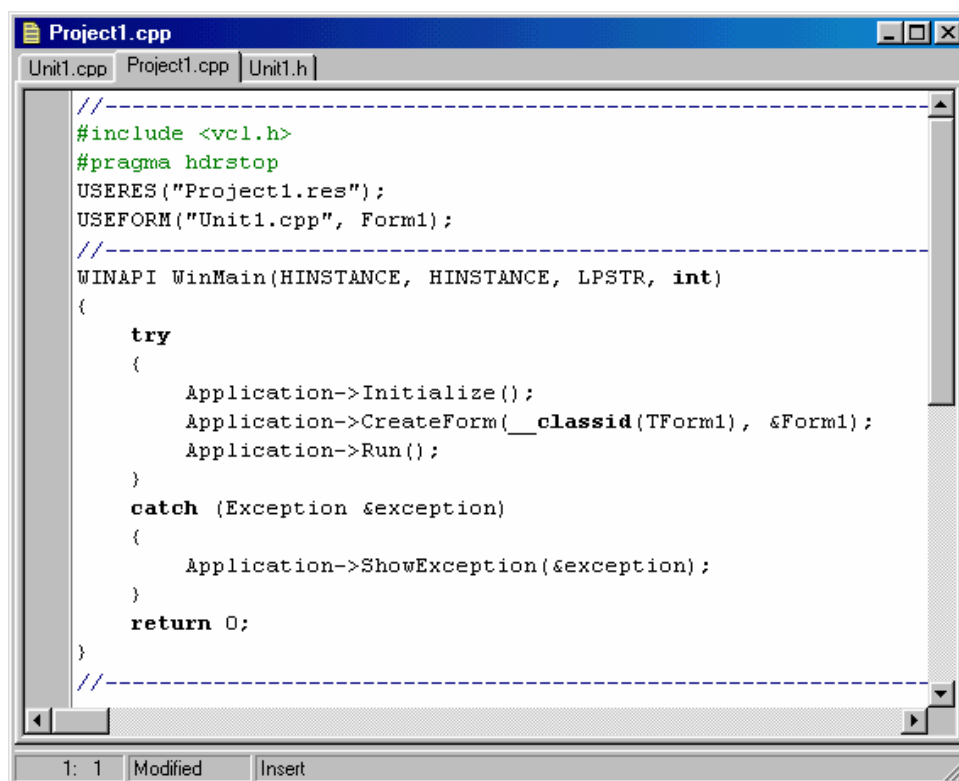
```

Pulsando con el botón derecho sobre el editor de código de la unidad podremos acceder mediante la opción Open Source/Header File a su fichero de cabecera:



```
//-----  
#ifndef Unit1H  
#define Unit1H  
//-----  
#include <Classes.hpp>  
#include <Controls.hpp>  
#include <StdCtrls.hpp>  
#include <Forms.hpp>  
//-----  
class TForm1 : public TForm  
{  
    published:    // IDE-managed Components  
        void __fastcall FormClose(TObject *Sender, TCloseAction &Action);  
private:    // User declarations  
public:    // User declarations  
        __fastcall TForm1(TComponent* Owner);  
};  
//-----  
extern PACKAGE TForm1 *Form1;  
//-----  
#endif
```

Y accediendo en el menú View a la opción Project Source accederemos al código fuente del proyecto:



```
//-----  
#include <vcl.h>  
#pragma hdrstop  
USERES("Project1.res");  
USEFORM("Unit1.cpp", Form1);  
//-----  
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)  
{  
    try  
    {  
        Application->Initialize();  
        Application->CreateForm(__classid(TForm1), &Form1);  
        Application->Run();  
    }  
    catch (Exception &exception)  
    {  
        Application->ShowException(&exception);  
    }  
    return 0;  
}  
//-----
```

## EJERCICIO PRÁCTICO 1. Trabajar con el Formulario.

Un formulario es uno de los componentes más habituales en una aplicación porque representa la ventana en la que se desarrollará la acción del programa. No es un componente que sea necesario insertar, puesto que forma parte del proyecto.



Los pasos que vamos a seguir nos irá introduciendo en el conocimiento de las propiedades de un formulario, los eventos asociados y los métodos con los que cuenta.

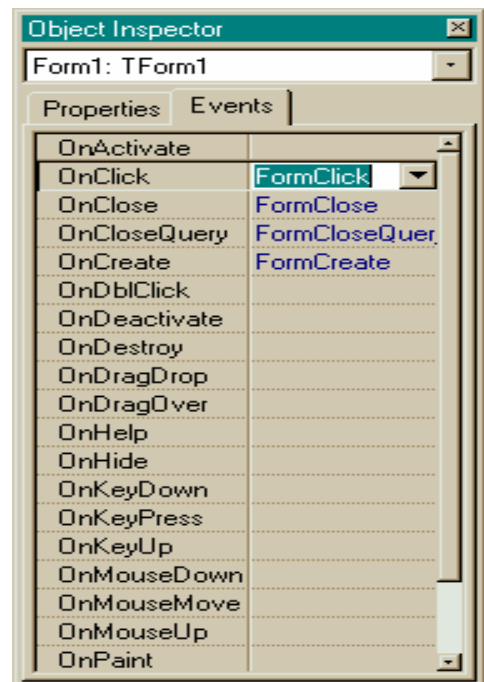
Sobre el Object Inspector, realizar:

1. Pulse sobre la propiedad Caption para cambiar el título del formulario. Escriba “Colores del parchís”.
2. Pulse sobre el conjunto de propiedades BorderIcons y cambie a false los valores de biMinimize y biMaximize para eliminar las opciones de maximizar y minimizar que aparecen por defecto en la barra de título. Aunque sigan apareciendo en la fase de diseño no lo harán en la fase de ejecución.
3. Asigne a la propiedad BorderStyle el valor bsSingle, para que la ventana no sea dimensionable (modificable en su tamaño).
4. Asigne a la propiedad Position el valor psScreenCenter, para que la ventana siempre aparezca centrada en pantalla.
5. Asigne a la propiedad Cursor el valor crHandPoint, para que cuando el puntero del ratón esté situado sobre el formulario cambie su apariencia.
6. Modifique con el ratón la dimensión fija que va a tener el formulario.
7. Mediante el botón de acceso rápido Toggle Form/Unit acceda a la ventana de código y escriba detrás de TForm \*Form1; la línea int ncolor; con la que definiremos una variable de tipo entero en nuestro programa. Esta variable tomará los valores 1, 2, 3 ó 4.
8. Haga doble clic en la página de eventos del formulario sobre OnCreate y escriba el siguiente código:

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    ncolor=1;

    Form1->Color=clRed;
}
```

Obsérvese que tan sólo hay que escribir lo que está resaltado, puesto que la cabecera de la función y las llaves las genera automáticamente el C++Builder. El hecho de anteponer a la propiedad Color el componente al que corresponde, en este caso, no sería necesario, puesto que la función escrita corresponde al propio formulario.





9. Haga doble clic sobre el evento `OnCloseQuery`, que se producirá cuando se solicite el cierre del formulario, y escriba el siguiente código:

```
void __fastcall TForm1::FormCloseQuery(TObject *Sender, bool &CanClose)
{
    if (Application->MessageBox("¿Seguro?", "Cerrar", MB_YESNO) == ID_NO)
        CanClose=false;
}
```

De lo que se trata es de preguntar al usuario si quiere o no abandonar el programa, evitándolo si su respuesta es “no”. Para ello usamos el método `MessageBox()` del objeto `Application`, que representa a nuestra aplicación y es creada automáticamente cuando se inicia el programa.

El método `MessageBox()` presenta una ventana con una serie de botones y devuelve un valor indicando el botón que pulsó el usuario. En nuestro caso hemos puesto una ventana con dos botones (“Si” y “No”), asignando al parámetro `CanClose` el valor `false` si la respuesta es “no”. Las distintas ventanas que puede usar este método se irán conociendo mediante ejemplos.

10. Por último, haga doble clic sobre el evento `OnClick` para hacer que cada vez que se pulse el ratón aparezca el fondo del formulario de uno de los colores del parchís. Para ello escriba el código:

```
void __fastcall TForm1::FormClick(TObject *Sender)
{
    switch (ncolor)
    {
        case 1:Color=clBlue; //Form1->Color=clBlue;
            break;

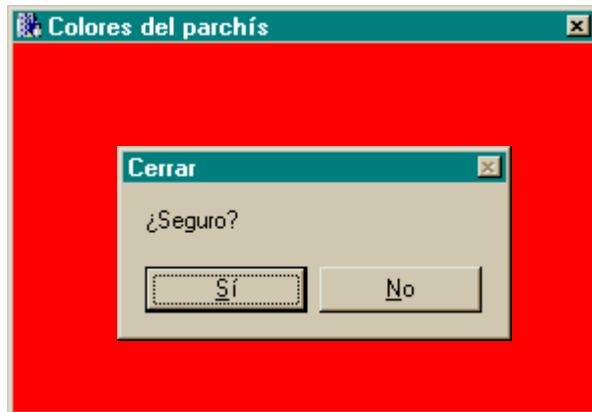
        case 2:Color=clYellow;
            break;

        case 3:Color=clGreen;
            break;

        default:Color=clRed;
            ncolor=0;
    }

    ncolor++;
}
```

Guardar el programa con el nombre `PE1.BPR` (Proyecto) y `E1.CPP` (Unidad). Ya basta depurar los posibles errores de compilación, linkado y ejecución para ejecutar el programa. La ventana de nuestro programa tendrá la siguiente apariencia al solicitar su cierre:

**Ejercicios a realizar.**

- Basándonos en el ejercicio anterior, añadir el código necesario para que aparezca una ventana de despedida de la aplicación con el texto “Adiós”.
- Basándonos en el ejercicio anterior, realizar pruebas, accediendo a la ayuda <F1>, con las propiedades:

BorderIcons	HorzScrollBar	Height
BorderStyle	VertScrollBar	Left
FormStyle	Caption	Name
Icon	Cursor	Top
Visible	Enabled	Width
AutoScroll	Font	

## Borland C++ Builder. COMPONENTES DE LA PALETA STANDARD

Los componentes que podemos insertar en un formulario, como ya hemos visto, aparecen en la Paleta de Componentes del entorno de trabajo. Esta paleta está dividida en varias páginas que permiten la agrupación de componentes según su finalidad.

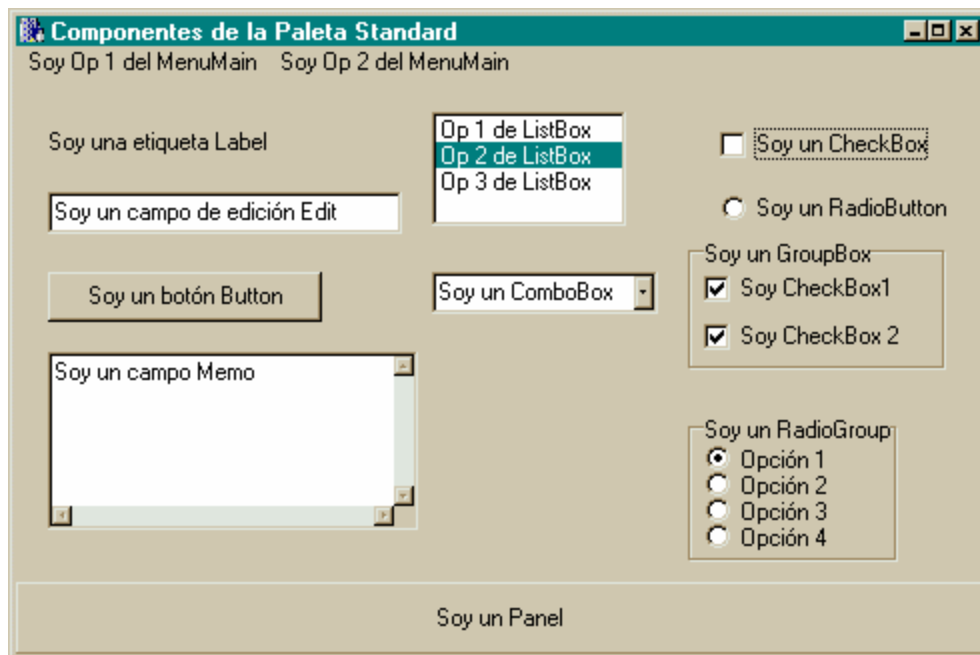
Los componentes pueden ser **visuales** y **no visuales**. Los visuales tienen prácticamente el mismo aspecto al diseñar la ficha que al ejecutarla (un botón, una etiqueta, etc...), mientras que los no visuales no se ven en tiempo de ejecución. Un ejemplo de componente no visual lo encontramos en `Timer` de la página `System`, que en fase de diseño ofrece una representación visual de la función que los temporizadores proporcionan a la función, mientras que en tiempo de ejecución el temporizador mide los intervalos pero no es un elemento visible de la interfaz.

Empezaremos estudiando los componentes de la página `Standard` de la Paleta de Componentes. Estos componentes gestionan los elementos de control más comunes (estándar) de Windows:

Componente	Visual (Sí/No)	Sirve para
<b>MainMenu</b>	N	Diseñar una barra de menús y sus correspondientes menús desplegables.
<b>PopupMenu</b>	N	Diseñar menús emergentes (locales) que queden disponibles en fichas y controles cuando el usuario seleccione el componente y haga clic en el botón derecho del ratón.
<b>Label</b>	S	Crear un control sin ventana que muestre texto no accesible para el usuario, como los títulos. Suele tratarse de texto que actúa como etiqueta de control.
<b>Edit</b>	S	Presentar un área donde el usuario pueda introducir o modificar una sola línea de texto (cuadro de edición).
<b>Memo</b>	S	Presentar un área donde el usuario pueda introducir o modificar varias líneas de texto (cuadro memo).
<b>Button</b>	S	Crear un botón que los usuarios puedan seleccionar para realizar una operación (por ejemplo, interrumpir, iniciar o cancelar un proceso).
<b>CheckBox</b>	S	Crear una casilla de verificación, que representan alternativas del tipo Sí/No, Verdadero/Falso o Activado/Desactivado. Las casillas de verificación son independientes entre sí, ya que las opciones que representan no son mutuamente exclusivas.
<b>RadioButton</b>	S	Crear botones de radio, que representan opciones mutuamente exclusivas. Estos botones suelen utilizarse en combinación con cuadros de grupo para formar conjuntos donde sólo está disponible una de las opciones a la vez.
<b>ListBox</b>	S	Mostrar una lista de opciones de las cuales el usuario puede seleccionar una o varias (cuadro de lista).
<b>ComboBox</b>	S	Crear un cuadro combinado, que reúne la funcionalidad de un cuadro de edición y uno de lista, para presentar una lista de opciones. Los usuarios pueden introducir el texto en el cuadro de edición o seleccionar una opción de la lista.

Componente	Visual (Sí/No)	Sirve para
<b>ScrollBar</b>	S	Crear una barra de desplazamiento para poder desplazar en pantalla la parte visible de una lista o ficha, o trasladarse por un rango en incrementos.
<b>GroupBox</b>	S	Crear un cuadro de grupo para agrupar componentes relacionados en una ficha.
<b>RadioGroup</b>	S	Agrupar botones de radio en una ficha.
<b>Panel</b>	S	Agrupar en un panel otros componentes, como botones rápidos en una barra de herramientas. También se emplea para crear barras de estado.

En la pantalla siguiente aparecen ejemplos del aspecto que en tiempo de ejecución tienen todos los componentes de la paleta Standard que pueden incluirse en un formulario:



## EJERCICIO PRÁCTICO 2. Trabajar con Botones y Etiquetas.

Un botón **-Button-** es un área rectangular tridimensional con un título en su interior. Este título lo almacena la propiedad `Caption`.

Un botón puede contar con una tecla de acceso rápido que, utilizada con la tecla `<Alt>`, permite pulsar el botón sin necesidad de desplazarse a él. Esta tecla será uno de los caracteres que forman parte del título, al que se le antepone el carácter `&`, apareciendo subrayado.

En un formulario pueden existir varios botones. De todos ellos dos pueden ser activados de una forma especial que consiste en pulsar la tecla `<Intro>` o la tecla `<Esc>`. Si la propiedad `Default` tiene el valor `true` el botón actuará como botón por defecto, es decir, que se activará cuando se pulse la tecla `<Intro>`. Si `Cancel` está a `true`, el botón será el botón de cancelación (tecla `<Esc>`). Por supuesto, no puede haber más de un botón con estas propiedades a `true` dentro del mismo formulario.

El evento principal de un botón es `OnClick` (evento de pulsación). Lo normal es que se asocie un método al evento `OnClick` de un botón, ejecutando el código que proceda en el momento en que el usuario lo pulse.

El evento de pulsación sucederá cuando el botón es pulsado, indistintamente de cómo se ha hecho. Podremos pulsar el botón utilizando el cursor del ratón, la tecla de acceso rápido, desplazándonos hasta él con la tecla `<Tab>` y pulsando la barra espaciadora, con la tecla `<Intro>` si es el botón por defecto, o con la tecla `<Esc>` si es el botón de cancelación.

Una etiqueta **-Label-** es un control que presenta un texto en el formulario (`Caption`) pero que no puede tomar el foco de entrada ni podemos acceder a él con la tecla `<Tab>`. Sólo recibe eventos de ratón.

Las propiedades más comunes de una etiqueta son, además de las que definen su aspecto (`Width`, `Color`, etc.), son:

Propiedad	Para qué sirve	Valores que puede tomar
<code>AutoSize</code>	Para que el tamaño de la etiqueta se ajuste automáticamente al contenido de ésta o no.	<code>True/false</code>
<code>Alignment</code>	Para la alineación del texto	A izquierda: <code>taLeftJustify</code> A derecha: <code>taRightJustify</code> Centrado: <code>taCenter</code>
<code>FocusControl</code>	Tiene sentido cuando la etiqueta se usa como título de otro control. Sirve para que la etiqueta de texto sepa a qué control debe pasar el foco de entrada cuando se pulse la tecla de acceso rápido ( <code>&amp;</code> ).	Nombre del control que recibe el foco.
<code>WordWrap</code>	Para permitir que el texto ocupe varias líneas	<code>True/false</code>
<code>Transparent</code>	Para tomar como color de fondo el del componente que lo contiene.	<code>True/false</code>

Como ejemplo práctico realizaremos un proyecto que presente una ficha con 3 botones y una etiqueta que irá alternando su justificación según la elección de los botones.

1. Comenzar un nuevo proyecto y asignar al formulario las siguientes propiedades:

Caption          Pruebas con botones y etiquetas

Color            clAqua

2. Insertar una etiqueta y dar el valor false a la propiedad `AutoSize` y redimensionar su tamaño de forma que ocupe prácticamente todo el ancho del formulario. Asignar a la etiqueta el `Color clYellow`. Verificar que la propiedad `Alignment` tiene el valor `taLeftJustify`. Asignar a `Caption` el valor "Soy la etiqueta".

3. Insertar 3 botones de la forma siguiente:

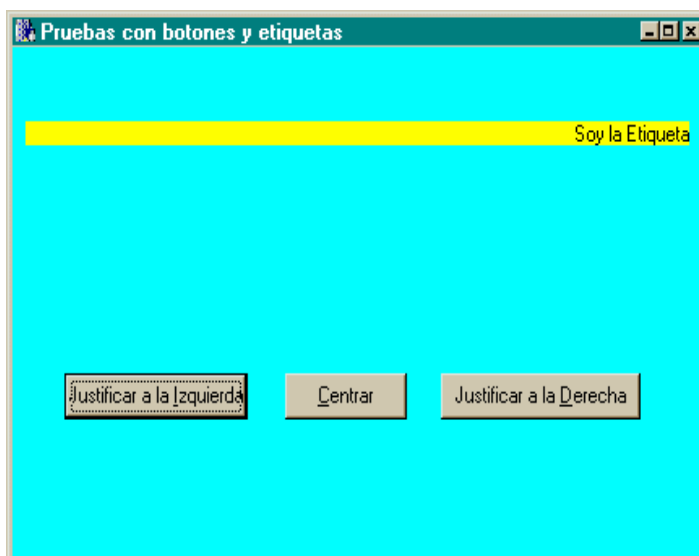
Botón 1:      Caption                      Justificar a la Izquierda

Botón 2:      Caption                      Centrar

Botón 3:      Caption                      Justificar a la Derecha

4. Asignar true a la propiedad `Default` del primer botón (justificación izquierda) y true a la propiedad `Cancel` del tercer botón (justificación a la derecha)
5. Hacer clic en el evento `OnClick` del primer botón y asignar, mediante código, el valor `taLeftJustify` a la propiedad `Alignment` de la etiqueta.
6. Hacer lo mismo con el resto de los botones asignando los valores `taCenter` y `taRightJustify` en los eventos de pulsación del segundo y tercer botón, respectivamente.
7. Almacenar el programa con los nombres PE2.BPR (proyecto) y E2.CPP (unidad).
8. Mejorar la aplicación anterior permitiendo acceder con teclas rápidas a cada uno de los botones y establecer el color de fondo de la etiqueta al que actualmente tenga el formulario cuando el ratón esté situado sobre la misma.

La apariencia que deberá tener la ventana del proyecto con esta última mejora es la siguiente:



### **EJERCICIO PRÁCTICO 3. Trabajar con Campos de Edición.**

La finalidad de un campo de edición **Edit** es permitir la entrada de un texto por parte del usuario. Este texto se almacena en la propiedad `Text`, a la que es posible dar un valor inicial en la fase de diseño. El control `Edit` no cuenta con una propiedad `Caption`, por lo que se suele disponer una etiqueta de texto adjunta que sirve como título.

Cuando en ejecución el usuario modifica el contenido de un `Edit`, la propiedad `Modified` de éste toma el valor `true`. Desde el código de nuestro programa podemos comprobar el valor de esta propiedad y, en caso de que proceda, recuperar la entrada realizada por el usuario leyendo la propiedad `Text`.

#### Longitud del Texto

Por defecto el control `Edit` no pone ningún límite a la cantidad de texto que el usuario puede introducir y, en caso de que sea necesario, desplazará el contenido actual del control a medida que se introducen nuevos caracteres. Si se desea, se puede limitar el número de caracteres que es posible introducir mediante la propiedad `MaxLength`. El valor 0 indica que no hay un límite establecido y cualquier otro número entero fija la longitud máxima de la propiedad `Text`.

#### Selección de texto

Mientras se edita el contenido de un control `Edit`, el usuario puede marcar una porción del texto, mediante la combinación de la tecla <Mayus> y los cursores o bien con el ratón, con el fin de realizar alguna operación que le afecte, como puede ser eliminarlo o copiarlo al portapapeles.

En cualquier momento podemos saber qué texto es el que hay seleccionado en el control, para lo que disponemos de las propiedades `SelStart`, `SelLength` y `SelText`. La primera contiene el carácter a partir del cual se ha seleccionado, sabiendo que el primero de los existentes es el carácter 0. La segunda propiedad contiene el número de caracteres que hay marcados y la tercera contiene el texto.

El valor de estas propiedades también puede ser establecido por el código de nuestro programa, seleccionando automáticamente el texto que nos interese. Si deseamos marcar todo el texto contenido en el control podemos realizar una simple llamada al método `SelectAll()`.

Las operaciones de copiar, cortar y pegar con el portapapeles, pueden ser también realizadas mediante código gracias a la existencia de varios métodos al efecto. El método `ClearSelection()` elimina el texto que hay seleccionado (cómo si el usuario hubiera pulsado la tecla <Supr>). El método `CopyToClipboard()` copia el texto al portapapeles, mientras que `CutToClipboard()` lo copia al portapapeles y lo elimina del campo de edición. `PasteFromClipboard()`, finalmente, nos permite recuperar el texto que hay en el portapapeles insertándolo en `Edit` en la posición actual del cursor.

#### Texto de sólo lectura y oculto

Si lo que queremos es solamente mostrar un valor en un campo de edición e impedir que el usuario pueda modificarlo bastará asignar a la propiedad `ReadOnly` el valor `true`.

Si lo que queremos es que al solicitar cualquier tipo de dato éste no sea visible por terceras personas, por ejemplo, cuando se solicita una clave de acceso a una aplicación, mediante la propiedad `PasswordChar` podemos conseguir que el campo de edición vaya representando cada uno de los caracteres introducidos mediante un cierto símbolo, como puede ser un asterisco. En realidad se puede asignar a esta propiedad cualquier carácter.

#### Controles de entrada

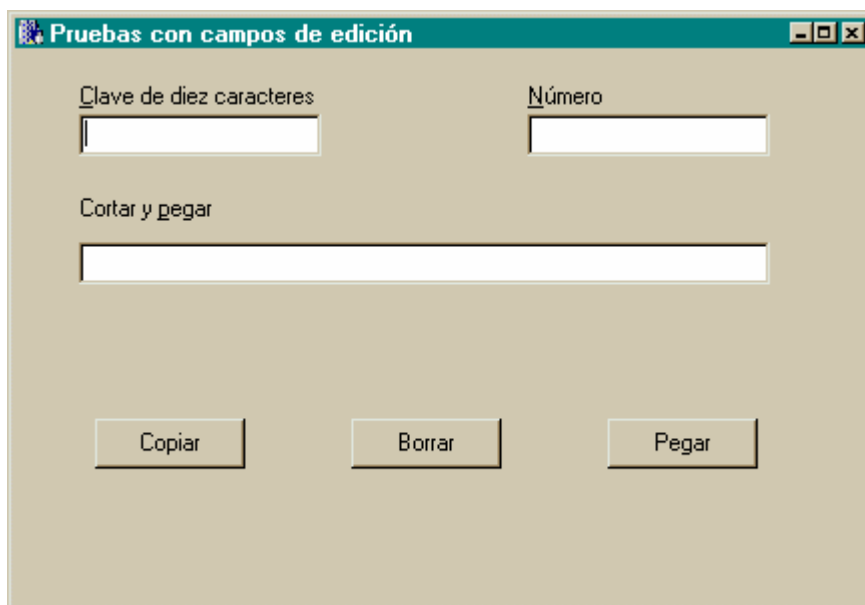
La introducción de ciertos datos puede requerir que todas las letras sean facilitadas en mayúsculas o en minúsculas con el fin de evitar posteriores fallos de comparación. Mediante la propiedad `CharCase` podemos

forzar a ello, asignándole los valores `ecNormal`, `ecUpperCase` o `ecLowerCase`.

El componente `Edit` no realiza ningún tipo de comprobación previa a la admisión de los caracteres que el usuario pueda introducir. Sin embargo, éste es un aspecto que podemos controlar nosotros mismos gracias a que cada vez que el usuario pulsa una tecla, y justo antes de que el correspondiente carácter sea almacenado en la propiedad `Text`, el `Edit` genera un evento `OnKeyPress`, en el que podemos realizar las comprobaciones necesarias.

El evento `OnKeyPress` recibe un parámetro por referencia con el carácter pulsado, de forma que podemos tanto comprobar su contenido como modificarlo o ignorarlo asignando el carácter nulo.

En la práctica, vamos a partir insertando en un formulario tres campos de edición (`Edit`), que estarán encabezados por tres etiquetas (`Label`) y tres botones (`Button`). El aspecto del formulario será el que se muestra a continuación:



El primer control `Edit`, que recibirá el foco desde la etiqueta `Label1` (propiedad `FocusControl` de la etiqueta) lo utilizaremos para comprobar el funcionamiento de las propiedades `MaxLength`, a la que asignaremos el valor 10, y `PasswordChar`, a la que asignaremos el valor `*`.

A continuación tenemos el segundo `Edit`, que recibirá el foco desde `Label2` y que usaremos para ver cómo podemos controlar la entrada de caracteres, permitiendo tan sólo la introducción de dígitos numéricos. Abra la página de eventos en el Inspector de Objetos de ese control y haga doble clic sobre el evento `OnKeyPress`, escribiendo el código que se muestra a continuación:

```
Void __fastcall TForm1::Edit2KeyPress(TObject *Sender, char &Key)
{
    if ((Key < '0' || Key > '9') && Key != 8) Key=0;

    // Se comprueba que el carácter sea un dígito o la tecla de borrado. Los caracteres se
    // encierran entre comillas simples porque estamos comparando un carácter tipo char. Asignar a
    // Key el valor 0 es anular la pulsación.
}
```

El último control `Edit` (recibe el foco de `Label3`) lo utilizaremos para probar los métodos de copiar, borrar y



pegar el texto seleccionado. Será al pulsar cualquiera de los tres botones que hemos dispuesto debajo cuando se realice la acción de copiado, borrado o pegado, gracias al código que asociaremos al evento OnClick de cada uno de los botones de la forma que se muestra a continuación.

```
Void__fastcall TForm1::Button1Click(TObject *Sender)
{
    Edit3->SelectAll();
    Edit3->CopyToClipboard();
    Edit3->SetFocus();
}

Void__fastcall TForm1::Button2Click(TObject *Sender)
{
    Edit3->ClearSelection();
    Edit3->SetFocus();
}

Void__fastcall TForm1::Button3Click(TObject *Sender)
{
    Edit3->PasteFromClipboard();
    Edit3->SetFocus();
}
```

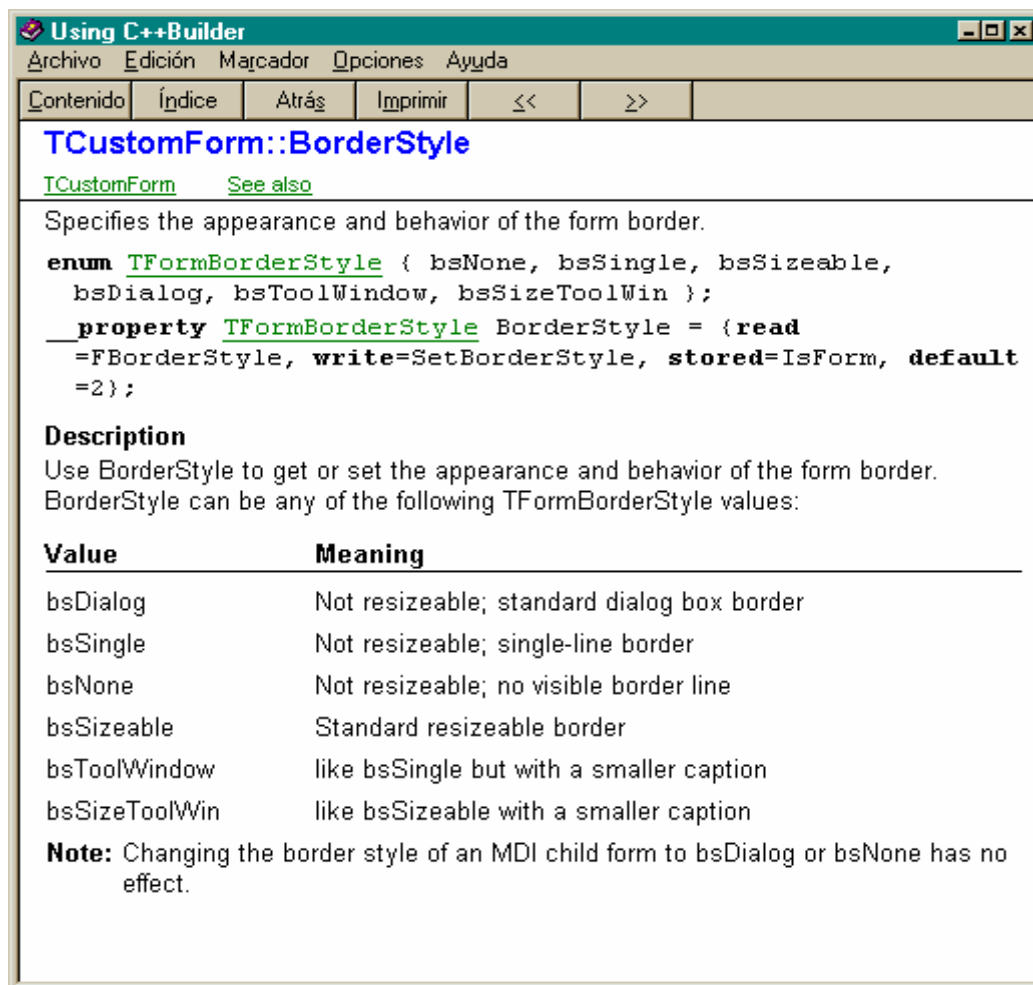
Observe que tras realizar la operación se hace una llamada al método SetFocus(), con el fin de devolver el foco de entrada al control Edit3.

## EJERCICIO PRÁCTICO 4. Trabajar con Memo, CheckBox y RadioGroup.

## Tipos de propiedades

Cada propiedad de un componente de la VCL tiene asignado un tipo de dato. Este tipo de dato determinará los posibles valores que puede tomar. Por ejemplo, la propiedad Visible es de tipo bool y, por ello, sólo puede tomar los valores true o false, en cambio, la propiedad Caption es de tipo cadena, pudiendo tomar como valor cualquier texto.

Existen propiedades de tipo **enumerado**, que son aquellas que solamente pueden tomar valores de una lista prefijada. Un ejemplo de ello es la propiedad BorderStyle de un formulario en cuya página de ayuda observamos lo siguiente:



Donde

```
enum TFormBorderStyle { bsNone, bsSingle, bsSizeable, bsDialog,
    bsToolWindow, bsSizeToolWin };
```

quiere decir que la propiedad BorderStyle de un formulario es del tipo TFormBorderStyle, un tipo enumerado, y que, por tanto, sólo puede tomar los 6 valores preestablecidos.

En el caso de los tipos enumerados, es posible referirse a los valores con su propio nombre, p.ej.

bsSizeable, o mediante una conversión de la posición que ocupa el valor dentro de los posibles (comenzando por la posición 0). Esto último se realiza con el operador `static_cast` de la forma

```
static_cast<tipo_de_dato_enumerado>(posición_del_valor)
```

Así pues, la propiedad `BorderStyle` de un formulario podría modificada de las siguientes formas:

- 1) Asignando directamente el valor en el Inspector de Objetos, por ejemplo el valor `bsSingle`.
- 2) Asignando, mediante código, el valor.

```
BorderStyle=bsSingle;
```

- 3) Transformando, mediante código, un número entero (en este caso 0,1,2,3,4 o 5) en un valor concreto.

```
BorderStyle= static_cast<TFormBorderStyle>(1)
```

Ya que el valor `bsSingle` ocupa la posición 1 del rango de valores.

### **El control TMemo:**

El control `TEdit` es el de uso más habitual cuando en un programa es necesario solicitar una entrada del usuario, pero esa entrada no puede exceder una línea. Cuando necesitemos una entrada de texto más amplia que ocupe varias líneas podemos usar un control `TMemo`.

Este control cuenta con muchas propiedades que ya hemos visto trabajando con un `TEdit`, como `ReadOnly`, `MaxLength`, `Text`, ..., así como con los métodos relacionados con las operaciones del portapapeles.

Un control `TMemo` puede contar con barras de desplazamiento, permitiendo trabajar con líneas más largas que el espacio horizontal disponible y con más líneas de las que es posible mostrar de forma simultánea. La propiedad `TScrollBars` permite el uso de estas barras de desplazamiento. Esta propiedad es de tipo `TScrollStyle`, que es un tipo enumerado, es decir, que sólo puede tomar los valores:

```
0 ssNone
1 ssVertical
2 ssHorizontal
3 ssBoth
```

El texto almacenado en un control `TMemo` es accesible mediante la propiedad `Text`, al igual que en un control `TEdit`. Sin embargo, es conveniente e incluso más útil poder acceder a ese texto línea a línea. Para ello, el control `TMemo` cuenta con la propiedad `Lines` que es de tipo `TStrings`. Este tipo de objetos tiene dos métodos muy interesantes con relación a un control `TMemo` que sirven para guardar o recuperar las líneas de texto en un archivo en disco. Estos métodos son `SaveToFile()` y `LoadFromFile()`, que solamente toman como único parámetro el nombre del fichero donde se va a guardar o del que se va a recuperar.

También podemos saber en cada momento con cuántas líneas cuenta un control `TMemo` mediante la propiedad `Count` del objeto `TStrings`, de la forma:

```
Memor1->Lines->Count
```

Más adelante, cuando veamos el `TRadioGroup`, conoceremos más métodos del tipo `TStrings`.

Un control `TMemo`, mientras está activo, puede permitir el uso de las teclas `<Intro>` y `<Tab>` como parte de la edición de texto, pero estas teclas tienen un funcionamiento especial dentro de un formulario. Para que al pulsarse la tecla `<Intro>` el control `TMemo` la reconozca como salto de línea y no como activación del botón por defecto, por ejemplo, es necesario asignar el valor `true` a la propiedad `WantReturns`. Lo mismo ocurre con la propiedad `WantTabs`, que si está activada permitirá insertar tabulaciones en el texto en lugar de pasar el foco de entrada al siguiente componente del formulario.

### **El control TCheckBox**

El control `TCheckBox` es adecuado para mostrar y permitir la entrada de cualquier información de tipo `bool` que es posible representar con tan sólo dos estados (sí o no, verdadero o falso, activado o desactivado). Los dos estados posibles se muestran visualmente mediante una casilla, que puede contener o no en su interior una marca.

La propiedad `Checked` es la más importante de este control, ya que nos permite tanto saber el estado actual del control como modificarlo. Estará a `true` cuando el control esté marcado y a `false` en caso contrario.

Podemos tener tantos controles `TCheckBox` en un formulario como queramos. El hecho de que alguno de ellos esté activo no afecta a los demás, es decir, las opciones que representan no son exclusivas entre sí. Para que varias opciones sean excluyentes se necesitan controles `TRadioButton`.

Normalmente, los controles `TCheckBox` se suelen agrupar en un contenedor `TGroupBox` con el fin de asignar un título común a los componentes `TCheckBox` que afecten a un mismo tema.

### **Los controles TRadioButton y TRadioGroup**

Como se dijo anteriormente, a veces nos puede interesar representar opciones de tipo `bool` que son exclusivas entre sí, de forma que sólo sea posible seleccionar una de ellas, al tiempo que, obligatoriamente, una debe estar activa. En este caso utilizaríamos controles `TRadioButton`, que son muy similares a los controles `TCheckBox` tanto en aspecto como en funcionamiento. Así pues, este control también cuenta con una propiedad `Checked` en la que se almacena el estado del control.

Si tenemos varios controles `TRadioButton` en un formulario, ¿qué ocurre si queremos solicitar varios datos diferentes que tengan, a su vez, opciones exclusivas entre sí?. Pues que deberemos agrupar los controles `TRadioButton` que representen una misma finalidad dentro de un `TRadioGroup`, que, además, permite asignar un título (`Caption`) al grupo de botones.

Un control `TRadioGroup` es capaz de almacenar múltiples elementos, cada uno de los cuales se muestra como un `TRadioButton` independiente. Para establecer las opciones existentes en un control `TRadioGroup` tendremos que editar el contenido de la propiedad `Items`. Esta propiedad (al igual que la propiedad `Lines` de un campo `TMemo`) es del tipo `TStrings`.

En modo de diseño basta con hacer doble clic sobre la propiedad `Items` para abrir el editor específico con el que cuenta, añadiendo los títulos de las opciones que deseemos mostrar. En ejecución podremos usar los métodos `Add()`, `Append()`, `Insert()` y `Clear()` para añadir o eliminar elementos.

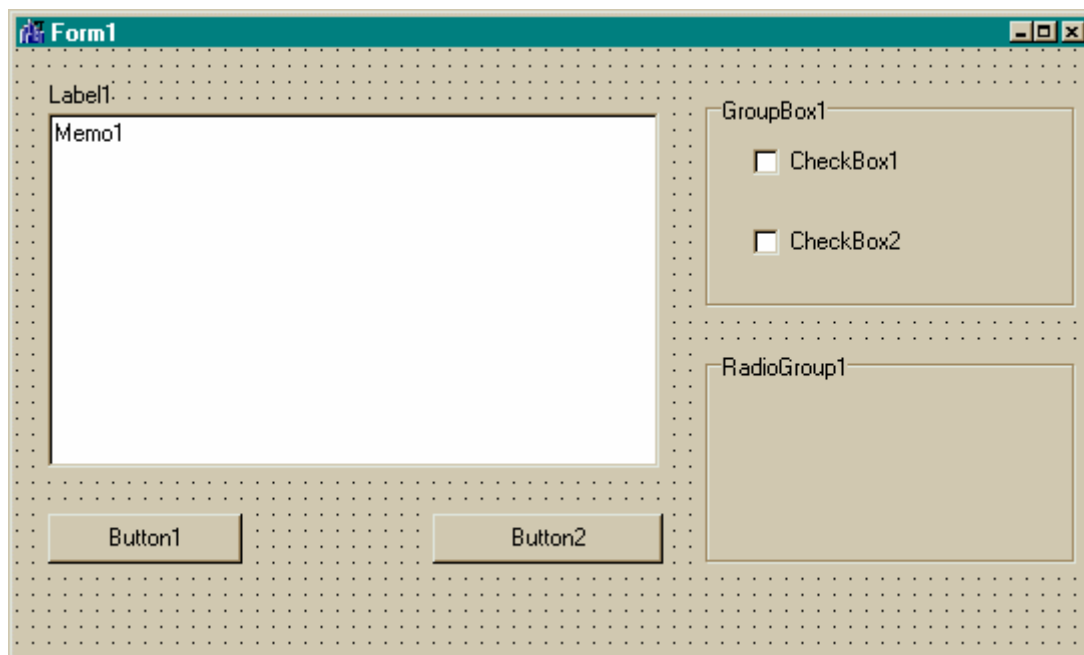
Los elementos existentes en un control `TRadioGroup` se numeran con un índice cuyo punto de partida es el 0. Este índice nos servirá para saber qué opción es la que está activa o bien para establecerla, mediante la propiedad `ItemIndex`. De esta forma no es necesario comprobar la propiedad `Checked` de cada uno de los

botones de radio que aparece en el RadioGroup.

### En la práctica

- Comenzar una nueva aplicación e incluir, en fase de diseño, los siguientes componentes:
  - o Un TLabel
  - o Un TMemo
  - o Un TGroupBox
  - o Dos TCheckBox dentro del TGroupBox
  - o Un TRadioGroup
  - o Dos TButton

de la forma que se presenta a continuación:



- Salvar el proyecto (Save All) con los nombres: E4.CPP (para la unidad) y PE4 (para el proyecto).
- Continuar los pasos necesarios para conseguir como código fuente de la unidad lo siguiente:

```
//-----  
#include <vcl.h>  
#pragma hdrstop  
  
#include "E4.h"  
//-----  
#pragma package(smart_init)  
#pragma resource "*.dfm"  
TForm1 *Form1;  
//-----  
__fastcall TForm1::TForm1(TComponent* Owner)  
    : TForm(Owner)  
{
```

```
}
//-----
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    Caption="Pruebas con Memo, CheckBox y RadioGroup";

    Label1->Caption="&Texto";
    Label1->FocusControl=Memo1;

    Memo1->Lines->Clear();
    Memo1->WantReturns=true; //Por defecto
    Memo1->WantTabs=false; //Por defecto
    Memo1->ScrollBars=ssNone; //Por defecto

    GroupBox1->Caption="Permitir ...";

    CheckBox1->Caption="Tabulación";
    CheckBox1->Checked=false;

    CheckBox2->Caption="Retorno de carro";
    CheckBox2->Checked=true;

    RadioGroup1->Caption="Barras de Desplazamiento";
    RadioGroup1->Items->Add("Ninguna");
    RadioGroup1->Items->Add("Horizontal");
    RadioGroup1->Items->Add("Vertical");
    RadioGroup1->Items->Add("Ambas");
    RadioGroup1->ItemIndex=0;

    Button1->Caption="&Guardar texto";
    Button2->Caption="&Recuperar texto";
    Button2->Enabled=false;
}
//-----
void __fastcall TForm1::CheckBox1Click(TObject *Sender)
{
    Memo1->WantTabs=CheckBox1->Checked;
}
//-----
void __fastcall TForm1::CheckBox2Click(TObject *Sender)
{
    Memo1->WantReturns=CheckBox2->Checked;
}
//-----
void __fastcall TForm1::RadioGroup1Click(TObject *Sender)
{
    Memo1->ScrollBars=static_cast<TScrollStyle>(RadioGroup1->ItemIndex);
}
//-----
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    Memo1->Lines->SaveToFile("memo.txt");
    Button2->Enabled=true;
}
//-----
void __fastcall TForm1::Button2Click(TObject *Sender)
{
    Memo1->Lines->LoadFromFile("memo.txt");
}
//-----
```

## EJERCICIO PRÁCTICO 5. Trabajar con Listas de Elementos.

Otro de los controles que nos permite mostrar listas de elementos de las cuales es posible seleccionar uno o varios es **TListBox**. Este control se conoce como lista, un recuadro en el cual aparecen una serie de líneas de texto siendo posible la selección de una o varias de ellas. El funcionamiento de este control es parecido al del **TRadioGroup**, aunque visualmente su aspecto es bastante diferente.

Los elementos con los que cuenta una lista están contenidos en la propiedad **Items**, que, como sabemos, es un objeto de tipo **TStrings**. En modo de diseño podemos usar el editor de código asociado para incluir los elementos que queramos en la lista simplemente haciendo doble clic sobre la propiedad **Items**. En cambio, en modo de ejecución, podemos usar los métodos siguientes:

Método	Explicación
<b>Add()</b>	Toma como parámetro una cadena de texto, añadiéndola como nuevo elemento al final de la lista, salvo si la propiedad <b>Sorted</b> de la lista está a <b>true</b> , en cuyo caso se insertará en la posición que le corresponda.
<b>Insert()</b>	Toma dos parámetros, indicando el primero la posición en la que se desea insertar el elemento y el segundo el texto a añadir.
<b>Delete()</b>	Este método nos permite eliminar un elemento de la lista, para lo cual deberemos facilitar su índice.
<b>Clear()</b>	No necesita parámetros. Elimina todas las cadenas existentes en la lista.
<b>IndexOf()</b>	Toma como parámetro una cadena, devolviendo el índice que ésta ocupa en la lista o <b>-1</b> en caso de no encontrarse.
<b>Move()</b>	Facilita el desplazamiento de un elemento de una posición a otra, para lo cual deberemos facilitar el índice en que se encuentra actualmente y el índice correspondiente al punto al que se desea mover.
<b>Exchange()</b>	Este método es similar al anterior, tomando como parámetros dos índices con el fin de intercambiar las posiciones de los elementos correspondientes.
<b>SaveToFile()</b>	Guarda el contenido de la lista a un archivo en disco, cuyo nombre deberemos facilitar.
<b>LoadFromFile()</b>	Recupera la lista de elementos de un archivo cuyo nombre se facilita como parámetro.

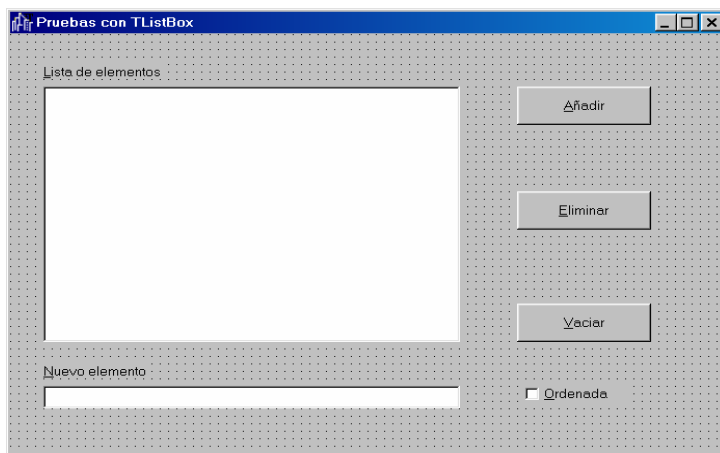
La propiedad **ItemIndex** de este control permite saber en cada momento qué elemento hay seleccionado (igual que en un **TRadioGroup**). El valor **-1** indica que en ese momento no hay ninguno.

Cuando una lista contiene más elementos de los que le es posible mostrar de forma simultánea, automáticamente aparece una barra de desplazamiento vertical que nos permite acceder al contenido completo de la propiedad **Items**. Podemos determinar, entonces, cuál es el primer elemento que está visible en un determinado momento, en la parte superior de la ventana, mediante la propiedad **TopIndex**, que almacena el índice de dicho elemento.

Al desplazar los elementos que contiene la lista podemos observar, posiblemente que el elemento que está en la parte superior o el situado en la parte inferior aparecen cortados, viéndose sólo un trozo de texto. Esto se debe a que el alto de la lista no es un múltiplo de la altura de cada elemento. Esto se soluciona asignando el valor **true** a la propiedad **IntegralHeight**.

El programa ejemplo que vamos a realizar nos permitirá añadir elementos a una lista en ejecución con el texto que se facilite en un campo de edición. También se podrá eliminar el elemento seleccionado o vaciar totalmente la lista. Los elementos podrán aparecer ordenados o no, según el valor de un campo **CheckBox**.

- Insertar en el formulario los controles necesarios para que la ventana tenga la siguiente apariencia:



- Dar el valor `true` a la propiedad `Default` del primer botón, haciendo así que la pulsación de la tecla `<Intro>` tenga como resultado la inserción en la lista del elemento cuyo texto se ha introducido en el `TEdit`.
- Asignar el valor `false` a la propiedad `Enabled` del segundo botón, ya que sólo debe estar activo cuando haya un elemento seleccionado.
- Asociar al evento `OnClick` del primer botón el siguiente código:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    // Al pulsar el botón Añadir añadimos a la lista el texto existente en el TEdit
    ListBox1->Items->Add(Edit1->Text);
    // Y marcamos todo el contenido de dicho control:
    Edit1->SelectAll();
}
```

- Asociar al evento `OnClick` de la lista el siguiente código:

```
void __fastcall TForm1::ListBox1Click(TObject *Sender)
{
    // Al pulsar en la lista activamos o desactivamos el botón de borrar
    // según haya o no seleccionado un elemento
    Button2->Enabled=ListBox1->ItemIndex != -1;
}
```

- Asociar al evento `OnClick` del segundo botón el siguiente código:

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{
    // Al pulsar el botón Eliminar eliminamos el texto seleccionado
    ListBox1->Items->Delete(ListBox1->ItemIndex);
    // Y desactivamos el botón de Eliminar
    Button2->Enabled=false;
}
```

- Asociar al evento `OnClick` del tercer botón el siguiente código:

```
void __fastcall TForm1::Button3Click(TObject *Sender)
{
    // Al pulsar el botón Vaciar dejamos vacía la lista
    ListBox1->Items->Clear();
}
```

- Asociar al evento `OnClick` del `CheckBox` el siguiente código:

```
void __fastcall TForm1::CheckBox1Click(TObject *Sender)
{
    // Al activar o desactivar la opción Ordenada damos el valor
    // adecuado a la propiedad Sorted de la lista
    ListBox1->Sorted=CheckBox1->Checked;
}
```



## EJERCICIO PRÁCTICO 6. Trabajar con Listas Combinadas.

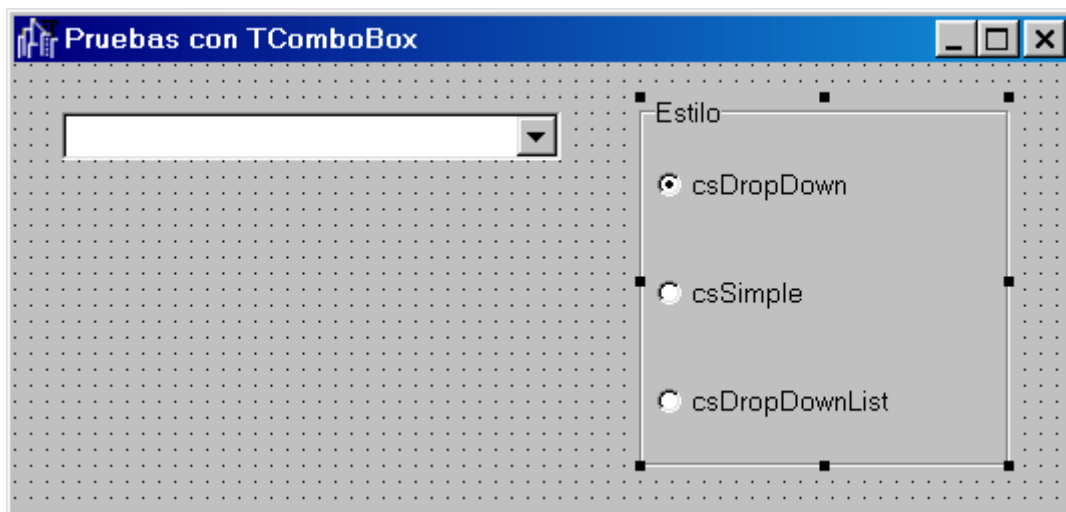
En el ejercicio anterior usamos un `TListBox` para almacenar una lista de elementos junto a un campo de edición para aceptar datos a introducir en la lista. Pues bien, existe un control denominado **TComboBox**, conocido como lista combinada, que es precisamente la unión de un `TEdit` con un `TListBox`.

Básicamente, un `TComboBox` incorpora las funciones de ambos controles y, por tanto, cuenta con propiedades que ya conocemos como `Text`, `MaxLength`, `Items`, `ItemIndex`, etc. Tanto el aspecto como el funcionamiento de una lista combinada dependerá del estilo que tenga, el cual dependerá del valor que se asigne a la propiedad `Style`. Esta propiedad, de tipo `TComboBoxStyle`, puede tomar 5 valores de los cuales solamente vamos a estudiar los siguientes:

Valor de la propiedad <code>Style</code>	Aclaración
<code>csDropDown</code>	Es el estilo por defecto. Presenta un campo de edición en el que es posible introducir texto como si se tratara de un <code>TEdit</code> . Aparece en el extremo derecho un botón que despliega una lista con los elementos contenidos actualmente.
<code>csSimple</code>	La lista combinada aparece como un <code>TEdit</code> sin botón ni lista desplegable adjunta. Para recorrer los elementos de ésta es posible usar las teclas de cursor arriba y abajo.
<code>csDropDownList</code>	Es el estilo usado cuando deseamos que el usuario pueda seleccionar un valor de la lista, desplegándola, pero no que pueda introducir un texto. El aspecto del control es idéntico al que presenta con el valor <code>csDropDown</code> .

Los valores `csOwnerDrawFixed` y `csOwnerDrawVariable` se estudiarán más adelante.

En la práctica vamos a diseñar un programa que presentará una ventana con la siguiente apariencia:



En la parte izquierda aparece un control `TComboBox` con el valor `csDropDown` en la propiedad `Style` (por defecto). Habrá que eliminar el contenido de la propiedad `Text` con el fin de que en ejecución aparezca vacío el campo destinado a la edición. A la derecha aparece un `TRadioGroup` con 3 opciones que nos van a permitir cambiar el estilo de la lista combinada durante la ejecución, lo que nos servirá para observar las diferencias entre unos y otros.

Deberemos controlar cada pulsación de tecla sobre el `TComboBox`, mediante el evento `OnKeyPress`, para que en el momento en que se detecte la pulsación de la tecla `<Intro>` se proceda a la inserción del texto en la lista, ignorando la pulsación. El código que deberemos escribir es el siguiente:

```
void __fastcall TForm1::ComboBox1KeyPress(TObject *Sender, char &Key)
{
    if (Key == 13)
    {
        ComboBox1->Items->Add(ComboBox1->Text);
        ComboBox1->SelectAll();
        Key=0;
    }
}
```

Por último, asignaremos el valor 0 a la propiedad `ItemIndex` del `TRadioGroup` y deberemos controlar su evento `OnClick` con el fin de establecer el estilo adecuado para la lista combinada según la opción que se seleccione. Esta asignación se realizará de forma directa llevando a cabo una conversión de tipo de la forma siguiente:

```
void __fastcall TForm1::RadioGroup1Click(TObject *Sender)
{
    ComboBox1->Style=static_cast<TComboBoxStyle>(RadioGroup1->ItemIndex);
}
```

---

## Lista de Ejercicios

1. Realizar mejoras sobre el ejercicio anterior de forma que...

- a) ... no se permita añadir elementos vacíos a la lista. (*propiedad Text*)
- b) ... la pulsación de un doble-clic haga que, automáticamente, aparezca seleccionado el siguiente elemento de la lista, siendo el siguiente al último el primero. (*propiedad Count de la propiedad Items*)

2. Diseñar un programa que presente un formulario para la introducción de los siguientes datos acerca de una persona:

- **Nombre** completo (apellidos y nombre de pila)
- **Edad**
- **Sexo** (Hombre o Mujer)
- **Estado Civil** (Soltero, Casado, Viudo o Divorciado)
- **¿Tiene hijos?** (S/N)  
(En caso afirmativo se deberá indicar el **Número de hijos**)
- **Provincia de Nacimiento**  
(Para facilitar la introducción de este dato se cuenta con el fichero `PROVIN.TXT` de tipo memo donde aparece una línea por cada una de las provincias españolas)

## EJERCICIO PRÁCTICO 7. Control de Excepciones.

Mientras se desarrolla y ejecuta una aplicación, generalmente surgen errores que le impiden funcionar en la forma correcta. Los errores pueden producirse en la fase de compilación o en la fase de ejecución.

Los **errores de compilación** son detectados por C++ Builder en el momento en que se compila el programa. En estos casos no se permite la ejecución y en la parte inferior del Editor de Código se abre una lista con la descripción de los errores encontrados, permitiéndonos el desplazamiento a la línea apropiada. Estos errores se producen cuando se teclea mal el nombre de un identificador, hemos olvidado cerrar un paréntesis, etc.

Por su parte, los **errores de ejecución** no son detectables durante la compilación y surgen de forma esporádica mientras el programa se ejecuta. Si no está adecuadamente controlado, un error de ejecución puede terminar interrumpiendo la ejecución de la aplicación. Estos errores se producen, por ejemplo, cuando se intenta usar un archivo inexistente, se pretende dividir por 0, etc. Es decir, no son detectados por el compilador porque más que errores del programa son errores que vienen provocados por alguna causa externa.

Cuando estas excepciones se producen, automáticamente aparece un cuadro de diálogo en el que se informa al usuario del error producido, pero aparece en inglés y de forma que no siempre el usuario puede entender, además de que se produce una parada en el módulo donde se ha producido el error y las siguientes instrucciones no llegan a ejecutarse. Una de las formas, por tanto, más eficientes de controlar los errores de ejecución consiste en interceptar las **excepciones** que producen, actuando en consecuencia.

### La instrucción **try ... catch**

En el código de un programa sólo algunas sentencias son susceptibles de generar errores de ejecución. Una simple asignación a una variable, por ejemplo, no suele generar una excepción, en cambio, el acceso a un dispositivo externo o una división, sí puede generar tal excepción, puesto que el archivo puede no existir o el dividendo puede ser 0.

Lo primero que hay que hacer, por tanto, para controlar los errores de ejecución es identificar los bloques de código en que pueden tener lugar las excepciones, ya que serán estos bloques los que hay que **proteger** mediante la construcción **try/catch**.

La sintaxis de **try/catch** es la que se muestra en el fragmento siguiente, en el que podemos distinguir unas sentencias a proteger, que son las dispuestas detrás de la palabra **try**, y unas sentencias que se ejecutarán sólo en caso de que se produzca una excepción, que son las escritas a continuación de la palabra **catch**. La construcción se finaliza con el cierre de llaves, de tal forma que la ejecución siempre continuará con la sentencia siguiente, tanto si se ha generado la excepción como si no.

```
try
{
    SentenciaAProteger;
}
catch (...)
{
    SentenciaDeControl;
}
```

Tanto tras la palabra **try** como tras la palabra **catch** podemos disponer múltiples sentencias.

### Clases de excepciones

Normalmente, después de la palabra **catch** basta escribir tres puntos entre paréntesis para detectar cualquier excepción, de la forma: **catch (...)**

Pero a veces es conveniente concretar el error producido y actuar en consecuencia dependiendo de la naturaleza de éste. Por ello también es posible encerrar entre paréntesis detrás de la palabra `catch` el nombre concreto del error añadiendo el carácter `&`.

Las excepciones, al igual que otros muchos elementos de C++Builder, son objetos que tienen cierta jerarquía, existiendo una clase de excepción base o genérica de la cual se van derivando clases más específicas. Existe, por ejemplo, una clase de excepción llamada `EIntError` de la que están derivadas todas las clases de excepciones relacionadas con números enteros, como `ERangeError` o `EDivByZero`.

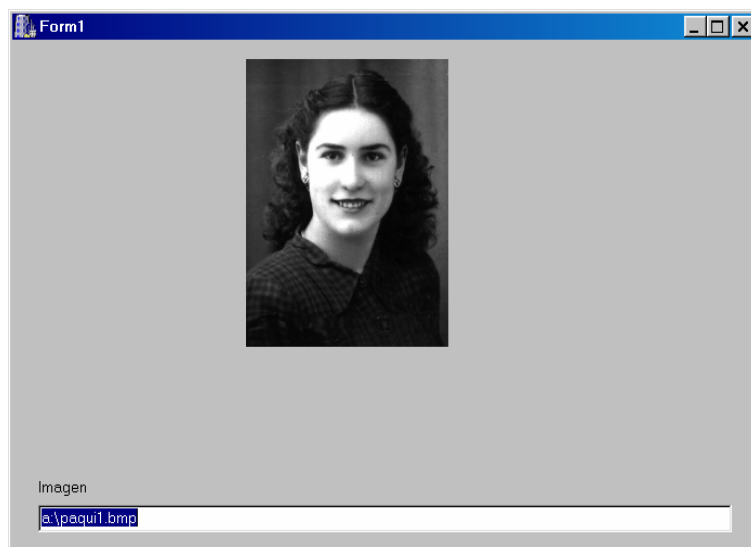
Para controlar la clase de excepción, detrás de la palabra `catch`, y entre paréntesis, podemos disponer el nombre de la clase de excepción a gestionar, por ejemplo, `catch (EDivByZero &)`. El código dispuesto en el bloque siguiente se ejecutará tan sólo en caso de que la excepción que se ha producido sea de la clase indicada. Podemos usar varios apartados `catch` para interceptar excepciones de diferentes clases.

En la práctica, para comprobar el funcionamiento de las excepciones y ver cómo podemos controlarlas, vamos a diseñar un pequeño programa que nos permita presentar en pantalla una imagen que será tomada de un archivo con extensión `.bmp`.

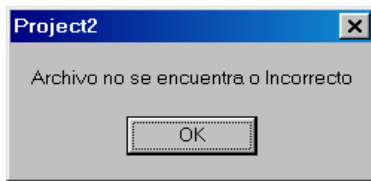
Para ello debemos insertar un `TImage` (de la paleta `Additional`) con las propiedades `AutoSize` y `Center` a `true` y un control `TEdit` (de la paleta `Standard`). Al evento `OnKeyPress` del control `TEdit` asociaremos el código apropiado en caso de que el usuario pulse `<Intro>`; es decir, asignaremos como imagen el nombre del fichero que se ha escrito en el campo `TEdit`, controlando que no exista o no sea un archivo reconocible por un `TImage`. El código sería el siguiente:

```
void __fastcall TForm1::Edit1KeyPress(TObject *Sender, char &Key)
{
    if (Key == 13)
    {
        try {Image1->Picture->LoadFromFile(Edit1->Text);}
        catch (...) {ShowMessage("Archivo no se encuentra o Incorrecto");}
        Edit1->SelectAll();
    }
}
```

La apariencia de la aplicación en ejecución, en caso de que exista el archivo y sea reconocible como archivo imagen, sería la siguiente:

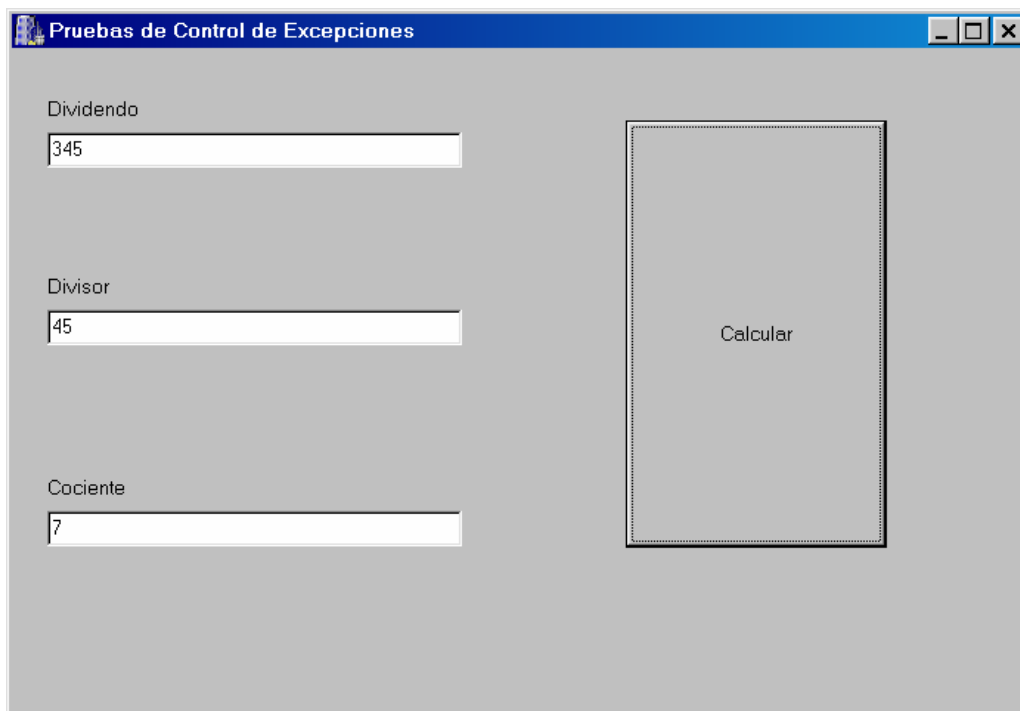


O aparecerá el cuadro de diálogo siguiente en caso de que el archivo no pueda cargarse:



### Ejercicio Propuesto.-

Realizar una aplicación que permita realizar divisiones de números enteros, para lo que tendremos que facilitar el dividendo y el divisor. La apariencia de la ventana en ejecución será la siguiente:



Habrá que asignar el valor `true` a la propiedad `ReadOnly` del campo de edición correspondiente al cociente, ya que éste será solo para mostrar el resultado de la operación y no para que el usuario introduzca un valor.

Habrá que seleccionar conjuntamente los dos primeros controles `Tedit` (de Dividendo y Divisor) para asignarles el mismo código asociado a su evento `OnKeyPress`, con el fin de controlar la entrada de caracteres.

```
if ( ( Key < '0' || Key > '9' ) && Key != 8 )  
    Key=0;
```

Cuando se pulse clic sobre el botón `Calcular` habrá que proteger las excepciones posibles en los siguientes casos:

Posible error	Excepción a controlar
Cuando el divisor sea 0	EdivByZero &
Cuando uno de los operandos sea tan grande que no se pueda convertir a entero	EconvertError &
Cualquier otro error desconocido con tratamiento de números enteros	EintError &

El código asociado es el siguiente:

```
int Dividendo, Divisor, Cociente;
try
{
    // Obtenemos el dividendo y el divisor haciendo una conversión de cadena a entero:
    Dividendo=StrToInt(Edit1->Text);
    Divisor=StrToInt(Edit2->Text);
    // Realizamos la división:
    Cociente=Dividendo/Divisor;
    // Y mostramos el resultado:
    Edit3->Text=Cociente;
}
// Y si se produce excepción:
catch (EdivByZero &) {ShowMessage("El divisor no puede ser cero");}
catch (EconvertError &) {ShowMessage("El dividendo o divisor no son válidos");}
catch (EintError &) {ShowMessage("Se produce un error desconocido");}
```

## **EJERCICIO PRÁCTICO 8. Construcción y uso de Menús.**

Una de las formas más tradicionales y eficientes de mostrar las distintas opciones de que dispone un programa consiste en construir un menú. Existen básicamente dos tipos de menús: principal y emergente.

Un **menú principal** es aquel que se muestra en la parte superior del formulario, justo debajo de la barra de título, y que cuenta con una serie de opciones principales que son las visibles en cada momento, en base a las cuales es posible acceder a la lista de subopciones.

El **menú emergente**, por el contrario, no está visible normalmente, apareciendo tan sólo cuando el código de programa lo solicita. C++ Builder es capaz de establecer una asociación entre un componente y un menú emergente, mostrando este último en el momento en que se pulsa el botón derecho del ratón sobre dicho componente.

### **Diseño de un menú principal.-**

Para crear un menú principal deberemos insertar en el formulario un componente `TMainMenu`, que encontraremos en la página `Standard` de la Paleta de Componentes. Un formulario sólo puede contar con un menú principal.

El siguiente paso consistirá en diseñar el menú, para lo cual tendremos que hacer doble clic sobre el propio componente `TMainMenu` o bien sobre el valor de su propiedad `Items`, en el Inspector de Objetos. En ambos casos aparecerá el Editor de Menús, una ventana en la que podremos ir introduciendo opciones y desplazándonos entre ellas construyendo nuestro menú.

Inicialmente el Editor de Menús aparecerá vacío, mostrando tan sólo un recuadro en el que podremos iniciar la introducción del título de la opción que deseemos definir. Observe que en el Inspector de Objetos existen una serie de propiedades que hacen referencia a la opción que se está definiendo.

Introduzca el título de la primera opción principal, que se llamará por ejemplo *&Archivo*, y pulse la tecla `<Intro>`, lo que le desplazará a la línea siguiente, en la que podrá iniciar la introducción de las subopciones. Vamos a añadir tres opciones, a las que vamos a llamar *&Guardar*, *&Recuperar* y *&Salir*.

### **Inserción y borrado de opciones**

En cualquier momento podemos eliminar opciones del menú e insertar otras nuevas entre algunas existentes. La primera operación la realizaremos simplemente pulsando la tecla `<Supr>`, mientras que la segunda la efectuaremos con una pulsación de la tecla `<Insert>`.

Sitúese encima de la última opción que hemos insertado, *&Salir*, y pulse la tecla `<Insert>`, lo que provocará la apertura de un espacio en blanco. Inserte en ese espacio un guión y pulse `<Intro>`, verá cómo automáticamente en el menú aparece una línea de separación entre las dos primeras opciones y la última.

### **Teclas de acceso rápido**

Además de las letras que precedemos con un signo *&* en los títulos de las opciones y que pueden ser utilizadas conjuntamente con la tecla `<Alt>` para acceder de una forma rápida, cada opción puede contar además con una tecla "caliente" o de acceso rápido.

Sitúese en la opción *&Recuperar* y a continuación despliegue la lista adjunta a la propiedad `Shortcut`, en el Inspector de Objetos. Verá aparecer una serie de teclas y combinaciones de tecla. Seleccione la combinación `<Control-R>`, que aparecerá indicada en el menú a la derecha de la opción. De igual forma puede asociar una tecla de acceso rápido a las demás opciones.

### Opciones con más opciones

Ciertas opciones de un menú pueden dar acceso a otras listas de opciones, lo cual se indica mediante una flecha apuntando hacia la derecha. El ejemplo más cercano lo podemos encontrar en la opción *Reopen* del menú *File* de C++ Builder.

Para añadir una lista de opciones a una opción del menú deberemos pulsar la combinación <Control-Flecha derecha>, o bien desplegar el menú emergente y seleccionar la opción *Create Submenu*. En cualquiera de los casos se abrirá una nueva lista a la derecha de la que ya teníamos, en la que podemos introducir las opciones que deseemos.

### Respuesta a la selección de una opción

Al igual que un botón o una lista, un menú genera el evento *OnClick* cuando se selecciona una cierta opción. Para acceder al código del método asociado a una de las opciones existentes lo único que tenemos que hacer es seleccionarla en modo de diseño en el mismo formulario, tras haber cerrado el editor de menús. También podemos hacer doble clic sobre la opción deseada en el propio Editor de menús, continuando posteriormente con la tarea de diseño.

Observe que cada una de las opciones de un menú es un objeto diferente, concretamente de tipo *TMenuItem*, y por lo tanto genera su propio evento.

### Construcción de un menú emergente.-

El método de diseño de un menú emergente es muy similar al que acabamos de ver para un menú principal, aunque existen algunas diferencias. La primera de ellas es que el componente a utilizar no es un *TMainMenu*, sino un *TPopupMenu*. Un menú emergente, a diferencia de un menú principal, no cuenta con una barra de opciones principales de las que se despliegan listas de opciones, sino que está compuesta de una sola lista de opciones, aunque éstas pueden a su vez dar paso a otras, creando subopciones según lo visto en el punto anterior.

### Desplegar un menú emergente

Como se comentaba anteriormente, un menú emergente puede ser desplegado automáticamente, al pulsar el botón derecho del ratón sobre el componente al que esté asociado, o bien ser abierto desde código, con una llamada al método *Popup()* del propio *TPopupMenu*. Al llamar a dicho método pasaremos como parámetros dos enteros, indicando la posición en la que deseemos hacer aparecer la ventana. El método más habitual es el primero y también el más cómodo y simple.

Para conseguir que un menú emergente se despliegue de forma automática lo primero que tenemos que hacer es dar el valor *true* a la propiedad *AutoPopup*, valor que tiene por defecto. El segundo paso consistirá en crear un enlace entre un componente y el menú, para lo cual asignaremos a la propiedad *PopupMenu* del componente el nombre del *TPopupMenu*. Es posible asociar un *TPopupMenu* al formulario, a un botón o a, prácticamente, cualquier otro control. Algunos componentes suelen tener ya asociado un *PopupMenu* por defecto, que, a no ser que se indique lo contrario o se asigne otro menú emergente, aparecerá de forma automática al pulsar el botón derecho del ratón. Esto último puede comprobarse en un campo *TMemo*, que, por defecto, tiene asociado un menú emergente con las típicas opciones de edición, como *Deshacer*, *Copiar*, *Pegar*, *Eliminar*, ...

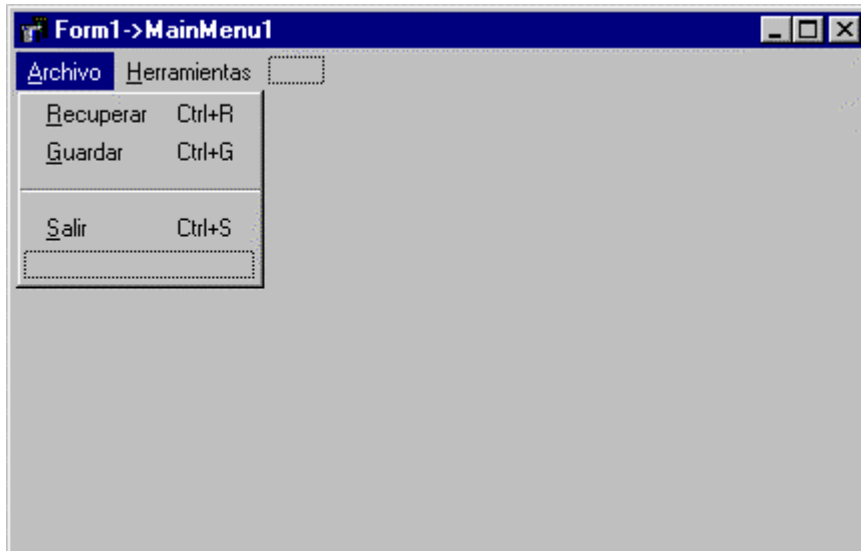
### En la práctica

Para practicar con el diseño y uso de menús vamos a realizar un proyecto que permita al usuario introducir información en el archivo "memo.txt". Para ello insertaremos un menú principal con dos opciones: *Archivo* y *Herramientas*. La opción *Archivo* contará con las opciones *Recuperar* (accesible sólo si ya se ha guardado la

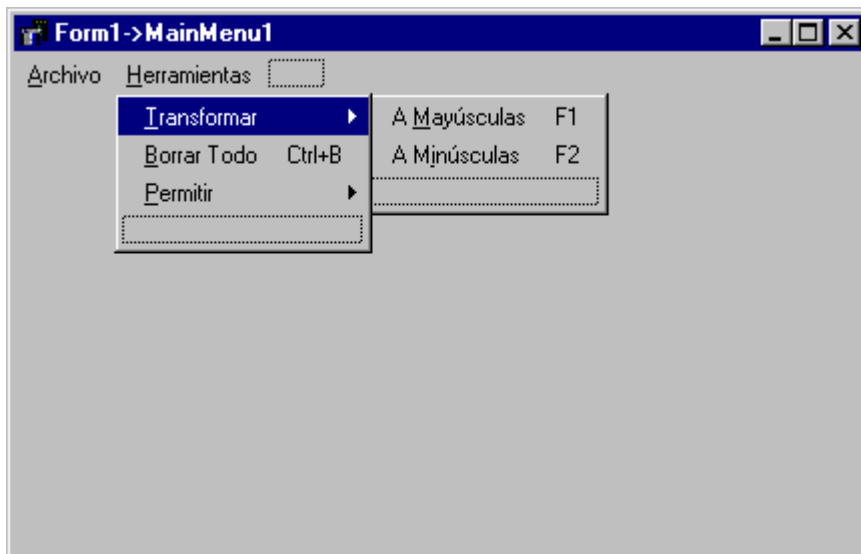


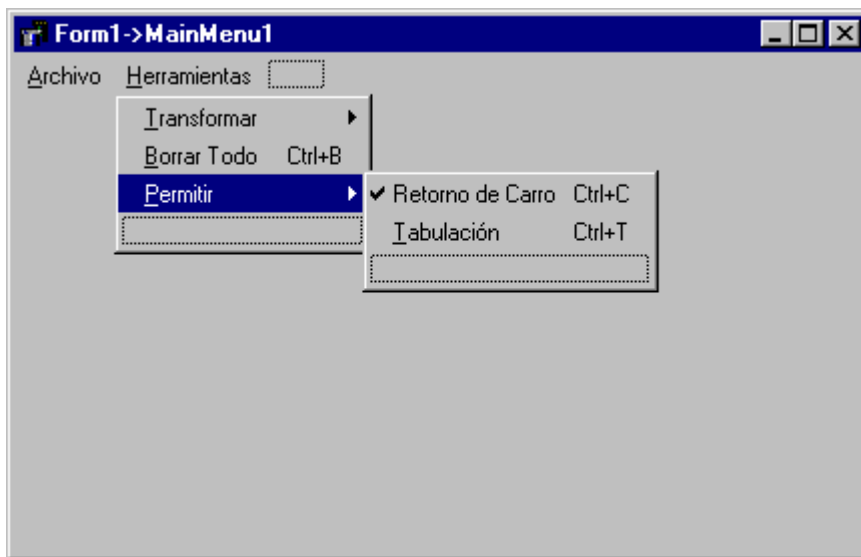
información), *Guardar* y *Salir*. La opción *Herramientas* contará con tres opciones: *Transformar* (a mayúsculas o minúsculas el texto seleccionado), *Borrar Todo* y *Permitir* (si tienen validez o no en el campo Memo las teclas <Intro> y <Tab>).

En fase de diseño ésta será la apariencia de la primera opción del menú principal:



Y estas las apariencias de la segunda opción:





La marca ☒ delante de la subopción “Retorno de Carro” se consigue asignando el valor true a la propiedad Checked, propiedad que modificaremos según se pulse o no la opción (actuaría como un CheckBox).

Por último, después del diseño del menú principal, habrá que insertar un control TMemor sobre el formulario donde se pueda editar el texto que desee el usuario. Este control aparecerá ocupando todo el espacio disponible del formulario o ficha. Para ello asignaremos a su propiedad Align el valor alClient.

A continuación se muestra el código fuente que tiene que asociarse al proyecto. El alumno debe interpretar cada una de las funciones y acceder a ellas según el evento correspondiente:

```
//-----
#include <vcl.h>
#pragma hdrstop

#include "Menu.h"
//-----
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//-----
__fastcall TForm1::TForm1(TComponent* Owner)
: TForm(Owner)
{
    Recuperar1->Enabled=false;
}
//-----
void __fastcall TForm1::Recuperar1Click(TObject *Sender)
{
    Memor1->Lines->LoadFromFile("memo.txt");
}
//-----
void __fastcall TForm1::Guardar1Click(TObject *Sender)
{
    Memor1->Lines->SaveToFile("memo.txt");
}
```

```
    Recuperar1->Enabled=true;
}
//-----
void __fastcall TForm1::Salir1Click(TObject *Sender)
{
    Close();
}
//-----
void __fastcall TForm1::AMayusculas1Click(TObject *Sender)
{
    Memo1->SelText=UpperCase(Memo1->SelText);
}
//-----
void __fastcall TForm1::AMinusculas1Click(TObject *Sender)
{
    Memo1->SelText=LowerCase(Memo1->SelText);
}
//-----
void __fastcall TForm1::BorrarTodo1Click(TObject *Sender)
{
    Memo1->Lines->Clear();
}
//-----
void __fastcall TForm1::RetornodeCarrol1Click(TObject *Sender)
{
    RetornodeCarrol->Checked=! (RetornodeCarrol->Checked);
    Memo1->WantReturns=RetornodeCarrol->Checked;
}
//-----
void __fastcall TForm1::Tabulacin1Click(TObject *Sender)
{
    Tabulacin1->Checked=! (Tabulacin1->Checked);
    Memo1->WantTabs=Tabulacin1->Checked;
}
//-----
```

**Borland C++ Builder. OTROS COMPONENTES.**

Una vez vistos todos los componentes de la paleta `Standard`, vamos a estudiar algunos otros que van a facilitar al usuario el uso de nuestras aplicaciones.

En algunos ejercicios realizados hasta el momento ha sido necesario facilitar el nombre de un archivo en disco, con el fin de recuperar o guardar información en él. En estos ejemplos el nombre del archivo era fijo o se solicitaba mediante un campo de edición. Para que el usuario del programa pueda elegir el archivo deseado, los controles que vamos a estudiar permitirán la selección de una unidad disponible en el sistema, el desplazamiento por los directorios de ésta y la selección de un archivo determinado, todo ello de forma visual, sin necesidad de tener que escribir código alguno. Estos componentes están en la página `Win 3.1` de la paleta de componentes.

Por último, estudiaremos el uso del reloj. Para ello veremos las propiedades y eventos del componente `TTimer` de la paleta `System`.

**EJERCICIO PRÁCTICO 9. Listas de unidades, carpetas y archivos.**

Los tres controles que vamos a conocer aparecen en la paleta `Win 3.1` y son derivados de `TComboBox` o `TListBox`, contando con muchas propiedades que ya conocemos, como `Items`, `ItemIndex`, `Text`, etc.

**Lista de unidades (TDriveComboBox)**

El control `TDriveComboBox` es una lista combinada en la que se muestran las unidades de disco disponibles en el sistema, pudiendo seleccionar cualquiera de ellas. Este control toma los elementos automáticamente en el momento de la creación inspeccionando el sistema.

Como cualquier otra lista, podemos obtener el texto de cada uno de los componentes accediendo a ellos mediante la propiedad `Items` y saber qué elemento es el que está seleccionado a través de `ItemIndex`. Sin embargo, la propiedad que más nos interesará será `Drive`, de tipo `char`, que contiene la letra correspondiente a la unidad seleccionada en cada momento. Esta letra podrá ser mayúscula o minúscula, dependiendo del valor que asignemos a la propiedad `TextCase`, que puede ser `tcLowerCase` o `tcUpperCase`.

Un control `TDriveComboBox` puede estar asociado con un `TDirectoryListBox`, cuyo funcionamiento se trata en el punto siguiente, de tal forma que en el momento que seleccionemos una unidad en el primero, automáticamente se cambie a dicha unidad en el segundo. Para ello lo único que hemos de hacer es asignar a la propiedad `DirList` del `TDriveComboBox` el nombre del `TDirectoryListBox`.

**Lista de directorios (TDirectoryListBox)**

Se usa para mostrar una lista de directorios y como su nombre indica es un derivado de `TListBox`. Este control muestra la jerarquía de directorios correspondiente a la unidad facilitada en la propiedad `Drive`, permitiendo el cambio de un directorio a otro.

La propiedad más interesante es `Directory`, ya que mediante ella podemos recuperar el camino completo que está actualmente seleccionado, sin necesidad de tener que acceder a los elementos de la lista individualmente. Este directorio puede ser mostrado automáticamente en un control `TLabel` simplemente asignando a la propiedad `DirLabel` el nombre de la etiqueta.

Al igual que un `TDriveComboBox` se puede asociar con un `TDirectoryListBox`, éste puede asociarse con un `TFileListBox` (que veremos a continuación), de tal forma que al cambiar de directorio automáticamente se muestren los archivos contenidos en él. Para ello deberemos asignar a la propiedad `FileList`

el nombre del `TFileListBox` a asociar.

### **Lista de archivos (TFileListBox)**

El último de los tres controles, `TFileListBox`, nos permite ver los archivos existentes en un cierto camino que habremos de facilitar en la propiedad `Directory`. Este valor se toma automáticamente si este control está asociado a un `TDirectoryListBox`.

Aunque por defecto se muestran todos los archivos que hay en el directorio, podemos limitar esta selección modificando el contenido de la propiedad `Mask`, a la que asignaremos una cadena conteniendo una o más referencias de selección. En caso de que sean varias deberemos separarlas mediante un punto y coma (por ejemplo `*.bmp;*.ico`).

También podemos limitar los archivos mostrados en el `TFileListBox` según sus atributos, mediante la propiedad `FileType`. Esta propiedad es un conjunto que puede contener `true/false` en los conceptos que se muestran a continuación, según los cuales se seleccionarán unos archivos u otros.

Elemento de la propiedad <code>FileType</code>	Si se le asigna el valor <code>true</code> permitirá ver ficheros...
<code>ftReadOnly</code>	De sólo lectura
<code>ftHidden</code>	Ocultos
<code>ftSystem</code>	De sistema
<code>ftVolumeID</code>	Etiqueta de volumen
<code>ftDirectory</code>	Directorio
<code>ftArchive</code>	Archivo
<code>ftNormal</code>	Normal (todos)

### **En la práctica**

Vamos a realizar un ejercicio que permita visualizar archivos de tipo `.ico` (iconos) y que también permita, mediante un botón, asignar un elemento elegido como icono del formulario. La apariencia de la aplicación en funcionamiento será la que se muestra en la página siguiente.

Insertaremos en el formulario un `TDriveComboBox` en la parte superior izquierda, un `TDirectoryListBox`, debajo del anterior y ocupando el resto de la parte izquierda, un `TFileListBox`, a la derecha del primero, un `TImage` justo debajo de la lista de archivos y, debajo de este último, un `TButton`. Habrá que asociar la lista de unidades con la de directorios y esta última con la de archivos dando los valores apropiados a las propiedades que ya conocemos `DirList` y `FileList`.

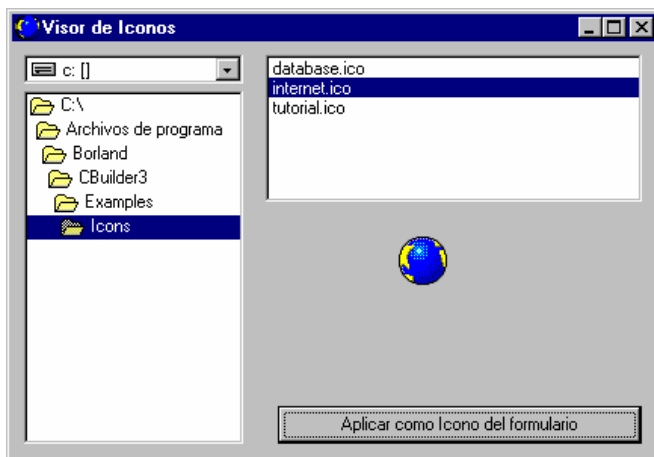
Con el fin de mostrar en la lista de archivos sólo aquellos que sean de tipo icono, habrá que asignar a la propiedad `Mask` el valor `*.ico`. Por último, dé el valor *“Aplicar como icono del formulario”* a la propiedad `Caption` del botón.

Cada vez que se seleccione uno de los archivos mostrados en el `TFileListBox` éste generará un evento `OnClick`, que aprovecharemos para recuperar el archivo y mostrar su contenido. Para ello tendremos que escribir el código siguiente:

```
void __fastcall TForm1::FileListBox1Click(TObject *Sender)
{
    if (FileListBox1->ItemIndex > -1)
        Image1->Picture->LoadFromFile(FileListBox1->FileName);
}
```

Y cuando se pulse sobre el botón habrá que asociar el código que se presenta a continuación.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    try
    {
        Form1->Icon->LoadFromFile(FileListBox1->FileName);
    } catch (EOpenError &)
    {
        ShowMessage("No hay ningún icono seleccionado");
    }
}
```



*Aspecto del visor de iconos*

## EJERCICIO PRÁCTICO 10. Eventos periódicos.

Hasta ahora todos los componentes que hemos conocido han sido controles, es decir, componentes visuales con los que es posible interactuar durante la ejecución. No todos los componentes de C++ Builder son de este tipo, existen también algunos que sin ser visibles desempeñan funciones muy importantes como se tendrá ocasión de ver en el manejo de Bases de Datos.

Uno de los componentes no visuales es `TTimer`, un componente cuya única finalidad es generar un evento a intervalos regulares con la frecuencia que nosotros mismos establezcamos. Este componente es el primero de la página `System`.

La frecuencia con la que el componente `TTimer` generará el evento `OnTimer` dependerá del valor que asignemos a la propiedad `Interval`, que por defecto es 1000. Esta medida está expresada en milisegundos, lo que quiere decir que por defecto el evento se producirá una vez cada segundo.

### En la práctica

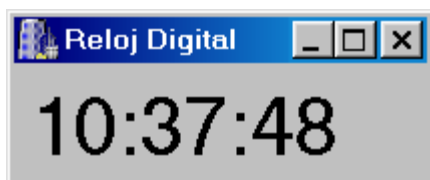
Vamos a realizar una aplicación que simplemente presente un reloj digital, mostrando la hora ocupando todo el espacio disponible del formulario.

Inserte un control `TLabel` y modifique el valor de la propiedad `Caption`, asignando la cadena `"hh:mm:ss"`. A continuación haga doble clic sobre la propiedad `Font`, en el Inspector de Objetos, y seleccione un tamaño de letra grande, ajustando el tamaño del formulario para que sea justo el necesario para mostrar la etiqueta de texto.

A continuación inserte un control `TTimer` y haga doble clic sobre su evento `OnTimer`, en el que escribiremos la sentencia que se muestra a continuación. La función `Time()` devuelve la hora actual, que asignaremos a la propiedad `Caption` del control `TLabel`.

```
Label1->Caption=Time();
```

El aspecto del programa en ejecución será el siguiente:



## EJERCICIO PRÁCTICO ESPECIAL. Trabajar con Paneles e Imágenes. Diseño e implementación de un sencillo juego.

Como ejercicio práctico y con el fin de comprobéis algunas posibilidades de la herramienta C++ Builder, vamos a realizar una aplicación para el entretenimiento del usuario. La aplicación contará con una ventana en la que aparecerán 16 paneles (numerados del 1 al 16) sobre los que el usuario podrá hacer clic con el ratón para hacer aparecer una imagen escondida asociada al panel. El objetivo del juego consistirá en buscar parejas de imágenes entre las 16 posibles, terminando cuando todas las parejas estén enlazadas de forma correcta.

Los controles por parte de la aplicación del buen funcionamiento del juego se escapan, ahora mismo, del objetivo del ejercicio, por lo que será el usuario el que se encargue de este aspecto. Más adelante se intentarán hacer mejoras.

Así pues, la apariencia que ofrecerá el juego en un momento determinado podría ser la siguiente:



Donde, como se observa, existen parejas de imágenes cuyas posiciones se establecerán al azar desde la propia aplicación.

### Conceptos previos al diseño de esta aplicación.

Para poder implementar este juego debemos saber lo siguiente:

1. Cualquier grupo de componentes que tengamos en un formulario pueden compartir un mismo código. Si se marcan (pulsando <Mayusc> y seleccionando los componentes) podemos acceder, en el Inspector de Objetos, a todos aquellos eventos que compartan los elementos seleccionados.

Cuando se accede al código asociado a un evento, el Borland C++ Builder genera la cabecera de la función donde aparece un parámetro denominado Sender, que es un puntero al componente asociado a ese código. El puntero Sender es del tipo genérico Tobject, del que están derivados todos los componentes.

Si quisiéramos acceder a una propiedad o un método, por ejemplo, del componente asociado, deberemos hacer una conversión del tipo Tobject al tipo que deseemos de la forma siguiente:

```
static_cast<tipo_de_componente_al_que_se_quiere_convertir *>(Sender)->propiedad|método
```



2. Un **TPanel** es un componente especial que funciona como un contenedor, al igual que un formulario o ficha. Las propiedades que vamos a usar de un panel son :

Caption.- Que da título al panel en el centro de éste.

Controls[].- Vector de elementos contenidos en el panel según el orden en el que se insertaron. Por ejemplo, si un botón está incluido en el panel, podremos acceder a su propiedad Caption de la forma siguiente: Panel1->Controls[0]->Caption="Salir"

Si el panel contuviera varios componentes accederíamos a ellos de la forma: Controls[0], Controls[1], Controls[2], etc.

3. Una **TImage** (de la paleta **Additional**) es un control que permite mostrar una imagen en el formulario obtenida desde un archivo en disco que puede estar en formato BMP, ICO o WMF.

Las propiedades que vamos a usar de una imagen son las siguientes:

Align.- Indica la alineación de la imagen con respecto al componente que la contiene. Si toma el valor alClient utilizará todo el espacio disponible del contenedor.

Stretch.- Tomará el valor verdadero cuando queramos que la imagen se adapte al tamaño del control.

Picture.- Es la propiedad más interesante de un Timage, puesto que su contenido hace referencia al archivo imagen asociado a este componente. A esta propiedad se le puede asignar un valor en fase de diseño simplemente haciendo doble clic sobre la propiedad en el Inspector de Objetos o bien, podremos asignarle un valor desde código usando el método LoadFromFile(). Por ejemplo, de la forma siguiente:

```
Image1->Picture->LoadFromFile("foto1.bmp");
```

4. Una vez insertado un componente en un formulario es posible marcarlo y copiarlo en el formulario tantas veces como deseemos. Imaginemos que ya hemos insertado un componente y que le hemos asignado unas propiedades concretas y también asociado unos eventos. Para que este componente pueda ser usado con esas características en ocasiones posteriores podemos usar la opción del menú principal **Component, Create Component Template**. Esta opción permite dar un nombre al control que queremos incluir como plantilla y, generalmente, se insertará en una nueva pestaña de la Paleta de Componentes denominada Templates. Este componente podrá ser usado cuando se quiera con sus propias características.

### Pasos a seguir para la realización del ejercicio

1. Crear un directorio de trabajo donde almacenaremos nuestro proyecto. Incluir en ese directorio al menos 8 imágenes .bmp.
2. Dar al formulario el título "El juego de las parejas".
3. Insertar un componente TPanel al que asignaremos el título '1' y asociaremos a su evento OnClick el siguiente código:

```
static_cast<TPanel *>(Sender)->Controls[0]->Visible=true;
```

4. Insertar un componente imagen Timage dentro del panel con la propiedad Visible a false. Asignar alClient a la propiedad Align, y true a la propiedad Stretch. Asociar a su evento OnClick el siguiente código:

```
static_cast<TImage *>(Sender)->Visible=false;
```

5. Marcar el panel junto con su componente imagen y acceder a la opción Create Component Template del menú inicial. Asignar el nuevo componente con el nombre MiPanel en la pestaña Templates.
6. Acceder a la pestaña Template e insertar 15 componentes más en el formulario de la forma que se vio en la figura de la página 21. Asignar a los paneles como título los números del 1 al 16.
7. Ya sólo queda, antes de ejecutar el programa, asignar los ficheros imagen a cada uno de los Timage que tenemos. Para ello pulsaremos sobre el evento OnActivate del formulario y escribiremos el siguiente código, que, partiendo de un conjunto de 8 imágenes que insertamos en un vector de tipo cadena (AnsiString en C++) crearemos otro vector del mismo tipo de 16 elementos con los ficheros que se van a asociar con cada imagen. La asignación posterior a cada imagen, por ahora, la realizaremos una por una.

```
void __fastcall TForm1::FormActivate(TObject *Sender)
{
    AnsiString tablaimagenes[8]={nombres de los 8 ficheros bmp que queremos utilizar};
    int esta[8]={0,0,0,0,0,0,0,0}; //Para comprobar las veces que se ha cargado una imagen
    AnsiString asociacion[16];
    int i,j;
    randomize();
    for (j=0;j<16;j++)
    {
        do
        {
            i=random(8);
            while (esta[i]==2);
            esta[i]++;
            asociacion[j]=tablaimagenes[i];
        }
        while (esta[i]==2);
        Image1->Picture->LoadFromFile(asociacion[0]);
        Image2->Picture->LoadFromFile(asociacion[1]);
        Image3->Picture->LoadFromFile(asociacion[2]);
        Image4->Picture->LoadFromFile(asociacion[3]);
        Image5->Picture->LoadFromFile(asociacion[4]);
        Image6->Picture->LoadFromFile(asociacion[5]);
        Image7->Picture->LoadFromFile(asociacion[6]);
        Image8->Picture->LoadFromFile(asociacion[7]);
        Image9->Picture->LoadFromFile(asociacion[8]);
        Image10->Picture->LoadFromFile(asociacion[9]);
        Image11->Picture->LoadFromFile(asociacion[10]);
        Image12->Picture->LoadFromFile(asociacion[11]);
        Image13->Picture->LoadFromFile(asociacion[12]);
        Image14->Picture->LoadFromFile(asociacion[13]);
        Image15->Picture->LoadFromFile(asociacion[14]);
        Image16->Picture->LoadFromFile(asociacion[15]);
    }
}
```

Por último, recordar que para usar las funciones relacionadas con los números pseudoaleatorios hay que incluir `<stdlib.h>`.