

CREANDO NUESTRA PROPIA CALCULADORA

Delphi: el hijo de Pascal

Alex VALENCIA
seija@anime.com.ar

Estudiante de Ciencias de la Computación y
experto en programación.

Generalmente, los **PRIMEROS PASOS** en el área de la programación se realizan con Pascal, debido a que sus sentencias no son complicadas ni difíciles de entender. La empresa **BORLAND** nos ofrece esta alternativa en **LENGUAJES VISUALES** que nos permitirán hacer nuestras propias aplicaciones con base en Pascal.

Delphi es un lenguaje de programación muy flexible y fácil de usar, sobre todo para aquellas personas que dominan el conocido lenguaje Pascal. Si bien en estos últimos años los lenguajes como Visual Basic o JAVA han tenido una gran repercusión dentro del mundo de la programación visual, Delphi no se queda atrás, y presenta las características típicas de los lenguajes visuales, lo que lo vuelve fácil de aprender para cualquiera que sepa usar este tipo de herramientas, o bien para quien quiera comenzar a usarlas. La interfase del programa resulta muy familiar para los programadores de Visual Basic. Contiene objetos muy similares y la mayoría de sus propiedades son las mismas (en la **Figura 1** se muestra la pantalla de un proyecto en blanco).

Sin embargo, existen ciertas diferencias entre este lenguaje y Visual Basic, desde del punto de vista del manejo de ciertas estructuras y, obviamente, sintáctico. El objetivo de esta nota es cubrir algunos aspectos básicos que nos resultarán útiles para nuestra calculadora de ejemplo y para guiarnos por los primeros pasos en Delphi.

Procedimientos y funciones

A diferencia de Visual Basic, el lenguaje Delphi usa procedimientos y funciones para ejecutar el código que utiliza nuestro programa. La diferencia principal entre ambos es que las funciones reciben parámetros y devuelven el resultado en el nombre de la función (como en VB). En cambio, los procedimientos se encargan de ejecutar un código prescrito que se basará en parámetros que se califican de “entrada”, “salida” y de “entrada/salida”, a través de los cuales enviaremos datos y recibiremos resultados. La forma de declararlos está en la **Tabla 1**.

Variables y constantes

A diferencia de otros lenguajes y al igual que C, Delphi usa referencias en las declaraciones de variable, lo que significa que utiliza punteros a una dirección donde se almacena el dato que queremos escribir o leer.

Esto lo vuelve más útil en algunas cosas; tal vez una de las más importantes sea el tipo de dato *string*, que es manejado

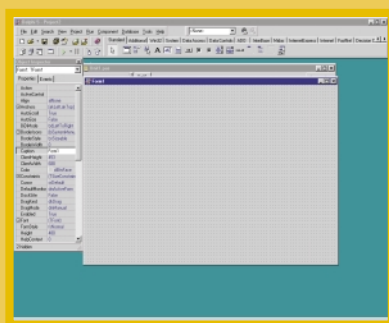


Figura 1. Ésta es la pantalla principal de Delphi. Como se ve, no es muy diferente de otros programas de lenguaje visual, por lo cual no nos resultará difícil familiarizarnos con él.

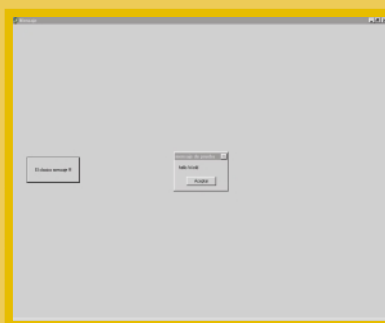


Figura 2. Aquí está nuestro primer programa. En una de esas, Bill nos compra los derechos para incluirlo en su próximo Windows.

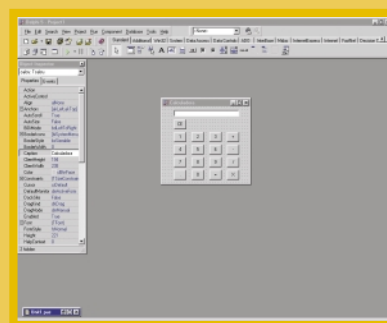


Figura 3. Aquí vemos cómo quedaría la interfase gráfica de la calculadora. Obviamente, podemos hacerla como más nos guste.



Si no tenemos Delphi, podemos bajarlo de la página oficial, www.borland.com/downloads, donde hay una versión trial de 30 días de duración.

como cadena, donde podremos indexar un carácter específico, lo que hace mucho más flexible el trabajo sobre este tipo de datos.

También encontraremos otros tipos de datos, que podremos utilizar en nuestras aplicaciones y que desarrollaremos a medida que avancemos en nuestra aplicación de ejemplo (se detallan en la **Tabla 2**). En esta ocasión, construiremos una sencilla calculadora.

El manejo de las constantes es similar que en los demás lenguajes, pero además, podremos usar constantes del tipo *typed*, que nos permitirán utilizar estructuras de datos (vectores, registros, punteros, etc.) como constantes.

El clásico mensaje

Para familiarizarnos y perderle el miedo a Delphi, empezaremos con el típico ejemplo del mensaje. En este caso, usamos el clásico "Hello World" u "Hola mundo", que aparecerá cuando presionemos un botón.

- 1) Creemos, dentro de un *form* vacío, sólo un botón, con el nombre **botón** y el *caption* **El clásico mensaje**. El *caption* será el texto que aparecerá en él.
- 2) Ahora hacemos doble clic para dirigirnos al código del evento *clíc*.
- 3) Escribimos el siguiente código:

```
begin {El Begin y End estan escritos por default}
application.messagebox('hello World', 'mensaje de prueba', MB_OK)
end;
```

Una vez terminado, podremos hacer correr el programa con <F9> y ejecutar esta "útil" aplicación. Esto debería quedar como se ve en la **Figura 2**. Delphi es más estricto que VB, ya que las funciones deben incluir todos sus parámetros; en caso contrario, mostrará un error.

Nuestra propia calculadora

Ahora comenzaremos a programar una aplicación un poco más útil y, de esta forma, podremos adentrarnos un poco más en el lenguaje Delphi. Haremos una calculadora con las funciones básicas, para lo cual la interfase gráfica quedará como nos muestra la **Figura 3**, con 16 botones y un objeto *memo* (debido a que nos permite alinear a la derecha), alineándolo con la propiedad **Align**.

Debemos tener en cuenta un par de cosas antes de empezar, como que hay que mantener el primer operando en el display hasta ingresar el otro, no perder el último operando para hacer que la tecla <=> funcione consecutivamente, etc., pero iremos viendo todo esto a medida que creemos las funciones.

Debido a que necesitaremos muchas comprobaciones (posibles acciones por parte del usuario), deberemos definir algunos *flags* o banderas que nos sirvan de guía. La **Tabla 3** muestra las variables que usaremos y definiremos como globales, o sea que se declaran fuera de los procedimientos (como en VB, pero sin la palabra **Public**) y debajo de la palabra reservada **VAR**.

Como toda buena calculadora, debe inicializarse con el 0, por lo cual en el evento **formcreate** de nuestra *form* escribiremos:

```
display.Text := '0'; {asignamos la cadena '0' al
```

Tabla 1: Parámetros

En esta tabla se indica la forma de declarar parámetros de entrada, salida y entrada/salida.

| Tipo de parámetro | Declaración |
|---------------------------------|------------------------------------|
| Entrada (por valor) | Procedure calc (X:integer) |
| Salida | Procedure calc (out num:integer) |
| Entrada/salida (por referencia) | Procedure calc(var X, Y: Integer); |

```
display}
{inicialización de flag}
```

Bien, ahora debemos hacer que los números nos respondan, así que creamos un procedimiento:

```
Procedure numero (display: TMemo; num:char; var
flag:boolean);
{ojo!!, hay que pasar al display por parámetro tam-
bién, otra cosa diferente a VB}
Begin
if (display.Text = '0') then {primero verificamos el
0}
    display.text:= num
else
begin
    if flag = true then {Este flag indica si esta-
mos en una operación}
begin
    display.text:= num ;
    flag:=false
end
else
    display.text:= display.text + num ;
end
end;
end;
```

Como muestra el código, debemos crear un flag (variable **boolean**) que pasará por parámetro y nos será útil para mantener el número del primer operador en el display cuando haya una operación activada.

Sólo basta colocar este procedimiento en el evento *Clic* de los botones de los números y el punto con los parámetros correctos, y listo:
numero (display,'1', flagop); {ejemplo para el botón de la tecla 1}

Para el punto (.) es diferente, ya que, si usamos el mismo procedimiento que utilizamos para los números, nos dará la opción de repetir más de una vez el carácter del punto (¿2.45.6? No es un número muy bonito). Por eso desarrollamos otro procedimiento para éste, que irá incluido directamente en el evento *Clic*, como se muestra a continuación, y donde usaremos la posibilidad de indexar un caracter para ve-

rificar la existencia del punto.

```
var
    aux:string;
    flagpun:boolean;
    j,i:integer;
begin
    aux:=display.text; {copiamos la cadena a una
variable auxiliar}
    i:=length(aux); {contamos sus caracteres con la
función 'length'}
    for j:=1 to i do {recorremos la cadena}
begin
    if aux[j] = '.' then {verificamos el punto}
        flagpun:=true
    end;
    if flagpun= false then {si no había, lo agrega-
mos}
        display.text:=display.text+'.'
    end;
end;
```

Cambio de tipo de datos

Aquí se nos presenta un importante problema, que será común a la mayoría de las aplicaciones que creamos en lenguajes visuales. Éste es el cambio de tipo de dato. Como se sabe, un objeto **memo** devuelve, en su propiedad **.text**, una cadena de caracteres y no un número real; por esta razón, si sumamos "1" + "2", el resultado será "12", ya que concatenamos texto. Debemos cambiar el tipo de dato a **real** para que los sume como números reales.

En VB existe la función **Val**, con la que pasamos por parámetro la cadena y nos devuelve el número; pero en Delphi no tenemos tanta suerte, y deberemos acudir a las variables de tipo **variant**, que nos permitirán cambiar el tipo de dato. Para esto, creamos dos funciones, **Textoreal** y **Realtotex**, que usaremos para este problema. Como son funciones, se trata de código transportable y reutilizable, y podremos usarlas en cualquier otra aplicación que creamos con sólo incluirla, o crear una **Unit** con nuestras propias funciones. El código sería el siguiente:

```
function textoreal (tex:string): real;
var
    aux: variant;
begin
    aux:= tex; {del texto al variant}
    textoreal:= aux; {del variant
al real}
end;

function realtotex (num:real):
string;
var
    aux: variant;
begin
    aux:= num; {del real al
variant}
    realtotex:= aux; {del variant
```

| Éstos son otros tipos de datos útiles que podemos utilizar en nuestras aplicaciones. | | |
|--|---|-----------------------------------|
| Tipo de dato | Descripción | Ejemplo |
| Enumerado | Dato de definición de valores | Type colores = (blue, green, red) |
| Puntero | Dato de dirección de memoria | Type pchar=^char |
| Subrango | Dato definido en un rango ordinal de un tipo de dato. | I := 0..99 I := 'A'..'Z' |

LISTADO*:

```

procedure opera (var operador:char;
operacion_actual:char ; display:TMemo; var
flagop,flagcons:boolean; var res:real);
begin
  {operador: último operador utilizado}
  {operación_actual: el nuevo operador}

  case operador of
    '+':
      begin
        if flagcons =false then
          begin
            numel:= textoreal(display.text);
            {cambiamos de tipo de dato}
            res:= suma(res,numel); {acumulamos el resultado}
            display.text:= realtotex (res); {mostramos el
            acumulador}
            flagop:=true
          end
        else
          begin
            flagcons:=false;
            flagop:=true
          end;
        end;

    '-':
      begin
        if flagcons =false then
          begin
            numel:= textoreal(display.text);
            res:= resta(res,numel);
            display.text:= realtotex (res);
            flagop:=true
          end
        else
          begin
            flagcons:=false;
            flagop:=true
          end;
        end;

    '*':
      begin
        if flagcons =false then
          begin
            numel:= textoreal(display.text);
            res:= producto(res,numel);
            display.text:= realtotex (res);
            flagop:=true
          end
        else
          begin

```

```

flagcons:=false;
flagop:=true
end;
end;

  '/':
  begin
    if flagcons =false then
      begin
        numel:= textoreal(display.text);
        res:= suma(res,numel);
        display.text:= realtotex (res);
        flagop:=true
      end
    else
      flagcons:=false;
      flagop:=true
    end;

    else {si no había ningún operador presionado...}
    begin
      if flagcons =false then
        begin
          flagcons:=false;
          operador :=operacion_actual;
          if flagpor=true then
            begin
              res:= textoreal(display.text);
              flagpor:=false;
              flagop:= true
            end
          else
            begin
              case operacion_actual of
                {verificamos cuál es la operación actual}
                '*':
                  begin
                    numel:= textoreal(display.text);
                    res:= producto(res,numel);
                    display.text:= realtotex (res);
                    flagop:=true
                  end;
                '/':
                  begin
                    numel:= textoreal(display.text);
                    res:= cociente(res,numel);
                    display.text:= realtotex (res);
                    flagop:=true
                  end;
                '+':
                  begin
                    numel:= textoreal(display.text);
                    res:= suma(res,numel);
                    display.text:= realtotex (res);

```

sigue en página 80

viene de página 79

```

flagop:=true
end;
\-' :
begin
numel:= textoreal(display.text);
res:= cociente(res,numel);
display.text:= realtotex (res);
flagop:=true
end;
end;
end;
end;
end;
end;
operador:= operacion_actual; {actualizamos el
operador}
end;
    
```

al texto}
end;

El procedimiento general

Como buenos programadores, debemos encontrar la generalidad en los procedimientos de **suma, resta, multiplicación y división**. No es muy difícil, porque todos actúan de la misma manera. Después de ingresar dos operandos, devuelven un resultado y listo. Lo único que debemos tener en cuenta son las posibles combinaciones de teclas que el usuario puede presionar. A continuación mostramos el código de este procedimiento; bastará con incluirlo en cada evento *Clic* de las operacio-

nes con los parámetros correctos, y asunto terminado. Recomendamos hacer la prueba de escritorio (seguimiento de la función de las variables con papel y lápiz) para que sea más comprensible, ya que explicar cada declaración nos llevaría unas cuatro o cinco revistas, porque el procedimiento contempla todas las posibles acciones por parte del usuario.

Con este procedimiento resolvemos el núcleo del programa. Lo agregamos a cada operación, y misión cumplida.

```

begin
opera(operador,'+',display, flagop, flagcons, res);
end;
    
```

Ahora nos falta asociar el código del botón **[CE]**, que sólo inicializa los *flags* y resetea el display.

```

procedure Tcalcu.resetClick(Sender: TObject);
begin
operador:= \ ' ; {no hay operaciones asociadas}
res:=0; {el resto se resetea}
display.Text := \0'; {reseteamos el display}
flagpor:= true;
flagcons:=false;
end;
    
```

Una vez terminado todo, estamos listos para compilar. Para ello utilizamos **[Project/Compile]**, y automáticamente se creará el archivo **EXE** de nuestra aplicación.

Antes de esto, podemos ponerle algún fondo o personalizarla de la manera que queramos. Ya estamos preparados para usar nuestra propia calculadora hecha en Delphi.

La proyección Delphi

Si bien Delphi no ha sido un lenguaje muy difundido o utilizado bajo entorno Windows como lo fue el famoso y popular Visual Basic, Linux lo vio con otros ojos. Su nueva versión del clásico sistema operativo, llamada **Kirix**, incluye gran parte de código de Delphi para las tareas de administración de procesos y de memoria. Los programadores y expertos en Linux aseguran que *“un buen programador debe saber programar en Delphi”*.

Entre otras cosas, Delphi se está orientando a la programación de sitios web dinámicos, lo que le abre el paso para competir con los lenguajes que están dominando esta área, como PHP, Pearl o ASP.

Por lo visto, ahora es muy necesario comenzar a entender Delphi, ya que parece ser un lenguaje prometedor que, en

Tabla 3: Variables del ejemplo

Éstas son las variables que usaremos como globales para la aplicación.

| Nombre | Tipo | Uso |
|------------|---------|---|
| Nume1 | Real | Operando auxiliar para los cálculos. |
| Operador | Char | Indica de qué operación se trata, +, -, * o /. |
| EFlagcons | Boolean | Indica si se presionó el '=' más de una vez. |
| Resultado | Variant | Variable utilizada para cambios de tipo de datos. |
| ERes | Real | Acumulador de resultados. |
| Flagpor | Boolean | Indica que se ha presionado el igual (=) y que se espera el primer operando. |
| EFlagpunto | Boolean | Indica que se ha ingresado un "." para no repetirlo. |
| Flagop | Boolean | Indica si se ingresó el primer operando para mantenerlo en el display hasta que se ingrese el otro. |

En el CD

Para que no tengas que copiar todo el código de los ejemplos, y para no tener problemas de errores, éste fue incluido en la carpeta PROGRAMACIÓN del CD que acompaña la revista. Además, ¡también encontrarás la calculadora funcionando!



BATALLA NAVAL BAJO DELPHI

Estructuras de datos

Alex VALENCIA
 avalencia@tectimes.com

Estudiante de Ciencias de la Computación,
 fanático del anime y la programación.

Ya dimos los **PRIMEROS PASOS** en este lenguaje al crear nuestra propia calculadora en Delphi.

Ahora **COMIENZA LO INTERESANTE**: haremos la batalla naval utilizando **ESTRUCTURAS DE DATOS**, una de las cosas más importantes que emplearemos para cualquier programa.

Las estructuras de datos son elementos que utilizaremos en cualquier programa que maneje un volumen grande de información, como, por ejemplo, una encuesta definida o cualquier aplicación que requiera un previo registro de datos.

Si debiéramos registrar cien tiempos por cronómetro en un programa, no sería óptimo crear una variable para cada tiempo, ya que causaría la lentitud del programa. Las variables se ubican en posiciones aleatorias de la memoria, y el acceso rápido de una a otra es mejor en una estructura, ya que las agrupa en alguna dirección con sus espacios consecutivos; de otra manera, el código sería horriblemente largo. El objetivo de las estructuras es “amontonar” espacios en memoria, a los que se asignará un nombre en común y un subíndice.

Algo importante es saber definir estructuras de datos que sean lo suficientemente pequeñas como para que entren bien en memoria, y así crear una aplicación transportable a máquinas con diferentes configuraciones; es decir, intentar que el programa use la menor cantidad de memoria posible para mejorar su velocidad y rendimiento. Por ejemplo, una

estructura tipo string sin indexar de 1.000 posiciones ocupará 256 KB; de modo que hay que economizar. A continuación, veremos las estructuras más utilizadas.

Vectores

Los vectores son una colección de una cantidad específica de datos de un tipo determinado. Por ejemplo, podemos utilizarlos para guardar un grupo de números. Podríamos guardar los números 14, 5, 19, 20 y 10 en un vector de cinco posiciones del tipo integer.

Matrices

A los que no hayan sufrido con ellas en la facultad les explicamos que las matrices son vectores de dos dimensiones y de un mismo tipo de datos, que se dirigen con dos coordenadas numéricas (representan filas y columnas). Veremos el ejemplo en la misma batalla naval, donde usaremos una matriz de 10 x 10 de tipo boolean para representar el tablero de juego. Existen matrices de más dimensiones, pero, para el caso, con ésta nos sobra.

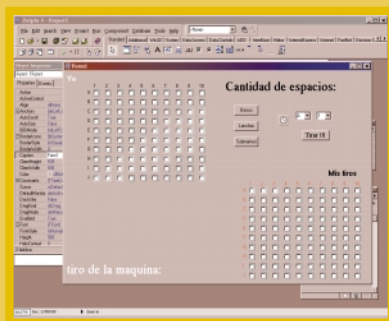


Figura 1. Así quedaría el tablero principal del juego. Ésta es una opción, pero podemos hacerlo como queramos.

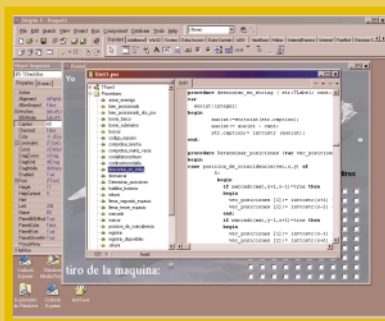


Figura 2. Éste es el code explorer. Cuando lo abrimos, lo tenemos al lado de la form, de modo de acceder a él rápidamente y que no se nos escapen las ideas.

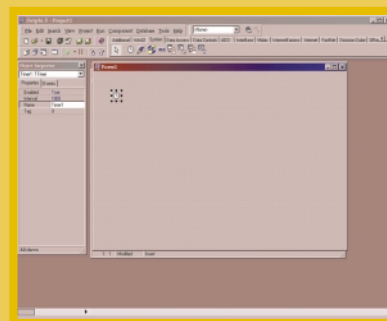
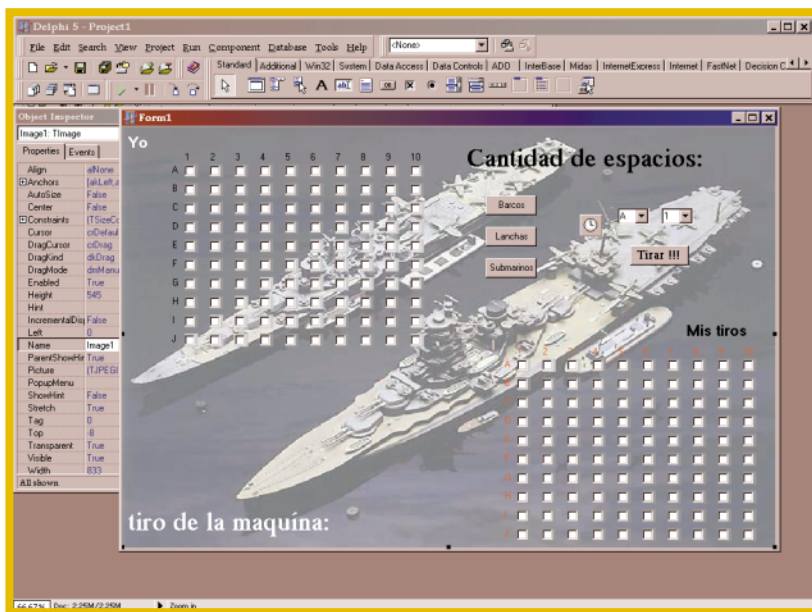


Figura 3. Éste es el objeto Timer, muy útil para cualquier cosa que hagamos. Sólo se ve en tiempo de diseño.



Aquí está el programa terminado. Con la herramienta Image de la solapa [Additional], arriba de todo, podemos agregar un fondo a la form. Es bastante incómodo, pero no figura la opción [Picture] de VB.

Registros

El registro es uno de los más importantes, ya que nos dará paso a los archivos; pero, para que nos demos una idea, es un conjunto de datos de diferentes tipos. Por sí solo, únicamente nos sirve para aplicarlo a archivos, que veremos más adelante; pero aplicado a vectores forma una estructura muy útil.

Vectores de registro

Los vectores de registro son el resultado de mezclar registros con vectores, y resultan inmensamente útiles en el momento de recolectar muchos datos de diferentes tipos, como en el ejemplo de la encuesta, ya que necesitaríamos varios datos de diferentes tipos (por ejemplo, nombre, edad, etc.). Es la estructura con la cual operan las bases de datos cuando levantan una tabla en memoria; el problema es que en el momento de crearlo, no solemos medirlo y nos hace saltar la RAM, así que ¡ojo con éste!

¡A zarpar!

Obviamente, este programa es mucho más complejo que la calculadora que creamos en el número pasado. Por esa razón, nos resulta imposible mostrar todo el código del programa en esta nota, y nos remitiremos sólo a las partes importantes, en las que trabajemos con las estructuras de datos. El resto del

código está dentro del CD que acompaña a la revista.

Podemos identificar dos grandes secciones en el programa: la que corresponde al manejo del usuario y la de la máquina. Recordemos que el usuario debe confeccionar en su tabla la posición de los diferentes objetos.

Para ésta usaremos cien checkbox colocados como muestra la **Figura 1**. En este caso, definimos tres tipos de objetos: lanchas (ocupan una sola posición), barcos (dos posiciones contiguas) y submarinos (tres posiciones contiguas). Nuestro tablero está relacionado con una matriz pública que copiará su configuración constantemente. La sintaxis de definición se encuentra en la **Tabla 1**, junto con las estructuras que ya mencionamos.

El tablero también consta de tres botones que corresponden a cada uno de los tipos de objeto. Al presionar uno de ellos, podremos ir ubicando cada objeto en nuestro tablero.

Debemos tener en cuenta una serie de reglas de llenado; recordemos que debemos verificar que estamos llenando una lancha, un barco o un submarino, para poder determinar posiciones y comprobar que los casilleros sean contiguos. En el **Listado 1** vemos el procedimiento general de las checkbox que usaremos, así que basta pegarlo en todas las propiedades **click** con los parámetros correctos, y problema resuelto.

Este procedimiento es importante porque nos resuelve la mitad del problema, pero no tiene nada de estructuras, ya que éstas están en los procedimientos y funciones que utiliza (recordemos que es uno general, lo que significa que usa la menor cantidad de código posible). Entonces veremos, por ejemplo, el procedimiento **Registrar_disponibles**, que devuelve un vector con las posiciones posibles para el segundo espacio de un barco, donde pasamos por parámetro la matriz que corresponde a nuestro tablero, la primera posición y el vector donde saldrán los datos (entrada y salida). Nota: la función **inttostr** pasa de entero a string.

Tabla 1: Definición de estructuras

| Estructura | Declaración |
|---------------------|--|
| Vector | Vec=array [1..100] of string; |
| Matriz | Mat=array [1..10] of array [1..10] of string; |
| Registro | Reg = Record Año:integer; Mes:string; Dia:1..31; {entero entre 1 y 31 End; |
| Vector de registros | Vec_reg= arrar [1..10] of reg; |

```

procedure registrar_disponibles (var vec:vector;
mat:matriz; x,y:integer);
begin
{recordemos que son ocho posiciones alrededor de un
casillero}
if mat [x-1,y+1] <> true then {registramos si no es-
ta marcado}
vec[1]:= inttostr(x-1) + inttostr(y+1); {lo re-
gistramos}
if mat [x,y+1] <> true then
vec[2]:= inttostr(x) + inttostr(y+1);
if mat [x+1,y+1] <> true then
vec[3]:= inttostr(x+1) + inttostr(y+1);
if mat [x+1,y] <> true then
vec[4]:= inttostr(x+1) + inttostr(y);
if mat [x+1,y-1] <> true then
vec[5]:= inttostr(x+1) + inttostr(y-1);
if mat [x,y-1] <> true then
vec[6]:= inttostr(x) + inttostr(y-1);
if mat [x-1,y-1] <> true then
vec[7]:= inttostr(x-1) + inttostr(y-1);
if mat [x-1,y] <> true then
vec[8]:= inttostr(x-1) + inttostr(y);
end;

```

```

function comprobar_matriz_vacia (mat:matriz):boo-
lean;
var
i,j:integer;
begin
comprobar_matriz_vacia:=true;

for i:=1 to 10 do {recorre las filas de la matriz}
begin
for J:=1 to 10 do {recorre las columnas}
begin
if (mat[i,j] <> false) then {si es true
ya no está vacía}
comprobar_matriz_vacia:=false;
end;
end;
end;

```

Armando al enemigo

Ésta puede ser la parte más difícil del programa, ya que debemos hacer que la máquina sola arme su propia flota de combate con la cual nos enfrentará y a la que nosotros tendremos que destruir. Para esta tarea usaremos la función **Random()**, a la que le pasaremos por parámetro el número máximo no incluido, hasta el cual nos podrá devolver un número al azar. Pueden ver el código completo en el **Listado 2** (página 82).

Éste es el procedimiento que llena automáticamente la matriz enemiga; obviamente, irá en el evento **Form_create** de nuestro programa, ya que es lo primero que debemos hacer. Además de la matriz enemiga, trabaja con vectores que registran las posiciones que marcó la máquina, para después poder saber qué derribamos.

El tema de los tiros y la funcionalidad consiste en interpretar a dónde quiere tirar el usuario, y con otro **random** hacer que la máquina nos tire, con el agregado de un vector que irá registrando nuestros tiros y los de la máquina, para que no haya repeticiones. Como no tiene mucho de nuevo para nuestro tema específico, no publicamos el código en la revista, pero pueden encontrarlo en el CD.

En la figura grande que está al principio de esta nota, vemos cómo queda nuestro programa terminado. Lindo, ¿no? Habrá notado que esta vez me tomé mi tiempo para diseñar una bonita interfase.

Como observamos, dentro de los corchetes ([]) se ubica el subíndice del objeto al que nos referimos, y la cantidad de números está determinada por las coordenadas que tenga la estructura. Obviamente, usaremos estructuras de repetición para referirnos a todos los objetos, como **for** o **while**; así lo muestra la función **comprobar_matriz_vacia**, que utilizamos para decidir si el juego terminó o no. En la **Tabla 2** hay algunos procedimientos y funciones del programa que trabajan con estructuras; les recomiendo que los miren para ver cómo funcionan.

Como se habrán dado cuenta, este programa está muy “procedimentado”, lo cual es muy bueno para tener un código ordenado. Delphi no ofrece el **code explorer (Figura 2)** que nos muestra en forma de árbol todos los procedimientos, funciones y variables que hayamos creado. Lo bueno de esto es que, si hacemos doble clic sobre el nombre de la función o el procedimiento, nos manda directamente a éste. Esto resulta muy útil en caso de que el código sea demasiado largo, pero si está bien “procedimentado”, el **code explorer** se vuelve indispensable.

Tabla 2: Procedimientos y funciones que usan estructuras

| Procedimiento o función | Descripción | Estructura que usa |
|--------------------------|---|-----------------------------------|
| Marcar | Marca una posición (pone en true) en la matriz. | Matriz de 10 x 10. |
| Desmarcar | Desmarca una posición (pone en false) en la matriz. | Matriz de 10 x 10. |
| Marcado | Avisa si cierta coordenada está marcada o no. | Matriz de 10 x 10. |
| Bien_posicionado | Avisa si el segundo espacio de un barco o un submarino es consecutivo o no. | Vector de 8 posiciones de string. |
| bien_posicionado_dos_pos | Avisa si el tercer espacio de un submarino es consecutivo o no. | Vector de 2 posiciones de string. |
| Ver_marc | Busca si hay casilleros marcados alrededor de uno específico. | Matriz de 10 x 10. |



LISTADO 1:



```

procedure codigo_espacio (cordenada:string; Cant_espacios:TLabel; but_barcos, but_lanchas, but_submarinos:TButton; box:TCheckBox);
var
    auxint: integer;
    str:string;
begin
    cordalfatocordnum(cordenada,y,x); {cambia de coordenadas alfanuméricas a numéricas}
    if marcado (mat,y,x) = false then {verifica que esté marcado}
    begin
        if (cant_espacios.Caption <> '') and (cant_espacios.Caption <> '0') then
        begin
            {Código para el barco}
            if barco= true then
            begin
                if cant_espacios.caption = '2' then
                begin
                    registrar_disponibles (posiciones_libres,mat,x,y);
                    {registra las posiciones posibles para el 2º espacio}
                    marcar(mat,x,y); {marca el espacio}
                    descontar_en_string(cant_espacios,1);
                end
            else
            begin
                if cant_espacios.caption='1' then
                begin
                    if bien_posicionado(posiciones_libres,x,y) = true then
                    begin
                        marcar(mat,x,y);
                        descontar_en_string(cant_espacios,1);
                        habilitar_botones (but_barcos,but_lanchas,but_submarinos);
                    end
                else
                begin
                    if flagmens =true then
                    begin
                        Application.MessageBox('debe marcar un casillero contiguo','Error de marcado',MB_OK);
                        flagmens:=false;
                    {box' es un objeto Tcheckbox pasado por parámetro de entrada/salida}
                        box.checked:=false;
                    end;
                    flagmens:=true;
                end;
            end;
        end;
    end;

```

```

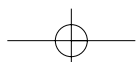
        end;
    end;
end;

{código para lancha}
if lancha= true then
begin
    marcar(mat,x,y);
    descontar_en_string(cant_espacios,1);
    habilitar_botones (but_barcos,but_lanchas,but_submarinos);
end;

{código para submarino}
if submarino= true then
begin
    if cant_espacios.caption = '3' then
    begin
        registrar_disponibles (posiciones_libres,mat,x,y);
        marcar(mat,x,y);
        descontar_en_string(cant_espacios,1);
    end
    else
    begin
        if cant_espacios.caption='2' then
        begin
            if bien_posicionado(posiciones_libres,x,y) = true then
            begin
                marcar(mat,x,y);
                descontar_en_string(cant_espacios,1);
                determinar_posiciones(vector_dos_posiciones,posiciones_libres,mat,x,y); {devuelve las dos posiciones posibles en los extremos para el 3º espacio del submarino}
            end
            else
            begin
                if flagmens =true then
                begin
                    Application.MessageBox('debe marcar un casillero contiguo','Error de marcado',MB_OK);
                    flagmens:=false;
                    box.checked:=false;
                end;
                flagmens:=true;
            end;
        end
    else
    begin

```

sigue en página 80



viene de página 79

```

                                if cant_espacios.caption='1'
                                end;
then                                flagmens:=true;
                                end;
                                begin                                end;
                                if bien_posicionado_dos_pos(vector-                                end;
                                _dos_posiciones,x,y)=true then                                end;
                                begin                                end;
                                marcar(mat,x,y);                                end;
                                descontar_en_string(cant_espacios,1);                                end;
                                habilitar_botones (but_barcos-                                end;
                                ,but_lanchas,but_submarinos);                                else
                                end                                begin
                                else                                box.checked:=false;
                                begin                                end;
                                if flagmens =true then                                end;
                                begin                                else
                                Application.MessageBox('debe mar-                                begin
                                car un casillero contiguo','Error de marcado',M-                                box.checked:= true;
                                B_OK);                                end;
                                flagmens:=false;                                end;
                                box.checked:=false;
    
```

Algunas cosas importantes

Noten que en este ejemplo utilizamos dos matrices de 10 x 10 posiciones booleanas. A simple vista, parece una barbaridad, pero el espacio que ocupa un dato booleano en memoria es de un solo byte (8 bits). Por lo tanto, la matriz completa ocupa sólo cien bytes (mucho menos de lo que ocupa un dato de tipo string).

Es importante fijarse bien en el tipo de datos que guardamos en las variables. Por ejemplo, en los vectores que usamos para registrar las posiciones de la máquina, lo más grande que se puede guardar es el par de coordenadas 10,10 (j10 en coordenadas alfanuméricas), por lo cual no se justifica la utilización de un vector de tipo string completo que ocupe 256 bytes por posición. Un vector indexado de cuatro posiciones (que ocupa sólo 4 bytes) es más que suficiente para los datos que va a guardar, ya que un string no es más que una cadena de caracteres, que, por ser un tipo de datos encapsulado, no podemos

modificar. Para más detalles, observen la **Tabla 3**, que explica cuánto ocupa cada tipo de dato en memoria.

Este tipo de aspectos, a los que si tenemos una máquina grande generalmente no les damos importancia, resultan fundamentales para que nuestro código sea transportable a la mayor cantidad de máquinas posible. Quizás en un sistema pequeño no influya, pero sí en sistemas que manejen un volumen mayor de datos.

Otra característica importante para resaltar es la función del objeto Timer (**Figura 3**), que es sumamente útil para cualquier aplicación que se nos ocurra hacer. Este objeto cumple la función de temporizador, que, a partir de cierto tiempo de activación, ejecuta la porción de código que le asignemos al evento. En este caso lo usamos para separar un tiempo entre los tiros del usuario y los de la máquina.

OK. Todo listo para incluir nuestra batalla naval en los juegos de Windows, así que prepárense para fundar su propia empresa de juegos bajo Delphi.

| Tabla 3: Tipos de datos | |
|--|--|
| Éstas son las variables que más se utilizan. La cantidad de espacio varía según el subtipo de variable (ej.: Shortint o Smallint). | |
| Nombre | Espacio que ocupa |
| Boolean | 1 byte |
| String | 256 bytes |
| Integer | 32 bits |
| Char | 1 byte |
| Real | 4 a 10 bytes (depende de los dígitos significativos) |

Algunas sugerencias para practicar

Como habrán visto hasta acá, falta agregar un montón de cosas al programa para perfeccionarlo y refinarlo, así que, para que puedan practicar, les recomiendo que intenten agregarlas por su cuenta, ya que con lo que vimos es suficiente. Aquí les sugiero algunas de ellas.

- La opción de definir la cantidad de barcos, lanchas y submarinos con los que queremos jugar.
- Poner la opción de reiniciar el juego una vez terminado.
- Porcentaje de tiros acertados de ambos jugadores.
- Crear varias interfases gráficas, y que vayan cambiando cada vez que reiniciemos el juego.

LISTADO 2:



```

procedure armar_enemigo(var mat:matriz);
var
  i,x,y:integer;
begin
  randomize; {procedimiento que usamos para
inicializar el random}
  for i:=1 to 5 do
  begin
    x:=random(11); {random de 0 hasta 10; es-
ta función viene con Delphi}
    if x = 0 then
      x:=x+1;
    y:=random(11);
    if y = 0 then
      y:=y+1;

    if marcado(mat,y,x) = false then {¿no estaba
esta lancha?}
    begin
      marcar (mat,x,y);
      {registramos en un vector las lan-
chas que puso}
      lanchas_ene[i]:= inttostr(x)+inttostr(y)
    end
    else
    begin
      randomize;
      x:=random(11);
      if x = 0 then
        x:=x+1;
      y:=random(11);
      if y = 0 then
        y:=y+1;
      marcar (mat,x,y);
      lanchas_ene[i]:= inttostr(x)+inttostr(y);

    end;

  end;

end;

{llenamos los barcos}
for i:=1 to 3 do
begin
  x:=random(11);
  if x = 0 then
    x:=x+2;
  y:=random(11);
  if y = 0 then
    y:=y+2;

  if marcado(mat,y,x) = false then
  begin

```

```

    marcar (mat,x,y);
    {registramos en un vector los bar-
cos que puso}
    barcos_ene[i,1]:= inttostr(x)+inttostr(y)
  end
  else
  begin
    randomize;
    x:=random(11);
    if x = 0 then
      x:=x+1;
    y:=random(11);
    if y = 0 then
      y:=y+1;
    marcar (mat,x,y);
    barcos_ene[i,1]:= inttostr(x)+inttostr(y);
  end;

  {calculamos los espacios posibles
para el 2º espacio}
  registrar_disponibles(posiciones_libres,mat,x,y);
  {Este procedimiento hace un random
de 1 a 8 para determinar la dirección de
la 2º posición}
  llenar_segundo_espacio(posiciones_libres,x,y);
  if marcado (mat,y,y) <> true then
  begin
    marcar (mat,x,y);
    barcos_ene[i,2]:= inttostr(x)+inttostr(y)
  end
  else
  begin
    registrar_disponibles(posiciones_libres,mat,x,y);
    llenar_segundo_espacio(posiciones_libres,x,y);
    marcar(mat,x,y);
    barcos_ene[i,2]:= inttostr(x)+inttostr(y);
  end;
end;

{llenamos los submarinos}
for i:=1 to 2 do
begin
  x:=random(11);
  if x = 0 then
    x:=x+3;
  y:=random(11);
  if y = 0 then
    y:=y+3;

  if marcado(mat,y,x) = false then
  begin
    marcar (mat,x,y);
    submarinos_ene[i,1]:= inttostr(x)+inttostr(y)
  end

```

```

else
begin
  randomize;
  x:=random(11);
  if x = 0 then
    x:=x+3;
  y:=random(11);
  if y = 0 then
    y:=y+3;
  marcar (mat,x,y);
  submarinos_ene[i,1]:= inttostr(x)+inttostr(y);
  end;
  registrar_disponibles(posiciones_libres,mat,x,y);
  llenar_segundo_espacio(posiciones_libres,x,y);
  if marcado (mat,y,x) <> true then
  begin
    marcar (mat,x,y);
  submarinos_ene[i,2]:= inttostr(x)+inttostr(y)
  end
  else
  begin
    registrar_disponibles(posiciones_libres,mat,x,y);
    llenar_segundo_espacio(posiciones_libres,x,y);
    marcar(mat,x,y);
    submarinos_ene[i,2]:= inttostr(x)+inttostr(y);
  end;
  end;
  determinar_posiciones(vector_dos_posiciones,posi-
ciones_libres,mat,x,y);
  llenar_tercer_espacio(vector_dos_posiciones,x,y);
  if marcado (mat,y,x) <> true then
  begin
    marcar (mat,x,y);
    submarinos_ene[i,3]:= inttostr(x)+inttostr(y)
  end
  else
  begin
    determinar_posiciones(vector_dos_posiciones,posi-
ciones_libres,mat,x,y);
    llenar_tercer_espacio(vector_dos_posiciones,x,y);
    marcar(mat,x,y);
    submarinos_ene[i,3]:= inttostr(x)+inttostr(y);
  end;
end;
end;

```

Delphi: el hijo de Pascal

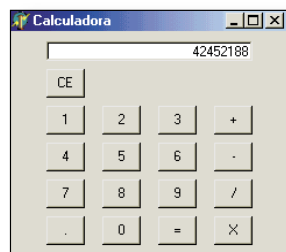
El lector **Leandro Lorge** nos hizo llegar sus críticas sobre la nota de Delphi publicada en USERS #117. En primer lugar, nos aclara que no pudo encontrar el código fuente en el CD de la revista. Es cierto, el código no apareció, por eso decidimos publicarlo en el CD de este número, en la carpeta **PROGRAMACIÓN**.

En segundo lugar, nos hace algunas aclaraciones de cómo mejorar el código. Por ejemplo, nos cuenta que justificamos la utilización del objeto *memo* para el display, ya que se puede alinear a la derecha, pero que el objeto *label* también lo permite y pesa menos que un *memo*.

También nos hace notar que dijimos que para solucionar el problema del cambio de tipo de dato, no tenemos la suerte de contar con un equivalente de la función **Val** de Visual Basic y deberemos acudir a variables de tipo **variant**. Sin embargo, nos recuerda que en Delphi existen varias funciones de conversión entre distintos tipos de datos, como **StrToFloat** y **FloatToStr**, que incluso son mucho más rápidas.

Otra cosa que recomienda es evitar las repeticiones de código para mejorar su rendimiento y velocidad. Además pide que en ediciones próximas tabulemos las líneas de código para hacerlo más legible.

También recibimos una alter-



Un lector nos envió sus críticas e interesantes mejoras para el proyecto de la calculadora de USERS #117.

nativa para el código de los botones numéricos de la calculadora. En el *caption* de cada botón tenemos su número, por lo que podemos saber qué botón se presionó tomando la propiedad **caption** del objeto disparador del evento, con lo cual podemos asociar el evento **OnClick** de todos los botones numéricos a un único procedimiento, parecido al siguiente:

```

procedure TForm1.Button1Click(Sender: TObject);
begin
  if Flagop then
    begin labell.caption:=(Sender As
Tbutton).caption;
                                Flagop:=false
    end
  else labell.caption:=labell.caption+(Sender
As Tbutton).caption
end;

```

De esta forma evitamos tener que escribir una llamada diferente para cada botón cambiando los parámetros que se pasan, y ya no es necesario ni pasar por parámetro ni calificar al objeto que hace las veces de display (en este caso, **label1**).

Agradecemos tus sugerencias y te aseguramos que las tendremos en cuenta. De todos modos, queremos contarles a todos los lectores que la nota de Delphi está pensada para los usuarios que conocen poco del lenguaje, y si bien hay muchas funciones y operaciones que se pueden hacer de diferentes maneras para optimizar el código, la nota sólo persigue un fin didáctico. ❌

¡¡¡TIEMBLA WORD!!!

Archivos en Delphi

Alex VALENCIA
 avalencia@tectimes.com

Estudiante de Ciencias de la Computación,
 fanático del anime y la programación.

Ya les tomamos la mano a las variables y probamos las estructuras de datos bajo **DELPHI**. Estamos preparados para dar el próximo paso hacia los **ARCHIVOS**, que nos permitirán crear **APLICACIONES MÁS PROFESIONALES** y más útiles. Comenzaremos con los de texto hasta llegar a los binarios.

Los archivos son una parte fundamental de cualquier lenguaje de programación, ya que gracias a ellos podemos utilizar nuestro disco rígido para almacenar datos que después procesaremos en las aplicaciones. Hay dos tipos de archivos bien distinguidos: el de texto y el binario. En esta oportunidad nos abocaremos a conocer el funcionamiento de los primeros, que nos servirán para almacenar cualquier cosa que deseemos en forma de texto. Pero antes veamos una definición de archivo.

Definición de archivo

Muchas veces hablamos de archivos como de algo corriente, pero cuando comenzamos a trabajar con ellos debemos comprender su estructura, cómo funcionan y cómo son realmente.

Un archivo es un espacio de memoria (recordemos que memoria no es sólo la RAM, sino también un CD-ROM o un disco rígido) con principio y fin, donde es posible almacenar datos interpretables por la máquina. La PC acude a la FAT (*File Allocation Table*) para ver en qué dirección empieza el archivo que solicitamos, y lo lee hasta encontrar su final.

Esta pequeña definición, que tal vez sea bastante fácil de deducir, es la clave para aprender cómo funciona un archivo y cómo debemos trabajar con él.

Los archivos de texto

El archivo de texto no es otro que el clásico y famoso **.txt**, que utilizaremos en esta oportunidad. Ahora quizá relacionen otras extensiones con este tipo de archivo, como **.doc** (Word) o **.sam** (Ami pro). Estos archivos también son de texto, aunque si los abren con WordPad, por ejemplo, verán texto mezclado con ASCII. Esto se debe a que dichos signos son interpretados por el procesador de textos, y de esta forma podemos usar fonts u otros efectos de los que ya estamos acostumbrados a usar en cualquiera de estas aplicaciones. Las fonts no son más que gráficos que el procesador de textos relaciona con la matriz de caracteres que se encuentra en la ROM de video y, a partir de ahí, las vemos en nuestro monitor. Pero sin profundizar mucho más en el tema, comencemos con nuestro procesador de textos basado en Delphi, hecho por y para nosotros mismos.

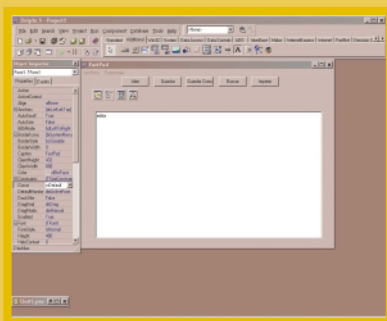


Figura 1. Nace nuestro nuevo editor, creado por nosotros mismos. Así podría ser la primera fachada que le demos. Sean creativos con la interfase.

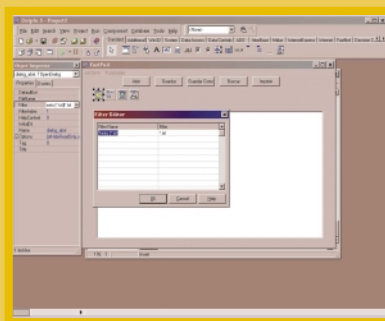


Figura 2. Aquí configuramos los filtros de archivos que queremos leer, con las famosas wildcards (*.txt, *.doc, etc.), para encontrar fácilmente cualquier archivo.

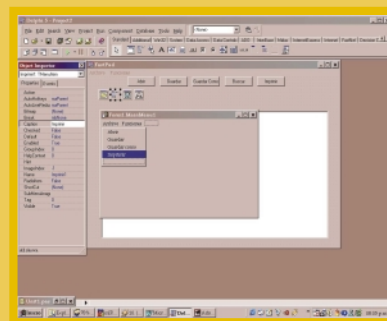
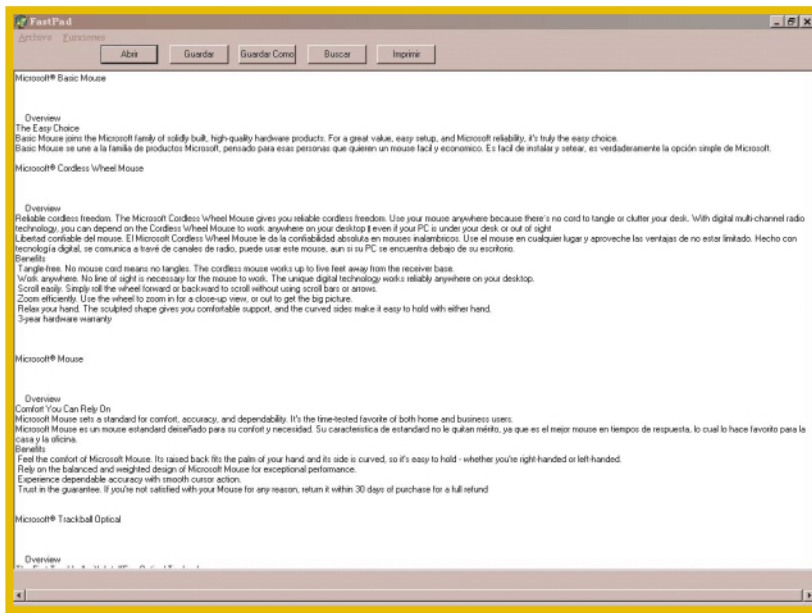


Figura 3. Ésta es una herramienta muy sencilla para crear un menú; simplemente le ponemos un nombre y le asociamos código al seleccionarlo. El código se ejecutará según los eventos.



Digámosle adiós al odioso WordPad y démosle la bienvenida al FastPad. No sólo es más rápido para cargar, sino que lo hicimos nosotros mismos y lo vimos crecer, snif.

Cambiando WordPad por FastPad

FastPad es el nombre de nuestro programa, que reunirá las opciones que más utilizamos en WordPad (**[Abrir]**, **[Guardar]**, **[Guardar como]**, **[Buscar]** e **[Imprimir]**) en cuatro botones y un menú que contendrá las mismas funciones. Les recomiendo que, a partir de esto, vayan agregándole cosas para ver las distintas herramientas que Delphi ofrece.

En nuestro form principal ubicaremos los objetos que describe la **Tabla 1**, con los que nos quedará algo como lo que muestra la **Figura 2**. Una vez hecho esto, estamos listos para comenzar.

Abrir un archivo

En este programa hay dos funciones fundamentales que realmente nos importan: **Abrir** y **Guardar**, ya que con ellas abriremos archivos, ya sea para escribir o para leer, y los almacenaremos.

Para trabajar con un archivo debemos abrirlo, lo que significa buscar el comienzo de éste y empezar a usarlo. Cuando nosotros pedimos, por ejemplo, el directorio de una carpeta, lo que vemos no es más que un reflejo de lo que la FAT tiene almacenado. La máquina sólo toca la FAT cuando creamos, bo-

ramos o modificamos (ABM), ya que no es otra cosa que una base de datos de archivos. Cuando abrimos el archivo, la FAT nos dice en qué posición del disco empieza para poder comenzar a usarlo.

Para los fines de la programación, en el momento en que abrimos un archivo de texto, éste se convierte en una variable del tipo **textfile** (en el caso particular de Delphi), y a partir de ahí podremos trabajar con ella como lo hacemos con cualquier otra variable.

Hay varias formas de abrir un archivo, pero las más importantes son dos: lectura o escritura. Existe una tercera, que es para lectura y escritura, pero como ven, desciende de las anteriores. En la **Tabla 2** se indican las sintaxis y las maneras de abrir archivos.

Abriendo un archivo para lectura

Ahora nos encargaremos de abrir nuestro archivo de texto para leerlo. Como habrán imaginado, esta acción va asociada a la función **Abrir** de nuestro programa. Como no queremos que nuestro programa sea menos que cualquier otro de Windows, usaremos para esto un objeto del tipo TopenDialog, que

Tabla 1: Objetos de la aplicación

Objetos que usaremos en nuestra aplicación FastPad.

| Nombre del objeto | Descripción |
|-------------------|--|
| RichEdit | Es un campo editor como el memo o el textbox, que nos permite trabajar con texto enriquecido (usar fonts y otros efectos), además de darnos información del texto (medidas, párrafos, etc.). Está hecho, precisamente, para un procesador. |
| Botones (x 4) | No hay demasiada explicación para éstos... |
| OpenDialog | Es la clásica caja de diálogo para abrir archivos, sólo se ve en tiempo de ejecución. |
| SaveDialog | Ídem OpenDialog, pero para guardar. |
| FindDialog | Es la caja de diálogo para buscar palabras (de cualquier editor de texto). |
| MainMenu | Objeto que nos permite crear la barra de comandos superior del form. La Figura 1 nos muestra la herramienta para crearlo. |

en su método **execute** no permite utilizar el tan conocido diálogo de **Abrir**, que deberemos configurar (Figura 3). Pero veamos esto directamente en el código que está asociado al evento **clíc** del botón **[Abrir]**, que lo único que hace es llamar al **TopenDialog**:

```
begin
    dialog_abrir.Execute; {llamamos al TopenDialog}
end;
```

Ahora asociamos el código para abrir el archivo a la función de clic del botón **[Abrir]** del **TopenDialog**:

```
var
    auxtext:string;
begin
    editor.text:='';{reseteamos el contenido del cuadro TrichEdit}
    filename:= dialog_abrir.FileName;
    {filename es una variable string}
    AssignFile(text_file,filename);
    {asignamos el archivo externo a una variable tipo textfile}
    reset(text_file); {abrimos para lectura !!}
    while not eof(text_file) do
    {mientras no lleguemos al final del archivo}
        begin
            readln(text_file,auxtext);
            {leemos una línea y la ponemos en auxtext del tipo string}
            editor.text:=editor.text+aux-
            text+chr(13)+chr(10);
            {Ascii 13=return of carry (cursor al principio), Ascii 10=line feed (nueva línea)}
        end;
        closefile(text_file); {cerramos el archivo siempre después de usarlo}
    end;
```

Observen una cosa muy importante: para leer el archivo usamos el comando **readln**, que lee la línea hasta encontrar el **EOL (End Of Line)**; éste viene dado por un carácter ASCII que indica dicho fin. Como vamos a abrir el archivo respetando las líneas (si no, veríamos cualquier cosa), después de cada línea asignamos caracteres ASCII con la función **chr()**, que asigna el carácter ASCII correspondiente al número que pasemos por parámetro. Por eso pasamos el 13 y el 10, que harán retroceder el cursor y crearán una nueva línea.

Tabla 2: Modos de apertura

Éstas son las distintas maneras de abrir un archivo.

| Sentencia | Tipo de apertura |
|-----------|----------------------|
| Reset | Sólo lectura. |
| Rewrite | Sólo escritura. |
| Append | Lectura y escritura. |

¡Hora de escribir!

Bien, ya sabemos cómo leer; ahora nos falta la otra mitad: aprender a guardar, o lo que se traduce como abrir archivos para escritura. Esta acción está asociada al botón **[Guardar como]**, que, a su vez, está vinculado al objeto **TSaveDialog**. El código es:

```
var
    auxtext:string ;
begin
    filename:= dialog_salvar.FileName;
    {filename almacena el path + nombre del archivo a crear}
    AssignFile(text_file,filename+'.txt');
    {asignamos a text_file el nombre del archivo para crear}
    rewrite(text_file); {abrimos para escribir}
    write(text_file,editor.text);
    {ponemos lo que hay en el editor en el archivo}
    closefile(text_file); {cerramos el archivo}
end;
```

Si sólo quisiéramos guardar en el archivo abierto, entonces el código estaría asociado al botón **[Guardar]**, y la diferencia de éste con el código de **[Guardar como]** es que no levantamos el nombre del archivo de **TSaveDialog**, sino que lo guardamos en el archivo que ya se encuentra en la variable **filename**.

Buscando e imprimiendo

Aparte de las funciones que trabajan netamente con archivos, también agregamos estas dos funciones, que son muy simples de ver y harán que nuestro **FastPad** se vea un poco más profesional.

El código que va asociado al diálogo del objeto **TFindDialog** es el que utilizan todos los procesadores de texto para la función **Buscar**, y como es tan empleado, Delphi lo provee en el mismo archivo de ayuda. Por eso, lo único que nos queda por hacer es aplicarlo a nuestro programa, por lo que quedaría así:

```
begin
    with editor do
        begin
            {Empieza desde el lugar donde se quedó la última búsqueda}
            {Si no, empieza desde el comienzo del texto}
            if SelLength <> 0 then
                StartPos := SelStart + SelLength
            else
                StartPos := 0;
            {ToEnd es la longitud desde la posición inicial hasta el final del texto en el Richedit}
            ToEnd := Length(Text) - StartPos;
            FoundAt := FindText(dialogo_buscar.FindText,
            StartPos, ToEnd, [stMatchCase]);
            if FoundAt <> -1 then
                begin
                    SetFocus;
```

```
SelStart := FoundAt;
SelLength := Length(dialogo_buscar.FindText);
end;
end;
end;
```

Eso es todo para la función **Buscar**. La función **Imprimir** es mucho más fácil; sólo debemos asociar esta línea al botón **[Imprimir]**, y listo:

```
begin
editor.Print (filename); {función para imprimir el
archivo pasado por parámetro}
end;
```

Obviamente, podríamos haber hecho esto también con un objeto tipo TPrintDialog para tener aparte la opción de elegir cantidad de copias, configuración de impresoras, etc. Generalmente usamos el shortcut de impresión rápida, pero si creen necesaria esta opción, es muy fácil agregarla, y les podrá servir de práctica para aprender a manejar mejor otros diálogos para futuras aplicaciones. Como el objetivo de esta nota es el manejo de archivos en particular, lo vamos a dejar ahí.

Si todo salió bien, nuestro FastPad quedará como lo muestra la **Figura 4**.

Como siempre, les aconsejo que sigan ampliando esta aplicación para poder conocer otras herramientas de Delphi. Acá les tiro algunas sugerencias:

- 1) Agregar la opción para fonts.
- 2) Crear la personalización de la impresión (TPrintDialog).
- 3) Crear la opción de reemplazar.
- 4) Poder agregar la opción para copiar al clipboard.

Algo más acerca de archivos

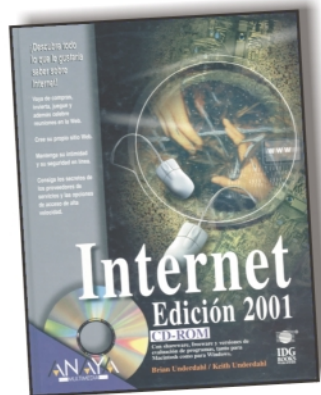
Si bien usamos un procesador de textos para poder entender el funcionamiento de un archivo, los archivos manejan todo el ámbito de nuestra máquina, no sólo en forma de programas sino que, mirándolo desde el punto de vista de la programación, son indispensables dentro de cualquier aplicación. Por ejemplo, con ellos podemos crear flags permanentes, que es donde las aplicaciones se fijan para conservar configuraciones. Los famosos ***.ini** no son más que archivos de texto que informan a su creador cómo fue configurada cada aplicación.

Por otro lado, los archivos de registro, generalmente llamados binarios, son la raíz donde se fundan todas las bases de datos, pero veremos este tema en el próximo número. La administración de memoria "Swap" está basada en el manejo de archivos. Éstos son algunos ejemplos, en el próximo



Microsoft Project 2000.
Paso a Paso.
de Carl S. Chatfield

\$ 26



La Biblia de Internet.
Edición 2001.
de Brian Underdahl

\$ 68



cusptide.com

Tel.: 4322-8868

e-mail: libros@cusptide.com

- Florida 628. Buenos Aires
- Suipacha 764. Buenos Aires
- Av. Gral. Paz 57. Córdoba
- Av. Santa Fe 1818. Buenos Aires
- Av. Córdoba 2067. Buenos Aires
- Suipacha 1045. Buenos Aires
- Village Recoleta
- Village Pilar
- Village Rosario
- Vicente López 2050. Buenos Aires
- Ruta Panamericana km. 50. Pilar
- Av. Eva Perón 5856. Rosario

CÓMO TRABAJAR CON ELLOS DESDE DELPHI

Archivos binarios

Alex VALENCIA
 avalencia@tectimes.com

Estudiante de Ciencias de la Computación,
 fanático del anime y la programación.

Ya estamos a pasos de sacarle el jugo a nuestro disco rígido. Con lo que aprendimos sobre archivos de texto, sólo abarcamos una parte de lo que podemos hacer. Ahora nos falta la otra mitad: **LOS ARCHIVOS BINARIOS**, que brindarán a nuestras aplicaciones la posibilidad de trabajar con registros sin necesidad de un motor de base de datos.

El tema de los passwords suele ser un dolor de cabeza cuando son demasiados. Muchos optan por tener un username y un password para todo, pero no es un método muy efectivo. Por esta razón, se me ocurrió tomar todos los papeletos sueltos con usernames y passwords que tenía por ahí, y hacer un programa donde poder almacenarlos, que sea chico para tenerlo cargado en memoria y utilizarlo cuando lo necesite. De paso, introducirme en el tema de los archivos binarios, que es lo que nos toca para esta edición de la revista.

Qué es un archivo binario

Es un archivo creado especialmente para almacenar registros. Subdivide su longitud en espacios iguales del tamaño que ocupa un registro común, y a medida que agregamos registros, añade un espacio. Se accede a él de manera similar que a uno de texto y de forma consecutiva (primero lee el registro 1, luego el 2, y así sucesivamente) mediante el procedimiento **read ()**.

La gran ventaja que ofrecen es que, si nuestra aplicación requiere el almacenamiento de datos y no se trata de muchos registros, nos convendrá usar un archivo binario más que una

base de datos, ya que, entre otras cosas, evitaremos tener que iniciar el motor de la base y podremos disponer de más flexibilidad en el manejo de nuestros registros (aunque esto signifique programar más).

“Archivos binarios” es un nombre que se usa en la jerga informática, pero cada lenguaje los llama de maneras diferentes. En el caso de Delphi, se dividen en dos: los archivos tipo **Typed** y los de tipo **Untyped**. La diferencia entre ellos es que los primeros tienen definido un bloque de lectura o escritura, o, lo que es lo mismo, leen por registro ya que están divididos de esta forma; mientras que los archivos del tipo Untyped necesitan que se les defina la longitud del bloque a leer o escribir. Hay ciertos procedimientos que son permitidos exclusivamente para cierta clase de archivo. La **Tabla 1** nos muestra los procedimientos admitidos para los archivos **Typed**.

Definiendo el archivo

Antes que nada, debemos saber bien cómo tienen que ser los registros que utilizaremos en nuestro programa y luego definirlos encima de las variables de la siguiente forma:

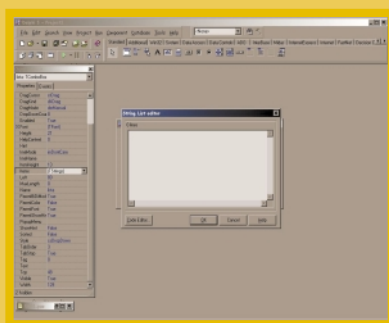


Figura 1. El editor de lista para el ComboBox es amigable y muy fácil de usar; es una verdadera lástima que sólo podamos acceder a él en tiempo de ejecución.

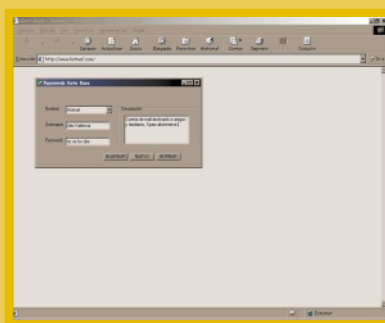


Figura 2. Así es el programa terminado, listo para tomarse el trabajo de memorizar users y passwords. De una vez por todas podremos decirles adiós a los papeletos.

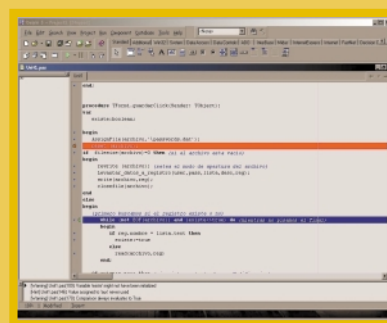
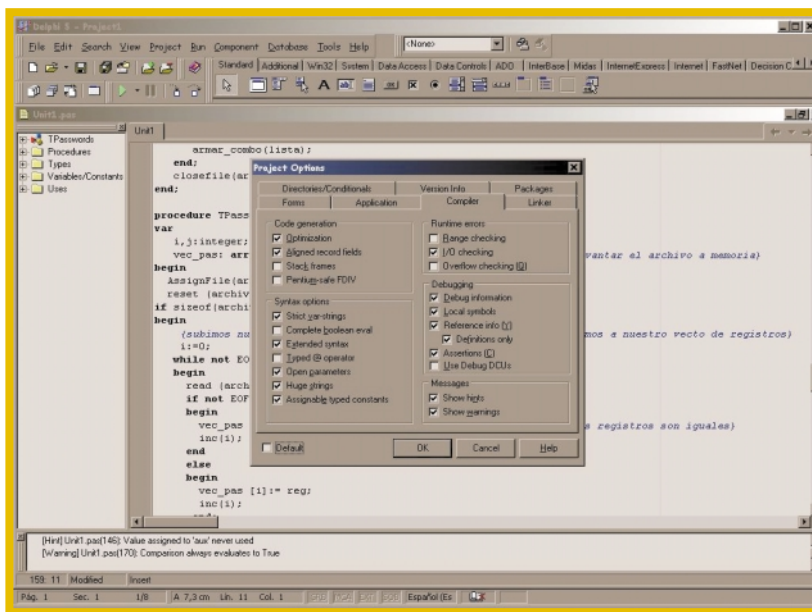


Figura 3. Recordemos usar el debug en nuestros programas para verificar, entre otras cosas, la lógica del código y los errores de ejecución que pudieran llegar a producirse.



Acá podemos setear la forma de compilar nuestro programa. Muchas veces es posible solucionar problemas retocando algunas cosas de aquí.

Type {Palabra clave que indica que vamos a definir nuestro propio tipo de dato}

```
Pasreg = record
    Nombre: string[50];
    user, pass: string[30];
    descripcion: string [255];
end;
```

Éste es el registro que vamos a usar en nuestro programa, así que ahora definimos el archivo y, de forma infaltable, el registro que usaremos para escribirlo y leerlo.

```
reg:pasreg;
archivo: file of pasreg; {Si no definimos el tipo de archivo, lo toma como Untyped}
```

¡Programando la base!

Como ya habrán observado, éste es un típico programa de ABM (Altas, Bajas y Modificaciones), y tenemos muchas maneras de controlarlo. Vamos a basarnos en que la modificación en el campo clave es tomada como un nuevo registro.

El campo clave será aquél por el cual buscaremos, crearemos e identificaremos cada registro. Nuestro registro está conformado por un nombre, un username, un password y una descripción, donde nuestro campo clave será el nombre del registro.

En nuestro form tendremos un ComboBox, tres EditText y tres botones para guardar, crear y borrar un registro.

La A de alta

La alta de un registro o, lo que es lo mismo, la creación de un registro estará dada por el hecho de comprobar la existencia de éste en el archivo. Si no existe, lo damos de alta. El procedimiento, que está asociado al botón **[Guardar]**, será así:

```
begin
    AssignFile(archivo, '\passwords.dat');
```

```
    reset (archivo);
    if filesize(archivo)=0 then {si el archivo está vacío}
        begin
            rewrite (archivo); {setea el modo de apertura del archivo en escritura desde el principio del mismo}
            Levantar_datos_a_registro(user,pass,lista,desc,reg); {Procedimiento que pasa los datos de los objetos al registro}
            write(archivo,reg);
            closefile(archivo);
        end
    else
        begin
            {primero buscamos si el registro existe o no}
            while (not EOF(archivo)) and (existe<>true)
            do {mientras no pisemos el final}
                begin
                    if reg.nombre = lista.text then
                        existe:=true
                    else
                        read(archivo,reg)
                end;

            if existe= true then {si existe, se trata de una Modificación}
                begin
                    reg.user:=user.text;
                    reg.pass:=pass.text;
                    reg.descripcion:=desc.text;
                    write(archivo,reg);
                    closefile (archivo);
```

```
end
else
```

```
begin
    while not eof (archivo) do {Si no,
de una Alta}
        read(archivo);
        Levantar_datos_a_registro(user,pass-
,lista,desc,reg);
        write(archivo,reg); {Escribe en el
final}
        armar_combo(lista); {Procedimiento
que rearma el ComboBox con el campo Nombre}
    end;
end;
end;
```

Este mismo procedimiento nos indica si se trata de una modificación o de un alta, por lo cual también tenemos la parte de las modificaciones resuelta en el mismo procedimiento.

Dentro del procedimiento **armar_combo()**, donde se pasa por parámetro el ComboBox llamado Lista, podemos ver que ésta tiene una propiedad denominada Items, heredada del objeto **Tstring** (es una lista de cadenas), por lo cual deberemos acceder también a la propiedad **Add**, para poder agregar una cadena a la lista del **ComboBox**. La **Figura 1** nos muestra el amigable editor que Delphi nos ofrece para armar la lista en tiempo de diseño.

Una de las cosas que podemos observar es que el trato para el primer registro es diferente que para los demás, ya que habitualmente debemos buscar el final del archivo (registro con la marca EOF, *End Of File*) y agregar el registro nuevo. En cambio, para el primer registro usamos **Rewrite**, que es una de las funciones dadas por el set de I/O (*Input/Output*) de archivos, cuyo fin es abrir un archivo en forma de escritura, borrar todos los registros existentes, si hubiera alguno, y posicionar el puntero al principio del archivo o en BOF (*Begin Of File*). Además, en caso de no encontrar el archivo externo asociado en el procedimiento AssignFile, lo crea automáticamente.

La B de baja

Este procedimiento nos mostrará la forma más elegante y correcta de trabajar con archivos binarios.

La baja estará asociada al evento **Clic** del botón **[Borrar]**, y el procedimiento es así:

```
var
    i,j:integer;
    vec_pas: array [0..limite] of pasreg; {Vector de
registro para levantar el archivo a memoria}
begin
    AssignFile(archivo,'\passwords.dat');
    reset (archivo);
    if sizeof(archivo) <> 0 then
    begin
        {subimos nuestro archivo a memoria o, lo que es
lo mismo, lo pasamos a nuestro vector de registros}
        i:=0;
        while not EOF (archivo) do
        begin
            read (archivo,reg);
            if not EOF (archivo) then
            begin
                vec_pas [i]:= reg; {esta asignación sólo es
posible cuando la estructura de los registros es
igual}
                inc(i);
            end
            else
            begin
                vec_pas [i]:= reg;
                inc(i);
            end;
        end;
        {ahora trabajamos en nuestra estructura ya en
memoria}
```

| Tabla 1 | |
|---|--|
| Éstos son algunos procedimientos y funciones útiles al momento de trabajar con archivos binarios: | |
| Procedimiento | Descripción |
| AssignFile | Asocia una variable tipo file con un archivo externo. |
| ChDir | Cambia el directorio actual. |
| CloseFile | Cierra y disocia la variable con el archivo externo, actualizando este último. |
| Eof | Indica si el registro actual tiene marca de End Of File. |
| Erase | Borra el archivo externo asociado a la variable. |
| FilePos | Devuelve el número de registro actual. |
| FileSize | Devuelve el tamaño del archivo. |
| MkDir | Crea un subdirectorío. |
| Read | Lee un registro del archivo y lo pone en un registro que le pasemos por parámetro. |
| Rename | Renombra el archivo externo asociado. |
| Reset | Ubica el puntero del archivo en el comienzo de éste y lo abre para lectura. |
| Rewrite | Crea y abre un nuevo archivo. |
| Rmdir | Borra un subdirectorío. |
| Truncate | Trunca el archivo a partir del registro actual. |
| Write | Escribe el registro que se le pase por parámetro en el archivo, en la posición donde se encuentre. |

```
i:=0;
while lista.Text <> vec_pas[i].Nombre do
  inc(i); {INC es una función que incrementa
la variable que pasamos por parámetro}
```

{si es igual, es que encontré el registro a borrar, por lo cual corro todo el resto un espacio más atrás}

```
while vec_pas[i].nombre <> '' do
begin
  j:=i+1;
  vec_pas[i]:=vec_pas[j];
  inc(i);
end;
{Por último, pasamos nuevamente los registros
al archivo}
rewrite (archivo);
i:=0;
while vec_pas[i].nombre <> '' do
begin
  write(archivo,vec_pas[i]);
  inc(i);
end;
closefile(archivo);
armar_combo(lista);
clear_fields(user,pass,lista,desc);
end
else
begin
  lista.items.Clear;
  clear_fields(user,pass,lista,desc);
end;
end;
```

Habitualmente, la forma de trabajar con archivos es, primero, levantarlos a memoria a través de un vector de registros (en el caso de los archivos binarios), luego trabajar con ellos en memoria, o sea, en nuestro vector de registros, para finalmente volver a pasarlos al archivo. En este caso, lo levantamos a memoria, buscamos el registro a eliminar y escribimos todo otra vez en el archivo. Como el acceso es secuencial, una vez encontrado el registro (buscado por el campo clave Nombre), corremos todos los que lo preceden un espacio hacia atrás; de esta forma nos quedará todo listo para escribir el archivo en el disco y guardar.

Otro aspecto importante es que, si bien en el archivo es posible agregar nuevos registros ya que está manejado por un puntero, el vector de registro que utilizaremos para levantarlo a memoria no nos ofrece esta posibilidad. En este caso, creamos una estructura de 100 posiciones, lo que significa que, mientras manejemos hasta 100 registros, no tendremos problemas para borrar. Esto se debe a que la estructura nos limita; lo óptimo en este caso sería manejarlo con una lista que relacione registros del tipo **pasreg**, y de esta forma podríamos ir creando en memoria tanta cantidad de espacios como nece-

sitemos directamente en tiempo de ejecución. Como es algo que aún no hemos visto, nos las arreglaremos definiendo una constante llamada **limite**, que tiene asignado el valor 100; así que, si se necesita más espacio (cosa que dudo mucho), se cambia la constante y se recompila el programa. La función **INC**, utilizada en el procedimiento, incrementa la variable en una unidad. Es mejor utilizar esto que el clásico **i:= i+1**, ya que está hecho con instrucciones más parecidas a las del compilador, lo que se traduce en mayor velocidad.

Otra posibilidad interesante que Delphi nos ofrece es la de evadir la terminación de un programa debido a un error de I/O; podemos lograrlo si desmarcamos esta característica en las opciones del proyecto, en la solapa **[Compiler]**, donde además tenemos otras opciones para personalizar la compilación de nuestro proyecto.

Actualizando la pantalla

Esta acción está relacionada con el evento **Clíc** del Combo-Box, que contiene el nombre de nuestro registro. De esta manera, cada vez que seleccionemos un registro por su nombre, inmediatamente se actualizarán los demás campos. Con una base de datos esto se hace de manera casi automática, pero con un archivo binario es diferente, ya que deberemos programarlo. Las líneas correspondientes a dicho procedimiento se presentan a continuación.

```
begin
  AssignFile(archivo, '\passwords.dat');
  reset(archivo);
  read (archivo,reg);
  while lista.text <> reg.nombre do
    read (archivo,reg);
  Levantar_datos_a_pantalla(user,pass,lista,desc,reg); {pasa los datos del registro al form}
  closefile (archivo);
end;
```

Como ven, es muy importante usar bien los ciclos de repetición para recorrer el archivo. En este caso, el EOF no nos preocupa, ya que seguramente el registro que buscamos existe en el archivo.

Por último, el botón **[Nuevo]** se limitará a limpiar los campos del formulario con el procedimiento **Clear_fields()**. La **Figura 2** nos muestra el programa terminado.

A perfeccionarlo

Algo útil para poder ver bien la forma de manejo de estos archivos es correr nuestro programa con el debug, como lo muestra la **Figura 3**, para asegurarnos de que estamos abriendo y cerrando el archivo correctamente.

Usar este tipo de archivos nos ayudará muchísimo a comprender el funcionamiento de las bases de datos, ya que, sin importar de qué base se trate, su funcionamiento es, por demás, parecido a éste. Delphi nos permite manejar de forma muy flexible esta herramienta, lo que le adjudica más puntos a favor a este lenguaje y otra razón para preferirlo al momento de crear nuestras aplicaciones. ❌

APRENDIENDO A APUNTAR CON DELPHI

Punteros en Delphi



Alex VALENCIA
avalencia@tectimes.com

Estudiante de Ciencias de la

Computación, fanático del anime y la programación.

No se trata de un arma ni nada por el estilo; los **PUNTEROS** son muy útiles para cualquier aplicación. Si bien al principio es difícil entenderlos, son muy flexibles y conforman el conjunto **FUNDAMENTAL** de elementos para las bases de cualquier lenguaje de programación. ¡Apuntemos a ellos!

Estuvimos viendo muchos elementos que son imprescindibles al momento de hacer una aplicación, siempre bajo el lente de Delphi (mostrando su sintaxis y modo de uso). Para cerrar esta lista nos faltan los punteros, que son importantísimos, ya que, una vez que sepamos usarlos, más lo aprendido en números anteriores de USERS, estaremos preparados para hacer aplicaciones casi de cualquier tipo.

El concepto de punteros es un poco complicado de entender la primera vez, ya que estamos manipulando directamente la memoria de nuestra máquina, pero no se preocupen, porque lo explicaremos detenidamente; y con los ejemplos dados, ¡serán expertos en punteros!

Conociendo nuestra memoria

Ya vimos nuestro rígido como campo de trabajo donde almacenar y leer archivos. Ahora nuestro campo de operaciones es la memoria de la máquina, por lo que nos conviene conocerla un poco antes de explicar cómo trabaja un puntero.

La RAM (*Random Access Memory*, o memoria de acceso alea-

torio) es la memoria de rápido acceso de nuestra máquina. Existen muchos tipos, como ya vimos, pero la RAM es la más veloz porque el micro accede a ella en forma directa.

No toda nuestra memoria RAM está disponible a nuestro antojo, sino que sólo contamos con una sección de ella. Hay lugares reservados, por ejemplo, donde se almacena nuestro sistema operativo, el cual necesita acceso a ciertos programas y funciones para su correcto funcionamiento, como drivers o programas de administración. Si tenemos Windows o alguno con interfase gráfica de trabajo (ya casi nadie trabaja sobre sistemas operativos no gráficos), se suman a la lista los programas que la interfase sube a memoria; en el caso particular de Windows, los que se encuentran en la Task Bar (barra de Tareas). Hay otros lugares de memoria reservados como buffers, que también residen allí. Por eso, vemos que no toda nuestra memoria está completamente disponible para su libre uso; una vez cargado todo, lo que queda de ella es para nosotros, y aquí es donde vamos a posicionar nuestra nueva herramienta de trabajo. Cuando almacenamos algo

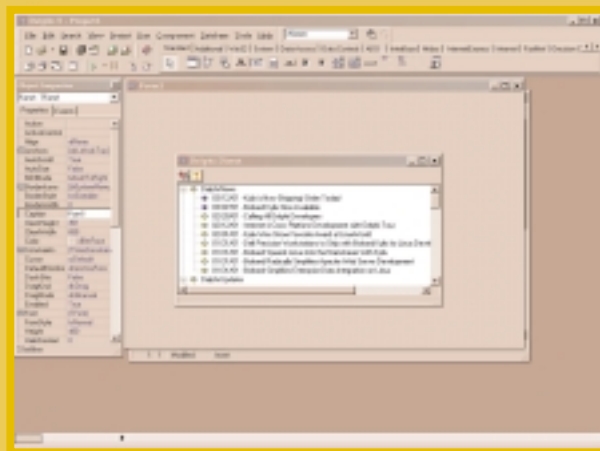


Figura 1: Para los programadores más ociosos, el servicio Delphi Direct ofrece las últimas noticias del mundo Delphi.

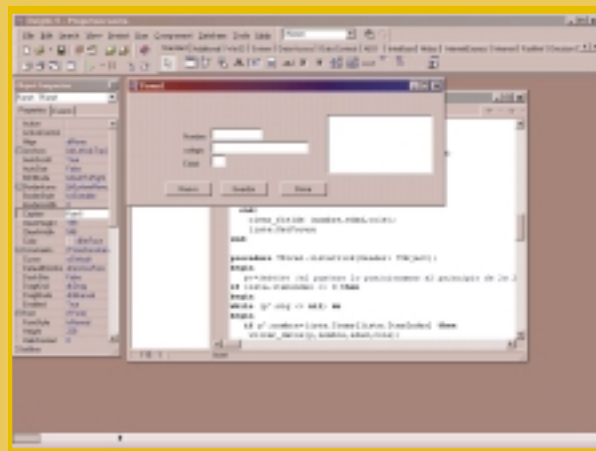


Figura 2: "Encuestas" nos servirá muchísimo para entender los punteros; el ejecutable y el fuente están en el CD.



Ésta es la página principal del site www.clubdelphi.com, donde podremos encontrar muchos trucos y técnicas de programación en español.

en la memoria, la máquina lo hace de manera aleatoria; esto significa que toma el dato que se almacenará y genera una dirección permitida donde guardarlo, acelerando el proceso de lectura y escritura.

El objetivo de los punteros es usar la memoria RAM como nuestro soporte de almacenamiento de datos, lo cual nos brinda muchísimas ventajas que iremos viendo durante esta nota.

Apuntando la atención a los punteros

La pregunta ahora es: ¿qué es un puntero? Es una variable que contiene la dirección de memoria de un dato específico. Por ejemplo, podemos declarar un puntero que almacene la dirección de memoria de algún número que hayamos guardado previamente en memoria; cuando solicitemos el valor de ese puntero, veremos una dirección, pero si pedimos ver a qué apunta éste, nos devolverá el número.

Como sucede con las variables, hay distintas clases de punteros. Por ejemplo, si queremos hacer lo mencionado anteriormente, deberemos usar un puntero a **integer** (si el número es un entero). Hay punteros para todas las clases de variables, así que debemos asegurarnos de usar el correcto.

Tal vez se hayan dado cuenta de que el puntero por sí so-

lo no tiene mucho sentido y que para apuntar a algo en memoria es más efectivo usar una variable; esto es cierto, pero se vuelve útil cuando tenemos más de una cosa para subir a memoria, como, por ejemplo, los registros de un archivo binario.

El uso real de los punteros aparece cuando los aplicamos a los registros; para entenderlo mejor, veamos la siguiente declaración de un tipo de dato.

```

type
  pun=^reg; {puntero a un registro}

  reg=record {registro}
    edad:string[2];
    cole:string [50];
    nombre: string[50];
    sig:pun;
  end;
    
```

Podemos dividir esta declaración en dos partes. Empecemos por la segunda: estamos declarando un registro con un cam-

Tabla 1

Éstas son las estructuras más conocidas para armar con punteros:

| Tipo de estructura | Descripción |
|--------------------|---|
| Lista (FIFO) | Es la más común de todas, ya que no requiere un código muy complejo para armarlo, y es del tipo FIFO: <i>First In, First Out</i> . |
| Pila (LIFO) | Es parecida a la cola, pero invirtiendo el orden de lectura; resulta útil cuando lo que leemos es siempre lo más reciente, por ejemplo, un registro de errores. Es del tipo LIFO (<i>Last In, First Out</i>). |
| Anillo | Es muy útil cuando leemos algunos datos cíclicamente; el final es igual que el principio, lo que forma, como su nombre indica, un anillo. |
| Árbol (Figura 2) | Es, casi por excelencia, la estructura más difícil de armar, pero es la más útil de todas, ya que muchas veces la búsqueda de registros es más rápida que en las demás, aunque frecuentemente nos trae dolores de cabeza. |

po **edad**, uno **cole** y un nombre del tipo **string**, y, por último, un campo **sig** del tipo **pun**. La primera parte de la declaración nos indica que **pun** es un puntero a **reg**, o sea, al registro declarado abajo. La traducción de esto es un registro común con un campo puntero del tipo **reg**, es decir que dicho campo puede almacenar una dirección de memoria donde se encuentra otro registro igual. ¡Eureka!

Este procedimiento es realmente útil, ya que podemos enganchar los registros entre sí de la forma que más nos convenga (según el programa que hagamos). Hay muchas formas de engancharlos, y deberemos utilizar la más óptima, dependiendo del uso que les vayamos a dar a los registros en el programa. La **Tabla 1** nos muestra algunas maneras de armar nuestros punteros.

Empleando los punteros

Si tuviéramos que realizar una aplicación que levantara registros de un archivo para hacerles, por ejemplo, una modificación a todos (habitualmente llamado proceso Batch), tendríamos de dos opciones. Una de ellas es crear un vector de registros lo más grande posible para almacenar los registros que levantemos, y la otra opción, mucho más eficaz, es subirlos a una lista de punteros, ya que es más rápido, utilizamos menor cantidad de recursos de memoria y es más correcto desde el punto de vista de la programación.

A continuación, veremos el ejemplo encuesta, que, si bien no es muy útil para uso personal, nos ayudará a comprender mejor la sintaxis de los punteros.

La **Tabla 2** muestra los elementos que componen nuestra aplicación, y la **Figura 2**, el modo en que los pusimos en el form. La encuesta es de número indefinido de registros, o sea que no sabemos la cantidad de encuestados, por lo que el uso de punteros resulta ideal ya que acudiremos a uno de sus mejores recursos: la creación de espacios en memoria en tiempo de ejecución.

Obviamente, la aplicación "encuesta" no es más que un programa ABM, por lo cual iremos viendo cómo usar los punteros en cada caso.

Comenzando con la encuesta

El registro que utilizaremos lo declaramos al principio de la nota; tiene los campos **nombre**, **edad** y **cole** (colegio), y suponemos que es una encuesta a estudiantes.

El código de la Alta está relacionado con el botón **[Guardar]**, por lo que quedará algo así:

```
begin
  if lista.Items.Count= 0 then {si no hay registros...}
  begin
    new (p); {creamos el primer nodo de la lista}
    p^.sig=nil;
    levantar_datos (p,nombre,edad,cole); {levantamos
los datos del form al nodo o registro}
    lista.Items.add(nombre.text);
    inicio:=p; {el puntero 'inicio' apunta al primer
```

```
nodo, que será el principio de la lista}
    ultimo:=p; {como es el único, también es el último}
  end
  else
  begin
    new (p); {creamos un nuevo nodo en la lista}
    p^.sig=nil;
    levantar_datos (p,nombre,edad,cole);
    ultimo^.sig:=p;
    ultimo:=p;
    lista.Items.add(nombre.text);
  end;
  clear_fields (nombre,edad,cole);
end;
```

Una lista no es más que muchos registros enganchados con un principio y un fin, y es una estructura del tipo FIFO (*First In First Out*: primero en entrar, primero en salir), lo que podríamos imaginar como un tren con cada vagón representando a un nodo (registro).

Hay ciertos punteros que debemos declarar antes de trabajar con cualquier tipo de estructura, que deben ser del tipo del registro con el que estemos operando. Necesitamos estos punteros para direccionar nuestra lista (en este caso), o bien para marcar ciertos nodos importantes en la estructura; en el caso de la lista, el principio y el final.

Fíjense en el código la utilización de la función **new()**, que crea un espacio en memoria en tiempo de ejecución –de forma aleatoria, claro está–, y apunta al puntero pasado por parámetro a ese espacio. En nuestro caso, le pasamos el puntero "p", que es del tipo **pun** (puntero al registro), por lo cual creará un espacio del mismo tamaño que nuestro registro.

Después de esa línea, mostramos cómo direccionar a un campo con un puntero; la línea **p^.sig = nil** significa que estamos asignando el valor **nil** al campo **sig** del registro apuntado por "p". **Nil** no es más que "Nulo"; al asignar **nil** a un puntero, le estamos indicando que no apunte a nada, a ninguna dirección en memoria.

También es importante ver que el trato al primer registro a crear no es igual que el de los demás, debido a que el primer nodo será también el comienzo y el final de la lista, y por eso le asignamos el puntero **inicio** y **final** al mismo nodo. Para los demás, el trato es igual, salvo que iremos corriendo el puntero **final** para que siempre apunte al último nodo creado. Es importante ver que antes de posicionar el puntero en el último nodo creado (llamado **último**), debemos apuntar el **sig** del nodo anterior al nuevo puntero, para que de esta forma se vayan enganchando los nodos a la lista.

También vamos agregando los nombres de los encuestados a la **ListBox** para tener a la vista los que vamos creando.

Liberando memoria


Cuando deseamos eliminar un nodo de nuestra lista, en caso de que queramos borrar un registro, lo que deberemos ha-

LISTADO 1:

```

type
  pun:=^pasreg_puntero;
  Pasreg_puntero = record
    Nombre: string[50];
    user, pass: string[30];
    descripcion: string [255];
    sig: pun;
  end;
var
  i,j:integer;
  p, p_final, p_inicio, P_anterior, p_siguien-
te:pun;
begin
  if lista.text <> '' then
  begin
    AssignFile(archivo, '\passwords.dat');
    reset (archivo);
    if sizeof(archivo) <> 0 then
    begin
      {subimos nuestro archivo, pero en una lista}
      read (archivo,reg);
      new(p_inicio);
      p_inicio^.Nombre:=reg.Nombre;
      p_inicio^.user:=reg.user;
      P_inicio^.pass:=reg.pass;
      p_inicio^.descripcion:=reg.descripcion;
      p_inicio^.sig:=nil;
      p:=p_inicio;
      p_final:=p_inicio;
      p_anterior:=p_inicio;
      while not EOF(archivo)do
      begin
        read(archivo,reg);
        new (p);
        p^.Nombre:=reg.Nombre;
        p^.user:=reg.user;
        P^.pass:=reg.pass;
        p^.descripcion:=reg.descripcion;
        p_anterior.sig:=p;
        p.sig:=nil;
        p_final:=p;
        p_anterior:=p;
      end;
      {ahora trabajamos en la lista ya en memoria}
      {nos fijamos si hay más de un nodo en la lista}
      if p_inicio^.sig<>nil then
      begin
        p:=p_inicio;
        i:=0;
        while lista.Text <> p^.Nombre do
          begin
            inc(i); {INC es una función que in-
crementa la variable que le pasa por parámetro}

```



```

        p:=p^.sig;
      end;
      {si el nodo a borrar es el lro. o el último}
      p_anterior:=p_inicio;
      if p <> p_inicio then {si el puntero a
borrar no es el primero...}
      begin
        for j:=1 to (i-1) do
          p_anterior:=p_anterior^.sig; {posi-
cionamos un puntero en la posición anterior}
        end
      else {si el puntero a borrar es el lro.}
      begin
        if p_inicio.sig <> nil then {si hay
un nodo después del inicio...}
          p_inicio:=p_inicio^.sig; {el ini-
cio será ahora el próximo nodo}
        end;
        {veamos el caso en el que el nodo a borrar
sea el último}
        p_siguiete:=p_inicio;
        if p <> p_final then {si no es el úl-
timo}
        begin
          for j:=1 to (i+1) do
            p_siguiete:=p_siguiete^.sig; {po-
sicionamos un puntero en la posición siguiente}
          end
        else {si el puntero es el último...}
        begin
          p_final:=p_anterior; {el nuevo final}
        end;
        {si el nodo está en medio de la lista, só-
lo bastará conectar el anterior con el siguiente}
        if (p<>p_inicio) and (p<>p_final) then
          p_anterior^.sig:=p_siguiete;
        freemem(p);
      end
      else
      begin
        freemem(p_inicio);
        p:=nil;
      end;
      {pasamos los registros al archivo}
      rewrite (archivo);
      if p<>nil then
      begin
        p:=p_inicio;
        while p<>p_final do
          begin
            reg.nombre:=p^.Nombre;
            reg.user:=p^.user;
            reg.pass:=p^.pass;

```

Continúa en la página 84

Viene de la página 83

```

reg.descripcion :=p^.descripcion;
write(archivo,reg);
p:=p^.sig;
end;
reg.nombre:=p^.Nombre;
reg.user:=p^.user;
reg.pass:=p^.pass;
reg.descripcion :=p^.descripcion;
write(archivo,reg);
closefile(archivo);
armar_combo(lista);
clear_fields(user,pass,lista,desc);
end
else
begin
closefile(archivo);
armar_combo(lista);
lista.items.Clear;
clear_fields(user,pass,lista,desc);
end;
end;
end;
end;

```

cer es liberar el espacio que ocupaba en memoria. Obviamente, antes deberemos buscar el registro a borrar, que estará indicado por la ListBox. Veamos cómo sería.

```

if lista.itemindex <> -1 then {si no hay nodos en la lista...}
begin
if lista.itemindex<> 0 then
begin
p:=inicio;
anterior:=inicio;
for i:=1 to lista.ItemIndex do {nos posicionamos en el nodo a borrar}
p:=p^.sig;
for i:=1 to (lista.ItemIndex - 1) do {colocamos otro puntero en una posición menos para enganchar la lista}
anterior:=anterior^.sig;
if p^.sig <> nil then {si el nodo que borramos no es el último...}
begin
posterior:=P;
posterior:=posterior^.sig; {puntero es un nodo adelante de p}
anterior^.sig:=posterior; {conectamos el anterior con el posterior}
freemem(p); {liberamos el espacio en memoria del nodo borrado}
end
else {si es el último nodo...}

```

```

begin
anterior^.sig:= nil; {el anteúltimo nodo será el último}
freemem(p);
end;
{end;}
end
else
begin
p:=inicio;
inicio:=inicio^.sig;
freemem(p);
end;
lista.Items.Delete(lista.ItemIndex); {sacamos el registro de la lista}
clear_fields (nombre,edad,cole);
end;
end;

```

Freemem() es la función que nos permitirá liberar el espacio de memoria asignado al puntero, pero, obviamente, ése no es nuestro único trabajo cuando queremos borrar un registro. Declaramos dos punteros nuevos (sólo existentes en este procedimiento), que usaremos para posicionarlos antes y después del registro a borrar. Debemos analizar cuatro casos por separado. Si el nodo que vamos a borrar es el único en la lista, sólo liberamos el puntero; si el nodo es el último, debemos asignar como último (nuevo) al nodo anterior a éste, marcado con el puntero **anterior**; si el nodo a borrar es el primero, deberemos asignar como comienzo de lista al nodo indicado por el puntero **siguiente**; y, por último, si el nodo está en el medio de la lista, deberemos asignar a **anterior^.sig** la dirección de memoria correspondiente al puntero **siguiente**. De esta manera, la lista quedará unida. Es importante acordarse de liberar el nodo después de unir la lista correctamente, ya que, si no, los errores serán inevitables.

Otro punto que debemos tener en cuenta es la actualización de los datos al clickear en un registro que queramos ver en la lista. El código asociado a este procedimiento sería:

```

begin
p:=inicio; {el puntero lo posicionamos al principio de la lista}
if lista.itemindex <> 0 then
begin
while (p^.sig <> nil) do
begin
if p^.nombre=lista.Items[lista.ItemIndex] then
volcar_datos(p,nombre,edad,cole);
p:=p^.sig;
if (p^.sig= nil) and ( p^.nombre=lista.Items[lista.ItemIndex])then
volcar_datos(p,nombre,edad,cole);
end;
end
else

```

```
volcar_datos(p,nombre,edad,cole);
end;
```

Actualizando el Password DB

En el número anterior de USERS creamos nosotros mismos el programa Password DB, que podíamos usar como base de datos de nuestros usernames y passwords, para lo que usamos archivos binarios (explicado en el mismo número). Como el ejemplo de la encuesta es introductorio al tema de los punteros, aplicaremos lo aprendido al Password DB, para ver también cómo trabajaremos cuando tengamos que levantar datos de un archivo a una lista.

Vamos a centrar el problema en el procedimiento **borrar** del programa, que lo que hacía anteriormente era levantar todos los registros al archivo a memoria para eliminar el seleccionado, y luego volver a grabar en el archivo el vector modificado.

En el CD encontrarán lo mismo pero con una lista dinámica (hecha con punteros) con el resto de las fuentes de esta nota.

La diferencia entre este procedimiento **borrar** y el de la encuesta es que estamos levantando los registros desde un archivo binario. Al principio del procedimiento declaramos un registro igual del que encontramos en el archivo, pero con la diferencia de que contiene el campo puntero **sig**, que nos permitirá formar la lista con la cual trabajaremos (véase la declaración del tipo dentro del procedimiento), que sólo será válida dentro del procedimiento.

No sabemos la cantidad de registros que hay dentro del archivo; por eso los punteros son las estructuras más útiles para estos casos, ya que usan solamente la memoria necesaria y son mucho más rápidos en el momento de la ejecución.

Este ejemplo se podría hacer con menos punteros, pero para poder entenderlo y para nuestros primeros programas con esta estructura, lo mejor es que usemos los que necesitamos, ya que un puntero es una variable que sólo guarda una dirección de memoria y casi no ocupa espacio. Una vez que estamos cancheros, podremos probar los programas con menos punteros, y nos quedará un código más profesional y legible. Por ejemplo, podríamos hacer desaparecer el puntero que indica el final de la lista, ya que este nodo siempre se reconoce porque su **sig** apunta a **nil**.

Otro punto importante a observar es la utilización de declaraciones de tipos dentro del procedimiento; haciéndolo, podemos ver el puntero declarado directamente allí. Esto nos impide escribir otro procedimiento para eliminar código repetido que utilice este tipo de dato, ya que sólo es válido dentro del procedimiento. Por otro lado, es recomendable usar este tipo de declaraciones, que hacen que los procedimientos sean más independientes del programa; esto significa que el código es transferible a cualquier otra

aplicación y no tendremos que definir nada nuevo para utilizarlo.

Algunos otras operaciones con punteros

Hasta ahora hemos creado el puntero y luego el espacio en memoria. En caso de que quisiéramos marcar un puntero a un dato ya creado, podríamos hacer lo siguiente:

```
Var
  i, j: integer;
  p:^integer;
begin
  i:=5;
  p:=@i;
  j:=p^;
end;
```

El prefijo **@** usado en una variable nos devuelve la dirección de memoria en que ésta se guarda. En el caso anterior, no hacemos más que copiar el 5, almacenado en la variable **i**, a la variable **j** (obviamente, del mismo tipo), pasando por un puntero.

Los punteros también tienen operaciones asociadas; ya vimos **>**, **<**, **=**, **<=** y **^**. Pero también podemos usarlos con **+** y **-** con un número entero. Como resultado, nos devolverá una nueva dirección con lo que hayamos sumado o sustraído del entero. Esto es lo que hacemos, por ejemplo, cuando recorremos un vector. Podemos ir sumando la longitud del tipo de dato del array al puntero para lograr lo mismo. Éstas son todas las operaciones que podemos realizar con ellos.

Punteros forever

El de los punteros es un tema sumamente interesante y con muchísimas posibilidades; de hecho, sin darnos cuenta los estamos utilizando desde hace tiempo: cuando direccionamos un vector, no hacemos otra cosa que decirle a un puntero **[i]** dónde posicionarse, o cuando asignamos el nombre de una variable a un archivo, no es más que un puntero al archivo.

Generalmente nos dicen que antes de escribir un código, debemos razonarlo primero en lápiz y papel. Lamentablemente, pocos lo hacen, y casi siempre nos ponemos a escribir un código sin pensarlo previamente, y que salga lo que Dios quiera. Con los punteros nos vamos a acostumbrar a hacerlo, ya que no hay otra manera de entenderlos que dibujando lo que queremos hacer. Generalmente, es muy difícil entender punteros de buenas a primeras, pero al dibujarlos se facilita mucho la comprensión.

La gran mayoría de los programas que trabajan con registros es del tipo ABM. Con lo que hemos aprendido hasta hoy, tenemos todo lo necesario para realizar un buen programa de manipulación de datos, algo bastante importante. Queremos resaltar que si captaron los conceptos entregados desde la USERS #117 (donde empezamos a hablar de Delphi), están capacitados para realizar proyectos de gran nivel, y si a esto le sumamos la integridad y flexibilidad del lenguaje Delphi, se darán cuenta de que lograron mucho. Los animamos a que sigan adelante con Delphi. ❌

— Tabla 2: Objetos principales

| Tipo de objeto | Nombre |
|----------------|-------------------------|
| TButtons (x3) | Nombre, Guardar, Borrar |
| TEdit (x3) | Nombre, Cole, Edad |
| TListBox | Lista |