

PROGRAMACIÓN EN DELPHI



HERNAN PEÑARANDA V.

materias-hpv.comli.com

Sucre, 2010

1. FUNDAMENTOS DE LA PROGRAMACIÓN ESTRUCTURADA

1.1 Introducción

El software es un producto intangible que sin embargo tiene un gran valor económico. Sus ingresos superan incluso los de otros productos tangibles y tradicionales como aquellos relacionados con la industria del petróleo. El software ha dado lugar a empresas multimillonarias como Microsoft, Sun Microsystems, Borland, ULead, etc., cuyos ingresos superan los de las grandes petroleras e industrias mundiales como la Coca Cola.

Su éxito ha hecho que actualmente se encuentre en todas las actividades productivas y no productivas del ser humano, aún la comunicación inalámbrica e Internet no serían posibles si no contarán con el software adecuado.

Al ser un producto comercial, es importante que los procesos empleados en su desarrollo y producción estén por lo menos al nivel de los procesos industriales tradicionales, sin embargo, ello es algo que aún no se ha conseguido en la práctica, en parte porque la producción de software no es un proceso meramente mecánico (es fundamentalmente un producto de la creatividad e ingenio humano), pero principalmente porque, en su creación, no todos siguen los mismos principios y procedimientos, lo que dificulta y en la práctica impide su normalización y consiguiente estandarización.

Desde la invención de las computadoras han surgido varias metodologías que han tratado de uniformar criterios con el fin de estandarizar su producción, sin embargo, no se ha tenido éxito en este propósito, existiendo actualmente varias metodologías, cada una con sus punto a favor y en contra.

Mientras no se uniformen los criterios y se estandarice su producción, no queda otra alternativa que estudiar las metodologías que más éxito han tenido en la práctica, para tratar de producir un software de calidad más o menos estándar.

En este curso estudiaremos y emplearemos una de dichas metodologías: "la metodología de programación estructurada". Al concluir el mismo deberán estar capacitados para "aplicar los principios de la programación estructurada en la creación, mantenimiento y actualización de software, con el fin de producir, eficientemente, software de alta calidad".

1.2 Surgimiento de la programación estructurada

En los primeros años de la computación y debido a los limitados recursos instalados en las computadoras, se elaboraban programas relativamente pequeños para resolver problemas específicos en los campos científico y militar^[2,3]. Generalmente estos programas eran elaborados por una sola persona y era el mismo programador el encargado de manejar y mantener el programa.

Bajo estas condiciones prácticamente cada programador desarrollaba su propio estilo y método^[4] de programación, el cual por supuesto sólo era entendido por él mismo. Por lo tanto los programas sólo podían ser corregidos o modificados por el creador del programa. Adicionalmente estos programas eran elaborados en lenguaje de máquina o ensamblador y en este lenguaje cada tipo de computadora tiene su propio conjunto de instrucciones, lo que dificultaba el mantenimiento y corrección de los programas.

A medida que las computadoras contaban con más recursos y se hacían más accesibles, se crearon lenguajes de programación más fáciles de entender. Es así que surgieron lenguajes como FORTRAN y BASIC^[1]. Estos lenguajes, denominados de "alto nivel", tienen una estructura más cercana a la del lenguaje

hablado, siendo en consecuencia más fáciles de aprender y emplear. Esta característica facilitó el desarrollo de programas más complejos, los que debido a su tamaño, comenzaron a ser elaborados por grupos de programadores.

Pronto se hizo evidente que no era suficiente contar con un lenguaje de programación fácil de comprender, pues cada programador seguía teniendo un estilo diferente y programando de acuerdo a su propio criterio. Los problemas comenzaron a surgir al momento de corregir y actualizar los programas, pues para entonces varios de los autores de los programas originales ya no pertenecían al grupo de trabajo, por lo que el mismo debía ser llevado a cabo por otro programador, el cual se veía ante la difícil tarea de descifrar primero la lógica empleada en el programa, para tratar luego de actualizarlo o corregirlo. Con frecuencia, el tiempo empleado en esta labor consumía más tiempo que el consumido en el desarrollo del programa original, con lo que el mantenimiento y actualización de programas se convirtió en una actividad muy costosa.

Puesto que en la práctica todos los programas de software requieren, tarde o temprano, un mantenimiento y actualización, se hizo evidente la necesidad de contar con una metodología de programación que fuera fácilmente entendida por todos los programadores y que proporcionara un conjunto de principios, sólidos y simples, para la elaboración y mantenimiento del software.

Es así que a finales de los años 60, surgió la **programación estructurada (PE)**, en base a las prácticas y métodos que ya empleaban algunos programas y que habían comprobado su eficacia en la elaboración y mantenimiento de programas. La programación estructurada formó parte de las llamadas *técnicas para el mejoramiento de la productividad en programación*^[6].

La programación estructurada fue la metodología de programación que dominó el desarrollo de aplicaciones en el ámbito de la informática durante los años 70 y 80, y aunque actualmente no ocupa ese lugar privilegiado, los fundamentos en los que se basa siguen siendo tan útiles y válidos como cuando fue creada.

1.3 Programación no estructurada

En la programación no estructurada prácticamente no existen lineamientos para la construcción de programas, el único requisito práctico es que el programa funcione eficientemente, es decir lleve a cabo la tarea para la cual ha sido creado consumiendo la menor cantidad posible de recursos, entendiéndose que los recursos más valiosos en informática son el tiempo de ejecución y la memoria. Al no existir lineamientos claros, surge una multitud de estilos y métodos de programación, los cuales con frecuencia tienen lógicas tan complejas que al cabo de un tiempo, inclusive el mismo autor del programa no comprende inmediatamente qué es lo que hizo.

Gran parte de la complejidad asociada a los primeros programas se debía al excesivo uso de instrucciones *GOTO* (saltos incondicionales), lo que en cierta manera era alentado por los lenguajes de alto nivel más empleado en ese entonces: FORTRAN, BASIC y C. La instrucción *GOTO* permite pasar el control del programa a cualquier parte del mismo, con lo que es posible hacer saltos hacia delante o hacia atrás, dando así libertad y flexibilidad al programador^[5], pero introduciendo gran complejidad en la lógica, dificultando en consecuencia la depuración, mantenimiento y actualización de los programas.

La lógica de los programas elaborados con este enfoque, aún en pequeños programas, solía verse como el diagrama de actividades de la figura 1.1.

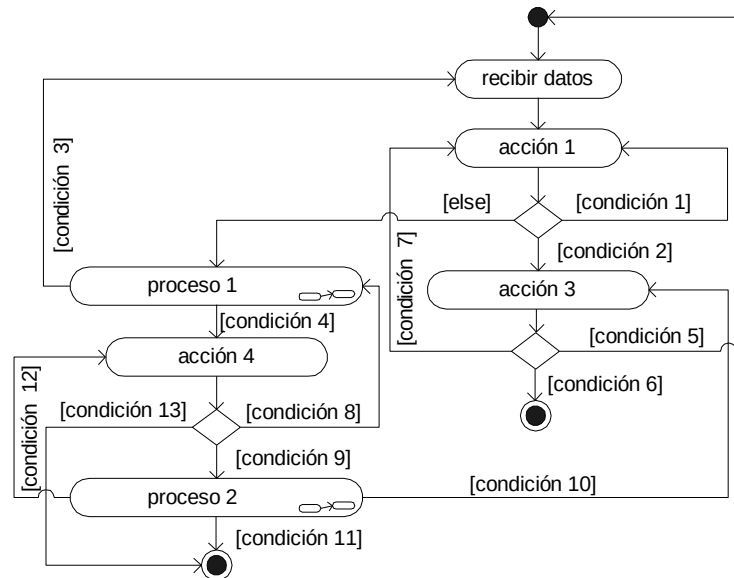


Figura 1.1. Lógica común en un programa no estructurado.

Como se puede observar, en un programa no estructurado, el control del programa puede ir hacia delante, hacia atrás, saltar de un proceso a otro, y terminar en dos o más lugares. Es esta flexibilidad la que da lugar a una lógica compleja, que por lo enredado y los múltiples cruces de línea que tiene se conoce como programación *tipo Espagueti*.

Otro inconveniente para el mantenimiento de los programas, asociado con algunos lenguajes de alto nivel de la primera generación (principalmente BASIC), es que realmente no permitían la creación de subprogramas independientes (módulos), de manera que el programa debía ser resuelto como un todo, incrementando así la complejidad de la lógica. Adicionalmente, estos lenguajes, sólo contaban con un conjunto muy limitado de tipos de datos, por lo que la lógica debía ser modificada para adaptarse a los mismos.

1.4 Fundamentos De La Programación Estructurada

Como ya se mencionó previamente, la programación estructurada está constituida por los principios, técnicas y métodos que demostraron su eficacia en la creación, mantenimiento y actualización del software. Dichos principios son los siguientes^[12]:

- Dividir un problema complejo en problemas más sencillos (programación descendente).*
- Emplear estructuras estándar para construir la totalidad del programa.*
- Emplear tipos de datos a la medida.*

La aplicación correcta de estos tres principios simplifica considerablemente la elaboración, depuración, prueba y sobre todo el mantenimiento y actualización de los programas.

1.4.1 Programación descendente (Modularidad)

El dividir un problema en problemas más sencillos y estos a su vez en otros, hasta que cada problema represente una tarea lo suficientemente sencilla como para poder ser resuelta en forma independiente, se conoce como *programación descendente* o *programación modular* ^[12,13,15].

Se conoce como programación descendente (TOP-DOWN) porque se comienza analizando el problema como un todo y luego se va *descendiendo* a problemas cada vez menos generales hasta llegar a problemas específicos que pueden ser resueltos independientemente.

Esquemáticamente el enfoque de la programación descendente puede ser representado como se muestra en el diagrama de componentes de la figura 1.3.

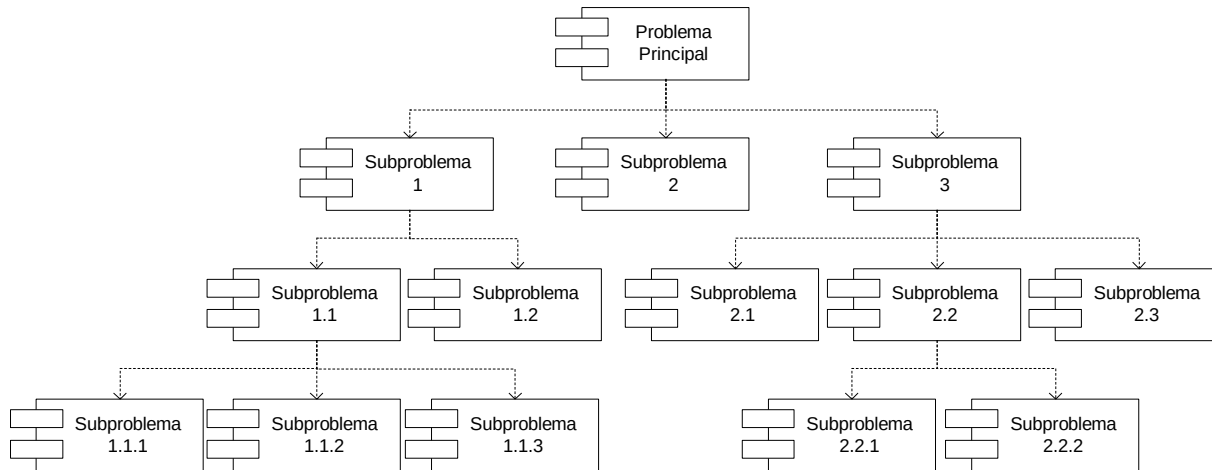


Figura 1.3. Modularidad. División de un problema en problemas específicos

Por supuesto, la programación descendente puede ser vista también como *programación ascendente* (DOWN-TOP), pues se resuelven primero los problemas más sencillos y se va *ascendiendo* hacia los más complejos. Normalmente sucede que el análisis del problema es *descendente*, mientras que su solución es *ascendente*.

La *programación descendente* es también conocida como *programación modular*: cada uno de los problemas en los cuales se subdivide el problema principal se conoce como *módulo*.

La característica principal de un *módulo* es que puede ser analizado, codificado y probado independientemente. Por supuesto si un módulo hace uso de otros módulos, dichos módulos deberán haber sido elaborados previamente.

Siempre y cuando el módulo efectúe la tarea para la cual ha sido creado, su lógica interna puede ser modificada tantas veces como se requiera sin afectar la funcionalidad del programa (o de otros módulos). Esta característica facilita considerablemente la elaboración y mantenimiento de los programas, posibilitando además la creación de grandes proyectos mediante grupos de programadores, donde cada programador está a cargo del desarrollo de uno o más módulos.

Adicionalmente los módulos permiten la reutilización eficiente del código: Las funciones realizadas por algunos módulos pueden ser útiles en dos o más módulos o en otros programas, entonces, cuando al resolver un problema se requiere de dichas funciones, simplemente se llama al módulo que las realiza, evitando así la duplicación de tareas y la repetición de código.

La utilización de módulos no sólo permite la reutilización del código, sino que además facilita la corrección y mantenimiento de los programas. Cuando se realiza un programa modular, los errores se detectan en módulos específicos, de manera que sólo es necesario analizar y corregir dichos módulos y no todo el programa. Igualmente cuando se quiere mejorar la eficiencia de un programa, se detectan los módulos que más tiempo y memoria consumen y sólo se optimizan dichos módulos.

Con el tiempo se detectan módulos que son de utilidad general y se procede a agruparlos en archivos comunes dando lugar así a las *librerías*. Dichas librerías hacen más eficiente aún la reutilización del código y reducen considerablemente el tiempo empleado en la elaboración de programas.

1.4.2 Estructuras estándar

En 1964 dos matemáticos: Corrado Böhm y Guiseppe Jacopini publicaron un artículo, donde dieron a conocer el *teorema de la programación estructurada*: "todo programa puede ser escrito empleando sólo tres estructuras: secuencia, selección e iteración"^[6,12,13,15] y en 1966 demostraron la validez de tal afirmación.

Desde entonces este teorema se convirtió en el pilar fundamental de la programación estructurada. El objetivo principal de este teorema es el evitar el uso del comando GOTO, que como vimos anteriormente, promueve el desarrollo de algoritmos con una lógica difícil de entender, depurar y mantener. El no GOTO fue apoyado por muchos otros académicos del área de la informática, particularmente por Edsger Dijkstra (uno de los padres de la programación estructurada), quien en 1968 mandó a la revista *Communications of de ACM* (Comunicaciones de la Asociación de Cálculo Automático de USA) una carta que se publicó bajo el título "Go-to statement considered harmful"^[14], en la cual explica el por qué, en su criterio, la instrucción GOTO es perjudicial para la creación de software, y no sólo sugiere el tratar de evitarlo, sino que propone inclusive su completa abolición.

Aunque el artículo generó controversia, fue de gran influencia en el desarrollo posterior de la programación estructurada y de los lenguajes de programación estructurados.

La principal característica de una estructura, en la programación estructurada, es la de contar con un solo punto de entrada y uno de salida y es esta característica la que respetaremos en el presente curso. Esquemáticamente una estructura estándar puede ser representada como se muestra en la figura 1.2.

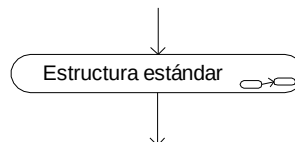


Figura 1.2 Representación esquemática de una estructura estándar

Gracias a esta característica los programas estructurados se construyen enlazando estructuras estándar una a continuación de otra, de manera que la lógica del programa puede ser seguida con facilidad leyendo el programa de arriba hacia abajo sin perder continuidad en dicha lectura.

Internamente cada estructura tiene sus propias características y puede contener incluso otra u otras estructuras, pero siempre contará con una sola entrada y una sola salida. El programa estructurado, visto como un todo, igualmente tendrá una sola entrada (inicio) y una sola salida (fin).

Debemos aclarar no obstante que un programa estructurado no necesariamente tiene una lógica más eficiente que uno no estructurado. Un programa no estructurado puede ser más eficiente que un programa estructurado en el sentido de consumir menos recursos del sistema (memoria y tiempo), pero no debemos olvidar que con la programación estructurada no sólo se busca eficiencia, sino también claridad, de manera que el software pueda ser mantenido y actualizado con relativa facilidad.

Sin embargo, existen algunas ocasiones, no muy frecuentes, donde un programa no estructurado no sólo es más eficiente, sino también más claro que uno estructurado. Cuando ello sucede se puede emplear la versión no estructurada, pues no se está violando realmente el propósito principal de la programación estructurada (la de crear programas claros). Debido a que esta situación puede presentarse, la mayoría de los lenguajes estructurados cuentan con instrucciones no estructuradas (incluido el comando *GOTO*).

Entonces, no se debe olvidar que el objetivo principal de la programación estructurada es la de permitir el planteamiento claro y ordenado del problema, lo que a su vez reduce los errores, facilita la reutilización del código y sobre todo facilita su mantenimiento y actualización.

Varios autores han demostrado la existencia de problemas que pueden ser planteados y resueltos con mayor claridad, facilidad y eficiencia empleando instrucciones no estructuradas. En la práctica, cuando este sea el caso, se pueden emplear instrucciones no estructuradas, teniendo el cuidado de crear bloques que tengan una sola entrada y una sola salida, para de esa manera mantener la claridad y orden propios de los programas estructurados.

1.4.3 Tipos de datos (datos a la medida)

El tercer principio de la *programación estructurada* señala que debemos definir con claridad y precisión tanto los tipos de datos que requiere un módulo (o programa) como los tipos de resultados que devuelve.

El emplear tipos de datos claramente definidos permite reducir los errores que de otra manera se producen cuando: a) se manda información a un módulo (por ejemplo cuando se intenta mandar un texto a un módulo que trabaja con números); b) al procesar la información dentro del módulo (por ejemplo cuando se intenta realizar operaciones aritméticas con cadenas de texto) y c) al utilizar los resultados devueltos por un módulo (por ejemplo cuando se espera un número entero y el resultado devuelto por el módulo es un número real).

De acuerdo a este principio, si no se cuenta con un tipo de dato adecuado para resolver un problema en particular, se debe definir uno nuevo (un tipo de dato a la medida).

En la práctica, la aplicación de este principio implica la revisión cuidadosa de los tipos de datos que se emplean en un módulo (o programa). Por ejemplo: si se está creando un módulo para calcular el factorial de un número, entonces se debe emplear un tipo de dato que sólo permita números enteros positivos (pues no existe el factorial de números negativos); igualmente, si se está calculando la raíz cúbica de un número real y se requiere un resultado con una precisión de 18 dígitos, el resultado deberá ser de tipo real y permitir al menos 18 dígitos; si se ha creado una función para determinar si un número es o no primo, el resultado deberá ser de tipo lógico, porque sólo existen dos posibilidades: que el número sea primo (True) o que no lo sea (False).

1.5 La crisis del software

A medida que los programas se hacían más y más grandes y se disponía de menos tiempo para su conclusión, se hizo evidente que el proceso de análisis, diseño, prueba e implementación del software consumían más tiempo del que realmente se disponía, dando lugar así a la denominada *crisis del software*^[4].

Esta crisis se caracteriza fundamentalmente porque la demanda de software es mayor a la producción. La consecuencia principal de esta crisis es la

baja calidad del software, debido al escaso número de pruebas a las que es sometido.

Para afrontar esta crisis era necesario que la producción del software se normalice y estandarice de manera que se pueda garantizar su producción y calidad. Es con ese fin que surge la metodología de programación estructurada (y posteriormente otras metodologías más). No obstante, tanto la metodología de programación estructurada, como las subsiguientes, no consiguieron realmente normalizar y estandarizar el proceso de producción de software, debido principalmente al egoísmo y recelo de las empresas de software, quienes para no pagar derechos de autor y percibir así la totalidad de los ingresos originados por la venta del producto, desarrollan sus propios módulos y librerías y, como consecuencia, los mismos módulos se elaboran, de manera irracional, una y otra vez, dando lugar no sólo a la duplicación de software, sino también a un considerable retraso en el desarrollo del mismo, originando así la "*crisis del software*".

Lejos de llevar a cabo un análisis autocrítico, la comunidad informática buscó la causa de esta crisis en aspectos formales y técnicos: como las metodologías de programación, y algunos académicos concluyeron, erróneamente, que esa era la causa de la crisis del software.

Es así que surgieron otras metodologías, siendo una de las de mayor aceptación la "*metodología de programación orientada a objetos (POO)*", la cual en realidad surgió a principios de los años 60, antes incluso que la programación estructurada, pero que recién se popularizó a mediados de los años 80, donde fue promocionada como la metodología que daría solución a la crisis del software^[7].

1.6 Programación estructurada y programación orientada a objetos

Si bien la PE y la POO tienen elementos en común, abordan los problemas desde puntos de vista muy diferentes: la PE se centra en las funcionalidades del sistema mientras que la POO se centra en los "objetos" con los que debe contar el sistema.

De acuerdo a los defensores de la POO, esta metodología representa una manera más natural de abordar los problemas, pues en el mundo real estamos acostumbrados a trabajar con "objetos", sin embargo los detractores de la POO arguyen que esta "naturalidad" es engañosa pues en la POO es necesario con frecuencia trabajar con "objetos abstractos", los cuales por supuesto no existen en el mundo real^[8,9].

Otra de las supuestas ventajas de la POO es la de posibilitar una reutilización más eficiente del código. En contraposición sin embargo, estudios estadísticos llevados a cabo en el año 1998^[10] han demostrado que la reutilización real del software con la POO es de sólo un 15%, porcentaje menor aún que los logrados en décadas pasadas con la programación tradicional y menor aún a los logrados con la PE.

Lo cierto es que más de 20 años después de haber entrado en vigencia, la POO no ha logrado resolver el problema de la *crisis del software*^[11], la crisis es actualmente inclusive mayor a la que existía cuando esta metodología fue introducida. Muchos informáticos de alto nivel cuestionan actualmente la POO y piden que se lleve a cabo un estudio científico e imparcial de la misma para justificar o no su uso.

La POO ha sido la metodología que ha dominado el desarrollo de software en las dos últimas décadas, pero no por razones científicas, sino principalmente por razones comerciales. A mediados de los años 80 y durante la década

de los 90, si un software no traía el título de "orientado a objetos", simplemente no se vendía (pues la POO estaba de moda).

Otra de las razones para la popularidad de la POO es la relación errónea que se ha establecido entre POO y los objetos gráficos como ventanas, iconos, menús, etc. De hecho la mayoría de las personas creen que los objetos gráficos son el resultado de la POO, asocian erróneamente las interfaces gráficas de usuario (GUI por sus siglas en inglés) con la POO, cuando las interfaces gráficas, con casi todos los objetos con los que cuentan actualmente (ventanas, menús, celdas, iconos, etiquetas, etc.), fueron creadas con la metodología de programación estructurada y en consecuencia no son "objetos", sino funciones, implementadas en la interfaz de aplicaciones del sistema operativo (API).

Al respecto debemos recordar que la primera computadora con un sistema operativo gráfico fue lanzada al mercado por *Apple Macintosh* en 1984^[2] y dicha interfaz fue desarrollada en su totalidad siguiendo los principios de la programación estructurada, no de la programación orientada a objetos.

Para comprender la importancia y utilidad actual de la PE, con relación a la POO, debemos echar una mirada a los sistemas operativos actuales. Tanto Windows (en su última versión) como Linux (en todas sus versiones) siguen siendo esencialmente estructurados. Entonces debemos cuestionarnos: si la POO tiene tantas bondades y ventajas, como le atribuyen sus promotores, ¿Por qué los sistemas operativos actuales son estructurados y no orientados a objetos? ¿No es esta una señal clara de que algo anda mal con la POO?

Como se puede deducir de la anterior exposición la PE no es una metodología obsoleta, simplemente no está de moda. Tampoco ha fallado en su propósito de resolver la crisis del software, pues como ya se dijo, esta crisis no ha sido resuelta ni será resuelta mientras predomine el egoísmo y los intereses mezquinos. La PE es una metodología confiable y de gran utilidad como lo demuestran aplicaciones importantes como los sistemas operativos y muchas otras en el campo comercial y científico que han sido y aún son desarrolladas con éxito empleando esta metodología^[5].

Actualmente han surgido otras metodologías de programación, siendo la *programación guiada por agentes* la que más atención recibe en este momento y se cree que sea la metodología que domine el desarrollo de software durante el resto de esta década. En esta metodología el software se construye simplemente seleccionando los agentes adecuados para llevar a cabo las tareas. Un agente es un subsistema inteligente que puede identificar el medio en el que se encuentra: Cuando un agente es colocado en un sistema detecta los agentes que se encuentran en el mismo pidiendo a los otros agentes que se identifiquen, entonces los otros agentes le informan sobre su identidad y funciones y a su vez registran las funciones del nuevo agente. De esa manera un agente puede resolver problemas y realizar tareas, recurriendo a los agentes presentes en el sistema.

La programación guiada por agentes implica no sólo datos y código, sino también inteligencia artificial, por lo que se considera el siguiente paso en la evolución de las metodologías de programación.

1.7 Preguntas

1. ¿Actualmente la producción del software está estandarizada?
2. ¿Cuál es el objetivo del presente curso?
3. En los primeros años de la computación ¿Existía alguna metodología de programación que dominara el desarrollo de software?

4. ¿Por qué resultaba muy difícil mantener y actualizar programas no estructurados?
5. ¿Cómo se denomina a los lenguajes de programación como BASIC y FORTRAN?
6. ¿Cómo se denomina la lógica que da lugar el excesivo uso del comando GOTO?
7. ¿Qué recomienda Edsger Dijkstra con relación al comando GOTO?
8. ¿Cuáles son los fundamentos de la programación estructurada?
9. ¿En qué consiste la programación descendente?
10. ¿Con qué otro nombre se conoce a la programación descendente y por qué se denomina así?
11. ¿Cuál es la principal característica de un módulo?
12. ¿Cuáles son los principales beneficios de emplear módulos?
13. ¿Qué dice el teorema de la programación estructurada?
14. ¿Cuál es la principal característica de un bloque estructurado?
15. ¿Un programa estructurado es siempre más eficiente que uno no estructurado?
16. ¿Por qué es importante emplear en un programa tipos de datos a la medida?
17. ¿En la práctica cómo se aplica el tercer fundamento de la PE?
18. ¿En qué consiste la crisis del software?
19. ¿A qué se atribuyó la crisis del software?
20. ¿Por qué no se ha resuelto aún la crisis del software?
21. ¿Cómo se abordan los problemas en la programación estructurada?
22. ¿Cómo se abordan los problemas en la programación orientada a objetos?
23. La programación orientada a objetos ¿ha resuelto la crisis del software?
24. Los objetos como *ventanas, botones, menús, etc.*, ¿son el resultado de la POO?
25. Los sistemas operativos actuales ¿Han sido programados con la metodología de programación orientada a objetos?
26. ¿Cuál es la metodología que se cree se domine el desarrollo de software en la presente década?

1.8 Referencias bibliográficas

1. Mendez J. Lenguajes de programación. www.monografias.com; 2002
2. Gerald De Freitas C. Marco histórico de las computadoras. www.monografias.com; 2002.
3. Camacho A. Historia de la computación. www.monografias.com; 2002.
4. Pressman R. Ingeniería del Software. Tercera Edición. McGrawHill. 1993.

5. Catambay B. The Pascal Programming Language. <http://www.pascal-central.com>; 2001.
6. Cabrera H. Programación Estructurada. www.monografias.com; 2002.
7. Martin J. Odell J. Análisis y diseño orientado a objetos. México: Prentice Hall Hispanoamericana; 1994.
8. Jacobs B. Object Oriented Programming Oversold. Segunda Edición. www.geocities.com; 2002.
9. Meyer B. Critique of Object Oriented Software Construction. www.geocities.com; 2001.
10. Finch L. So Much OO, So Little Reuse. <http://www.ddj.com>; May 7, 1998.
11. Mullins G. The Great Debate. <http://www.byte.com>; April 1994.
12. Ferrer A. Introducción al Pascal: Programación Estructurada. Chile: Antártica S. A.; 1986.
13. Aguilar LJ. Turbo Basic: Manual de programación. España: McGraw-Hill; 1989.
14. Dijkstra E. Go-to statement considered harmful. Communications of the ACM, 1968 XI, 3:147-148. Disponible en <http://www.cs.utexas.edu/users/EWD>.
15. Courington Rebecca. Structured Programming. <http://acweb.colum.edu/users/rcourington/Progclass/progclass.html>; 1998

2. INTRODUCCIÓN A DELPHI

Si bien la metodología de programación estructurada puede ser enseñada y aplicada en cualquier lenguaje, lo ideal para su aprendizaje es emplear un lenguaje estructurado. Existen muchos lenguajes estructurados en el mercado que podemos elegir para este fin, sin embargo, uno en particular: Pascal, ha sido creado con el propósito específico de enseñar a programar siguiendo los principios de la programación estructurada, por lo que la elección de lenguaje es obvia.

Una vez elegido el lenguaje (Pascal) debemos elegir el compilador a emplear. Actualmente existe una gran variedad de compiladores y ambientes de desarrollo en *Pascal*, algunos de ellos gratuitos y otros comerciales.

Dentro las herramientas gratuitas destacan *Free Pascal* (y su ambiente de desarrollo gráfico *Lázarus* (www.freepascal.org)) y *GNU Pascal* que también cuenta con una serie de ambientes de desarrollo. La principal ventaja de estas herramientas, aparte de ser gratuitas, es que están en constante desarrollo, por lo que se actualizan frecuentemente corrigiendo algunos errores, añadiendo nuevas funcionalidades y proporcionando el código fuente para que se puedan hacer las modificaciones que se vean por conveniente. La principal desventaja, por el contrario, es que al ser gratuitas, no existe garantía del producto y de la continuidad del mismo, así el desarrollo del software puede ser suspendido en cualquier momento y el software elaborado con estas herramientas puede presentar errores, aunque esto último no es muy frecuente y también se da en el software comercial.

Existen igualmente una gran variedad de compiladores y ambientes de desarrollo comerciales para Pascal, pero la que más ha destacado en las últimas décadas es la versión *Pascal* de *Borland*, que inicialmente surgió para el sistema operativo *DOS* con el nombre de *Turbo Pascal* (en sus versiones 1 a 7), posteriormente lanzó la versión para *Windows* con el nombre *Borland Pascal para Windows* y finalmente su ambiente gráfico de desarrollo rápido bajo el nombre de *Delphi*.

Puesto que el compilador de Borland para Pascal es un producto estable y confiable, que se ha mantenido en el mercado por décadas y no se prevé que pueda desaparecer en un futuro inmediato, elegiremos el ambiente de desarrollo de esta firma "Delphi" para elaborar los programas en esta materia. En ese sentido, el propósito del presente capítulo es que al concluir el mismo estén capacitados para elaborar, ejecutar y depurar programas en Pascal bajo el ambiente de desarrollo rápido de Delphi, haciendo uso eficiente de las herramientas disponibles.

2.1 El ambiente de desarrollo de aplicaciones de Delphi

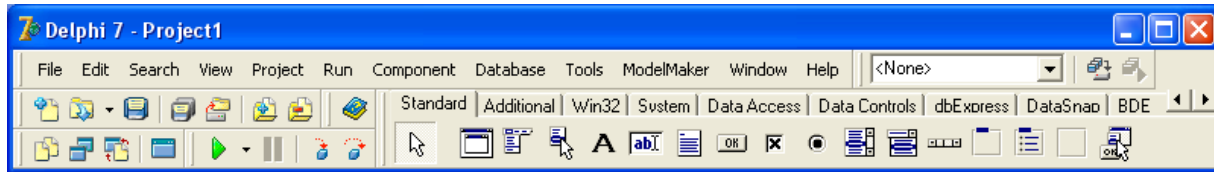
Delphi adquirió gran popularidad debido a la facilidad y rapidez con la que es posible crear aplicaciones con interfaces gráficas.

Desde su aparición han surgido las versiones 1, 2, 3, 4, 5, 6, 7, 8, *Delphi 2005*, hasta *Delphi 2009* y aunque pareciera lógico emplear la última versión disponible, pues las últimas versiones siempre corrigen errores e incorporan nuevas herramientas, en este curso emplearemos la versión 7, por dos razones: a) Las principales mejoras, a partir de la versión 8.0, están relacionadas a la programación para aplicaciones .NET, WEB y Bases de Datos, las mismas que no son el objeto de estudio del presente curso, por lo que dichas mejoras no son de importancia para el propósito del presente curso y b) Porque las últimas versiones son muy exigentes en cuanto al hardware necesario, requiriendo computadoras de última generación, que no siempre están disponibles.

En el año 2006 y para hacer frente a la fuerte competencia que representa el software libre, Borland lanzó versiones gratuitas de sus productos, denominados "Turbo Explorer". Así en el caso de Delphi se tiene "Turbo Delphi Explorer", con el mismo ambiente y la mayoría de las herramientas de las versiones comerciales, aunque por supuesto con algunas limitantes como la de no permitir añadir nuevos componentes a las paletas y la de solo permitir uno de sus productos (Delphi, C++Builder, JBuilder, etc.) por computadora. Lamentablemente, Borland no mantuvo esta política durante los siguientes años. En la actualidad sólo está disponible la versión del año 2006 en <https://downloads.embarcadero.com/free/delphi>.

Antes de comenzar a estudiar el entorno de desarrollo debemos aclarar que el lenguaje Pascal de Delphi es "Object Pascal", se trata entonces de una versión de Pascal orientada a objetos, razón por la cual explicaremos algunos conceptos de la programación orientada a objetos, porque aunque **no** elaboraremos programas orientados a objetos, haremos uso de las herramientas y componentes que nos proporciona Delphi, los cuales están orientados a objetos.

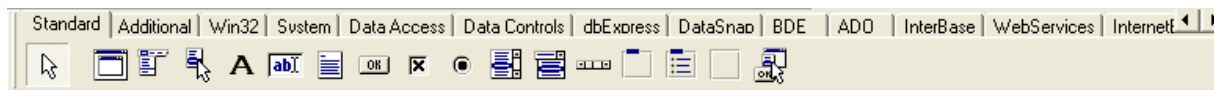
Al ingresar a Delphi 7, podrá observar cuatro ventanas. En la parte superior verá una ventana similar al de la siguiente figura:



Como puede observar, en esta ventana se encuentra la barra de menús de Delphi (*File, Edit, Search, etc.*), donde puede encontrar los comandos y opciones que le permiten trabajar y configurar el entorno de Delphi.

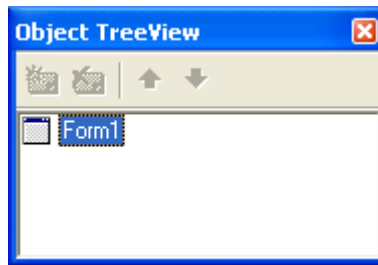
Abajo del menú, en la parte izquierda se encuentran los botones de acceso rápido, donde se encuentran los botones de uso más frecuente, tales como abrir: , guardar , guardar todo , abrir proyecto , ejecutar , ver unidad , ver forma , cambiar entre ver unidad y ver forma , nueva forma , ayuda , ejecutar paso a paso , ejecutar instrucción por instrucción , etc. Los botones de acceso rápido pueden ser personalizados quitando o añadiendo nuevos botones. Para ello simplemente se hace clic, con el botón derecho del mouse, en cualquier parte de los botones, y en el menú emergente se seleccionan o deselectan los grupos de botones, pudiendo también elegir botones individuales a través de la opción "Customize".

En la parte derecha podemos ver la paleta de componentes:



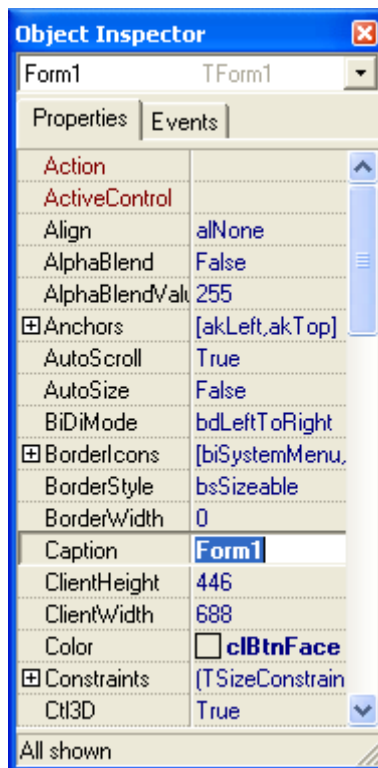
Los componentes, que son los objetos reutilizables de Delphi, están agrupados por su funcionalidad en páginas: "Standard", "Additional", "Win32", etc. Para utilizar uno de estos componentes simplemente se hace clic en el componente y luego clic en la forma, con lo que aparece una instancia (objeto) del componente en la forma. Al igual que los botones de acceso rápido, la paleta de componentes puede ser personalizada, para ello se hace clic con el botón derecho del mouse en cualquier parte de la misma y en el menú emergente se elige la opción "properties", entonces, en la ventana que aparece, se pueden reordenar, añadir o eliminar componentes.

Otra ventana que es visible al ingresar a Delphi es el árbol de objetos:



Esta ventana, que está ubicada en la parte superior derecha de la pantalla, permite ver y explorar todos los objetos que forman parte de la aplicación. Como se puede ver en la figura, inicialmente sólo existe un objeto en una aplicación: la forma (Form1).

Por debajo del árbol de objetos se encuentra el "inspector de objetos":



Antes de explicar las funcionalidades de inspector de objetos es necesario hablar brevemente de las "clases - objetos" y la "programación guiada por eventos".

Los objetos y las clases tienen la misma relación que las variables y los tipos de datos, así como una variable es de algún tipo, un objeto es de alguna clase, por lo tanto las clases son los tipos de objetos y los objetos las variables de ese tipo. En la POO cuando se crea una variable de alguna clase se dice que se ha creado una "instancia" de esa clase.

Ahora bien las clases (y en consecuencia los objetos) son elementos de software que están conformados tanto por datos como por los procedimientos y funciones que controlan dichos datos, realizan acciones y/o devuelven resultados en base a los mismos. En la POO los datos de una clase se denominan propiedades o atributos y los procedimientos y funciones se denominan métodos. Se dice que una clase (y en consecuencia un objeto) "encapsula" sus

propiedades y métodos, porque "oculta" los detalles de su implementación a sus usuarios: para emplear un objeto sólo se necesita conocer las funciones y operaciones que realiza (implementa), pero no como las realiza, igualmente, los datos quedan "ocultos" a los usuarios pues sólo se puede acceder a los mismos a través de los métodos del objeto, de esa manera se evita su corrupción por uso impropio. Al conjunto de métodos que implementa una clase y que pueden ser empleados por los usuarios de sus objetos, se denomina "interfaz" de la clase.

En cuanto a la "programación guiada por eventos", que no es parte de la POO, se trata de un método de programación en el cual el programa reacciona en respuesta a "eventos" generados usualmente (aunque no siempre) por el usuario. Un "evento" es un suceso al cual debe responder un programa o aplicación, como se dijo, los eventos son generados normalmente por el usuario, como por ejemplo cuando pulsa una tecla o una combinación de teclas, hace clic con el mouse, mueve el mouse, arrastra una figura, etc., pero también puede ser generado por el sistema como cuando se produce una división entre cero o el desbordamiento de la pila.

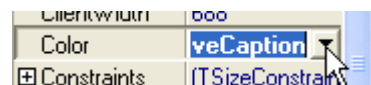
La programación guiada por eventos, es ideal para aplicaciones con interfaces gráficas, donde el usuario interactúa con el sistema a través de eventos. En la programación guiada por eventos, existe un ciclo iterativo que se repite continuamente en busca de eventos a los cuales pueda responder. Difiere de la programación tradicional donde el programa se detiene esperando que el usuario introduzca datos y/o instrucciones y termina cuando devuelve una respuesta o realiza la tarea pedida.

Como ya se dijo, la programación guiada por eventos no es parte de la POO, aunque con frecuencia se la relaciona erróneamente con la misma. Así por ejemplo sistemas operativos como Windows (que no son orientados a objetos) interactúan con sus usuarios a través de eventos, gracias al uso de la programación guiada por eventos, en Windows, los ciclos iterativos que controlan los eventos son procesos, no objetos. Igualmente, otros sistemas operativos como RPL, que no soportan siquiera el concepto de objetos, hacen uso de la programación guiada por eventos.

Ahora que sabemos que son las propiedades y los eventos, volvamos al inspector de objetos. Como se puede observar, tiene dos páginas: *properties* (propiedades) y *events* (eventos). La página *properties* nos permite modificar las propiedades más usuales del objeto seleccionado (para seleccionar un objeto, simplemente se hace clic sobre el mismo). Para modificar el valor de una propiedad simplemente se escribe el nuevo valor en el campo adyacente al nombre del campo, por ejemplo para cambiar el título (Caption) de la forma a "Mi Ventana" se escribe:



Si el campo adyacente, tiene una flecha hacia abajo, como en la propiedad color:



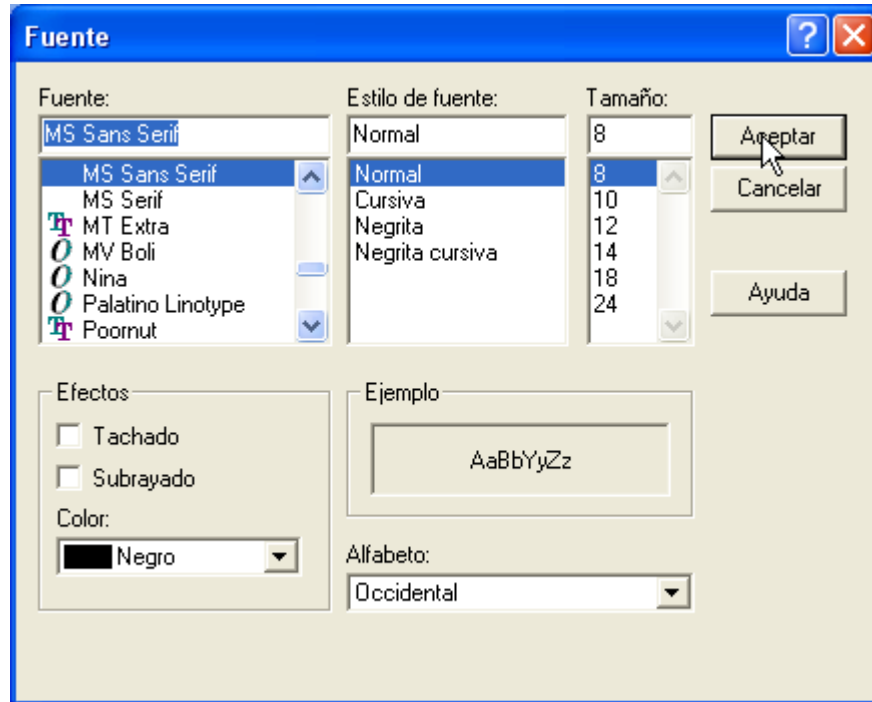
Haciendo clic en la flecha aparece una lista de la cual se puede elegir un valor para la propiedad:



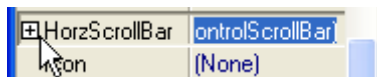
Si existen tres puntos en el campo, como en la propiedad Font:



Haciendo clic en los tres puntos (o doble clic en el campo) aparece una nueva ventana, donde se pueden modificar los valores de la propiedad:



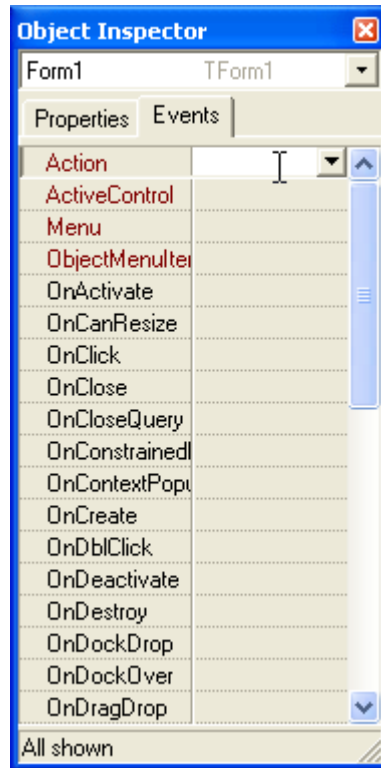
Si la propiedad está precedida por un signo "+" como la propiedad "HorzScrollBar" (y también la propiedad Font):



Ello indica que es una propiedad compuesta por dos o más propiedades (pues en realidad dicha propiedad es a su vez un objeto). Para ver y modificar las propiedades de las que está compuesta se hace clic en el signo "+":



Por otra parte, la página de eventos (que se muestra en la figura de la siguiente página) nos permite programar las acciones que llevará a cabo la aplicación en respuesta a los eventos que puede controlar el objeto. Al igual que cada objeto tiene sus propiedades, tiene también un conjunto de eventos a los cuales puede responder.



Para programar la respuesta a un evento simplemente se hace doble clic en el campo adyacente al mismo, entonces Delphi crea la estructura del procedimiento que dará la respuesta a dicho evento. Así por ejemplo, cuando se hace doble clic en el campo del evento "OnCreate", Delphi crea la estructura del siguiente procedimiento:

```
procedure TForm1.FormCreate(Sender: TObject);
begin

end;
```

En general los nombres de los eventos nos dan una idea más o menos clara del evento al que responden, así el evento "onCreate" responde a la creación del objeto, es decir es el evento que se activa cuando se está creando el objeto. El evento "onClick" es el evento que se activa cuando se hace clic sobre el objeto, "onClose" es el evento que se activa cuando se cierra el objeto, "onDblClick" es el evento que se activa cuando se hace doble clic sobre el objeto, "onPaint" es el evento que se activa cuando se "dibuja" el objeto, "onResize" es el evento que se activa cuando se cambia el tamaño del objeto, "onShow" es el evento que se activa cuando se "muestra" el objeto, "OnDragDrop" es el evento que se activa cuando el objeto está siendo arrastrado y es soltado, etc.

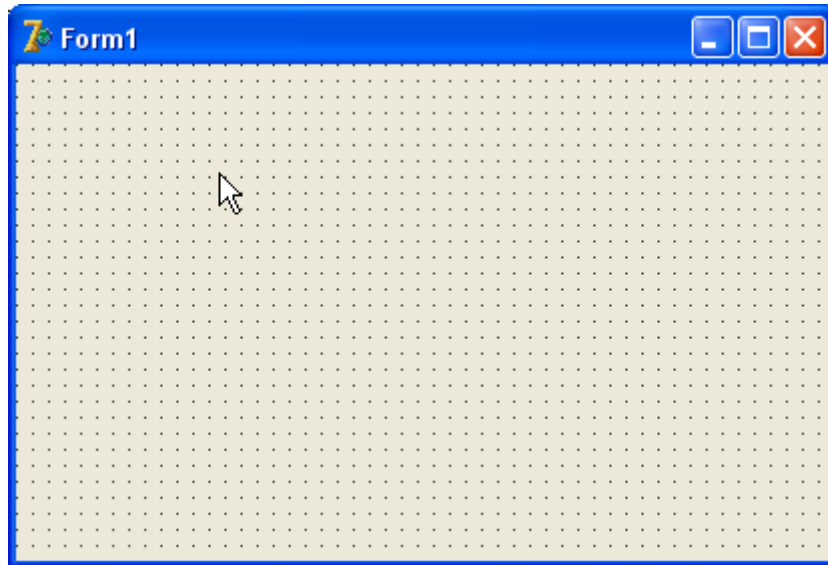
Una vez creada la estructura del procedimiento por Delphi, simplemente se escriben las instrucciones dentro del mismo, por ejemplo la siguiente instrucción hace que aparezca el mensaje "FORMA CREADA" cuando se hace correr la aplicación:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    ShowMessage('FORMA CREADA');
end;
```

En ocasiones puede ocurrir que se cierre inadvertidamente la ventana del inspector de objetos, para que vuelva a aparecer se va al menú "View" y se

elige la opción "Object Inspector" (o alternatively se pulsa la tecla F11). A través de este menú se hacen visibles también otras ventanas, incluida la ventana del árbol de objetos (Object TreeView).

Finalmente, en la parte central de la pantalla se encuentra una ventana, que se conoce con el nombre de "forma" y que por defecto tiene el nombre "Form1":

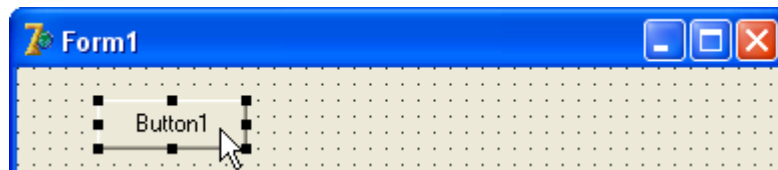


Es en la "forma" donde se construye la interfaz de la aplicación, colocando los componentes necesarios de la paleta de componentes. Los programas que se elaboran en el ambiente de Windows se les da el denominativo de "aplicaciones", y en general están conformados por dos o más archivos: el programa propiamente y por lo menos una librería.

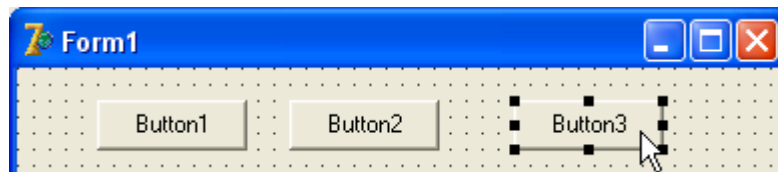
Como ya se dijo, para colocar un componente en la forma, se hace clic en el componente y clic en la forma, así para colocar un botón en la forma se hace clic en el componente "Button":



Y luego clic en la forma:



Observe que por defecto Delphi nombra los objetos con el nombre del componente seguido de un número secuencial, por ejemplo añadiendo dos botones más a la forma les asigna los nombres "Button2" y "Button3" respectivamente:



Sin embargo, el nombre de un objeto es sólo una más de sus propiedades, la propiedad "Name", por lo que puede ser cambiado fácilmente. Así por ejem-

ploteo para cambiar el nombre del "Button1" a "bAceptar", hacemos clic en el objeto:



Luego escribimos el nuevo nombre en la propiedad "Name":



Con lo que el título del objeto cambia automáticamente a "bAceptar":

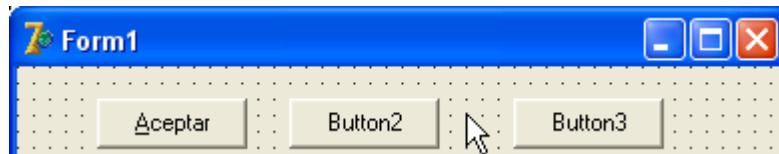


Es una práctica recomendable la de cambiar los nombres por defecto por nombres más significativos, es también recomendable que dichos nombres comiencen con alguna sigla o siglas que identifique el tipo de objeto, así la "b" de "bAceptar" identifica el tipo de objeto "Button" y la palabra "Aceptar" nos indica su funcionalidad. Sin embargo es importante que este cambio de nombres sea hecho al principio, antes de comenzar a escribir código, pues una vez que se ha escrito código, el cambio de nombres tiene que ser hecho también dentro del código, lo que casi siempre da lugar a una serie de errores y confusiones.

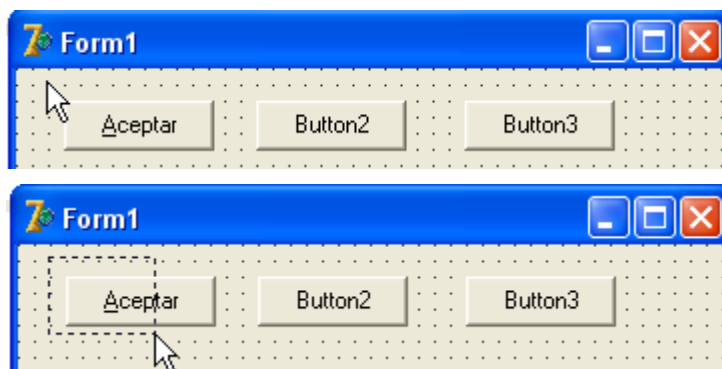
Por otra parte, Delphi asigna por defecto al título el mismo texto del nombre, sin embargo, el título es otra propiedad, la propiedad "Caption", por lo que no es necesario y en realidad no es aconsejable, que el título y el nombre sean iguales. Así por ejemplo, podemos cambiar el título del botón "bAceptar" a "Aceptar" simplemente:

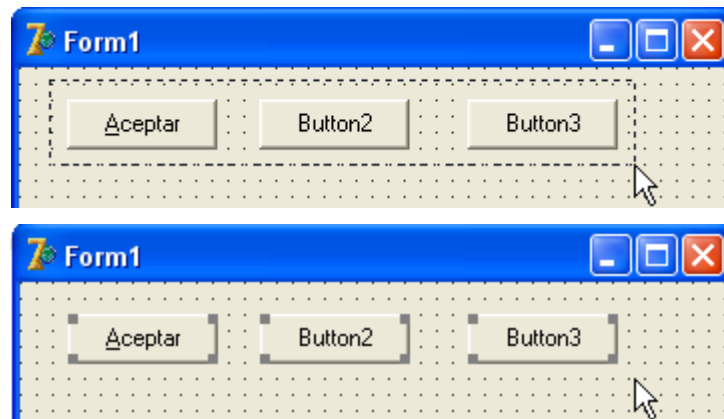


Donde la "&" delante de la letra "A", hace que la misma aparezca subrayada y crea automáticamente el atajo para dicho objeto (recuerde que los atajos son las teclas que se pueden pulsar en combinación con la tecla Alt, en lugar de hacer clic con el mouse). Con las anteriores modificaciones, los botones de la forma quedan como se muestra en la figura:

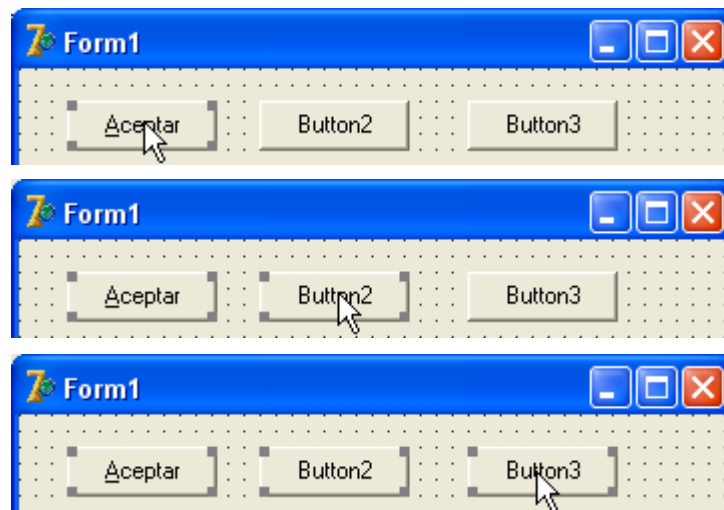


Con frecuencia cuando se elabora la interfaz de una aplicación es necesario seleccionar dos o más componentes a la vez, ello se consigue en Delphi de dos maneras: a) Arrastrando con el mouse hasta que los objetos a seleccionar estén dentro del recuadro:

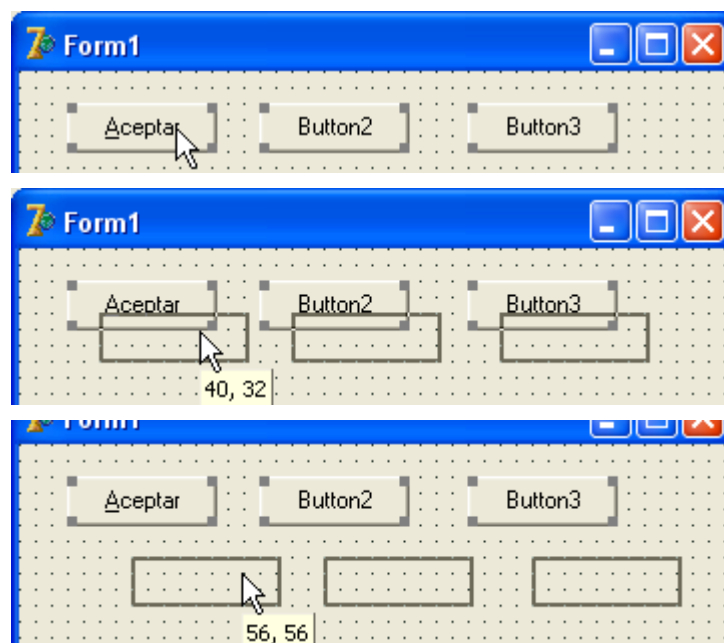


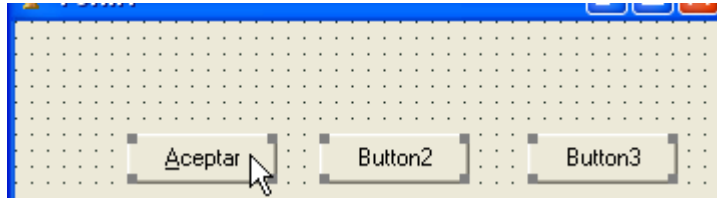


b) Manteniendo presionada la tecla "Shift" mientras se hace clic sobre los objetos que se quiere seleccionar:



Para mover uno o más objetos en una forma, una vez seleccionados, existen dos formas: a) Se arrastran a la posición deseada con el mouse:



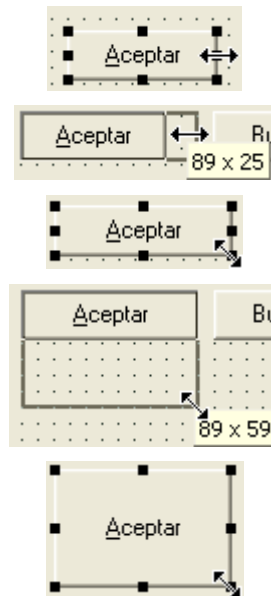


b) Manteniendo pulsada la tecla "Ctrl" y se pulsán las teclas de los cursores. Si se quieren movimientos más amplios se mantienen pulsada las teclas "Ctrl+Shift".

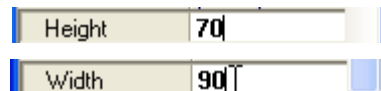
La posición de un objeto puede ser cambiada también modificando el valor de sus propiedades "Left" (izquierda) y "Top" (arriba), que establecen la posición del objeto con relación a los márgenes izquierdo y superior de la forma:



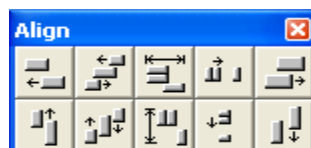
Para cambiar el tamaño de un objeto existen tres formas: a) Una vez seleccionado el objeto se lleva el puntero a uno de los cuadraditos que aparecen en los bordes del objeto, entonces cuando el puntero cambia de forma, se arrastra para cambiar el tamaño del objeto:



b) Se mantiene pulsada la tecla Shift y se utilizan las teclas de los cursores para cambiar el tamaño del objeto. c) Se cambian las propiedades "Height" (alto) y "Width" (ancho) del objeto:

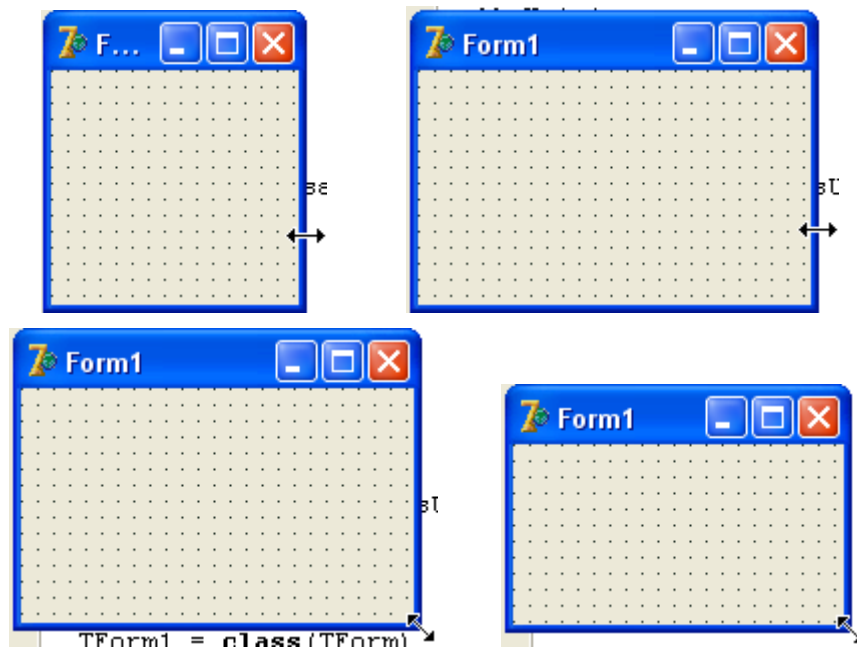


Para alinear los objetos de la forma es recomendable emplear de la paleta de alineación (menú View -> Alignment Palette):



Las figuras de la paleta dan una idea más o menos clara de la alineación que llevan a cabo. El primer botón de la fila superior alinea los objetos seleccionados a la izquierda, el segundo los centra verticalmente, el tercero centra los objetos horizontalmente con relación a la forma, el cuarto separa los objetos verticalmente (con espacios uniformes) y el quinto alinea los objetos a la derecha. El primer botón de la fila inferior alinea los objetos hacia arriba, el segundo los centra horizontalmente, el tercero centra los objetos verticalmente con relación a la forma, el cuarto separa los objetos verticalmente (con espacios uniformes) y el quinto los alinea hacia abajo.

Por supuesto también es posible cambiar el tamaño de la forma siguiendo el procedimiento estándar de Windows, es decir llevando el puntero hacia uno de los bordes o esquinas hasta que cambie de forma y arrastrando luego para cambiar el tamaño:

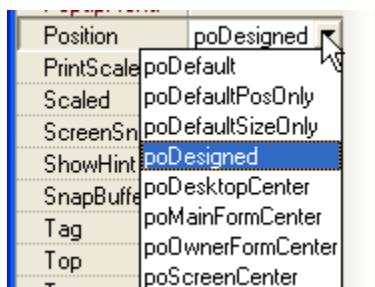


También es posible cambiar el tamaño de la "forma" modificando las propiedades "Height" y "Width", igual que ocurre con otros objetos.

Para mover la "forma", se puede seguir el procedimiento estándar para mover ventanas, es decir arrastrarla haciendo clic en la barra del título:

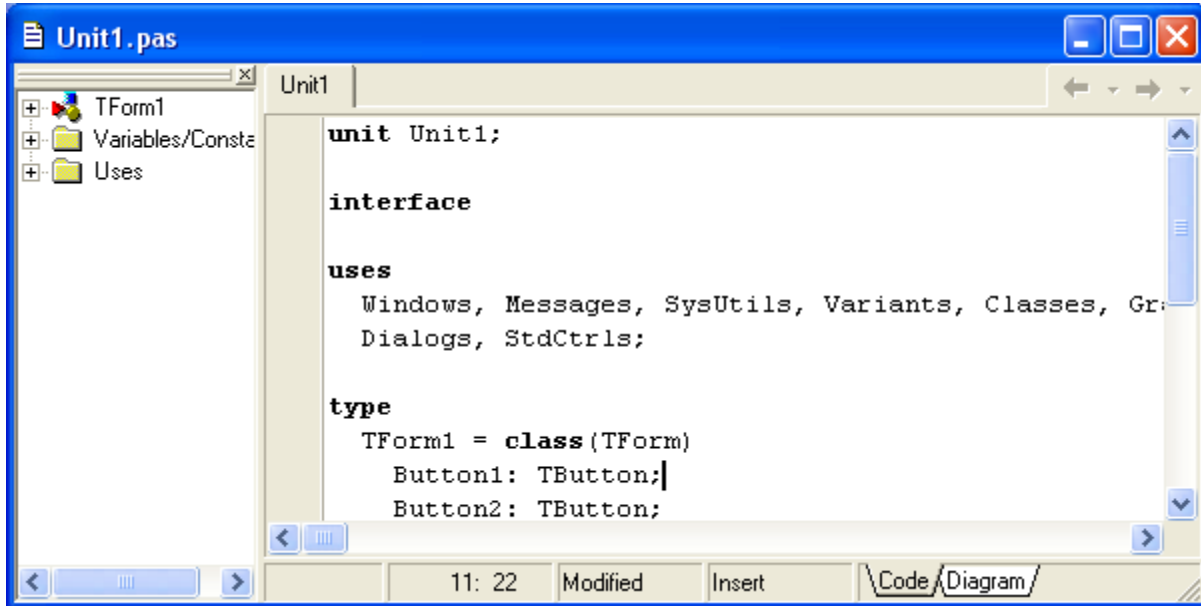


También es posible mover la "forma" cambiando el valor de sus propiedades "Left" y "Top", igual que ocurre con otros objetos. Para establecer la posición de la "forma" al momento de su ejecución se fija el valor de la propiedad "Position":



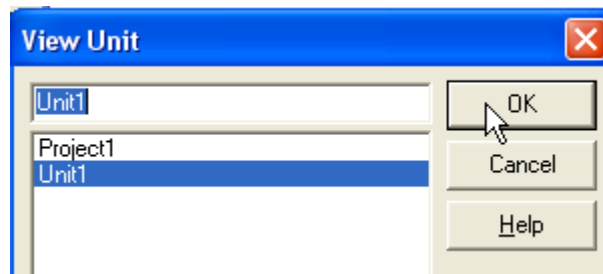
Por defecto esta propiedad tiene el valor "poDesigned", que hace aparecer la "forma" en el mismo lugar donde fue colocada al elaborar la aplicación, pero puede ser conveniente por ejemplo que aparezca centrada con relación al escritorio "poDesktopCenter", centrada con relación a la "forma" principal de la aplicación "poMainFormCenter", centrada con relación a la forma a la que pertenece "poOwnerFormCenter", centrada con relación a la pantalla "poScreenCenter", con el tamaño y posición fijadas por el sistema operativo "poDefault", con la posición fijada por el sistema operativo "poDefaultPosOnly" o con el tamaño fijado por el sistema operativo "poDefaultSizeOnly".

En Delphi el código se escribe y modifica dentro de "unidades", las mismas que se visualizan en la ventana de edición:



Esta ventana usualmente está casi oculta detrás de la "forma", para visualizarla se puede hacer clic en cualquier parte visible de la misma y puede ser necesario mover la forma para hacer una parte de la ventana de edición visible. Generalmente resulta más cómodo pulsar la tecla "F12" (View->Toggle Form/Unit o el botón de acceso rápido: ) , para "alternar" entre la "forma" y la "unidad" (haciendo visible la forma y/o la unidad).

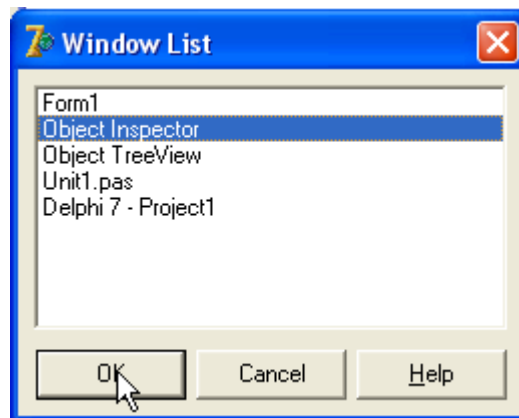
Cuando se tienen dos o más unidades puede resultar más rápido pulsar las teclas "Ctrl+F12" (View->Units... o el botón de acceso rápido: ) , la misma que nos muestra una ventana de la cual se puede seleccionar el programa (Project1) o una de las unidades de la aplicación:



Igualmente cuando existen dos o más formas es más cómodo pulsar las teclas "Shift+F12" (View->Forms... o el botón de acceso rápido: ) , la misma que nos muestra una ventana de la cual se puede seleccionar una de las "formas" de la aplicación:



Finalmente, para seleccionar cualquiera de las ventanas de la aplicación se pueden pulsar las teclas "Alt+0" (Alt+cero o View->Window list...)



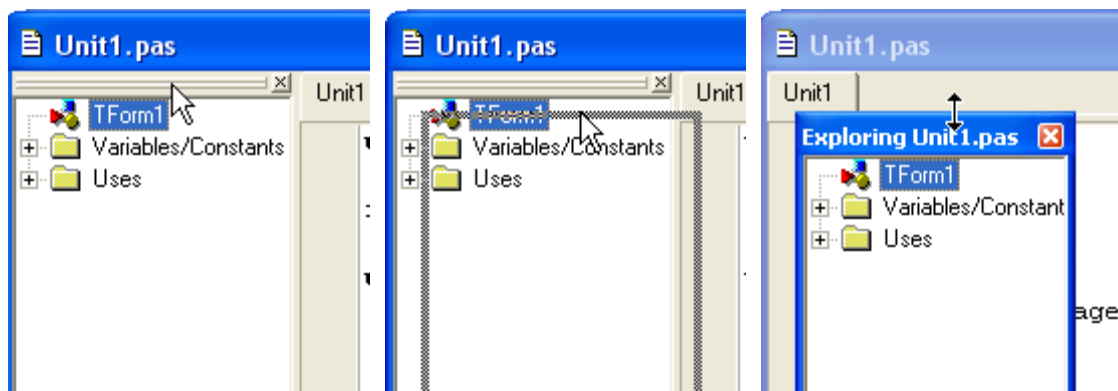
Al hacer visible la ventana de edición se puede observar que la unidad de la "forma" ya tiene escrito código, aunque no se ha escrito realmente nada. Este código es escrito automáticamente por Delphi cuando por ejemplo se coloca un nuevo objeto en la aplicación o se hace doble clic en el campo de un evento. A menos que sea un programador experto, no debe borrar ni modificar el código escrito por Delphi.

Como se puede observar la ventana de edición no sólo muestra la "unidad", sino también el "explorador de código" en la parte izquierda. Puesto que el explorador ocupa parte de la ventana de edición, puede resultar conveniente en ocasiones ocultarlo haciendo clic en el icono de cierre:



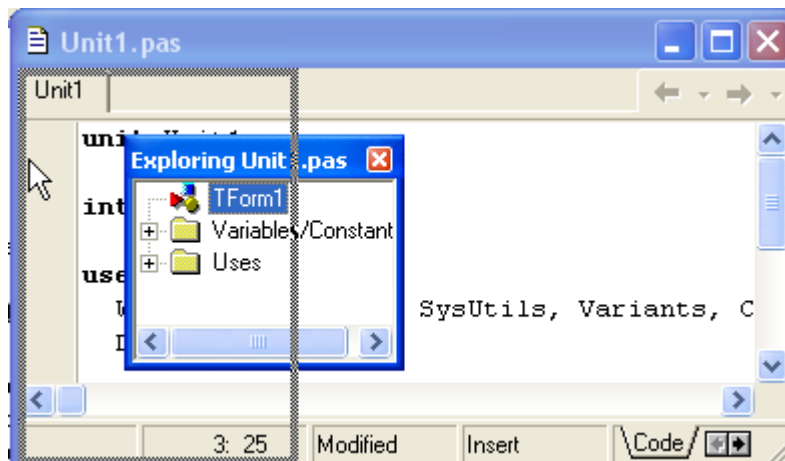
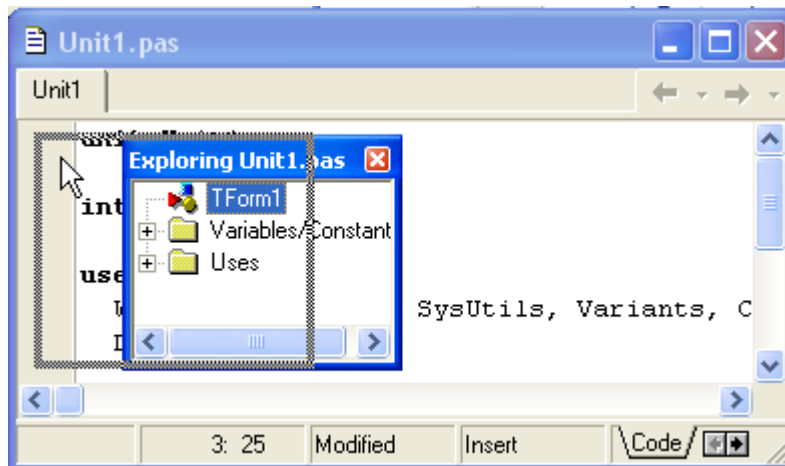
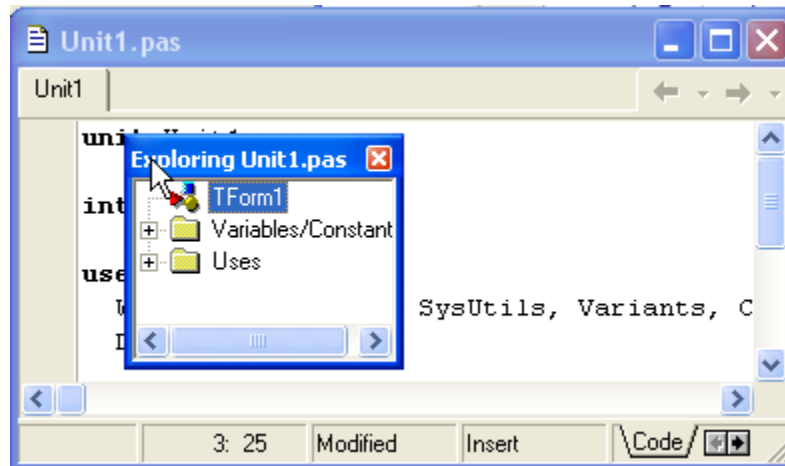
Posteriormente se puede hacer visible el explorador de código pulsando las teclas Shift+Ctrl+E (View->Code Explorer).

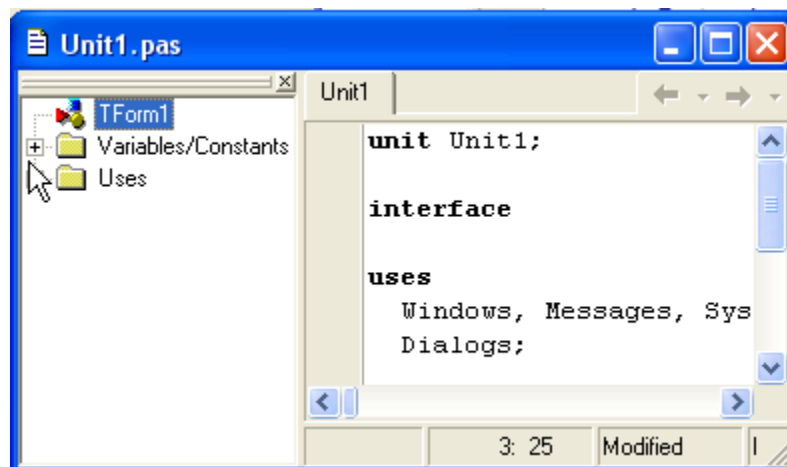
Es posible también separar el explorador de la ventana de edición arrastrándolo fuera de su posición:



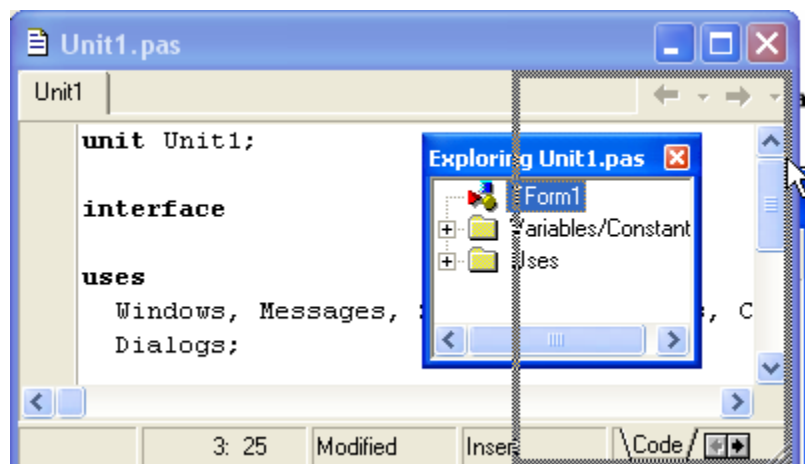
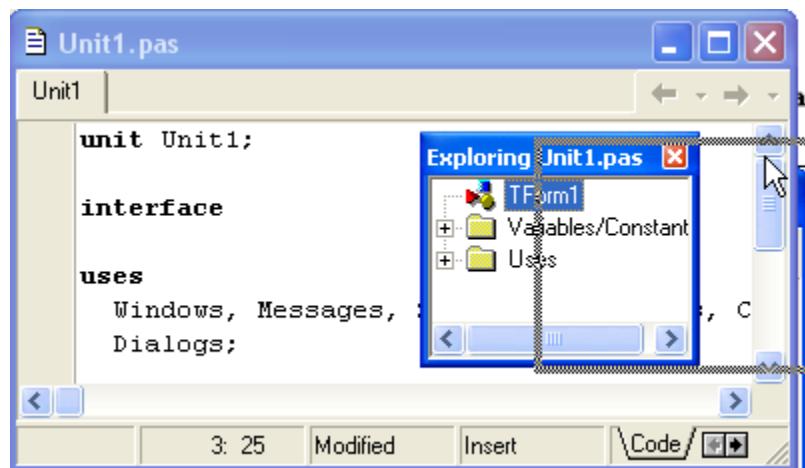
Una vez separado, su tamaño puede ser cambiado y reacomodado en una posición más conveniente. Este procedimiento es válido también para otras ventanas compuestas de Delphi.

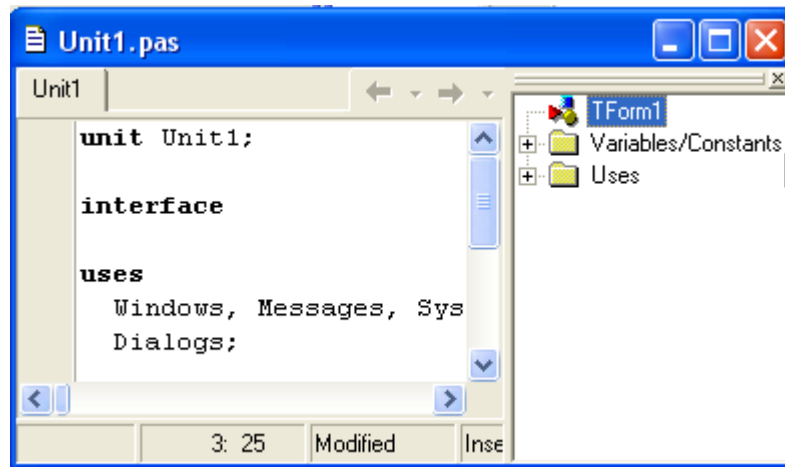
La ventana del explorador de código puede ser reinserada en la ventana de edición arrastrándola hacia uno de sus bordes. Al llegar el puntero al borde de la ventana de edición se crea un recuadro, correspondiente al lugar donde será insertada la ventana del explorador, entonces se suelta la ventana y queda insertada en esa posición. Así para insertar la ventana a la izquierda se procede de la siguiente manera:



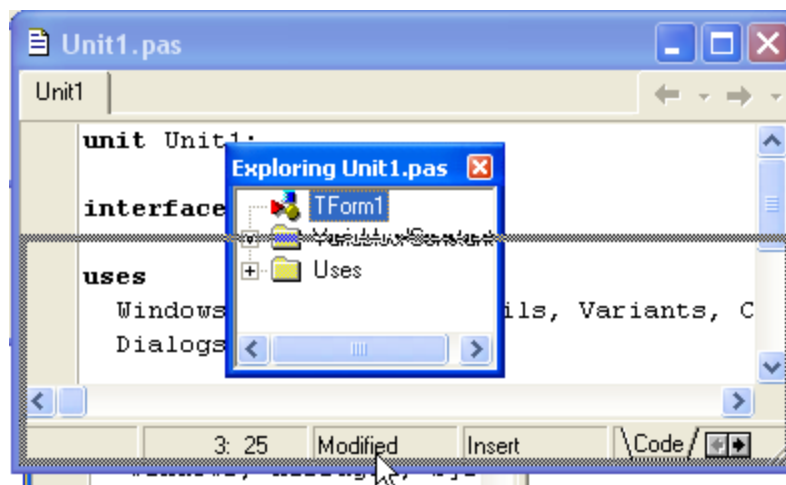
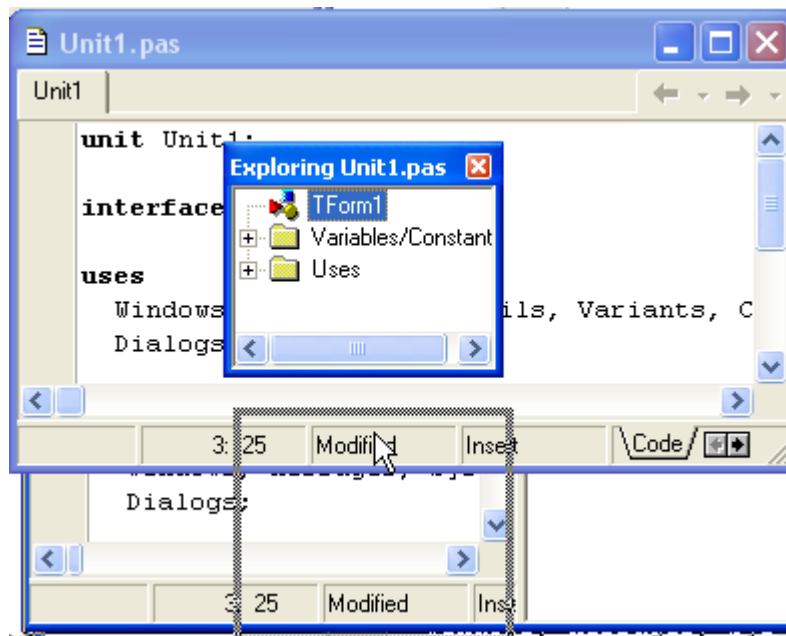


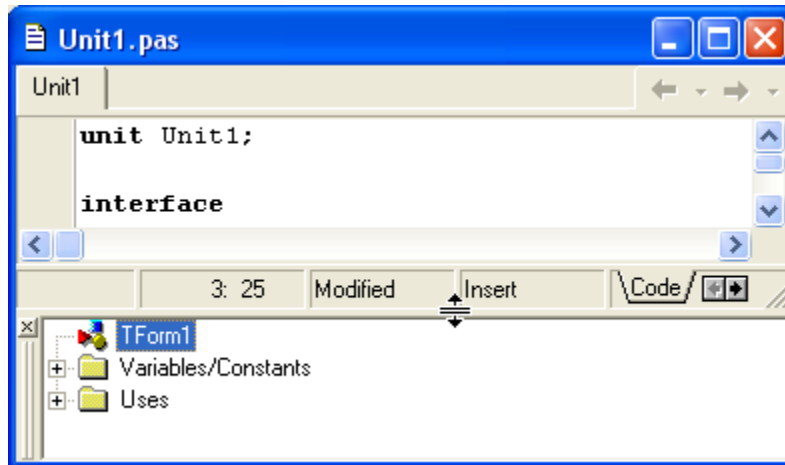
De manera similar se puede insertar la ventana a la derecha:





También es posible insertar la ventana abajo:





Los procedimientos antes mostrados son válidos no solo con la ventana del explorador y la ventana de edición, sino con la mayoría de las ventanas de Delphi (haga la prueba).

2.1.1 Escritura de código empleando objetos

Como ya vimos en el anterior acápite, las propiedades más frecuentes de los objetos pueden ser modificadas directamente en el inspector de objetos. En la mayoría de los casos elegiremos este método, pues aparte de su sencillez es un método seguro, sin embargo, existen ocasiones donde es necesario modificar una propiedad cuando la aplicación está corriendo y otras en las que la propiedad que se quiere modificar no existe en el inspector de objetos. En esos casos no queda otra alternativa que modificar la propiedad (o leer su valor) mediante código.

En general, las propiedades de un objeto se comportan igual que cualquier variable simple del mismo tipo, sólo que la propiedad debe ser escrita precedida del nombre del objeto de acuerdo a la siguiente sintaxis:

Nombre_del_objeto.Propiedad

Por ejemplo para cambiar el título (Caption) de la forma "Form1" por "Mi Aplicación" e incrementar el ancho de la forma en 20 puntos se escribe:

```
Form1.Caption:= 'Mi Aplicación';
Form1.Width:= Form1.Width+20;
```

Como ya se vio anteriormente, algunas propiedades pueden ser a su vez objetos (tal como las propiedades "Font" y "HorzScrollBar"), en esos casos las propiedades de esta propiedad deben estar precedidas por el nombre de la propiedad y el nombre del objeto de acuerdo a la siguiente sintaxis:

Nombre_del_objeto.Propiedad.Propiedad

Por ejemplo para cambiar el tipo de letra a "Times New Roman", el tamaño de la fuente a 18 puntos y el color a azul, se escribe:

```
Form1.Font.Name:= 'Times New Roman';
Form1.Font.Size:= 18;
Form1.Font.Color:= clBlue;
```

Puede ocurrir incluso que la propiedad de una propiedad sea a su vez otro objeto y así sucesivamente, en estos casos simplemente se extiende el procedimiento anterior hasta el nivel que sea necesario. Así por ejemplo para cambiar el estilo de la brocha (*brush*) del lienzo (*canvas*) de la forma (*form*) a "bsFDiagonal" se escribe:

```
Form1.Canvas.Brush.Style := bsFDiagonal;
```

Los métodos de un objeto sólo pueden ser empleados mediante código (pues no aparecen en el inspector de objetos). Para ello se sigue la misma sintaxis que las propiedades. Por ejemplo, para crear un rectángulo con el método "Rectangle" y escribir el texto "DELPHI 7.0" con el método "TextOut", de la propiedad "Canvas" de la forma se escribe:

```
Form1.Canvas.Rectangle(10,10,210,210);  
Form1.Canvas.TextOut(45,90,'DELPHI 7.0');
```

Opcionalmente y con el único objetivo de reducir el código se puede emplear la instrucción With, de acuerdo al siguiente formato:

```
with Nombre_del_Objeto do begin  
    ...  
    instrucciones  
    ...  
end
```

Cuando se emplea la instrucción "with" se escribe directamente el nombre de la propiedad (o método) sin que sea necesario precederla del nombre del objeto. Por ejemplo las anteriores instrucciones, empleando "with", serían:

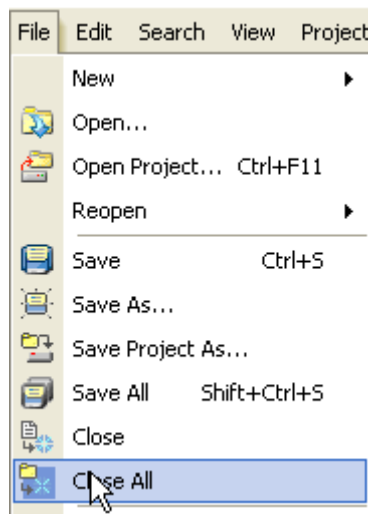
```
with Form1.Canvas do begin  
    Rectangle(10,10,210,210);  
    TextOut(45,90,'DELPHI 7.0');  
end;
```

Con la instrucción "with" se puede trabajar también con dos o más objetos a la vez, en cuyo caso se escriben los nombres de los objeto separados por comas, no obstante esta práctica no es muy recomendable, porque dos o más objetos pueden tener propiedades con los mismos nombres, lo que puede dar lugar a confusiones y errores en la escritura del código.

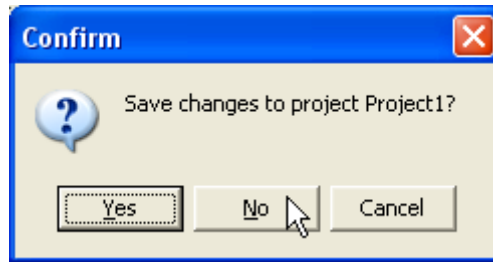
2.2 Creación de una sencilla aplicación en Delphi

Para comenzar a acostumbrarnos al uso de Delphi escribiremos una aplicación muy sencilla, que lo único que hará es mostrar el texto "HOLA MUNDO" cuando se haga doble clic sobre la forma.

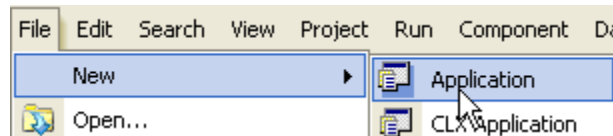
Lo primero que debemos hacer es cerrar la aplicación actual:



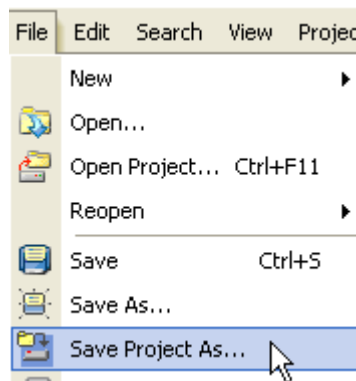
Si se ha hecho alguna modificación, entonces Delphi mostrará la siguiente ventana de diálogo, pidiendo confirmación para guardar (Yes) o no (No) las modificaciones:



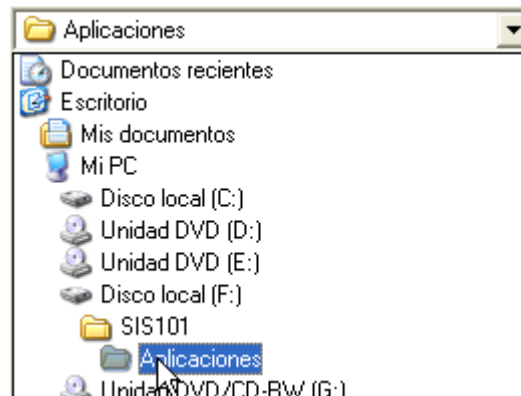
Como hasta ahora no se ha creado realmente ninguna aplicación, hacemos clic en "No". Entonces procedemos a crear una nueva aplicación:



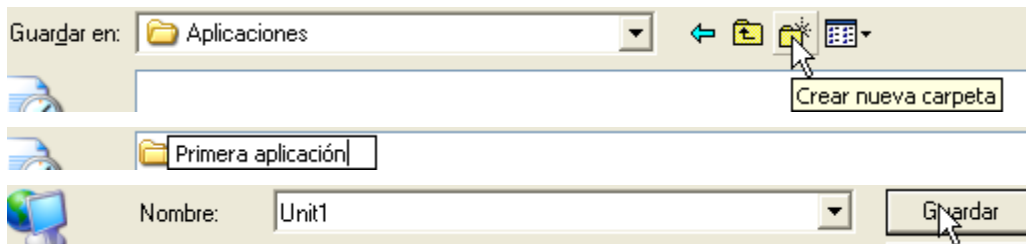
Con lo que aparece una nueva aplicación en blanco, y aún cuando no hemos hecho todavía nada en ella procedemos a guardarla, esto con la finalidad de asignarle nombres significativos tanto al programa como a la unidad que contiene el código de la forma:



En la ventana del explorador de Windows que aparece, seleccionamos el directorio donde guardaremos las aplicaciones: "F:\SIS101\Aplicaciones" para este ejemplo:



Dentro de este directorio creamos un nuevo directorio para guardar los archivos de la aplicación (esto es algo que debe hacer siempre, pues toda aplicación consta de por lo menos 6 archivos):



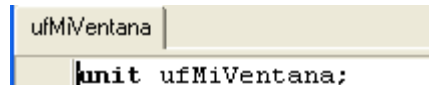
Ahora cambiamos el nombre de "Unit1" a "ufMiVentana":



Y el nombre del programa de "Project1" a "pMiVentana":



Donde la "uf" son las iniciales de nidad y forma respectivamente y "p" es la inicial de programa. Ahora si ve la ventana de edición observará que el nombre de la unidad ha cambiado a "ufMiVentana":

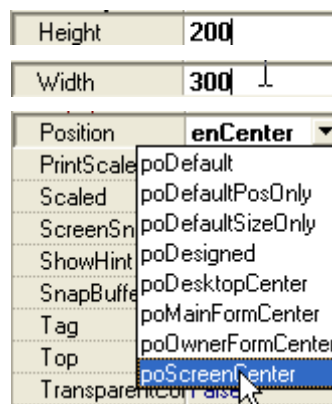


Y lo mismo ha ocurrido con el nombre del programa (Ctrl+F12). Ahora que tanto el programa como el proyecto han sido guardados, cualquier modificación futura puede ser guardada con save:  (o File->Save), si se ha modificado un solo archivo, o save all:  (o File->Save All), si se han modificado dos o más archivos.

Cambiamos ahora el nombre de la forma a "fMiVentana" y el título de la forma a "Mi Primera Aplicación:"



Cambiamos el alto, el ancho y la posición inicial de la forma:



Finalmente escribimos el código en el evento onDblClick de la forma:



```
procedure TfMiVentana.FormDblClick(Sender: TObject);
begin
    fMiVentana.Canvas.TextOut(100,70,'HOLA MUNDO');
end;
```


Ahora guardamos los cambios (File->Save All) y hacemos correr la aplicación:  ("F9" o Run->Run), entonces al hacer doble clic sobre la forma aparece el texto:



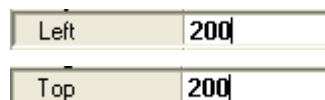
Para cerrar la aplicación, se hace clic en el icono de cierre de la ventana:  (o Alt+F4).

2.3 Segunda aplicación en Delphi

Como segundo ejercicio crearemos una aplicación un tanto más extensa, donde cambiaremos el color de la forma, dibujaremos algunas figuras y escribiremos texto con un tipo de letra, tamaño y estilo determinados.

En primer lugar y al igual que la aplicación anterior, cerramos la aplicación actual (File->Close All), creamos una nueva aplicación (File->New->Application) y guardamos la nueva aplicación en la carpeta "Segunda aplicación" con el nombre "ufMiVentana2" para "Unit1" y "pMiVentana2" para "Project1". Cambiamos también el nombre de la forma a "fMiVentana2", el título de la forma a "Mi Segunda Aplicación", el alto y ancho de la forma a 200 y 300 puntos respectivamente (vea el anterior ejercicio para más detalles).

Dejaremos la posición (Position) en poDesigned y colocaremos la ventana a 200 puntos tanto del margen izquierdo como del superior de la pantalla:



Si bien podemos cambiar el color de la forma directamente en el inspector de objetos, para acostumbrarnos a programar eventos, cambiaremos el color a "clInactiveCaption" en el evento onCreate:

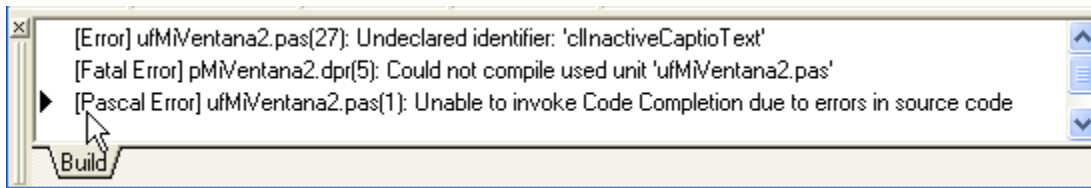


```
procedure TfMiVentana2.FormCreate(Sender: TObject);
begin
  fMiVentana2.Color:= clInactiveCaptionText;
end;
```

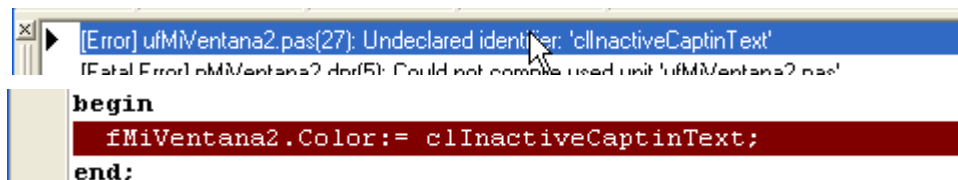
Para ver el efecto de esta instrucción, compile el proyecto, pulsando las teclas *Ctrl+F9* o a través del menú *Project -> Compile Project1*. Entonces si no ha cometido ningún error aparecerán unos puntitos azules a la izquierda del código:

```
 fMiVentana2.Color:= clInactiveCaptionText;
 end;
```

Por el contrario si ha cometido algún error, aparecerá el mensaje de error (o los mensajes de error) en una nueva ventana ubicada en la parte inferior de la ventana de edición. Por ejemplo si en lugar de `clInactiveCaptionText` ha escrito `clInactiveCaptinText`, al compilar aparece la siguiente ventana (haga la prueba):



Que nos informa con relación al error y el lugar donde se encuentra. Para corregir el error se hace doble click en el mensaje de error, (las líneas que comienza con [Error]) con lo que el cursor se ubica en el lugar donde se ha detectado el error, para que pueda ser corregido.



Una vez corregidos los errores se puede hacer correr el programa (*F9*, Run -> Run, o ) con lo que se puede ver el nuevo color de la forma.

Para continuar escribiendo código cierre la ventana de la aplicación ( o *Alt+F4*). Para dibujar y escribir en la forma (y en otros objetos gráficos), Delphi nos da acceso al lienzo (Canvas), el lápiz (Pen) y la brocha (Brush) de Windows. Tanto el lápiz como la brocha de Windows son especiales: no sólo permiten pintar con diferentes colores y en diferentes grosores, sino también con una variedad de patrones. La brocha inclusive permite pintar empleando como modelo una imagen.

Todos los objetos gráficos de Windows tienen su propio lápiz y brocha. Para cambiar la forma en que pintan se deben modificar sus propiedades.

Algunas de las propiedades más usuales del lápiz son: "Color", para cambiar el color del lápiz, "Width" para cambiar el ancho o grosor del lápiz, y "Style" para cambiar el modelo (o patrón) con el que pinta el lápiz, siendo los patrones disponibles: `psSolid` (línea sólida), `psDash` (línea con guiones), `psDot` (línea con puntos), `psDashDot` (línea con guiones y puntos alternados), `psDashDotDot` (línea con guiones y dos puntos alternados), `psClear` (sin línea), `psInsideFrame` (línea sólida pero que puede utilizar un diferente color si el ancho del lápiz es mayor a 1). Así para dibujar un rectángulo, cuando se hace clic en la forma, empleando una línea de 5 puntos de grosor, en color verde y con guiones escribimos lo siguiente en el evento "on-Click":

```
procedure TfMiVentana2.FormClick(Sender: TObject);
begin
    fMiVentana2.RePaint;
    fMiVentana2.Canvas.Pen.Width:= 8;
    fMiVentana2.Canvas.Pen.Color:= clGreen;
    fMiVentana2.Canvas.Pen.Style:= psDash;
    fMiVentana2.Canvas.Rectangle(70,40,170,120);
end;
```

Donde, "Repaint", es el método que vuelve a dibujar el objeto (en este caso la forma), lo en este ejemplo causa el borrado de cualquier dibujo o texto existente, de manera que si esta instrucción se escribe al final y no al principio, no se llega a ver ningún cambio pues el rectángulo es borrado inmediatamente termina de ser dibujado (haga la prueba). Ejecutando la aplicación y haciendo clic en la forma se obtiene:



Como se dijo, la brocha también es especial siendo sus propiedades más usuales: "Color" para cambiar el color de la brocha, "Style" para seleccionar el patrón o modelo con el que pinta la brocha y "Bitmap" para establecer la imagen que será empleada como patrón. Los patrones disponibles para la propiedad "Style" son: bsSolid ■, bsCross ▦, bsDiagCross ▩, bsHorizontal ≡, bsVertical |||, bsFDiagonal \\\, bsBDiagonal /// y bsClear .

Así para crear un círculo donde el borde (lápiz) tenga un grosor de 3 puntos, sea sólido y de color rojo y el relleno (brocha) sea de color azul, con un patrón de líneas diagonales, escribimos el siguiente código en el evento "onDblClick":

```
procedure TfMiVentana2.FormDblClick(Sender: TObject);
begin
  fMiVentana2.Repaint;
  fMiVentana2.Canvas.Pen.Width:= 3;
  fMiVentana2.Canvas.Pen.Color:= clRed;
  fMiVentana2.Canvas.Pen.Style:= psSolid;
  fMiVentana2.Canvas.Brush.Color:= clBlue;
  fMiVentana2.Canvas.Brush.Style:= bsFDiagonal;
  fMiVentana2.Canvas.Ellipse(70,10,220,160);
end;
```

Ejecutando la aplicación y haciendo doble clic en la forma se obtiene:



Cuando se escribe texto en el lienzo, con el método "TextOut" o "TextRect", se puede cambiar el tipo de letra a través de la propiedad "Font", cuyas propiedades más usuales son: "Color" para el color de la letra, "Name" para el nombre del tipo de letra, "Size" para el tamaño de la letra en puntos y "Style" para el estilo de la letra, pudiendo ser el estilo: fsBold (negrita), fsItalic (cursiva), fsUnderline (subrayado) y fsStrikeOut (tachado), escritos en forma de conjunto, es decir entre corchetes y separados con comas.

Así para escribir el texto "Mi Aplicación" con el tipo de letra "Alba Super", en color rojo, con un tamaño de 35 puntos y con los estilos: negrita, cursiva y subrayado, escribimos lo siguiente en el evento "onContextPopup", este evento se activa cuando se hace clic con el botón derecho del mouse:

```
procedure TfMiVentana2.FormContextPopup(Sender: TObject; MousePos: TPoint;
var Handled: Boolean);
begin
  fMiVentana2.Repaint;
  fMiVentana2.Canvas.Font.Name:= 'Alba Super';
  fMiVentana2.Canvas.Font.Color:= clRed;
  fMiVentana2.Canvas.Font.Size:= 35;
  fMiVentana2.Canvas.Font.Style:= [fsBold,fsItalic,fsUnderline];
  fMiVentana2.Canvas.TextOut(10,30,'Mi Ventana');
end;
```

Ejecutando la aplicación y haciendo clic con el botón derecho del mouse se obtiene:



2.4 Tercera aplicación en Delphi

Como tercer ejercicio crearemos una aplicación que empleará una figura como el patrón de la brocha.

Para ello y al igual que en los anteriores ejercicio cerramos la aplicación actual (File->Close All), no olvidando antes guardar las modificaciones hechas (File->Save All), creamos una nueva aplicación (File->New->Application), guardamos la nueva aplicación (File->Save Project As...) en el directorio "Tercera aplicación", con los nombres "ufVentana3" para "Unit1" y "pVentana3" para "Project1".

Cambiamos también el nombre (Name) de la forma a "fVentana3", el título (Caption) de la forma a "Mi Tercera Aplicación", el ancho (Width) a 300 puntos, el alto (Height) a 200 puntos y la posición (Position) a "poDesktopCenter".

Ahora procedemos a fijar como patrón de la brocha una imagen, puesto que el objetivo de la presente aplicación es que la imagen sea el fondo de la

forma, emplearemos la brocha de la forma y no la del lienzo (canvas). Para este fin recurrimos a la propiedad "Bitmap" de la brocha, la cual es a su vez otro objeto que cuenta con varios métodos para cargar una imagen, siendo uno de los más usuales "LoadFromFile(Nombre del archivo)", el cual carga la imagen desde un archivo de tipo BMP. Para este ejercicio cargaremos la imagen del archivo "abanicos.bmp", la cual generalmente se encuentra el directorio Windows del disco duro, sin embargo, si no cuenta con este archivo puede emplear cualquier otro que tenga la extensión BMP:

```
procedure TfVentana3.FormCreate(Sender: TObject);  
begin  
    fVentana3.Brush.Bitmap:= TBitmap.Create;  
    fVentana3.Brush.Bitmap.LoadFromFile('C:\Windows\abanicos.bmp');  
end;
```

Sin embargo no haga correr todavía la aplicación, porque como en este caso estamos creando un objeto, con la instrucción "TBitmap.Create", debemos encargarnos también de destruirlo, pues de lo contrario quedaría registrado en la memoria haciendo que la misma se vaya corrompiendo y provocando fallos del sistema. Para destruir (o liberar) un objeto se emplea el método *Free*, el cual debe ser llamado cuando no se utiliza más el objeto en la aplicación. En el presente ejercicio, el objeto creado (el Bitmap) se emplea mientras corre la aplicación (pues es el fondo de la misma), por lo que el momento adecuado para destruirlo es cuando la aplicación se cierra, es decir cuando ocurre el evento "OnClose":

```
procedure TfVentana3.FormClose(Sender: TObject; var Action: TCloseAction);  
begin  
    fVentana3.Brush.Bitmap.Free;  
end;
```

Ahora si puede hacer correr la aplicación (F9):



El mismo procedimiento se sigue para asignar una imagen a cualquier brocha, así, en el evento "onClick" crearemos un rectángulo el cual tendrá de relleno la imagen correspondiente al archivo "pompas.bmp" que también se encuentra en el directorio Windows. Previamente debemos modificar el evento "onCreate" para crear en el mismo otro objeto "Bitmap" para la brocha del lienzo:

```
procedure TfVentana3.FormCreate(Sender: TObject);  
begin  
    fVentana3.Brush.Bitmap:= TBitmap.Create;  
    fVentana3.Brush.Bitmap.LoadFromFile('C:\Windows\abanicos.bmp');  
    fVentana3.Canvas.Brush.Bitmap:= TBitmap.Create;  
    fVentana3.Canvas.Brush.Bitmap.LoadFromFile('C:\Windows\pompas.bmp');  
end;
```

Entonces el código del evento "onClick" es:

```
procedure TfVentana3.FormClick(Sender: TObject);
begin
    fVentana3.Repaint;
    fVentana3.Canvas.Rectangle(100,50,200,100);
end;
```

Una vez más, antes de hacer correr la aplicación, no se debe olvidar destruir el nuevo objeto creado, por lo que debemos modificar también el evento "onClose":

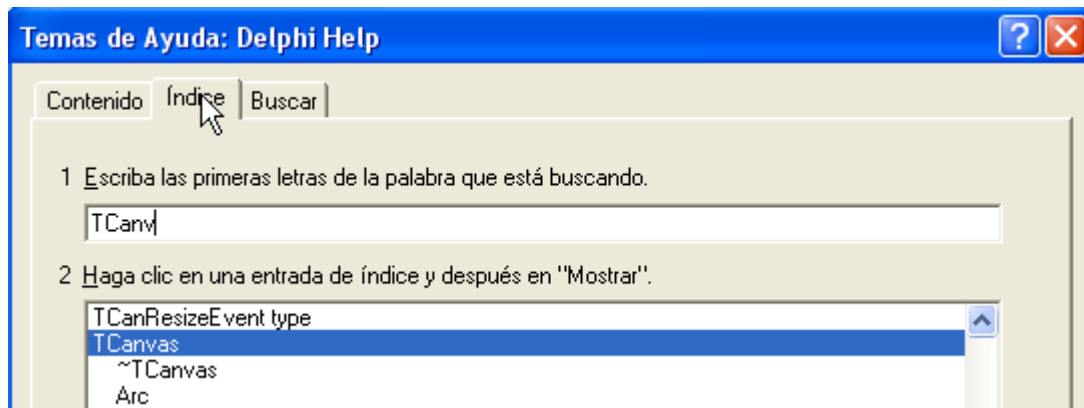
```
procedure TfVentana3.FormClose(Sender: TObject; var Action: TCloseAction);
begin
    fVentana3.Brush.Bitmap.Free;
    fVentana3.Canvas.Brush.Bitmap.Free;
end;
```

Ejecute la aplicación (F9) y haga clic en la misma:



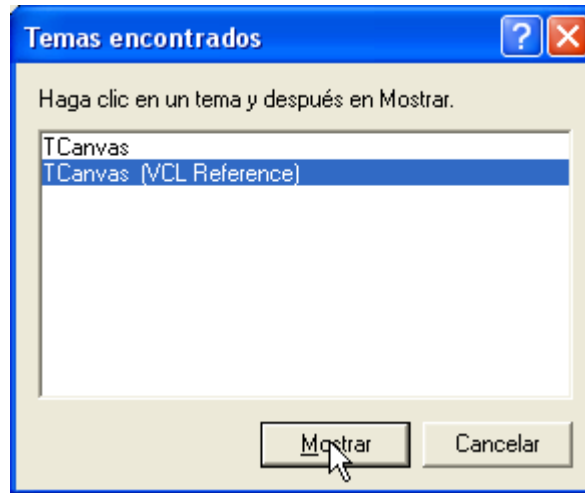
2.5 La ayuda en línea de Delphi

Hasta el momento hemos empleado sólo algunas de las propiedades y métodos de algunos objetos. Para ver todas las propiedades, métodos y eventos de las clases, así como para obtener ayuda con relación a cualquier aspecto del entorno de Delphi, debe recurrir a la ayuda:  (o Help->Delphi Help). En la página "Índice" de la ayuda se escribe parte del nombre del elemento con relación al cual se quiere pedir ayuda, hasta que se visualiza el nombre completo, por ejemplo para obtener ayuda de la clase "TCanvas" escribimos:

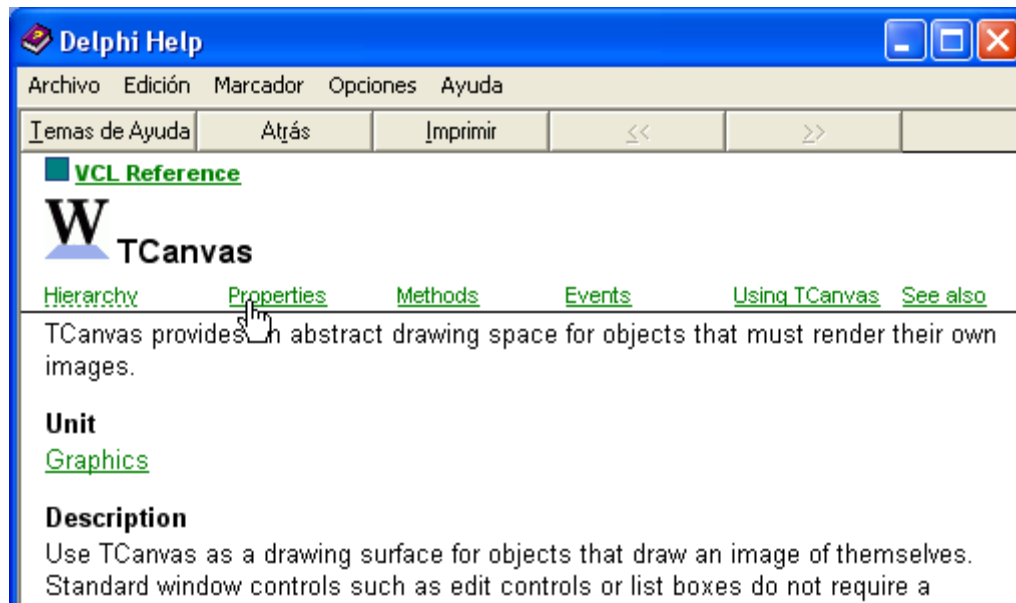


Entonces se pulsa la tecla "Enter" o se hace doble clic en el nombre de la clase, con lo que aparece una nueva ventana, para elegir más específica-

mente la ayuda a consultar, esto debido a que Delphi 7.0, cuenta con dos conjuntos de clases, uno exclusivo para Windows (VCL) y otro multiplataforma para Windows y GNU-Linux (CLX). En este curso emplearemos los componentes de Windows (VCL), por lo que siempre elegiremos la opción "VCL":



Entonces aparece la ventana de ayuda de este componente, donde podemos ver la unidad a la que pertenece (algo que como veremos luego es importante al momento de desarrollar aplicaciones), una breve descripción del objeto y enlaces a sus propiedades, métodos, eventos, temas relacionados (See also), forma de uso (Using TCanvas) y en ocasiones ejemplos (Examples):



Haciendo clic en los enlaces se puede ver en detalle las propiedades, métodos y eventos de cada clase. También se puede acceder a la ayuda con relación a una propiedad o método específico si se pulsa la tecla "F1" estando en una propiedad o evento en el inspector de objetos.

Como seguramente ya ha notado, Delphi le brinda ayuda permanente al momento de escribir código, basta con que escriba el nombre del objeto, propiedad o método (seguido de un punto) para que Delphi le muestre todos los procedimientos y métodos del mismo. Igualmente puede pedir ayuda con relación a cualquier instrucción de Delphi, colocando el cursor en la instrucción y pulsando "Ctrl+F1" (haga la prueba).

2.6 Ejercicios

1. Elabore una aplicación que cambie el fondo de la forma a líneas cruzadas de color rojo. Al hacer clic en la forma deberá aparecer un círculo amarillo relleno con líneas diagonales, al hacer doble clic un rectángulo azul de 4 puntos de grosor, relleno con líneas verticales verdes y al hacer clic con el botón derecho del mouse deberá aparecer en la forma su nombre y apellido escritos en "Times New Roman", color clBackground, con 30 puntos y con los estilos negrita y cursiva.
2. Elabore una aplicación cuya forma que tenga como fondo la figura "plumas.bmp" y que al hacer clic sobre la misma aparezca una elipse de color rojo, con 6 puntos de grosor, siendo su relleno la figura "vientto.bmp" (u otra figura de su elección).
3. Elabore una aplicación que tenga como fondo la figura "Gotas de agua.bmp", al hacer clic en la forma debe aparecer un cuadrado de color amarillo con un relleno cuadriculado de color verde. Al hacer doble clic debe aparecer su nombre y apellido escrito con el tipo de letra "Arial", en color Rojo, con un tamaño de 25 puntos, con los estilos negrita, cursiva y subrayado. Al hacer clic con el botón derecho del mouse deberá aparecer una elipse dibujada con línea discontinua de color negro, relleno con la figura "lienzo.bmp" (u otra figura de su elección).

3. SECUENCIA: MÓDULOS - OPERADORES

En este capítulo comenzaremos el estudio de la primera estructuras estándar: *la secuencia*. El conocimiento y aplicación de esta estructura nos ayudará a aplicar el segundo principio de la programación estructurada: *emplear estructuras estándar para construir la totalidad del programa*. Sin embargo, no sólo aplicaremos este principio, sino los tres principios de la programación estructurada.

El objetivo del presente capítulo es que al concluir el mismo estén capacitados para resolver problemas simples empleando la estructura secuencial y aplicando los principios de la programación estructurada.

No debemos olvidar que el propósito de los principios de la programación estructurada es el lograr un estilo de programación claro y ordenado, de manera que facilite la creación y mantenimiento del software.

3.1 Diagramas de actividades

Antes de comenzar el estudio de la secuencia explicaremos la simbología que emplearemos en la elaboración de los algoritmos: los *diagramas de actividades*.

Los algoritmos describen la secuencia lógica de acciones que deben llevarse a cabo para resolver un determinado problema. Existen muchas formas en las que se pueden elaborar algoritmos, de ellas emplearemos en esta materia los *diagramas de actividades*.

Los *diagramas de actividades* son uno de las varias clases de diagramas que forman parte de *UML*, el *Lenguaje de Modelado Unificado*, que actualmente es el lenguaje estándar para el modelado de sistemas de software. Se ha elegido este tipo de diagramas no sólo por ser un estándar, sino porque permiten expresar los algoritmos de manera clara, ordenada, sencilla y porque además son muy versátiles, permitiendo representar diferentes tipos de estructuras.

El inicio de un diagrama de actividades se representa con un círculo relleno:



El final del diagrama de actividades se representa con un círculo que contiene un círculo relleno en su interior:



Una acción (o sentencia) se representa con un rectángulo con sus bordes redondeados:



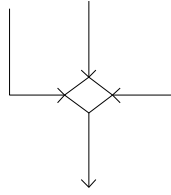
Un flujo, que es el símbolo empleado para unir los elementos del diagrama y señalar la dirección en que se deben seguir las acciones, se representa por una flecha continua abierta:



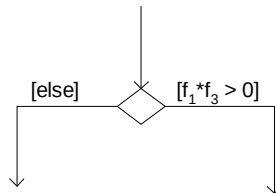
Una unión o bifurcación se representa por un pequeño rombo:



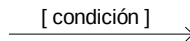
Cuando se emplea como unión llegan dos o más flujos y sale uno.



Cuando se emplea como bifurcación ingresan uno o más flujos y salen dos o más flujos acompañados de alguna condición.



Una condición, tal como se puede observar en la anterior figura, se representa como texto encerrado entre corchetes y siempre está relacionado a algún flujo:



El texto de la condición puede ser una expresión matemática, una expresión relacional, una condición textual, etc. Para la condición por defecto se puede emplear *[else]* (caso contrario).

Una actividad representa un conjunto de acciones (o un subprograma) que se detalla luego por separado empleando otro diagrama de actividades. Se representa como una acción con un icono de una acción llamando a otra en la parte inferior derecha:



Las notas se emplean para comentar y documentar los diagramas de actividades. Se representan como una hoja con una esquina doblada y asociada, mediante una línea discontinua, a cualquiera de los elementos de un diagrama de actividades:



Al elaborar un diagrama de actividades se debe tener presente que es un lenguaje y como tal es necesario respetar su sintaxis (es decir los símbolos) pues cada uno de ellos tiene su propio significado y si se cambia, cambia también la lógica del algoritmo. Por ejemplo, para representar un flujo siempre se debe emplear la flecha continua abierta, no una flecha cerrada y rellena: $\longrightarrow$, una flecha cerrada no rellena: $\longrightarrow$, o una flecha discontinua abierta: $\cdots\longrightarrow$ pues cada una de ellas tienen un significado muy diferente (mensaje, herencia y dependencia respectivamente).

3.2 Diagrama de actividades de una secuencia

La *secuencia* es la estructura básica para la elaboración de cualquier programa. Una secuencia es una serie de instrucciones que se ejecutan una a continuación de la otra. El diagrama de actividades de una secuencia se ve aproximadamente como se muestra en la siguiente figura:

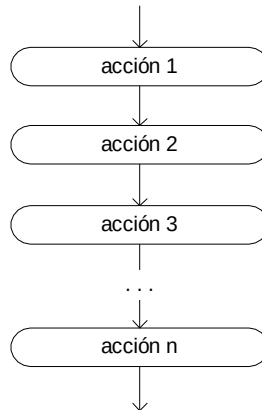


Figura 2.1. Diagrama de actividades de una secuencia

En este diagrama las acciones pueden ser instrucciones simples (como instrucciones de asignación) o instrucciones compuestas (como otra estructura estándar o inclusive otra secuencia). En consecuencia en *programación estructurada*, todo programa, subprograma o módulo es en última instancia una secuencia.

Debido a lo anterior, una secuencia puede ser representada también, de manera abreviada, como una actividad.

3.3 Escritura de una secuencia en Delphi

Puesto que *Delphi* es *Pascal*, la secuencia se implementa de la misma forma que en *Pascal* estándar, es decir encerrándola entre *begin* y *end* y separando las sentencias (o acciones) de su interior con puntos y comas (;):

```
begin  
    instrucción 1;  
    instrucción 2;  
    ...  
    instrucción n;  
end;
```

Como ya se dijo todo programa, subprograma o módulo (procedimiento o función) es en última instancia una secuencia, por ello todo programa y módulo están encerrados siempre entre "*begin*" y "*end*", es decir siempre comienza después de la palabra "*begin*" y termina antes de la palabra "*end*". En todas las aplicaciones que hemos elaborado hasta el momento no ha sido necesario escribir las palabras "*begin*" y "*end*" porque Delphi ha creado por nosotros (al invocar el evento) la estructura del procedimiento y ha escrito en consecuencia las palabras "*begin*" y "*end*", sin embargo en los módulos que creemos manualmente, no debemos olvidar encerrar toda secuencia entre "*begin*" y "*end*".

3.4 Módulos

Para aplicar el primer principio de la programación estructurada debemos resolver el problema en módulos, que en Pascal corresponden a los procedimientos y funciones. Todos los módulos que creemos en esta materia serán escritos dentro de unidades, las cuales, como veremos, son creadas automáticamente por Delphi.

Una unidad es una pieza independiente de software que puede contener diferentes elementos: variables, constantes, etiquetas, tipos de datos, procedimientos y funciones entre otros. Los elementos de una unidad pueden ser

utilizados desde cualquier programa o unidad simplemente escribiendo la instrucción "uses" seguida del nombre de la unidad (o los nombres de las unidades separados con comas). Así por ejemplo la instrucción:

```
uses Windows, Dialogs, MiUnidad;
```

Permite utilizar los elementos que se encuentran en las unidades: *Windows*, *Dialogs* y *MiUnidad*, respectivamente. Una de las ventajas de *Pascal* con relación a otros lenguajes, como *C* por ejemplo, es que sólo se añaden al programa él o los elementos de la unidad que realmente se utilizan y no todos los elementos contenidos en ella.

La estructura básica de una unidad es la siguiente:

```
unit Nombre_de_la_unidad;
interface
  Sector público
implementation
  Sector privado
end.
```

La cual, como dijimos, es creada automáticamente por Delphi cuando se incorpora una nueva unidad al proyecto mediante "File -> New -> Unit".

En el *sector público* (interface) se escriben las variables, constantes, etiquetas, tipos de datos y las cabeceras de los procedimientos y funciones que podrán ser empleados desde cualquier otro programa (o unidad).

En el sector privado (implementation) se escribe el código de los procedimientos y funciones cuyas cabeceras fueron declaradas en el sector público. En este sector se escriben también las variables, constantes, etiquetas, tipos de datos, procedimientos y funciones que sólo son útiles internamente.

Cualquier función, procedimiento, constante, variable, etc., que es declarada en el sector de implementación, pero no en la interfaz, es privada, es decir sólo puede ser empleada en la unidad, pero no desde otras unidades o programas.

Los procedimientos se emplean cuando el módulo no devuelve ningún resultado o devuelve más de un resultado. Su estructura es la siguiente:

```
procedure Nombre_del_procedimiento(lista_de_parámetros);
variables, constantes e identificadores locales
begin
  instrucciones
end;
```

Las funciones se emplean cuando el módulo devuelve **un** resultado. Su estructura es la siguiente:

```
function Nombre_de_la_función(lista_de_parámetros):tipo_de_resultado;
variables, constantes e identificadores locales
begin
  instrucciones;
  result:= resultado; (o Nombre_de_la_función:= resultado;)
  instrucciones;
end;
```

Como se puede observar, la diferencia fundamental entre un procedimiento y una función es que la función siempre devuelve un resultado, y para ello existen dos alternativas: a) asignar el resultado a la variable *result* (que se crea automáticamente en toda función) y b) asignar el resultado al nombre de la función.

Los parámetros son nombres locales que se dan a los datos que se reciben o a los resultados que se devuelve. Los parámetros no son obligatorios, por lo que un módulo puede no tener parámetros.

En Delphi existen diferentes tipos de parámetros, siendo los más usuales los siguientes: por valor, por referencia (var), constantes (const), de salida (out) y por defecto.

Los parámetros por valor, como su nombre sugiere, reciben valores que son almacenados en los parámetros, se declaran de acuerdo al siguiente formato:

Nombre_del_parámetro : tipo_de_dato

Cuando se manda una variable, a un módulo que recibe un parámetro por valor, lo que en realidad se manda es el valor de dicha variable, no la variable en sí, por lo que cualquier modificación que se haga a dicho valor no afecta al valor de la variable original.

Los parámetros por referencia por el contrario, reciben variables, no valores. Entonces cuando se trabaja con un parámetro por referencia en realidad se está trabajando con la variable original (solo que internamente se le asigna otro nombre), por lo tanto cuando se modifica el valor de un parámetro por referencia en realidad se está modificando el valor de la variable original. Los parámetros por referencia se declaran de acuerdo al siguiente formato:

var *Nombre_del_parámetro : tipo_de_dato*

Los parámetros constantes se caracterizan porque no permiten modificar los valores recibidos, por lo que son empleados en aquellos casos donde es importante que los valores originales no sean modificados. Los parámetros constantes se declaran de acuerdo al siguiente formato:

const *Nombre_del_parámetro : tipo_de_dato*

Los parámetros de salida reciben variables (igual que los parámetros por referencia), sin embargo, descartan los valores originales, pues sirven para devolver resultados, no para recibir datos. Los parámetros de salida se declaran de acuerdo al siguiente formato:

out *Nombre_del_parámetro : tipo_de_dato*

Los parámetros por defecto se caracterizan porque tienen asignados valores que son empleados cuando no se manda ningún valor para los mismos. Son útiles en aquellos casos donde algunos datos suelen tener "por defecto" valores predeterminados, que sólo cambian ocasionalmente. Los parámetros por defecto deben ser declarados siempre al final de la lista de parámetros y pueden ser declarados como parámetros por valor o como constantes:

[const] *Nombre_del_parámetro : tipo_de_dato = valor por defecto*

Para emplear un procedimiento o función simplemente se escribe su nombre y los datos entre paréntesis. Por ejemplo, si se han creado los siguientes módulos:

```
function cubo(x: real): real;  
begin  
    result:= x*x*x;  
end;  
procedure Mensaje(s: string);  
begin  
    ShowMessage(Mensaje);  
end;
```

Pueden ser empleados de la siguiente manera:

```
y:= 3*cubo(x)+5;
z:= sqrt(cubo(3))+exp(cubo(y))-4*cubo(4*y);
Mensaje('El resultado ha sido calculado con éxito');
```

Como se ve en los ejemplos, cuando se llama a un procedimiento o función, los nombres de las variables son irrelevantes, sin embargo, son importantes el orden y los tipos de datos, es decir siempre se deben pasar los datos en el orden en que han sido declarados y los tipos de datos deben ser compatibles con los tipos de datos de los parámetros.

3.5 Constantes

Una constante es un identificador al cual se le asigna un valor que no puede ser modificado durante la ejecución del programa. Las constantes son útiles para facilitar la lectura de un programa, pues asignan a ciertos valores nombres significativos. Ayudan también a su mantenimiento, pues el valor es escrito en un solo lugar, de manera que si se ha cometido algún error o es necesario modificarlo, sólo se debe corregir en un lugar del código y no en todos los lugares donde ha sido empleado.

Las constantes se declaran con la palabra "const" de acuerdo al siguiente formato:

```
const nombre_de_la_constante = valor;
```

Así por ejemplo si en una aplicación se emplea con frecuencia el número de Nepper, podemos crear la constante:

```
const e = 2.71828182846;
```

Posteriormente se emplea "e" en todos los lugares donde se requiera el número de Nepper.

3.6 Variables

Una variable es un espacio de memoria reservado al cual se le asigna un nombre (o identificador).

Para declara una variable se emplea la palabra reservada "var" de acuerdo al siguiente formato:

```
var Nombre : Tipo de dato;
```

Si se quiere declarar más de una variable de un mismo tipo, se escriben los nombres de las variables separados con comas, por ejemplo la siguiente instrucción declara 5 variables de tipo cadena:

```
var s1,s2,s3,s4,s5: string;
```

En ocasiones se requiere que la variable no sólo sea declarada, sino que además se le asigne un valor inicial, es decir que sea una variable inicializada. Ello se consigue en Delphi con la palabra reservada "const", de acuerdo al siguiente formato:

```
const Nombre : Tipo de dato = valor inicial;
```

No obstante, en esta declaración existe una contradicción, pues se está empleando la palabra "const" (constante) para declara una variable. Con la intención de evitar esta contradicción Delphi 7.0, no permite (por defecto) que se modifique este valor, así se comporta realmente como una constante y el código parece más coherente. Para declarar variables inicializadas Delphi 7.0 recomienda emplear el siguiente formato:

```
var Nombre: Tipo de dato = valor inicial;
```

Sin embargo, esta declaración sólo funciona con variables globales y no con variables locales, que son las de mayor interés para nosotros pues resolveremos los problemas en módulos. Por lo tanto, no nos queda otra alternativa que emplear la primera forma a pesar de su aparente contradicción. Para que esta declaración funcione en Delphi 7.0 es necesario encender el interruptor de compilación {\$J}, con la instrucción: {\$J+} o {\$WRITEABLECONST ON} pues por defecto, en Delphi 7.0, está apagado: {\$J-} o {\$WRITEABLECONST OFF}. Esta instrucción se puede escribir antes de declarar la variable inicializada:

```

{$J+}
const MiVariable : integer = 7584;

```

O en cualquier lugar del código previo a la declaración (o declaraciones). No obstante, una forma más sencilla de cambiar esta directiva es mediante el menú "Project -> Options... -> Compiler -> Assignable typed constants".

Debemos aclarar no obstante, que dentro de un módulo, la declaración con "const" crea realmente una variable local estática, es decir una variable que mantiene su valor entre llamadas.

Otro aspecto que es importante tener siempre en cuenta, cuando se trabaja con Delphi, es que las variables no tienen ningún valor por defecto, por lo que no se puede asumir, por ejemplo, que son iguales a cero.

3.7 Operadores aritméticos

Los operadores aritméticos que se pueden emplear en Delphi son los siguientes:

Operador	Operación	Tipos de operando	Tipo de resultado
*	Multiplicación	Entero, real	Entero, real
/	División	Entero, real	Real
Div	Cociente	Entero	Entero
Mod	Resíduo	Entero	Entero
+	Suma	Entero, real	Entero, real
-	Resta	Entero, real	Entero, real

El signo "-", cuando se encuentra entre dos números realiza la operación de resta, sin embargo si se encuentra delante de un número o expresión, cambia el signo del número o expresión. En una expresión los operadores se evalúan de acuerdo al siguiente orden (o prioridad):

Prioridad	Operador
1 (más alta)	- (negación o cambio de signo)
2	*, /, Div, Mod
3 (más baja)	+, -

Donde primero se evalúan los operadores de prioridad más alte, luego los de que le siguen y así sucesivamente hasta llegar a los operadores de prioridad más baja. En caso de existir operadores con igual prioridad, como la multiplicación y la división por ejemplo, se evalúan de izquierda a derecha.

Para cambiar el orden de prioridad de los operadores se emplean paréntesis, los cuales tienen el orden de prioridad más alto. Cuando se utilizan paréntesis, se llevan a cabo primero todas las operaciones que se encuentran dentro de los paréntesis, en caso de existir paréntesis anidados (paréntesis dentro de otros paréntesis), se evalúan primero las expresiones que se encuentran en los paréntesis más internos. Dentro de los paréntesis se mantie-

ne el orden de prioridad antes indicado. Así por ejemplo para programar la siguiente expresión:

$$w = \frac{x - y + z}{4xyz}$$

Podemos escribir:

```
w:= (x-y+z)/(4*x*y*z)
```

O también:

```
w:= (x-y+z)/4/x/y/z
```

Es posible también asignar el resultado de una expresión a una constante. En tales expresiones se pueden emplear todos los operadores aritméticos, valores literales y constantes declaradas con anterioridad, pero no procedimientos o funciones. Por ejemplo la siguiente asignación es válida:

```
const x = 5;
      y = 3.4;
      z = 6.78;
      w = (x*y+z)/(x-2*y+3*z);
```

Pero la siguiente no, porque se emplea una función:

```
const x = 6.45;
      y = Exp(x);
```

3.8 Ejemplos

1. En una unidad elabore los módulos que permitan leer de un "Edit" un número real ("x" y "y"), mostrar en un "Edit" un número real ("z") y calcular el valor de "z" con la siguiente expresión:

$$z = \sqrt{\frac{x^2 + 3xy}{2xy^2 + 5x^3}}$$

Luego (en la forma) cree la interfaz con 3 "label", 3 "Edit" y un "Button" y programe el evento "onClick" del "Button" de manera que lea los valores "x" y "y", calcule el valor "z" y muestre el valor calculado.

Puesto que en este caso la lógica es directa, no elaboraremos algoritmos para su solución, sin embargo, como resulta claro del planteamiento, aún en un problema tan simple como este, se divide en módulos más específicos: para calcular el resultado, leer los datos, mostrar el resultado, un módulo principal que llama a los otros módulos y en este caso un módulo adicional para la interfaz de la aplicación.

Para resolver el problema primero creamos una nueva aplicación: "File -> New -> Application", y una nueva unidad: "File -> New -> Unit" y guardamos las unidades y la aplicación (File -> Save All) en el directorio "Secuencia\Ejemplo 1" con los nombres: "fEjemplo1" para "Unit 1", "uEjemplo1" para "Unit 2" y "pEjemplo1" para "Project1".

De estos módulos, el módulo que lee el número real es una función (pues devuelve un resultado), igualmente es una función el módulo que calcula el resultado. El módulo que muestra el resultado es un procedimiento (porque no devuelve un resultado), el módulo que llama a los otros módulos es también un procedimiento y el módulo que modifica la interfaz de la aplicación es también un módulo (porque tampoco devuelve un resultado).

En la interfaz de la unidad "uEjemplo1" escribimos la cabecera de los módulos:

```
unit uEjemplo1;
```

```
interface
```

```
    uses StdCtrls, SysUtils;
```

```
    function LeerNumReal(e: TEdit): real;
```

```
    procedure MostrarNumReal(x: real; e: TEdit);
```

```
    function CalcularZ(x,y: real):real;
```

Donde se han incluido las unidades "StdCtrls", porque en ella se encuentra el tipo "TEdit" y "SysUtils" para las funciones de conversión. En el sector de implementación se escribe el código de estos tres módulos:

```
implementation
```

```
    function LeerNumReal(e: TEdit): real;
```

```
    begin
```

```
        result:= StrToFloat(e.Text);
```

```
    end;
```

```
    procedure MostrarNumReal(x: real; e: TEdit);
```

```
    begin
```

```
        e.Text:= FloatToStr(x);
```

```
    end;
```

```
    function CalcularZ(x,y: real):real;
```

```
    begin
```

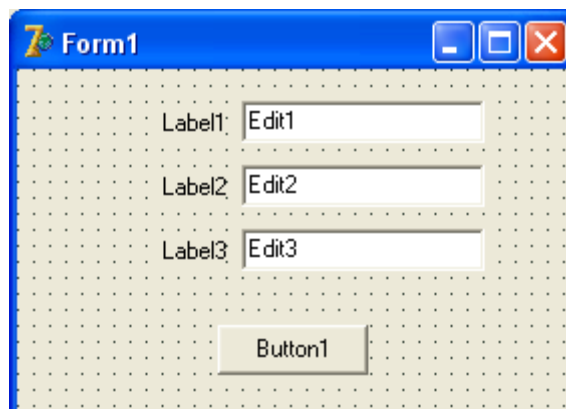
```
        result:= sqrt((sqr(x)+3*x*y)/(2*x*sqr(y)+5*sqr(x)*x));
```

```
    end;
```

```
end.
```

En estos módulos "StrToFloat" convierte un texto a un número real y "FloatToStr" hace lo contrario, es decir convierte un número en texto. Ambas funciones se encuentran en "SysUtils" y esa es la razón por la cual se ha incluido dicha unidad (con uses). Compile la unidad (Ctrl+F9) y guarde los cambios (File->Save).

Ahora colocamos los objetos: 3 TLabel **A**, 3 TEdit **ab** y 1 TButton **OK**, en la forma (alineándolos con la paleta de alineación: "View -> Alignment Palette"):



Y aunque es más sencillo modificar sus propiedades en el inspector de objetos (como ya vimos en el anterior capítulo), las modificaremos en el evento "onCreate", para acostumbrarnos a trabajar con los objetos y acceder así los métodos y propiedades que no visibles en el inspector de objetos:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Form1.Caption:= 'EJEMPLO 1';
  Form1.Brush.Style:= bsFDiagonal;
  Form1.Brush.Color:= clBlue;
  Form1.Icon.LoadFromFile('Sword.ico');
  Label1.Transparent:= True;
  Label1.Alignment:= taRightJustify;
  Label1.Font.Color:= clBlue;
  Label1.Font.Style:= [fsBold];
  Label1.Caption:= 'Valor de x: ';
  Label2.Transparent:= True;
  Label2.Alignment:= taRightJustify;
  Label2.Font.Color:= clBlue;
  Label2.Font.Style:= [fsBold];
  Label2.Caption:= 'Valor de y: ';
  Label3.Transparent:= True;
  Label3.Alignment:= taRightJustify;
  Label3.Font.Color:= clBlue;
  Label3.Font.Style:= [fsBold];
  Label3.Caption:= 'Valor de z: ';
  Edit1.Text:= '0';
  Edit1.Font.Color:= clBlue;
  Edit2.Text:= '0';
  Edit2.Font.Color:= clBlue;
  Edit3.Text:= '0';
  Edit3.Font.Color:= clBlue;
  Edit3.ReadOnly:= True;
  Edit3.TabStop:= False;
  Button1.Caption:= '&Calcular';
  Button1.Default:= True;
end;
```

En este código, la propiedad "Icon", de la forma, permite cambiar el icono de la misma mediante el método "LoadFromFile" (igual que un Bitmap). Para que este código compile es necesario copiar el icono "Sword.ico" (u el icono que se desee) al directorio "Secuencia\Ejemplo 1". La propiedad "Transparent" de los "Label" hace que el fondo del texto sea transparente y la propiedad "Alignment" permite alinear el texto (en este caso a la derecha).

La propiedad "Text" de los "Edit" permite asignar o leer el texto de los mismos, la propiedad "ReadOnly" (sólo lectura) determina si el texto puede ser modificado o no, si está en "True" el texto no puede ser modificado, si está en "False" si. Puesto que el último "Edit" (Edit3) sólo es empleado para mostrar resultados, no debe permitir la modificación del texto, esta es la razón por la cual su propiedad "ReadOnly" ha sido colocada en "True".

Por otra parte en una forma es posible moverse a través de los objetos con la tecla "Tab" (en sentido directo) o "Shif Tab" (en sentido inverso). Para que al navegar el control se detenga en un objeto, su propiedad "TabStop" debe estar en "True" y en "False" para que no se detenga. Puesto que "Edit3" no permitirá modificar texto, no tiene sentido que el control se detenga en el mismo, siendo esa la razón por la cual su propiedad "TabStop" ha sido colocada en "False".

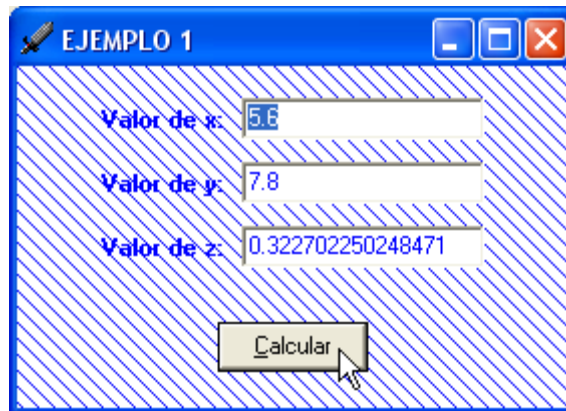
Se ha colocado la propiedad "Default" del botón "Button1" en "True" para que sea el botón por defecto, es decir para que al pulsar la tecla "Enter" el mismo se active, es decir tenga el mismo efecto que un "click" sobre el mismo. Compile la unidad (Ctrl+F9), guarde los cambios (File->Save) y haga correr la misma (F9) para ver los cambios.

Antes de programar el evento "onClick" del "Button1", es necesario incluir en la unidad "fEjemplo1", la unidad "uEjemplo1", pues es en la misma donde están los módulos a los cuales se llama desde este evento. Para ello se puede escribir "uses uEjemplo1;" en el sector de implementación, o dejar que Delphi escriba por nosotros este código "File->Use Unit...->uEjemplo1".

Ahora escribimos el código del evento "onClick" del "Button1":

```
procedure TForm1.Button1Click(Sender: TObject);
var x,y,z: real;
begin
  x:= LeerNumReal(Edit1);
  y:= LeerNumReal(Edit2);
  z:= Calcularz(x,y);
  MostrarNumReal(z,Edit3);
  Edit1.SetFocus;
end;
```

En este módulo "Edit1.SetFocus" hace que el control vuelva a "Edit1", y ello simplemente para facilitar la introducción de nuevos datos. Ejecutando la aplicación (F9) e introduciendo los siguientes datos deberá obtener el resultado que se muestra a continuación:



2. Elabore una aplicación que devuelva el valor de la siguiente función:

$$f(x_i, x, y) = \frac{k_x}{k_y} + \frac{y - y_i}{x - x_i}$$

$$y_i = -0.0405 + 1.03785714286x_i$$

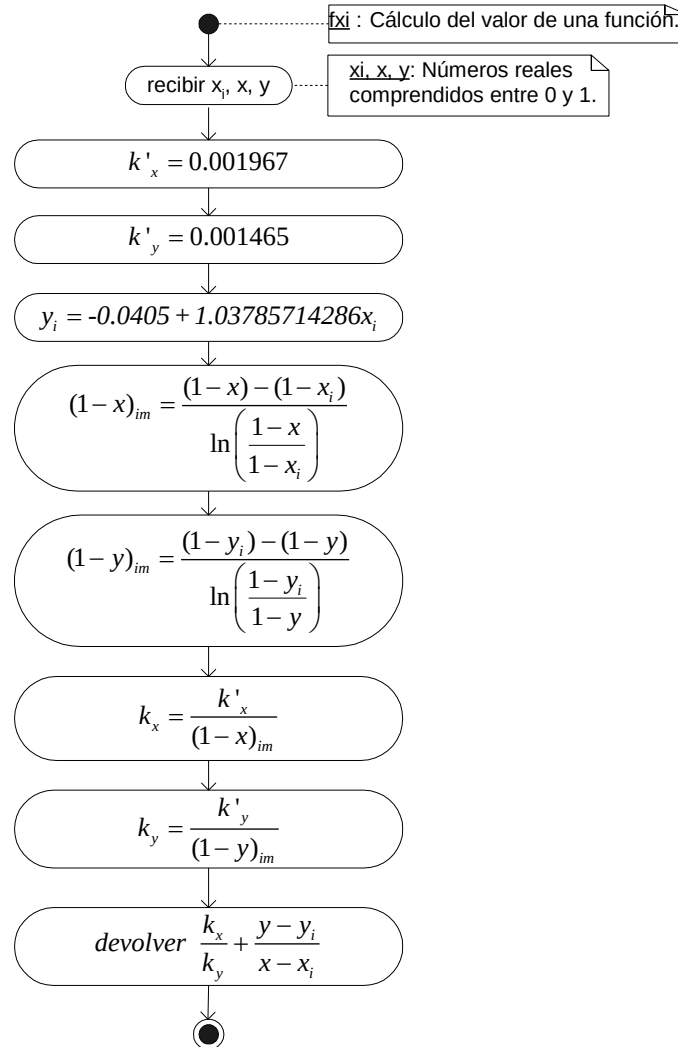
$$k_x = \frac{k'_x}{(1-x)_{im}}; \quad k_y = \frac{k'_y}{(1-y)_{im}}$$

$$(1-x)_{im} = \frac{(1-x) - (1-x_i)}{\ln\left(\frac{1-x}{1-x_i}\right)}; \quad (1-y)_{im} = \frac{(1-y_i) - (1-y)}{\ln\left(\frac{1-y_i}{1-y}\right)}$$

$$k'_x = 0.001967; \quad k'_y = 0.001465$$

Al igual que en la anterior aplicación, el problema se divide en varios módulos (leer datos, mostrar resultado, calcular el valor de la función y 2 módulos para los eventos "onClick" y "onCreate").

En este caso el cálculo de la función implica un cierto orden lógico, por lo que es conveniente elaborar un algoritmo para la misma:



Observe que en algunas de estas operaciones el orden no es importante, por ejemplo el valor de " k_y " puede ser calculado antes que el de " k_x " (porque el valor de " k_x " no se emplea en " k_y " o viceversa) mientras que, en otras el orden es muy importante, por ejemplo antes de calcular $(1-y)_{im}$ es imprescindible calcular el valor de " y_i " (pues se emplea en dicho cálculo). En general, cuando tenemos expresiones secuenciales, como las de este ejemplo, los valores que se requieren para el cálculo de otros, deben ser evaluados primero.

Ahora creamos una nueva aplicación (File->New Application) y una nueva unidad (File->New Unit) y guardamos la aplicación en el directorio "\Secuencia\Ejemplo 2", con los nombres "fEjemplo2" para "Unit 1", "uEjemplo2" para "Unit 2" y "pEjemplo2" para "Project1".

En la unidad escribimos los tres módulos que resuelven el problema:

```
unit uEjemplo2;
```

interface

```

uses StdCtrls, SysUtils;

function LeerNumReal(e: TEdit): real;
function fxi(xi,x,y: real): real;
procedure MostrarNumReal(x: real; e: TEdit);

```

implementation

```

function LeerNumReal(e: TEdit): real;
begin
    result:= StrToFloat(e.Text);
end;

function fxi(xi,x,y: real): real;
const kpx = 0.001967; kpy = 0.001465;
var yi,kx,ky,xim,yim: real;
begin
    yi:= -0.0405+1.0378571428*xi;
    xim:= ((1-x)-(1-xi))/ln((1-x)/(1-xi));
    yim:= ((1-yi)-(1-y))/ln((1-yi)/(1-y));
    kx:= kpx/xim;
    ky:= kpy/yim;
    result:= kx/ky+(y-yi)/(x-xi);
end;

procedure MostrarNumReal(x: real; e: TEdit);
begin
    e.Text:= FloatToStr(x);
end;

```

end.

Como se puede observar en el módulo "fxi", "kpx" y "kpy" (equivalentes a k'_x y k'_y) han sido declarados como constantes y a $(1-x)_{im}$ y $(1-y)_{im}$ se les ha dado los nombres "xim" y "yim" respectivamente (esto porque los nombres de las variables no pueden contener operadores aritméticos).

Ahora en la forma colocamos 4 "Label", 4 "Edit" y un "Button", alineamos los mismos (con la paleta de alineación) y modificamos sus propiedades en el evento "onCreate":

```

procedure TForm1.FormCreate(Sender: TObject);
begin
    Form1.Caption:= 'EJEMPLO 2';
    Form1.Icon.LoadFromFile('Fish.ico');
    Form1.Brush.Bitmap:= TBitmap.Create;
    Form1.Brush.Bitmap.LoadFromFile('Lienzo.bmp');
    Form1.Position:= poScreenCenter;
    Label1.Transparent:= True;
    Label1.Alignment:= taRightJustify;
    Label1.Font.Color:= clRed;
    Label1.Font.Style:= [fsBold];
    Label1.Caption:= 'Valor de xi: ';
    Label2.Transparent:= True;
    Label2.Alignment:= taRightJustify;
    Label2.Font.Color:= clRed;
    Label2.Font.Style:= [fsBold];
    Label2.Caption:= 'Valor de x: ';

```

```

Label3.Transparent:= True;
Label3.Alignment:= taRightJustify;
Label3.Font.Color:= clRed;
Label3.Font.Style:= [fsBold];
Label3.Caption:= 'Valor de y: ';
Label4.Transparent:= True;
Label4.Alignment:= taRightJustify;
Label4.Font.Color:= clRed;
Label4.Font.Style:= [fsBold];
Label4.Caption:= 'Valor de la función: ';
Edit1.Color:= clBlue;
Edit1.Font.Color:= clYellow;
Edit1.Font.Style:= [fsBold];
Edit1.Text:= '0';
Edit2.Color:= clBlue;
Edit2.Font.Color:= clYellow;
Edit2.Font.Style:= [fsBold];
Edit2.Text:= '0';
Edit3.Color:= clBlue;
Edit3.Font.Color:= clYellow;
Edit3.Font.Style:= [fsBold];
Edit3.Text:= '0';
Edit4.Color:= clBlue;
Edit4.Font.Color:= clYellow;
Edit4.Font.Style:= [fsBold];
Edit4.Text:= '0';
Edit4.ReadOnly:= True;
Edit4.TabStop:= False;
Button1.Caption:= '&Calcular';
Button1.Default:= True;
Edit1.TabOrder:= 0;
Edit2.TabOrder:= 1;
Edit3.TabOrder:= 2;
Button1.TabOrder:= 3;
end;

```

Para que este código corra sin problemas, es necesario copiar el icono "Fish.ico" (u otro icono de su elección) y el bitmap "Lienzo.bmp" (u otro bitmap de su elección) en el directorio "\Secuencia\Ejemplo 2".

Además de las propiedades ya antes modificadas, en este módulo se ha modificado la propiedad "TabOrder" de los objetos "Edit1", "Edit2", "Edit3" y "Button1". La propiedad "TabOrder" determina el orden en que se mueve el control cuando se pulsa la tecla "Tab" (o "Shift+Tab" para el sentido contrario), así el primer objeto que recibe el control es el que tiene el "TabOrder", "0", el segundo el que tiene "1" y así sucesivamente (esta propiedad puede ser modificada visualmente con "Edit->Tab Order...").

Puesto que en este caso se crea un objeto de tipo "TBitmap", es necesario liberar el mismo, lo que hacemos en el evento "onClose":

```

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
    Form1.Brush.Bitmap.Free;
end;

```

El módulo principal se programa en el evento "onClick" del "Button1" (recordando previamente incluir la unidad "uEjemplo2": "File -> Use Unit -> uEjemplo2"):

```

procedure TForm1.Button1Click(Sender: TObject);

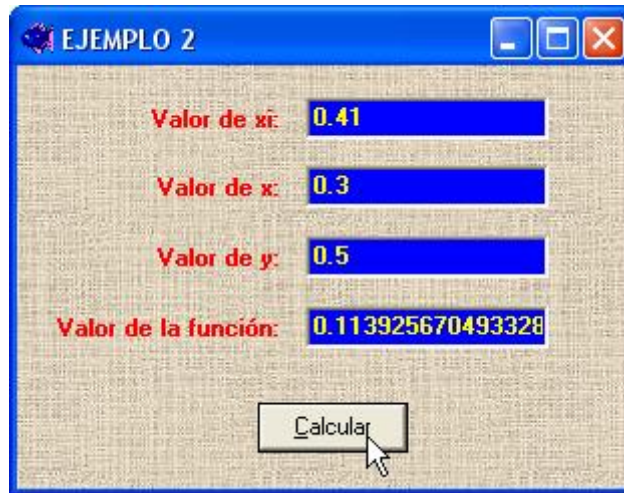
```

```

var xi,x,y,r: real;
begin
  xi:= LeerNumReal(Edit1);
  x:= LeerNumReal(Edit2);
  y:= LeerNumReal(Edit3);
  r:= fxi(xi,x,y);
  MostrarNumReal(r,Edit4);
  Edit1.SetFocus;
end;

```

No debe olvidar guardar frecuentemente los cambios hechos (Ctrl+S). Haciendo correr el programa se obtiene el resultado que se muestra en la siguiente figura (para los datos mostrados en la misma):



3.9 Ejercicios

1. Elabore una aplicación para calcular el valor de la siguiente expresión:

$$z = \left[\frac{\frac{xy^3 - 5x}{3x + 2y^2} + \ln \frac{x^3 + x \cdot y}{2x^2 y}}{\sqrt{\frac{e^{0.12x - 0.27y}}{4x^2 y + 3y^3}}} \right]^{-0.225}$$

Debe programar los módulos que resuelven el problema en una unidad separada (uEjercicio1), modificar las propiedades de los objetos en el evento "onCreate" de la forma y el módulo principal en el evento "onClick" del botón. La forma deberá tener su propio icono y su fondo deberá ser de color verde con líneas diagonales hacia adelante (Forward).

2. Elabore una aplicación que calcule el valor de la siguiente función:

$$\begin{aligned}
 f(x) &= x + w + y - 39 \\
 x^{1/2} + z^2 &= 35 \\
 y + e^{z/2} - \frac{8}{z} &= 30 \\
 w + z^2 - 2z &= 31
 \end{aligned}$$

Debe elaborar el diagrama de actividades del módulo que calcula el valor de la función, programar los módulos en una unidad separada (uEjercicio3), modificar las propiedades de los objetos en el evento "onCreate"

de la forma y programar el módulo principal en el evento "onClick" del botón. La forma deberá tener su propio icono, estar centrada y su fondo deberá ser del color "clInactiveCaption",

3. Elabore una aplicación que calcule el valor de la siguiente función:

$$f(x_1, L_o, x_o, V_o, y_o) = L_1 * x_1 + V_1 * y_1 - C_0$$

$$L_1 = \frac{L_c}{1 - x_1}$$

$$V_1 = M - L_1$$

$$y_1 = 1420 * x_1$$

$$L_c = L_0 * (1 - x_0)$$

$$M = L_0 + V_0$$

$$C_0 = L_0 * x_0 + V_0 * y_0$$

Debe elaborar el diagrama de actividades del módulo que calcule el valor de la función, programar los módulos en una unidad separada (uEjercicio3), modificar las propiedades de los objetos en el evento "onCreate" de la forma y programar el módulo principal en el evento "onClick" del botón. La forma deberá tener su propio icono, estar centrada en la pantalla, emplear como fondo una figura y los objetos deberán contar con un orden de tabulación coherente (pruebe la aplicación con $L_0=300$, $x_0=0$, $V_0=100$, $y_0=0.2$).

4. SECUENCIA: TIPOS DE DATOS

4.1 Tipos de datos simples en Delphi

Hasta ahora en todos los módulos elaborados hemos asumido que los datos numéricos son de tipo "real". Sin embargo, para aplicar con propiedad el tercer principio de la programación estructurada (emplear tipos de datos a la medida), debemos conocer los tipos de datos disponibles en Delphi.

4.1.1 Tipos enteros

Los tipos enteros, que deben ser elegidos cuando los datos con los que se trabajan pertenecen al conjunto de los números enteros son:

Tipo	Rango	Tamaño (bytes)
Shortint	-128..127	1
Smallint	-32768..32767	2
Longint	-2147483648..2147483647	4
Integer	-2147483648..2147483647	4
Int64	$-2^{63}..2^{63}-1$	8
Byte	0..255	1
Word	0..65535	2
Longword	0.. 4294967295	4
Cardinal	0.. 4294967295	4

Como se puede observar, el tipo genérico "Integer", en Delphi, es equivalente a "Longint" y el tipo genérico "Cardinal" es equivalente a "Longword". La elección del tipo de dato entero depende del problema que se esté resolviendo. Así si en la solución de un problema sólo se trabaja con números positivos, entonces quedan descartados los enteros que permiten valores negativos, por lo tanto se puede elegir entre "Byte", "Word" y "Longword" (o cardinal), si además los números con los que se trabaja pueden llegar hasta 10000, pero no superan este límite, entonces se descarta el tipo "Byte" (pues su valor máximo es 255), también se descarta el tipo "Longword", porque aunque cubre el intervalo de valores, en realidad es demasiado grande, pues permite valores superiores a los 4 mil millones, cuando el valor máximo a almacenar apenas alcanza los 10000. Por lo tanto, el tipo de dato más a la medida para este caso sería el tipo "Word".

4.1.2 Tipos reales

Los tipos reales, que se emplean cuando los datos o resultados de las operaciones tienen parte fraccionaria, son:

Tipo	Rango	Tamaño (bytes)	Precisión
Single	$1.5 \times 10^{-45}..3.4 \times 10^{38}$	4	7-8
Real48	$2.9 \times 10^{-39}..1.7 \times 10^{38}$	6	11-12
Double	$5.0 \times 10^{-324}..1.7 \times 10^{308}$	8	15-16
Real	$5.0 \times 10^{-324}..1.7 \times 10^{308}$	8	15-16
Extended	$3.6 \times 10^{-4951}..1.1 \times 10^{4932}$	10	19-20
Currency	-922337203685477.5808.. 922337203685477.5807	8	19-20
Comp	$-2^{63}+1..2^{63}-1$	8	19-20

La precisión se refiere al número de dígitos con que se realizan las operaciones y el primer número de la tabla corresponde a los números negativos (pues un dígito es empleado para el signo) y el segundo para los números positivos.

El tipo "Currency" (monetario), permite minimizar los errores de redondeo en operaciones monetarias al trabajar con 19 dígitos de precisión. Almacena sólo cuatro dígitos en la parte fraccionaria, lo que es suficiente cuando se trabaja con cantidades monetarias, pues por lo general sólo se conservan dos dígitos después del punto(hasta los centavos).

El tipo "Comp" (computacional), es en realidad un entero (equivalente a Int64), pero está clasificado como real porque con algunas funciones y operadores no se comporta como un entero, por ejemplo no se puede incrementar (con Inc) o decrementar (con Dec) una variable de tipo "Comp". Por ello se recomienda emplear Int64 en lugar de "Comp".

En Delphi para Win32, El tipo genérico "Real" es equivalente al tipo "Double".

4.1.3 Tipos carácter

Los tipos carácter, que se emplean cuando se trabaja con caracteres sueltos, tales como letras, números y otros símbolos simples, son:

Tipo	Tamaño (bytes)
AnsiChar	1
Char	1
WideChar	2

El tipo genérico "Char" es equivalente a "AnsiChar" y se emplea para representar caracteres del código ASCII. El tipo "WideChar" se emplea para trabajar con caracteres internacionales tales como los caracteres chinos, árabes y japoneses, los cuales requieren 2 bytes en lugar de 1. Se emplean para crear aplicaciones multilingües.

4.1.4 Tipos ordinales

Los tipos ordinales son aquellos que tienen un orden predefinido, por lo tanto, son ordinales los tipos enteros, carácter y lógicos (que estudiaremos luego). Sin embargo, en Delphi, podemos crear nuestros propios tipos ordinales. Por ejemplo si se está elaborando una aplicación donde se trabaja con los nombres de los meses del año y algunos colores, podemos crear dos tipos ordinales empleando la instrucción "type":

type

```
tMeses = (Enero,Febrero,Marzo,Abril,Mayo,Junio,Julio,Agosto,Septiembre,
          Octubre, Noviembre, Diciembre);
```

```
tColores = (rojo,amarillo,verde,azul,marron,celeste,naranja,purpura);
```

A este tipo de datos se conoce como "*tipos enumerados*", porque a cada uno de sus elementos se le asocia un número, comenzando con el cero, así "Enero" está asociado a "0", "Febrero" a "1", "Marzo" a "2" y así sucesivamente.

Una vez creado un tipo de dato se utiliza igual que cualquier otro tipo predefinido, así por ejemplo podemos declarar variables de estos tipos:

```
var mes: tMeses;
```

```
    color: tColores;
```

Y asignarle luego valores:

```
mes:= Marzo;
```

```
color:= verde;
```

Con los tipos ordinales se pueden emplear las siguientes funciones especializadas: "Ord(valor o expresión ordinal)" devuelve la posición (u orden)

del "valor o expresión ordinal", así por ejemplo `Ord(Abril)` devuelve "3", `Ord('A')` devuelve 65; "High(variable o tipo)" devuelve el valor más alto del tipo o del tipo de la variable, así `High(tMeses)` devuelve "Diciembre", `High(color)` devuelve "purpura", `High(Integer)` devuelve "2147483647"; "Low(variable o tipo)" devuelve el valor más bajo del tipo o del tipo de la variable, así `Low(color)` devuelve "rojo" y `Low(Integer)` devuelve "-2147483648", "Pred(valor o expresión ordinal)" devuelve el predecesor del valor, así `Pred(mes)` devuelve "Febrero", `Pred('B')` devuelve "A"; "Succ(valor o expresión ordinal)", devuelve el sucesor del valor, así `Succ('B')` devuelve "C", `Succ(Julio)` devuelve "Agosto".

Igualmente existen dos procedimientos que trabajan con valores ordinales: "Inc(variable)", incrementa en uno el valor de la variable ordinal e "Inc(variable,n)", incrementa en "n" el valor de la variable ordinal, así si "mes" es "Marzo", `Inc(mes)`, cambia el valor de la variable "mes" a "Abril" e `Inc(mes,4)`, cambia el valor de la variable "mes" (asumiendo que todavía sea Marzo) a "Julio". El procedimiento contrario es "Dec(variable)" y "Dec(variable,n)", que disminuyen el valor de la variable en 1 y en "n" respectivamente, así `Dec(mes)` devuelve "Febrero" y `Dec(mes,2)` devuelve "Enero".

Otro tipo ordinal que podemos definir es el "tipo subrango", así por ejemplo, si se está creando una aplicación para trabajar con los meses de Febrero a Octubre y con los días del 1 al 31, podemos definir los siguientes tipos:

type

```
tMesesLaborables = Febrero..Octubre;
tDiasMes = 1..31;
```

Al igual que con los tipos enumerados (y en general cualquier otro tipo creado por el usuario), los tipos subrango se emplean como cualquier tipo estándar de Delphi, así por ejemplo podemos declarar variables:

var

```
mesl: tMesesLaborables;
diasm: tDiasMes;
```

Y trabajar con estas variables asignándoles variables y empleando algunos procedimientos:

```
mesl:= Julio; dias:= 15;
Inc(mesl,2); Dec(diasm,3);
```

4.1.5 Tipos lógicos

Los tipos lógicos, que deben emplearse cuando los datos o resultados son falsos o verdaderos, son:

Tipo	Tamaño (bytes)
ByteBool	1
Boolean	1
WordBool	2
LongBool	4

Todos estos tipos sólo pueden tener uno de dos valores: *False* (Falso) o *True* (Verdadero). El tipo genérico: "Boolean", es el tipo que se prefiere al momento de declarar variables lógicas, los otros tipos existen para proveer compatibilidad con otros lenguajes y las librerías de los sistemas operativos.

"ByteBool", "WordBool" y "LongBool", se comportan de manera diferente al tipo genérico Boolean, así cuando se trabaja con el tipo "Boolean", el ordinal de "True" siempre es 1, sin embargo con los otros tipos, el ordinal de "True" es cualquier valor diferente de cero, por esta misma razón, con el tipo "Boolean" "False" es menor a "True", mientras que con los otros tipos "False" es diferente de "True" (pero no se puede determinar si es menor o mayor a "True"), igualmente con el tipo "Boolean" el predecesor de "True" es "False", pero con los otros tipos el predecesor y sucesor de "False" es "True" (pero el predecesor de "True" puede ser "True" o "False").

Con cualquier tipo lógico, el ordinal de "False" es cero y el sucesor de "False" "True" (pues es "True" cualquier valor diferente de cero).

4.1.6 Tipos cadena (String)

Aunque los tipos cadena no son realmente tipos simples sino compuestos (están compuestos por una "cadena" de caracteres), son estudiados y clasificados como tipos simples porque se cuenta con una amplia variedad de procedimientos, funciones y operadores que facilitan su manipulación.

Delphi cuenta con los siguientes tipos cadena:

Tipo	Máxima longitud	Memoria
ShortString	255 caracteres	2 a 256 bytes
AnsiString	$\sim 2^{31}$ caracteres	4 bytes a 2 GB
String	$\sim 2^{31}$ caracteres	4 bytes a 2 GB
WideString	$\sim 2^{30}$ caracteres	4 bytes a 2GB

Siendo el tipo AnsiString, también conocido como Long String, el tipo preferido para la mayoría de los propósitos. Por defecto el tipo genérico String es equivalente a AnsiString.

Si se cambia el interruptor de compilación "\$H" de encendido {\$H+}, que es su valor por defecto, a apagado {\$H-}, entonces el tipo String se hace equivalente a ShortString. Igualmente, si después de la palabra String, se especifica la longitud de la cadena, como por ejemplo en "String[20]", entonces el tipo genérico String se hace equivalente a ShortString.

El tipo ShortString, se conserva prácticamente sólo por razones de compatibilidad con las versiones anteriores de Delphi (y Pascal), donde era el único tipo de cadena disponible.

El tipo WideString, por otra parte, se emplea sobre todo para elaborar aplicaciones multilingües.

Como ya se dijo, aun cuando los tipos cadena no son simples, en algunas operaciones se comportan como si lo fueran, así por ejemplo la asignación de valores, la suma (concatenación) y, como veremos posteriormente, las operaciones relacionales, son directas, tal como se muestra en el siguiente segmento de código:

```
var s1,s2,s3: string;
begin
  s1:= 'El lenguaje de Delphi';
  s2:= 'Es Borland Pascal'
  s3:= s1+s2;
  ShowMessage(s3);
```

Por esta razón en la práctica trataremos a los "String" como tipos simples, sin embargo, no debemos olvidar que en realidad se tratan de vectores (arrays) de cadenas.

Como se dijo Pascal cuenta con una gran variedad de funciones que permiten manipular cadenas. Algunas de las más usuales son las siguientes:

Length(cadena)

Devuelve la longitud (el número de caracteres) de la cadena, así "Length('hola')" devuelve 4.

Copy(cadena, i, n)

Devuelve una subcadena, donde se han copiado "n" caracteres de la "cadena" comenzando desde la posición "i", siendo el primer carácter el número 1. Así Copy('Programación estructurada',14,12), devuelve "estructurada".

Delete(variable cadena, i, n)

Borra "n" caracteres del valor de la "variable cadena", comenzando con la posición "i", siendo el primer carácter el número 1. Así si "s='Delphi 7.0'", "Delete(s,7,4)" devuelve "Delphi".

Insert(subcadena, variable cadena, i)

Inserta la "subcadena" en el valor de la "variable cadena", comenzando con la posición "i", siendo el primer carácter el número 1. Así si "s='Programación en Pascal'", Insert(' Borland',s,16) devuelve "Programación en Borland Pascal".

LowerCase(cadena)

Convierte las letras de una cadena de mayúsculas a minúsculas (no toma en cuenta los acentos ni las eñes). Así LowerCase('Letras Mayúsculas'), devuelve "letras mayúsculas".

UpperCase(cadena)

Convierte las letras de una cadena de minúsculas a mayúsculas (no toma en cuenta los acentos ni las eñes). Así Uppercase('Letras Mayúsculas') devuelve "LETRA MINÚSCULAS".

AnsiLowerCase(cadena)

AnsiUpperCase(Cadena)

Hacen lo mismo que "LowerCase" y "UpperCase" respectivamente, pero toman en cuenta los acentos y las eñes, así "AnsiLowerCase('ÉXITO EN EL NUEVO AÑO') " devuelve "éxito en el nuevo año" y "AnsiUpperCase('éxito en el nuevo año') " devuelve "ÉXITO EN EL NUEVO AÑO".

Pos(subcadena, cadena)

Devuelve la posición que la que se encuentra la "subcadena" dentro de la "cadena", si la "subcadena" no se encuentra en la "cadena", "Pos" devuelve 0. Por ejemplo "Pos('Pascal','Borland Pascal') " devuelve "9" y "Pos('Delphi','Borland Pascal') " devuelve "0".

4.2 Ejemplos

1. Elabore una aplicación y en la misma cree una unidad con los siguientes módulos: "Modulo1", que reciba un número entero comprendido entre 0 y 200, incremente su valor en 50, reemplace el número por el cociente de su división entera entre 2, disminuya el resultado en 7 y devuelva el residuo del resultado entre 7; "Modulo2", que reciba dos números enteros comprendidos entre -5 millones y +5 millones, sume los dos números recibidos, calcule el cociente de la suma entre 5, incremente el resultado en 100, calcule el residuo del resultado entre 7 y devuelva ese resultado incrementado en 23. "Modulo3", que reciba tres números enteros com-

prendidos entre 0 y 3 mil millones, calcule la suma las raíces cuadradas de los tres números, calcule el logaritmo natural del resultado y devuelva el coseno de ese resultado.

Pruebe estos tres módulos en el evento "onClick" de la forma con 150 para "Modulo1", 330000 y 400000 para "Modulo2" y 5500000, 230000 y 332023400 para "Modulo3", mostrando los resultados en el tipo de letra "Arial", en color azul, con los estilos "Negrita y Cursiva", tamaño de letra 24 puntos, siendo el fondo de la forma de color blanco. La forma debe estar centrada en la pantalla y tener su propio icono.

Primero creamos una nueva aplicación (File->New Application) y una nueva unidad (File->New->Unit) y guardamos los archivos (File->Save All) con los nombres "fEjemplo3" para "Unit1", "uEjemplo3" para "Unit2" y "pEjemplo3" para "Project1" (como siempre en una carpeta: \Secuencia\Ejemplo 3).

En la unidad "uEjemplo3" escribimos el código de los tres módulos:

```
unit uEjemplo3;

interface

    function Modulo1(n: byte): byte;
    function Modulo2(n1,n2: LongInt): LongInt;
    function Modulo3(n1,n2,n3: Cardinal): double;

implementation

    function Modulo1(n: byte): byte;
    begin
        inc(n,50);
        n:= n div 2;
        dec(n,7);
        result:= n mod 7;
    end;

    function Modulo2(n1,n2: LongInt): LongInt;
    var s: LongInt;
    begin
        s:= n1+n2;
        s:= s div 5;
        inc(s,100);
        s:= s mod 7;
        inc(s,23);
        result:= s;
    end;

    function Modulo3(n1,n2,n3: Cardinal): double;
    var s: double;
    begin
        s:= sqrt(n1)+sqrt(n2)+sqrt(n3);
        s:= ln(s);
        result:= cos(s);
    end;

end.
```

Como se puede ver, el parámetro del "Modulo1" es de tipo "Byte", porque al estar el número comprendido entre 0 y 200, se encuentra dentro del intervalo de un Byte. Los parámetros del segundo módulo son de tipo "LongInt" (o Integer), porque los números pueden ser positivos o negativos y están comprendidos entre -5 millones y +5 millones, valor que supera el tipo inmediato inferior "SmallInt", pero que está por debajo del superior "Int64". Los parámetros del tercer módulo son de tipo "Cardinal" porque los números son positivos e inferiores a 4 mil millones, sin embargo, en este módulo el resultado es de tipo real "double", porque las operaciones que se llevan a cabo con los números son de tipo real, por lo tanto el resultado es de tipo real y, al no contar con más información, asumimos que dicho tipo es el real más genérico ("double" o lo que es lo mismo "real").

Ahora en la unidad de la forma (fEjemplo3) escribimos el código para probar los módulos (no olvidando incluir la unidad "uEjemplo3": File->Use Unit...->uEjemplo3).

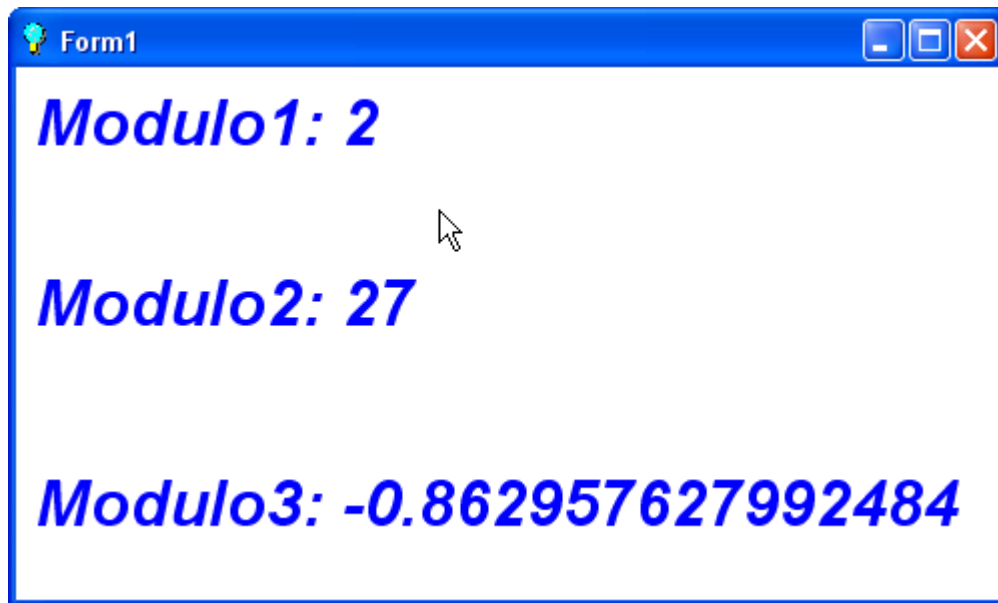
Primero programamos el evento "onCreate" para establecer las características de la forma así como las propiedades del tipo de letra:

```
procedure TForm1.FormCreate(Sender: TObject);  
begin  
    Form1.Position:= poScreenCenter;  
    Form1.Width:= 500;  
    Form1.Height:= 300;  
    Form1.Icon.LoadFromFile('Idea 3.ico');  
    Form1.Color:= clWhite;  
    Form1.Canvas.Font.Name:= 'Arial';  
    Form1.Canvas.Font.Color:= clBlue;  
    Form1.Canvas.Font.Style:= [fsBold,fsItalic];  
    Form1.Canvas.Font.Size:= 24;  
end;
```

Finalmente probamos los módulos programando el evento "onClick" de la forma:

```
procedure TForm1.FormClick(Sender: TObject);  
var r1: byte; r2: LongInt; r3: double;  
begin  
    Form1.Repaint;  
    r1:= Modulo1(150);  
    Form1.Canvas.TextOut(10,10,'Modulo1: '+IntToStr(r1));  
    r2:= Modulo2(330000,4000000);  
    Form1.Canvas.TextOut(10,100,'Modulo2: '+IntToStr(r2));  
    r3:= Modulo3(5500000,230000,332023400);  
    Form1.Canvas.TextOut(10,200,'Modulo3: '+FloatToStr(r3));  
end;
```

Aunque se puede establecer las propiedades de la fuente en el evento "onClick", en este caso es más eficiente establecerlas en el evento "onCreate", porque al ser constantes, no tiene sentido cambiarlas una y otra vez al activarse el evento "onClick" de la forma. Haciendo correr la aplicación, y haciendo click con el mouse, se obtiene:



Modulo1: 2

Modulo2: 27

Modulo3: -0.862957627992484

2. Elabore una aplicación y en la misma cree una unidad con tres módulos para calcular el valor de las siguientes funciones.

$$f(x) = x + y^2 + z^2 - 14.0$$

$$3x^{2.1} - 5y = 7.0$$

$$y^{1.2} + 4z = 14.3$$

$$f(V) = \frac{RT}{V-b} - \frac{\Omega_a}{\Omega_b} \frac{RTbF}{V(V+b)} - P$$

$$b = \frac{\Omega_a RT_c}{P_c}$$

$$\Omega_a = \left[9 \left(2^{1/3} - 1 \right) \right]^{-1}$$

$$\Omega_b = \frac{2^{1/3} - 1}{3}$$

$$F = \frac{1}{T_r} \left[1 + (0.480 + 1.574\omega - 0.176\omega^2)(1 - T_r^{0.5}) \right]^2$$

$$T_r = \frac{T}{T_c}$$

$$R = 0.08206$$

$$T = 320; T_c = 450; P_c = 24.3; P = 20; w = 0.045$$

$$f(T_2) = \frac{H_2 - H_1}{T_2 - T_1} + \frac{CS_1}{\lambda_2}$$

$$H_2 = \frac{HP_2 * HS_2}{100}$$

$$HS_2 = \frac{18.02}{28.97} * \frac{PV_2}{P - PV_2}$$

$$PV_2 = e^{\left(\frac{18.3036 - 3816.44}{T_2 + 227.02} \right)}$$

$$CS_1 = 1.005 + 1.88 * H_1$$

$$\lambda_2 = 2501.4 * \left(\frac{1 - \frac{T_2}{647.3}}{1 - 0.422} \right)^{0.375}$$

$$T_1 = 20$$

$$H_1 = 0.002$$

$$P = 540$$

$$HP_2 = 80$$

Se requiere que los valores de la primera función sean calculados con 18 dígitos de precisión. Se sabe que los valores de la segunda función están comprendidos entre 10^{-300} y 10^{300} y que no se requiere más de 14 dígitos de precisión. Se sabe que los valores de la tercera función están comprendidos entre 10^{-30} y 10^{30} y que no se requiere más de 11 dígitos de precisión.

Pruebe estos tres módulos en el evento "onContextPopUp" con "x=3.2" para la primera función, "V=2.4" para la segunda y "T₂=320" para la tercera. Los resultados deben ser mostrados con el tipo de letra "Times New Roman", en color rojo, 30 puntos, subrayado. La forma debe estar centrada

en la pantalla, su fondo debe ser la figura "Mármol blanco.bmp" y su icono "Ligth Bulb.ico".

Como de costumbre creamos una nueva aplicación (File->New Application), una nueva unidad (File->New->Unit) y guardamos los archivos en el directorio "\Secuencia\Ejemplo 4" con los nombres: "fEjemplo4" para "Unit1", "uEjemplo4" para "Unit2" y "pEjemplo4" para "Project1".

En la unidad "uEjemplo4" escribimos el código de las cuatro funciones:

```
unit uEjemplo4;
```

```
interface
```

```
    uses Math;
```

```
    function fx(x: extended):extended;
```

```
    function fv(v: double): double;
```

```
    function ft2(t2: real48): real48;
```

```
implementation
```

```
    function fx(x: extended):extended;
```

```
    var y,z: extended;
```

```
    begin
```

```
        y:=(3*Power(x,2.1)-7)/5;
```

```
        z:=(14.3-Power(y,1.2))/4;
```

```
        result:=x+sqr(y)+sqr(z)-14;
```

```
    end;
```

```
    function fv(v: double): double;
```

```
    const R=0.08206; T=320; Tc=450; Pc=24.3; P=20; w=0.045;
```

```
    var b,Ra,Rb,F,Tr: double;
```

```
    begin
```

```
        Tr:=T/Tc;
```

```
        F:=1/Tr*Sqr(1+(0.48+1.574*w-0.176*Sqr(w))*(1-Sqrt(Tr)));
```

```
        Rb:=(Power(2,1/3)-1)/3;
```

```
        Ra:=1/(9*(Power(2,1/3)-1));
```

```
        b:=Ra*R*Tc/Pc;
```

```
        result:=R*T/(v-b)-Ra/Rb*R*T*b*F/(v*(v+b))-P;
```

```
    end;
```

```
    function ft2(t2: real48): real48;
```

```
    const T1=20; H1=0.002; P=540; HP2=80;
```

```
    var H2,HS2,PV2,cs1,L2: real48;
```

```
    begin
```

```
        L2:=2501.4*Power((1-T2/647.3)/(1-0.422),0.375);
```

```
        cs1:=1.005+1.88*H1;
```

```
        PV2:=Exp(18.3036-3816.44/(T2+227.02));
```

```
        HS2:=18.02/28.97*PV2/(P-PV2);
```

```
        H2:=HP2*HS2/100;
```

```
        result:=(H2-H1)/(T2-T1)+cs1/L2;
```

```
    end;
```

```
end.
```

En esta unidad se ha incluido la unidad "Math" (uses Math) porque se hace uso de la función "Power" que se encuentra en la misma.

Para cada función se ha elegido el tipo de dato adecuado y se ha escrito el código de manera que se calculen primero los valores que se emplean luego en el cálculo de otros valores.

Luego, para probar los módulos, modificamos primero las propiedades de la forma en el evento "onCreate" de "Form1", como antes, cambiamos en este evento las propiedades de la fuente (Font) porque en realidad dichos valores son constantes.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Form1.Position:= poScreenCenter;
  Form1.Width:= 650;
  Form1.Height:= 300;
  Form1.Icon.LoadFromFile('Light Bulb.ico');
  Form1.Brush.Bitmap:= TBitmap.Create;
  Form1.Brush.Bitmap.LoadFromFile('Mármol blanco.bmp');
  Form1.Canvas.Font.Name:= 'Times New Roman';
  Form1.Canvas.Font.Color:= clRed;
  Form1.Canvas.Font.Size:= 30;
  Form1.Canvas.Font.Style:=[fsUnderline];
end;
```

Como se ha creado un objeto de tipo "TBitmap" es necesario liberarlo, lo que como de costumbre hacemos programando el evento "onClose":

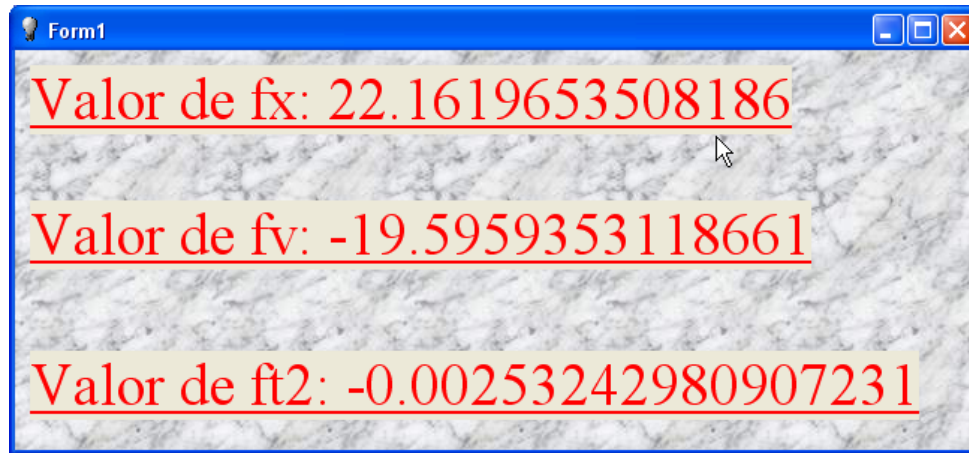
```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  Form1.Brush.Bitmap.Free;
end;
```

Ahora podemos probar los módulos de la unidad "uEjemplo4", llamándolos desde el evento "onContextPopUp", no olvidando incluir antes dicha unidad en la unidad fEjemplo4: File->Use Unit...->uEjemplo4.

```
procedure TForm1.FormContextPopup(Sender: TObject; MousePos: TPoint;
  var Handled: Boolean);
var r1: extended; r2: double; r3: real48;
begin
  r1:=fx(3.2);
  Form1.Canvas.TextOut(10,10,'Valor de fx: '+FloatToStr(r1));
  r2:=fv(60);
  Form1.Canvas.TextOut(10,100,'Valor de fv: '+FloatToStr(r2));
  r3:=ft2(200);
  Form1.Canvas.TextOut(10,200,'Valor de ft2: '+FloatToStr(r3));
end;
```

En esta unidad, una vez más, se han declarado las variables empleando tipos que concuerdan con los tipos de datos devueltos por los módulos, el no proceder de esta manera constituiría un error.

Haciendo correr la aplicación y haciendo click con el botón derecho del mouse se obtiene:



3. Elabore una aplicación y cree una unidad y en la misma defina los siguientes tipos: "tMeses" con los meses del año, "tNumMeses" con los números del 1 al 12, "tDias" con los días de la semana, "tNumDias" con los números del 1 al 7. Luego empleando los tipos definidos cree los siguientes módulos: "NumToMes" que reciba un número comprendido entre 1 y 12 y devuelva el mes equivalente, "MesToString" que reciba un mes y devuelva el texto correspondiente al número de mes recibido, "NumToDia" que reciba un número comprendido entre 1 y 7 y devuelva el día equivalente, "DiaToString" que reciba un día y devuelva el texto correspondiente al día recibido.

Pruebe los módulos elaborados en el evento "onDbClick" de la forma, mandando el número 6 a "NumToMes" y mostrando el mes devuelto con "MesToString"; mandando el número 3 a "NumToDia" y mostrando el día devuelto con "DiaToString". La forma debe estar centrada en la pantalla, tener el icono "Fish.ico", su fondo debe ser con líneas diagonales de 45° en color verde, la letra debe ser "Alba Super", de color negro, 30 puntos, cursiva.

Como de costumbre creamos una aplicación (File->New Application), una nueva unidad (File->New Unit) y guardamos los archivos en un directorio (\secuencia\ejemplo 5) con los nombres: "fEjemplo5" para "Unit1", "uEjemplo5" para "Unit2" y "pEjemplo5" para "Project1".

En la unidad "uEjemplo5" creamos los tipos y elaboramos los módulos requeridos:

```
unit uEjemplo5;
```

```
interface
```

```
  type
```

```
    tmeses = (enero, febrero, marzo, abril, mayo, junio, julio, agosto,
              septiembre, octubre, noviembre, diciembre);
    tNumMeses = 1..12;
    tdias = (domingo, lunes, martes, miercoles, jueves, viernes, sabado);
    tNumDias = 1..7;
```

```
  function NumToMes(n: tNumMeses): tmeses;
  function MesToString(m: tmeses): string;
  function NumToDia(n: tNumDias): tdias;
  function DiaToString(d: tDias): string;
```

```
implementation
```

```

function NumToMes(n: tNumMeses): tmeses;
begin
    result:= TMeses(n-1);
end;

function MesToString(m: tmeses): string;
const Meses: array [0..11] of string = ('enero','febrero','marzo','abril',
    'mayo','junio','julio','agosto','septiembre','octubre',
    'noviembre','diciembre');
begin
    result:= Meses[Ord(m)];
end;

function NumToDia(n: tNumDias): tdias;
begin
    result:= TDias(n-1);
end;

function DiaToString(d: tdias): string;
const dias : array [0..6] of string = ('domingo','lunes','martes',
    'miércoles','jueves','viernes','sábado');
begin
    result:= dias[Ord(d)];
end;

end.

```

En estos módulos, para transformar un número en un día o mes se ha empleado el moldeado de tipos. En el moldeado de tipos se emplea el nombre del tipo como una función que transforma el valor que se escribe como argumento en el tipo respectivo, así Char(65) transforma el número "65" en el carácter equivalente (en este caso la letra "A").

En los módulos "NumToMes" y "NumToDia", al número mandado se le resta uno pues, como se ha explicado, los tipos definidos por el usuario comienzan en cero y no en uno.

Para transformar un tipo definido por el usuario en un "string", en los módulos "MesToString" y "DiaToString", se ha creado dos vector de strings, con los nombres de los meses y días respectivamente. De esta forma, cuando se pide el elemento correspondiente a un día (o mes) se obtiene la cadena respectiva.

Para probar los módulos modificamos primero las propiedades de la forma (y de la fuente del lienzo) en el evento onCreate:

```

procedure TForm1.FormCreate(Sender: TObject);
begin
    Form1.Position:= poScreenCenter;
    Form1.Icon.LoadFromFile('Fish.ico');
    Form1.Width:= 500;
    Form1.Height:= 200;
    Form1.Brush.Color:= clGreen;
    Form1.Brush.Style:= bsBDiagonal;
    Form1.Canvas.Font.Name:= 'Àlba Super';
    Form1.Canvas.Font.Color:= clBlack;
    Form1.Canvas.Font.Size:= 30;
    Form1.Canvas.Font.Style:= [fsItalic];
end;

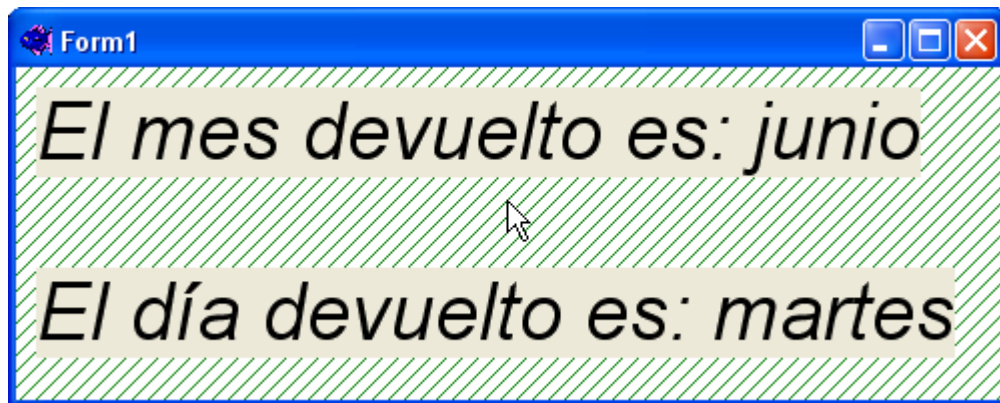
```

Finalmente probamos los módulos llamándolos desde el evento "onDbClick" de la forma:

```
procedure TForm1.FormDbClick(Sender: TObject);
var mes: tMeses; dia: tDias;
begin
    mes:= NumToMes(6);
    Form1.Canvas.TextOut(10,10,'El mes devuelto es: '+MesToString(mes));
    dia:= NumToDia(3);
    Form1.Canvas.TextOut(10,100,'El día devuelto es: '+DiaToString(dia));
end;
```

Igual que en los ejemplos anteriores, las variables locales de este módulo han sido declaradas empleando los tipos devueltos por las funciones, pues como se dijo, el no proceder de esta manera constituiría un error.

Ejecutando la aplicación y haciendo doble click en la forma se obtiene lo siguiente:



4.2.1 Ejercicios

1. Elabore una aplicación y en la misma cree una unidad con los siguientes módulos: "Modulo1", que reciba un número entero comprendido entre 0 y 50000, incremente su valor en 500, reemplace el número por el cociente de su división entera entre 3, disminuya el resultado en 70 y devuelva el residuo del resultado entre 7; "Modulo2", que reciba dos números enteros comprendidos entre -20000 y 20000, sume los dos números recibidos, calcule el cociente de la suma entre 4, incremente el resultado en 200, calcule el residuo del resultado entre 7 y devuelva ese resultado incrementado en 42. "Modulo3", que reciba tres números enteros comprendidos entre -100 y 100, calcule la suma las raíces cúbicas de los tres números, calcule el logaritmo natural del resultado, la raíz cuadrada del resultado y devuelva el coseno de ese resultado.

Pruebe estos tres módulos en el evento "onContextPopUp" de la forma con 5000 para "Modulo1", 3000 y 2000 para "Modulo2" y 55, 23 y 32 para "Modulo3", mostrando los resultados en el tipo de letra "Georgia", en color verde, con los estilos "cursiva y subrayado", tamaño de letra 30 puntos, siendo el fondo de la forma de color amarillo. La forma debe estar centrada en la pantalla y tener su propio icono.

2. Elabore una aplicación y en la misma cree una unidad con tres módulos para calcular el valor de las siguientes funciones.

$$\begin{aligned}
 f(k) &= \theta_c - \frac{T_0}{k} \left(1 - \cos \left(\sqrt{\frac{k}{I_0}} t \right) \right) & f(p) &= \cos(kl) \cosh(kl) + 1 & f(Y_{A1}) &= L_N + V_1 - M \\
 T_0 &= T_c + \theta_k k & k^2 &= p/a & Y_{B1} &= 2Y_{A1} + Y_{A1}^2 + 3.44Y_{A1}^3 \\
 \theta_c &= 2.97 & a^2 &= EIg/(Ay) & X_{BN} &= 2.2 + 5.1X_{AN} - 7.2X_{AN}^{2.2} \\
 \theta_k &= 3.27 & l &= 120 & L_N &= \frac{C_{A0}Y_{B1} - C_{B0}Y_{A1}}{X_{AN}Y_{B1} - X_{BN}Y_{A1}} \\
 T_c &= 0.625 & I &= 170.6 & V_1 &= \frac{C_{A0} - L_N X_{AN}}{Y_{A1}} \\
 t &= 0.0191 & E &= 3 \times 10^6 & X_{AN} &= 0.12; M = 32.1; \\
 I_0 &= 0.0006 & y &= 0.066 & C_{A0} &= 20.2; C_{B0} = 3.42 \\
 & & A &= 32 & & \\
 & & g &= 386 \text{ plg/s}^2 & &
 \end{aligned}$$

Se requiere que los valores de la primera función sean calculados con 11 dígitos de precisión. Se sabe que los valores de la segunda función están comprendidos entre 10^{-30} y 10^{30} y que no se requiere más de 8 dígitos de precisión. Se sabe que los valores de la tercera función están comprendidos entre 10^{-500} y 10^{500} y que se requiere un mínimo de 17 dígitos de precisión.

Pruebe estos tres módulos en el evento "onDb1Click" con "k=1.2" para la primera función, "p=4.5" para la segunda y "Y_A=0.25" para la tercera. Los resultados deben ser mostrados con el tipo de letra "Monotype Corsiva", en color azul, 25 puntos, subrayado y negrita. La forma debe estar centrada en la pantalla, su fondo debe ser la figura "Lienzo.bmp" y su icono "Bulb.ico".

- Elabore una aplicación y cree una unidad y en la misma defina los siguientes tipos: "tPaises" con diez países del continente, "tNumPaises" con los números del 1 al 10, "tDepartamentos" con los Departamentos de bolivia, "tNumDepartamentos" con los números del 1 al 9. Luego empleando los tipos definidos cree los siguientes módulos: "NumToPais" que reciba un número comprendido entre 1 y 10 y devuelva el país equivalente, "PaisToString" que reciba un país y devuelva el texto correspondiente al país recibido, "NumToDepartamento" que reciba un número comprendido entre 1 y 9 y devuelva el Departamento equivalente, "DepartamentoToString" que reciba un Departamento y devuelva el texto correspondiente al Departamento recibido.

Pruebe los módulos elaborados en el evento "onClick" de la forma, mandando el número 7 a "NumToPais", mostrando el país devuelto con "PaisToString"; mandando el número 5 a "NumToDepartamento", mostrando el Departamento devuelto con "DepartamentoToString". La forma debe estar centrada en la pantalla, tener el icono "Angel Fish.ico", su fondo debe ser con líneas diagonales de 135° en color azul, la letra debe ser "Century Gothic", de color rojo, 30 puntos, negrita y subrayado.

5. ESTRUCTURA IF-THEN-ELSE

Como ya se mencionó en el capítulo anterior, sólo algunos problemas triviales pueden ser resueltos empleando únicamente la secuencia. Los problemas reales requieren al menos una estructura selectiva y frecuentemente una o más estructuras iterativas.

Continuando con las estructuras estándar, en este capítulo comenzaremos el estudio de las estructuras selectivas, por lo tanto al concluir el estudio de estas estructuras deben estar capacitados para resolver problemas empleando las estructuras selectivas y siguiendo los principios de la programación estructurada.

El teorema de la programación estructurada señala que sólo se requiere una estructura selectiva, la estructura: *IF-THEN-ELSE*, sin embargo cuando la lógica involucra muchas condiciones consecutivas, el uso exclusivo de esta estructura da lugar a un algoritmo complejo, difíciles de entender y mantener, algo que va en contra del objetivo principal de la programación estructurada que es la de crear programas fáciles de entender y mantener. Es por esta razón que en este capítulo estudiaremos dos estructuras selectivas: la estructura *IF-THEN-ELSE* y la estructura *CASE*.

Antes de comenzar el estudio de estas estructuras repasaremos brevemente algunos conceptos.

5.1. Expresiones lógicas

Una *expresión lógica* o *condición* es aquella que devuelve un valor lógico. Un valor lógico sólo puede ser o falso (*false*) o verdadero (*true*).

Una expresión lógica se construye con operadores relacionales y/o operadores lógicos.

5.1.1. Operadores relacionales

Los operadores relacionales nos permiten comparar dos valores. En Pascal podemos emplear los siguientes operadores relacionales:

>	Mayor que
<	Menor que
>=	Mayor o igual que
<=	Menor o igual que
=	Igual a
!=	Diferente de

Los valores a comparar se escriben a ambos lados del operador. El resultado de un operador relacional es siempre o falso o verdadero, por ejemplo la expresión lógica:

Temperatura > 200

Devuelve verdadero (*true*) si la variable *Temperatura* contiene un valor mayor a 200 y falso (*false*) en caso contrario. La expresión:

Nombre = 'Carlos'

Devuelve verdadero si la variable *Nombre* contiene la cadena '*Carlos*', caso contrario devuelve falso. Y la expresión:

Base <= Altura

Devuelve verdadero si la variable *Base* tiene un valor menor o igual al de la variable *Altura*, caso contrario devuelve falso.

Al emplear operadores relacionales se debe tener cuidado de comparar tipos de datos compatibles, así se pueden comparar números entre sí, cadenas y caracteres entre sí, pero no números con una cadenas (tercer principio de la programación estructurada).

Cuando a ambos lados del operador relacional se tienen expresiones matemáticas, las mismas son evaluadas previamente y los resultados comparados. Así la siguiente expresión:

$$X-15+\ln(Z) > Y*12$$

Devuelve verdadero si el resultado de $X-15+\ln(Z)$ es mayor al resultado de $Y*12$, caso contrario devuelve falso. Se debe aclarar que esto es válido en *Pascal*, pero no en todos los lenguajes, muchos lenguajes requieren paréntesis para que las expresiones sean evaluadas antes de realizar la comparación. En general se recomienda emplear paréntesis para construir expresiones más universales y evitar ambigüedades, así la anterior expresión resulta más clara si se escribe en la siguiente forma:

$$(X-15+\ln(Z)) > (Y*12)$$

5.1.2. Operadores lógicos

Los operadores lógicos, como su nombre sugiere, sólo trabajan con valores lógicos (falso o verdadero). Puesto que los operadores relacionales devuelven valores lógicos, pueden emplearse en combinación con los operadores lógicos para construir expresiones lógicas más ricas.

En Pascal contamos con los siguientes operadores lógicos:

not	Negación lógica
and	Y lógico
Or	O lógico
xor	O excluyente

El operador *not* cambia un valor lógico de falso (*false*) a verdadero (*true*) o viceversa. El operador *and* requiere dos valores lógicos y devuelve verdadero si ambos valores son verdaderos y falso en cualquier otro caso. El operador *or*, requiere igualmente dos valores lógicos y devuelve falso si ambos valores son falsos y verdadero en cualquier otro caso. El operador *xor*, requiere también dos valores lógicos y devuelve falso si ambos valores son iguales y verdadero en cualquier otro caso.

Todas las posibles combinaciones de estos operadores se presentan en la siguiente tabla (*A* y *B* son variables lógicas):

A	B	Not A	A and B	A or B	A xor B
False	False	True	False	False	False
False	True	True	False	True	True
True	False	False	False	True	True
True	True	False	True	True	False

Algunos ejemplos de expresiones lógicas que incluyen tanto operadores relacionales como lógicos son:

$$(Edad>12) \text{ And } (Mes=12)$$


```
(Nombre='Ana') Or (Apellido='Poppe') Or (Sexo='F')
((Mes=11) And (Ciudad='Sucre')) Or (Item=104)
```

En *Pascal*, los operadores lógicos tienen una prioridad más alta que los operadores relacionales, es decir se evalúan primero, razón por la cual es necesario (no opcional) emplear paréntesis. El orden de prioridades de los operadores en *Pascal*, incluyendo los operadores aritméticos y algunos a nivel de bits, es el siguiente:

<i>Operador</i>	<i>Prioridad</i>
()	0 (más alta)
Not, - (como cambio de signo)	1
*, /, Div, Mod, And, Shl, Shr	2
+, - , Or, Xor	3
<, <=, = <>, >=, >, In	4 (más baja)

Otro aspecto que es importante tomar en cuenta al momento de trabajar con expresiones lógicas en *Delphi* y en otras versiones de *Pascal*, es que por defecto las expresiones se evalúan en *corto circuito*: Si al evaluar parte de una expresión lógica, se puede asegurar cual será el resultado, sin necesidad de evaluar el resto de la expresión, entonces se devuelve directamente dicho resultado. Por ejemplo en la expresión:

```
(a=b) and (c>d)
```

Si el resultado de la primera operación ($a=b$) es falso, se sabe con seguridad que el resultado de toda la expresión es falso, porque el operador *and* sólo devuelve verdadero si los dos valores son verdaderos, con la evaluación en *corto circuito* *Delphi* no evalúa la segunda operación ($c>d$) y devuelve directamente el resultado *falso*. Esta característica acelera la ejecución de los programas y puede ser empleada para resolver más eficientemente algunos problemas.

5.2. Indentación

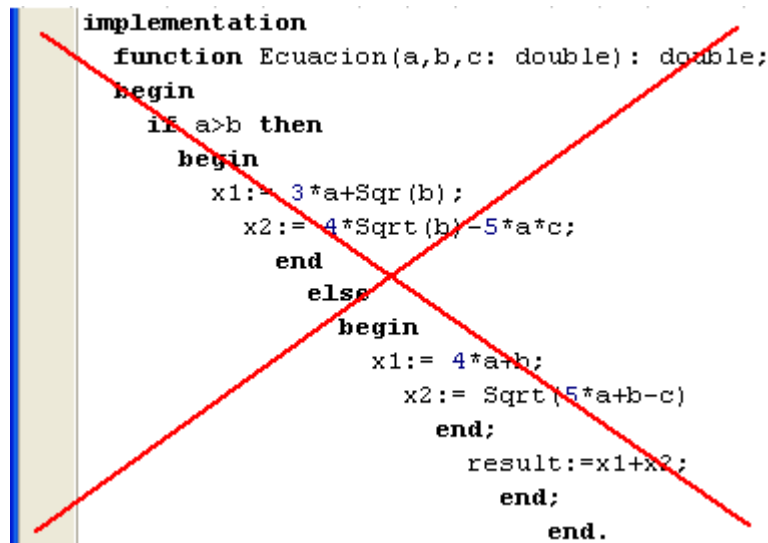
Antes de pasar a analizar las estructuras selectivas, comentemos brevemente sobre una práctica que se debe seguir cuando se escribe código: la "indentación":



```
implementation
function Ecuacion(a,b,c: double): double;
begin
  if a>b then
  begin
    x1:= 3*a+Sqr(b);
    x2:= 4*Sqrt(b)-5*a*c;
  end
  else
  begin
    x1:= 4*a+b;
    x2:= Sqrt(5*a+b-c)
  end;
  result:=x1+x2;
end;
end.
```

Como se puede observar en la anterior figura, "indentar" consiste en escribir el código que pertenece a una estructura un poco más a la derecha (con relación al margen izquierdo), de esta manera es mucho más sencillo identificar las estructuras y corregir errores. Debe quedar claro que sólo se escribe más a la derecha el código que pertenece a una estructura, dentro de una estructura, las instrucciones que se ejecutan de manera secuencial se escriben al mismo nivel, como sucede con `x1:=...` y `x2:=...` e igualmente con `if` y `result:=...` en la anterior figura.

Indentar no consiste en escribir cada línea de código más y más a la derecha, hasta que ya no quede más espacio:



```

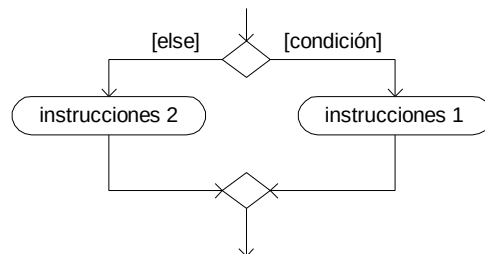
implementation
function Ecuacion(a,b,c: double): double;
begin
  if a>b then
    begin
      x1:= 3*a+Sqr(b);
      x2:= 4*Sqrt(h)-5*a*c;
    end
  else
    begin
      x1:= 4*a+b;
      x2:= Sqrt(5*a+b-c);
    end;
    result:=x1+x2;
  end;
end.

```

El indentar código no tiene ningún efecto en el rendimiento del programa, es decir no hace que el mismo sea más o menos eficiente, pero si en la facilidad con la que se pueden identificar las estructuras y corregir los errores. Puesto que el principal propósito de la programación estructurada es el facilitar el mantenimiento del código, y la indentación contribuye claramente a ese fin, es una práctica que seguiremos en la elaboración de programas estructurados. De hecho Delphi ayuda a este propósito y por defecto indenta el código a medida que se escribe.

5.3. Estructura IF-THEN-ELSE

La estructura *IF-THEN-ELSE* tiene el siguiente diagrama de actividades:



Como se puede ver si la *condición* es verdadera se ejecutan las "instrucciones 1", caso contrario se ejecutan las "instrucciones 2". La forma de implementar esta estructura en *Pascal* es la siguiente:

```
if condición then instrucciones 1 else instrucciones 2;
```

Si las "instrucciones 1" o las "instrucciones 2" constan de más de una instrucción (es decir son secuencias) deben ser encerradas entre "begin" y "end". La acción por defecto "else" es opcional.

Las "instrucciones" pueden ser también otras estructuras estándar, incluida otra estructura *IF-THEN-ELSE* en cuyo caso se dice que las estructuras están *anidadas*.

El caso contrario "else" de esta estructura es opcional, es decir puede no ser escrita, en cuyo caso si la condición es falsa, la instrucción "if" no hace nada y el control pasa directamente a la instrucción que se encuentre después del "if".

Un detalle que se debe tomar en cuenta es el de no escribir punto y coma (;) antes de la instrucción "else", esto debido a que el punto y coma señala el fin de una instrucción en Pascal, por lo tanto con el punto y coma le estaríamos indicando a Pascal que la instrucción termina antes del "else", cuando en realidad termina después de este.

5.4. Control de errores en la introducción de datos (validación)

Algo que se debe tener muy en cuenta cuando se elaboran aplicaciones es que los usuarios siempre cometen errores. No se debe asumir que un usuario introducirá los datos correctos, en el orden correcto y que pulsará los botones correctos (el usuario perfecto), por el contrario siempre se debe asumir que el usuario hará las cosas de forma incorrecta, porque en el mundo real no existe el usuario perfecto.

Por ejemplo, si se pide a un usuario que introduzca un número, para calcular la raíz cuadrada, con seguridad que existirán usuarios que escribirán un texto, por ejemplo "hola", en lugar del número. Por ello se deben tomar todas las medidas posibles a fin de impedir que el usuario cometa errores. A partir de las aplicaciones de este capítulo comenzaremos a controlar los tipos de datos que un usuario puede escribir en un objeto a fin de evitar que ocurra el ejemplo dado.

Para ello trabajaremos en el evento "OnKeyPress" de los objetos que reciben los datos (hasta ahora sólo empleamos para este fin los *Edits*). El evento "OnKeyPress" se activa cuando el usuario pulsa alguna de las teclas del *teclado principal* de la computadora.

Es oportuno aclarar que para fines de programación el teclado de una computadora se divide en dos: a) "El teclado principal" que es el sector donde se encuentran las letras, números, barra de espacio, tabulador (Tab), aceptar (Enter), etc. y b) "El teclado extendido" que está conformado por todas las teclas que no pertenecen al teclado principal, tales como las teclas de funciones, cursores, inicio (Home), insertar (Insert), suprimir (Delete), etc.

El evento "OnKeyPress" sólo responde al teclado principal. Un evento más general, que responde a la pulsación de cualquier tecla o combinación de teclas, es el evento "OnKeyDown". Como por el momento nos interesa controlar los datos que se introducen en forma de texto o número, y todas las teclas que se emplean para este fin están en el teclado principal, programaremos el evento "OnKeyPress" y no el evento "OnKeyDown".

El controlar los datos que el usuario puede escribir (o elegir) se conoce también como "validación" y con ello se reduce la probabilidad de error, pero no se la elimina, aún así el usuario puede cometer errores, como escribir "4.5.34.3.23.4" cuando se le pide un número real. Como veremos luego se deben tomar otras medidas adicionales para reducir aún más la probabilidad de error.

El módulo que responde al evento `onKeyPress` tiene la siguiente estructura:

```
procedure TNombre_de_la_Forma.Nombre_del_ObjetoKeyPress(Sender: TObject; var
Key: Char);
begin
end;
```

Como ocurre con todos los eventos de Delphi, el primer parámetro es "Sender" que es el objeto que está respondiendo al evento (no emplearemos este parámetro por el momento). El segundo parámetro "Key", es el carácter correspondiente a la tecla pulsada y es el parámetro que emplearemos para controlar las teclas pulsadas, así si se ha pulsado la tecla "1", "Key" tendrá el valor '1', si se ha pulsado la tecla "A", "Key" tendrá el valor 'A' o 'a' (si se está escribiendo en minúsculas).

Algunas teclas como *Aceptar (Enter)*, *Retroceso (BackSpace)*, *Tab* y *Escape*, no tienen un símbolo asignado en el código ASCII, por lo que se identifican por su número de código ASCII precedidas del símbolo "#". Así la tecla *Enter* es: #13, la tecla *BackSpace*: #8, la tecla *Tab*: #9, la tecla *Escape*: #27, etc. Esta notación también es válida para cualquier otro carácter, así en lugar de 'A' se puede escribir #65.

Como primer ejemplo veamos cual sería el evento "onKeyPress" del edit "eNumero" en la forma "MiForma" si sólo se permite la introducción de números enteros positivos:

```
procedure TMiForma.eNumeroKeyPress(Sender: TObject; var Key: Char);
begin
  if not ((Key=#8) or ((Key>='0') and (Key<='9')) then
    begin
      Beep; Abort;
    end;
end;
```

Donde si la tecla no es "BackSpace" (#8) o está comprendida entre '0' y '9', se hace sonar la alarma de la computadora con "Beep" y se aborta el evento con el comando "Abort", el cual termina el evento y es como si el mismo no hubiese ocurrido.

Como otro ejemplo veamos como sería el evento "onKeyPress" del edit "eReal", de la forma "MiForma", cuando se permite la escritura de números reales:

```
procedure TMiForma.eRealKeyPress(Sender: TObject; var Key: Char);
begin
  if not ((Key=#8) or ((Key>='0') and (Key<='9')) or (Key='.') or (Key='E')
    or (Key='e') or (Key='+') or (Key='-')) then
    begin
      Beep; Abort;
    end;
end;
```

Donde si la tecla no es "BackSpace" (#8), un número, un punto, el signo más, el signo menos o las letras "e" o "E" (para notación científica) se hace sonar la alarma de la computadora (Beep) y se cancela evento (Abort).

Como tercer ejemplo veamos como sería el evento `onKeyPress` del edit "eNombre" de la forma "MiForma", cuando se debe permitir la escritura de letras mayúsculas y el espacio.

```
procedure TMiForma.eNombreKeyPress(Sender: TObject; var Key: Char);
begin
```

```

if not ((Key=#8) or((Key>='A') and (Key<='Z')) or (Key='Ñ')) then
begin
    Beep; Abort;
end;
end;

```

Donde si la tecla no es "BackSpace" (#8), una letra mayúscula o la letra "Ñ", se hace sonar la alarma (Beep) y se cancela el evento (Abort).

La condición de la estructura "if-then-else" puede hacerse más simple y clara si se trabaja con conjuntos (o si se trabaja la estructura "Case" como veremos luego). Para ello sin embargo es necesario que conozcamos algunos conceptos con relación a los conjuntos. Estudiaremos los conjuntos con mayor detalle en un capítulo posterior, pero dado que los mismos forman parte de varias propiedades (como "Style") y son de utilidad en varios eventos (como "onKeyPress"), estudiaremos brevemente algunos conceptos básicos con relación a los mismos.

5.5. Conjuntos, conceptos básicos

Para crear un conjunto en Pascal, se escriben sus elementos separados por comas y entre corchetes. Por ejemplo en las siguientes instrucciones se definen algunos conjuntos:

```

const C1 = [1,3,7,9..20,50];
        C2 = ['A','E','I','O','U'];
        C3 = ['C','F','O'..'Z'];

```

El primer conjunto "C1" está conformado por 1, 3, 7, los números del 9 al 20 y el número 50. El segundo conjunto "C2" está conformado por las vocales en mayúsculas. El tercer conjunto "C3" está conformado por las letras "C", "F" y las letras comprendidas entre "O" y "Z".

Dos de las principales limitantes de los conjuntos en Pascal son: a) Los elementos de un conjunto sólo pueden ser valores ordinales, b) Un conjunto puede tener como máximo 256 elementos.

El operador de uso más frecuente cuando se trabaja con conjuntos es el operador de pertenencia: **in**. Este operador verifica si un elemento está en un conjunto, devolviendo verdadero en caso afirmativo y falso en caso contrario, así: "'V' in ['A'..'Z']", devuelve verdadero porque la letra "V" está en el conjunto, mientras que: '2' in ['A'..'Z'] devuelve falso porque el carácter '2' no se encuentra en el conjunto.

Otros dos operadores de uso más o menos frecuente son el de unión (+) y el de diferencia (-). El operador de unión **(+)** devuelve un conjunto formado por todos los elementos de los conjuntos que une, así: [1, 3, 5] + [2, 4, 6] devuelve el conjunto [1..6], es decir un conjunto formado por todos los números comprendidos entre 1 y 6.

Una característica que se debe tomar muy en cuenta cuando se trabaja con conjuntos es que los mismos no tienen elementos repetidos, así [1, 3, 5] + [1, 3, 7] devuelve el conjunto: [1, 3, 5, 7] y no como se podría pensar: [1, 1, 3, 3, 5, 7].

El operador de diferencia **(-)** devuelve un conjunto formado por todos los elementos que existen en el primer conjunto pero que no existen en el segundo, así: [1, 3, 5, 7] - [2, 3, 7] devuelve el conjunto: [1, 5].

Ahora que sabemos algo con relación a los conjuntos volvamos a escribir los eventos "onKeyPress" de los edit "eNumero", "eReal" y "eNombre" de la forma "MiForma":

```

procedure TMiForma.eNumeroKeyPress(Sender: TObject; var Key: Char);
begin
  if not (Key in [#8,'0'..'9']) then
    begin
      Beep; Abort;
    end;
end;
procedure TMiForma.eRealKeyPress(Sender: TObject; var Key: Char);
begin
  if not (Key in [#8, '0'..'9','.', 'E','e','+', '-']) then
    begin
      Beep; Abort;
    end;
end;
procedure TMiForma.eNombreKeyPress(Sender: TObject; var Key: Char);
begin
  if not (Key in [#8,'A'..'Z','Ñ']) then
    begin
      Beep; Abort;
    end;
end;

```

Como se puede observar, al utilizar conjuntos, las condiciones se simplifican considerablemente y son más fáciles de entender.

3.6. Ejemplos

1. Resolución de una ecuación cuadrática

Como primer ejemplo elaboraremos una aplicación para resolver la ecuación cuadrática: $ax^2+bx+c=0$.

Para ello creamos primero una nueva aplicación (File -> New Application), una nueva unidad (File -> New Unit) y guardamos el proyecto (File -> Save Project As...) con los nombres: "ufCuadratica" para "Unit1", "uCuadratica" para "Unit2", y "pCuadratica" para "Project1" en el directorio "Seleccion\Cuadratica\".

Como siempre, comenzamos a resolver el problema identificando primero los módulos en los que se puede dividir el problema: a) Leer los coeficientes, b) Resolver la ecuación cuadrática y c) Mostrar los resultados calculados.

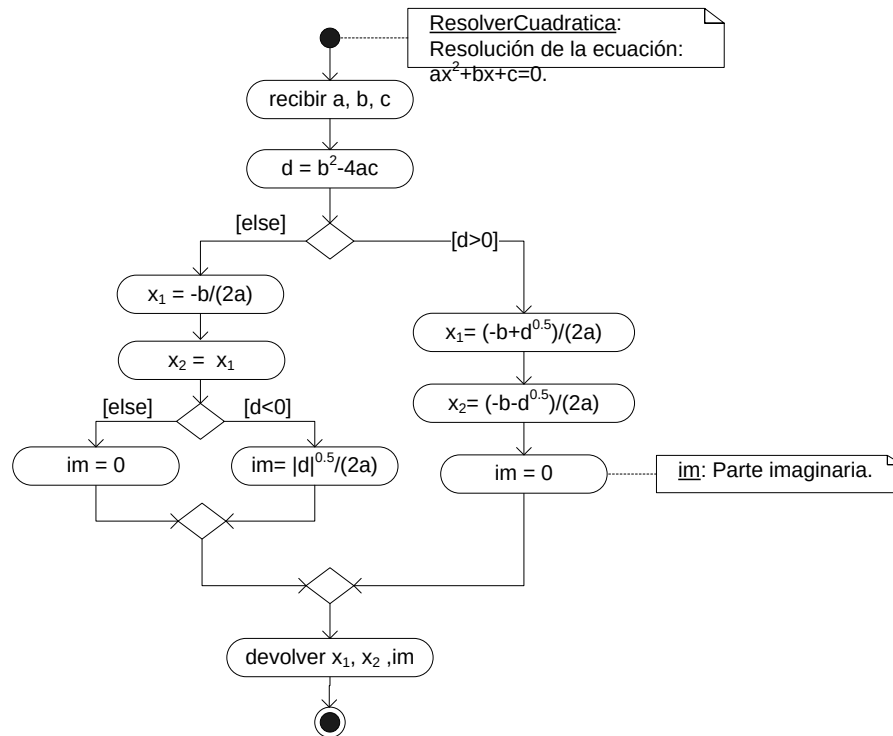
Los algoritmos de los módulos relacionados con la lectura y presentación de resultados son relativamente sencillos, razón por la cual serán codificados directamente.

Vayamos entonces a resolver el problema principal: *encontrar las soluciones de la ecuación cuadrática*. Para ello debemos recordar que soluciones de la ecuación cuadrática se encuentran con la expresión:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Donde, dependiendo del valor del discriminante: " b^2-4ac ", se pueden presentar uno de los siguientes casos: a) si el discriminante es positivo las soluciones son reales y distintas; b) si el discriminante es igual a cero las dos soluciones son reales e iguales y c) si el discriminante es negativo las dos soluciones son complejas (tienen parte imaginaria).

El algoritmo elaborado tomando en cuenta los tres casos se presenta en el siguiente diagrama de actividades:



Como se puede observar se han empleado dos estructuras *IF-THEN-ELSE* (una anidada). Puesto que el módulo devuelve 3 resultados: " x_1 ", " x_2 " e " im ", debe ser implementado como un procedimiento, con parámetros de salida (out) y empleando tipos reales porque tanto los datos como los resultados pueden tener parte fraccionaria. De estos tres resultados, " x_1 " y " x_2 " son los valores correspondientes a la parte real de las soluciones e " im " es el valor de la parte imaginaria, siendo las soluciones el complejo conjugado:

$$\begin{aligned} r_1 &= x_1 + im \, i \\ r_2 &= x_2 - im \, i \end{aligned}$$

Donde " i " representa a la raíz cuadrada de -1 . En el módulo que muestra los resultados se ha tomado en cuenta que cuando " im " es cero las soluciones son reales (no complejas), por lo tanto sólo se muestra la parte imaginaria cuando " im " es diferente de cero.

El código del módulo principal y de los módulos de lectura y presentación de resultados es el siguiente:

```
unit uCuadratica;
```

```
interface
```

```
uses StdCtrls, SysUtils;
```

```
procedure LeerCoeficientes(ea,eb,ec: tEdit; out a,b,c: double);
```

```
procedure ResolverCuadratica(a,b,c: double; out x1,x2,im: double);
```

```
procedure MostrarRaices(ex1,ex2: tEdit; x1,x2,im: double);
```

```
implementation
```

```
procedure LeerCoeficientes(ea,eb,ec: tEdit; out a,b,c: double);
```

```
begin
```

```
  a:= StrToFloat(ea.Text);
```

```
  b:= StrToFloat(eb.Text);
```

```
  c:= StrToFloat(ec.Text);
```

```
end;
```

```

procedure ResolverCuadratica(a,b,c: double; out x1,x2,im: double);
var d: double;
begin
  d:= Sqr(b)-4*a*c;
  if d>0 then
    begin
      x1:= (-b+Sqrt(d))/(2*a);
      x2:= (-b-Sqrt(d))/(2*a);
      im:= 0;
    end
  else
    begin
      x1:= -b/(2*a);
      x2:= x1;
      if d<0 then im:= Sqrt(Abs(d))/(2*a) else im:=0;
    end;
  end;

procedure MostrarRaices(ex1,ex2: tEdit; x1,x2,im: double);
begin
  if im=0 then
    begin
      ex1.Text:= FloatToStr(x1);
      ex2.Text:= FloatToStr(x2);
    end
  else
    begin
      ex1.Text:= FloatToStr(x1)+'+'+FloatToStr(im)+'i';
      ex2.Text:= FloatToStr(x1)+'-'+FloatToStr(im)+'i';
    end;
  end;

end.

```

Una vez escrito el código que resuelve el problema, compilamos la unidad (Ctrl+F9), corregimos los errores y guardamos los cambios (Ctrl+S). Ahora para probar y emplear la solución elaborada debemos trabajar en la interfaz de la aplicación y para ellos debemos ir primero a la forma (Shift+F12 -> Form1).

En esta aplicación emplearemos los objetos que ya hemos utilizado en aplicaciones anteriores, por lo que no se requiere mayor explicación con relación a los mismos.

Las propiedades modificadas, que pueden ser modificadas en el inspector de objetos o mediante código (generalmente en el evento "onCreate" de la forma) son las siguientes:

Form1: Name = 'fCuadratica', Caption = 'Ecuación cuadrática: $a \cdot x^2 + b \cdot x + c$ ', Width = 334, Height = 280, Position = poScreenCenter.

Label1: Caption='Coeficientes de la ecuación cuadrática:'; Label2: Caption='a:'; Label3: Caption='b:'; Label4: Caption='c:'. Todas estas etiquetas tienen las siguientes propiedades comunes: Transparent = True, Font.Color = clSkyBlue, Font.Style = [fsBold]. Recuerde que para asignar el mismo valor a dos o más objetos a la vez, simplemente debe seleccionar los objetos y cambiar el valor de la o las propiedades comunes.

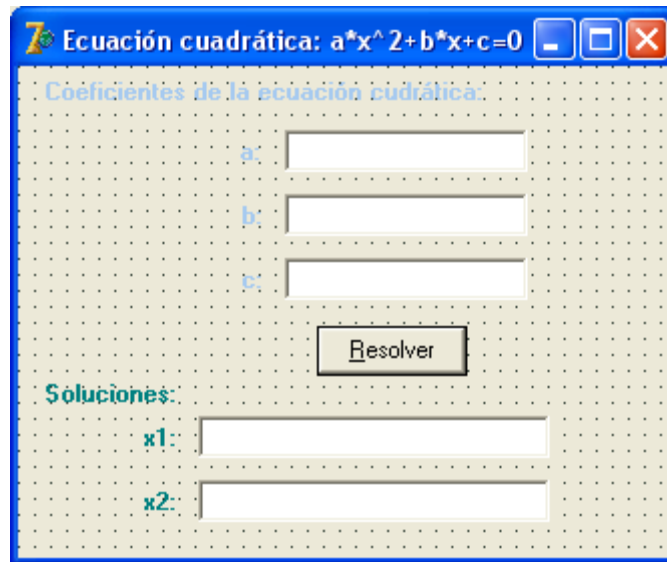
Label5: Caption='Soluciones:'; Label6='x1:'; Label7='x2'. Siendo las propiedades comunes: Transparent=True, Font.Color=clTeal, Font.Style=[fsBold].

Edit1: Name = 'ea'; Edit2: Name = 'eb'; Edit3: 'Name' = ec. Propiedades comunes: Text = '', Font.Color = clBlue.

Edit4: Name='ex1'; Edit5: Name='ex2'. Propiedades comunes: Text='', Font.Color = clGreen, Font.Style=[fsBold], ReadOnly=True, TabStop=False.

Button1: Name = 'bResolver', Caption = '&Resolver', Default = True.

Coloque los componentes en la forma, cambie sus propiedades de acuerdo a lo especificado anteriormente, ordénelos (con la paleta de alineación) y cambie sus tamaños de manera que queden aproximadamente como se muestra en la siguiente figura:



Recuerde guardar las modificaciones hechas frecuentemente (File -> Save o File -> Save All).

En esta aplicación la forma tendrá la figura "plumas.bmp", para lo cual es necesario programar los eventos "onCreate" y "onClose":

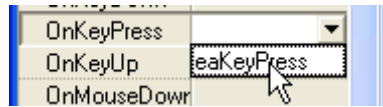
```
procedure TfCuadratica.FormCreate(Sender: TObject);
begin
    fCuadratica.Brush.Bitmap:= tBitmap.Create;
    fCuadratica.Brush.Bitmap.LoadFromFile('C:\Windows\plumas.bmp');
end;
```

```
procedure TfCuadratica.FormClose(Sender: TObject;
    var Action: TCloseAction);
begin
    fCuadratica.Brush.Bitmap.Free;
end;
```

También se validará la introducción de datos de manera que los Edits "ea", "eb" y "ec" sólo permitan introducir caracteres correspondientes a los números reales, para ello se debe programar el evento "onKeyPress" de "ea":

```
procedure TfCuadratica.eaKeyPress(Sender: TObject; var Key: Char);
begin
    if not (Key in [#8,'0'..'9','.',',','+','-','e','E']) then
        begin
            Beep; Abort;
        end;
end;
```

Con ello queda validada la introducción de datos a "ea". Puesto que los otros dos edits "eb" y "ec" tendrán la misma validación, lo que se hace es seleccionar los objetos y elegir el módulo recién programado (eaKeyPress) en el inspector de objetos:

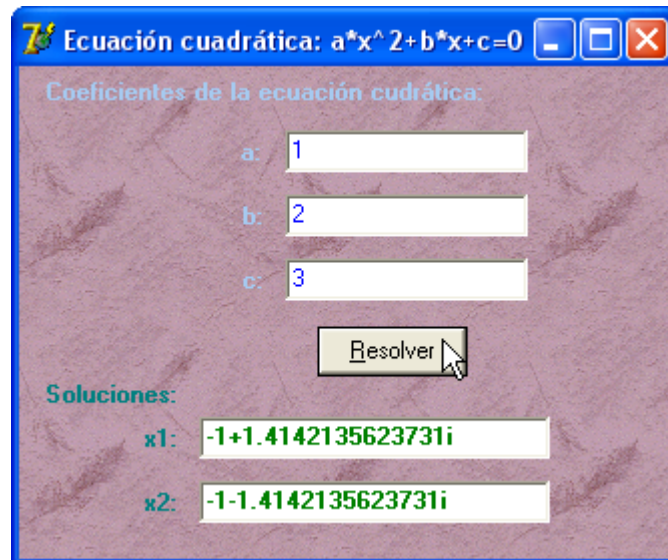


De esa manera los tres objetos comparten el mismo código de validación.

Finalmente, para dar funcionalidad a la aplicación programamos el evento "onClick" del botón "bResolver":

```
procedure TfCuadratica.bResolverClick(Sender: TObject);
var a,b,c,x1,x2,im: double;
begin
  LeerCoeficientes(ea,eb,ec,a,b,c);
  ResolverCuadratica(a,b,c,x1,x2,im);
  MostrarRaices(ex1,ex2,x1,x2,im);
end;
```

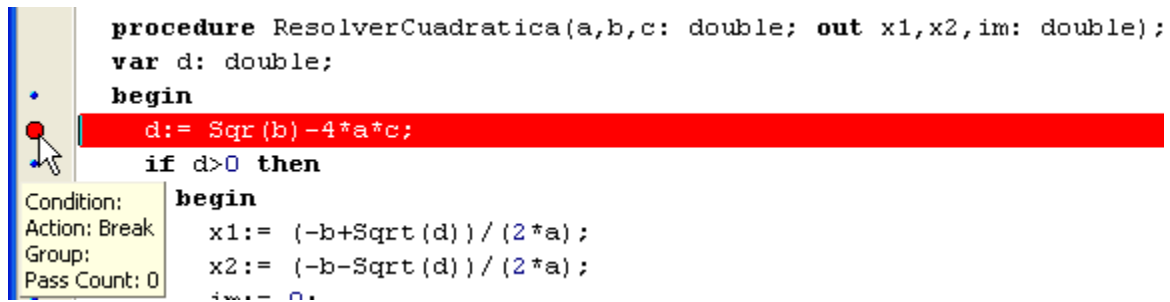
Ahora se compila la aplicación, se corrigen los errores, se guardan los cambios y al hacer correr la aplicación se tendrá algo similar a la siguiente figura:



5.7. Ejecución de la aplicación línea por línea

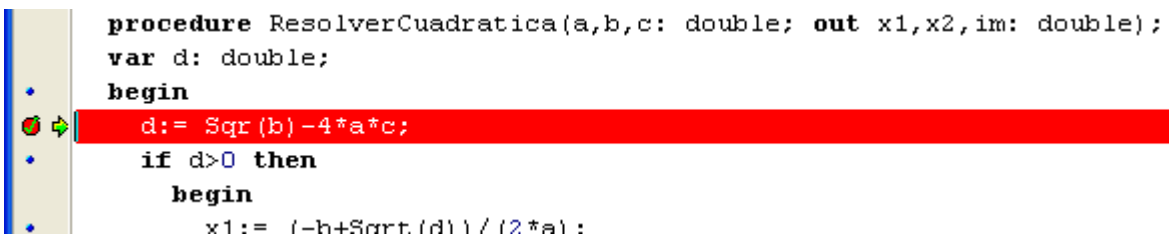
Ahora que la lógica de los módulos no es sólo secuencial sino que puede seguir uno u otro camino dependiendo del valor de la condición, es conveniente hacer correr la aplicación línea por línea para comprender mejor la lógica y corregir errores lógicos que son los más difíciles de encontrar y corregir pues no son detectados por el compilador.

Para hacer correr una aplicación línea por línea se coloca un punto de ruptura (*break point*), en la línea a partir de la cual se quiere ejecutar el programa línea por línea. Para colocar un punto de ruptura simplemente se hace click en el espacio en blanco a la izquierda de la línea (o estando en la línea se pulsa la tecla "F5"). Así por ejemplo para colocar un "break point" en la primera línea del módulo "ResolverCuadratica" se hace clic en el espacio en blanco al lado izquierdo de la misma:



Como puede observar la línea queda resaltada y el punto de la izquierda se torna de color rojo. Para quitar un punto de ruptura simplemente se vuelve a hacer clic sobre el punto rojo (o estando en la línea se pulsa la tecla "F5").

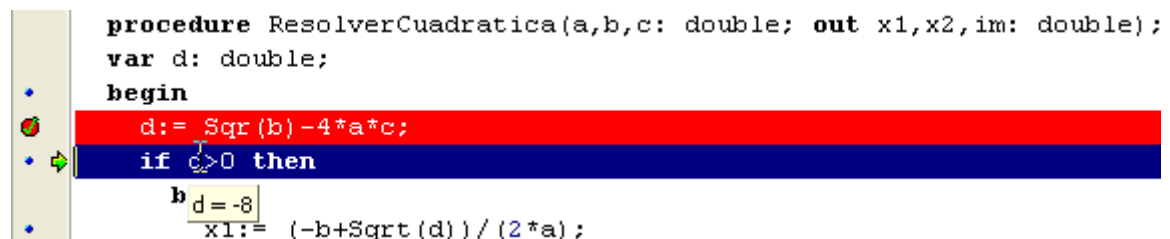
Ahora se hace correr el programa (F9), se introducen datos para los coeficientes (por ejemplo 1, 2 y 3) y cuando se hace clic en el botón "bResolver" el programa se detiene en el punto de ruptura del módulo "ResolverCuadratica":



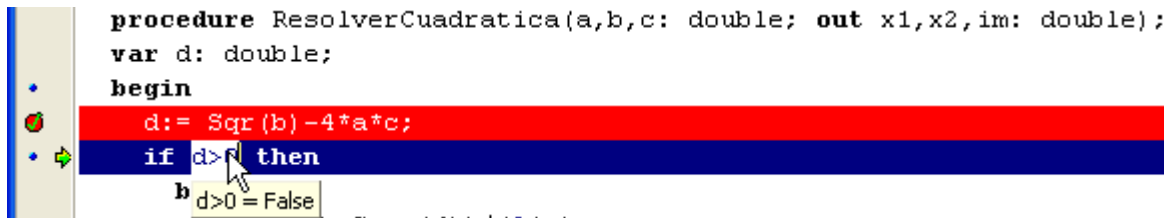
Una vez detenido el programa se puede hacer correr el mismo línea por línea. Para ello existen dos posibilidades "Step Over":  ("F8" o "Run -> Step Over") y "Trace Into":  ("F7" o "Run -> Trace Into"). En ambos casos el programa se ejecuta línea por línea, pero la diferencia radica en que si al hacer correr una línea existe una llamada a otro procedimiento o función, "Step Into" ingresa a dicho procedimiento o función y continúa la ejecución del programa dentro del mismo (como lo sugiere su icono: ) , mientras que "Step Over" no ingresa al procedimiento o función y pasa directamente a la siguiente línea (como también lo sugiere su icono: ) .

Puesto que en el módulo "ResolverCuadrática" (que estamos utilizando como ejemplo) no existen llamadas a otros procedimientos o funciones, tanto "Step Over" como "Trace Into" hacen lo mismo. Por lo tanto en este caso en particular se puede emplear indistintamente "Step Over" (F8) o "Trace Into" (F7), sin embargo se debe recordar que "Trace Into" ingresa a los procedimientos o funciones, mientras que "Step Over" no.

Ahora hagamos correr el programa una línea (F8), con lo que se calcula el valor de "d". Podemos ver el valor calculado colocando el puntero del mouse sobre la variable:



Para ver el resultado de una expresión, como "d>0", se la selecciona y se coloca el puntero del mouse sobre la misma:

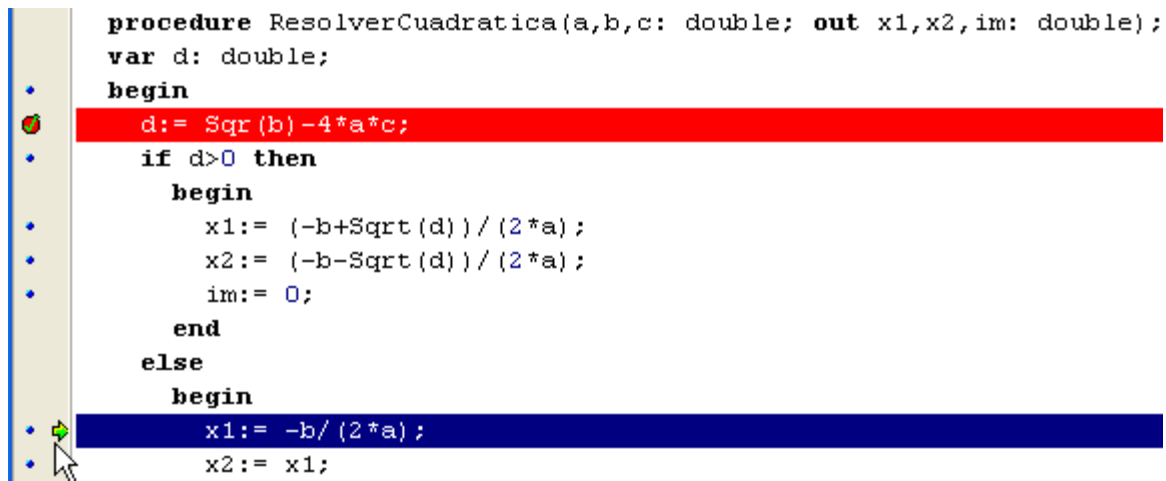


```

procedure ResolverCuadratica(a,b,c: double; out x1,x2,im: double);
var d: double;
begin
    d:= Sqr(b)-4*a*c;
    if d>0 then
        b d>0 = False

```

En este caso por ejemplo vemos que la condición es falsa, porque como ya hemos visto, el valor de "d" es negativo. Por lo tanto, si hacemos correr el programa otra línea (F8) el programa debe pasar al bloque "else", tal como efectivamente ocurre:



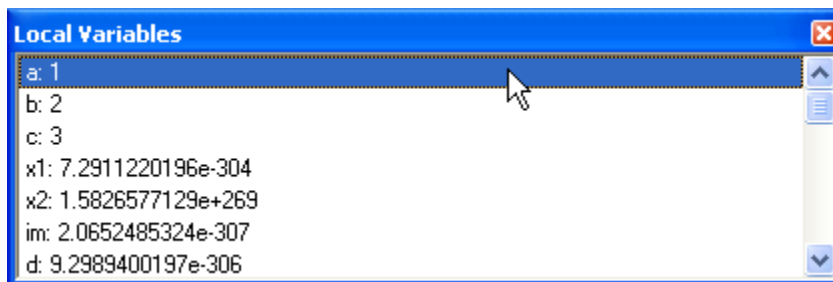
```

procedure ResolverCuadratica(a,b,c: double; out x1,x2,im: double);
var d: double;
begin
    d:= Sqr(b)-4*a*c;
    if d>0 then
        begin
            x1:= (-b+Sqrt(d))/(2*a);
            x2:= (-b-Sqrt(d))/(2*a);
            im:= 0;
        end
    else
        begin
            x1:= -b/(2*a);
            x2:= x1;
        end

```

Procediendo de esta manera, es posible comprobar si la lógica es o no correcta y corregir los errores cometidos. Para volver al modo de ejecución normal se pulsa "F9" (o se hace clic en el botón ).

Con frecuencia para comprender y corregir el código, es necesario ver el valor de todas las varias variables del módulo, entonces el procedimiento antes descrito (colocar el puntero del mouse sobre la variable), resulta muy tedioso. En esos casos es más cómodo y práctico emplear la ventana de variables locales "Local Variables", la cual se hace visible pulsando las teclas "Ctrl+Alt+L" o a través del menú "View -> Debug Windows -> Local Variables":



La ventana de ejemplo muestra los valores de las variables locales al momento de ingresar al módulo "ResolverCuadratica" y como se puede las variables no calculadas tienen valores aleatorios. Para tenerla siempre visible es conveniente insertarla en la parte inferior de la ventana de código (siguiendo el procedimiento explicado en "introducción a Delphi"). En la siguiente figura se muestra la ventana insertada, con los valores de las variables cuando ya se han calculado los valores de "d" y "x1":

```

begin
  x1:= -b/(2*a);
  x2:= x1;
  if d<0 then im:= Sqrt(Abs(d))/(2*a) else im:=0;
end;
end;

```

32: 1 Modified Insert \Code/

```

a: 1
b: 2
c: 3
x1: -1
x2: 1.5826577129e+269
im: 2.0652485324e-307
d: -8

```

Haga pruebas colocando puntos de ruptura en otros módulos y eventos y empleando tanto "Step Over" como "Step Into".

2. Ordenación de tres números enteros

Como segundo ejemplo elaboraremos una aplicación para ordenar ascendente o descendientemente tres números reales.

Los problemas en los que se puede subdividir el problema son: a) Leer los números, b) Ordenar los números ascendientemente, c) Ordenar los números descendientemente y e) Mostrar los números ordenados.

Codificaremos directamente los módulos relacionados a la lectura y presentación de resultados y nos concentraremos en los módulos que realmente resuelven el problema, es decir los que ordenan los números.

La lógica para ordenar los números ascendente o descendientemente es esencialmente la misma, sólo cambia el sentido de las condiciones. En general, sin embargo, sólo se requiere de una de ellas: si la lista está ordenada ascendientemente, se obtiene la lista descendente invirtiendo el orden de los elementos (o viceversa).

En consecuencia sólo es necesario que uno de los módulos realmente ordene los números mientras que el otro simplemente invierte los resultados.

En nuestro caso, el módulo que realmente ordenará los números es el de ordenación ascendente. Como casi siempre ocurre cuando se resuelve un problema en el campo de la informática, no existe una solución única, sino varias posibilidades de solución. En este problema analizaremos tres de dichas posibilidades.

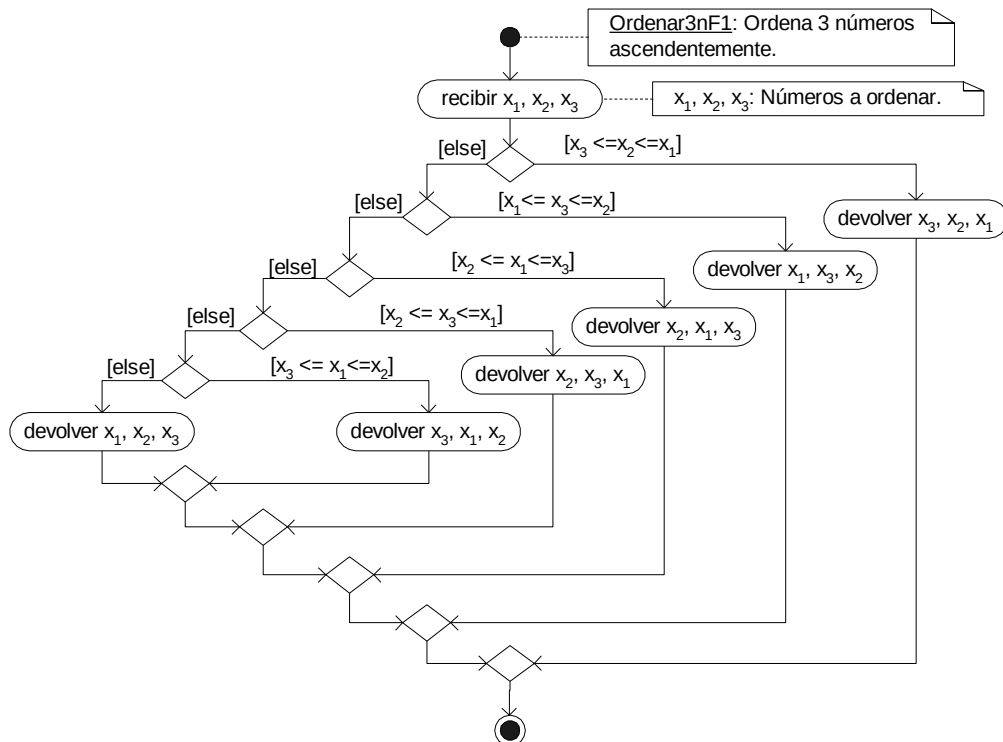
En la primera, que tiene el razonamiento más sencillo, aunque no la lógica más eficiente, simplemente se comprueban sistemáticamente los seis posibles casos de ordenación:

```

x1<=x2<=x3
x1<=x3<=x2
x2<=x1<=x3
x2<=x3<=x1
x3<=x2<=x1
x3<=x1<=x2

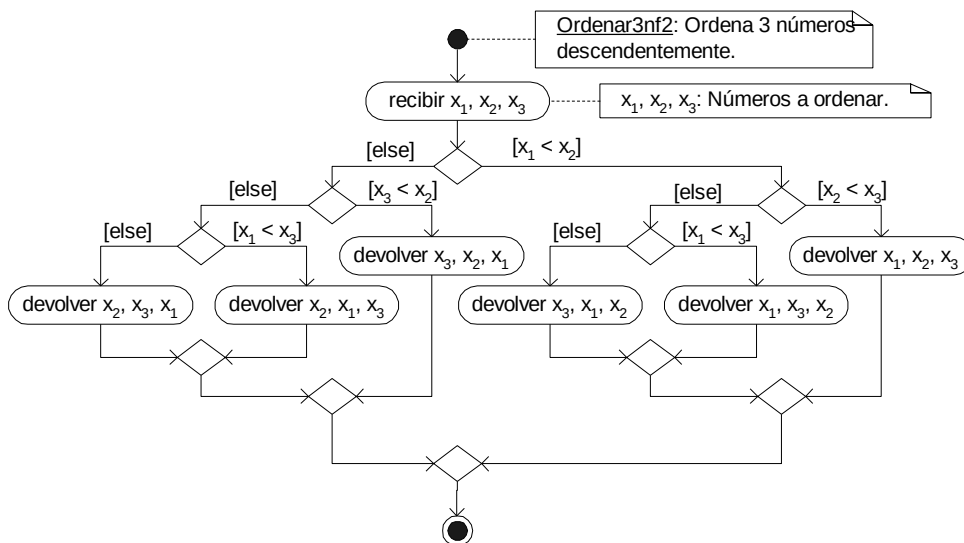
```

El diagrama de actividades de esta alternativa es el siguiente:



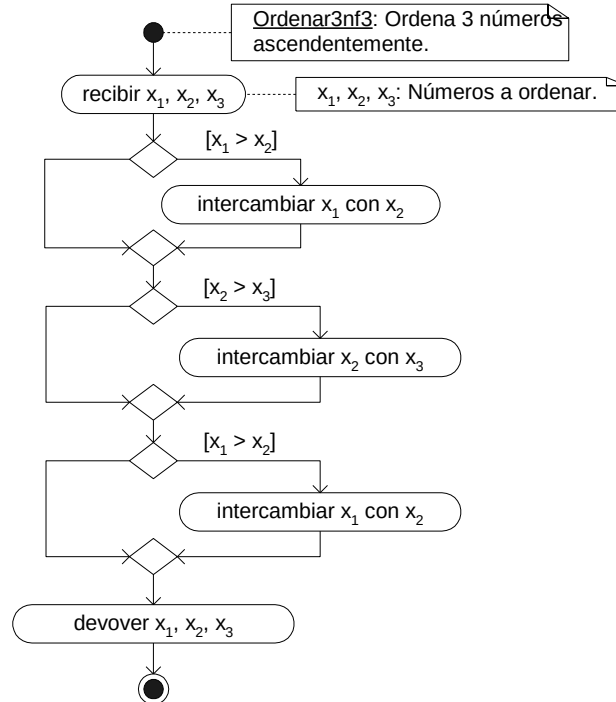
Y como se puede ver se trata de 5 estructuras IF-THEN-ELSE anidadas donde sólo se consideran 5 de los 6 casos, pues el sexto es el caso por defecto. Como veremos un poco después, este tipo de lógica puede ser programada de manera más eficientemente con la estructura "Case".

En la segunda alternativa se comparan pares de valores y como resultado de dicha comparación se sabe cual de los dos es el mayor (o menor), entonces se comparan estos valores con el tercero para determinar sus posiciones relativas. El algoritmo de esta alternativa es el siguiente:



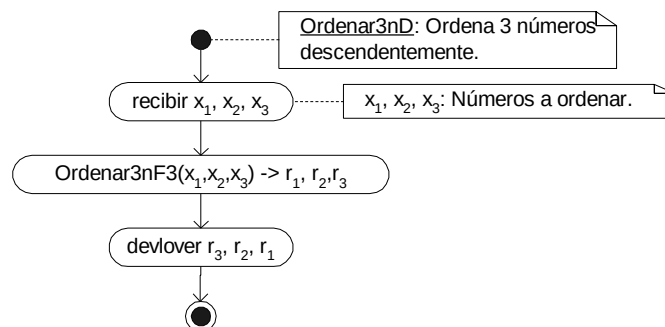
Como se puede observar la solución involucra también estructuras IF-THEN-ELSE anidadas, sin embargo esta solución es más eficiente (aunque más difícil de entender que la primera) porque sólo requiere un máximo de 3 comparaciones en lugar de las 5 de la primera alternativa.

Una tercera alternativa que es más eficiente que las dos anteriores y que sin embargo, es también más fácil de entender es la siguiente:



En esta lógica, las dos primeras preguntas garantizan que el mayor valor se encuentre en " x_3 " y la última garantiza que el segundo valor se encuentre en " x_2 ".

El módulo que ordena los números descendientemente llama a cualquiera de los módulos anteriores y simplemente invierte el resultado. El algoritmo de dicho módulo es el siguiente:



Una vez resuelta la lógica del problema podemos codificar la solución. Para ello creamos una nueva aplicación (File -> New -> Application), una nueva unidad (File -> New -> Unit) y guardamos la aplicación (File -> Save Project As...) en el directorio "Selección\Ordenar3n\" con los nombres: "ufOrdenar3" para "Unit1", "uOrdenar3" para "Unit2" y "pOrdenar3" para "Project1".

Entonces en la unidad "uOrdenar3" escribimos el código de los 6 módulos:

```

unit uOrdenar3;

interface

uses StdCtrls, Sysutils;
  
```

```

function LeerNumReal(e: TEdit): real;
procedure Ordenar3nF1(var x1,x2,x3: real);
procedure Ordenar3nF2(var x1,x2,x3: real);
procedure Ordenar3nF3(var x1,x2,x3: real);
procedure Ordenar3nD3(var x1,x2,x3: real);
procedure MostrarNumReal(x: real; e: TEdit);

```

implementation

```

function LeerNumReal(e: TEdit): real;
begin
    result:= StrToFloat(e.Text);
end;

procedure Ordenar3nF1(var x1,x2,x3: real);
var r1,r2,r3: real;
begin
    r1:= x1; r2:= x2; r3:= x3;
    if (x3<=x2) and (x2<=x1) then
        begin x1:=r3; x3:=r1 end
    else
        if (x1<=x3) and (x3<=x2) then
            begin x2:=r3; x3:=r2 end
        else
            if (x2<=x1) and (x1<=x3) then
                begin x1:=r2; x2:=r1; end
            else
                if (x2<=x3) and (x3<=x1) then
                    begin x1:=r2; x2:=r3; x3:=r1 end
                else
                    if (x3<=x1) and (x1<=x2) then
                        begin x1:=r3; x2:=r1; x3:=r2 end;
            end;
    end;

procedure Ordenar3nF2(var x1,x2,x3: real);
var r1,r2,r3: real;
begin
    r1:=x1; r2:=x2; r3:=x3;
    if x1<=x2 then
        if x2<=x3 then
            else
                if x1<=x3 then
                    begin x2:=r3; x3:=r2 end
                else
                    begin x1:=r3; x2:=r1; x3:=r2 end
            end
        else
            if x3<=x2 then
                begin x1:=r3; x3:=r1 end
            else
                if x1<=x3 then
                    begin x1:=r2; x2:=r1; end
                else
                    begin x1:=r2; x2:=r3; x3:=r1 end;
            end;
    end;

procedure Ordenar3nF3(var x1,x2,x3: real);
procedure intercambiar(var x1,x2: real);
var aux: real;

```



```

begin
  aux:=x1; x1:=x2; x2:=aux;
end;
begin
  if x1>x2 then intercambiar(x1,x2);
  if x2>x3 then intercambiar(x2,x3);
  if x1>x2 then intercambiar(x1,x2);
end;

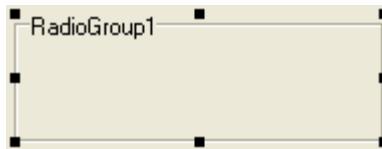
procedure Ordenar3nD3(var x1,x2,x3: real);
var r1,r2,r3: real;
begin
  r1:=x1; r2:=x2; r3:=x3;
  Ordenar3nF2(r1,r2,r3);
  x1:=r3; x2:=r2; x3:=r1;
end;

procedure MostrarNumReal(x: real; e: TEdit);
begin
  e.Text:= FloatToStr(x);
end;

end.

```

En esta aplicación emplearemos un nuevo componente el "RadioGroup": , que se encuentra en la página "Standard" y que al ser colocado en la forma tiene la siguiente apariencia:

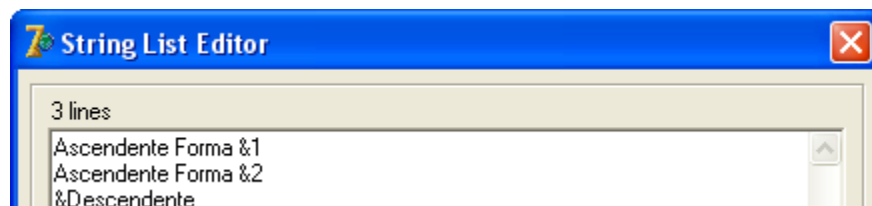


El "RadioGroup" agrupa un conjunto de "RadioButtons":



Los cuales se caracterizan porque de todos los "RadioButtons" existentes en un grupo, sólo puede estar seleccionado uno. Por lo tanto el "RadioGroup" se emplea cuando se da al usuario dos o más opciones, pero sólo se le permite elegir una de ellas.

Las propiedades de uso más frecuente del "RadioGroup" son: "Name" para el nombre, "Caption" para el título; "Columns" para el número de columnas; "Items" para los títulos de cada uno de los "RadioButtons" e "ItemIndex" para el número de "RadioButton" seleccionado, siendo "-1" cuando no está seleccionado ningún "RadioButton", "0" cuando está seleccionado el primero, "1" cuando está seleccionado el segundo y así sucesivamente. En la presente aplicación se han modificado las siguientes propiedades de este componente: Name = rgOrden, Caption = Orden; Font.Style = [fsBold]; ItemIndex = 0 y en la propiedad "Items", para la cual se abre una nueva ventana, se han escrito los títulos:



Los otros componentes han sido empleados ya en otras aplicaciones. Las propiedades que se han modificado en los mismos son:

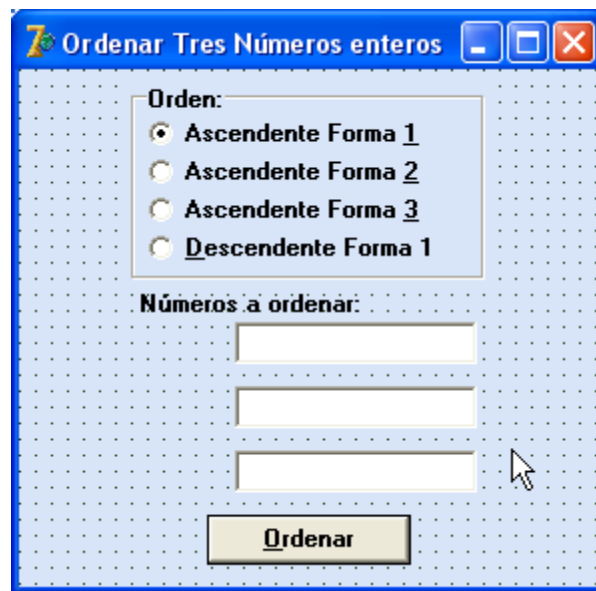
Form1: Name = 'fOrdenar', Caption = 'Ordenar Tres Números Enteros', Width = 298, Height = 263, Position = poScreenCenter, Color = clInactiveCaptionText.

Label1: Caption = 'Números a ordenar:', Transparent = True, Font.Style = [fsBold].

Edit1: Name = 'ex1'; Edit2: Name = 'ex2'; Edit3: Name = 'ex3'. Propiedades comunes: Text = '', Font.Style = [fsBold].

Button1: Name = 'bOrdenar', Caption = '&Ordenar', Default = True.

Coloque estos componentes en la forma, cambie sus propiedades a los valores dados, cambie sus tamaños y ordénelos (con la paleta de alineación) de manera que queden aproximadamente como se muestra en la figura:



En esta aplicación, como se puede ver, el fondo de la forma y del Radio-Group tienen el estilo de líneas diagonales cruzadas (las cuales deben ser modificadas en el evento "onCreate" de la forma):

Como esta aplicación permite ordenar números enteros, programamos el evento "onKeyPress" del Edit "ex1" de manera que sólo permita escribir los caracteres correspondientes a números reales:

```
procedure TfOrdenar.ex1KeyPress(Sender: TObject; var Key: Char);
begin
  if not (Key in [#8,'0'..'9','.','-']) then
    begin
      Beep; Abort;
    end;
end;
```

Al igual que en la anterior aplicación, para que los Edits "ex2" y "ex3" queden validados, simplemente se selecciona el módulo "ex1KeyPress" en el evento "onKeyPress" de dichos objetos.

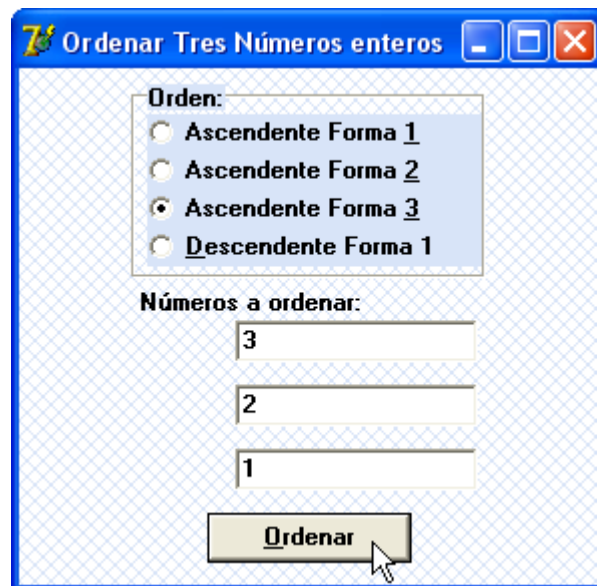
Finalmente, para dar funcionalidad a la aplicación programamos el evento "onClick" del botón "bOrdenar", no olvidando antes incluir la unidad "uCua-dratica" en la unidad "ufCuadrática" (File -> Use Unit... -> uCuadratica).

```

procedure TfOrdenar.bOrdenarClick(Sender: TObject);
var x1,x2,x3: real;
begin
  x1:= LeerNumReal(ex1);
  x2:= LeerNumReal(ex2);
  x3:= LeerNumReal(ex3);
  if rgOrden.ItemIndex=0 then Ordenar3nF1(x1,x2,x3)
  else if rgOrden.ItemIndex=1 then Ordenar3nF2(x1,x2,x3)
  else if rgOrden.ItemIndex=2 then Ordenar3nF3(x1,x2,x3)
  else if rgOrden.ItemIndex=3 then Ordenar3nD3(x1,x2,x3);
  MostrarNumReal(x1,ex1);
  MostrarNumReal(x2,ex2);
  MostrarNumReal(x3,ex3);
end;

```

Finalmente se compila la aplicación (Ctrl+F9), se corrigen los errores y al ejecutarla obtendrá algo parecido a la siguiente figura:



Para comprender mejor la lógica de los módulos creados en esta aplicación, coloque puntos de ruptura en algunos de ellos y haga correr la aplicación línea por línea.

5.8. Ejercicios

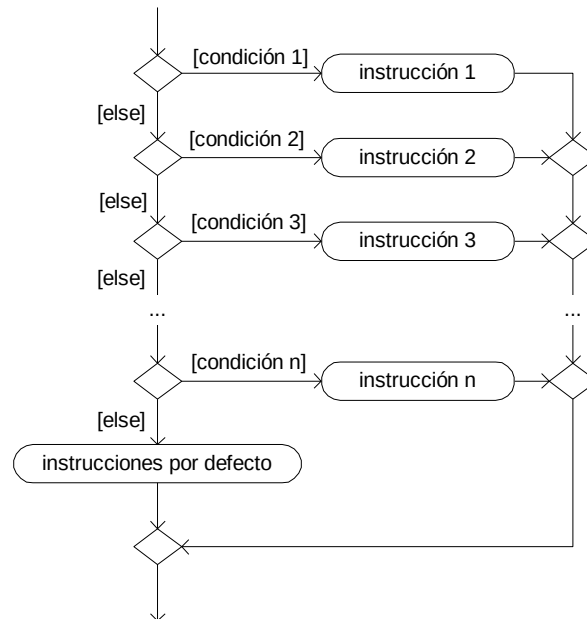
1. Elabore una aplicación para determinar si tres lados dados conforman o no un triángulo: tres lados conforman un triángulo si la suma de cualesquier dos lados es siempre mayor al tercero. La aplicación debe tener su propio título, icono, estar centrada en la pantalla, su fondo debe ser de color verde y con líneas de 45 grados y la introducción de datos debe estar validada. Debe elaborar además el diagrama de actividades del módulo que determina si tres lados conforman o no un triángulo.
2. Elabore una aplicación para determinar si un año es o no bisiesto. Un año es bisiesto si es divisible entre 4, excepto los años que terminan en dos ceros, los cuales deben ser divisibles entre 400 (o lo que es lo mismo sin los dos ceros deben ser divisibles entre 4). La aplicación debe tener su propio título, icono, estar centrada en la pantalla, su fondo debe ser una figura y la introducción de datos debe estar validada.

Debe elaborar además el diagrama de actividades del módulo que determina si el año es o no bisiesto.

3. Elabore una aplicación que ordene ascendente y descendentemente tres números enteros, devolviendo los tres números ordenados en las variables de entrada. La aplicación debe tener su propio título, icono, estar centrada en la pantalla, su fondo debe estar con el color "clInactiveCaptionText". Debe elaborar además los diagramas de actividades de los módulos que ordenan los números y en la lógica de los mismos debe emplear un máximo de 3 estructuras "if-then".

6. ESTRUCTURA CASE

Estudiemos ahora la segunda estructura selectiva: la estructura CASE. Esta estructura resulta más sencilla y eficiente cuando la lógica involucra varias condiciones consecutivas. El diagrama de actividades de esta estructura es el siguiente.



En esta estructura se llevan a cabo una serie de preguntas consecutivas ejecutándose la instrucción correspondiente a la primera condición verdadera. Una vez que se ejecuta una de las instrucciones el control sale de la estructura y continúa con el resto del programa. Si ninguna de las condiciones es verdadera se ejecutan las instrucciones por defecto, o no se ejecuta nada, si las mismas no existen, pues son opcionales.

Debido a que la estructura CASE no es una estructura estándar (no forma parte del teorema de la programación estructurada) ha sido implementada de diferentes formas en los diferentes lenguajes de programación, teniendo inclusive diferentes nombres. En el caso específico de Pascal tiene dos limitantes importantes: a) Las condiciones sólo pueden ser igualdades o intervalos y b) los valores que se comparan sólo pueden ser de tipo ordinal.

En Pascal la estructura Case tiene la siguiente estructura:

```

case valor o expresión ordinal of
  caso 1: instrucciones 1;
  caso 2: instrucciones 2;
  caso 3: instrucciones 3;
  . . .
  caso n: instrucciones n;
else
  instrucciones por defecto
end;

```

Donde *valor o expresión ordinal* es una variable o una expresión cuyo resultado es un valor ordinal. Los casos (*caso 1, caso 2, etc.*), son valores ordinales separados por comas o intervalos separados por dos puntos (de la misma forma que los elementos de un conjunto). Así por ejemplo:

1,2,7: a:= sqr(x)+2;

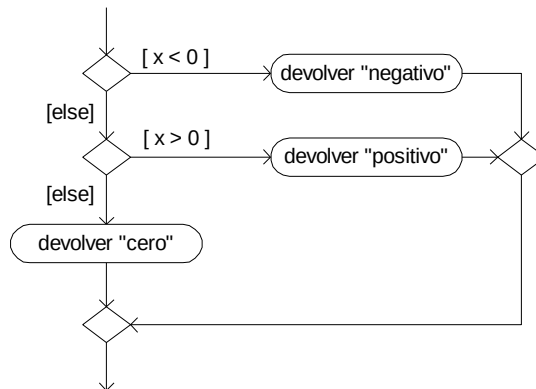
Es un caso que se ejecuta cuando el valor o expresión ordinal es igual a 1, 2 o 7. De la misma forma:

`'A', 'C', 'R'..'Z': b:= x+y;`

Es un caso que se ejecuta si el valor o expresión ordinal es igual a 'A', 'C' o está comprendido entre 'R' y 'Z'.

En Pascal, cuando las condiciones no son igualdades y valores ordinales, la estructura debe ser implementada con *IF-THEN-ELSE* o alternativamente las condiciones transformadas en una expresión ordinal.

Para convertir una condición en un valor ordinal, se emplea la función *Ord()*, que como sabemos transforma un valor ordinal de tipo "boolean" en "0" cuando es "False" y en "1" cuando es "True". Por ejemplo, para codificar el diagrama de actividades:



Escribimos:

```

c:= ord(x<0)+ord(x>0)*2;
case c of
  1: result:= 'negativo';
  2: result:= 'positivo';
  else result:= 'cero';
end;

```

Analicemos como es que se han transformado las condiciones en valores ordinales.

Cuando "x" es menor a "0", la expresión "x<0" devuelve "True" y la función "Ord" transforma este valor en "1", ahora bien, si "x" es menor "0", entonces no puede ser mayor a "0", por lo tanto la expresión "x>0" devuelve "False" y la función "Ord" transforma este valor en "0", que al ser multiplicado por "2" sigue siendo "0", en consecuencia el resultado de toda la expresión es "1+0=1" y es este valor el que se guarda en la variable "c". Entonces cuando "x<0" se transforma en el valor ordinal "1".

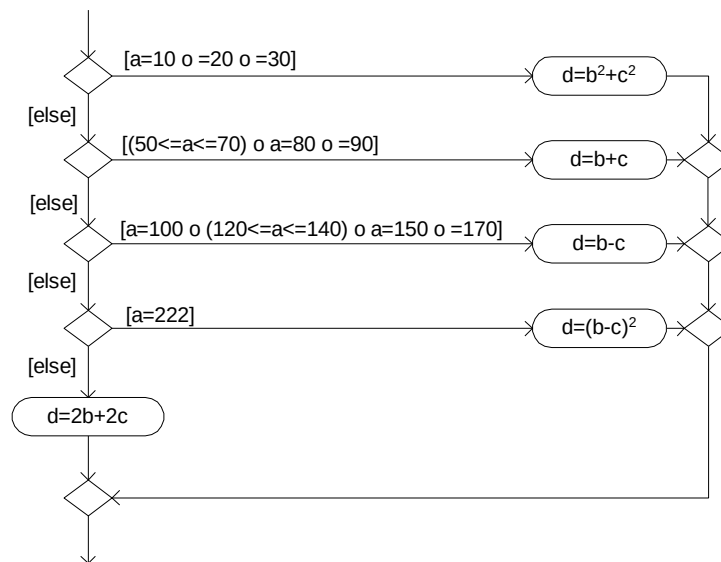
Por el contrario, si "x" es mayor a "0", la expresión "x>0" devuelve "True" y la función "Ord" la transforma en "1", que al ser multiplicado por "2" devuelve "2". Pero si "x" es mayor a "0", no puede ser menor a "0", por lo tanto la expresión "x<0" devuelve "False" que la función "Ord" transforma en "0". En consecuencia el resultado de toda la expresión es "0+2=2", que es el valor que se guarda en la variable "c". Por lo tanto cuando "x>0" se transforma en el valor ordinal "2".

Este procedimiento puede extenderse a cualquier número de condiciones, multiplicando el valor ordinal de las condiciones por 1, 2, 3, 4, 5 y así sucesivamente. De esta manera, si la tercera condición es verdadera se ob-

tiene el valor ordinal "3", si la cuarta sea verdadera, el valor ordinal "4" y así sucesivamente.

Para que el anterior procedimiento funcione, es imprescindible que las condiciones (o casos) de la estructura *CASE* sean excluyente, es decir que nunca 2 o más condiciones sean verdaderas al mismo tiempo. Así si en el anterior ejemplo las condiciones fueran " $x \leq 0$ " y " $x \geq 0$ ", entonces ambas serían verdaderas cuando " x " es igual a "0", por lo que no podrían ser convertidos con seguridad en valores ordinales, no obstante, esto es algo que no se presenta cuando la lógica es correcta, por el contrario su presencia es una prueba inequívoca de un error lógico.

Cuando las condiciones son sólo igualdades o intervalos y se trata de valores ordinales, la traducción a *Pascal* es directa y generalmente más sencilla que en otros lenguajes, así la codificación del siguiente diagrama de actividades:

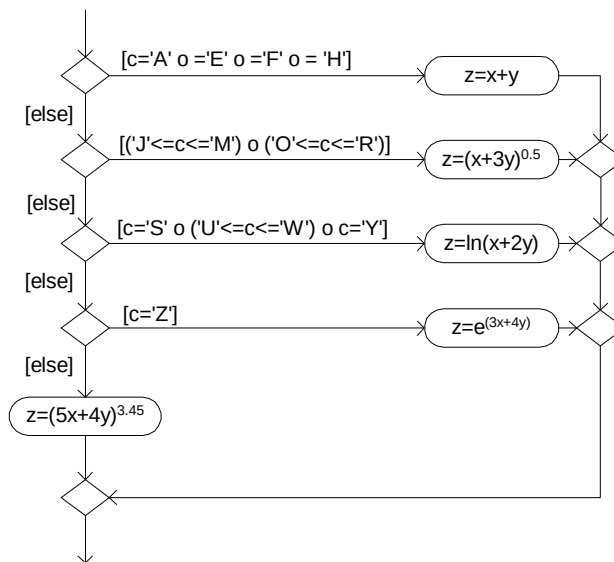
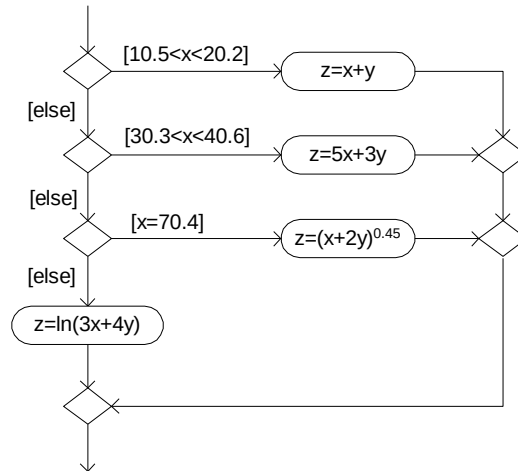


Es:

```

case a of
  10,20,30: d:= sqr(b)+sqr(c);
  50..70,80,90: d:= b+c;
  100,120..140,150,170: d:= b-c;
  222: d:= sqr(b-c);
else
  d:= 2*b+2*c;
end;
  
```

Para practicar con esta estructura, elabore el código de los siguientes diagramas de actividades:



6.1. Ejemplos

1. Ecuación cuadrática

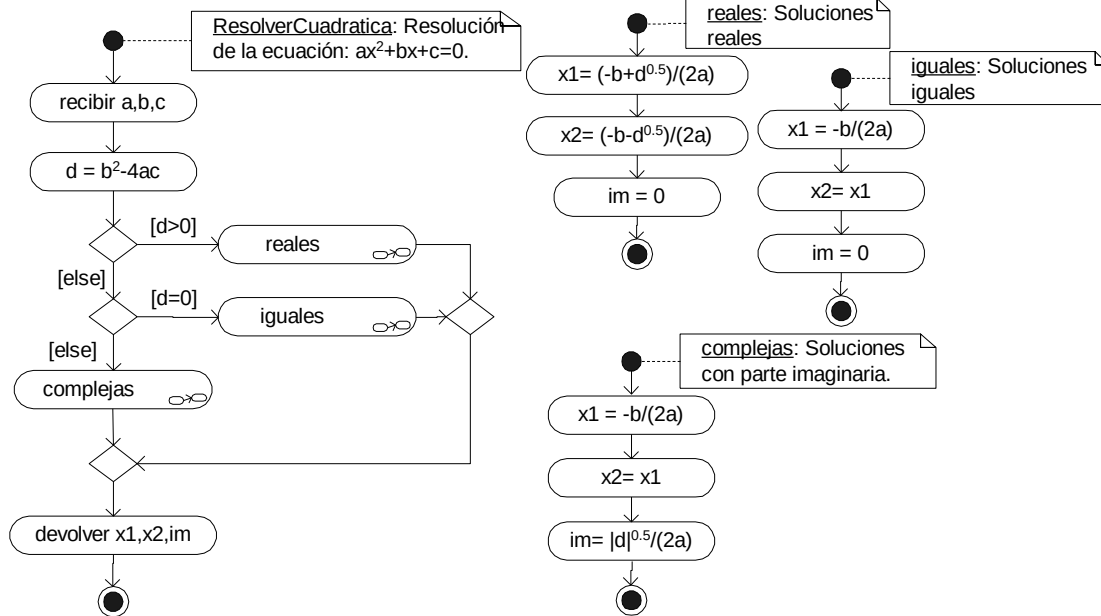
Volveremos a resolver el problema de la ecuación cuadrática, pero empleando la estructura *CASE*.

Lo único que varía con relación a la solución dada con la estructura *IF*, es la lógica del módulo principal que ahora se plantea en forma de casos en función al discriminante "*d*": "*d* > 0", cuando las soluciones son reales; "*d* = 0", cuando las soluciones son iguales y "*d* < 0" (o caso contrario) cuando las soluciones son complejas, tal como se muestra en el diagrama de actividades de la siguiente página.

En este ejemplo, y sólo por razones de claridad, los diferentes casos se detallan en diagramas independientes en forma de actividades.

Ahora codifiquemos la solución, que esencialmente es la misma que en el mencionado ejemplo, con la diferencia de que en esta ocasión emplearemos, siempre que sea posible, la estructura "*case*", en lugar de "*if*". Para ello creamos una nueva aplicación (File -> New Application), una nueva unidad (File -> New Unit) y guardamos la aplicación (File -> Save Project As...) en

el directorio "Selección\Cuadrática-Case\" con los nombres: "ufCuadraticaC" para "Unit1", "uCuadraticaC" para "Unit2" y "pCuadraticaC" para "Project1".



Entonces escribimos el código que da solución al problema en la unidad "uCuadraticaC":

```
unit uCuadraticaC;
```

```
interface
```

```
uses StdCtrls, SysUtils;
function LeerNumReal(e: tEdit):double;
procedure ResolverCuadratica(a,b,c: double; out x1,x2,im: double);
procedure MostrarRaices(x1,x2,im: double; ex1,ex2: tEdit);
```

```
implementation
```

```
function LeerNumReal(e: tEdit):double;
begin
  result:= StrToFloat(e.Text);
end;

procedure ResolverCuadratica(a,b,c: double; out x1,x2,im: double);
var d: double; casos: integer;
  procedure reales;
  begin
    x1:= (-b+Sqrt(d))/(2*a); x2:= (-b-Sqrt(d))/(2*a); im:= 0;
  end;
  procedure iguales;
  begin
    x1:= -b/(2*a); x2:= x1; im:= 0;
  end;
  procedure complejas;
  begin
    x1:= -b/(2*a); x2:= x1;
    if d<0 then im:= Sqrt(Abs(d))/(2*a) else im:=0;
  end;
begin
```

```

d:= Sqr(b)-4*a*c;
casos:= Ord(d>0)+Ord(d=0)*2;
case casos of
  1: reales;
  2: iguales;
  else complejas;
end;
end;

procedure MostrarRaices(x1,x2,im: double; ex1,ex2: tEdit);
begin
  case im=0 of
    True:
      begin
        ex1.Text:= FloatToStr(x1); ex2.Text:= FloatToStr(x2);
      end
    else
      ex1.Text:= FloatToStr(x1)+'+'+FloatToStr(im)+'i';
      ex2.Text:= FloatToStr(x1)+'-'+FloatToStr(im)+'i';
    end;
  end;
end.

```

La interfaz de la aplicación es la misma que en la estructura "if", por lo que no se requiere ninguna explicación adicional. Sin embargo, en esta ocasión, la validación del Edit "ex1" se la hace empleando la estructura "case" en lugar de "if":

```

procedure TfCuadratica.eaKeyPress(Sender: TObject; var Key: Char);
begin
  case Key of
    #8, '0'..'9', '.', '+', '-', 'e', 'E':
      else Beep; Abort;
  end;
end;

```

El evento "onClick" del botón cambia también ligeramente, porque en este caso los coeficientes se leen uno por uno y no todos en uno como se hizo al emplear if, además se ha cambiado el orden de los parámetros en el módulo "MostrarRaices":

```

procedure TfCuadratica.bResolverClick(Sender: TObject);
var a,b,c,x1,x2,im: double;
begin
  a:= LeerNumReal(ea);
  b:= LeerNumReal(eb);
  c:= LeerNumReal(ec);
  ResolverCuadratica(a,b,c,x1,x2,im);
  MostrarRaices(x1,x2,im,ex1,ex2);
end;

```

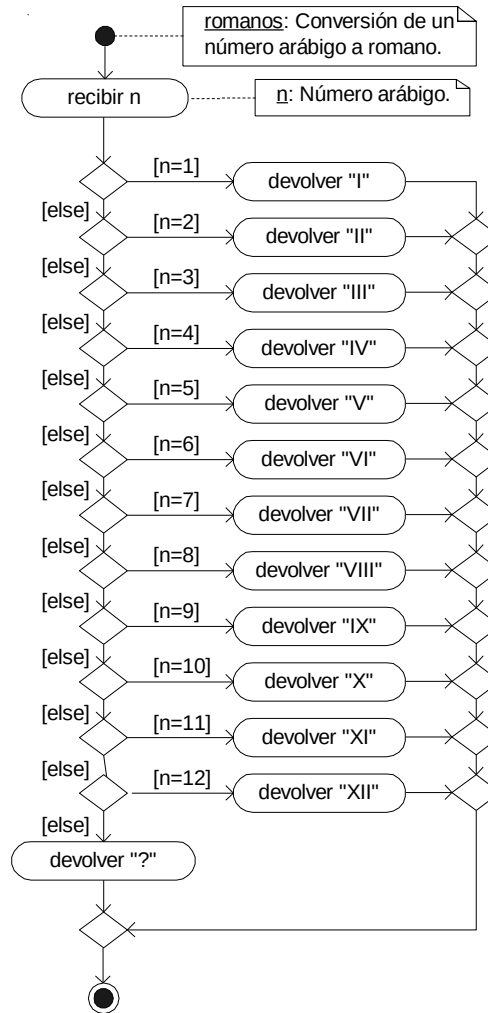
El resto de los eventos programados en esta aplicación no cambian con relación a la aplicación hecha para la estructura "if".

2. Números romanos

Ahora elaboraremos una aplicación para convertir números arábigos comprendidos entre 1 y 12 en números romanos.

Los módulos que podemos identificar en este problema son: a) Leer el número arábigo, b) Convertir el número arábigo en romano y c) Mostrar el número convertido.

Codificaremos directamente los módulos (a) y (c). La lógica del módulo principal (b) es también bastante sencilla: si el número arábigo es "1" se devuelve el equivalente romano "I", si es "2" se devuelve "II" y así sucesivamente, tal como se muestra en el siguiente diagrama de actividades:



Esta lógica encaja perfectamente en la estructura "case" de Pascal. Para codificar el código creamos una nueva aplicación (File -> New Application), una nueva unidad (File -> New Unit) y guardamos el proyecto (File -> Save Project As...) en el directorio "Selección\Romanos\" con los nombres: "ufRomanos" para "Unit1", "uRomanos" para "Unit2" y "pRomanos" para "Project1".

Ahora escribimos el código en la unidad "uRomanos":

```
unit uRomanos;
```

```
interface
```

```
  uses StdCtrls, Sysutils;
```

```
  function LeerArabigo(cb: TComboBox): byte;
```

```
  function Romanos(n: byte): string;
```

```
  procedure MostrarRomano(e: TEdit; r: string);
```

implementation

```

function LeerArabigo(cb: TComboBox): byte;
begin
    result:= StrToInt(cb.Text);
end;

function Romanos(n: byte): string;
begin
    case n of
        1: result:= 'I';
        2: result:= 'II';
        3: result:= 'III';
        4: result:= 'IV';
        5: result:= 'V';
        6: result:= 'VI';
        7: result:= 'VII';
        8: result:= 'VIII';
        9: result:= 'IX';
        10: result:= 'X';
        11: result:= 'XI';
        12: result:= 'XII';
    else
        result:= '?';
    end;
end;

procedure MostrarRomano(e: TEdit; r: string);
begin
    e.Text:= r;
end;

```

end.

Una vez compilada la unidad (Ctrl+F9) y corregido los errores guardamos las modificaciones (Ctrl+S).

Como puede observar, en el código se hace referencia a un nuevo tipo "TComboBox" (que se encuentra en la unidad StdCtrls). Este tipo corresponde al nuevo componente que emplearemos en esta aplicación un "ComboBox": , el cual se encuentra en la página "Standard" y que al ser colocado en la forma tiene la siguiente apariencia:



Este componente es empleado cuando se permite escribir información o elegir la de una lista de opciones. Se denomina "Combo" (de combinar) porque combina las funcionalidades y propiedades de un "Edit" y de un "ListBox". Como un "Edit" permite escribir datos y como un "ListBox" permite seleccionar valores de la lista que aparece cuando se hace clic en el botón: .

Las propiedades más usuales de este componente son: "Name" para el nombre del objeto, "Text" para el texto que aparece en el cuadro de edición, "ItemIndex" para el número de opción elegida (igual que en el "RadioGroup"), "Items" para las opciones que aparecen en la lista (igual que en el "RadioGroup") y "Style" para el estilo de "ComboBox" a emplear.

Los estilos (Style) de "ComboBox" son: csDropDown cuando se permite escribir y escoger elementos de la lista, csDropDownList cuando sólo se permi-

te escoger elementos de la lista, csOwnerDrawFixed que es igual a "csDropDown" pero donde se puede fijar la altura de las opciones; csOwnerDrawVariable que es igual a "csOwnerDrawFixed" pero donde cada elemento puede tener una altura diferente y csSimple que no muestra la lista de opciones pero que permite elegir valores de la misma desplazándose a través de ellas con las teclas de los cursores.

Como ya vimos cuando trabajamos con el "RadioGroup", la propiedad "Items" presenta una nueva ventana donde se pueden escribir los elementos de la lista, que en el caso del "RadioGroup" corresponden a los títulos de los "RadioButtons" y que en el caso del "ComboBox" corresponden a las opciones que aparecen en la lista.

La propiedad "Items" es en realidad un objeto de tipo *TStrings* y debido a que esta clase es empleada en diferentes componentes, es conveniente conocer algunas de sus propiedades y métodos. Desde un punto de vista simplificado, un *TStrings* puede ser visto como un vector (array) de cadenas (strings), sin embargo, al ser un objeto tiene propiedades y métodos que nos permiten manipular y obtener información de la lista.

Algunas de sus propiedades de uso frecuente son: "Count": que devuelve el número de elementos de la lista; "Text": que devuelve la lista como una sola cadena, donde cada línea está separada por retornos de carro y saltos de línea (códigos #13 y #10); "Strings": que permite acceder a cualquier elemento de la lista por su índice (posición), siendo el primer elemento el número 0, la propiedad "Strings" es la propiedad por defecto de este objeto y en *Delphi* las propiedades por defecto pueden omitirse, así obtenemos el mismo resultado si escribimos *ComboBox1.Strings[1]* o *ComboBox1[1]*.

Algunos de los métodos de uso más frecuente son: "Append": añade un elemento a la lista, así *Append('Hola')*, añade la palabra "Hola" como último elemento de la lista; "Clear": borra los elementos de la lista; "Delete": borra un elemento de la lista, así *Delete(2)*, borra el tercer elemento de la lista; "IndexOf": devuelve la posición en la que se encuentra una cadena en la lista, así *IndexOf('hola mundo')*, devuelve la posición (o número de fila) en la que se encuentra el texto "hola mundo", devolviendo -1 si la cadena no está en la lista; "Insert": inserta un elemento en una posición específica, así *Insert(1,'Hola')*, inserta 'Hola' en la segunda fila; "SaveToFile": guarda la lista en un archivo, así *SaveToFile('c:\listas\opciones1.lst')*, guarda la lista en el archivo "opciones1.lst" del directorio "c:\listas\"; "LoadFromFile": carga una lista desde un archivo, así *LoadFromFile('c:\Listas\opciones1.lst')*, carga la lista del archivo "opciones1.lst" del directorio "c:\listas\"; "BeginUpdate": impide que los controles se actualicen mientras se cambian los valores de la lista; "EndUpdate": vuelve a activar la actualización de los controles cuando se cambian los valores de la lista.

Ahora que sabemos algo más con relación al "ComboBox" y a su propiedad "Items", podemos trabajar en la interfaz de la aplicación. Las propiedades modificadas para los componentes que se emplean en esta aplicación son:

Form1: Name = 'fOrdenar', Caption = 'Ordenar Tres Números Enteros', Width = 250, Height = 133, Position = poScreenCenter, Color = clInactiveCaptionText.

ComboBox1: Name = 'cbArabigo', Items = {1,2,3,4,5,6,7,8,9,10,11,12}, ItemIndex = 0, Style = csDropDownList.

Label1: Caption = 'Número arábigo:'; Label2: Caption = 'Número romano:'.
Propiedades comunes: Font.Color = clBlue, Transparent = True.

```
Edit1: Name = 'eRomano', Text = 'I', Font.Color = clNavy, ReadOnly =
True; TabStop = False.
```

Coloque los componentes en la forma, cambie sus propiedades, modifique sus tamaños y ordénelos de manera que queden aproximadamente como se muestra en la figura (no olvide guardar las modificaciones):



En esta aplicación el fondo de la forma se pintará con el estilo "bsFDiagonal". Para este fin programamos el evento "onCreate":

```
procedure TfRomanos.FormCreate(Sender: TObject);
begin
    fRomanos.Brush.Style:= bsFDiagonal;
end;
```

En esta ocasión no validamos la introducción de datos, porque en realidad sólo se permite seleccionar datos, no introducirlos, pues el estilo de "ComboBox" es "csDropDownList". Tampoco existe un botón debido a que la conversión se llevará a cabo directamente cuando se seleccione un nuevo número arábigo. Para este fin programamos el evento "onChange" del "ComboBox" "cbArabigo" (el cual ocurre cuando cambia su contenido):

```
procedure TfRomanos.cbArabigoChange(Sender: TObject);
var n: byte; r: string;
begin
    n:= LeerArabigo(cbArabigo);
    r:= Romanos(n);
    MostrarRomano(eRomano,r);
end;
```

Finalmente compilamos la aplicación (Ctrl+F9), corregimos los errores, guardamos las modificaciones (Ctrl+S) y hacemos correr la aplicación (F9) :



3. Control del movimiento de una figura

Ahora elaboraremos una aplicación donde controlaremos el movimiento de una figura en la pantalla mediante las teclas de los cursores.

Para esta aplicación requeriremos una figura en formato *BMP*. Dicha figura se puede crear en *Paint* de *Windows*: cree una nueva imagen (*Archivo -> Nuevo*), cambie el ancho a 30 píxeles y el alto a 21 píxeles (*Imagen -> Atributos*), amplíe la imagen (*Ver -> Zoom -> Personalizado -> 800%*) y cree una

imagen con la apariencia de un avión o cohete dirigido hacia la izquierda, manteniendo el fondo de la figura en color blanco.

Una vez creada la imagen guárdela como un BMP, con el nombre *Izquierda*, en el directorio "Selección\Avión\". Voltee horizontalmente la imagen (*Imagen -> Voltear y girar -> Voltear horizontalmente*) y guárdela con el nombre *Derecha*. Gire la imagen 270° (*Imagen -> Voltear y girar -> Girar por ángulo -> 270°*) y guárdela con el nombre *Arriba*. Finalmente gire la imagen verticalmente (*Imagen -> Voltear y girar -> Voltear verticalmente*) y guárdela con el nombre *Abajo*, de esa manera contará con cuatro figuras dirigidas a los cuatro lados.

Ahora cree una nueva aplicación en Delphi (File -> New Application), una nueva unidad (File -> New Unit) y guarde la aplicación (File -> Save Project As...) con los nombres: "ufAvion" para "Unit1", "uAvion" para "Unit2" y "pAvion" para "Project1".

Desde el punto de vista lógico, la resolución del problema es bastante sencilla: Cuando se pulsa el cursor hacia arriba se deberá emplear la imagen "Arriba.bmp" y la misma deberá ser dibujada una cierta distancia más arriba; cuando se pulsa el cursor hacia abajo se deberá emplear la imagen "Abajo.bmp" y la misma deberá ser dibujada una cierta distancia más abajo, procediendo de manera similar para los otros dos cursores.

Programaremos cada uno de los movimientos en un módulo independiente y en cada uno de ellos verificaremos si se ha llegado al borde de la ventana, de ser así, se hará sonar la alarma de la computadora para advertir al usuario que el avión no puede seguir avanzando en esa dirección. Sin embargo, dado que estos módulos sólo serán llamados desde la misma unidad, a través del módulo principal (MoverAvion), no son declarados en el sector de interface, sino directamente en el sector de implementación:

```
unit uAvion;
```

```
interface
```

```
  uses Forms, Graphics, QControls;
```

```
  type tDireccion = (arriba, abajo, izquierda, derecha, origen);
```

```
  procedure MoverAvion(f: TForm; avion: TBitmap; Direccion: tDireccion);
```

```
implementation
```

```
  var dx,dy: integer;
```

```
  procedure MoverOrigen(f: TForm; avion: TBitmap; var x,y: integer);
  begin
    f.Repaint; x:= 0; y:= 0; f.Canvas.Draw(x,y,avion);
  end;
```

```
  procedure MoverArriba(f: TForm; avion: TBitmap; var x,y: integer);
  begin
    if y>4 then
      begin
        f.repaint; y:=y-dy; f.Canvas.Draw(x,y,avion);
      end
    else Beep;
  end;
```

```
  procedure MoverAbajo(f: TForm; avion: TBitmap; var x,y: integer);
```

```

begin
  if y<(f.ClientHeight-Avion.Height-dy) then
    begin
      f.repaint; y:= y+dx; f.Canvas.Draw(x,y,avion);
    end
  else Beep;
end;

procedure MoverIzquierda(f: TForm; avion: TBitmap; var x,y: integer);
begin
  if x>4 then
    begin
      f.repaint; x:= x-dx; f.Canvas.Draw(x,y,avion)
    end
  else Beep;
end;

procedure MoverDerecha(f: TForm; avion: TBitmap; var x,y: integer);
begin
  if x<(f.ClientWidth-Avion.Width-dx) then
    begin
      f.repaint; x:= x+dx; f.Canvas.Draw(x,y,avion);
    end
  else Beep;
end;

procedure MoverAvion(f: TForm; avion: TBitMap; Direccion: TDireccion);
const x:integer=0; y:integer=0;
begin
  case Direccion of
    arriba: MoverArriba(f,avion,x,y);
    abajo: MoverAbajo(f,avion,x,y);
    izquierda: MoverIzquierda(f,avion,x,y);
    derecha: MoverDerecha(f,avion,x,y);
    else
      MoverOrigen(f,avion,x,y);
  end;
end;

initialization

  dx:= 5; dy:=5;

end.

```

Observe que en esta unidad se han empleado dos variables globales privadas: "dx" y "dy", las mismas que toman sus valores iniciales en el sector de "initialization" de la unidad. Las instrucciones de este sector se ejecutan cuando corre la aplicación y se emplea principalmente para inicializar variables.

Observe también que en el módulo principal "MoverAvion", se han empleado dos variables locales estáticas: "x" y "y", las cuales guardan las coordenadas (posición) del avión. Estas variables deben ser estáticas porque los valores de dichas coordenadas deben conservarse entre llamadas, pues de lo contrario se perdería la posición del avión y no habría forma de continuar el movimiento. El problema puede ser resuelto también empleando variables globales privadas, como se ha hecho con "dx" y "dy", se les deja la implementación de esta opción como ejercicio.

En todos los módulos el método "Repaint" de la forma ha sido llamado para volver a dibujar el fondo y con ello conseguir borrar la figura del avión. Sin esta orden, la figura anterior permanecería y entonces no se lograría el efecto de movimiento deseado (haga la prueba eliminando la llamada al método Repaint una vez que concluya la aplicación). El método Repaint no es la forma más eficiente de borrar la figura, pues se vuelve a dibujar todo el fondo, lo que consume mucho tiempo, existen otros métodos que nos permiten dibujar sólo la parte de la imagen que queremos modificar, estudiaremos algunos de estos métodos en los ejemplos de otros capítulos.

Compile la unidad (Ctrl+F9), corrija los errores y guarde los cambios (Ctrl+S). Ahora vaya a la forma (Shift+F12) y cambie las siguientes propiedades: Name = 'fAvion', Caption = 'Avión', Height = 160, Width = 228, Position = poScreenCenter.

Para las cuatro imágenes que emplearemos en esta aplicación, declararemos cuatro variables: "bmDerecha", "bmIzquierda", "bmArriba" y "bmAbajo" en el sector privado (private) de la forma "fAvion":

```
type
  TfAvion = class (TForm)
  private
    bmIzquierda: TBitmap;
    bmDerecha: TBitmap;
    bmArriba: TBitmap;
    bmAbajo: TBitmap;
  public
    { Public declarations }
  end;
```

Ahora cargamos las imágenes para los aviones y para el fondo de la forma en el evento "onCreate" y las liberamos en el evento "onClose":

```
procedure TfAvion.FormCreate(Sender: TObject);
begin
  fAvion.Brush.Bitmap := TBitmap.Create;
  fAvion.Brush.Bitmap.LoadFromFile('C:\WINDOWS\Roca verde.bmp');
  bmIzquierda:= TBitmap.Create;
  bmIzquierda.LoadFromFile('C:\SIS101\Selección\Avión\izquierda.bmp');
  bmIzquierda.Transparent:= True;
  bmIzquierda.TransparentColor:= clWhite;
  bmDerecha:= TBitmap.Create;
  bmDerecha.LoadFromFile('C:\SIS101\Selección\Avión\derecha.bmp');
  bmDerecha.Transparent:= True;
  bmDerecha.TransparentColor:= clWhite;
  bmArriba:= TBitmap.Create;
  bmArriba.LoadFromFile('C:\SIS101\Selección\Avión\arriba.bmp');
  bmArriba.Transparent:= True;
  bmArriba.TransparentColor:= clWhite;
  bmAbajo:= TBitmap.Create;
  bmAbajo.LoadFromFile('C:\SIS101\Selección\Avión\abajo.bmp');
  bmAbajo.Transparent:= True;
  bmAbajo.TransparentColor:= clWhite;
end;

procedure TfAvion.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  bmIzquierda.Free;
  bmDerecha.Free;
  bmArriba.Free;
  bmAbajo.Free;
```

```
fAvion.Brush.Bitmap.Free;
end;
```

Como puede observar se han cambiado también las propiedades *"Transparent"* y *"TransparentColor"* de las imágenes. Se ha procedido así para que al dibujar el avión no se dibujen las zonas del mismo que están en blanco (white).

Como ya se dijo, el movimiento del avión será controlado con las teclas de los cursores y como las mismas son parte del teclado extendido, debemos trabajar con el evento *"onKeyDown"* de la forma. Este evento recibe dos parámetros: *"Key"* que es un valor entero que contiene el código de la tecla pulsada y *"ShiftState"*, que es un conjunto que puede contener los siguientes elementos: *"ssShift"*, si la tecla *"Shift"* está presionada, *"ssAlt"* si la tecla *"Alt"* está presionada, *"ssCtrl"* si la tecla *"Ctrl"* está presionada, *"ssLeft"* si el botón izquierdo del mouse está presionado, *"ssRight"* si el botón derecho del mouse está presionado, *"ssMidle"* si el botón medio del mouse está presionado y *"ssDouble"* si se ha hecho doble clic con el mouse.

Los códigos *"Key"* de prácticamente todas las teclas están declarados como constantes en *Windows*. Algunas de dichas constantes han sido declaradas también en *Delphi* y comienzan con las letras: *"VK_"*, así la tecla de desplazamiento derecho es: *"VK_RIGHT"*, la tecla *"F10"* es: *"VK_F10"*, el número 1 del teclado numérico es: *"VK_NUMPAD1"*, etc. Para las teclas alfanuméricas *Delphi* no ha declarado constantes, por lo que se hace referencias a ellas directamente por su código ASCII, así el número 65 identifica a la letra *"A"*, el número 48 al número *"0"*, etc.

En nuestra aplicación la figura se moverá 5 píxeles en la dirección de la tecla de desplazamiento pulsada y para llevar la figura a su posición inicial se pulsa la tecla *"Home"* (inicio), antes de programar el evento no olvide incorporar la unidad *"uAvion"* (File -> Use Unit -> uAvion):

```
procedure TfAvion.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  case Key of
    VK_HOME: MoverAvion(fAvion,bmDerecha,origen);
    VK_LEFT: MoverAvion(fAvion,bmIzquierda,izquierda);
    VK_RIGHT: MoverAvion(fAvion,bmDerecha,derecha);
    VK_UP: MoverAvion(fAvion,bmArriba,arriba);
    VK_DOWN: MoverAvion(fAvion,bmAbajo,abajo);
  end;
end;
```

Finalmente para que el avión aparezca en su posición inicial cuando aparece la forma se programa el evento *"onActivate"*:

```
procedure TfAvion.FormActivate(Sender: TObject);
begin
  MoverAvion(fAvion,bmDerecha,origen);
end;
```

La aplicación en ejecución (F9) tiene la apariencia de la figura:



6.2. Ejercicios

1. Elabore una aplicación que empleando la estructura *CASE* determine si un número entero es positivo, negativo o cero. La aplicación debe estar centrada en la pantalla, tener su propio icono y su fondo debe estar con una imagen.
2. Elabore una aplicación que empleando la estructura *CASE* convierta un número entero comprendido entre 1 y 7 en el día de la semana respectivo. El día debe seleccionarse de un "ComboBox", el fondo de la pantalla debe ser de color azul, con líneas horizontales, contar con su propio icono y estar centrada en la pantalla.
3. Elabore una aplicación que empleando la estructura *CASE*, devuelva el número de días que tiene un mes en un año dado. Para determinar el número de días del mes de febrero debe elaborar un módulo empleando la estructura *case*. La aplicación debe contar con su propio icono, estar centrada en la pantalla, su fondo debe ser de color "SkyBlue" y el mes debe ser elegido de un "ComboBox".
4. Elabore una aplicación que mueva una figura (un auto) en la pantalla empleando las teclas del teclado numérico: 4: izquierda; 6: derecha; 8: arriba; 2: abajo; 9: Diagonal arriba-derecha; 7: Diagonal arriba-izquierda; 3: Diagonal abajo-derecha; 1: Diagonal abajo-izquierda; 5: Posición inicial.

7. ESTRUCTURA WHILE

Para resolver problemas, no son suficientes las estructuras secuencial y selectiva. En muchos problemas es necesario repetir una serie de acciones un determinado número de veces o hasta que se cumpla una determinada condición, cuando esto ocurre, es decir cuando en la solución de un problema se repite una o más veces una serie de acciones, se dice que la solución es iterativa.

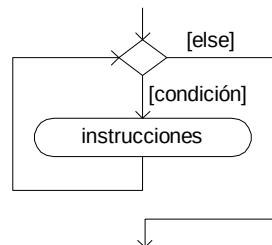
De acuerdo con el teorema de la programación estructurada (propuesto por Böhm y Jacopini), la única estructura necesaria para resolver problemas iterativos es la estructura *mientras* (*while*).

Sin embargo, el uso exclusivo de esta estructura puede complicar innecesariamente la resolución de los problemas. Por esta razón la mayoría de los lenguajes estructurados admiten al menos otras dos estructuras iterativas: *until* (*hasta*) y *for* (*desde*).

Al concluir el estudio de las estructuras iterativas, deberán estar capacitados para resolver problemas iterativos empleando las estructuras más adecuadas para cada caso y siguiendo, en todo momento, los fundamentos de la programación estructurada.

Comenzaremos el estudio de las estructuras iterativas con la estructura *WHILE*. Esta estructura es apropiada para resolver problemas en los cuales no se sabe el número de veces que se repetirá el proceso y donde es probable inclusive que el proceso no deba repetirse ni una sola vez.

El diagrama de actividades de la estructura *WHILE* es el siguiente:



En esta estructura, las instrucciones se repiten *mientras* la condición sea verdadera y terminan cuando la condición es falsa. Si inicialmente la condición es falsa, las instrucciones no se repiten ni una sola vez.

Como todas las estructuras de control estándar, la estructura *WHILE* tiene un solo punto de inicio y uno de finalización.

En pascal, esta estructura se codifica de la siguiente manera:

while condición **do** instrucción;

Como de costumbre, si existe más una instrucción, es decir es una secuencia, las mismas deben estar encerradas entre *begin* y *end*.

Esta estructura se emplea cuando no se sabe el número de veces que debe repetirse el proceso iterativo y es especialmente indicada cuando existe la posibilidad de que dicho proceso no deba repetirse ni una sola vez. Al emplear esta estructura se debe cuidar que la condición, en algún momento, sea falsa, pues de lo contrario el ciclo se repetiría indefinidamente (ciclo infinito).

7.1. Ejemplos

1. Raíz cúbica de un número

Como primer ejemplo elaboraremos un módulo para calcular de la raíz cúbica de un número real "n". Se trata de un procedimiento iterativo en el cual se aplica la siguiente ecuación, deducida con el método de Newton-Raphson:

$$x_2 = \frac{1}{3} \left(2x_1 + \frac{n}{x_1^2} \right)$$

El proceso es el siguiente: se asume un valor inicial para la raíz cúbica (usualmente $x_1=1$), se reemplaza este valor en la ecuación de Newton con lo que se obtiene un nuevo valor para la raíz cúbica (x_2), entonces se comparan estos valores (x_1 con x_2) y si son aproximadamente iguales el proceso concluye (siendo la solución x_2), caso contrario se realiza un cambio de variables ($x_1=x_2$) y se vuelve a calcular un nuevo valor de x_2 . Este proceso se repite hasta que x_1 y x_2 son aproximadamente iguales, o lo que es lo mismo mientras x_1 y x_2 son diferentes.

Para comprender mejor el proceso antes descrito calcularemos manualmente la raíz cúbica de 7: $\sqrt[3]{7}$ (entonces $n=7$).

Comenzamos asumiendo un valor inicial igual a 1 ($x_1=1$) y con este valor calculamos x_2 :

$$x_2 = \frac{1}{3} \left(2 \cdot 1 + \frac{7}{1^2} \right) = 3$$

Puesto que x_1 y x_2 son diferentes, efectuamos un cambio de variables: $x_1=x_2=3$ y volvemos a calcular un nuevo valor de x_2 :

$$x_2 = \frac{1}{3} \left(2 \cdot 3 + \frac{7}{3^2} \right) = 2.259225925926$$

Dado que x_1 y x_2 siguen siendo diferentes se debe repetir el proceso, de esta manera se obtienen los valores que se muestran en la siguiente tabla:

x_1	x_2
1	3
3	2.25925925926
2.25925925926	1.96330801822
1.96330801822	1.91421275416
1.91421275416	1.91293204059
1.91293204059	1.91293118277
1.91293118277	1.91293118277

Como se puede observar en la última fila los valores son iguales, entonces el proceso concluye, siendo la raíz cúbica de 7: $x_2 = 1.91293118277$.

En la práctica el proceso se repite hasta que los dos últimos valores son iguales en un determinado número de dígitos. Para determinar si dos valores: x_1 y x_2 , son iguales en un determinado número de dígitos se emplea la siguiente expresión lógica (condición):

$$\left| \frac{x_1}{x_2} - 1 \right| < 1 \times 10^{-n}$$

Que devuelve verdadero si los valores comparados (x_1 y x_2) son iguales en los primeros n dígitos y falso en caso contrario. Así por ejemplo para determinar si las variables s_1 y s_2 son iguales en los 6 primeros dígitos la condición es:

$$\left| \frac{s_1}{s_2} - 1 \right| < 1 \times 10^{-6}$$

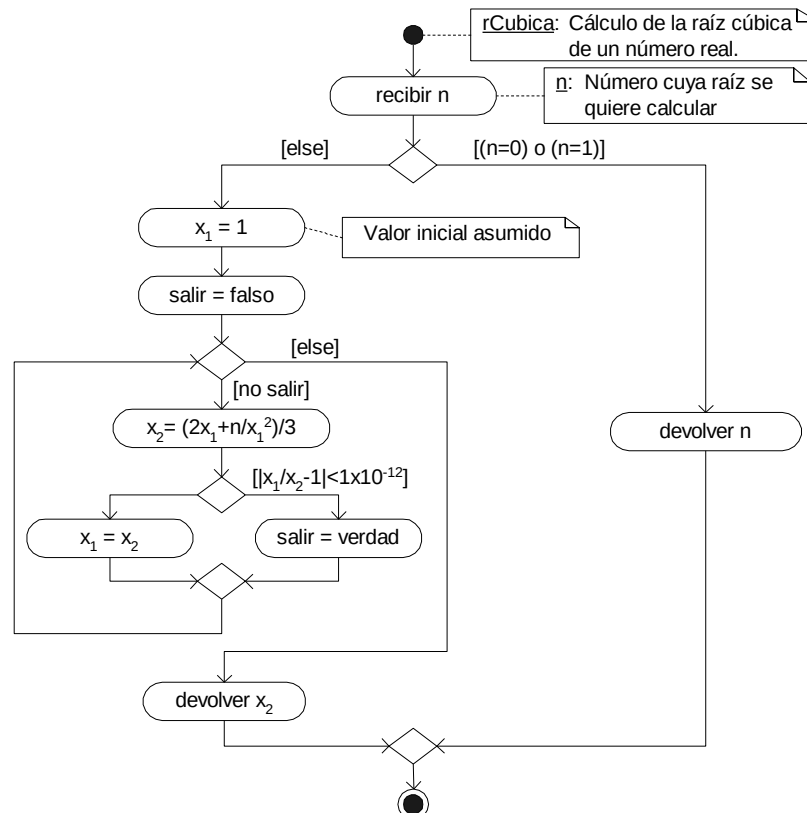
Como en este ejemplo, sucede con frecuencia que la lógica no se ajusta precisamente a ninguna de las estructuras iterativas estándar, por lo que se puede optar por uno de tres caminos: a) Modificar la lógica para que se adapte a la estructura; b) Emplear banderas (interruptores) en lugar de la condición y c) Modificar la estructura estándar.

La tercera opción depende del lenguaje en el que se esté programando y será estudiada en un capítulo posterior. Las dos primeras opciones son siempre posibles sin importar el lenguaje que se emplee y ambas tienen sus ventajas y desventajas: Si se modifica la lógica para adaptarla a una estructura, se pierde claridad, por el contrario si se emplean banderas en lugar de la condición, se modifica en cierto grado la lógica del ciclo.

Dado que el principal objetivo de la programación estructurada es la de facilitar el mantenimiento del programa y a este propósito contribuye una lógica clara, optaremos por la segunda opción.

Las banderas o interruptores, son simplemente variables que pueden tomar uno de dos valores: encendido (verdadero) y apagado (falso).

El algoritmo para resolver el problema, empleando banderas, se presenta en el siguiente diagrama de actividades:



Ahora creemos la aplicación (File -> New Application) y la unidad (File -> New Unit) y guardemos la aplicación (File -> Save Project As...) en el directorio "Iteración\Raíz Cúbica" con los siguientes nombres: "ufRCubica" para "Unit1", "uRCubica" para "Unit2" y "pRCubica" para "Project1".

Y escribamos el código (incluyendo los módulos de lectura y escritura) en la unidad "uRCubica":

```

unit uRCubica;

interface

    uses StdCtrls, SysUtils;

    function LeerNumero(m: tMemo): double;
    function RCubica(n: double): double;
    procedure MostrarRaiz(m: tMemo; r: double);

implementation

    function LeerNumero(m: tMemo): double;
    begin
        result:= StrToFloat(m.Text);
    end;

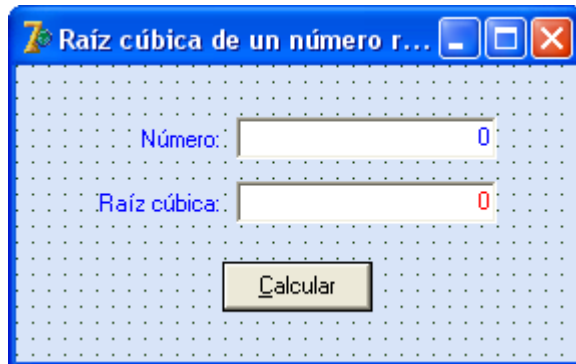
    function RCubica(n: double): double;
    var x1,x2: double; salir: boolean;
    begin
        if (n=0) or (n=1) then result:= n
        else begin
            x1:= 1; salir:= False;
            while not salir do begin
                x2:= (2*x1+n/sqr(x1))/3;
                if abs(x1/x2-1)<1E-12 then salir:= True else x1:= x2;
            end;
            result:= x2;
        end;
    end;

    procedure MostrarRaiz(m: tMemo; r: double);
    begin
        m.Text:= FloatToStr(r);
    end;

end.

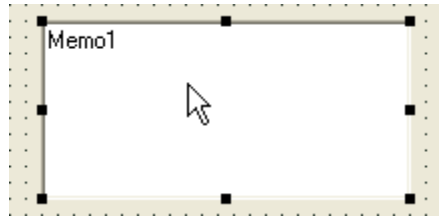
```

Ahora elaborare la interfaz de la aplicación de manera que se vea aproximadamente como se muestra en la figura:



Observe que en este caso el texto está alineado a la derecha, ello se debe a que en esta aplicación se emplean *Memos*, en lugar de *Edits* y los "Memos" permiten alienar el texto a la derecha.

El componente *TMemo* , se encuentran en la página estándar y al ser colocado en la forma tiene la siguiente apariencia:



Que como se puede observar es mayor a la de un "Edit", por lo que su tamaño debe ser modificado. El "Memo" tiene por defecto este tamaño porque se emplea principalmente para mostrar y editar texto que ocupa más de una línea.

Algunas de las propiedades más usuales del *Memo* son: Name: para el nombre del memo; Width: para el ancho del memo; Height: para el alto del memo; Lines: para el texto del memo y que es de tipo "TStrings", igual que la propiedad *Items* del *ComboBox* y *RadioGroup*; Text: Todo el texto del memo en forma de una cadena de caracteres, con las líneas separadas por retornos de carro (#13) y saltos de línea (#10); Alignment: que es la razón por la cual estamos empleando el *Memo* en esta aplicación y que permite alinear el texto a la izquierda "taLeftJustify", al centro "taCenter" y a la derecha "taRightJustify"; WantReturns: que cuando está en *True* (que es su valor por defecto) añade nuevas líneas al pulsar la tecla *Enter* y cuando está en *False* no; MaxLength: que limita el número de caracteres que se puede escribir en el memo, correspondiendo el número 0 a un número ilimitado de caracteres.

Las propiedades modificadas en esta aplicación son:

```
Form1: Name = 'fRaiz'; Caption = 'Raíz cúbica de un número real'; Color =
clInactiveCaptionText; Height = 182; Width = 288; Position = poScreenCenter.
```

```
Label1: Caption = 'Número'; Font.Color = clBlue; Transparent = True.
```

```
Label2: Caption = 'Raíz Cúbica'; Font.Color = clBlue; Transparent = True.
```

```
Memo1: Name = 'mNumero'; Alignment = taRightJustify; Font.Color = clBlue;
Height = 21; Width = 130, WantReturns = False; MaxLength = 15; Lines = '0'.
```

```
Memo2: Name = 'mRaiz'; Alignment = taRightJustify; Font.Color = clRed;
Height = 21; Width = 130, WantReturns = False; MaxLength = 15; Lines = '0';
ReadOnly = True; TabStop = False;
```

```
Button1: Name = 'bCalcular'; Caption = '&Calcular'; Default = True.
```

En el *Memo1*, al igual que en un *Edit*, se debe restringir la introducción de datos de manera que sólo permita la introducción de números reales. Para ello se programa el módulo correspondiente al evento "onKeyPress":

```
procedure TfRaiz.mNumeroKeyPress(Sender: TObject; var Key: Char);
begin
  case Key of
    #8, '0'..'9', '.', 'E', 'e', '+', '-':
      else Beep; Abort;
  end;
end;
```

Como ya vimos anteriormente, el orden de navegación a través de los componentes se modifica en código con la propiedad "TabOrder", sin embargo, es posible modificar el orden de navegación visualmente, para ello haga click

en la forma con el botón derecho del *mouse* y en el menú emergente elija la opción *Tab Order...* (o vaya al menú *Edit -> Tab Order...*). Entonces en la ventana que aparece ordene los componentes de manera que el orden sea: *Mem1*, *Memo2* y *Button1*. El orden en que se colocan los componentes en esta ventana es el orden en que se puede navegar a través de ellos con la tecla *TAB*. Por supuesto en nuestra aplicación no se puede navegar a través del *Memo2* pues su propiedad *TabStop* ha sido colocada en *False*.

Es recomendable también fijar el "foco". El "foco" determina el componente activo, es decir el componente que está recibiendo los eventos. En este ejemplo, es conveniente que al comenzar la aplicación el foco se encuentre en el componente "*mNumero*". Para asignar el "foco" a un componente se emplea el método *SetFocus*.

Asimismo es conveniente que el texto de "*mNumero*" aparezca seleccionado, para poder introducir directamente nuevos valores sin necesidad de borrar los existentes. El texto se selecciona con el método *SelectAll*.

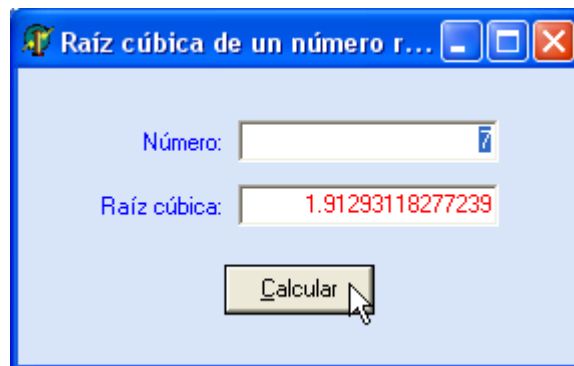
Un lugar conveniente para llamar a estos métodos es el evento *onActivate* de la forma, el mismo que ocurre cuando la forma se activa, es decir, cuando la forma recibe el "foco":

```
procedure TfRaiz.FormActivate(Sender: TObject);
begin
    Mem1.SetFocus;
    Mem1.SelectAll;
end;
```

Finalmente programamos el evento *OnClick* del botón "bCalcular":

```
procedure TfRaiz.bCalcularClick(Sender: TObject);
var n,r: real;
begin
    n:= LeerNumero(mNumero);
    r:= RCubica(n);
    MostrarRaiz(mRaiz,r);
    mNumero.SetFocus;
    mNumero.SelectAll;
end;
```

Como se puede observar, una vez mostrado el resultado "r" en el memo "*mRaiz*", se devuelve el *foco* al *Mem1* y se selecciona el texto del mismo para permitir la introducción directa de nuevos valores. No olvide guardar frecuentemente las modificaciones hechas (File -> Save o File -> Save All). Haciendo correr la aplicación deberá obtener una interfaz similar a la siguiente:



2. Cálculo del Fibonacci

Como segundo ejemplo calcularemos el *Fibonacci* de un número entero. El *Fibonacci* de un número "n" se calcula mediante la siguiente ecuación:

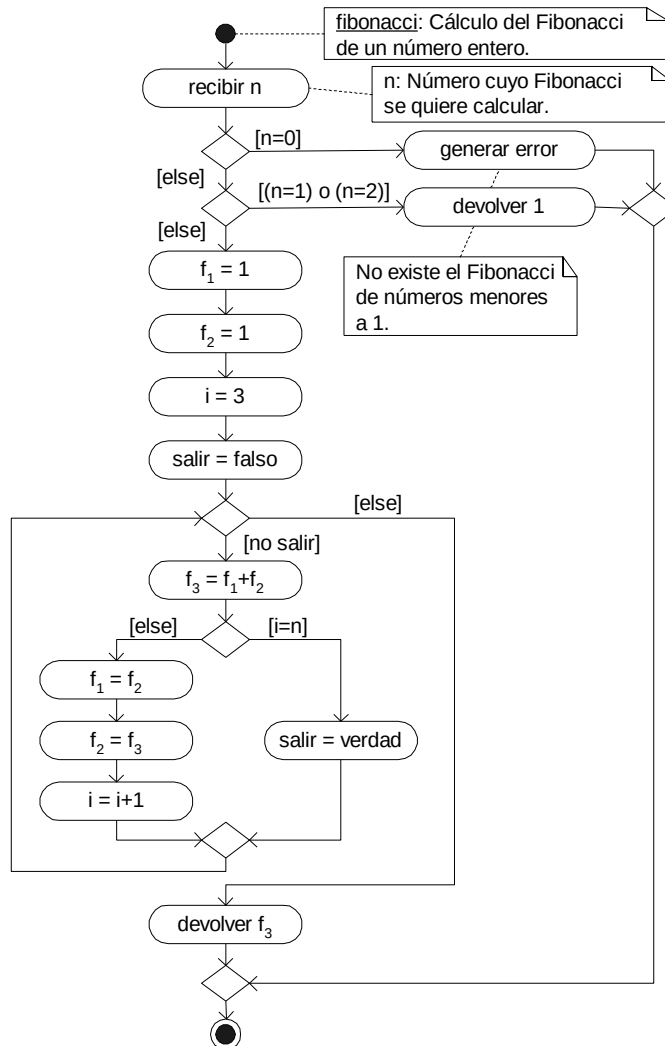
$$F_n = F_{n-1} + F_{n-2}$$

Donde por definición $F_1 = F_2 = 1$.

Para números mayores a 2 el cálculo es iterativo: con los valores conocidos del *Fibonacci* (F_1 y F_2) se calcula el *Fibonacci* de 3, con el *Fibonacci* de 3 se calcula el *Fibonacci* de 4, con el *Fibonacci* de 4 se calcula el *Fibonacci* 5 y así sucesivamente hasta llegar al número cuyo *Fibonacci* se quiere calcular. Por ejemplo, para calcular el *Fibonacci* de 8, se llevan a cabo las siguientes operaciones:

$F_3 = F_2 + F_1 = 1 + 1 = 2;$
 $F_4 = F_3 + F_2 = 2 + 1 = 3;$
 $F_5 = F_4 + F_3 = 3 + 2 = 5;$
 $F_6 = F_5 + F_4 = 5 + 3 = 8;$
 $F_7 = F_6 + F_5 = 8 + 5 = 13;$
 $F_8 = F_7 + F_6 = 13 + 8 = 21;$

Es decir repetimos el proceso 6 veces, empleando en cada repetición los valores calculados en la iteración anterior. El algoritmo es el siguiente:



Ahora creemos la aplicación (File -> New Application) y la unidad (File -> New Unit) y guardemos la aplicación (File -> Save Project As...) en el

directorio "Iteración\Fibonacci" con los siguientes nombres: "ufFibonacci" para "Unit1", "uFibonacci" para "Unit2" y "pFibonacci" para "Project1".

Escribimos el código (incluyendo los módulos de lectura y escritura) en la unidad "uFibonacci":

```
unit uFibonacci;

interface

  uses StdCtrls, SysUtils, Math;

  function LeerNumero(m: TMemo): word;
  function Fibonacci(n: word): Extended;
  procedure MostrarFibonacci(m: TMemo; f: Extended);

implementation

  function LeerNumero(m: TMemo): word;
  begin
    result:= StrToInt(m.Text);
  end;

  function Fibonacci(n: word): Extended;
  var f1,f2,f3: Extended; i: word; salir: boolean;
  begin
    case n of
      0 : raise EInvalidArgument.Create('El número debe ser mayor a cero');
      1,2 : result:= 1;
    else
      f1:=1; f2:=1; i:=3; salir:= False;
      while not salir do begin
        f3:= f1+f2;
        if i=n then salir:= True else begin f1:= f2; f2:= f3; inc(i); end;
      end;
      result:= f3;
    end;
  end;

  procedure MostrarFibonacci(m: TMemo; f: Extended);
  begin
    m.Text:= FloatToStr(f);
  end;

end.
```

Observe que el resultado del módulo "Fibonacci" es de tipo real (extended), cuando debería ser de tipo entero, pues todas las operaciones involucran números enteros. Se ha elegido un tipo real porque el resultado supera fácilmente el límite de los enteros, inclusive del entero más grande: Int64, así el Fibonacci de 300 es aproximadamente: $2.2223224462942 \times 10^{62}$, que está muy por encima del límite de Int64 (aproximadamente 1.9×10^{19}).

Observe también que existen dos aspectos nuevos: a) Los datos se leen y muestran en un objeto de tipo "TMemo" (el cual ya hemos descrito previamente) y b) Se genera un error con el comando "raise".

El comando "raise", interrumpe la ejecución del código y continúa en el primer bloque de tratamiento de errores (el bloque "try - except", que será estudiado más adelante). Si en el módulo no existe ningún bloque de tratamiento de errores (como en el ejemplo), entonces sale del módulo y busca un

bloque de tratamiento de errores en el módulo que hizo la llamada, continuando así hasta que encuentra un bloque de tratamiento de errores o hasta que llega al primer módulo que inició el proceso, en cuyo caso muestra una ventana con un mensaje de error.

El comando "raise" tiene el siguiente formato:

```
raise Clase_de_error.Create('Mensaje');
```

Donde *Clase_de_error* es la "clase" de error a generar, existiendo diferentes clases para diferentes tipos de errores, algunas de ellas son: "EDivByZero" que se genera cuando se divide un entero entre cero; "EZeroDivide" que es igual al anterior pero con reales; "EInvalidOp" que se genera cuando se realiza una operación no válida como por ejemplo multiplicar dos cadenas; "EOverflow", que se genera cuando un valor sobrepasa el límite permitido para un determinado tipo de dato; "EUnderFlow", igual al anterior pero cuando se sobrepasa el límite inferior; "EIntOverflow", igual que *EOverflow* pero para números enteros; "ERangeError", cuando el valor sobrepasa por exceso o por defecto los valores permitidos (generalmente con enteros); "EInvalidArgument" cuando los datos que se mandan a un procedimiento o función no son correctos, etc.

En nuestro caso por ejemplo la clase de error es "EInvalidArgument" y el mensaje 'El número debe ser mayor a cero'. Este error se genera cuando el parámetro que se manda al módulo es "0", que es un parámetro (o argumento) erróneo, pues no existe el Fibonacci de "0".

Como puede observar, todas las clases de error comienzan con la letra E. Existe una variedad de errores predefinidos en diferentes unidades, así la mayoría de los errores antes mencionados se encuentran en la unidad *SysUtils*, mientras que "EInvalidArgument" se encuentra en la unidad "Math" (siendo esta la razón por la cual se incluye esta unidad en el código). Recordemos que para averiguar en qué unidad se encuentra un determinado error (o cualquier otro elemento de Delphi), se debe emplear la ayuda de *Delphi*.

Cuando se generan errores, el programa se comporta de diferente forma si se hace correr la aplicación desde Delphi o directamente desde Windows (haciendo doble clic en el icono de la aplicación). Si se hace correr el programa desde *Delphi*, aparece una ventana con un mensaje (no el mensaje escrito en *raise*), informando que se ha producido una excepción (error), entonces al hacer click en aceptar *Delphi* detiene la ejecución del programa y muestra el lugar del código donde se ha generado el error. Para continuar la ejecución del programa se debe pulsar la tecla F9, entonces recién aparece la ventana con el mensaje escrito. Por el contrario, si se hace el programa desde *Windows*, aparece directamente la ventana con el mensaje de error escrito.

Ahora elaboraremos la interfaz de la aplicación de manera que se vea aproximadamente como se muestra en la figura de la siguiente página.

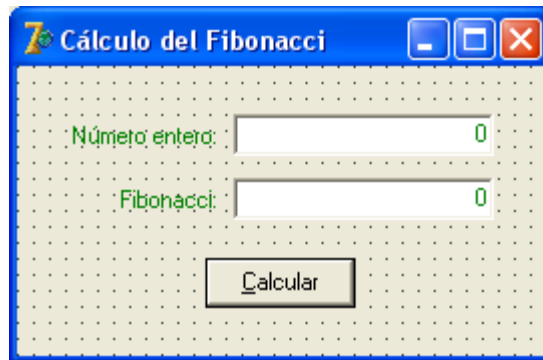
Esta interfaz no tiene componentes nuevos, siendo las propiedades modificadas:

```
Form1: Name = 'fFibonacci'; Caption = 'Cálculo del Fibonacci'; Height = 179; Width = 272; Position = poScreenCenter.
```

```
Label1: Caption = 'Número entero:'; Label2: Caption = 'Fibonacci:'; propiedades comunes: Font.Color = clGreen; Transparent = True.
```

```
Memor1: Name = 'mNumero'; Memor2: Name = 'mFibonacci'; ReadOnly = True; TabStop = False; Propiedades comunes: Alignment = taRightJustify; Font.Color = clGreen; Height = 21; Width = 130, WantReturns = False; MaxLength = 15; Lines = '0'.
```

Button1: Name = 'bCalcular'; Caption = '&Calcular'; Default = True.



En esta aplicación cambiaremos el estilo de la brocha a "bsFDiagonal" en el evento "onCreate" de la forma:

```
procedure TfFibonacci.FormCreate(Sender: TObject);
begin
    fFibonacci.Brush.Color:= clHighLight;
    fFibonacci.Brush.Style:= bsFDiagonal;
end;
```

Al igual que en la anterior aplicación nos aseguramos que el orden de navegación (*Edit* -> *Tab Order*) sea: *mNumero*, *mFibonacci* y *bCalcular*. Asignamos el foco a *mNumero* y seleccionamos el texto del mismo, programando el evento "onActivate" de la forma:

```
procedure TfFibonacci.FormActivate(Sender: TObject);
begin
    mNumero.SetFocus;
    mNumero.SelectAll;
end;
```

Validamos la introducción de datos al memo "mNumero", para que sólo permita números enteros positivos, programando el evento "onKeyPress":

```
procedure TfFibonacci.mNumeroKeyPress(Sender: TObject; var Key: Char);
begin
    if not (Key in [#8,'0'..'9']) then begin Beep; Abort; end;
end;
```

Finalmente damos funcionalidad a la aplicación programando el evento "onClick" del botón "bCalcular", recuerde importar previamente la unidad uFibonacci (File -> Use Unit... -> uFibonacci).

```
procedure TfFibonacci.bCalcularClick(Sender: TObject);
var n: word; f: Extended;
begin
    n:= LeerNumero(mNumero); f:= Fibonacci(n); MostrarFibonacci(mFibonacci,f);
    mNumero.SetFocus; mNumero.SelectAll;
end;
```

En todas las aplicaciones no olvide guardar frecuentemente las modificaciones hechas (File -> Save). Finalmente compilamos la aplicación (Ctrl+F9), corregimos los errores y hacemos correr la aplicación (F9). Comprobamos que la lógica sea correcta haciendo correr el programa con el número "8", para el cual deberemos obtener el resultado "21" y con el número "50" para el cual el resultado es "12586269025".

3. Seno hiperbólico

Como tercer ejemplo se calculará el seno hiperbólico de un número real. La serie de Taylor que nos permite calcular el seno hiperbólico es la siguiente:

$$\sinh(x) = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} + \dots = \sum_{i=1,3,5,\dots}^{\infty} \frac{x^i}{i!}$$

Este es un cálculo iterativo donde en cada iteración se añade un nuevo término a la serie. Debido a que en la práctica los cálculos no pueden ser efectuados hasta el infinito, se añaden nuevos términos *mientras* los dos últimos valores calculados difieren en un determinado número de dígitos.

Entonces en cada iteración es necesario calcular el valor del término: $(x^i/i!)$. Se podría pensar en calcular este nuevo valor elevando en cada iteración el número real a la potencia entera respectiva y dividiendo entre el factorial del número, es decir:

$$\frac{x^i}{i!} = \frac{x * x * x * x * \dots i \text{ veces}}{1 * 2 * 3 * 4 * \dots * i}$$

De esta manera se logra calcular el resultado correcto, pero la lógica es errónea pues se vuelven a calcular innecesariamente una y otra vez los mismos valores. Para comprender el por qué este planteamiento es ilógico supongamos por ejemplo que se está calculado el término $x^{11}/11!$:

$$\frac{x^{11}}{11!} = \frac{x * x * x * x * x * x * x * x * x * x * x}{1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 * 9 * 10 * 11}$$

Entonces en la siguiente iteración se calculará el término $x^{13}/13!$:

$$\frac{x^{13}}{13!} = \frac{x * x * x * x * x * x * x * x * x * x * x * x * x}{1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 * 9 * 10 * 11 * 12 * 13}$$

Como se puede observar, al calcular $x^{13}/13!$ se vuelve a calcular el valor de $x^{11}/11!$:

$$\frac{x^{13}}{13!} = \left(\frac{x * x * x * x * x * x * x * x * x * x * x}{1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 * 9 * 10 * 11} \right) \left(\frac{x * x}{12 * 13} \right) = \left(\frac{x^{11}}{11!} \right) \left(\frac{x^2}{12 * 13} \right)$$

Lo cual es ilógico porque ¿para qué se vuelve a calcular un valor que ya ha sido calculado previamente? Por supuesto este problema se repite desde los primeros términos, así, al calcular $x^{11}/11!$ se vuelve a calcular $x^9/9!$; al calcular $x^9/9!$ se vuelve a calcular $x^7/7!$ y así sucesivamente. Proceso que como se ve es extremadamente ineficiente pues se vuelven a calcular una y otra vez los mismos valores, así cuando se llega al término $x^{21}/21!$, se habrá calculado 10 veces el valor de $x^3/3!$.

Lo lógico sería utilizar los valores calculados para calcular los nuevos, así por ejemplo, si ya se ha calculado el término $x^9/9!$, el nuevo término se calcula con:

$$\frac{x^{11}}{11!} = \frac{x^9}{9!} * \frac{x^2}{10 * 11}$$

Y una vez calculado $x^{11}/11!$, el nuevo término se calcula con:

$$\frac{x^{13}}{13!} = \frac{x^{11}}{11!} * \frac{x^2}{12 * 13}$$

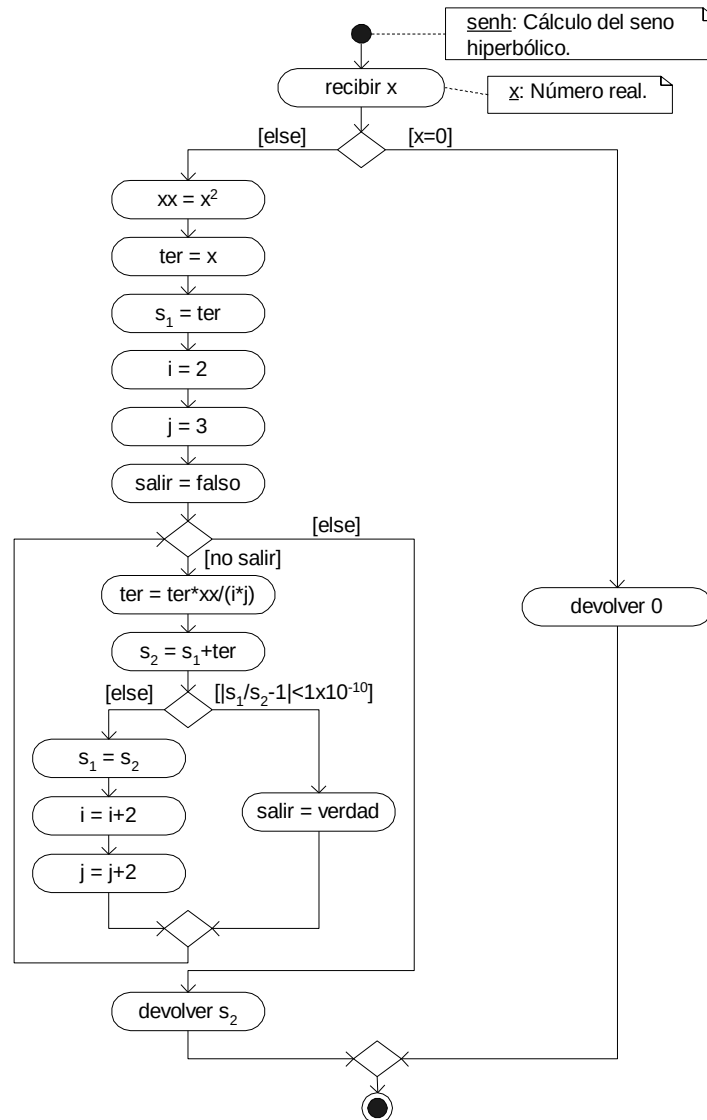
En general entonces se puede calcular el nuevo término multiplicando el término anterior por x^2 y dividiendo entre 2 contadores que deberán incrementar de 2 en 2 en cada iteración:

$$\text{nuevo término} = \text{término anterior} * \frac{x^2}{i * j}$$

Antes de comenzar el proceso iterativo se debe verificar que el número sea diferente de cero, por dos razones: a) Se sabe que el seno hiperbólico de cero es cero y b) Si "x" es cero, entonces "s₂" es cero, lo que da lugar a un error por división entre cero al evaluar la condición:

$$\left| \frac{s_1}{s_2} - 1 \right| < 1 \times 10^{-n}$$

Puesto que una vez más la lógica no corresponde exactamente a la estructura "mientras", la resolvemos empleando banderas:



Observe que el resultado se calcula con 10 dígitos de precisión y que se emplea una variable temporal "xx" para guardar el cuadrado de "x", pues de

lo contrario este valor sería calculado una y otra vez, de manera ilógica, cada vez que se repite el ciclo.

La lógica empleada para resolver este problema es la que se sigue para resolver la mayoría de las series: a) inicializar variables; b) calcular el nuevo término; c) calcular el nuevo valor de la sumatoria; d) comparar los dos últimos valores calculados, si son aproximadamente iguales el proceso concluye siendo la respuesta el último valor calculado, caso contrario se intercambia variables, se incrementa contadores y se repite el proceso desde el paso (b).

Ahora creemos la aplicación (File -> New Application), la unidad (File -> New Unit) y guardemos la aplicación (File -> Save Project As...) en el directorio "Iteración\Seno hiperbólico" con los siguientes nombres: "ufSenh" para "Unit1", "uSenh" para "Unit2" y "pSenh" para "Project1". Y escribamos el código (incluyendo los módulos de lectura y escritura) en la unidad "uSenh":

```
unit uSenh;

interface

    uses StdCtrls, SysUtils;

    function LeerNumero(m: TMemo): real;
    function Senh(x: real): real;
    procedure MostrarSenh(m: TMemo; s: real);

implementation

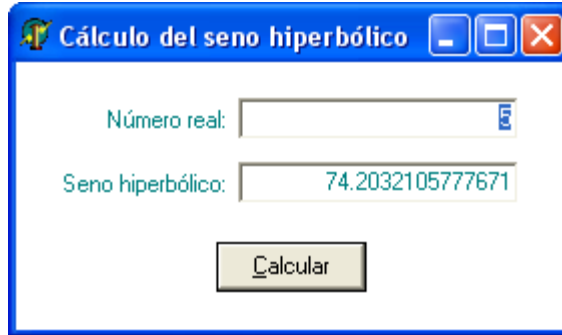
    function LeerNumero(m: TMemo): real;
    begin
        result := StrToFloat(m.Text);
    end;

    function Senh(x: real): real;
    var xx, ter, s1, s2: real; i, j: word; salir: boolean;
    begin
        if x=0 then result := 0
        else
            begin
                xx := sqr(x); ter := x; s1 := ter; i := 2; j := 3;
                salir := False;
                while not salir do
                    begin
                        ter := ter*xx/i/j;
                        s2 := s1+ter;
                        if abs(s1/s2-1)<1E-10 then salir := True
                        else begin s1 := s2; inc(i,2); inc(j,2); end;
                    end;
                result := s2;
            end;
        end;

    procedure MostrarSenh(m: TMemo; s: real);
    begin
        m.Text := FloatToStr(s);
    end;

end.
```

Ahora elaboraremos la interfaz de la aplicación de manera que el ejecutable se vea aproximadamente como se muestra en la figura:



Las propiedades modificadas son:

Form1: Name = 'fSenh'; Caption = 'Cálculo del seno hiperbólico'; Height = 168; Width = 283; color = clWhite; Position = poScreenCenter.

Label1: Caption = 'Número real:'; Label2: Caption = 'Seno hiperbólico:'; propiedades comunes: Font.Color = clTeal; Transparent = True.

Memo1: Name = 'mNumero'; Memo2: Name = 'mSenh'; ReadOnly = True; TabStop = False; Propiedades comunes: Alignment = taRightJustify; Font.Color = clTeal; Height = 21; Width = 130, WantReturns = False; MaxLength = 15; Lines = '0'.

Button1: Name = 'bCalcular'; Caption = '&Calcular'; Default = True.

Se programan además los eventos "onActivate" de la forma y "onKeyPress" del memo:

```
procedure TfSenh.FormActivate(Sender: TObject);
begin
    mNumero.SetFocus; mNumero.SelectAll;
end;
```

```
procedure TfSenh.mNumeroKeyPress(Sender: TObject; var Key: Char);
begin
    if not (key in [#8,'0'..'9','.', '+', '-', 'E', 'e']) then
        begin Beep; Abort; end;
end;
```

Finalmente damos funcionalidad a la aplicación programando el evento "onClick" del botón bCalcular:

```
procedure TfSenh.bCalcularClick(Sender: TObject);
var x: real;
begin
    x:= LeerNumero(mNumero); x:= Senh(x); MostrarSenh(mSenh,x);
    mNumero.SetFocus; mNumero.SelectAll;
end;
```

Compilando (Ctrl+F9), corrigiendo errores y haciendo correr la aplicación (F9), deberá obtener el resultado que se muestra en la figura.

7.2. Ejercicios

1. Elabore una aplicación y escriba el evento "onCreate" de la misma de manera que la forma se encuentre centrada en la pantalla, que la forma tenga una figura en su fondo, que el título de la forma sea "ESTRUCTURA WHILE" y que tenga su propio icono.

2. Elabore una aplicación y escriba el evento "onCreate" de la misma de manera que el fondo de la forma sea de color verde con líneas diagonales cruzadas, que tenga el título "Seno de un ángulo en radianes", que esté centrada en la pantalla y tenga su propio icono.
3. Elabore una aplicación con un RadioGroup y en el evento "onCreate" de la forma establezca el título del RadioGroup en "Ángulo en:", añada las opciones "Radianes" y "Grados", seleccione la opción "Grados" y fije el número de columnas del RadioGroup en 2.
4. Elabore una aplicación con un RadioGroup con dos opciones: "Radianes" y "Grados" y un BitBtn con el título "Calcular". Programe el evento "onClick" del BitBtn para que calcule y muestre en el lienzo el seno y coseno de 456, siendo las unidades fijadas por la opción elegida en el RadioGroup.
5. Elabore una aplicación con un memo y valide la introducción de datos al mismo de manera que sólo permita escribir números reales.
6. Elabore una aplicación con un memo y valide la introducción de datos al mismo de manera que sólo permita introducir las letras de la "A" a la "z" (incluida la "Ñ") tanto en mayúsculas como en minúsculas.
7. Elabore una aplicación con un memo y en el evento "onCreate" fije el ancho del memo en 140 puntos, el alto en 21 puntos, la alineación a la derecha, la longitud máxima del texto en 20 caracteres, impida la adición de nuevas líneas con la tecla "Enter", establezca el texto del memo en su nombre y apellido, siendo la letra "Times New Roman" en color amarillo sobre fondo azul y con el estilo cursiva.
8. Elabore una aplicación con un memo y en la misma escriba un módulo que reciba como datos un número entero comprendido entre 3×10^{20} y -3×10^{20} y un memo y muestre en el memo el valor del número entero recibido en color rojo sobre fondo verde, con el tipo de letra "Arial" con los estilos negrita y cursiva. Pruebe el módulo en el evento "onDblClick" del memo con el número 295482489324.
9. Elabore una aplicación con un memo y en la misma escriba un módulo que reciba como datos un número real, comprendido entre 10^{-250} y 10^{250} , y un memo y muestre en el memo el número real recibido, en color amarillo sobre fondo azul, con el tipo de letra "Monotype Corsiva", con 10 puntos, con los estilos Negrita y Subrayado. Pruebe el módulo en el evento "onClick" del memo con el número -42093.2324.
10. Elabore una aplicación con un memo y añada al mismo, en el evento "onCreate", los meses del año.
11. Elabore una aplicación y en la misma cree un módulo que empleando la estructura "mientras" invierta los dígitos de un número entero positivo o negativo, así si el número es 12345 debe devolver 54321. Debe probar el módulo en el evento "onDblClick" de la forma con los números 12345, 04533, -392834 y 7345000, mostrando los resultados en el lienzo (canvas) de la forma.
12. Elabore una aplicación y en la misma cree un módulo que empleando la estructura "mientras" calcule el seno de un ángulo con la serie:

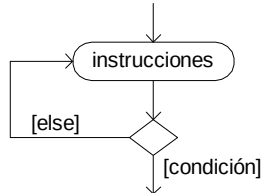
$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$, $-\infty < x < \infty$, donde el ángulo está en radianes, en el módulo se deben convertir los ángulos mayores a 2π a su equivalente comprendido entre 0 y 2π . Debe probar el módulo en el evento "onContextPopup" de la forma con los ángulos en radianes: 1.324, 654.343, 189423.423 y -324544.3234 mostrando los resultados obtenidos con la función elaborada y la función "sin" de Delphi en el lienzo (canvas) de la forma.

13. Elabore una aplicación y en la misma cree un módulo que empleando la estructura "mientras" calcule el Chebyshev enésimo de un número real x , con: $Ch_{n+1}(x) = 2xCh_n(x) - Ch_{n-1}(x)$; $Ch_0(x) = 1$; $Ch_1(x) = x$. Debe probar el módulo en el evento "onClick" de la forma con los números: $n=0$, $x=4.5$; $n=1$, $x=6.43$; $n=2$, $x=7.98$; $n=5$, $x=9.23$, mostrando los resultados obtenidos en el lienzo de la forma.
14. Elabore una aplicación y en la misma cree un módulo que empleando la estructura "mientras" calcule la raíz cuadrada de un número real "n" con la fórmula de Newton: $x_2 = (x_1 + n/x_1)/2$ (el número debe generar un error en caso de que el número sea negativo). Pruebe el módulo en el evento "onKeyPress" de la forma de manera que al pulsar la tecla "C" calcule y muestre las raíces cuadradas de 2, 3.24, 6.534 y al pulsar la tecla "I" calcule y muestre la raíz cuadrada de -3.45.

8. ESTRUCTURA UNTIL

La estructura *UNTIL* es el complemento de la estructura *WHILE*, en esta estructura el proceso iterativo se repite *hasta* que una condición se hace verdadera, por lo tanto resulta más adecuada cuando el proceso iterativo tiene que repetirse por lo menos una vez.

El diagrama de actividades de esta estructura es el siguiente:



Como se puede observar en esta estructura la instrucciones se repiten *hasta* que la *condición* es verdadera.

El código de esta estructura en *Pascal* es:

***repeat* instrucciones *until* condición *end*;**

Donde en este caso las instrucciones no requieren estar encerradas entre *begin* y *end* (pues ya está encerrada entre *repeat* y *until*).

8.1. Ejemplos

1. Raíz cuadrada de un número

Como primer ejemplo de "Until" elaboraremos una aplicación para calcular la raíz cuadrada de un número "n" empleando la ecuación de *Newton*:

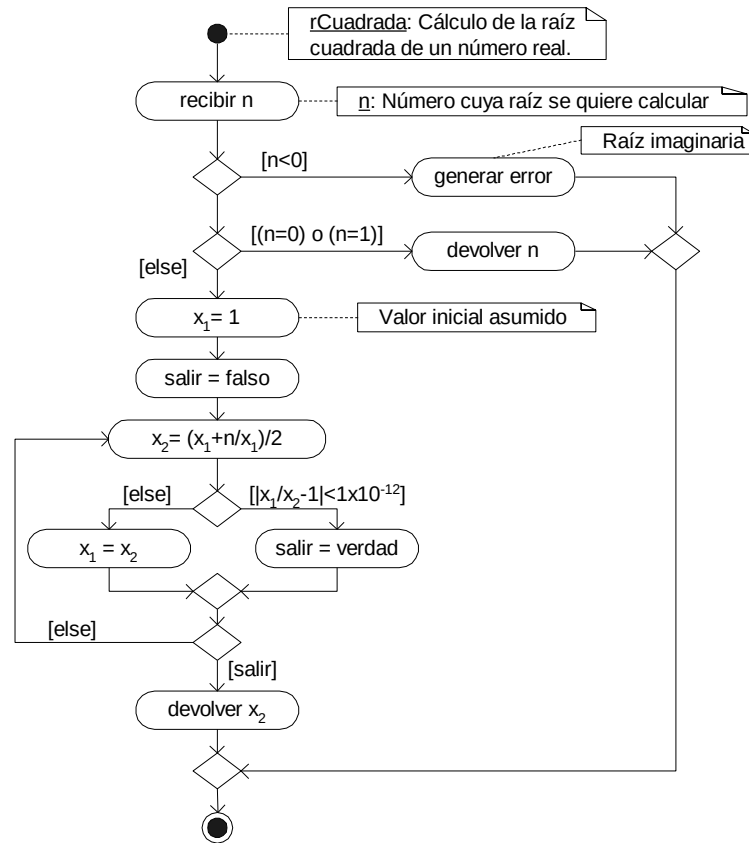
$$x_2 = \frac{1}{2} \left(x_1 + \frac{n}{x_1} \right)$$

El algoritmo para resolver esta ecuación es esencialmente el mismo que para la raíz cúbica: a) Se asume un valor inicial para x_1 (generalmente 1); b) Con x_1 se calcula x_2 ; c) Se compara x_1 con x_2 y si son aproximadamente iguales el proceso concluye siendo la solución x_2 , caso contrario x_1 toma el valor de x_2 ($x_1 = x_2$) y se repite el proceso desde el paso (b).

Una vez más la lógica no se ajusta precisamente a la estructura "hasta" por lo que el problema es resuelto empleando banderas, como se muestra en el diagrama de actividades de la siguiente página.

En el mismo se han tomado en cuenta los casos 0 y 1, para los cuales se sabe que la raíz cuadrada es 0 y 1 respectivamente. También se ha tomado en cuenta el caso para el cual el número es negativo, generándose un mensaje de error.

Ahora creemos la aplicación (File -> New Application), la unidad (File -> New Unit) y guardemos la aplicación (File -> Save Project As...) en el directorio "Iteración\Raíz cuadrada" con los siguientes nombres: "ufRCuadrada" para "Unit1", "uRCuadrada" para "Unit2" y "pRCuadrada" para "Project1". Y escribamos el código (incluyendo los módulos de lectura y escritura) en la unidad "uRCuadrada":



```
unit uRCuadrada;
```

```
interface
```

```
uses StdCtrls, SysUtils;
```

```
function LeerNumero(m: TMemo): double;
```

```
function RCuadrada(n: double): double;
```

```
procedure MostrarRaiz(m: TMemo; r: double);
```

```
implementation
```

```
function LeerNumero(m: TMemo): double;
```

```
begin
```

```
    result:= StrToFloat(m.Text);
```

```
end;
```

```
function RCuadrada(n: double): double;
```

```
var x1,x2: double; c: byte; salir: boolean;
```

```
begin
```

```
    c:= Ord(n<0)+Ord((n=0) or (n=1))*2;
```

```
    case c of
```

```
        1: raise EInvalidOp.Create('Resultado Imaginario');
```

```
        2: result:= n;
```

```
    else
```

```
        x1:= 1; salir:= False;
```

```
        repeat
```

```
            x2:= (x1+n/x1)/2;
```

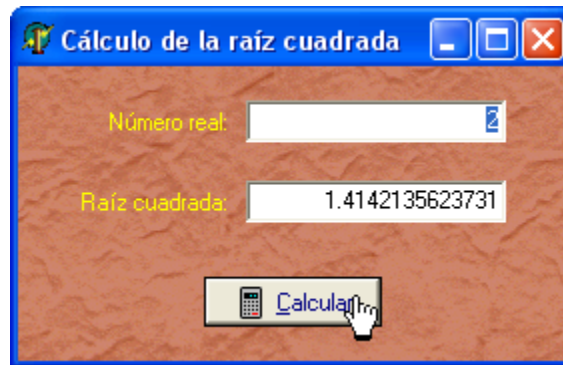
```

        if abs(x1/x2-1)<1E-12 then salir:= True else x1:= x2;
    until salir;
    result:= x2;
end;
end;

procedure MostrarRaiz(m: TMemo; r: double);
begin
    m.Text:= FloatToStr(r);
end;
end.

```

Ahora elaboramos la interfaz de la aplicación de manera que en ejecución tenga la apariencia que se muestra en la siguiente figura:



Observe que en este caso el botón tiene un dibujo, ello se debe a que no es un botón estándar, sino un "BitBtn" (OK), el cual se encuentra en la página "Additional" y que se caracteriza por mostrar una figura a más del título.

Existen dos formas en las que se puede incluir la figura que aparece en el *BitBtn*: a) Seleccionando uno de los botones predefinidos de la propiedad *Kind* y b) Eligiendo una imagen, desde un archivo, con la propiedad *Glyph*. En el primer caso (con la propiedad *Kind*) una vez seleccionado el tipo de botón, se puede cambiar el título con la propiedad *Caption*. En el segundo caso se debe hacer doble clic en el campo de la propiedad *Glyph* (o clic en los tres puntos ubicados a la derecha del campo), en la ventana que aparece se hace clic en el botón "Load...", con lo que aparece el explorador de Windows, el cual permite buscar la imagen a emplear en el botón.

Por defecto los botones que vienen con *Delphi* se encuentran en el directorio: "C:\Archivos de programa\Archivos comunes\Borland Shared\Images\Buttons" y existen muchos otros más que pueden ser bajados de Internet (por ejemplo de *ClubDelphi.com*) o creados en un editor de imágenes como *Paint* o el editor de imágenes de *Delphi*. Los botones son imágenes de tipo *Bitmap* de 16 pixeles de ancho por 16 de alto, si tiene una imagen para un botón o 32, 48 y 64 pixeles de ancho por 16 de alto si tiene dos, tres y cuatro imágenes para un botón respectivamente.

Observe también que el puntero del mouse tiene la forma de una mano con el dedo índice extendido. Ello se consigue cambiando la propiedad "Cursor" del *BitBtn*. "Cursor" es la figura que identifica al puntero del *Mouse*. Todos los componentes visibles tienen la propiedad *Cursor* y en una aplicación es conveniente que el puntero del *mouse* cambie en función del componente (o lugar del componente) que está apuntando, de esa manera se proporciona una ayuda visual adicional al usuario. Por ejemplo cuando el puntero del *mouse* está sobre un botón, es conveniente que el cursor cambie a una mano con el

dedo índice extendido (como en esta aplicación), así queda claro que se puede hacer click sobre el mismo, por el contrario cuando está sobre un objeto de edición, como un memo o un edit, es conveniente que tenga la forma de una barra en forma de "I", que es lo usual en los procesadores de palabras.

Las propiedades modificadas en los componentes son:

```
Form1: Name = 'fRCuadrada'; Caption = 'Cálculo de la raíz cuadrada';
Height = 181; Width = 283; Position = poScreenCenter.
```

```
Label1: Caption = 'Número real:'; Label2: Caption = 'Raíz cuadrada:';
propiedades comunes: Font.Color = clYellow; Transparent = True.
```

```
Memo1: Name = 'mNumero'; Cursor = crIBeam; Memo2: Name = 'mRaiz'; ReadOn-
ly = True; TabStop = False; Cursor = crArrow; Propiedades comunes: Alignment
= taRightJustify; Font.Color = clWindowText; Height = 21; Width = 130,
WantReturns = False; MaxLength = 20; Lines = '0'.
```

```
BitBtn1: Name = 'bbCalcular'; Caption = '&Calcular'; Default = True;
Glyph = "calc.bmp"; Cursor = crHandPoint.
```

En esta aplicación, el fondo de la forma es una imagen: "Santa Fe.bmp". Como de costumbre asignaremos la imagen a la brocha (brush) en el evento "onCreate" y liberaremos la imagen en el evento "onClose":

```
procedure TfRCuadrada.FormCreate(Sender: TObject);
begin
  fRCuadrada.Brush.Bitmap:= TBitmap.Create;
  fRCuadrada.Brush.Bitmap.LoadFromFile('Santa fe.bmp');
end;
```

```
procedure TfRCuadrada.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  fRCuadrada.Brush.Bitmap.Free;
end;
```

Sin embargo, para que este código funcione es necesario que la imagen "Santa fe.bmp" sea copiada en el directorio donde estamos creando la aplicación ("Iteración\Raíz cuadrada")

Validamos también la introducción de datos en el memo "mNumero":

```
procedure TfRCuadrada.mNumeroKeyPress(Sender: TObject; var Key: Char);
begin
  case Key of
    #8, '0'..'9', '.', '-', '+', 'e', 'E':
      else Beep; Abort;
  end;
end;
```

Finalmente damos funcionalidad a la aplicación programando el evento "on-Click" del BitBtn "bbCalcular":

```
procedure TfRCuadrada.bbCalcularClick(Sender: TObject);
var x: double;
begin
  x:= LeerNumero(mNumero); x:= RCuadrada(x); MostrarRaiz(mRaiz,x);
  mNumero.SetFocus; mNumero.SelectAll;
end;
```

Al hacer correr la aplicación, la interfaz deberá tener la apariencia y calcular el resultado que se muestra en la figura de la anterior página.

2. Legendre enésimo de un número real

Como segundo ejemplo calcularemos el *Legendre* enésimo de un número real empleando la ecuación:

$$(n+1)Le_{n+1}(x) - (2n+1)xLe_n(x) + nLe_{n-1}(x) = 0$$

Donde por definición: $Le_0(x)=1$; $Le_1(x)=x$

El cálculo del *Legendre* es similar al del *Fibonacci*: Con los dos valores iniciales conocidos (Le_0 y Le_1) se calcula el tercero (Le_2), con el segundo y el tercero (Le_1 y Le_2) se calcula el cuarto (Le_3) y así sucesivamente hasta llegar al *Legendre* que se quiere calcular.

Sin embargo, antes de elaborar el algoritmo es necesario que la ecuación de *Legendre* sea colocada en una forma parecida a la ecuación de *Fibonacci*, pues de lo contrario el término más alto es "n+1" y no "n", y en esa forma si por ejemplo queremos calcular el *Legendre* de 5 ($n=5$), deberíamos conocer el *Legendre* de 6 ($n+1=5+1$), algo que es incoherente, pues el *Legendre* de 6 se calcula a partir del *Legendre* de 5 y no al revés. A fin de evitar estas incoherencias debemos hacer un cambio de variables para que el término más alto sea Le_n y no Le_{n+1} . Con ese fin despejamos Le_{n+1} :

$$Le_{n+1}(x) = \frac{(2n+1)xLe_n(x) - nLe_{n-1}(x)}{(n+1)}$$

Y efectuamos el cambio de variable: $n'=n+1$:

$$Le_{n'}(x) = \frac{(2n'-1)xLe_{n'-1}(x) - (n'-1)Le_{n'-2}(x)}{n'}$$

Finalmente en lugar de n' escribimos simplemente n y realizamos algunas simplificaciones:

$$Le_n(x) = \left(2 - \frac{1}{n}\right)xLe_{n-1}(x) - \left(1 - \frac{1}{n}\right)Le_{n-2}(x)$$

Ahora que la ecuación está en una forma adecuada (donde el término más alto es Le_n) elaboramos el algoritmo que se muestra en la siguiente página.

Creemos entonces la aplicación (File -> New pplication), la unidad (File -> New Unit) y guardemos la aplicación (File -> Save Project As...) en el directorio "Iteración\Legendre" con los siguientes nombres: "ufLegendre" para "Unit1", "uLegendre" para "Unit2" y "pLegendre" para "Project1". Y escribamos el código (incluyendo los módulos de lectura y escritura) en la unidad "uLegendre":

```
unit uLegendre;
```

```
interface
```

```
uses StdCtrls, SysUtils;
```

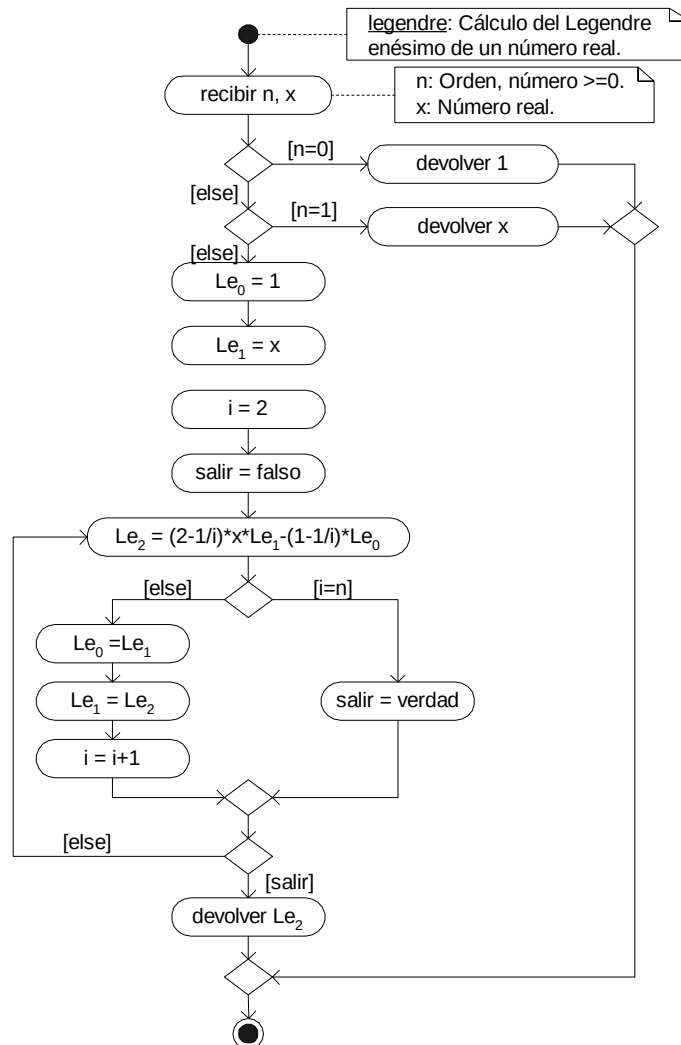
```
function LeerReal(m: TMemo): real;
```

```
function LeerOrden(m: TMemo): byte;
```

```
function legendre(n: byte; x: real): real;
```

```
procedure MostrarLegendre(m: TMemo; l: real);
```

```
implementation
```



```

function LeerReal(m: TMemor): real;
begin
    result:= StrToFloat(m.Text);
end;

```

```

function LeerOrden(m: TMemor): byte;
begin
    result:= StrToInt(m.Text);
end;

```

```

function legendre(n:byte; x:real):real;
var Le0,Le1,Le2: real; i: byte; salir: boolean;
begin
    case n of
        0: result:= 1;
        1: result:= x;
    else
        i:= 2; Le0:= 1; Le1:= x; salir:= False;
        repeat
            Le2:= (2-1/i)*x*Le1-(1-1/i)*Le0;
            if i=n then salir:=True else begin Le0:=Le1;Le1:=Le2;inc(i); end;
        until salir;
    end;

```

```

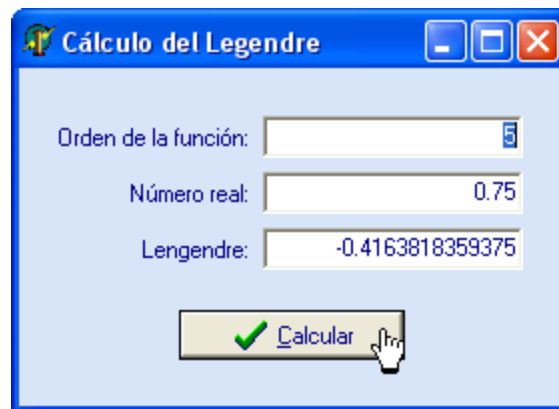
        until salir;
        result:= Le2;
    end;
end;

procedure MostrarLegendre(m: TMemo; l: real);
begin
    m.Text:= FloatToStr(l);
end;

end.

```

Creemos ahora la interfaz de la aplicación de manera que en ejecución tenga la apariencia que se muestra en la siguiente figura:



Las propiedades modificadas de los componentes de esta aplicación son:

Form1: Name = 'fLegendre'; Caption = 'Cálculo del Legendre'; Color = clInactiveCaptionText; Height = 203; Width = 280; Position = poScreenCenter;

Label1: Caption = 'Orden de la función:'; Label2: Caption = 'Número real:'; Label3: Caption = 'Legendre:'; propiedades comunes: Font.Color = clNavy; Transparent = True.

Memo1: Name = 'mOrden'; Cursor = crIBeam; MaxLength = 3; Memo2: Name = 'mNumReal'; Cursor = crIBeam; MaxLength = 20; Memo3: Name = 'mLegendre'; ReadOnly = True; TabStop = False; Cursor = crArrow; Propiedades comunes: Alignment = taRightJustify; Font.Color = clNavy; Height = 21; Width = 130, WantReturns = False; MaxLength = 20; Lines = '0'.

BitBtn1: Name = 'bbCalcular'; Kind = bkOK; Caption = '&Calcular'; Cursor = crHandPoint.

Debemos asegurarnos también que el orden de tabulación (Edit -> Tab Order...) sea: mOrden, mNumReal, mLegendre y bbCalcular.

Antes de programar los eventos recuerde incorporar la unidad "uLegendre" en la unidad "ufLegendre" (File -> Use Unit... -> uLegendre).

Para que al hacerse visible la aplicación aparezca seleccionado el texto del memo "mOrden", programamos el evento "onActivate" de la forma:

```

procedure TfLegendre.FormActivate(Sender: TObject);
begin
    mOrden.SetFocus; mOrden.SelectAll;
end;

```

Para que al pasar el foco al memo "mNumReal", aparezca seleccionado el texto del mismo, programamos el evento "onEnter" del memo (el evento *OnEnter* ocurre cuando un componente recibe el foco):

```
procedure TfLengendre.mNumRealEnter(Sender: TObject);
begin
    mNumReal.SelectAll;
end;
```

Para validar la introducción de datos a los memos "mOrden" y "mNumReal", programamos los evento "onKeyPress" de dichos memos:

```
procedure TfLengendre.mOrdenKeyPress(Sender: TObject; var Key: Char);
begin
    if not (Key in [#8,'0'..'9']) then begin Abort; Beep; end;
end;
```

```
procedure TfLengendre.mNumRealKeyPress(Sender: TObject; var Key: Char);
begin
    if not (Key in [#8,'0'..'9','.', '+', '-', 'e', 'E']) then
        begin Beep; Abort end;
end;
```

Y para dar funcionalidad a la aplicación, programamos el evento "onClick" del BitBtn "bbCalcular":

```
procedure TfLengendre.bbCalcularClick(Sender: TObject);
var n: byte; x,l: real;
begin
    n:= LeerOrden(mOrden); x:= LeerReal(mNumReal); l:= Legendre(n,x);
    MostrarLegendre(mLegendre,l); mOrden.SetFocus; mOrden.SelectAll;
end;
```

Al hacer correr la aplicación, la interfaz deberá tener la apariencia y calcular el resultado que se muestra en la figura de la anterior página.

3. Cálculo del coseno

Como tercer ejemplo elaboraremos una aplicación para calcular el coseno de un número real empleando la serie de *Taylor*:

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots, \quad -\infty < x < \infty$$

Donde "x" es el ángulo en radianes. Como ya se dijo en el ejemplo del seno hiperbólico, la lógica para resolver series es esencialmente la misma: a) inicializar variables; b) calcular el nuevo término; c) calcular el nuevo valor de la sumatoria; d) comparar las dos últimas sumatorias y si son aproximadamente iguales el proceso concluye siendo la respuesta la última sumatoria calculada, caso contrario se intercambia variables, se incrementa contadores y se repite el proceso desde el paso (b).

En esta serie los nuevos términos se calculan multiplicando el término anterior por $-x^2$ y dividiendo entre dos valores sucesivos, así por ejemplo para calcular el cuarto término, en base al tercero, la operación es:

$$\left(\frac{x^4}{4!} \right) \left(\frac{-x^2}{5*6} \right) = -\frac{x^6}{6!}$$

Cuando una serie, como la del presente ejemplo, involucra restas de números muy grandes o muy pequeños se pierden definitivamente dígitos del número

más pequeño. Por lo tanto, siempre que sea posible, tratamos de evitar res-tas de números muy grandes o muy pequeños.

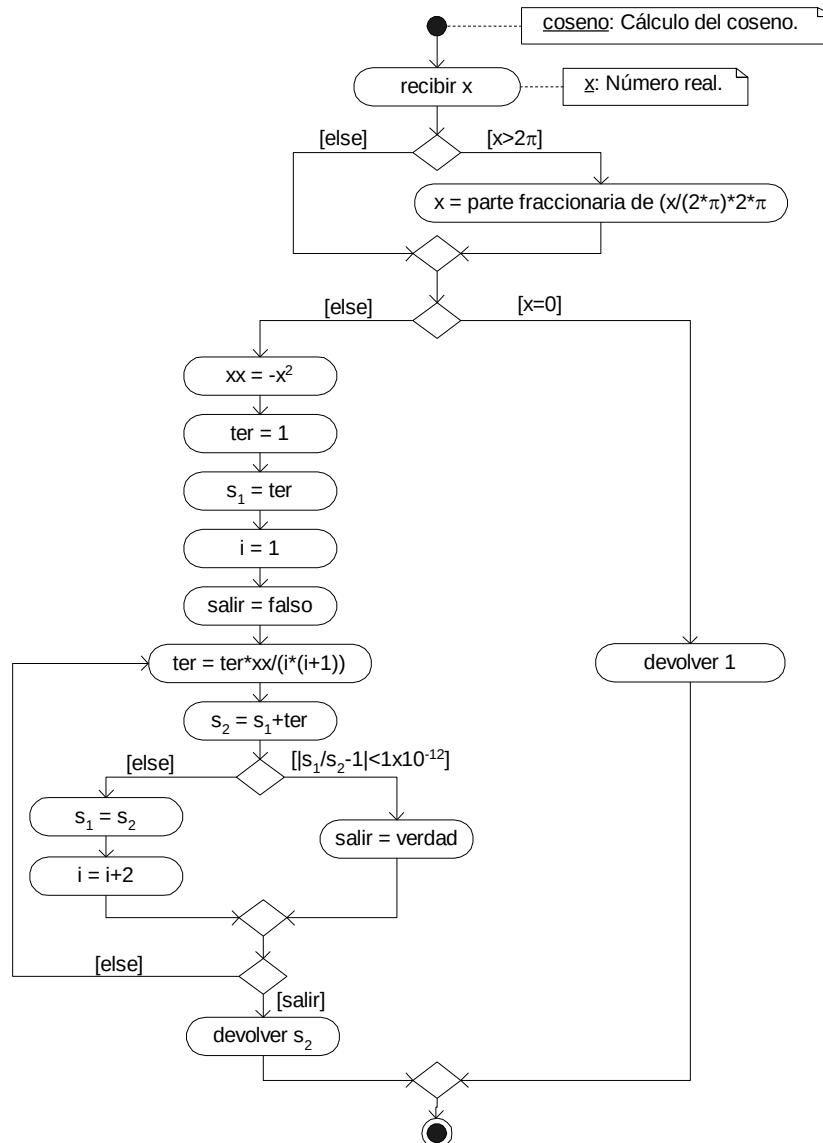
En este caso es posible evitar trabajar con números grandes reduciendo los ángulos mayores a 2π a su equivalente comprendido entre 0 y 2π . Para ello recordamos que en el coseno (al igual que en el seno) se repiten los valores entre 0 y 2π sin importar cuantas vueltas se de al cuadrante, por lo tanto el resultado es el mismo si se emplea el número completo o sólo la fracción de la última vuelta al cuadrante:

$$\text{Ángulo comprendido entre } 0 \text{ y } 2\pi = \text{Parte fraccionaria de} \left(\frac{\text{Ángulo original}}{2\pi} \right) * 2\pi$$

Así por ejemplo se obtiene el mismo resultado calculando el coseno de 3483.47 que de:

$$\text{Parte fraccionaria de} \left(\frac{3483.47}{2\pi} \right) * 2\pi = 2.58533982047$$

El algoritmo que toma en cuenta las anteriores consideraciones es:



Creemos entonces la aplicación (File -> New Application), la unidad (File -> New Unit) y guardemos la aplicación (File -> Save Project As...) en el directorio "Iteración\Coseno" con los siguientes nombres: "ufCoseno" para "Unit1", "uCoseno" para "Unit2" y "pCoseno" para "Project1". Y escribamos el código (incluyendo los módulos de lectura y escritura) en la unidad "uCoseno":

```
unit uCoseno;
```

```
interface
```

```
    uses StdCtrls, ExtCtrls, SysUtils;
```

```
    function LeerAngulo(m: TMemo; rg: TRadioGroup): double;
```

```
    function coseno(x: double): double;
```

```
    procedure MostrarCoseno(m: TMemo; c: double);
```

```
implementation
```

```
    function LeerAngulo(m: TMemo; rg: TRadioGroup): double;
```

```
    begin
```

```
        if rg.ItemIndex = 0 then
```

```
            result:= StrToFloat(m.Text)*Pi/180
```

```
        else
```

```
            result:= StrToFloat(m.Text);
```

```
    end;
```

```
    function coseno(x: double): double;
```

```
    var xx,ter,s1,s2: double; i: word; salir: boolean;
```

```
    begin
```

```
        if x>2*pi then x:= Frac(x/(2*pi))*2*pi;
```

```
        if x=0 then result:= 1
```

```
        else begin
```

```
            xx:= -sqr(x); ter:= 1; s1:= ter; i:=1; salir:= False;
```

```
            repeat
```

```
                ter:= ter*xx/(i*succ(i));
```

```
                s2:= s1+ter;
```

```
                if abs(s1/s2-1)<1E-12 then salir:= True
```

```
                    else begin s1:= s2; inc(i,2); end;
```

```
            until salir;
```

```
            result:= s2;
```

```
        end;
```

```
    end;
```

```
    procedure MostrarCoseno(m: TMemo; c: double);
```

```
    begin
```

```
        m.Text:= FloatToStr(c);
```

```
    end;
```

```
end.
```

Ahora elaboremos la interfaz de la aplicación de manera que en ejecución tenga la apariencia que se muestra en la siguiente figura:



Las propiedades modificadas de los componentes de esta aplicación son:

Form1: Name = 'fCoseno'; Caption = 'Cálculo del coseno'; Color = clInactiveCaptionText; Height = 187; Width = 282; Position = poScreenCenter;

Label1: Caption = 'Ángulo:'; Label2: Caption = 'Coseno:'; Propiedades comunes: Font.Color = clGreen; Transparent = True.

Memo1: Name = 'mAngulo'; Cursor = crIBeam; MaxLength = 16; Lines = '0'
Memo2: Name = 'mCoseno'; ReadOnly = True; TabStop = False; Cursor = crArrow; Lines = '1'; Propiedades comunes: Alignment = taRightJustify; Font.Color = clGreen; Height = 21; Width = 130, WantReturns = False; Lines = '0'.

BitBtn1: Name = 'bbCalcular'; Glyph = 'Comp2'; Caption = '&Calcular'; Cursor = crHandPoint.

RadioGroup1: Name = 'rgUnidad'; Caption = 'Unidad del ángulo'; Columns = 2; Cursor = crArrow; Font.Color = clGreen; Items = ('Grados', 'Radianes').

Debemos asegurarnos también que el orden de tabulación (Edit -> Tab Order...) sea: mAngulo, mCoseno, bbCalcular y rgUnidad.

Antes de programar los eventos recuerde incorporar la unidad "uCoseno" en la unidad "ufCoseno" (File -> Use Unit... -> uCoseno).

Para que el fondo de la aplicación tenga las líneas diagonales cruzadas, programamos el evento "onCreate" de la forma:

```
procedure TfCoseno.FormCreate(Sender: TObject);
begin
  fCoseno.Color:= clInactiveCaptionText;
  fCoseno.Brush.Style:= bsDiagCross;
  rgUnidad.Brush.Style:= bsDiagCross;
end;
```

Para que al iniciar la aplicación el foco se encuentre en el memo "mAngulo" y que el texto del mismo esté seleccionado, programamos el evento "onActivate" de la forma:

```
procedure TfCoseno.FormActivate(Sender: TObject);
begin
  mAngulo.SetFocus; mAngulo.SelectAll;
end;
```

Validamos la introducción de datos al memo "mAngulo" programando el evento "onKeyPress" del memo:

```
procedure TfCoseno.mAnguloKeyPress(Sender: TObject; var Key: Char);
begin
  case Key of
    #8, '0'..'9', '.', 'e', 'E', '+', '-':
```

```

    else Beep; Abort;
end;
end;

```

Damos funcionalidad a la aplicación programando el evento "onClick" del BitBtn "bbCalcular":

```

procedure TfCoseno.bbCalcularClick(Sender: TObject);
var x: double;
begin
    x:= LeerAngulo(mAngulo,rgUnidad); x:= Coseno(x); MostrarCoseno(mCoseno,x);
    mAngulo.SetFocus; mAngulo.SelectAll;
end;

```

Finalmente compilamos la aplicación (Ctrl+F9), corregimos los errores de sintaxis, guardamos los cambios (File -> Save All) y hacemos correr la aplicación, entonces la interfaz deberá tener la apariencia que se muestra en la figura de la anterior página y deberá calcular el mismo resultado. Si el resultado no concuerda debe corregir el código haciendo correr el programa paso a paso para corregir la lógica y encontrar el lugar donde se produce el error. Haga la prueba también con otros ángulos en grados y radianes.

8.2. Ejercicios

1. Elabore una aplicación y escriba el evento "onCreate" de la misma de manera que la forma se encuentre centrada en el escritorio, tenga el título "ESTRUCTURA UNTIL", su propio icono, líneas horizontales y verticales de color verde.
2. Elabore una aplicación y escriba el evento "onCreate" de la misma de manera que el fondo de la aplicación sea una figura, que el título sea "ESTRUCTURAS ITERATIVAS", tenga su propio icono y la forma esté centrada en la pantalla.
3. Elabore una aplicación con un memo y en el evento "onCreate" fije el ancho del memo en 120 puntos, el alto en 23 puntos, la alineación al centro, la longitud máxima del texto en 15 caracteres, impida la adición de nuevas líneas con la tecla "Enter", establezca el texto del memo en "ESTRUCTURA UNTIL", con la letra "Arial" en color azul sobre fondo amarillo y con los estilo negrita y subrayado.
4. Elabore una aplicación con un botón y en el evento "onCreate" de la forma haga que tenga el título "Botón 1", la imagen correspondiente al botón "Retry", que el puntero cambie a una mano apuntando cuando esté sobre el botón y que se active con la tecla "Enter".
5. Elabore una aplicación con un botón y en el evento "onCreate" de la forma haga que tenga el título "Botón 2", la imagen "calculat.bmp", que esté inhabilitado, que el puntero cambie a un reloj de arena cuando esté sobre el botón y que se active con la tecla "Cancel".
6. Elabore una aplicación con dos memo y escriba los eventos "onEnter" y "onExit" de los mismos de manera que al ingresar a los memos aparezca el texto "ha ingresado al memo 1" (o del memo 2), quedando el texto seleccionado y al salir aparezca el texto "ha salido del memo 1" (o del memo 2).
7. Elabore una aplicación con un "RadioGroup" y un "BitBtn". En el evento "onCreate" de la forma añada al "RadiGroup" los días de la semana dis-

tribuidos en dos columnas, quedando seleccionado el primer día. En el evento "onClick" del "BitBtn" haga que se seleccione el siguiente día, pasando al primer día cuando se ha llegado al último.

8. Elabore una aplicación y en la misma cree un módulo que empleando la estructura "hasta" resuelva la ecuación de tercer grado:

$$ax^3 + bx^2 + cx + d = 0$$

Empleando la ecuación de Newton:

$$x_2 = \frac{2ax_1^3 + bx_1^2 - d}{3ax_1^2 + 2bx_1 + c}$$

El módulo debe permitir introducir el valor inicial asumido " x_1 " y generar un error si el número de iteraciones sobrepasa las 50. Pruebe el módulo en el evento "onContextPopUp" con las ecuaciones " $x^3 + 2x^2 + 3x + 4$ ", " $3x^3 - 2x^2 + 3x - 5$ ", " $2x^3 - 5x^2 - 3x + 2$ ".

9. Elabore una aplicación y en la misma cree un módulo que empleando la estructura "hasta" calcule el Chebyshev enésimo de un número real " x ":

$$Ch_{n+1}(x) = 2xCh_n(x) - Ch_{n-1}(x); \quad Ch_0(x) = 1; \quad Ch_1(x) = x$$

Pruebe el módulo elaborado en el evento "onClick" de la forma con " $n=0$, $x=3.2$ ", " $n=1$, $x=5.5$ ", " $n=2$, $x=3.3$ ", " $n=3$, $x=5.1$ " y " $n=4$, $x=3.2$ ".

10. Cree una aplicación y en la misma cree un módulo que empleando la estructura "hasta" calcule el exponente de un número real empleando la serie:

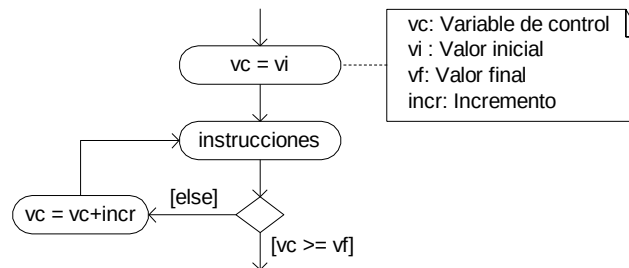
$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots, \quad -\infty < x < \infty$$

Debe tomar en cuenta que cuando " x " es negativo en la serie aparecen restas, las cuales como se sabe origina errores debidos al redondeo, razón por la cual debe evitar trabajar con números negativos, tomando en cuenta que $e^{-x} = 1/e^x$. Pruebe el módulo en el evento "onDbClick" de la forma con " $x=0$ ", " $x=3.24$ ", " $x=6.899$ ", " $x=2892.423$ ", " $x=-1$ ", " $x=-5.322$ " y " $x=-78932.323$ ".

9. ESTRUCTURA FOR

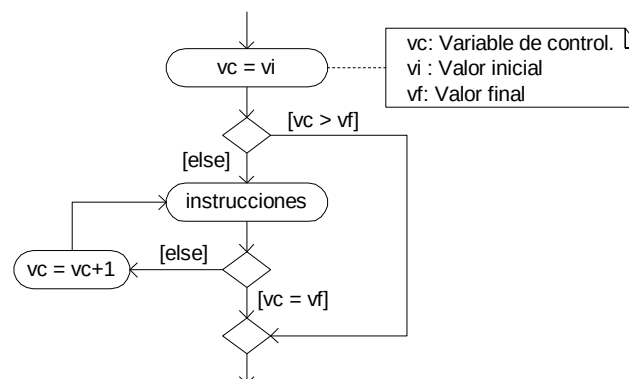
De todas las estructuras iterativas empleadas en la programación estructurada, la estructura *FOR* es la menos estándar. La implementación de esta estructura (y en consecuencia su lógica) varía de un lenguaje a otro.

En general, sin embargo, el propósito de esta estructura es la de repetir una o más instrucciones un determinado número de veces, desde un valor inicial hasta un valor final. La lógica general de esta estructura es la siguiente:



Donde la acción se repite desde que la variable de control tiene un valor inicial (*vi*) hasta que alcanza o supera un valor final (*vf*). En cada repetición el valor de la variable de control es incrementado en un cierto valor (*incr*).

En el caso específico de Pascal, el diagrama de actividades de la estructura *FOR*, es el siguiente:



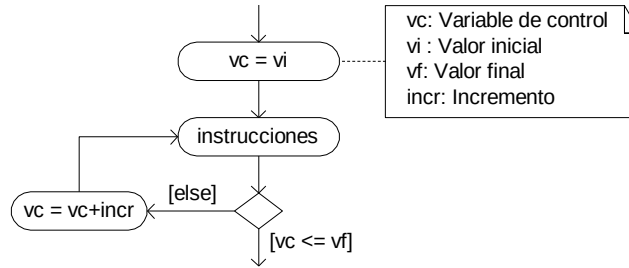
Como se puede observar, antes de ejecutar las instrucciones la estructura *For* de pascal pregunta si la variable de control es mayor al valor final y de ser así las instrucciones no se ejecutan ni una sola vez; las instrucciones del ciclo dejan de ejecutarse cuando la variable de control es igual al valor final, por lo tanto, en el caso específico de Pascal, cuando el ciclo termina la variable de control es igual al valor final; igualmente, en el caso de Pascal, en cada repetición del ciclo la variable de control incrementa en 1.

La forma de codificar esta estructura es la siguiente:

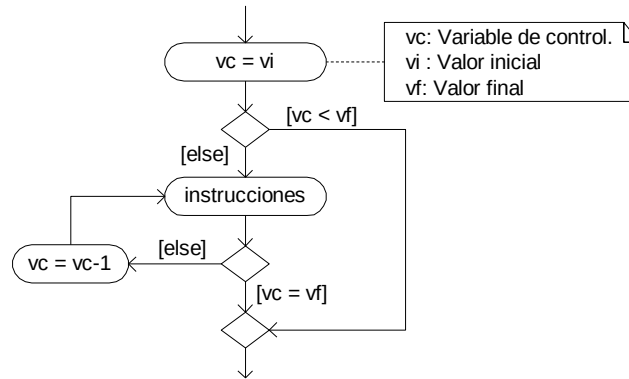
for *vc:=vi to vf do* instrucción;

Donde, como de costumbre, si la *instrucción* consta de dos o más instrucciones, deben ser encerradas entre *begin* y *end*.

En la estructura *FOR* el valor de la variable de control puede disminuir en cada iteración, en lugar de incrementar, en cuyo caso la lógica es la siguiente:



Y que para el caso específico de *Pascal* es:



La forma de codificar la forma descendente en *Pascal* es:

```
for vc:=vi downto vf do instrucción;
```

En la estructura "For" de *Pascal* la variable de control (*vc*) tiene que ser de tipo ordinal. La estructura *FOR* es particularmente útil para resolver problemas iterativos cuando se sabe el número de veces que debe repetirse el proceso, así es la estructura que deberíamos elegir para resolver el Fibonacci y el Legendre.

9.1. Ejemplos

1. Cálculo del factorial

La ecuación de definición del factorial es:

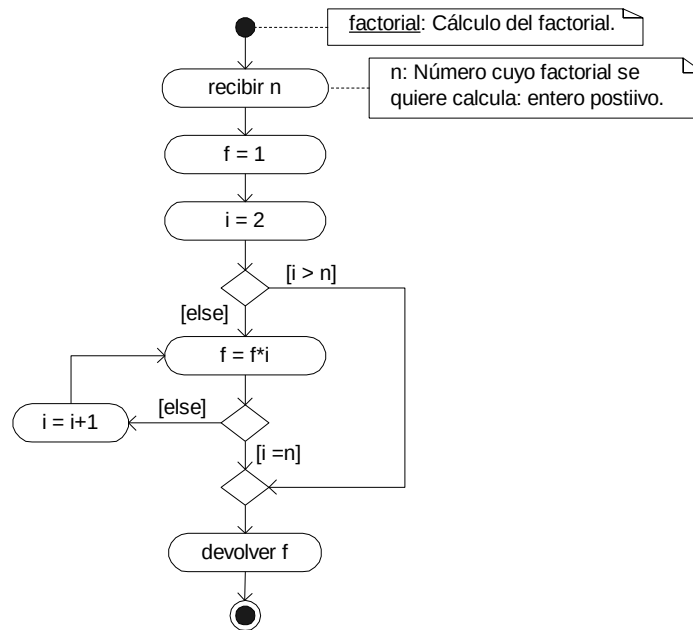
$$n! = 1 * 2 * 3 * \dots * n = \prod_{i=1}^n i; \quad 0! = 1$$

En consecuencia el factorial se calcula multiplicando los valores de "*i*" desde 1 hasta "*n*", por lo tanto es básicamente una aplicación directa de la estructura "For", tal como se muestra en el diagrama de actividades de la siguiente página.

Creemos entonces la aplicación (File -> New Application), la unidad (File -> New Unit) y guardemos la aplicación (File -> Save Project As...) en el directorio "Iteración\Factorial" con los siguientes nombres: "ufFactorial" para "Unit1", "uFactorial" para "Unit2" y "pFactorial" para "Project1". Y escribamos el código (incluyendo los módulos de lectura y escritura) en la unidad "uFactorial":

```
unit uFactorial;
```

```
interface
```



```
uses StdCtrls, Mask, SysUtils;
```

```
function LeerNumero(me: TMaskEdit): word;
```

```
function factorial(n: word): extended;
```

```
procedure MostrarFactorial(m: TMemo; f: extended);
```

implementation

```
function LeerNumero(me: TMaskEdit): word;
```

```
begin
```

```
    result:= StrToInt(me.Text);
```

```
end;
```

```
function factorial(n: word): extended;
```

```
var f: extended; i: word;
```

```
begin
```

```
    f:=1;
```

```
    for i:=2 to n do f:=f*i;
```

```
    result:= f;
```

```
end;
```

```
procedure MostrarFactorial(m: TMemo; f: extended);
```

```
begin
```

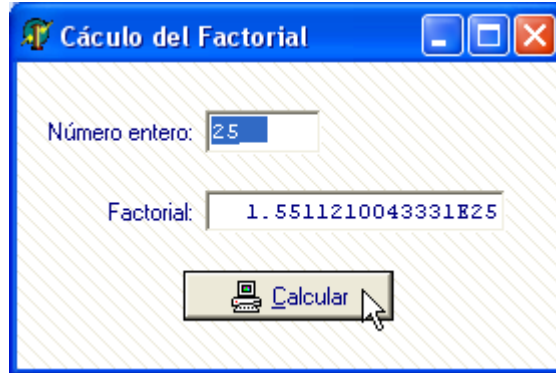
```
    m.Text:= FloatToStr(f);
```

```
end;
```

```
end.
```

Al igual que ocurre con el Fibonacci, el factorial de un número es un valor entero, sin embargo, el límite de los tipos enteros, inclusive el entero más grande: Int64, es superado rápidamente, pues por ejemplo el factorial de 30 es $2.65252859812191 \times 10^{32}$ (es decir un número entero con 32 dígitos), el cual supera ampliamente a los 19 permitidos en Int64. Es por esta razón que en la función "Factorial" se ha empleado un tipo real, y el tipo real más grande disponible "Extended", porque por ejemplo el factorial de 250 es $3.23285626090911 \times 10^{492}$ (es decir un número entero con 492 dígitos), el cual supera ampliamente el límite de un tipo "double" (o "real").

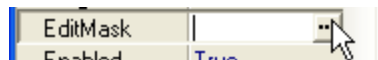
Elaboremos ahora la interfaz de la aplicación, que en ejecución, deberá tener la apariencia que se muestra en la siguiente figura:



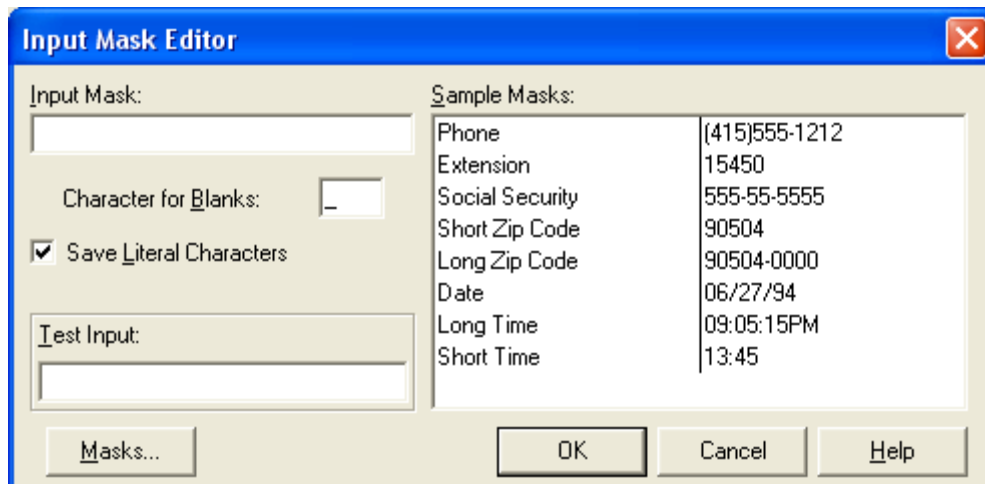
En esta interfaz el número es introducido en un componente nuevo un "MaskEdit": `MaskEdit`, el cual se encuentra en la página "Additional" y que está declarado en la unidad "Mask", siendo esta la razón por la cual se ha incluido esta unidad en "uFactorial". El componente "MaskEdit" está pensado principalmente para aquellos datos con un formato fijo, como fechas, números de teléfono, códigos postales, cédulas de identificación, etc.

Por lo tanto, la presente aplicación no es la más adecuada para este tipo de componente, pero dado que un "MaskEdit" valida la introducción de datos lo emplearemos como una alternativa al evento "onKeyPress".

La mayoría de las propiedades de este componente son las mismas que las de un "Edit", siendo su principal diferencia la propiedad "EditMask", que es la propiedad en la cual introducimos la máscara que valida y da formato a los datos que se muestran en este componente. Cuando se hace doble clic en el campo de esta propiedad (o click en el botón con tres puntos):



Aparece la siguiente ventana:



En el campo "Input Mask" se introduce la máscara que empleará el componente. En el campo "Test Input" se puede probar la máscara. En "Sample Masks" se tienen varias máscaras de ejemplo y se puede elegir cualquiera de ellas para utilizarla directamente o modificarla para crear otra máscara. En el campo "Character for Blanks" se escribe el carácter que se empleará para los espacios en blanco y en el CheckBox "Save Literal Characters" se deter-

mina si se guardaran todos los caracteres de la máscara (chequeado) o sólo los caracteres que son datos (sin chequear).

Una máscara está conformada por tres partes, separadas por puntos y comas, de acuerdo al siguiente formato:

Máscara; Guardar_literales; Caracter_en_blanco

La primera parte es la "Máscara" propiamente y corresponde a los caracteres que serán permitidos en el componente cuando se introducen o muestran datos. Para crear la máscara se emplean algunos caracteres que tienen un significado especial, así por ejemplo si en la máscara aparece un "0", en ese lugar de la máscara se debe introducir un número, no se permite ningún otro carácter y tampoco se permite que quede en blanco, es decir que la introducción del número en esa posición de la máscara es obligatorio. Por otra parte si en la máscara aparece un "9", en esa posición se permite introducir un número, pero a diferencia del "0", se permite que el mismo quede en blanco.

A más del "0" y el "9" se pueden emplear los siguientes caracteres para construir una máscara: "#": para un número, el signo "+" o el signo "-", permite dejar en blanco. "L": para una letra, no permite dejar en blanco. "l": para una letra, permite dejar en blanco. "A": para una letra o número, no permite dejar en blanco. "a": para una letra o número, permite dejar en blanco. ">": A partir de la posición en la que aparece este carácter, todas las letras son convertidas en mayúsculas. "<": A partir de la posición en que aparece este carácter, todas las letras son convertidas en minúsculas. "<>": A partir de la posición en que aparecen estos caracteres, las letras no se convierten ni a mayúsculas ni a minúsculas. "/": para separar los meses, días y años en una fecha. ":": para separar las horas, minutos y segundos de una hora. "_": para insertar espacios en blanco en la máscara. "\": el carácter que se escribe después de este símbolo aparece como un literal en la máscara, se emplea para que aparezcan en la máscara caracteres especiales como el "0", "#" y el "9".

La segunda parte de la máscara "Guardar_literales" es un "1" si se guardan los literales de la máscara, es decir si se guarda todo el texto y "0" si sólo se guardan los datos propiamente, pero no los literales. Por ejemplo si la máscara es:

'99.999_m^2'

Los literales son "." y " m^2", entonces si "Guardar_literales" es "1" y se introduce "45.423", el texto que se guarda es: "45.423 m^2", por el contrario si "Guardar_literales" es "0" el texto que se guarda es: "45423".

La tercera parte de la máscara "Caracter_en_blanco" es el carácter que se mostrará en el "MaskEdit" para los espacios en los cuales todavía no se han introducido datos, siendo por defecto el guión bajo "_". Por ejemplo si la máscara es:

'999.999;1;0'

La máscara, antes de la introducción de datos, se muestra como:

'000.000'

Analice los ejemplos que tiene en la ventana "Input Mask Editor" y haga pruebas con diferentes máscaras para familiarizarse con su uso.

Los otros componentes de la forma son ya conocidos y las propiedades modificadas son las siguientes:

Form1: Name = 'fFactorial'; Caption = 'Cálculo del Factorial'; Height = 187; Width = 219; Position = poScreenCenter;

Label1: Caption = 'Número entero: '; Label2: Caption = 'Factorial: '; Propiedades comunes: Font.Color = clNavy; Transparent = True.

MaskEdit1: EditMask = '9999;0;_'; Font.Color = clNavy; Font.Name = 'Courier New'; Cursor = crIBeam; Text = '0'.

Memo1: Name = 'mFactorial'; Cursor = crArrow; Lines = '1' Memo2: Name = 'mCoseno'; ReadOnly = True; TabStop = False; Font.Color = clNavy; Height = 21; Width = 130, WantReturns = False; Font.Name = 'Courier New'.

BitBtn1: Name = 'bbCalcular'; Glyph = 'Comppc1'; Caption = '&Calcular'; Cursor = crHandPoint; Font.Color = clNavy.

Debemos asegurarnos también que el orden de tabulación (Edit -> Tab Order...) sea: meNumero, mFactorial y bbCalcular.

Antes de programar los eventos recuerde incorporar la unidad "uFactorial" en la unidad "ufFactorial" (File -> Use Unit... -> uFactorial).

Para que el fondo de la aplicación tenga las líneas diagonales programamos el evento "onCreate" de la forma:

```
procedure TfFactorial.FormCreate(Sender: TObject);
begin
    fFactorial.Brush.Style:= bsFDiagonal;
end;
```

En este caso no validamos la introducción de datos, pues la máscara lleva a cabo esa labor al permitir introducir sólo números enteros. Finalmente damos funcionalidad a la aplicación programando el evento "onClick" del botón "bbCalcular":

```
procedure TfFactorial.bbCalcularClick(Sender: TObject);
var n: word; f: extended;
begin
    n:= LeerNumero(meNumero); f:= Factorial(n);
    MostrarFactorial(mFactorial,f);
    meNumero.SetFocus; meNumero.SelectAll
end;
```

Entonces al hacer correr la aplicación (después de corregir los errores y guardar los cambios), la interfaz deberá tener la apariencia y calcular el resultado mostrado anteriormente en la figura.

2. Cálculo del Chebyshev

Como segundo ejemplo elaboraremos una aplicación para calcular el Chebyshev enésimo de un número real.

La fórmula de definición del *Chebyshev* es:

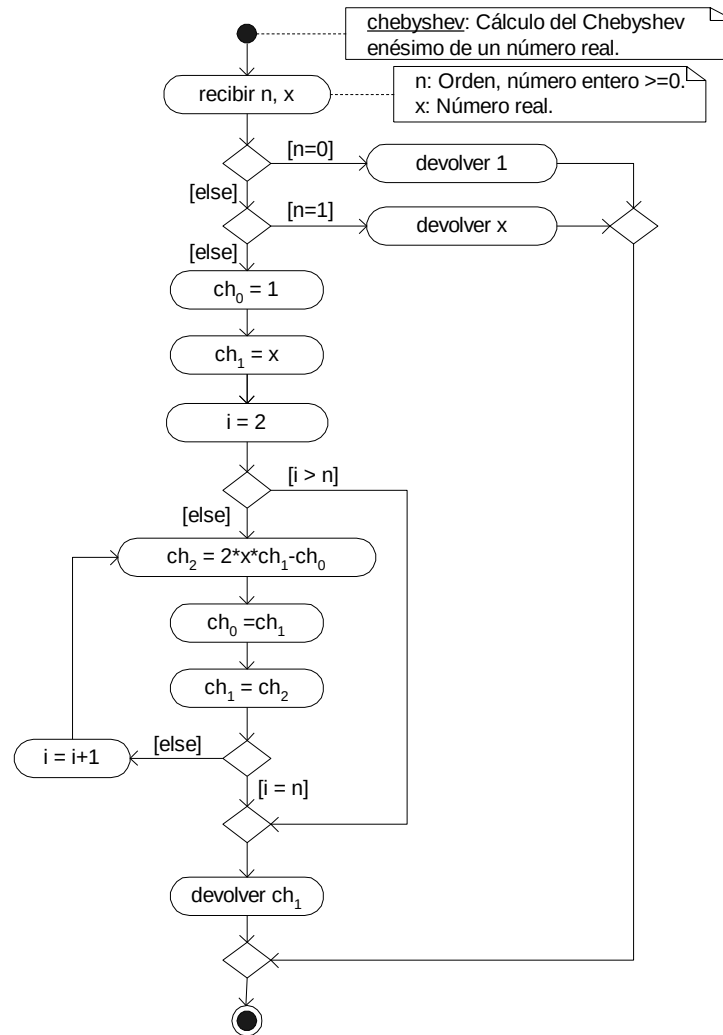
$$Ch_{n+1}(x) = 2xCh_n(x) - Ch_{n-1}(x)$$

$$Ch_0(x) = 1$$

$$Ch_1(x) = x$$

La lógica para resolver este problema es básicamente la misma que para el Fibonacci y el Legendre: Comenzando con los dos valores conocidos (Ch_0 y Ch_1) se calcula el tercero (Ch_2), entonces con los dos últimos valores calculados (Ch_1 y Ch_2) se calcula el cuarto y así sucesivamente hasta llegar al Chebyshev que se quiere calcular. En otras palabras, el proceso se debe repetir desde 2 hasta n.

El algoritmo para resolver esta ecuación se presenta en el diagrama de actividades de la siguiente página:



Observe que en este caso la condición " $i > n$ " es innecesaria, porque nunca se presenta ese caso, sin embargo la mantenemos debido a que es parte de la estructura "For" de Pascal y por lo tanto en realidad no se la escribe.

Para codificar el algoritmo creemos la aplicación (File -> New Application), la unidad (File -> New Unit) y guardemos la aplicación (File -> Save Project As...) en el directorio "Iteración\Chebyshev" con los siguientes nombres: "ufChebyshev" para "Unit1", "uChebyshev" para "Unit2" y "pChebyshev" para "Project1". Y escribamos el código (incluyendo los módulos de lectura y escritura) en la unidad "uChebyshev":

```
unit uChebyshev;
```

```
interface
```

```
  uses StdCtrls, Mask, SysUtils;
```

```
  function LeerOrden(me: TMaskEdit): byte;
```

```
  function LeerNumeroReal(m: Tmemo): real;
```

```
  function Chebyshev(n: byte; x: real): real;
```

```
  procedure MostrarChebyshev(m: TMemo; c: real);
```

```
implementation
```

```

function LeerOrden(me: TMaskEdit): byte;
begin
    result:= StrToInt(me.Text);
end;

function LeerNumeroReal(m: Tmemo): real;
begin
    result:= StrToFloat(m.Text);
end;

function Chebyshev(n: byte; x: real): real;
var ch0,ch1,ch2: real; i: byte;
begin
    case n of
        0 : result:= 1;
        1 : result:= x;
        else
            ch0:=1; ch1:= x;
            for i:=2 to n do begin
                ch2:= 2*x*ch1-ch0; ch0:= ch1; ch1:= ch2;
            end;
            result:= ch2;
        end;
    end;

procedure MostrarChebyshev(m: TMemor; c: real);
begin
    m.Text:= FloatToStr(c);
end;

```

end.

Elaboremos ahora La interfaz de la aplicación, que en ejecución, deberá tener la apariencia que se muestra en la siguiente figura:



Las propiedades modificadas de los componentes son:

Form1: Name = 'fChebyshev'; Caption = 'Cálculo del Chebyshev'; Height = 192; Width = 259; Position = poScreenCenter;

Label1: Caption = 'Orden:'; Label2: Caption = 'Número real:'; Label3: Caption = 'Chebyshev:'; Propiedades comunes: Font.Color = clYellow; Transparent = True.

MaskEdit1: EditMask = '999;0;_'; Font.Color = clRed; Font.Name = 'Courier New'; Cursor = crIBeam; Text = '0'.

```

Memo1: Name = 'mNumReal'; Cursor = crIBeam; Memo2: Name = 'mChebyshev:';
ReadOnly = True; TabStop = False; Cusor = crArrow; Propiedades comunes:
Font.Color = clRed; Height = 21; Width = 130, WantReturns = False; Font.Name
= 'Courier New'.

```

```

BitBtn1: Name = 'bbCalcular'; Kind = bkOK; Caption = '&Calcular'; Cursor
= crHandPoint; Font.Color = clRed.

```

Debemos asegurarnos también que el orden de tabulación (Edit -> Tab Order...) sea: meOrden, mNumreal, mChebyshev y bbCalcular.

Antes de programar los eventos recuerde incorporar la unidad "uChebyshev" en la unidad "ufChebyshev" (File -> Use Unit... -> uFactorial).

En esta aplicación el fondo tiene la figura "A pescar.BMP", la misma que se asigna en el evento "onCreate" de la forma y se libera en el evento "onClose" (recuerde copiar la figura "A pescar.BMP" al directorio "Iteración\Chebyshev"):

```

procedure TfChebyshev.FormCreate(Sender: TObject);
begin
    fChebyshev.Brush.Bitmap:= TBitmap.Create;
    fChebyshev.Brush.Bitmap.LoadFromFile('A pescar.bmp');
    meOrden.Text:= '0';
    mNumReal.Text:= '1';
    mChebyshev.Text:= '1';
end;

```

```

procedure TfChebyshev.FormClose(Sender: TObject; var Action: TCloseAction);
begin
    fChebyshev.Brush.Bitmap.Free;
end;

```

Observe que en el evento "onCreate" se ha inicializado también el texto del "MaskEdit" y de los "Memos". Como para el número real se ha empleado un memo, debido a que el "MaskEdit" no es adecuado para este fin, debemos validar la introducción de datos programando el evento "onKeyPress":

```

procedure TfChebyshev.mNumRealKeyPress(Sender: TObject; var Key: Char);
begin
    case Key of
        #8, '0'..'9', '.', '+', '-':
            else Beep; Abort;
    end;
end;

```

Para que cuando el foco pase al memo "mNumReal", el texto quede seleccionado y se pueda escribir directamente un nuevo valor, sin borrar el texto anterior, programamos el evento "onEnter" del memo "mNumReal":

```

procedure TfChebyshev.mNumRealEnter(Sender: TObject);
begin
    mNumReal.SelectAll;
end;

```

Finalmente damos funcionalidad a la aplicación programando el evento "onClick" del BitBtn "bbCalcular":

```

procedure TfChebyshev.bbCalcularClick(Sender: TObject);
var n: byte; x,c: real;
begin
    n:= LeerOrden(meOrden);
    x:= LeerNumeroReal(mNumReal);

```

```

c:= Chebyshev(n,x);
MostrarChebyshev(mChebyshev,c);
meOrden.SetFocus; meOrden.SelectAll;
end;

```

Entonces al hacer correr la aplicación (después de corregir los errores y guardar los cambios), la interfaz deberá tener la apariencia y calcular el resultado mostrado anteriormente en la figura.

3. Integración numérica con la ecuación de Simpson

Como tercer ejemplo elaboraremos una aplicación para calcular el valor de la siguiente integral con el método de *Simpson*.

$$\int_a^b f(x)dx = \int_a^b ydx = \int_a^b \left(\frac{3x^2 + 2x - 4}{5x^3 - 4x^{2.1}} \right) dx$$

La ecuación de *Simpson* es la siguiente:

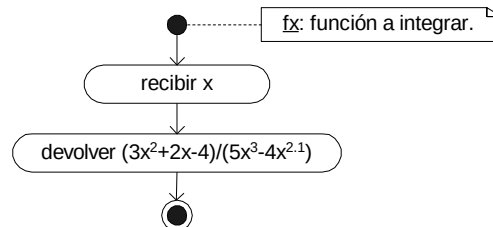
$$\int_a^b f(x)dx = \frac{h}{3} \left(f(x_1) + 4 \sum_{i=2,4,6}^n f(x_i) + 2 \sum_{i=3,5,7}^{n-1} f(x_i) + f(x_{n+1}) \right)$$

$$h = \frac{b-a}{n}$$

Donde "f" es la función a integrar, "x₁" es el límite inferior ("a"), "x_{n+1}" es el límite superior ("b"), "n" es el número de segmentos que existe entre "a" y "b" (el cual siempre debe ser par), "h" es el ancho de cada uno de esos segmentos.

Los valores de "x₂", "x₃", etc., se calculan sumando al valor anterior "h", así "x₂" es igual a "x₁+h", "x₃" es igual a "x₂+h", "x₄" es igual a "x₃+h" y así sucesivamente. Las dos sumatorias corresponden simplemente a un ciclo "For" que va desde 2 hasta n, donde cuando el número es par se guarda la sumatoria en una variable y cuando es impar en otra. El algoritmo para el método de Simpson se muestra en la siguiente página y como se puede ver, antes de comenzar los cálculos se pregunta si el número de divisiones "n" es impar y de ser así se le suma "1" para que se convierta en un número par, pues como se dijo, para el método de Simpson "n" siempre debe ser par.

Y el algoritmo de la función a integrar es:



Dado que en Pascal no existe un tipo de dato que sea función o procedimiento, para mandar una función o procedimiento como parámetro a un módulo, se debe declarar un nuevo tipo de dato de acuerdo a la siguiente sintaxis:

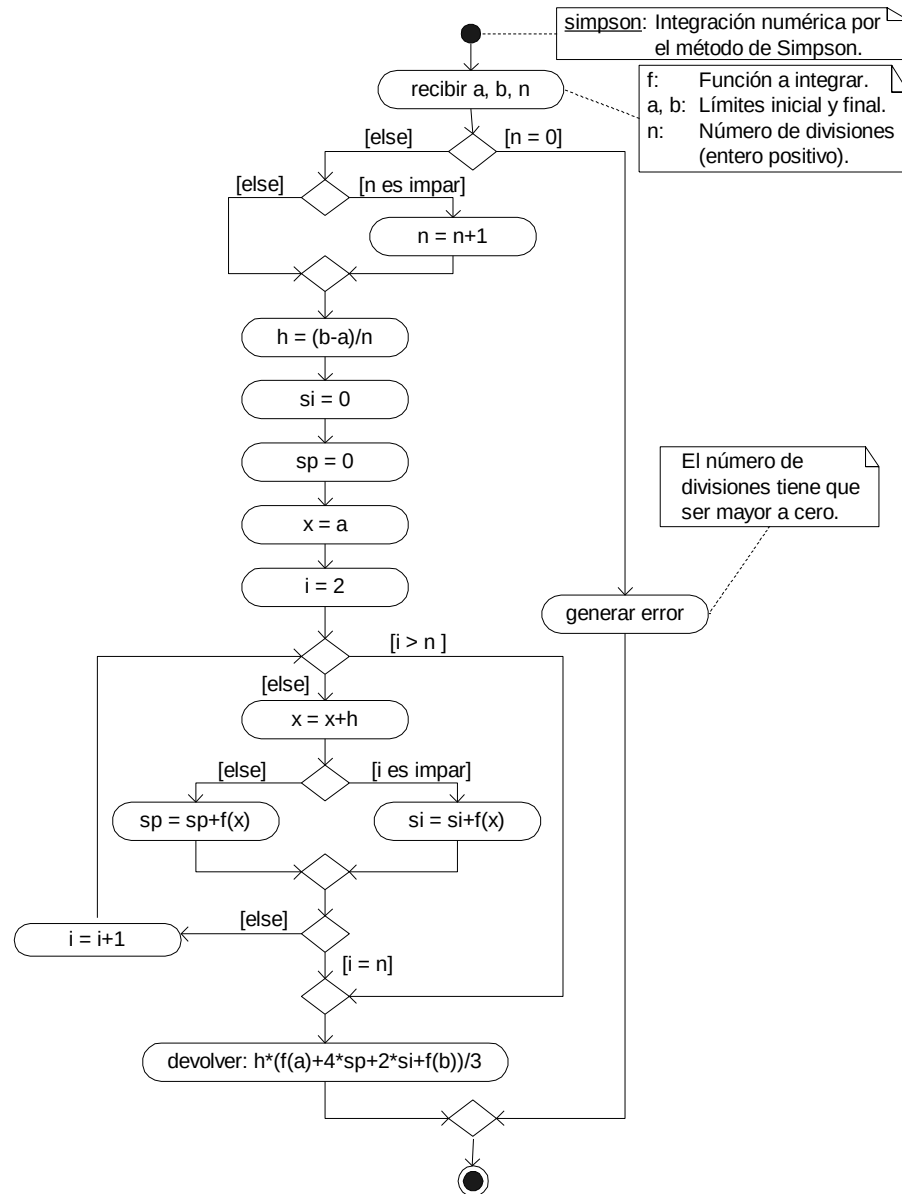
type

```

Nombre_del_tipo = function (lista_de_parámetros): tipo_de_resultado;
Nombre_del_tipo = procedure (lista_de_parámetros);

```

Donde los parámetros pueden tener cualquier nombre, pues en realidad los nombres no se utilizan, pero si es importante el orden de los parámetros y los tipos de datos, así por ejemplo si creamos el siguiente tipo de dato:

**type**

```
tMiFun = function (x: double; n: integer): double;
```

Resulta adecuado para las siguientes funciones (porque en las mismas se tienen los tipos de datos adecuados y en el orden correcto):

```
function Fun1(y: double; r: integer): double;
```

```
function Fun2(r: double; m: integer): double;
```

```
function Fun3(z: double; o: integer): double;
```

```
function Fun4(x: double; n: integer): double;
```

Pero no para las siguientes, porque en las mismas el orden y/o los tipos de datos son erróneos:

```
function Fun5(n: integer; x: double): double;
```

```
function Fun6(x: integer; n: double): double;
```

```
function Fun7(x: extended; n: byte): LongInt;
```

```
function Fun8(x: Single; n: cardinal): extended;
```

Para codificar el algoritmo creemos la aplicación (File -> New Application), la unidad (File -> New Unit) y guardemos la aplicación (File -> Save Project As...) en el directorio "Iteración\Simpson" con los siguientes nombres: "ufSimpson" para "Unit1", "uSimpson" para "Unit2" y "pSimpson" para "Project1". Y escribamos el código (incluyendo los módulos de lectura y escritura) en la unidad "uSimpson":

```

unit uSimpson;
interface

  uses StdCtrls, SysUtils, Math;
  type
    tfx = function (x: real): real;

  function LeerNumeroReal(m: TMemo): real;
  function LeerNumeroEntero(m: TMemo): word;
  function Simpson(f:tfx; a,b:real; n: word): real;
  function fx(x: real): real;
  procedure MostrarIntegral(m: TMemo; i: real);

implementation

  function LeerNumeroReal(m: TMemo): real;
  begin
    result:= StrToFloat(m.Text);
  end;

  function LeerNumeroEntero(m: TMemo): word;
  begin
    result:= StrToInt(m.Text);
  end;

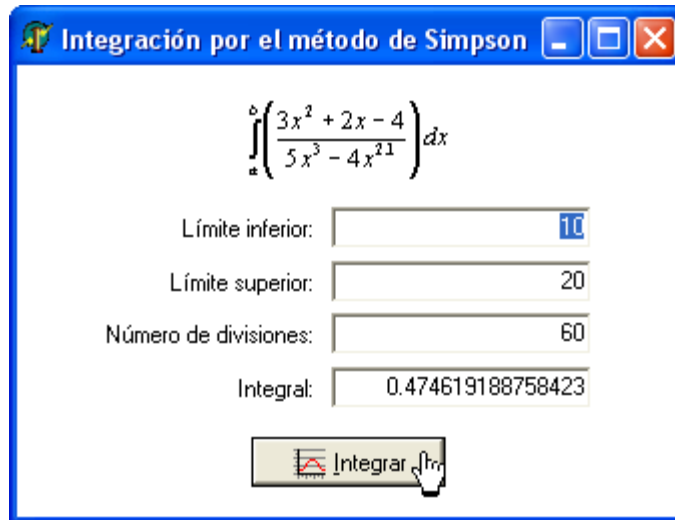
  function Simpson(f:tfx; a,b:real; n: word): real;
  var h,si,sp,x: real; i: word;
  begin
    if n=0 then raise EInvalidArgument.Create(
      'El número de divisiones tiene que ser mayor a cero')
    else begin
      if odd(n) then inc(n);
      h:= (b-a)/n; si:= 0; sp:= 0; x:= a;
      for i:=2 to n do
        begin
          x:= x+h;
          if odd(i) then si:= si+f(x) else sp:= sp+f(x);
        end;
      result:= h*(f(a)+4*sp+2*si+f(b))/3;
    end;
  end;

  function fx(x: real): real;
  begin
    result:= (3*sqr(x)+2*x+4)/(5*IntPower(x,3)-4*Power(x,2.1));
  end;

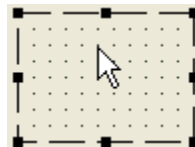
  procedure MostrarIntegral(m: TMemo; i: real);
  begin
    m.Text:= FloatToStr(i);
  end;
end.

```

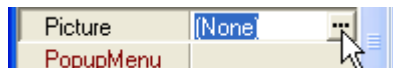
Elaboremos ahora La interfaz de la aplicación, que en ejecución, deberá tener la apariencia que se muestra en la siguiente figura:



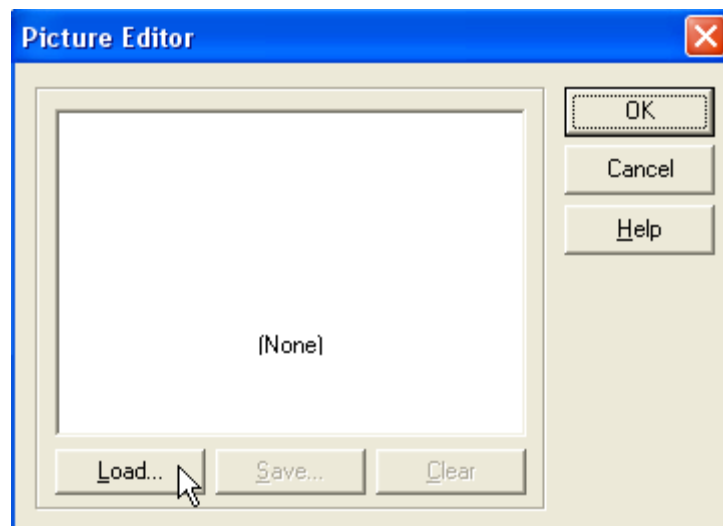
En esta aplicación existe un nuevo componente "TImage": , que se encuentra en la página "Additional" y que sirve para añadir imágenes a la forma, en esta aplicación ha sido empleado para mostrar la integral, la cual está en forma de una imagen. Cuando se coloca este componente en la forma tiene la siguiente apariencia:



La propiedad en la cual se asigna la imagen a este componente es la propiedad "Picture", que es similar a la propiedad "Glyph" del BitBtn. Para asignar una imagen se hace doble clic en el campo de esta propiedad (o clic en el botón con tres puntos de la misma):



Entonces aparece la siguiente ventana:



Al hacer clic en el botón "Load" aparece el explorador de Windows y en el mismo se puede ubicar el archivo que será mostrado en el componente. La imagen que se muestra en el componente puede ser de tipo "jpg", "jpeg", "bmp", "ico", "emf" o "wmf".

Para este ejemplo se ha creado la ecuación en MathType y se la ha guardado, como un archivo "wmf" (Save Copy As...), en el directorio "Iteración\Simpson" con el nombre "ecuacion.wmf".

Al igual que con la brocha (Brush) del lienzo (Canvas) o de la forma (Form), la imagen puede ser cargada también con el método "LoadFromFile" de la propiedad Picture. Así para cargar esta imagen podemos escribir la siguiente instrucción en el evento "onCreate" de la forma:

```
iEcuacion.Picture.LoadFromFile('ecuacion.wmf')
```

Con "LoadFromFile" se pueden cargar por defecto imágenes de los tipos "bmp", "ico" y "wmf". Para cargar otros tipos de imágenes se debe emplear además el método "RegisterFileFormat":

```
RegisterFileFormat(Extensión, Descripción, Tipo de Imagen)
```

Así por ejemplo para cargar imágenes "jpg" se escribe:

```
iEcuacion.Picture.RegisterFileFormat('jpg','Imágenes JPEG',TJPEGImage)
```

Donde la clase "TJPEGImage" está definida en la unidad "jpeg".

Algunas otras propiedades útiles del componente "TImage" son "AutoSize": que cuando está en verdadero (True) ajusta automáticamente el tamaño del componente al tamaño de la imagen; "Stretch": que cuando su valor es verdadero (True) hace lo contrario de "AutoSize", es decir ajusta el tamaño de la imagen al tamaño del componente; "Transparent": que al igual que en un "Label" deja transparente el fondo de la imagen (o el color especificado en la propiedad Picture.Bitmap.TransparentColor).

Las propiedades modificadas en los componentes de esta aplicación son:

Form1: Name = 'fSimpson'; Caption = 'Integración por el método de Simpson'; Height = 260; Width = 339; Position = poScreenCenter; color = clWhite.

Image1: Picture = 'ecuacion.wmf'; Autosize = True; Transparent = True.

Label1: Caption = 'Límite inferior:'; Label2: Caption = 'Límite superior:'; Label3: Caption = 'Número de divisiones:'; Label4: Caption = 'Integral:'. Propiedades comunes: Transparent = True.

Memo1: Name = 'mLimiteInferior'; Cursor = crIBeam; Memo2: Name = 'mLimiteSuperior'; Cursor = crIBeam; Memo3: Name = 'mNumDivisiones'; Cursor = crIBeam; Memo4: Name = 'mIntegral'; ReadOnly = True; TabStop = False; Cursor = crArrow; Propiedades comunes: Height = 21; Width = 130, WantReturns = False.

BitBtn1: Name = 'bbIntegrar'; Glyph = 'mean.bmp'; Caption = '&Integrar'; Cursor = crHandPoint.

Debemos asegurarnos también que el orden de tabulación (Edit -> Tab Order...) sea: mLimiteInferior, mLimiteSuperior, mNumDivisiones, mIntegral, bbIntegrar.

Antes de programar los eventos recuerde incorporar la unidad "uSimpson" en la unidad "ufSimpson" (File -> Use Unit... -> uFactorial).

Para que al recibir el foco quede seleccionado el texto del memo "mLimiteInferior" programamos el evento "onEnter":

```
procedure TfSimpson.mLimiteInferiorEnter(Sender: TObject);  
begin
```



```

    TMemo(Sender).SelectAll;
end;

```

Observe que en este caso se está empleando el parámetro "Sender", el mismo que corresponde al objeto que ha capturado el evento. Normalmente el objeto que captura el evento es aquel para el cual se escribe el código, sin embargo, cuando dos o más objetos emplean el mismo código para un evento, como sucede en este caso, el evento puede haber sido capturado por cualquiera de ellos, entonces para trabajar con el objeto que ha capturado el evento se emplea el parámetro "Sender". El parámetro "Sender" es de tipo "TObject" y "TObject" es el tipo base (o padre) de todos los objetos de Delphi y como tal puede ser tratado como cualquiera de los objetos descendientes (o hijos). Para tratar un objeto o variable de un tipo como si fuera de otro se puede emplear el moldeado de tipos, que como recordarán consiste simplemente en escribir el nombre del tipo (en este caso "TMemo") y entre paréntesis el nombre de la variable u objeto (en este caso "Sender").

Al escribir "TMemo(Sender).SelectAll", le estamos informando a Delphi que el objeto que ha capturado el evento es un "Memo" y le damos la instrucción (SelectAll) para que seleccione el texto de dicho memo.

Ahora asignamos este módulo "mLimiteInferiorEnter" al evento "onEnter" de los memos "mLimiteSuperior" y "mNumDivisiones". De esa manera, cuando cualquiera de estos tres memos recibe el "foco", queda seleccionado el texto del mismo.

El procedimiento que acabamos de explicar es el que se debe seguir siempre que dos o más objetos deban responder de la misma forma a un evento, con ello se evita la repetición del código y se facilita el mantenimiento pues sólo es necesario mantener el código en un módulo y no en dos o más. En capítulos posteriores veremos que alternativamente al moldeado de tipos se puede emplear el operador "as".

Ahora para que al activarse la forma quede seleccionado el texto del memo "mLimiteInferior", es suficiente pasarle el foco en el evento "onActivate":

```

procedure TfSimpson.FormActivate(Sender: TObject);
begin
    mLimiteInferior.SetFocus;
end;

```

En esta aplicación asignaremos el texto de los memos en el evento "onCreate", esto por dos razones: a) El inspector de objetos no muestra la propiedad "Text" del memo y b) Cuando se escribe el texto en la propiedad "lines", por defecto le añade una línea, lo que provoca un error cuando se trata de convertir este texto en número:

```

procedure TfSimpson.FormCreate(Sender: TObject);
begin
    mLimiteInferior.Text:= '0';
    mLimiteSuperior.Text:= '0';
    mNumDivisiones.Text:= '0';
    mIntegral.Text:= '0';
end;

```

Debemos validar también la introducción de datos en el memo "mLimiteInferior", programando el evento "onKeyPress":

```

procedure TfSimpson.mLimiteInferiorKeyPress(Sender: TObject; var Key: Char);
begin
    case Key of
        #8, '0'..'9', '.', '-', '+', 'e', 'E':
            else Beep; Abort;
    end;

```

```
end;
end;
```

Y asignar este módulo (mLimiteInferiorKeyPress) al evento "onKeyPress" del memo "mLimiteSuperior", pues en el mismo también se introducen números reales. Igualmente debemos validar la introducción de datos al módulo "mNumDivisiones" (en este caso para números enteros):

```
procedure TfSimpson.mNumDivisionesKeyPress(Sender: TObject; var Key: Char);
begin
  case Key of
    #8, '0'..'9':
      else Beep; Abort;
  end;
end;
```

Finalmente damos funcionalidad a la aplicación programando el evento "onClick" del BitBtn "bbIntegrar":

```
procedure TfSimpson.bbIntegrarClick(Sender: TObject);
var a,b,i: real; n: word;
begin
  a:= LeerNumeroReal(mLimiteInferior);
  b:= LeerNumeroReal(mLimiteSuperior);
  n:= LeerNumeroEntero(mNumDivisiones);
  i:= Simpson(fx,a,b,n);
  MostrarIntegral(mIntegral,i);
  mLimiteInferior.SetFocus;
end;
```

Al hacer correr la aplicación (después de corregir los errores y guardar las modificaciones) la interfaz de la aplicación deberá tener la apariencia y calcular el resultado que se muestra en la figura correspondiente a la interfaz de la aplicación.

9.2. Ejercicios

1. Elabore una aplicación con un memo y en la misma escriba un módulo que reciba como datos un número real, comprendido entre 10^{-20} y 10^{20} y muestre en el memo el número real, en color azul, sobre fondo cian, con el tipo de letra "Courier New", 12 puntos y con el estilo subrayado. Pruebe el módulo en el evento "onPaint" de la forma con el número "123.233x10¹⁰"
2. Elabore una aplicación con un "ComboBox" y en la misma escriba los eventos "onCreate" y "onClose" de la forma de manera que el fondo de la forma sea la figura "papel periódico.bmp", tenga el título "APLICACIÓN DE PRUEBA", el icono "AngelFish.ico", que la forma quede centrada en la pantalla, que el ComboBox tenga los Departamentos de Bolivia, que los mismos sólo puedan seleccionarse (no escribirse) y que al empezar la aplicación esté seleccionado el Departamento de Chuquisaca.
3. Elabore una aplicación y en la misma escriba los eventos "onCreate" y "onClose" de la forma "Form1", que tiene un objeto "BitBtn1", de manera que el fondo de la forma sea la figura "granito.bmp", el BitBtn tenga la figura 'BulbOn.bmp', el título "Generar", se active al pulsar la tecla "Cancel" y que la forma del puntero cambie a un signo de interrogación (ayuda) cuando se encuentre sobre el botón.

4. Elabore una aplicación con tres "MaskEdit" y programe el evento "onCreate" de la forma de manera que el "MaskEdit1" permita introducir sólo números reales con tres dígitos en la parte fraccionaria (siendo los números opcionales), el "MaskEdit2" permita introducir números enteros con cuatro dígitos (siendo los dígitos obligatorios) y el "MaskEdit3" permita introducir 8 letras, convirtiéndolas en mayúsculas y siendo su escritura obligatoria.
5. Elabore una aplicación con tres "MaskEdit" y programe el evento "onCreate" de la forma de manera que el "MaskEdit1" permita introducir fechas en el formato "dd/mm/aaaa", guardando los signos de separación, el "MaskEdit2" permita introducir tiempos en el formato "hh:mm:ss ", sin guardar los signos de separación, el "MaskEdit3" permita introducir números telefónicos en el formato "nn-nnnnn", guardando los caracteres extra.
6. Elabore una aplicación con tres "MaskEdit" y programe el evento "onCreate" de la forma de manera que el "MaskEdit1" permita introducir medidas de longitud en el formato "nnnn.nn cm", siendo los cuatro primeros números opcionales, los dos últimos obligatorios y guardando los caracteres extra, el "MaskEdit2" permita introducir carnets universitarios en el formato "CU: nnn-nnnnn", siendo todos los números obligatorios, estando por defecto el número 0 y sin guardar los caracteres extra, el "MaskEdit3" permita introducir los días de la semana, convirtiendo automáticamente los caracteres a minúsculas.
7. Elabore una aplicación con un "Image" (Image1) y programe el evento "onCreate" de la forma de manera que la imagen del "Image1" sea "Colinas Azules.jpg", que la imagen se ajuste al "Image" y que el "Image" ocupe toda la forma.
8. Elabore una aplicación con un "Image" (Image1) y programe el evento "onCreate" de la forma de manera que la imagen del "Image1" sea "TeaCup0.bmp", que el "Image1" se ajuste a la imagen, que la imagen sea transparente y que el color transparente sea el correspondiente al píxel ubicado en la posición x=10, y=10 de la imagen.
9. Elabore una aplicación y en la misma cree un módulo que empleando la estructura "desde" calcule la sumatoria de los primeros "n" números impares. Pruebe el módulo en el evento "onClick" de la forma con los números (n): "5", "20" y "50".
10. Elabore una aplicación y en la misma cree un módulo que empleando la estructura "desde" calcule la productoria de los números pares que existen menores o iguales a un número "n" dado. Luego pruebe el módulo en el evento "onContextPopUp" de la forma con los números (n): "11", "51" y "99".
11. Elabore una aplicación y en la misma cree un módulo que empleando la estructura "desde" calcule la siguiente productoria. Pruebe el módulo en el evento "onClick" de la forma con "n=10, x=3.2" y "n=20, x=20.3".

$$\prod_{i=2,4,6}^n (x+i)$$
12. Elabore una aplicación y en la misma cree un módulo que empleando la estructura "desde" calcule el *Legendre* enésimo (n) de un número real

"x". Pruebe el módulo con "n=0, x=3", "n=1, x=4.5", "n=3, x=6.5" y "n=5, x=3.2".

$$(n+1)Le_{n+1}(x) - (2n+1)xLe_n(x) + nLe_{n-1}(x) = 0$$

$$Le_0(x) = 1; \quad Le_1(x) = x$$

13. Elabore una aplicación y en la misma cree un módulo para integrar funciones con una incógnita por el método del trapecio empleando la estructura "desde":

$$\int_a^b f(x)dx = \frac{h}{2} \left(f(x_1) + 2 \sum_{i=2}^n f(x_i) + f(x_{n+1}) \right); \quad x_1 = a; \quad x_{n+1} = b; \quad h = \frac{b-a}{n}$$

Luego en el evento "onDbClick" calcule y muestre el resultado de la siguiente integral:

$$\int_a^b \left(\frac{\sqrt{2x^{4.1} + 3x^3 - 2x^{1.45}}}{\sqrt[3]{6.2x^{3.7} - 2x^{2.7} + 4}} \right) dx$$

10. MODIFICACIÓN DE LAS ESTRUCTURAS ESTÁNDAR

Como ya se vio en el capítulo anterior, la lógica de la mayoría de los problemas iterativos no se ajusta exactamente a ninguna de las estructuras estándar y, como dijimos, para programarlas tenemos tres alternativas: a) modificar la lógica, b) emplear banderas y c) modificar las estructuras. En el capítulo anterior hemos empleado la segunda opción (banderas) para resolver los problemas iterativos sin modificar la lógica ni las estructuras, en este capítulo emplearemos la tercera opción, es decir modificaremos las estructuras pero mantendremos la lógica.

La mayoría de los lenguajes de programación (incluido *Pascal*) permiten modificar las estructuras estándar, posibilitando con esas herramientas la resolución más eficiente y con frecuencia más clara de los problemas iterativos.

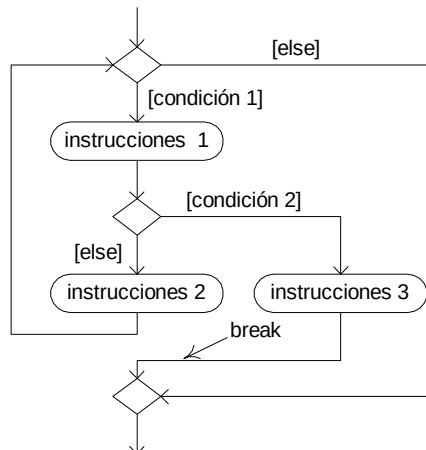
Sin embargo sólo se modifican las estructuras iterativas cuando la lógica no se ajusta a ninguna de ellas y cuando con la modificación se consigue una resolución más eficiente y sobre todo más clara del problema. Se debe tener presente también que ninguna de las modificaciones que se estudian en este capítulo es imprescindible, pues sigue siendo cierto el teorema de la programación estructurada: "todo problema, por muy complejo que sea, puede ser resuelto empleando sólo tres estructuras: secuencia, selección e iteración". No obstante recurrimos a la modificación de las estructuras estándar, porque en la práctica es preferible modificar una estructura, que es más una cuestión de forma, que modificar la lógica, que es una cuestión de fondo.

Por consiguiente el propósito del presente capítulo es que al concluir el mismo estén capacitados para resolver problemas iterativos recurriendo, cuando con ello se consigue una solución más clara, a la modificación de las estructuras estándar.

Básicamente son dos las formas en las que se pueden modificar las estructuras iterativas y 2 más en las que se puede modificar un módulo. En realidad existe una forma más: el comando *Goto*, la cual será estudiada con más detalle en un capítulo posterior.

10.1. Ruptura de un ciclo iterativo (Break)

Esta modificación permite romper un ciclo iterativo *WHILE*, *UNTIL* o *FOR*, saliendo del mismo desde cualquier punto intermedio. Con *WHILE* la lógica suele ser la que se muestra en el siguiente diagrama de actividades:



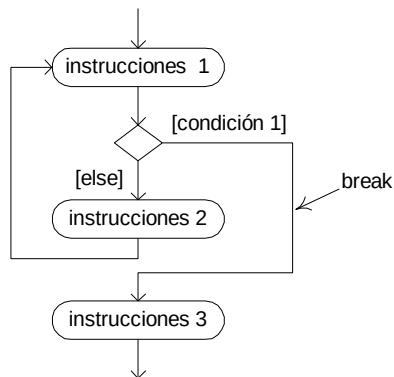
Donde sólo para ilustrar se ha señalado el flujo que corresponde a esta modificación con la palabra *break*, no obstante, esto es algo que no debe aparecer en un diagrama de actividades pues esta es una instrucción propia de un lenguaje específico (Pascal). La lógica de este diagrama se codifica en Pascal de la siguiente forma:

```

while condición 1 do
begin
  instrucciones 1;
  if condición 2 then
  begin
    instrucciones 3;
    break;
  end;
  instrucciones 2;
end;

```

Con frecuencia la "condición 1", no existe, sino que es directamente un ciclo repetitivo que termina cuando se cumple alguna condición intermedia:



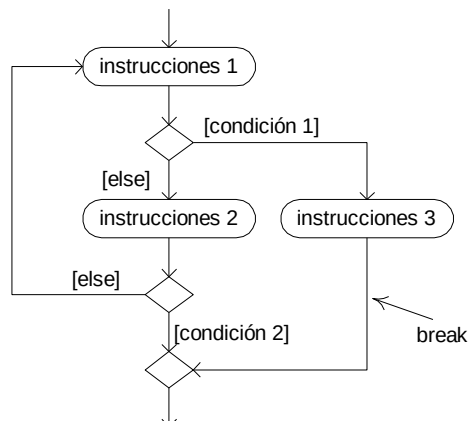
Para codificar esta lógica con "While", se escribe "True" en lugar de la condición del ciclo, tal como se muestra a continuación:

```

while True do
begin
  instrucciones 1;
  if condición 1 then break;
  instrucciones 2;
end;
instrucciones 3;

```

Con *UNTIL*, la modificación *break* suele tener la siguiente lógica:



La cual se codifica en Pascal de la siguiente forma:

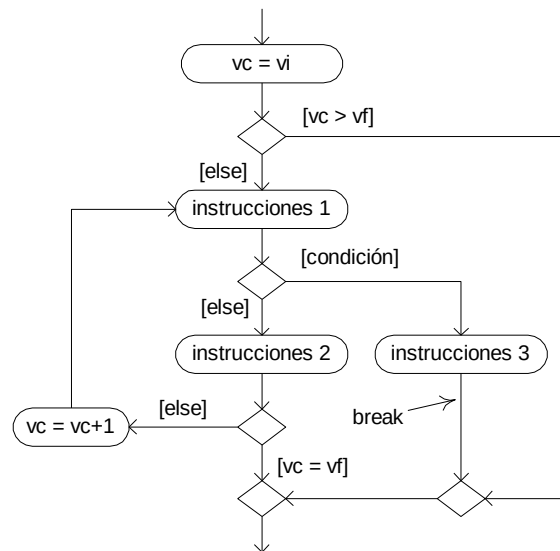
```
repeat
  instrucciones 1;
if condición 1 then
  begin
    instrucciones 3;
    break;
  end;
  instrucciones 2;
until condición 2;
```

Al igual que con "While", con frecuencia no existe la condición de la estructura "Until", sino que se trata de un ciclo que se repite hasta que se cumple alguna condición intermedia, tal como se muestra en el diagrama de la anterior página. Para codificar esta lógica con "Until" se escribe "False" en lugar de la condición del ciclo:

```
repeat
  instrucciones 1;
  if condición 1 then break;
  instrucciones 2;
until False;
instrucciones 3;
```

Como se puede observar, en estos casos no existe diferencia en el diagrama de actividades entre las estructura "While" y "Repeat".

Cuando se aplica la modificación "break" a la estructura FOR, la lógica suele ser la que se muestra en el siguiente diagrama de actividades:

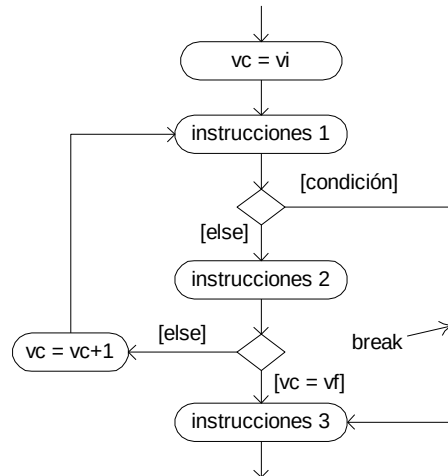


Y la forma de codificar en Pascal es:

```
for vc:=vi to vf do
begin
  instrucciones 1;
  if condición then
  begin
    instrucciones 3;
    break;
  end;
  instrucciones 2;
```

end;

Al igual que con las otras estructuras, con frecuencia la lógica suele ser un tanto más simple:



En cuyo caso se codifica de la siguiente forma:

```

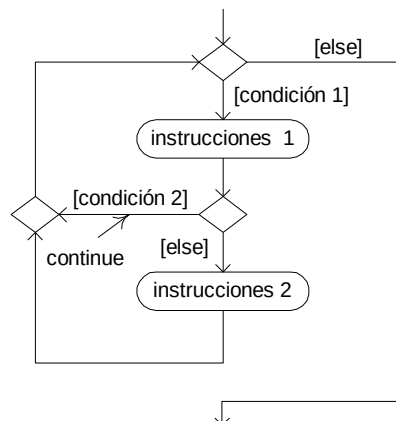
for vc:=vi to vf do
begin
  instrucciones 1;
  if condición then break;
  instrucciones 2;
end;
instrucciones 3;
  
```

La modificación "break" es la modificación que con mayor frecuencia se emplea en la práctica.

10.2. Salto al siguiente ciclo iterativo (Continue)

Esta modificación permite continuar con el siguiente ciclo iterativo desde cualquier punto intermedio del mismo. Es menos útil que la modificación "break", porque casi siempre se puede conseguir el mismo resultado con la estructura IF-THEN-ELSE.

Cuando se aplica esta modificación a la estructura WHILE, la lógica suele ser la que se muestra en el siguiente diagrama de actividades:



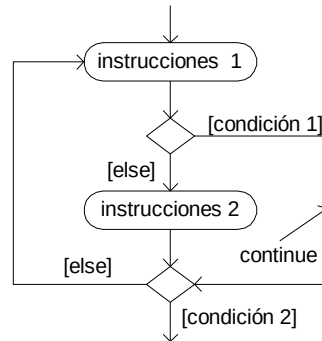
Que en Pascal se codifica de la siguiente forma:


```

while condición 1 do
begin
  instrucciones 1;
  if condición 2 then continue;
  instrucciones 2;
end;

```

Cuando se aplica la modificación "Continue" a la estructura "Until", la lógica suele ser la siguiente:



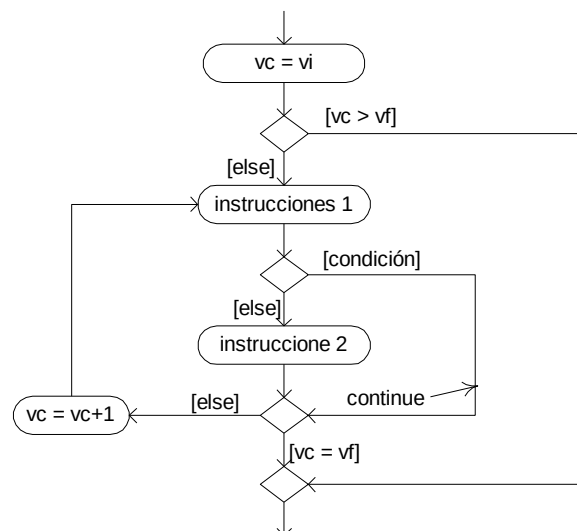
Que en Pascal se codifica como:

```

repeat
  instrucciones 1;
  if condición 1 then continue;
  instrucciones 2;
until condición 2;

```

Cuando se aplica esta modificación a la estructura *FOR* el diagrama de actividades se ve aproximadamente como se muestra en la siguiente figura:



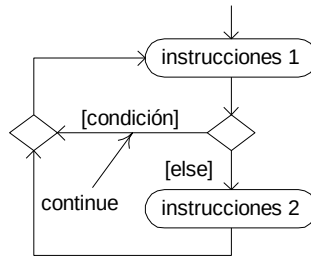
Y la forma de codificarla es:

```

for vc:=vi to vf do
begin
  instrucciones 1;
  if condición then continue;
  instrucciones 2;
end;

```

Esta modificación puede ser empleada también en un ciclo infinito y entonces tiene la apariencia que se muestra en el siguiente diagrama de actividades:



En la práctica, sin embargo, un ciclo infinito debe contar también con la modificación *break*, pues de otra forma se repetiría indefinidamente. La forma de codificar esta modificación con la estructura "While" es:

```

while True do
begin
  instrucciones 1;
  if condición then continue;
  instrucciones 2;
end;

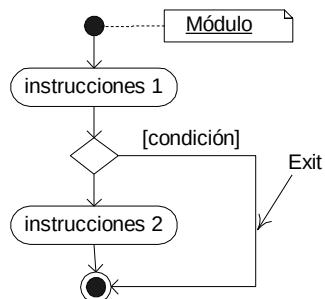
Y con la estructura "Until":
  
```

```

repeat
  instrucciones 1;
  if condición then continue;
  instrucciones 2;
until False;
  
```

10.3. Salida de un módulo (Exit)

Esta modificación permite saltar al final de un módulo desde cualquier punto intermedio del mismo. La lógica de un módulo, en el que se emplea esta modificación, se ve aproximadamente como se muestra en el siguiente diagrama de actividades:



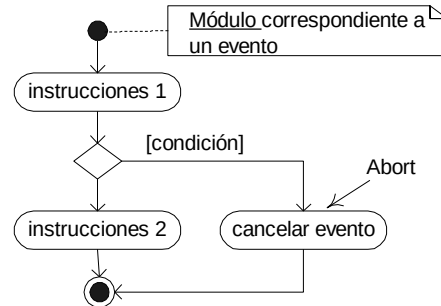
Y se programa en Pascal de la siguiente forma:

```

procedure Modulo;
begin
  instrucciones 1;
  if condición then exit;
  instrucciones 2;
end;
  
```

10.4. Finalización de un módulo (Abort y Raise)

Ya hemos empleado estas modificaciones en capítulos anteriores y como ya se dijo, "Abort" se emplea para terminar la ejecución del módulo correspondiente a un evento. Los módulos que emplean esta modificación tienen una lógica similar al de la figura:



A diferencia de "Exit", Abort no solamente termina la ejecución del módulo, sino también la ejecución del o los módulos que le llamaron. Esta es la razón por la cual se emplea para cancelar los eventos, pues con esta modificación es como si el mismo no hubiese ocurrido.

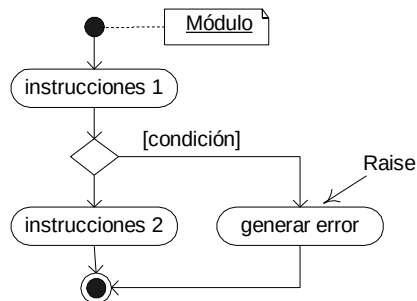
La forma de codificar esta modificación es:

```

procedure Modulo;
begin
  instrucciones 1;
  if condición then abort;
  instrucciones 2;
end;

```

Por otra parte "Raise" se emplea para generar un error y detener la ejecución del módulo, para continuar con el primer bloque de tratamiento de errores: **try ... except ... end** o **try ... finally ... end**, y si no existen terminar la ejecución del módulo y del o los módulos que hicieron la llamada.



La forma de codificar esta modificación es la siguiente:

```

procedure Modulo;
begin
  instrucciones 1;
  if condición then raise Tipo_de_error.Create(Mensaje);
  instrucciones 2;
end;

```

Donde, como recordarán existen varios tipos de errores predefinidos en Delphi y los nombres de todos ellos comienzan con la letra "E".

10.5. Ejemplos

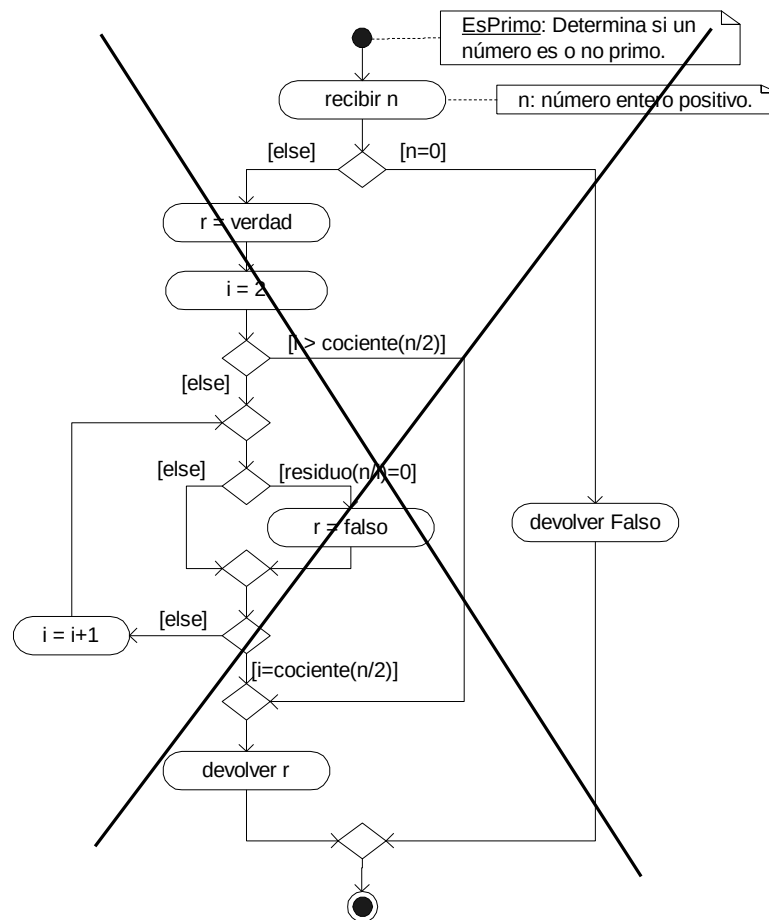
10.5.1. Determinar si un número es o no primo

Como primer ejemplo elaboraremos una aplicación para determinar si un número entero es o no primo.

Recordemos que un número es primo si sólo es divisible entre 1 y entre si mismo o en otras palabras un número no es primo si es divisible entre cualquier número excepto uno y si mismo.

Como ningún número es divisible entre números mayores a su mitad (en realidad números mayores a su raíz cuadrada), para determinar si un número es o no primo se debe comprobar que no sea divisible entre 2, 3, 4 y así hasta llegar a la mitad del número.

Podríamos pensar entonces en un algoritmo como el que se muestra en el siguiente diagrama de actividades:

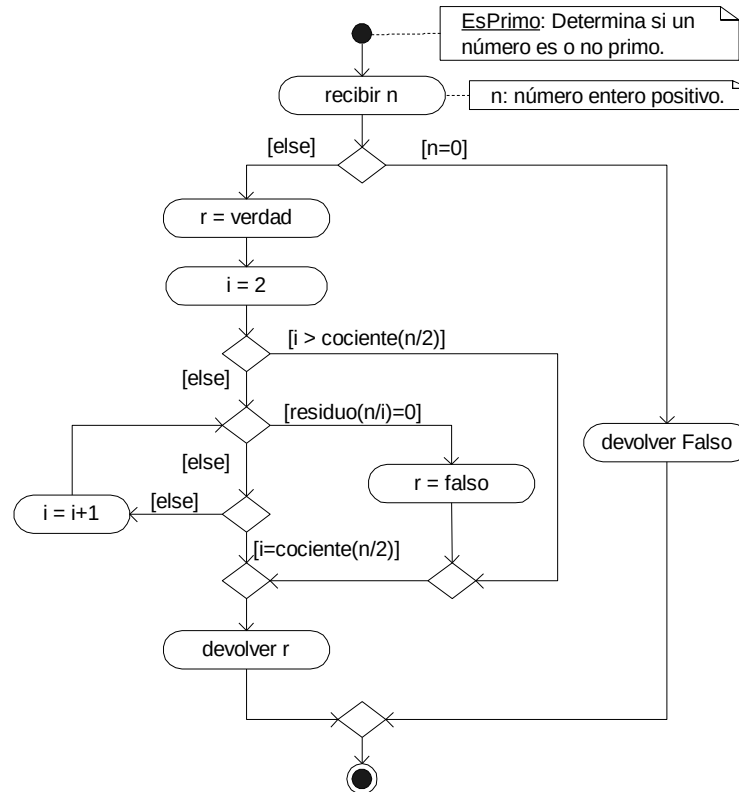


Como se puede observar en este algoritmo, si el número es divisible entre algún número comprendido entre 2 y la mitad del número se asigna el valor falso a la variable lógica "r" y al final del ciclo se devuelve ese valor, caso contrario el valor de la variable "r" permanece en verdadero.

Resulta entonces que el algoritmo "funciona", sin embargo, tiene una lógica absurda. Para ver el porque esta lógica es absurda analicemos como funciona este algoritmo con el número 1000000. Como 1000000 es divisible entre 2, en la primera iteración cambia el valor de la variable "r" a "falso", lo cual es correcto, no obstante el ciclo se repite 499998 veces más volviendo

asignar una y otra vez el valor "falso" a la variable "r", entonces cabe preguntar ¿por qué se repite 499999 veces el ciclo cuando en la primera iteración ya se sabe que el número no es primo? Por supuesto no existe ninguna respuesta razonable a esta pregunta, pues la lógica empleada es absurda.

Por lo tanto el algoritmo debe ser modificado de manera que el ciclo se repita sólo el número de veces que realmente es necesario. Para ello simplemente debemos salir del ciclo cuando sabemos que el número no es primo:



Observe que en estos algoritmos el resultado es de tipo lógico, que es el tipo de dato a la medida para este problema (tercer principio de la P.E.), pues el número es primo (*verdad*) o no es primo (*falso*), no existiendo otras posibilidades.

Ahora creemos la aplicación (File -> New Application) y la unidad (File -> New Unit) y guardemos la aplicación (File -> Save Project As...) en el directorio "Modificación\Primos" con los siguientes nombres: "ufPrimos" para "Unit1", "uPrimos" para "Unit2" y "pPrimos" para "Project1".

Y escribamos el código (incluyendo los módulos de lectura y escritura) en la unidad "uPrimos":

```
unit uPrimos;
```

```
interface
```

```
    uses Mask, SysUtils, QDialogs;
```

```
    function LeerNumero(me: TMaskEdit): cardinal;
```

```
    function EsPrimo(n: cardinal): boolean;
```

```
    procedure MostrarResultado(r: boolean);
```

```
implementation
```

```

function LeerNumero(me: TMaskEdit): cardinal;
begin
    result:= StrToInt(me.Text);
end;

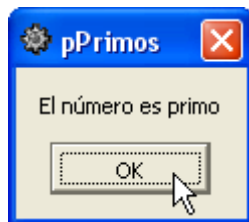
function EsPrimo(n: cardinal): boolean;
var i: cardinal;
begin
    if n=0 then
        result:= False
    else
        begin
            result:= True;
            for i:=2 to n div 2 do
                if n mod i = 0 then begin result:= False; break; end;
            end;
        end;
    end;

procedure MostrarResultado(r: boolean);
begin
    if r then ShowMessage('El número es primo')
    else ShowMessage('El número no es primo');
end;

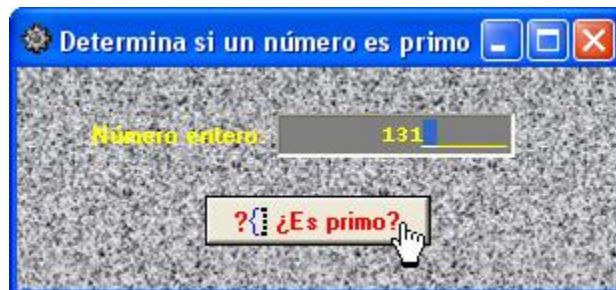
end.

```

En este caso, para mostrar el resultado se está utilizando el procedimiento "ShowMessage", que se encuentra en la unidad "QDialogs", y que muestra una ventana con el mensaje escrito y el botón "OK", como se muestra en la siguiente figura:



Ahora elaboremos la interfaz de la aplicación de manera que, en ejecución, se vea aproximadamente como se muestra en la figura:



En esta interfaz no existe ningún componente nuevo, las propiedades modificadas de los componentes de esta aplicación son:

```

Form1: Name = 'fPrimos'; Caption = 'Determina si un número es primo';
Height = 145; Width = 309; Position = poScreenCenter.

```

```

Label1: Caption = 'Número entero'; Font.Color = clYellow; Font.Style =
[fsBold]; Transparent = True.

```

```
MaskEdit1: Name = 'meNumero'; Font.Color = clYellow; Font.Style =
[fsBold]; Width = 121, EditMask = ' 999999;0;_'; Text=1.
```

```
BitBtn1: Name = 'bbEsPrimo'; Caption = '¿Es Primo?'; Font.Color = clRed;
Font.Style = [fsBold]; Glyph= 'query.bmp', Cursor = crHandPoint; Default =
True.
```

Puesto que el número a probar es introducido en un MaskEdit y la máscara asignada al mismo sólo permite números enteros, no necesitamos validar la introducción de datos.

La imagen del fondo de esta aplicación es "granito.bmp", la cual debe ser copiada al directorio "Modificación\Primos" y asignada a la brocha de la forma en el evento "onCreate" (y liberada en "onClose"):

```
procedure TfPrimos.FormCreate(Sender: TObject);
begin
  fPrimos.Brush.Bitmap:= TBitmap.Create;
  fPrimos.Brush.Bitmap.LoadFromFile('Granito.bmp');
end;
```

```
procedure TfPrimos.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  fPrimos.Brush.Bitmap.Free;
end;
```

Como de costumbre, damos funcionalidad a la aplicación en el evento "on-Click" del BitBtn bbEsPrimo (recordando antes incorporar la unidad "uPrimos"):

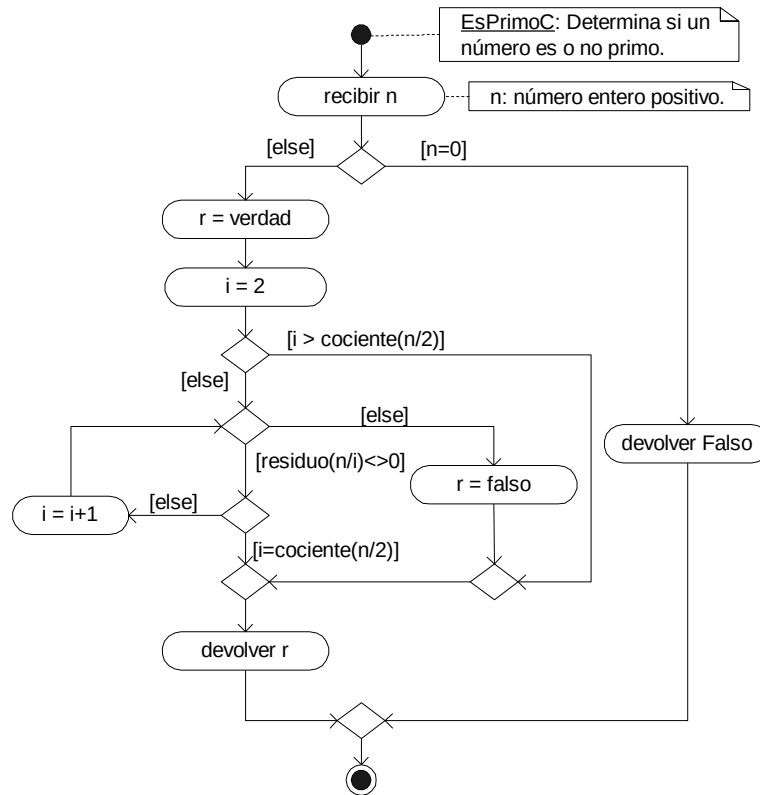
```
procedure TfPrimos.bbEsPrimoClick(Sender: TObject);
var n: cardinal; r: boolean;
begin
  n:= LeerNumero(meNumero);
  r:= EsPrimo(n);
  MostrarResultado(r);
  meNumero.SetFocus;
  meNumero.SelectAll;
end;
```

Una vez compilada la aplicación y corregidos los errores, deberá tener la apariencia mostrada en la figura de la anterior página.

El problema de los números primos puede ser resuelto también con la modificación "continue", de acuerdo a la lógica mostrada en el diagrama de actividades de la siguiente página.

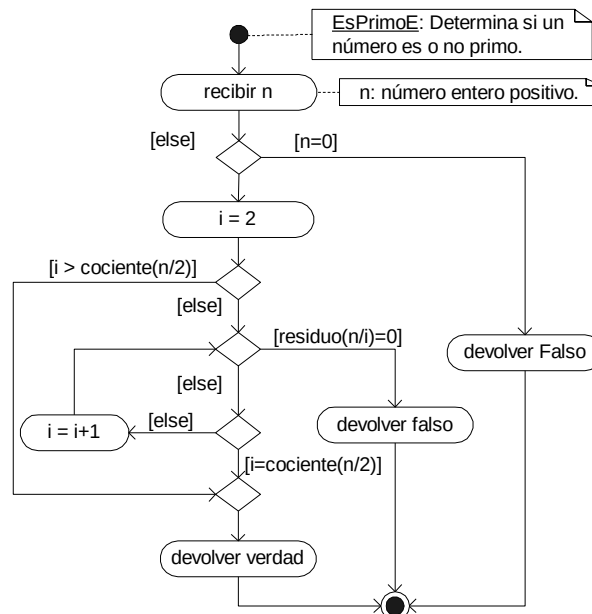
El cual puede ser codificado y añadido a la unidad "uPrimos":

```
function EsPrimoC(n: cardinal): boolean;
var i: cardinal;
begin
  if n=0 then
    result:= False
  else
    begin
      result:= True;
      for i:=2 to n div 2 do begin
        if n mod i <> 0 then continue;
        result:= False; break;
      end;
    end;
end;
```



No debemos olvidar escribir la cabecera de esta función en la **interface** de la unidad. Por supuesto es necesario modificar también el evento "onClick" del *BitBtn* "bbEsPrimo" de manera que llame al procedimiento "EsPrimoC" en lugar de "EsPrimo".

El problema puede ser resuelto también empleando la modificación *Exit*, de acuerdo a la siguiente lógica:



Esta lógica puede ser codificada y añadida a la unidad "uPrimos":

```
function EsPrimoE(n: cardinal): boolean;
```



```

var i: cardinal;
begin
  if n=0 then begin result:= False; exit; end;
  for i:=2 to n div 2 do
    if n mod i = 0 then begin result:= False; exit; end;
  result:= True;
end;

```

Por supuesto, se debe modificar el evento *OnClick* del *BitBtn* "bbEsPrimo" de manera que llame a *EsPrimoE* en lugar de *EsPrimoC* o *EsPrimo*.

10.5.2. Cálculo del arco tangente hiperbólico

Como segundo ejemplo calcularemos el arco tangente hiperbólico empleando la siguiente serie de *Taylor*:

$$\text{ArcTanh}(x) = x + \frac{x^3}{3} + \frac{x^5}{5} + \frac{x^7}{7} + \dots, |x| < 1$$

En este caso sólo se calcula el numerador de los nuevos términos, porque el denominador es simplemente un contador que incrementa de dos en dos. Como se puede observar, el numerador del nuevo término se calcula multiplicando el numerador del término anterior por x^2 .

El algoritmo que resuelve este problema y que esencialmente es el mismo que el elaborado en el anterior capítulo para resolver las series, se presenta en el diagrama de actividades de la siguiente página.

Para codificar el algoritmo creemos una aplicación (File -> New Application) una unidad (File -> New Unit) y guardemos la aplicación (File -> Save Project As...) en el directorio "Modificación\Arco TangenteH" con los siguientes nombres: "ufArcTanH" para "Unit1", "uArcTanH" para "Unit2" y "pArcTanH" para "Project1".

Y escribamos el código (incluyendo los módulos de lectura y escritura) en la unidad "uArcTanH":

```

unit uArcTanH;
interface

  uses StdCtrls, SysUtils, QDialogs;

  function ATanH(x: real): real;
  function CalcularATanH(m: TMemo): real;
  procedure MostrarATanH(m: TMemo; a: real);

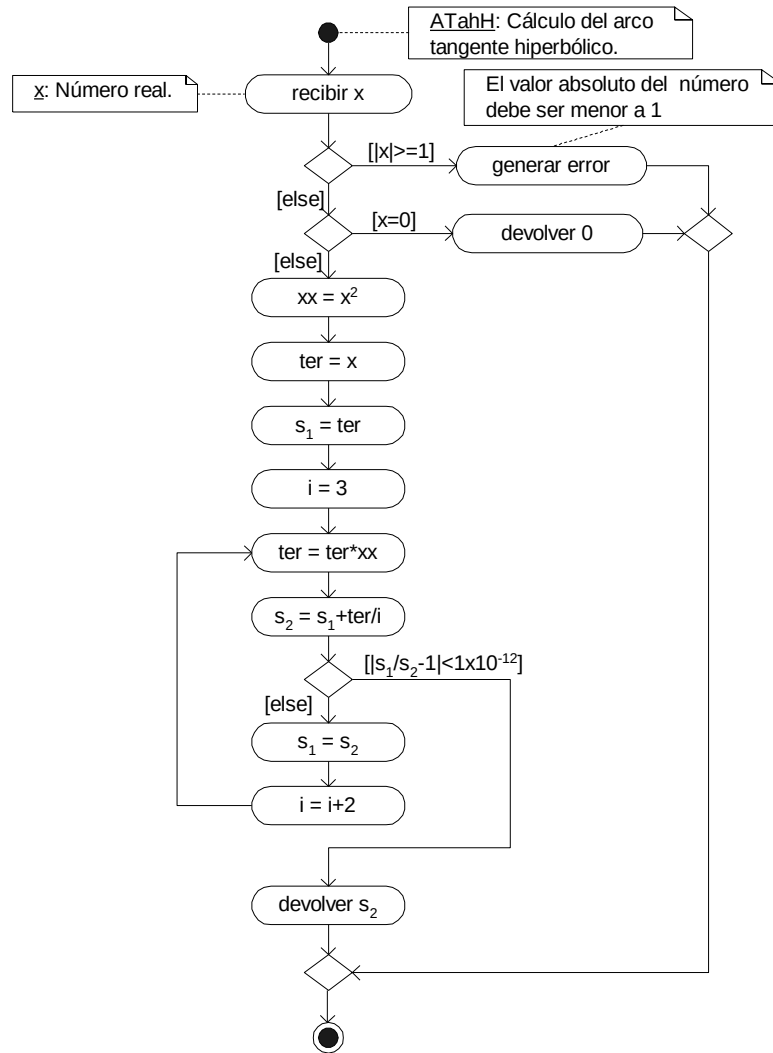
implementation

  function LeerNumero(m: TMemo): real;
  begin
    try
      result:= StrToFloat(m.Text);
    except
      on EConvertError do begin
        MessageDlg('El número está mal escrito',mtError,[mbOK],0);
        m.SetFocus; m.SelectAll;
      end;
    end;
  end;

  function CalcularATanH(m: TMemo): real;
  begin
    result:= LeerNumero(m);
  end;

  procedure MostrarATanH(m: TMemo; a: real);
  begin
    m.Lines.Add(Format('El arco tangente hiperbólico de %f es: %f', [a, result]));
  end;

```



```

function ATanH(x: real): real;
var s1,s2,ter,xx: real; i,c: word;
begin
  c:= Ord(abs(x)>=1)+Ord(x=0)*2;
  case c of
    1: raise EInvalidOp.Create(
      'El valor absoluto del número debe ser menor a cero');
    2: result:= 0;
  else
    xx:= sqr(x); ter:= x; s1:= ter; i:= 3;
    repeat
      ter:= ter*xx;
      s2:= s1+ter/i;
      if abs(s1/s2-1)<1E-12 then break;
      s1:= s2;
      inc(i,2);
    until False;
    result:= s2;
  end;
end;

function CalcularATanH(m: TMemo): real;
begin

```

```

    try
        result:= ATanH(LeerNumero(m));
    except
        on E: EInvalidOp do begin
            MessageDlg(E.Message,mtError,[mbOK],0);
            m.SetFocus; m.SelectAll;
        end;
    end;
end;

procedure MostrarATanH(m: TMemo; a: real);
begin
    m.Text:= FloatToStr(a);
end;

end.

```

En este código aparecen algunos elementos nuevos. En primer lugar estamos tratando los errores con el bloque **try ... except...end:**

```

try
    instrucciones 1;
except
    on variable: tipo_de_error_1 do instrucciones 2;
    on variable: tipo_de_error_2 do instrucciones 3;
    ...
    on variable: tipo_de_error_n do instrucciones n;
end;

```

Donde "instrucciones 1" son las instrucciones que se quiere ejecutar, es decir las instrucciones correspondientes al código normal del módulo. Si se produce algún error al ejecutar alguna de estas instrucciones, el programa salta al código escrito entre "except" y "end" y compara el error producido con cada uno de los tipos de errores escritos entre "on" y "do". Si el error concuerda con uno de estos tipos, entonces se ejecutan las instrucciones escritas a continuación de "do" (las cuales deben estar encerradas entre "begin" y "end" si son dos o más) y termina la ejecución del bloque, de modo similar a la estructura "case".

La "variable" que se escribe antes del tipo de error es opcional y se la emplea cuando se quiere mostrar el mensaje escrito al momento de generar el error, así por ejemplo en la función "LeerNumero" no se emplea una variable porque se escribe un mensaje nuevo, mientras que en la función "CalcularAtanH" se emplea una variable pues se muestra el mensaje escrito en el módulo "AtanH", cuando se genera el error "EInvalidOp".

El otro elemento que aparece en este código es el procedimiento "MessageDlg", el cual se encuentra en la unidad "QDialogs" y nos permite mostrar una ventana con un mensaje, de modo similar a "ShowMessage", pero con más opciones que dicho procedimiento. La sintaxis de este procedimiento es:

MessageDlg(Mensaje, Tipo, [botones], ayuda)

Donde "Mensaje" es el mensaje a mostrar en la ventana; "Tipo" es el tipo de ventana y puede ser: *mtError* (Error), *mtInformation* (Información), *mtWarning* (Peligro), *mtConfirmation* (Confirmación) y *mtCustom* (donde el título es el nombre de la aplicación y no existe icono); "[botones]" es un conjunto con los botones que serán mostrados en la ventana y sus elementos pueden ser: *mbYes*, *mbNo*, *mbOK*, *mbCancel*, *mbAbort*, *mbRetry*, *mbIgnore*, *mbAll*, *mbNoToAll*, *mbYesToAll* y *mbHelp*. "Ayuda" es el número de la ayuda que aparece cuan-

do el usuario pulsa la tecla *F1* (o el botón de ayuda), si no existe un archivo de ayuda (como en nuestro caso) se debe escribir el número cero.

Ahora elaboremos la interfaz de la aplicación de manera que, en ejecución, se vea aproximadamente como se muestra en la figura:



Las propiedades modificadas de los componentes de esta aplicación son:

Icono de la aplicación: technlgy.ico;

Form1: Name = 'fArcTanH'; Caption = 'Arco Tangente Hiperbólico'; Height = 158; Width = 296; Position = poScreenCenter.

Label1: Caption = 'Número real:'; Label2: Caption = 'Arco Tangente Hiperbólico'; Propiedades comunes: Font.Color = clBlue; Transparent = True; Alignment = taRightJustify.

Memo1: Name = 'mNumReal'; Cursor = crIBeam; Memo2: Name = 'mArcTanH'; Cursor = crArrow; ReadOnly = True; TabStop = False; Propiedades comunes: Font.Color = clBlue; Height = 21; Width = 130, WantReturns = False; Alignment = taRightJustify.

BitBtn1: Name = 'bbCalcular'; Caption = '&Calcular'; Font.Color = clBlue; Glyph= 'compmac.bmp', Cursor = crHandPoint; Default = True.

Asegúrese también que el orden de los componentes sea (Edit -> Tab Order): mNumReal, mArcTanH y bbCalcular.

Cambiamos entonces el estilo de la brocha e inicializamos el texto de los memos en el evento "onCreate" de la forma:

```
procedure TfArcTanH.FormCreate(Sender: TObject);
begin
  fArcTanH.Brush.Color:= clNavy;
  fArcTanH.Brush.Style:= bsBDiagonal;
  mNumReal.Text:= '0';
  mArcTanH.Text:= '0';
end;
```

Damos el foco y seleccionamos el texto del memo "mNumReal" en el evento "onActivate" de la forma:

```
procedure TfArcTanH.FormActivate(Sender: TObject);
begin
  mNumReal.SetFocus;
  mNumReal.SelectAll;
end;
```

Validamos la introducción de datos al memo "mNumReal" en el evento "onKeyPress":

```
procedure TfArcTanH.mNumRealKeyPress(Sender: TObject; var Key: Char);
begin
  case Key of
```

```

#8,'0'..'9','.', '+', '-', 'E', 'e':
else Beep; Abort;
end;
end;

```

Finalmente damos funcionalidad a la aplicación en el evento "onClick" del botón "bbCalcular":

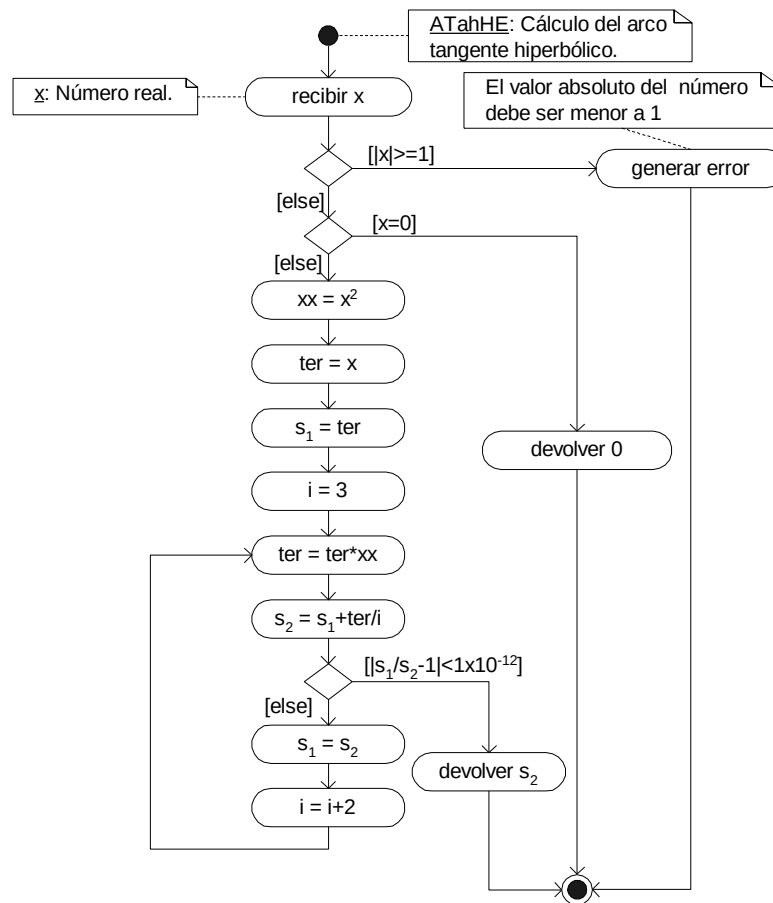
```

procedure TfArcTanH.bbCalcularClick(Sender: TObject);
var a: real;
begin
  a:= CalcularAtanH(mNumReal);
  MostrarATanH(mArcTanH,a);
  mNumReal.SetFocus;
  mNumReal.SelectAll;
end;

```

Entonces compilamos la aplicación (Ctrl+F9), corregimos los errores, guardamos los cambios (File -> Save All) y al hacer correr la aplicación debe tener la apariencia y calcular el resultado mostrado en la figura de la anterior página.

Como casi siempre ocurre, es posible resolver el problema empleando otra modificación, en este caso por ejemplo podemos emplear la modificación "Exit" de acuerdo a la lógica que se muestra en el diagrama de actividades:



Cuyo código es:

```

function ATanHE(x: real): real;
var s1,s2,ter,xx: real; i: word;
begin

```

```

if abs(x)>=1 then raise EInvalidOp.Create(
    'El valor absoluto del número debe ser menor a cero');
if x=0 then begin result:= 0; exit; end;
xx:= sqr(x); ter:= x; s1:= ter; i:= 3;
repeat
    ter:= ter*xx;
    s2:= s1+ter/i;
    if abs(s1/s2-1)<1E-12 then begin result:= s2; exit; end;
    s1:= s2;
    inc(i,2);
until False;
end;

```

El cual puede ser probado simplemente cambiando la llamada a "ATanH" por "ATanHE" en el módulo "CalcularATanH".

10.6. Cálculo del exponente de un número real

Como tercer ejemplo, calcularemos el exponente de un número real empleando la serie de Taylor:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots, \quad -\infty < x < \infty$$

La lógica para resolver este problema es la misma que se sigue en todas las series: a) inicializar variables; b) calcular el nuevo término y el nuevo valor de la serie; c) comparar los dos últimos valores calculados y si son iguales en un determinado número de dígitos terminar el proceso, caso contrario hacer cambio de variables, incrementar contadores y repetir el proceso desde el paso (b). No obstante, en este caso se debe tomar una medida adicional porque cuando "x" es negativo los términos con potencias impares son negativos y, como ya se dijo, cuando en una serie aparecen restas los errores por redondeo incrementan considerablemente, a tal punto que el resultado calculado puede ser del todo erróneo, a pesar de que la lógica sea correcta. En este caso afortunadamente podemos evitar trabajar con números negativos: cuando "x" es negativo, se calcula el valor del exponente con el valor absoluto de "x", siendo la solución la inversa del valor calculado, pues se sabe que "e^{-x}" es igual a "1/e^x". El diagrama de actividades con la lógica que resuelve el problema, tomando en cuenta la anterior consideración, se presenta en la siguiente página.

Para codificar esta algoritmo creemos una nueva aplicación (File -> New Application), una nueva unidad (File -> New Unit) y guardemos la aplicación en el directorio "Modificación\Exponente" con los siguientes nombres: "ufExponente" para "Unit1", "uExponente" para "Unit2" y "pExponente" para "Project1".

Escribamos el código (incluyendo los módulos de lectura y escritura) en la unidad "uExponente":

```

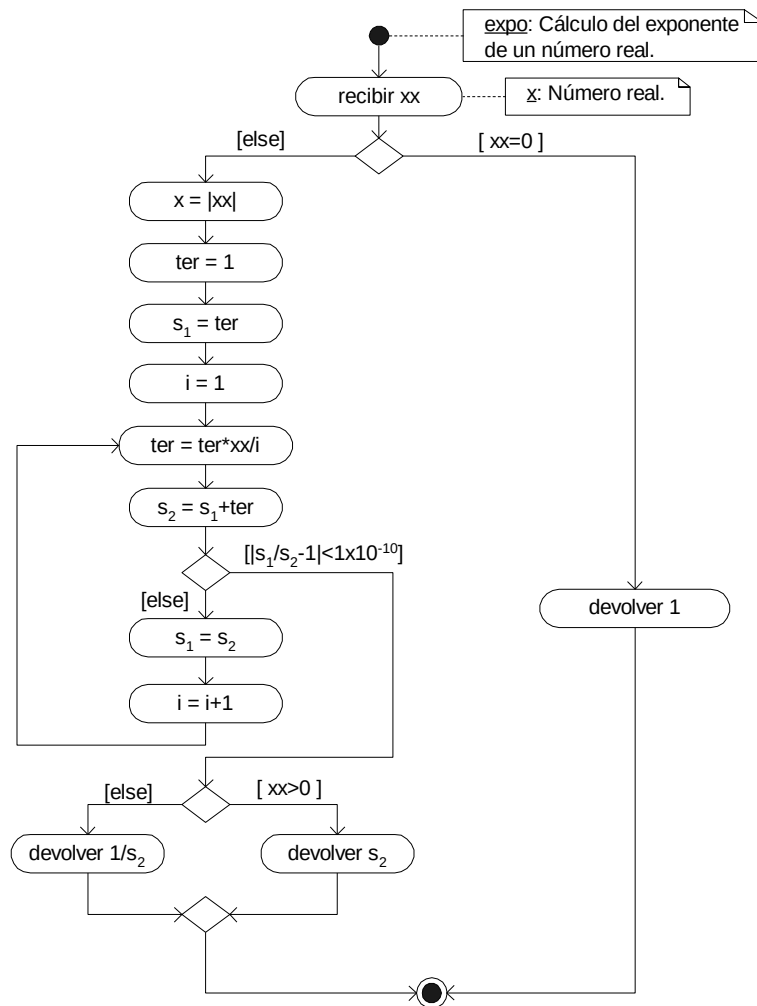
unit uExponente;

interface

    uses StdCtrls, SysUtils;

    function LeerNumReal(m: TMemo): extended;
    function Expo(xx: extended): extended;
    procedure MostrarExponente(m: TMemo; e: real);

```



implementation

```

function LeerNumReal(m: TMemo): extended;
begin
  try
    result:= StrToFloat(m.Text);
  except
    on EConvertError do begin
      MessageDlg('El número está mal escrito',mtError,[mbOK],0);
      m.SetFocus; m.SelectAll;
    end;
  end;
end;

function Expo(xx: extended): extended;
var x,ter,s1,s2: extended; i: word;
begin
  if xx=0 then begin result:= 0; exit; end;
  x:= abs(xx); ter:= 1; s1:= ter; i:= 1;
  while true do begin
    ter:= ter*x/i; s2:= s1+ter;
    if abs(s1/s2-1)<1E-14 then break;
    s1:= s2; inc(i);
  end;
end;

```

```

    if xx>0 then result:= s2 else result:= 1/s2;
end;

procedure MostrarExponente(m: TMemo; e: real);
begin
    m.Text:= FloatToStr(e);
end;

end.

```

Elaboremos ahora la aplicación de manera que en ejecución tenga la apariencia que se muestra en la siguiente figura:



Las propiedades modificadas de los componentes de esta aplicación son:

Icono de la aplicación: 'finance.ico';

Form1: Name = 'fExponente'; Caption = 'Exponente de un número real'; Height = 191; Width = 296; Position = poScreenCenter.

Label1: Caption = 'Número real:'; Label2: Caption = 'Exponente'; Propiedades comunes: Font.Color = clGreen; Font.Style = [fsBold]; Transparent = True; Alignment = taRightJustify.

Memo1: Name = 'mNumReal'; Cursor = crIBeam; Memo2: Name = 'mExponente'; Cursor = crArrow; ReadOnly = True; TabStop = False; Propiedades comunes: Font.Color = clGreen; Font.Style = [fsBold]; Height = 21; Width = 150; WantReturns = False; Alignment = taRightJustify.

BitBtn1: Name = 'bbCalcular'; Caption = '&Calcular'; Font.Color = clGreen; Font.Style = [fsBold]; Glyph= 'calculat.bmp', Cursor = crHandPoint; Default = True.

Asegúrese también que el orden de los componentes sea (Edit -> Tab Order): mNumReal, mExponente y bbCalcular.

Asignamos una imagen a la brocha de la forma e inicializamos el texto de los memos en el evento "onCreate" (recuerde copiar el archivo "Mármol blanco.bmp" al directorio "Modificación\Exponente"):

```

procedure TfExponente.FormCreate(Sender: TObject);
begin
    fExponente.Brush.Bitmap:= TBitmap.Create;
    fExponente.Brush.Bitmap.LoadFromFile('Mármol blanco.bmp');
    mNumReal.Text:= '0';
    mExponente.Text:= '1';
end;

```

Liberamos la imagen en el evento "onClose":

```

procedure TfExponente.FormClose(Sender: TObject; var Action: TCloseAction);

```


begin

fExponente.Brush.Bitmap.Free;

end;

Damos el foco y seleccionamos el texto del memo "mNumReal" en el evento "onActivate":

procedure TfExponente.FormActivate(Sender: TObject);

begin

mNumReal.SetFocus; mNumReal.SelectAll;

end;

Validamos la introducción de datos en el evento "onKeyPress" del memo "mNumReal":

procedure TfExponente.mNumRealKeyPress(Sender: TObject; **var** Key: Char);

begin

if not (Key in [#8,'0'..'9','.', '-', '+', 'e', 'E']) **then begin** Beep; Abort;

end;

Finalmente damos funcionalidad a la aplicación en el evento "onClick" del BitBtn "bbCalcular":

procedure TfExponente.bbCalcularClick(Sender: TObject);

var x,e: double;

begin

x:= LeerNumReal(mNumReal);

e:= Expo(x);

MostrarExponente(mExponente,e);

mNumReal.SetFocus; mNumReal.SelectAll;

end;

Una vez compilada la aplicación (Ctrl+F9) y corregidos los errores, la interfaz deberá verse y calcular el resultado mostrado en la figura de la anterior página.

10.7. Preguntas y ejercicios

1. Elabore una aplicación con un memo y escriba el evento "onKeyPress" del memo de manera que sólo permita escribir números enteros positivos y negativos.
2. Elabore una aplicación con un memo (Memo1) y escriba los eventos "onCreate" y "onClose" de la forma de manera que su fondo sea la figura "papel periódico.bmp", que el Memo tenga: 100 puntos de ancho y 150 de alto, con los días de la semana en color rojo, centrados (uno en cada línea), con los estilos negrita y cursiva y que no permita escribir ningún dato en el mismo.
3. Elabore una aplicación con un RadioGroup y escriba el evento "onCreate" de la forma de manera que su fondo sea de color rojo y con líneas diagonales cruzadas, el RadioGroup tenga los meses del año en tres columnas, que al empezar la aplicación esté seleccionado el sexto mes y que al hacer click con el botón derecho del mouse seleccione el siguiente mes (seleccionando el primero si ya no existen más meses).
4. Elabore una aplicación con un "BitBtn" y escriba los eventos "onCreate" y "onClose" de la forma de manera que su fondo sea la figura "granito.bmp", el BitBtn tenga la figura "Sort.bmp", el título "&Calcular", se active al pulsar la tecla "Enter" y que la forma del puntero cambie a al cursor de escritura cuando se encuentre sobre el botón.

5. Elabore una aplicación y en la misma escriba un módulo que, siguiendo la lógica natural, determine si un número es o no perfecto. Pruebe el módulo en el evento `onClick` de la forma con los números 6 y 35, mostrando los resultados en una ventana de mensajes.
6. Elabore una aplicación y en la misma elabore un módulo que, siguiendo la lógica natural, calcule el Legendre enésimo de un número real. Pruebe el módulo en el evento `onDblClick` de la forma con "`x=3.2, n=4`" y "`x=6.2 y n=3`" mostrando los resultados en una ventana de diálogo (con botones Aceptar y Cancelar).
7. Elabore una aplicación y en la misma escriba el módulo "EsPrimo" que determine si un número entero positivo es o no primo. Pruebe el módulo en el evento "`onClick`" de la forma, con los números 151 y 153, mostrando los resultados en una ventana de mensajes.
8. Elabore una aplicación y en la misma escriba el módulo "ATanH", que siguiendo la lógica natural, calcule el arco tangente hiperbólico de un número real empleando la serie de Taylor: $\tanh^{-1}(x) = x + x^3/3 + x^5/5 + x^7/7 + \dots$; $|x| < 1$. Pruebe el módulo con los números 0.56 y 0.73, mostrando los resultados del módulo, y los obtenidos con `ArcTanh`, en una ventana de diálogo (con botones "OK" y "Cancel").
9. Elabore una aplicación y en la misma escriba el módulo "Senh" que, siguiendo la lógica natural y la modificación `exit`, calcule el seno hiperbólico empleando la serie: $\sinh(x) = x + x^3/3! + x^5/5! + x^7/7! + \dots$. Pruebe el módulo con "`x=3.2`" y "`x=12.5`" mostrando los resultados del módulo, y los obtenidos con "Sinh", en una ventana de mensajes.
10. Elabore una aplicación y en la misma escriba el módulo "Coseno" que siguiendo la lógica natural calcule el coseno de un ángulo en radianes empleando la fórmula: $\cos(x) = 1 - x^2/2! + x^4/4! - x^6/6! + \dots$. Pruebe el módulo con "`x=7.89`" y "`x=6034.44`", mostrando los resultados obtenidos con el módulo, y los obtenidos con "Cos", en una ventana de diálogo con los botones: Si, Ignorar y Cancelar.
11. Elabore una aplicación y en la misma escriba el módulo "RCuad" que siguiendo la lógica natural calcule la raíz cuadrada de un número real "n" con la fórmula de Newton: $x_2 = (x_1 + n/x_1)/2$. El módulo debe generar el error `EInvalidOp`, con un mensaje adecuado, si el número real es negativo. Pruebe el módulo en el evento "`onContextPopUp`" de la forma, con los números 5.6 y -7.64, mostrando los resultados (o los mensajes de error) obtenidos con el módulo y con "Sqrt", en una ventana de mensajes.
12. Elabore el módulo "Cubica" que siguiendo la lógica natural calcule una de las soluciones de la ecuación cúbica: $ax^3 + bx^2 + cx + d = 0$, con la ecuación de Newton: $x_2 = (2ax_1^3 + bx_1^2 - d) / (3ax_1^2 + 2bx_1 + c)$. El valor inicial asumido "`x1`" debe ser un parámetro opcional, con un valor por defecto igual a 1.1. En el módulo se debe contar el número de veces que se repite el ciclo y si el mismo llega a 30, debe generar el error `EInvalidArgument` indicando que el método no está convergiendo. Pruebe el módulo en el evento "`onClick`" de la forma, con las ecuaciones $x^3 + 2x^2 + 3x + 4 = 0$ y $5x^3 - 9x^2 - 8x + 5 = 0$, mostrando en una ventana de diálogo (con los botones Si y No) el resultado o el mensaje de error generado.

11. RECURSIVIDAD

En este tema estudiaremos otro método que podemos emplear para resolver problemas iterativos: la recursividad.

La recursividad no es imprescindible para resolver problemas iterativos, sin embargo, en ocasiones, los problemas pueden ser resueltos de manera más clara y sencilla empleando este método.

Puesto que una solución más sencilla es también más entendible, el empleo de estos métodos puede facilitar el mantenimiento y actualización de los programas. Recordemos que la programación estructura ha surgido justamente con la intención de facilitar el mantenimiento y actualización de programas, por lo tanto cualquier método o técnica que facilite la elaboración y mantenimiento de programas debe ser tomado en cuenta y utilizado si cumple con ese propósito.

En consecuencia el objetivo del presente tema es que al concluir el mismo estén capacitados para resolver problemas iterativos aplicando el razonamiento recursivo.

11.1. RECURSIVIDAD

En general se dice que un término o concepto es recursivo si en su definición se emplea el término o concepto que se está definiendo.

Aplicado a la programación estructurada, se dice que un módulo es recursivo si en su solución se llama a si mismo.

11.2. Análisis recursivo

La recursividad implica sobre todo otra forma de pensar: cuando llevamos a cabo el análisis recursivo de un problema pensamos como obtener el resultado empleando un resultado devuelto por el mismo módulo. Por el contrario, cuando efectuamos el análisis no recursivo de un problema iterativo pensamos en cuantas veces o hasta cuando debemos repetir un determinado proceso.

Así por ejemplo, para calcular el factorial de un número entero el razonamiento no recursivo es el siguiente: inicializar un contador en 1 y repetir un ciclo desde 1 hasta n, donde en cada repetición se multiplica la variable por el contador del ciclo, tal como se hizo en el anterior capítulo.

Con el razonamiento recursivo, pensamos en como calcular el factorial empleando un resultado devuelto por el mismo módulo. Para ello, en este caso tomamos en cuenta que el factorial de un número puede ser calculado con:

$$n! = (n-1)! * n$$

Así el factorial de 5 se calcula multiplicando el factorial de 4 por 5 ($5! = 4! * 5$). Se sabe por definición que el factorial de 0 es 1, esta condición constituye la condición de finalización y siempre debe existir una condición de finalización en un módulo recursivo, pues de lo contrario el proceso se repetiría indefinidamente (o hasta que se acabe la memoria).

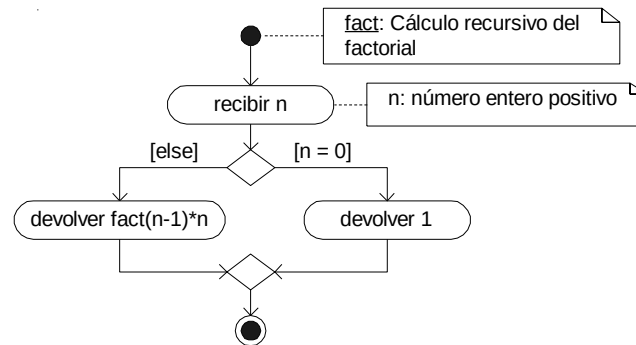
Entonces, para calcular el factorial de un número, de manera recursiva, simplemente calculamos el factorial del número anterior (llamando al mismo módulo) y multiplicamos el resultado por el número cuyo factorial queremos calcular, siendo la condición de finalización "n=0", para la cual se sabe que el factorial es 1. Al resolver el problema de manera recursiva sólo debemos preocuparnos de dos cosas: a) que la fórmula o procedimiento recursivo sea correcto, es decir que realmente resuelva el problema con la llamada o

llamadas recursivas y b) que exista una condición de finalización, es decir un valor o respuesta conocido para algún valor o valores dados.

Así por ejemplo en el caso del factorial debemos verificar que la fórmula recursiva sea correcta, es decir que el factorial de "n" realmente pueda ser calculado con $(n-1)! * n$, lo cual es fácil de verificar con algún valor de prueba, por ejemplo $n=5$, para el cual la fórmula es $5! = 4! * 5 = 24 * 5 = 120$, que como vemos devuelve el resultado correcto. Por otra parte debemos verificar que la condición de finalización sea alcanzable y sea correcta, en este caso se sabe (por definición) que el factorial de "0" es "1", por lo que la condición de finalización es correcta y es además alcanzable, pues en la fórmula recursiva se hace la llamada con $(n-1)$, por lo que en algún momento ese valor será cero.

Un error que generalmente se comete cuando se resuelve un problema de manera recursiva es el tratar de resolver y comprobar todo el proceso recursivo, en lugar de verificar sólo la validez del último proceso o fórmula recursiva. Así por ejemplo en el caso del factorial es suficiente comprobar que la fórmula recursiva devuelva el valor correcto, pero no debe preocuparnos cómo se calcula luego el factorial de "n-1" y después el de "n-2" y así sucesivamente, pues si la fórmula recursiva es correcta, con seguridad el resultado final también será correcto, pues para resolver el problema, el módulo recursivo simplemente se llama a si mismo "n" veces, por lo tanto repite "n" veces la misma lógica (o fórmula).

El diagrama de actividades para el cálculo recursivo del factorial es el siguiente:



Siendo el código respectio el siguiente:

```

proc fact n {
  if {= $n 0} {return 1.} else {* [fact [- $n 1]] $n}
}

```

```

function fact(n: word): extended;
begin
  if n<2 then result:= 1 else result:= fact(n-1)*n;
end;

```

Que evidentemente es más sencillo que la solución no recursiva, no obstante, desde el punto de vista de la eficiencia, las soluciones recursivas son siempre menos eficientes pues consumen más tiempo y memoria que las soluciones no recursivas.

La única razón por la cual se emplea la recursividad es porque facilita la resolución de algunos problemas iterativos.

Para comprender el porqué la recursividad es menos eficiente analicemos lo que sucede cuando llamamos al módulo recursivo para calcular el factorial de 5:

En la primera llamada (no recursiva) a *fact* el parámetro *n* toma el valor 5 y se ejecuta el código:

$$fact(n-1)*n = fact(4)*5$$

Que origina una llamada recursiva con *n*=4:

$$fact(n-1)*n = fact(3)*4$$

Que una vez más origina otra llamada recursiva con *n*=3:

$$fact(n-1)*n = fact(2)*3$$

Dando lugar así a una nueva llamada recursiva con *n*=2:

$$fact(n-1)*n = fact(1)*2$$

Que genera otra llamada recursiva con *n*=1:

$$fact(n-1)*n = fact(0)*2$$

Ahora se genera una nueva llamada recursiva con "*n*=0", pero en este caso, como la condición es verdadera devuelve el resultado 1:

$$fact(0)*1 = 1*1 = 1$$

Este resultado es empleado en el módulo que hizo la llamada con "*n*=1":

$$fact(1)*2 = 1*2 = 2$$

Y este resultado es empleado en el módulo que hizo la llamada con "*n*=2":

$$fact(2)*3 = 2*3 = 6$$

Una vez más este resultado es empleado en el módulo que hizo la llamada con "*n*=3":

$$fact(3)*4 = 6*4 = 24$$

Finalmente este resultado es empleado en el primero módulo que inició las llamadas recursivas con "*n*=4":

$$fact(4)*5 = 24*5 = 120$$

Y este resultado es devuelto al módulo que hizo la llamada original, por lo tanto al final del proceso recursivo el resultado devuelto es "120", que corresponde es el factorial de 5.

Como se puede observar el proceso es moroso, pero además consume recursos pues cada vez que un módulo se llama a si mismo se crea un nuevo conjunto de variables para el módulo. Por otra parte, cada vez que un módulo termina, el sistema debe encargarse de liberar la memoria ocupada. Estas actividades consumen tiempo y recursos siendo la razón por la cual las soluciones recursivas son menos eficientes.

Para comprender mejor como opera la recursividad es aconsejable que haga correr los módulos recursivos paso a paso (con F7), visualizando la ventanas de llamadas: "View->Debug Windows->Call Stack" (o Ctrl+Alt+S) y la ventana de variables locales "View->Debug Windows->Local Variables" (o Ctrl+Alt+L).

11.3. Algoritmos recursivos erróneos

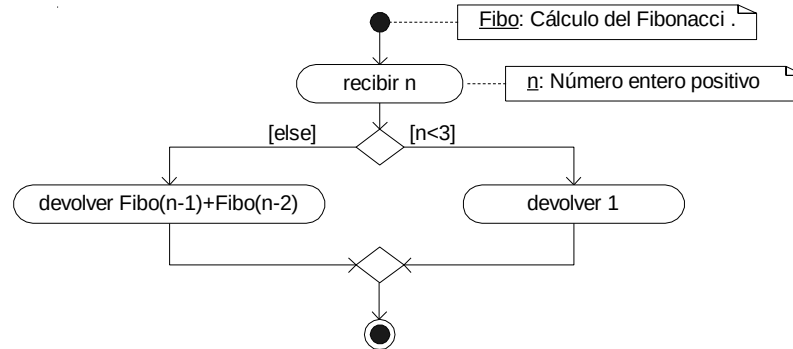
La recursividad, al igual que la lógica directa puede dar lugar a algoritmos erróneos y generalmente, en el caso recursivo, resulta más difícil detectar este tipo de errores.

Como ejemplo de una lógica errónea analicemos el cálculo del Fibonacci de un número entero positivo.

$$F_n = F_{n-1} + F_{n-2}$$

$$F_1 = F_2 = 1$$

Que como vemos es una definición recursiva. Entonces podríamos pensar en un algoritmo como el siguiente:



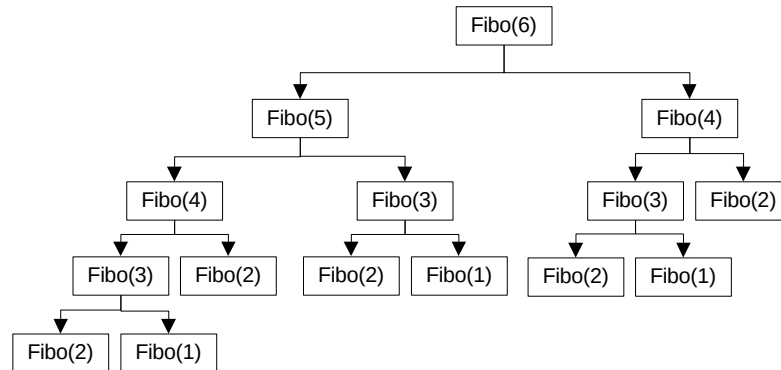
Siendo el código respectivo:

```

function fibo(n: word): extended;
begin
  if n<3 then result:= 1 else result:= fibo(n-1)+fibo(n-2);
end;

```

A pesar de ser una solución sencilla y devolver los resultados correctos, se trata de una lógica errónea. Para comprender el por qué la lógica es errónea, analicemos las llamadas recursivas que se llevan a cabo para calcular el *Fibonacci* de 6:

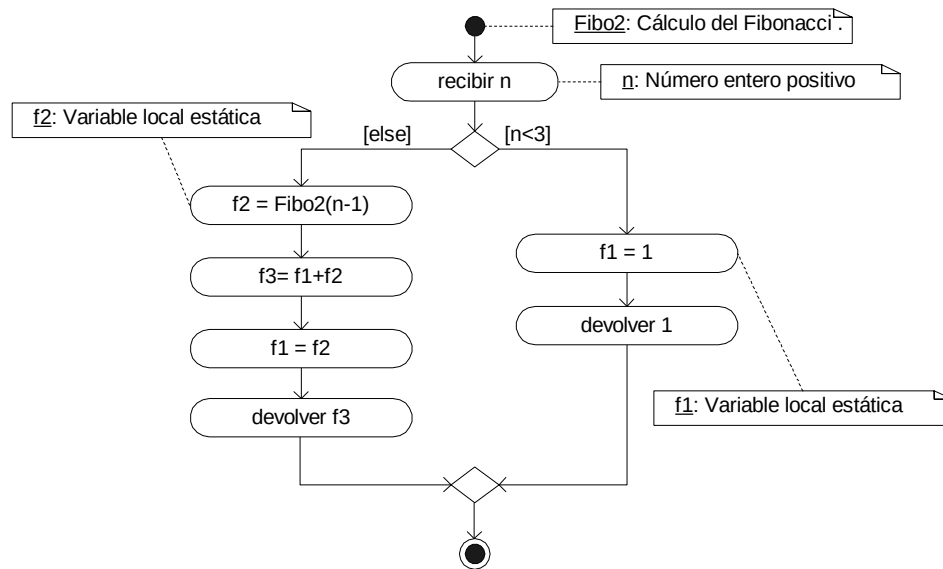


Como se puede observar, para calcular el Fibonacci de 6 se realizan 3 llamadas recursivas para calcular el Fibonacci de 1; 5 llamadas recursivas para calcular el Fibonacci de 2; 3 llamadas para el Fibonacci de 3; 2 llamadas para el Fibonacci de 4 y una llamada para el Fibonacci de 5.

En total se efectúan 14 llamadas recursivas, en las cuales se vuelven a calcular, de manera ilógica, valores que ya fueron calculados previamente. Si la lógica fuera correcta serían suficientes un máximo de 6 llamadas.

Este problema incrementa geométricamente a medida que incrementa el número del Fibonacci. Así por ejemplo para calcular el Fibonacci de 15 se requiere 1218 llamadas recursivas, para calcular el Fibonacci de 17 son necesarias 3192 llamadas recursivas, para calcular el Fibonacci de 30 se requieren 1664078 llamadas recursivas y para calcular el Fibonacci de 34 11405772 llamadas recursivas. En este último caso por ejemplo, con la lógica correcta serían necesarias un máximo de 35 llamadas, por lo tanto se realizan más de 11 millones de llamadas innecesarias, algo que por supuesto es ilógico.

Una lógica recursiva más eficiente aunque evidentemente menos sencilla es la siguiente:



Como se puede observar en este caso la solución recursiva no es más sencilla que la solución no recursiva, razón por la cual, no se justifica emplear la recursividad para resolver este problema (recuerde que la única razón para el empleo de la recursividad es la facilidad con la que se puede resolver el problema). Este es uno de los problemas donde aun cuando la definición es recursiva, la solución es por naturaleza no recursiva.

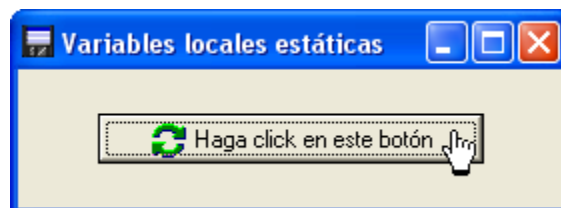
Como se puede observar, en esta lógica se emplean dos variables locales estáticas: "f1" y "f2". Recordemos que "las variables locales estáticas son aquellas que conservan su valor entre llamadas", son una especie de híbridos entre las variables globales y las variables locales, se comportan como variables globales en el sentido de que no pierden su valor cuando el módulo concluye, pero se comportan como variables locales en el sentido de que sólo pueden ser empleadas dentro del módulo.

Las variables locales estáticas toman su valor inicial la primera vez que se llama al módulo, posteriormente conservan el último valor que se les asigna, esta es la razón por la cual se asigna el valor "1" a la variable estática "f1", antes de concluir el módulo. Las variables locales estáticas sólo pierden su valor cuando el programa concluye.

Recordemos también que en *Pascal* las variables locales estáticas se declaran como constantes inicializadas:

const nombre: tipo = valor inicial;

Y que en Delphi 7.0 es necesario habilitar la directiva de compilación: "Assignable typed constants" (Project -> Options -> Compiler) para que permita el uso de variables locales estáticas. Para comprender mejor el funcionamiento de las variables locales estáticas elaboremos una pequeña aplicación cuya interfaz es la siguiente:



Y que tiene el siguiente evento:

```
procedure TForm1.BitBtn1Click(Sender: TObject);
const n: integer = 0;
begin
  inc(n);
  if n=1 then
    ShowMessage('Ha hecho click en el botón una vez')
  else
    ShowMessage('Ha hecho click en el botón '+IntToStr(n)+' veces');
end;
```

Donde, como se puede observar, se ha declarado la variable local estática: "n". Al hacer correr esta aplicación, cada vez que se hace clic en el botón, aparece un mensaje indicando cuantas veces ya se ha hecho clic en el mismo. Para ver la diferencia entre una variable local estática y una variable local normal (dinámica), modifique el evento reemplazando la constante por una variable local (*var n: integer;*) y haga correr el programa, entonces observará que al hacer clic en el botón aparece siempre el mismo mensaje, sin importar cuantas veces se haga clic sobre el botón (esto es así porque las variables normales "dinámicas" se crean cada vez que se llama al módulo y son destruidas cada vez el módulo termina).

Ahora que comprendemos mejor como funcionan las variables locales estáticas, podemos codificar y entender mejor el algoritmo recursivo propuesto en la anterior página:

```
function fibo2(n: word): extended;
const f1: extended = 0; f2: extended = 0;
begin
  if n<3 then
    begin f1:=1; result:= 1 end
  else
    begin f2:= fibo2(n-1); result:= f1+f2; f1:= f2; end;
end;
```

Como ya se dijo, para verificar la lógica recursiva sólo se debe analizar si la condición de finalización es correcta y si el último proceso recursivo devuelve el resultado esperado. En este caso verificamos fácilmente que la condición de finalización es correcta, pues se sabe que el Fibonacci de 1 y 2 (números menores a 3) es 1, y que esta condición será alcanzada en algún momento, porque cada una de las llamadas recursivas disminuye el valor del número original en 1. Lo que no resulta muy claro es el por qué se asigna el número "1" a la variable "f1", sin embargo, esto se aclara cuando tomamos en cuenta que "f1" es una variable local estática y que en la misma se guarda el valor del penúltimo Fibonacci calculado, el cual, para la condición $n < 3$, es el Fibonacci de 1 y como sabemos el Fibonacci de 1 es 1.

Ahora verifiquemos si el proceso recursivo devuelve el resultado correcto para un número mayor a 2, por ejemplo 8 ($n=8$). En este caso "f2" toma el valor del Fibonacci de 7 ($f(n-1)$), recordando que en el razonamiento recursivo no debemos preocuparnos de como se calcula dicho valor, entonces el resultado se calcula sumando el Fibonacci de 7 (f2) y el Fibonacci anterior a 7, que como ya se dijo se encuentra en la variable local estática "f1" y en este caso debe ser igual a 8 (una vez más no debe preocuparnos el como se calcula dicho valor) por lo tanto el resultado devuelto para " $n=8$ " es $8+13 = 21$, que es el resultado correcto, pues como se sabe el Fibonacci de 8 es 21. Finalmente, para que en cualquier llamada recursiva la variable local estática "f1" tenga el penúltimo valor calculado, se asigna el valor de "f2" a dicha variable (pues el último valor calculado se encuentra en la variable "result").

11.4. Ejemplos

11.4.1. Potencia entera de un número real

Como primer ejemplo elaboraremos una aplicación para calcular la potencia entera de un número real (x^n).

Desde el punto de vista recursivo el problema se resuelve multiplicando por "x" la potencia de x^{n-1} , es decir:

$$x^n = x^{n-1} * x$$

Y puesto que cualquier número elevado a uno ($n=1$) es igual al mismo número, esta será la condición de finalización.

No obstante, esta solución aunque matemáticamente correcta, no es eficiente. Por ejemplo para calcular $3.45^{2500000}$, se deben realizar 2 millones quinientas mil llamadas recursivas e igual número de multiplicaciones.

Una solución más eficiente se consigue tomando en cuenta que x^n puede ser calculado con $(x^{n/2})^2$ si n es par y $(x^{(n-1)/2})^2 * x$ si n es impar.

Así por ejemplo el valor de x^{4561} puede ser calculado con:

$$x^{4561} = (x^{(4561-1)/2})^2 * x = (x^{2280})^2 * x$$

Para calcular x^{2280} aplicamos el mismo procedimiento:

$$x^{2280} = (x^{2280/2})^2 = (x^{1140})^2$$

Prosiguiendo de esta manera los siguientes cálculos recursivos serían:

$$x^{1140} = (x^{1140/2})^2 = (x^{570})^2$$

$$x^{570} = (x^{570/2})^2 = (x^{285})^2$$

$$x^{285} = (x^{(285-1)/2})^2 * x = (x^{142})^2 * x$$

$$x^{142} = (x^{142/2})^2 = (x^{71})^2$$

$$x^{71} = (x^{71/2})^2 * x = (x^{35})^2 * x$$

$$x^{35} = (x^{(35-1)/2})^2 * x = (x^{17})^2 * x$$

$$x^{17} = (x^{(17-1)/2})^2 * x = (x^8)^2 * x$$

$$x^8 = (x^{8/2})^2 = (x^4)^2$$

$$x^4 = (x^{4/2})^2 = (x^2)^2$$

$$x^2 = (x^{2/2})^2 = (x^1)^2$$

$$x^1 = x$$

Con este procedimiento se consigue un considerable ahorro de tiempo y recursos, así en lugar de las 4561 llamadas recursivas (y 4561 multiplicaciones) sólo son necesarias 13 llamadas recursivas y 18 multiplicaciones.

Como se puede observar este planteamiento es por naturaleza recursivo, de manera que puede ser implementado con mayor facilidad aprovechando la recursividad.

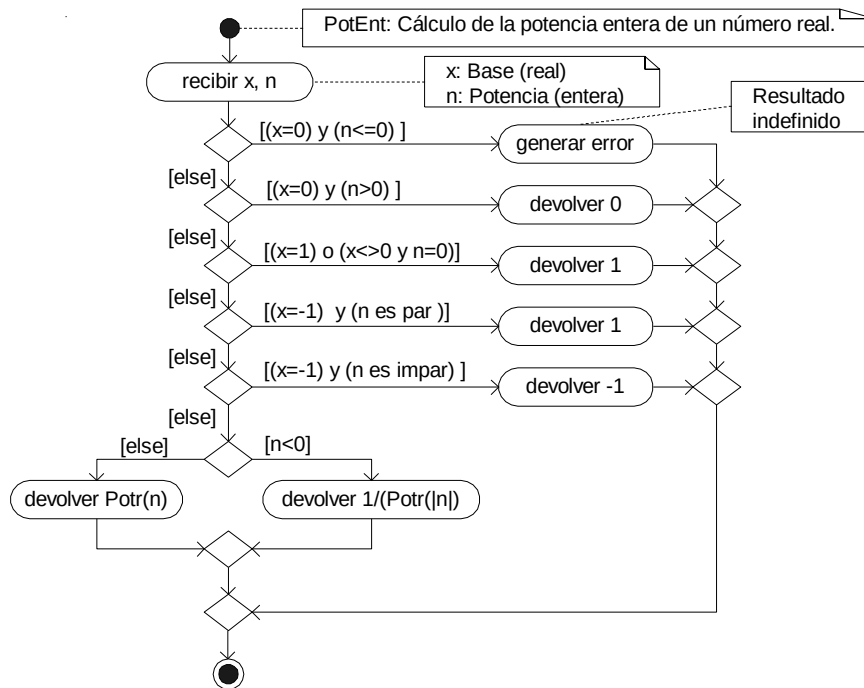
La potencia entera de un número real tiene muchos casos que deben ser analizados antes de proceder al cálculo, pues en ocasiones (como cuando la base es 1) se sabe el resultado sin necesidad de hacer ningún cálculo y en otros (como cuando la base es 0 y la potencia 0) el resultado es indefinido. Cuando se emplea la recursividad, todos estos casos deben ser analizados en un módulo separado, pues si se realizara el análisis en el módulo recursivo,

el mismo se repetiría de manera innecesaria cada vez que el módulo se llama a si mismo.

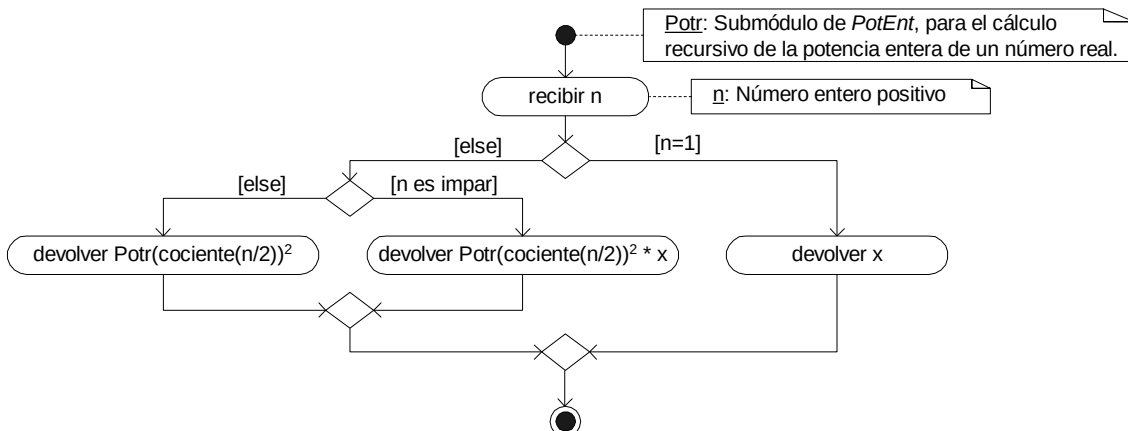
Por otra parte cuando, como en el presente caso, existen parámetros que en realidad son constantes (como el valor de "x" pues no cambia entre llamadas) no tiene sentido crear copias del mismo en cada llamada recursiva. En estos casos la mejor alternativa consiste en implementar el módulo recursivo dentro de otro módulo no recursivo. Entonces el módulo no recursivo recibe todos los parámetros que se requieren para resolver el problema mientras que el submódulo recursivo sólo aquellos que cambian entre llamadas.

Todos los lenguajes estructurados, incluido por supuesto *Pascal*, permiten crear submódulos, esta es justamente una característica fundamental de los lenguajes estructurados. Con un lenguaje no estructurado probablemente sea necesario recurrir a variables globales o variables locales estáticas.

El algoritmo del módulo principal, donde se reciben todos los parámetros, se analizan los posibles casos y se llama al módulo recursivo es el siguiente:



El algoritmo del submódulo recursivo es el siguiente:



Es en este submódulo es donde realmente se resuelve el problema y como se puede observar sólo recibe "n" (la potencia) pues es el único valor que cambia entre llamadas, además dicho valor es siempre positivo, porque, en el módulo principal, para calcular la potencia negativa se emplea la inversa de la potencia positiva: $x^{-n} = 1/x^n$.

Creemos entonces la aplicación (File -> New Application) y la unidad (File -> New Unit) y guardemos la aplicación (File -> Save Project As...) en el directorio "Recursividad\Potencia entera" con los siguientes nombres: "ufPotEnt" para "Unit1", "uPotEnt" para "Unit2" y "pPotEnt" para "Project1".

Y escribamos el código (incluyendo los módulos de lectura y escritura) en la unidad "uPotEnt":

```
unit uPotEnt;
```

```
interface
```

```
  uses SysUtils, StdCtrls, Math, QDialogs;
```

```
  function LeerBase(m: TMemo): extended;
```

```
  function LeerPotencia(m: TMemo): integer;
```

```
  function PotEnt(x: extended; n: integer): extended;
```

```
  function CalcularPotencia(x: extended; n: integer): extended;
```

```
  procedure MostrarResultado(m: TMemo; p: extended);
```

```
implementation
```

```
  function LeerBase(m: TMemo): extended;
```

```
  begin
```

```
    try
```

```
      result:= StrToFloat(m.Text);
```

```
    except
```

```
      on eConvertError do begin
```

```
        ShowMessage('La base está mal escrita');
```

```
        m.SetFocus; m.SelectAll; Abort; end;
```

```
    end;
```

```
  end;
```

```
  function LeerPotencia(m: TMemo): integer;
```

```
  begin
```

```
    try
```

```
      result:= StrToInt(m.Text);
```

```
    except
```

```
      on eConvertError do begin
```

```
        ShowMessage('La potencia está mal escrita');
```

```
        m.SetFocus; m.SelectAll; Abort; end;
```

```
    end;
```

```
  end;
```

```
  function PotEnt(x: extended; n: integer): extended;
```

```
  var c: byte;
```

```
    function Potr(n: cardinal): extended;
```

```
    begin
```

```
      if n=1 then result:= x
```

```
      else
```

```
        if odd(n) then
```

```
          result:= sqr(Potr(n div 2))*x
```

```
        else
```

```
          result:= sqr(Potr(n div 2));
```

```

    end;
begin
    c:= ord((x=0) and (n<=0))+
        ord((x=0) and (n>0))*2+
        ord((x=1) or ((x<>0) and (n=0)))*3+
        ord((x=-1) and (n mod 2 = 0))*4+
        ord((x=-1) and odd(n))*5;
    case c of
        1: raise EInvalidArgument.Create(
            'Resultado Indefinido: La base y la potencia son cero');
        2: result:= 0;
        3,4 : result:= 1;
        5: result:= -1;
        else
            if n<0 then result:= 1/Potr(abs(n))
            else result:= Potr(n);
        end;
    end;

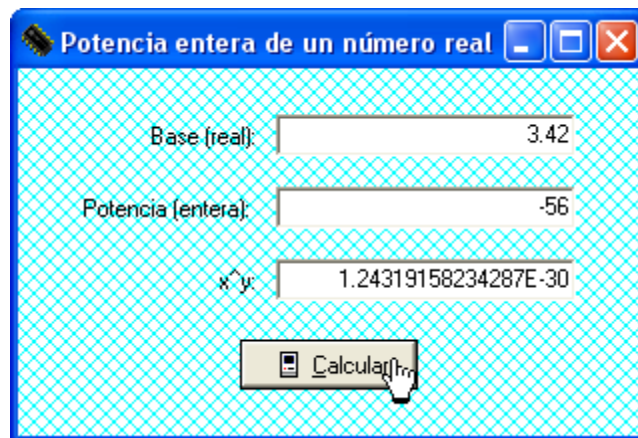
function CalcularPotencia(x: extended; n: integer): extended;
begin
    try
        result:= PotEnt(x,n);
    except
        on e: EInvalidArgument do begin
            ShowMessage(e.Message); Abort; end;
        end;
    end;

procedure MostrarResultado(m: TMemo; p: extended);
begin
    m.Text:= FloatToStr(p);
end;

end.

```

Ahora elaboremos la interfaz de la aplicación de manera que, en ejecución, se vea aproximadamente como se muestra en la figura:



En esta aplicación no existen componentes nuevos, siendo las propiedades modificadas de las siguientes:

Icono de la aplicación: technlgy.ico;

Form1: Name = 'fPotEnt'; Caption = 'Potencia entera de un número real'; Height = 217; Width = 320; Position = poScreenCenter.

Label1: Caption = 'Base (real)'; Label2: Caption = 'Potencia (entera)'; Label3: Caption = 'x^y'; Propiedades comunes: Transparent = True; Alignment = taRightJustify.

Memo1: Name = 'mBase'; Cursor = crIBeam; Memo2: Name = 'mPotencia'; Cursor = crIBeam; ReadOnly = True; Name = 'mResultado'; Cursor = crArrow; ReadOnly = True; TabStop = False; Propiedades comunes: Height = 21; Width = 150, WantReturns = False; Alignment = taRightJustify.

BitBtn1: Name = 'bbCalcular'; Caption = '&Calcular'; Glyph= 'comp-mac.bmp', Cursor = crHandPoint; Default = True.

Asegúrese también que el orden de los componentes sea (Edit -> Tab Order): mBase, mPotencia, mResultado y bbCalcular.

Cambiamos entonces el estilo de la brocha e inicializamos el texto de los memos en el evento "onCreate" de la forma:

```
procedure TfPotEnt.FormCreate(Sender: TObject);
begin
  fPotEnt.Brush.Color:= claqua;
  fPotEnt.Brush.Style:= bsDiagCross;
  fPotEnt.Position:= poScreenCenter;
  mBase.Text:= '1.';
  mPotencia.Text:= '1';
  mResultado.Text:= '1';
end;
```

Damos el foco y seleccionamos el texto del memo mBase en el evento "onActivate" de la forma:

```
procedure TfPotEnt.FormActivate(Sender: TObject);
begin
  mBase.SetFocus; mBase.SelectAll;
end;
```

Validamos la introducción de datos de los memos "mBase" y "mPotencia" en sus eventos "onKeyPress":

```
procedure TfPotEnt.mBaseKeyPress(Sender: TObject; var Key: Char);
begin
  case Key of
    #8, '0'..'9', '.', '+', '-', 'e', 'E':
      else Beep; Abort;
  end;
end;
```

```
procedure TfPotEnt.mPotenciaKeyPress(Sender: TObject; var Key: Char);
begin
  case Key of
    #8, '0'..'9', '-', '+': ;
    else Beep; Abort;
  end;
end;
```

Seleccionamos el texto del memo "mPotencia" al ingresar al mismo, programando su evento "onEnter":

```
procedure TfPotEnt.mPotenciaEnter(Sender: TObject);
begin
  TMemor(Sender).SelectAll;
```

end;

Finalmente damos funcionalidad a la aplicación programando el evento "on-Click" del BitBtn "bbCalcular":

```
procedure TfPotEnt.bbCalcularClick(Sender: TObject);
var n: integer; x,p: extended;
begin
    x:= LeerBase(mBase);
    n:= LeerPotencia(mPotencia);
    p:= CalcularPotencia(x,n);
    MostrarResultado(mResultado,p);
    mBase.SetFocus; mBase.SelectAll;
end;
```

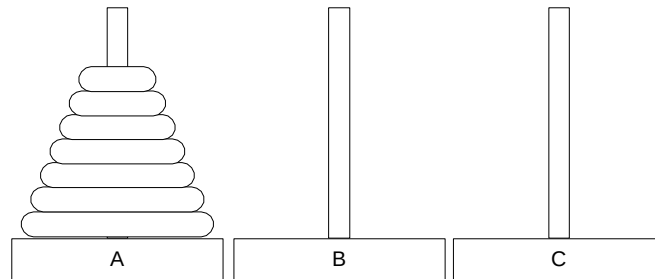
Entonces, al hacer correr la aplicación deberá tener la apariencia y calcular el resultado mostrado en la figura correspondiente a la interfaz de la aplicación. Además debe probar que la aplicación responda correctamente en todos los casos posibles, generando los errores adecuados cuando corresponda generarlos.

11.4.2. Las torres de Hanoi

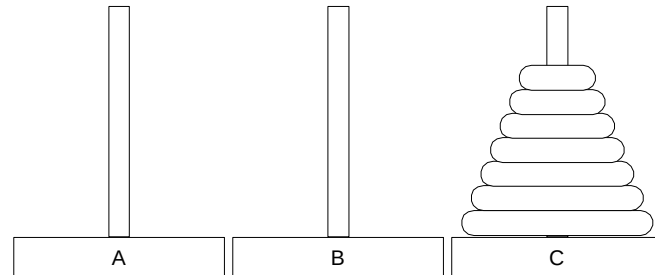
Como segundo ejemplo elaboraremos una aplicación que devuelva los movimientos que se deben realizar para resolver el juego de las torres de Hanoi.

Las torres de Hanoi es un juego que consta de tres postes y un número determinado aros de diferente diámetro.

Al inicio del juego los aros se encuentran ordenados en el primer poste "A", tal como se muestra en la figura:

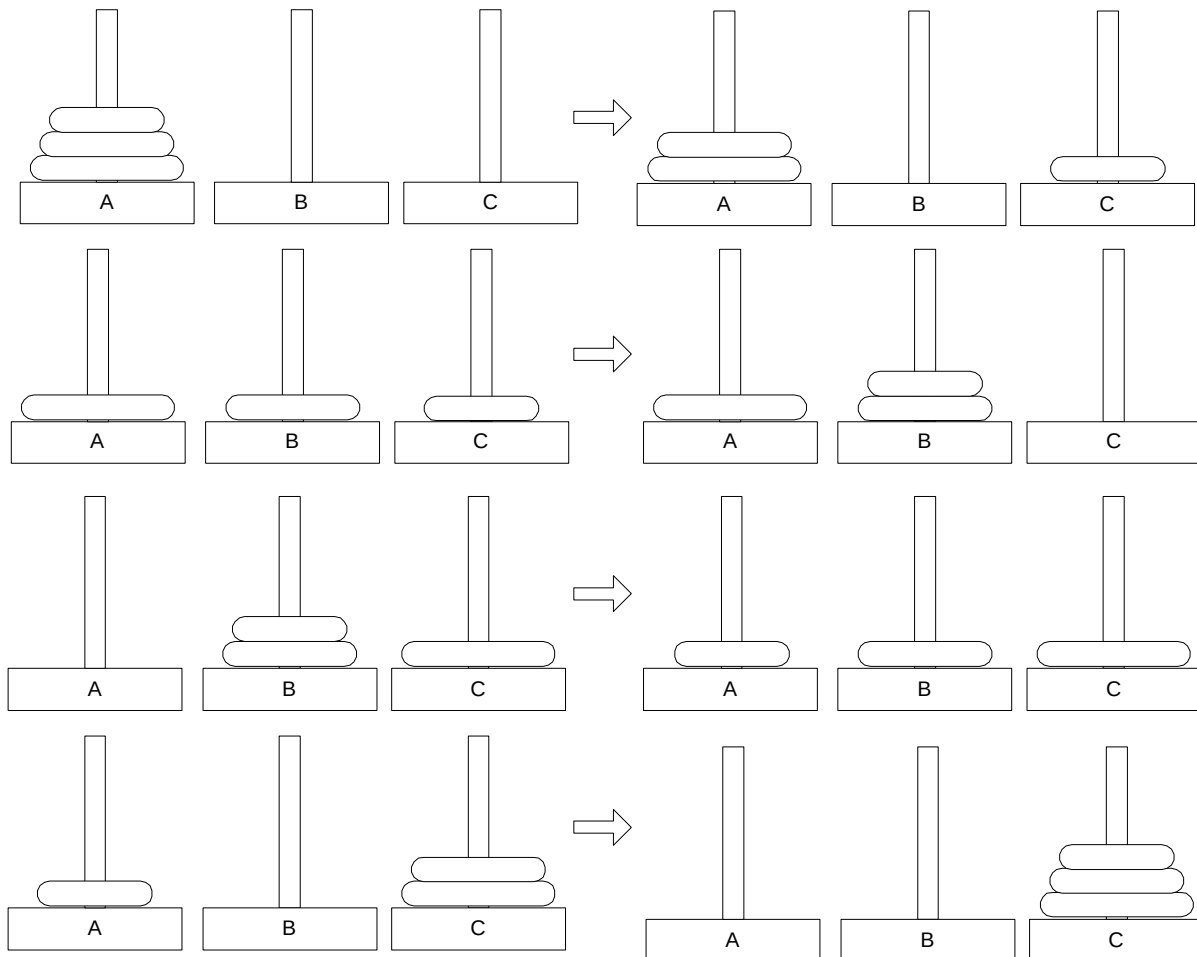


El objetivo del juego es hacer que al final estos aros queden ordenados en el tercer poste, tal como se muestra en la figura:



Para ello se debe mover un aro a la vez, empleando también el poste auxiliar "B" y teniendo en cuenta que no está permitido colocar un aro de mayor diámetro sobre otro de menor diámetro.

Por ejemplo si el juego consta tres aros, se deben realizar los movimientos que se detallan en las figuras siguientes, para alcanzar el objetivo del juego:



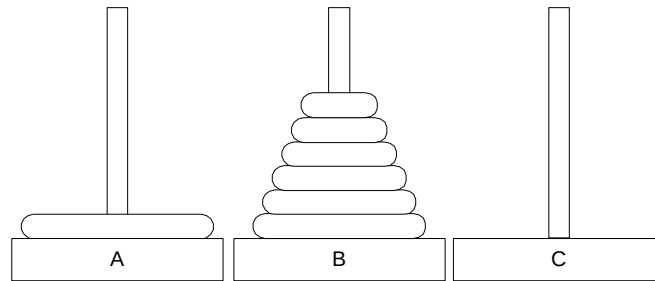
Por lo tanto los movimientos necesarios para ordenar los aros en el poste "C" son: "A->C", "A->B", "C->B", "A->C", "B->A", "B->C" y "A->C", que se entiende como: mover el último aro del poste "A" al poste "C", luego mover el último aro del poste "A" al poste "B", después mover el último aro del poste "C" al poste "B", mover el último aro del poste "A" al poste "C", mover el último aro del poste "B" al poste "A", mover el último aro del poste "B" al poste "C" y finalmente mover el último aro del poste "A" al poste "C".

Este es un problema por naturaleza recursivo, por lo tanto la solución es mucho más sencilla siguiendo un razonamiento recursivo, para ello recordemos que debemos contar con un proceso recursivo y una condición de finalización.

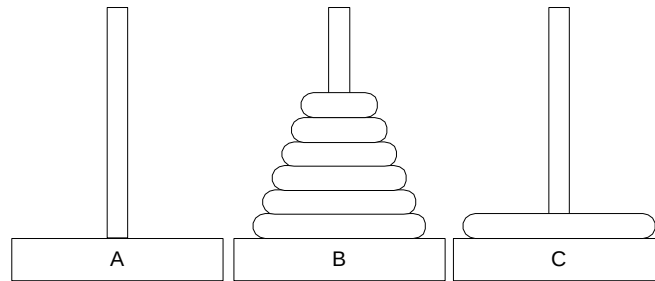
Cuando en un poste queda un solo aro, entonces la única opción posible es la de mover dicho aro a cualquiera de los otros postes, con lo que el proceso para dicho poste concluye (pues no existen más aros que mover). Por lo tanto esta será nuestra condición de finalización: cuando el número de aros en un poste es "1", se mueve dicho aro al poste respectivo y el proceso concluye, una vez más, con el razonamiento recursivo no debemos preocuparnos de la situación de los otros postes (si tienen o no aros), sólo debe importarnos que el razonamiento para un poste que queda con un aro sea correcto y en este caso es correcto, pues al hacer el último movimiento el poste queda sin aros y al no quedar más aros que mover el proceso para el mismo debe concluir.

Una vez que contamos con la condición de finalización pensamos en el proceso recursivo que se debe seguir para llevar "n" aros desde un poste dado

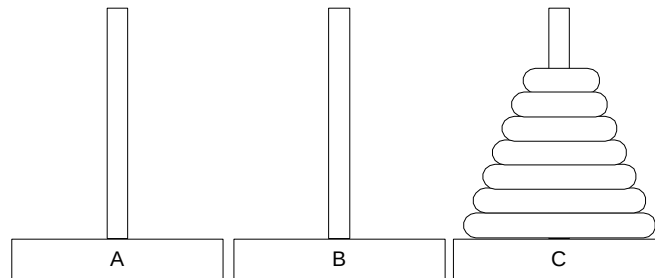
hasta otro y dicho proceso es bastante simple: para llevar "n" aros desde un poste de origen a otro de destino, debemos mover "n-1" aros del poste de origen al poste auxiliar (aquel que no es ni el origen ni el destino):



Luego movemos el último aro del poste de origen al poste de destino:

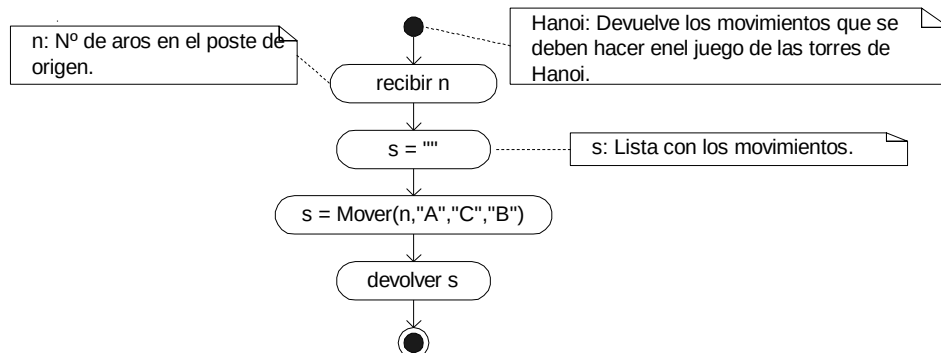


Finalmente movemos los "n-1" aros del poste auxiliar al poste de destino:

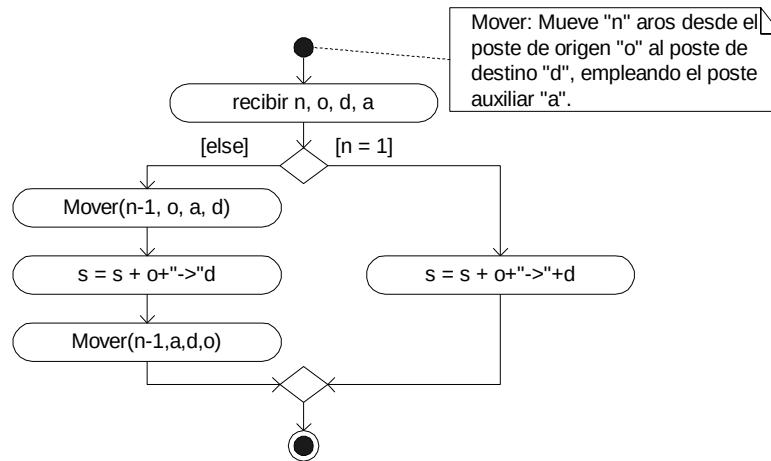


Una vez más, al seguir el razonamiento recursivo no debemos preocuparnos de como se mueven los "n-1" aros de un poste a otro, pues de ello se encargan las llamadas recursivas, sólo debemos verificar que se muevan los "n" aros del poste de origen al poste de destino.

Como en el anterior ejemplo, el algoritmo se ha dividido en dos: el módulo principal, donde se llama a módulo recursivo y se devuelve el resultado:



Y el módulo recursivo (Mover), que es donde realmente se resuelve el problema y se sigue la lógica antes explicada:



Creemos una nueva aplicación (File -> New Application) y una unidad (File -> New Unit) y guardemos la aplicación (File -> Save Project As...) en el directorio "Recursividad\Hanoi" con los siguientes nombres: "ufHanoi" para "Unit1", "uHanoi" para "Unit2" y "pHanoi" para "Project1".

Escribimos entonces el código (incluyendo los módulos de lectura y escritura) en la unidad "uHanoi":

```
unit uHanoi;
```

```
interface
```

```
Uses Classes, StdCtrls, SysUtils, QDialogs;
```

```
function LeerNumAros(cb: TComboBox): Word;
```

```
function Hanoi(n: word): tStringList;
```

```
function Tiempo: double;
```

```
procedure MostrarMovimientos(m: TMemo; s: TStringList);
```

```
procedure MostrarNumMovidas(m: TMemo; s: TStringList);
```

```
procedure MostrarTiempo(m: TMemo; t: double);
```

```
implementation
```

```
function LeerNumAros(cb: TComboBox): Word;
```

```
begin
```

```
try
```

```
result:= StrToInt(cb.Text);
```

```
except
```

```
on eConvertError do begin
```

```
ShowMessage('El número de aros está mal escrito');
```

```
cb.SetFocus; cb.SelectAll; Abort; end;
```

```
end;
```

```
end;
```

```
function Tiempo: double;
```

```
var h,m,s,ms: word;
```

```
begin
```

```
DecodeTime(Time,h,m,s,ms);
```

```
result:= h*3600+m*60+s+ms/1000;
```

```
end;
```

```
function Hanoi(n: word): tStringList;
```

```
var s: tStringList;
```

```

procedure Mover(n: word; o,d,a: Char);
begin
  if n=1 then
    s.Add(o+'->'+d)
  else begin
    Mover(n-1,o,a,d);
    s.Add(o+'->'+d);
    Mover(n-1,a,d,o);
  end;
end;
begin
  s:= TStringList.Create;
  Mover(n, 'A', 'C', 'B');
  result:= s;
end;

procedure MostrarMovimientos(m: TMemor; s: TStringList);
begin
  m.Lines:= s;
end;

procedure MostrarTiempo(m: TMemor; t: double);
begin
  m.Text:= FloatToStr(t);
end;

procedure MostrarNumMovidas(m: TMemor; s: TStringList);
begin
  m.Text:= IntToStr(s.Count);
end;

end.

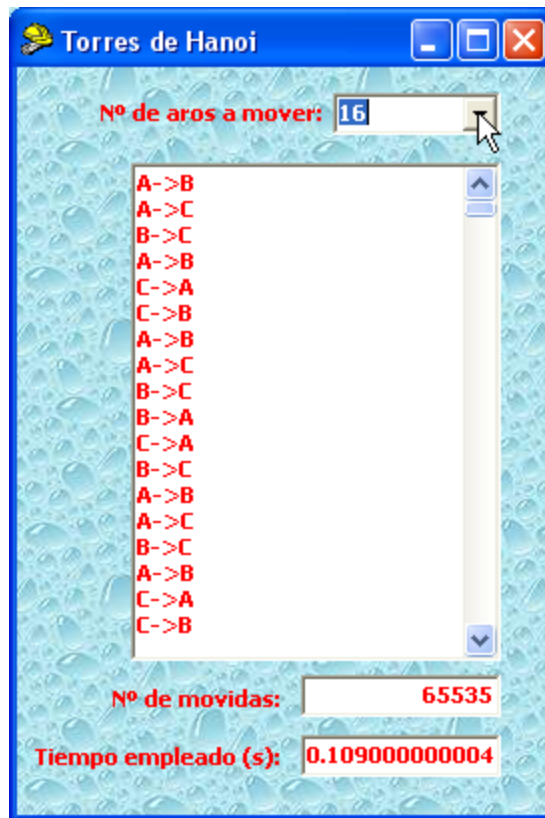
```

En este código hay algunas cosas nuevas: en primer lugar la función "Tiempo", que devuelve la hora de la computadora en segundos, lee el tiempo de la computadora con la función "Time", la misma que nos devuelve la hora de la computadora pero en el formato "DateTime", por ello se emplea el procedimiento "DecodeTime", para extraer las horas (h), los minutos (m), los segundos (s) y los milisegundos (ms) de dicho formato y se multiplican o dividen estos valores por los factores respectivos para convertirlos en segundos.

En segundo lugar se crea un objeto de tipo "TStringList", que como ya vimos en capítulos anteriores, permite manipular una lista de cadenas (strings). Lo nuevo en este código es que se está creando dicho objeto y como se puede ver se emplea un procedimiento similar al seguido para crear objetos de tipo "TBitmap". Como recordará, es posible acceder a cada una de las cadenas por su índice (como ocurre con un array), siendo el índice de la primera cadena "0". Recordará también que las propiedades de algunos objetos son de este tipo, así por ejemplo son objetos de este tipo la propiedad "Lines" de un memo y la propiedad "Items" de un ComboBox.

En este programa se emplea el objeto de tipo "TStringList" para guardar los movimientos que se deben hacer para resolver el problema y cada uno de dichos movimientos es añadido al objeto de tipo "TStringList" con el método "Add" (aunque también se puede emplear el método Append).

Ahora elaboremos la interfaz de la aplicación de manera que, en ejecución, se vea aproximadamente como se muestra en la figura de la siguiente página.



En esta aplicación no existen componentes nuevos, siendo las propiedades modificadas las siguientes:

Icono de la aplicación: construct.ico;

Form1: Name = 'fHanoi'; Caption = 'Torres de Hanoi'; Height = 408; Width = 274; Position = poScreenCenter.

Label1: Caption = 'Nº de aros a mover:'; Label2: Caption = 'Nº de movidas'; Label3: Caption = 'Tiempo empleado (s)'; Propiedades comunes: Transparent = True; Alignment = taRightJustify; Font.Color = clRed; Font.Style = [fsBold].

Memo1: Name = 'mMovimientos'; WantReturns = True; Alignment = taLeftJustify; Height = 249; width = 185; Memo2: Name = 'mNumMovidas'; WantReturns = False; Height = 21; Width = 100; Alignment = taRightJustify; Memo3: Name = 'mTiempo'; WantReturns = False; Height = 21; Width = 100; Alignment = taRightJustify; Propiedades comunes: ReadOnly = True; TabStop = False; Cursor = crArrow; Font.Color = clRed; Font.Style = [fsBold];

ComboBox1: Name = 'cbNumAros'; Style = csDropDownList; Font.Color = clRed; Font.Style = [fsBold].

Asegúrese también que el orden de los componentes sea (Edit -> Tab Order): cbNumAros, mMovimientos, mNumMovidas y mTiempo.

Cambiamos entonces el estilo de la brocha e inicializamos el ComboBox, así como el texto de los memos en el evento "onCreate" de la forma:

```
procedure TfHanoi.FormCreate(Sender: TObject);
var i: Integer;
begin
  fHanoi.Brush.Bitmap:= TBitmap.Create;
  fHanoi.Brush.Bitmap.LoadFromFile('Gotas de agua.bmp');
```

```

cbNumAros.Items.Clear;
for i := 1 to 100 do cbNumAros.Items.Add(IntToStr(i));
cbNumAros.ItemIndex:= 0;
mMovimientos.Lines.Clear; mNumMovidas.Text:= '0'; mTiempo.Text:= '0';
end;

```

Como de costumbre, liberamos el Bitmap el evento "onClose" de la forma:

```

procedure TfHanoi.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  fHanoi.Brush.Bitmap.Free;
end;

```

Finalmente damos funcionalidad a la aplicación en el evento "onChange" del ComboBox "cbNumAros":

```

procedure TfHanoi.cbNumArosChange(Sender: TObject);
var sl: TStringlist; t: double; n: word;
begin
  n:= LeerNumAros(cbNumAros);
  t:= Tiempo;
  sl:= Hanoi(n);
  t:= Tiempo-t;
  MostrarMovimientos(mMovimientos,sl);
  MostrarNumMovidas(mNumMovidas,sl);
  MostrarTiempo(mTiempo,t); sl.Free;
end;

```

Como se puede observar el objeto creado en el módulo "Hanoi" es liberado en este evento después de haber sido utilizado en los módulos "MostrarMovimientos" y "MostrarNumMovidas". En esta aplicación no se valida la introducción de datos porque la única forma de introducirlos es mediante el ComboBox "cbNumAros" y el estilo del mismo es "csDropDownList", que como sabemos no permite introducir datos, sino sólo seleccionarlos.

Una vez corregidos los errores la aplicación deberá verse y presentar los valores mostrados en la figura correspondiente a la interfaz de la aplicación. Al hacer correr el programa para un número de aros igual a 16, como se muestra en la figura, o números mayores a este, no se obtiene el resultado inmediatamente, a pesar de que como se muestra en el memo "mTiempo", el tiempo necesario apenas sobrepasa la décima de segundo. Esto se debe a que Delphi debe reservar memoria y llenar el StringGrid del memo, lo que consume mucho más memoria y tiempo que el consumido por el módulo que resuelve el problema (Hanoi). No obstante, como se puede ver, el número de movimientos necesarios para ordenar 16 aros es elevado: 65535, con este número de movimientos, si una persona mueve un aro cada 2 segundos necesitaría 131070 segundos, para terminar el juego, es decir 36.4 horas continuas.

11.4.3. Exponente de un número real

Como tercer ejemplo elaboraremos una aplicación para calcular el exponente de un número real, e^x .

Para resolver el problema de manera recursiva adoptaremos un enfoque diferente al de la forma iterativa. Si el número "x" es menor o igual a 0.5, calculamos su valor con la serie finita:

$$e^x = \sum_{i=0}^{16} \frac{x^i}{i!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots + \frac{x^{16}}{16!} \quad \{x \leq 0.5\}$$

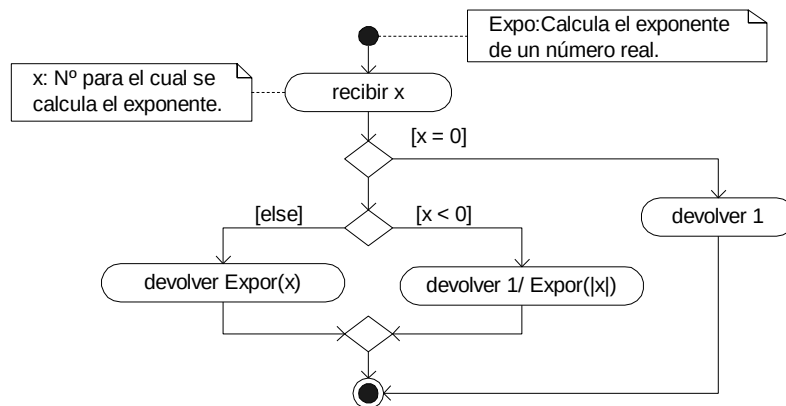
Pues se sabe que en estos casos es suficiente sumar los primeros 16 términos para calcular el resultado. Para números mayores a 0.5 calcularemos el

resultado de manera similar al de la potencia entera, es decir con el cuadrado del exponente de la mitad del número:

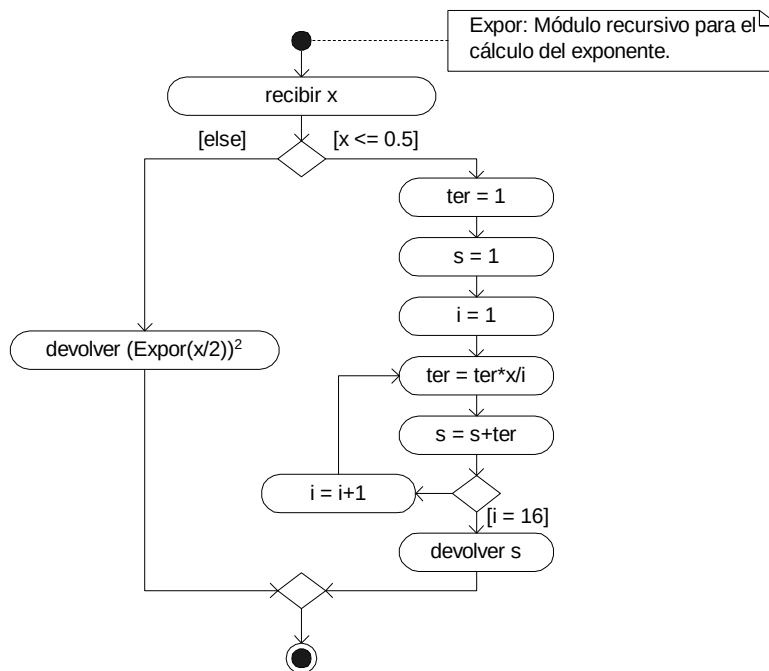
$$e^x = (e^{x/2})^2$$

Entonces la condición de finalización será que "x" sea menor o igual a "0.5", en cuyo caso la respuesta se calcula con la serie finita y el proceso iterativo simplemente consiste en llamadas recursivas con la mitad del número cuyo exponente se quiere calcular. En el proceso recursivo sólo se considerarán exponentes positivos, porque los negativos serán calculados en base a los negativos con la relación: $e^{-x} = 1/e^x$.

Como casi siempre sucede con este tipo de problemas es conveniente crear un módulo principal, donde se analizan los casos en los cuales no se debe realizar el cálculo, como cuando "x" es cero, así como el cálculo del exponente negativo y desde el cual se llama al módulo recursivo, que es donde realmente se resuelve el problema. El algoritmo del módulo principal es el siguiente:



El algoritmo del submódulo recursivo es:



Para codificar estos algoritmos creemos una nueva aplicación (File -> New Application), una nueva unidad (File -> New Unit) y guardemos la aplicación

(File -> Save Project As...) en el directorio "Recursividad\Exponente" con los siguientes nombres: "ufExponente" para "Unit1", "uExponente" para "Unit2" y "pExponente" para "Project1".

Escribimos entonces el código (incluyendo los módulos de lectura y escritura) en la unidad "uExponente":

```
unit uExponente;

interface

  uses StdCtrls, SysUtils, QDialogs;

  function LeerNumero(m: TMemo): extended;
  function Expo(x: extended): extended;
  procedure MostrarExponente(m: TMemo; e: extended);

implementation

  function LeerNumero(m: TMemo): extended;
  begin
    try
      result := StrToFloat(m.Text);
    except
      on EConvertError do begin
        ShowMessage('El número real ha sido mal escrito');
        m.SetFocus; m.SelectAll; Abort; end;
      end;
    end;

  function Expo(x: extended): extended;
  var ter, s: extended;
  function Expor(x: extended): extended;
  var i: byte;
  begin
    if x <= 0.5 then begin
      ter := 1; s := ter; i := 0;
      for i := 1 to 16 do begin ter := ter * x / i; s := s + ter; end;
      result := s; end
    else
      result := Sqr(Expor(x/2));
    end;
  begin
    if x = 0 then begin result := 1; exit; end;
    if x < 0 then result := 1/Expor(Abs(x)) else result := Expor(x);
  end;

  procedure MostrarExponente(m: TMemo; e: extended);
  begin
    m.Text := Format('%18.12f', [e]);
  end;

end.
```

Observe que en el módulo "Expo", se declara la variable "i" dentro del módulo recursivo "Expor". Por lo tanto cada vez que el módulo se llama a si mismo se crea una copia de dicha variable, aunque sólo se la utiliza en la última llamada. Se ha procedido así porque "Delphi" no permite declarar la variable de control de la estructura "For" fuera del módulo, una solución más eficiente consistiría en declarar dicha variable en el módulo "Expo" y

emplear la estructura "Until" (o "While") en lugar de la estructura "For", sin embargo, con ello se pierde claridad en el código y no se emplea la estructura adecuada, porque como sabemos, cuando se conoce el número de iteraciones se debe emplear la estructura "For".

En el módulo "MostrarExponente" se ha empleado una nueva función: "format". Esta función permite dar formato a diferentes tipos de datos, devolviendo el resultado en forma de texto, tiene la siguiente sintaxis:

```
format('formato', [datos a formatear])
```

Donde 'formato' es un texto que puede contar con los siguientes elementos:

```
formato = "%" [índice ":" ] ["-"] [ancho] [". " dígitos ] tipo
```

Los elementos que han sido escritos entre corchetes son *opcionales*, mientras que los elementos que se encuentran entre comillas son *obligatorios*, por consiguiente un *formato* debe constar por lo menos de un "%" y del tipo de dato. Si se incluye otro de los elementos, entonces se debe escribir *obligatoriamente* los caracteres que se encuentran entre comillas, así por ejemplo el siguiente texto:

```
'% 20.9f'
```

Da formato a un número real (*f*), donde la cadena resultante tendrá un ancho de 20 caracteres y el número será mostrado con 9 dígitos después del punto (redondeados).

El *índice* es un número entero que sirve para especificar el número de dato al cual se aplicará el *formato*. El signo menos (-) sirve para que el texto formateado sea alineado a la izquierda en lugar de la derecha como sucede por defecto.

Los tipos de datos que pueden ser empleados en el *formato* son los siguientes:

d: Dígito. El dato a formatear es de tipo entero. Si se especifica el número de dígitos después del punto y el dato tiene un menor número de dígitos, es completado con ceros a la izquierda.

u: Entero sin signo. Igual que el anterior pero el dato es positivo.

e: Notación científica. El dato es real y es formateado empleando notación científica. El número total de dígitos a emplear en el formato es determinado por el número de dígitos después del punto.

f: Fijo. El dato es real y el número es redondeado al número de dígitos especificado después del punto.

g: General. El dato es real y es formateado a la cadena más corta posible. El número total de dígitos en la cadena formateada es igual al número de dígitos especificado después del punto.

n: Número. El dato es real y es formateado con separador de miles.

m: Money (monetario). El dato es real y es formateado de acuerdo al formato especificado en el sistema operativo para datos de tipo monetario.

p: Puntero. El dato es un puntero y es devuelto como una cadena hexadecimal.

s: String (cadena). El dato es una cadena, si se especifica el número de dígitos después del punto y la cadena es más larga, entonces la cadena es truncada en dicho número de caracteres.

x: Hexadecimal. El dato es un entero y es devuelto como una cadena *hexadecimal*. Si se ha especificado el número de dígitos después del punto y el hexadecimal resultante tiene un menor número de dígitos es completado con ceros por delante.

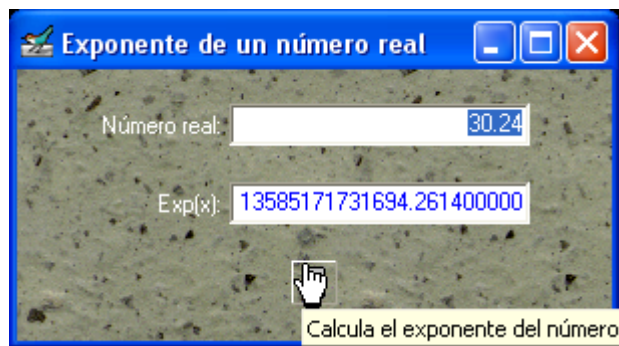
Por ejemplo las siguientes instrucciones devuelven las cadenas que se escriben a su derecha:

```
format('%12.d',[1432]) => '000000001432'
format('%14.6f',[42.32341432]) => '      42.323214'
format('%20.14s',['Facultad de Tecnología']) => '          Facultad de Te'
format('%12d %12.6e %0:12.8d',[ 5273 142.32442324 ]) => '          5273
1.42324E+002      00005273'

format('%12.5m %0:12.5f %0:12.5n %0:12.5g',[ 5142.723424824 ]) => ' $b
5,142.72342      5142.72342  5,142.72342      5142.7'

format('%12.6m %0:12.6f %0:12.6n %0:12.3g',[ 5142.723424824 ]) => ' $b
5,142.723425      5142.723425 5,142.723425      5.14E3'
```

Ahora elaboremos la interfaz de la aplicación de manera que, en ejecución, se vea aproximadamente como se muestra en la siguiente figura:



En esta aplicación existe un nuevo componente: un "SpeedButton": , el cual se encuentra en la página "Additional" y que se diferencia del "BitBtn", porque está pensado para mostrar un botón con una imagen, pero sin el título (aunque el título también puede ser añadido). La imagen de este componente se asigna a través de su propiedad "Glyph", igual que en el BitBtn. Este tipo de botones se emplea principalmente para crear barras de herramientas en paneles, como las que tiene Delphi.

Puesto que normalmente este botón no muestra un título es conveniente escribir alguna guía en la propiedad "Hint" del mismo y hacer que la misma sea visible cambiando la propiedad "ShowHint" a "True", entonces cuando el cursor está sobre el botón aparece el texto escrito en "Hint", tal como se muestran en la anterior figura. En esta aplicación se modifican también la propiedad "Flat" que si está en "True" muestra el botón como una figura plana y la propiedad "Transparent" que cuando está en "True" (al igual que en "TImage") hace transparente el fondo de la figura.

Los otros componentes de la forma son conocidos y las propiedades que se han modificado en los mismos son:

Icono de la aplicación: Bird.ico

Form1: Name = 'fExponente'; Caption = 'Exponente de un número real'; Height = 169; Width = 305; Position = poScreenCenter.

Label1: Caption = 'Número real:'; Label2: Caption = 'Exp(x)'; Propiedades comunes: Transparent = True; Alignment = taRightJustify; Font.Color = clWhite.

Memo1: Name = 'mNumero'; Cursor = crIBeam; Memo2: Name = 'mExponente' ReadOnly = True; TabStop = False; Cursor = crArrow; Propiedades comunes: Font.Color = clBlue; WantReturns = False; Height = 21; Width = 150; Alignment = taRightJustify;

SpeedButton1: Name = 'sbCalcular'; Glyph=calculat.bmp; Hint='Calcula el exponente del número real'; ShowHint=True; Cursor=crHandPoint; Flat=True; Transparent=True.

Asegúrese también que el orden de los componentes sea (Edit -> Tab Order): mNumero, mExponente y sbCalcular.

Ahora asignamos la imagen del fondo a la brocha e inicializamos el texto de los memos en el evento "onCreate":

```
procedure TfExponente.FormCreate(Sender: TObject);
begin
    fExponente.Brush.Bitmap:= TBitmap.Create;
    fExponente.Brush.Bitmap.LoadFromFile('Roca verde.bmp');
    mNumero.Text:= '0'; mExponente.Text:= '1';
end;
```

Como de costumbre liberamos la imagen en el evento "onClose":

```
procedure TfExponente.FormClose(Sender: TObject; var Action: TCloseAction);
begin
    fExponente.Brush.Bitmap.Free;
end;
```

Asignamos el foco al memo "mNumero" y seleccionamos el texto del mismo al inicializar la aplicación, programando el evento "onActivate":

```
procedure TfExponente.FormActivate(Sender: TObject);
begin
    mNumero.SetFocus; mNumero.SelectAll;
end;
```

Validamos la introducción de datos programando el evento "onKeyPress" del memo "mNumero":

```
procedure TfExponente.mNumeroKeyPress(Sender: TObject; var Key: Char);
begin
    case Key of
        #8, '0'..'9', '.', '-', 'e', 'E': ;
        #13: begin sbCalcular.Click; abort; end;
        else Beep; Abort;
    end;
end;
```

Observe que en este caso se toma en cuenta la pulsación de la tecla "Enter" (#13), de manera que cuando se pulse esta tecla se simula el evento "onClick" del SpeedButton mediante su método "Click". Se ha procedido así porque el SpeedButton no tiene la propiedad "Default" y con este código se consigue un efecto similar al de dicha propiedad.

Finalmente damos funcionalidad a la aplicación programando el evento "onClick" del SpeedButton: "sbCalcular":

```
procedure TfExponente.sbCalcularClick(Sender: TObject);
var x,e: extended;
begin
```

```

x:= LeerNumero(mNumero);
e:= Expo(x);
MostrarExponente(mExponente,e);
mNumero.SetFocus; mNumero.SelectAll;
end;

```

Ahora al hacer correr la aplicación deberá tener la apariencia y mostrar el resultado mostrado en la figura correspondiente a la interfaz de la aplicación.

11.4.4. Inversión de los dígitos de un número entero

Como cuarto ejemplo elaboraremos una aplicación para invertir los dígitos de un número entero, positivo o negativo, es decir si el número es "123456789", la aplicación deberá devolver "987654321".

Como de costumbre, para el razonamiento recursivo debemos establecer una condición de finalización y un proceso recursivo. La condición de finalización será que el número a invertir sea igual a cero, en cuyo caso la inversa es simplemente "0".

El proceso recursivo se basa en el siguiente razonamiento: Podemos invertir un número de "n" dígitos si conocemos la inversa de los primeros "n-1" dígitos y a dicha inversa le sumamos el último dígito multiplicado por 10 elevado a "n-1". Así por ejemplo podemos invertir el número "12345" si conocemos la inversa de "1234", es decir "4321" y a este número le sumamos el último dígito "5" multiplicado por 10^4 , es decir:

$$\text{Invertir}(12345) = \text{Invertir}(1234) + 5 \cdot 10^4 = 4321 + 50000 = 54321$$

Ahora bien "1234" es el cociente de la división de "12345" entre 10 y "5" es el residuo de la misma división, por lo tanto la fórmula, resultante del razonamiento recursivo, incluida la condición de finalización, es:

$$\begin{aligned} \text{Invertir}(n) &= \text{Invertir}\left(\text{cociente}\left(\frac{n}{10}\right)\right) + \text{residuo}\left(\frac{n}{10}\right) * 10^{\text{N}^\circ \text{ de dígitos de } n-1} \\ \text{Invertir}(0) &= 0 \end{aligned}$$

Donde el problema radica en el cálculo de 10 elevado al número de dígitos del número menos 1, pues la mayoría de los lenguajes, incluido Pascal, no cuentan con funciones predefinidas para este fin. En consecuencia antes de aplicar la anterior fórmula recursiva debemos desarrollar un algoritmo que nos permita calcular dicho valor.

El algoritmo para este cálculo también es recursivo y se basa en el siguiente razonamiento: Podemos calcular el valor de 10 elevado al número de dígitos de un número menos 1, si conocemos el valor de 10 elevado al número de dígitos del número menos 2, pues es simplemente ese valor multiplicado por 10. Así por ejemplo podemos calcular el valor de 10 elevado al número de dígitos de "462937" menos 1, si conocemos el valor de 10 elevado al número de dígitos de "46293" menos 1, es decir:

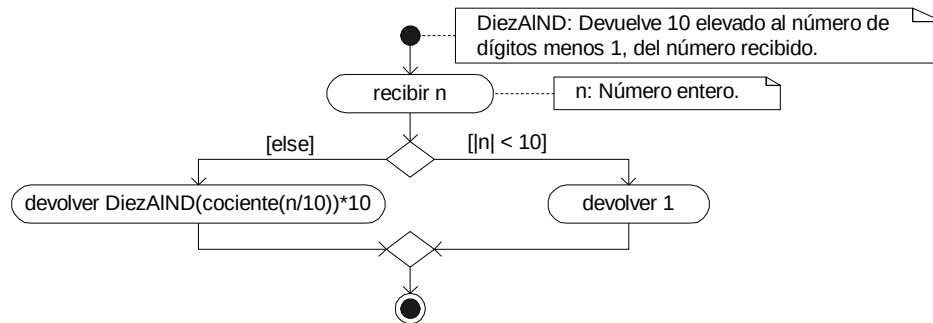
$$\text{DiezAlND}(462937) = \text{DiezAlND}(46293) * 10 = 10^4 * 10 = 10^5$$

Igual que antes "46293" es el cociente de "462937" entre 10. Para la condición de finalización tomaremos en cuenta que cuando el número solo tiene un dígito, es decir cuando su valor absoluto es menor a 10, el valor de $10^{\text{N}^\circ \text{ de dígitos del número}-1}$ es $10^{1-1} = 10^0 = 1$, entonces la fórmula recursiva es:

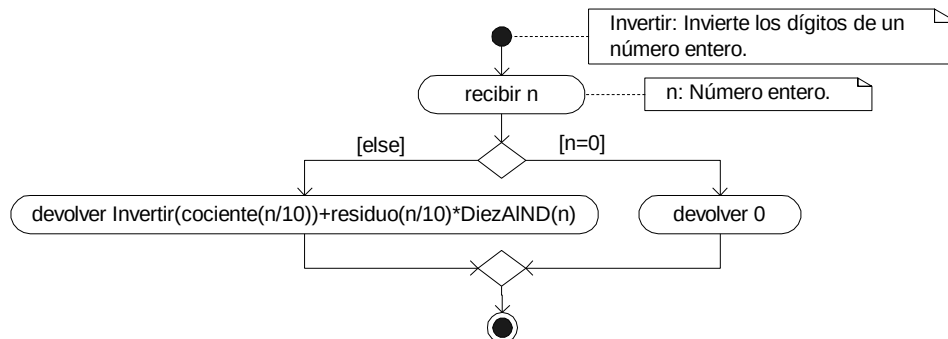
$$\text{DiezAlND}(n) = \text{DiezAlND}\left(\text{cociente}\left(\frac{n}{10}\right)\right) * 10$$

$$\text{DiezAlND}(n) = 1 \quad \{\text{si } |n| < 10\}$$

Y el algoritmo respectivo es:



Ahora podemos elaborar el algoritmo para invertir los dígitos:



Para codificar estos algoritmos creamos una nueva aplicación (File -> New Application), una nueva unidad (File -> New Unit) y guardamos la aplicación (File -> Save Project As...) en el directorio "Recursividad\Invetir" con los siguientes nombres: "ufInvertir" para "Unit1", "uInvertir" para "Unit2" y "pInvertir" para "Project1".

Escribimos entonces el código (incluyendo los módulos de lectura y escritura) en la unidad "uInvertir":

```
unit uInvertir;
```

```
interface
```

```
  uses StdCtrls, SysUtils, QDialogs, Math;
```

```
  function LeerNumero(m: TMemor): Int64;
```

```
  function DiezAlND(n: Int64): Int64;
```

```
  function Invertir(n: Int64): Int64;
```

```
  procedure MostrarNumero(m: TMemor; ni: Int64);
```

```
implementation
```

```
  function LeerNumero(m: TMemor): Int64;
```

```
  begin
```

```
    try
```

```
      result := StrToInt(m.Text);
```

```
    except
```

```
      on EConvertError do begin
```

```
        ShowMessage('El número entero está mal escrito');
```

```
        m.SetFocus; m.SelectAll; end;
```

```
  end;
```

```

end;

function DiezAlND(n: Int64): Int64;
begin
  if Abs(n)<10 then result:= 1 else result:= DiezAlND(n div 10)*10;
end;

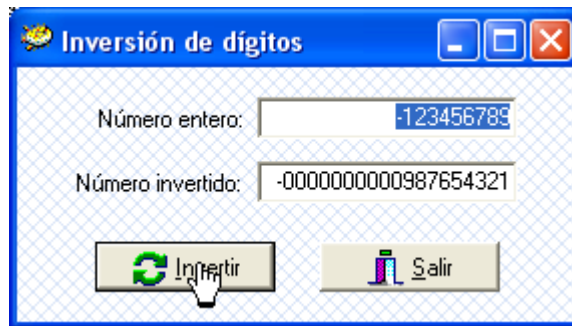
function Invertir(n: Int64): Int64;
begin
  if n=0 then result:=0 else
    result:=Invertir(n div 10)+(n mod 10)*DiezAlND(n);
end;

procedure MostrarNumero(m: TMemo; ni: Int64);
begin
  m.Text:= Format('%20.19d',[ni]);
end;

end.

```

Ahora elaboramos la interfaz de la aplicación de manera que, en ejecución, se vea aproximadamente como se muestra en la siguiente figura:



Todos los componentes de esta aplicación son conocidos, sólo se ha añadido un botón adicional para dar otra opción para salir de la aplicación. Las propiedades que se han modificado son las siguientes:

Icono de la aplicación: Fish.ico

Form1: Name = 'fInvertir'; Caption = 'Inversión de dígitos'; Height = 161; Width = 287; Position = poScreenCenter.

Label1: Caption = 'Número entero:'; Label2: Caption = 'Número invertido:'; Propiedades comunes: Transparent = True; Alignment = taRightJustify.

Memo1: Name = 'mNumeroEntero'; Cursor = crIBeam; Memo2: Name = 'mNumeroInvertido'; ReadOnly = True; TabStop = False; Cursor = crArrow; Propiedades comunes: WantReturns = False; Height = 21; Width = 130; Alignment = taRightJustify;

BitBtn1: Name = 'bbInvertir'; Kind = bkRetry; Caption = '&Invertir'; Default = True; BitBtn2: Name = 'bbSalir'; Kind = bkClose; Caption = '&Salir'; Propiedades comunes: Cursor=crHandPoint.

Asegúrese también que el orden de los componentes sea (Edit -> Tab Order): mNumeroEntero, mNumeroReal bbInvertir y bbSalir.

Ahora asignamos el diseño del fondo a la brocha e inicializamos el texto de los memos en el evento "onCreate":

```

procedure TfInvertir.FormCreate(Sender: TObject);
begin

```

```

fInvertir.Brush.Color:= clInactiveCaptionText;
fInvertir.Brush.Style:= bsDiagCross;
mNumeroEntero.Text:= '0'; mNumeroInvertido.Text:= '0';
end;

```

Damos el foco y seleccionamos el texto del memo "mNumeroEntero" en el evento "onActivate" de la forma:

```

procedure TfInvertir.FormActivate(Sender: TObject);
begin
    mNumeroEntero.SetFocus; mNumeroEntero.SelectAll;
end;

```

Validamos la introducción de datos al memo "mNumeroEntero" programando el evento "onKeyPress":

```

procedure TfInvertir.mNumeroEnteroKeyPress(Sender: TObject; var Key: Char);
begin
    case Key of
        #8, '0'..'9', '-':
            else Beep; Abort;
    end;
end;

```

Damos funcionalidad a la aplicación programando el evento "onClick" del BitBtn "bbInvertir":

```

procedure TfInvertir.bbInvertirClick(Sender: TObject);
var n,ni: Int64;
begin
    n:= LeerNumero(mNumeroEntero);
    ni:= Invertir(n);
    MostrarNumero(mNumeroInvertido,ni);
    mNumeroEntero.SetFocus; mNumeroEntero.SelectAll;
end;

```

Finalmente permitimos salir de la aplicación a través del BitBtn "bbSalir", programando su evento "onClick":

```

procedure TfInvertir.bbSalirClick(Sender: TObject);
begin
    fInvertir.Close;
end;

```

Que como vemos, lo único que hace es llamar al método "Close" de la forma. Una vez corregidos los errores la aplicación deberá tener la apariencia y devolver el resultado mostrado en la figura correspondiente a la interfaz de la aplicación.

11.4.5. Cálculo del capital acumulado en un determinado número de periodos

Como quinto ejemplo elaboraremos una aplicación para calcular el capital que se acumula cuando se deposita un ahorro, durante un determinado número de periodos, con una cierta tasa de interés por periodo.

El razonamiento recursivo para resolver este problema es el siguiente: Podemos calcular el capital acumulado al periodo "n", si conocemos el capital acumulado hasta el periodo "n-1", pues sería ese valor más el interés ganado por dicho valor en el último periodo.

Así por ejemplo podemos calcular el capital acumulado en 60 meses por 10000 dólares a una tasa de interés del 0.5% mensual, si al capital acumulado en 59 meses (132421.3945527) le sumamos el interés del último mes:

$$\text{CapAcum}(10000, 60) = \text{CapAcum}(10000, 59) + \text{CapAcum}(10000, 59) * 0.5/100$$

$$\text{CapAcum}(10000, 60) = \text{CapAcum}(10000, 59) * (1 + 0.5/100)$$

$$\text{CapAcum}(10000, 60) = 13421.3945527 * 1.005 = 13488.5015255$$

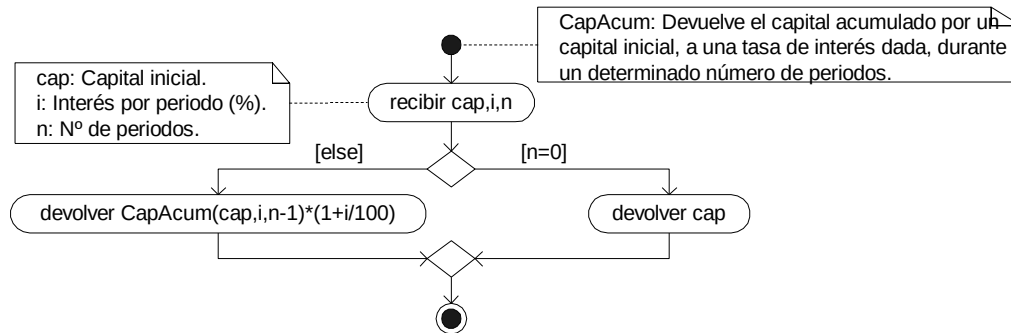
Dado que el capital no cambia en 0 periodos, pues no recibe ningún interés, esta será nuestra condición de finalización. Entonces la fórmula recursiva resultante del anterior razonamiento es:

$$\text{CapAcum}(\text{cap}, i, n) = \text{CapAcum}(\text{cap}, i, n-1) * (1 + i/100)$$

$$\text{CapAcum}(\text{cap}, 0) = \text{cap}$$

Donde "cap" es el capital inicial, "n" es el número de periodos e "i" es el interés (en porcentaje) por cada periodo.

El algoritmo que resuelve este problema es:



Para codificar este algoritmo creamos una nueva aplicación (File -> New Application), una nueva unidad (File -> New Unit) y guardemos la aplicación (File -> Save Project As...) en el directorio "Recursividad\Capital Acumulado" con los siguientes nombres: "ufCapAcum" para "Unit1", "uCapAcum" para "Unit2" y "pCapAcum" para "Project1".

Escribimos entonces el código (incluyendo los módulos de lectura y escritura) en la unidad "uCapAcum":

```
unit uCapAcum;
```

```
interface
```

```
  uses StdCtrls, SysUtils, Math, QDialogs;
```

```
  function LeerCapitalInicial(m: TMemo): double;
```

```
  function LeerInteres(m: TMemo): double;
```

```
  function LeerNumPeriodos(m: TMemo): word;
```

```
  function CapAcum(cap,i: double; n: word): double;
```

```
  procedure MostrarCapitalAcumulado(m: TMemo; cap: double);
```

```
implementation
```

```
  function LeerCapitalInicial(m: TMemo): double;
```

```
  begin
```

```
    try
```

```
      result:= StrToFloat(m.Text);
```

```
      if result<=0 then raise EInvalidArgument.Create(
```

```
        'El capital inicial no puede ser negativo ni cero');
```

```
    except
```

```
      on EConvertError do begin
```

```
        MessageDlg('El capital inicial está mal escrito',mtError,[mbOK],0);
```

```

        m.SetFocus; m.SelectAll; Abort; end;
    on e: EInvalidArgument do begin
        MessageDlg(e.Message,mtError,[mbOK],0);
        m.SetFocus; m.SelectAll; Abort; end;
    end;
end;

function LeerInteres(m: TMemo): double;
begin
    try
        result:= StrToFloat(m.Text);
        if result<=0 then raise EInvalidArgument.Create(
            'El interés por periodo no puede ser negativo ni cero');
    except
        on EConvertError do begin
            MessageDlg('El interés está mal escrito',mtError,[mbOk],0);
            m.SetFocus; m.SelectAll; Abort; end;
        on e: EInvalidArgument do begin
            MessageDlg(e.Message,mtError,[mbOK],0);
            m.SetFocus; m.SelectAll; Abort; end;
        end;
    end;
end;

function LeerNumPeriodos(m: TMemo): word;
begin
    try
        result:= StrToInt(m.Text);
    except
        on EConvertError do begin
            MessageDlg('El número de periodos está mal escrito',mtError,[mbOK],0);
            m.SetFocus; m.SelectAll; Abort; end;
        end;
    end;
end;

function CapAcum(cap,i: double; n: word): double;
begin
    if n=0 then result:= cap else result:= CapAcum(cap,i,n-1)*(1+i/100);
end;

procedure MostrarCapitalAcumulado(m: TMemo; cap: double);
begin
    m.Text:= Format('%14.12m',[cap]);
end;

end.

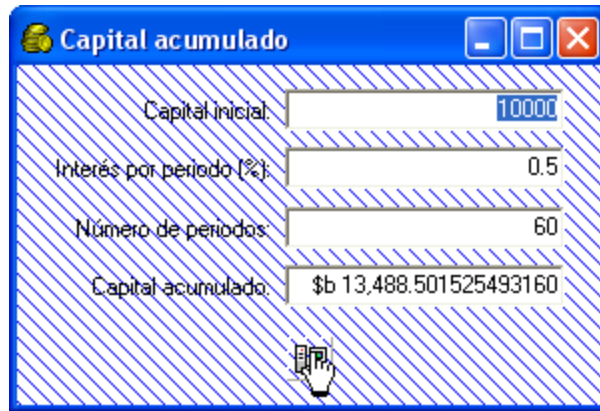
```

Ahora elaboramos la interfaz de la aplicación de manera que, en ejecución, se vea aproximadamente como se muestra en la figura de la siguiente página. Todos los componentes de esta aplicación son conocidos y las propiedades que se han modificado en los mismos son:

Icono de la aplicación: Finance.ico

Form1: Name = 'fCapAcum'; Caption = 'Capital acumulado'; Height = 204; Width = 301; Position = poScreenCenter.

Label1: Caption = 'Capital inicial:'; Label2: Caption = 'Interés por periodo (%):'; Label3: Caption = 'Número de periodos:'; Label4: Caption = 'Capital acumulado:'; Propiedades comunes: Transparent = True; Alignment = ta-RightJustify.



```

Memo1: Name = 'mCapitalInicial'; Cursor = crIBeam; Memo2: Name = 'mInteres';
Cursor = crIBeam; Memo3: Name = 'mNumPeriodos'; Cursor = crIBeam;
Memo4: Name = 'mCapitalAcumulado'; ReadOnly = True; TabStop = False; Cursor
= crArrow; Propiedades comunes: WantReturns = False; Height = 21; Width =
140; Alignment = taRightJustify;

```

```

SpeedButton1: Name = 'sbCalcular'; Glyph=comppec2.bmp; Hint= 'Calcula el
capital acumulado en n periodos'; ShowHint=True; Cursor=crHandPoint; Trans-
parent=True.

```

Asegúrese también que el orden de los componentes sea (Edit -> Tab Order): mCapitalInicial, mInteres, mNumPeriodos, mCapitalAcumulado y sbCalcular.

Ahora asignamos la imagen del fondo a la brocha e inicializamos el texto de los memos en el evento "onCreate":

```

procedure TfCapAcum.FormCreate(Sender: TObject);
begin
    fCapAcum.Brush.Color:= clBlue;
    fCapAcum.Position:= poScreenCenter;
    fCapAcum.Brush.Style:= bsFDiagonal;
    mCapitalInicial.Text:= '0';
    mInteres.Text:= '0';
    mNumPeriodos.Text:= '0';
    mCapitalAcumulado.Text:= '0';
end;

```

Damos el foco y seleccionamos el texto del memo "mCapitalInicial" programando el evento "onActivate" de la forma:

```

procedure TfCapAcum.FormActivate(Sender: TObject);
begin
    mCapitalInicial.SetFocus; mCapitalInicial.SelectAll;
end;

```

Validamos la introducción de datos a los memos "mCapitalInicial" y "mInteres" seleccionándolos y programando el evento "onKeyPress" del memo "mCapitalInicial":

```

procedure TfCapAcum.mCapitalInicialKeyPress(Sender: TObject; var Key: Char);
begin
    case Key of
        #8, '0'..'9', '.', ':';
        #13: sbCalcular.Click;
        else Beep; Abort;
    end;
end;

```


Validamos también la introducción de datos al memo "mNumperiodos" programando el evento "onKeyPress" del mismo:

```
procedure TfCapAcum.mNumPeriodosKeyPress(Sender: TObject; var Key: Char);
begin
  case Key of
    #8, '0'..'9':;
    #13: sbCalcular.Click;
    else Beep; Abort;
  end;
end;
```

Observe que en ambas validaciones se ha programado la tecla "Enter" (#13) para que simule el evento "onClick" del SpeedButton "sbCalcular".

Para que el texto quede seleccionado cuando se ingresa a los memos "mInteres" o "mNumPeriodos", seleccionamos ambos objetos y programamos el evento "onEnter" del memo "mInteres":

```
procedure TfCapAcum.mInteresEnter(Sender: TObject);
begin
  TMemo(Sender).SelectAll;
end;
```

Finalmente damos funcionalidad a la aplicación programando el evento "onClick" del SpeedButton "sbCalcular":

```
procedure TfCapAcum.sbCalcularClick(Sender: TObject);
var ci,ca,i: double; n: word;
begin
  ci:= LeerCapitalInicial(mCapitalInicial);
  i:= LeerInteres(mInteres);
  n:= LeerNumPeriodos(mNumPeriodos);
  ca:= CapAcum(ci,i,n);
  MostrarCapitalAcumulado(mCapitalAcumulado,ca);
  mCapitalInicial.SetFocus; mCapitalInicial.SelectAll;
end;
```

Una vez corregidos los errores la aplicación deberá tener la apariencia y devolver el resultado que se muestra en la figura correspondiente a la interfaz de la aplicación.

11.5. Ejercicios

1. Elabore una aplicación con un "BitBtn" y programe el evento "onClick" del "BitBtn" de manera que muestre en un cuadro de mensaje, el número de veces que se ha hecho clic sobre el mismo.
2. Elabore una aplicación con un "Memo" y en la misma programe los eventos "onCreate" y "onClose" de la forma de manera que su fondo sea la figura "gotas de agua.bmp", que el Memo tenga: 200 puntos de ancho y 300 de alto, con los días de la semana en color azul, centrados (uno en cada línea), con los estilos negrita, cursiva y subrayado y que no permita escribir ningún dato en el mismo.
3. Elabore una aplicación con un "ComboBox" y programe los eventos "onCreate" y "onClose" de la forma de manera que su fondo sea la figura "roca.bmp", tenga el título "RECURSIVIDAD", el icono "Libro.ico", que el ComboBox tenga los números del 1 al 10000, permitiendo seleccionar valores pero no escribirlos y que el al empezar la aplicación esté seleccionado el décimo elemento.
4. Elabore una aplicación con un "RadioGroup" y programe el evento "onCreate" de la forma de manera que su fondo sea de color verde y con lí-

neas diagonales, el RadioGroup tenga el título "Tipos enteros" (en color azul, 30 puntos y negrita) y sus elementos sean los tipos enteros existentes en Delphi, presentados en 4 columnas, estando seleccionado el tipo Integer.

5. Elabore una aplicación con un "SpeedButton" y programe los eventos "onCreate" y "onClose" de la forma de manera que su fondo sea la figura "arena.bmp", el SpeedButton tenga la figura "Sumatoria.bmp", se active al pulsar la tecla "Enter", que muestre la pista "Calcular la sumatoria de los datos" y que la forma del puntero cambie a una mano apuntando cuando se encuentre sobre el botón.
6. Elabore una aplicación con 2 Edits y en la misma cree el módulo "MostrarNumReal" que reciba un número real y un edit y muestre en el edit el número real en 18 espacios, redondeado al sexto dígito en la parte fraccionaria. Pruebe el módulo en el evento "onClick" de la forma con los números "432.32324232344" y "589084.32".
7. Elabore una aplicación con 2 memos y en la misma cree el módulo "MostrarNumEntero" que reciba un número entero y un memo y muestre en el memo el número entero con 10 dígitos, completando con ceros a la izquierda los dígitos faltantes. Haga la prueba en el evento "onContextPopup" de la forma con los números "43234" y "1348203".
8. Elabore una aplicación con 3 memos y en la misma cree el módulo "MostrarNumReal" que reciba un número real y un memo y muestre en el memo el número real en 20 espacios, con 4 dígitos en la parte fraccionaria y con separador de miles. Pruebe el módulo en el evento "onClick" de la forma con los números "4534.32344039", "804920390054.3" y "18.32".
9. Elabore una aplicación con tres "Memos" y en la misma cree el módulo "MostrarNumReal" que reciba un número real y un memo y muestre en el memo el número real en 22 espacios, con un máximo de 8 dígitos, pruebe el módulo en el evento "onDblClick" con los números "44343.323233434", "382.323" y "452.3x10⁴".
10. Elabore una aplicación con un memo y en la misma cree el módulo "LeerNumEntero", que reciba un memo y devuelva el número entero equivalente al texto del memo. El módulo debe generar un mensaje con ShowMessage en caso de que el número esté mal escrito. Se sabe que el número es positivo y que no supera los 20000. Pruebe el módulo con los números "4323" y "5904904", mostrando los resultados en un cuadro de diálogo.
11. Elabore una aplicación con un "Edit" y programe el evento "onKeyPress" del mismo de manera que sólo permita escribir números enteros y que al pulsar la tecla "Enter" muestre dicho número en formato hexadecimal, con 10 dígitos, llenando con ceros los dígitos faltantes.
12. Elabore una aplicación con un "Memo" y programe el evento "onKeyPress" del mismo de manera que sólo permita introducir números reales y que al pulsar la tecla "Enter" muestre dicho número en formato monetario con un ancho de 14 dígitos.
13. Elabore una aplicación con un "Edit" y programe el evento "onKeyPress" del mismo de manera que sólo permita escribir números reales y que al pulsar la tecla "Esc" muestre dicho número con un ancho de 16 caracteres y un total de 12 dígitos.
14. Elabore una aplicación con un "Memo" y programe el evento "onKeyPress" del mismo de manera que sólo permita escribir números reales y que al pulsar la tecla "Tab" muestre dicho número con un ancho de 20 caracteres, en notación científica con un número de dígitos igual a 12?
15. Elabore una aplicación con un "Edit" y programe el evento "OnKeyPress" del mismo de manera que sólo permita escribir números enteros (positi-

vos o negativos) y que al pulsar la barra de espacio muestre el número con 25 caracteres de ancho con un total de 20 dígitos.

16. Elabore una aplicación con un "Memo" y programe el evento "OnKeyPress" del mismo de manera que sólo permita escribir letras, convirtiéndolas automáticamente en mayúsculas y que al pulsar la tecla "Tab" muestre las letras en 20 caracteres de espacio, alineadas a la izquierda y mostrando un máximo de 10 letras.
17. Elabore una aplicación y en la misma cree el módulo "Segundos", que devuelva la hora del sistema operativo en segundos. Pruebe el módulo en el evento "onContextPopUp" de la forma (mostrando los resultados con TextOut).
18. Elabore una aplicación y en la misma cree el módulo "MilisegundosTranscurridos", que reciba dos tiempos y devuelva los milisegundos transcurridos entre los mismos. Pruebe el módulo en el evento "onClick" de la forma, siendo el primer tiempo el tiempo al momento de hacer correr la aplicación y el segundo el tiempo al momento de hacer click sobre la forma.
19. Elabore una aplicación y en la misma cree el módulo "SegundosTranscurridos", que reciba dos tiempos y devuelva los segundos transcurridos entre los mismos (con 4 dígitos después del punto). Pruebe el módulo en el evento "onClick" de la forma. Si es la primera vez que se hace click sobre la forma, el tiempo en ese momento debe ser registrado como el primer tiempo y el tiempo transcurrido debe ser cero, para la segunda y sucesivas veces que se hace click sobre la forma, el tiempo en ese momento debe ser registrado como el segundo tiempo y el tiempo anterior debe pasar a ser el primero.
20. Elabore una aplicación que empleando la recursividad, calcule el común divisor entre dos números (A y B), mediante el algoritmo de Euclides:

$$\begin{aligned} MCD(A, A) &= A \\ MCD(A, B) &= MCD(B, A) && \text{si } A > B \\ MCD(A, B) &= MCD(A, B - A) && \text{si } A < B \end{aligned}$$

21. Elabore una aplicación que, empleando la recursividad, calcule la función de Ackerman para cualquier par de números enteros no negativos:

$$a(p, q) = \begin{cases} q+1 & \text{si } p=0 \\ a(p-1, 1) & \text{si } p > 0 \text{ y } q=0 \\ a(p-1, a(p, q-1)) & \text{si } p > 0 \text{ y } q > 0 \end{cases}$$

22. Elabore una aplicación y en la misma cree un módulo recursivo que calcule la sumatoria de los primeros "n" números ($\sum i \{i=1, 2, 3, 4, \dots, n\}$). Pruebe el módulo en el evento "onDbClick" de la forma con "n=7" y "n=56", mostrando los resultados en una ventana de mensajes.
23. Elabore una aplicación y en la misma cree un módulo recursivo que calcule la sumatoria de los primeros "n" números impares ($\sum i \{i=1, 3, \dots\}$). Pruebe el módulo en el evento "onDbClick" de la forma con "n=5" y "n=50" mostrando los resultados con "TextOut".
24. Elabore una aplicación con un "Edit" y en la misma cree un módulo recursivo que devuelva el número de dígitos que tiene un número entero positivo o negativo. Programe el evento "onKeyPress" del edit de manera que sólo permita introducir números enteros (positivos o negativos) y que al pulsar la tecla "Esc" muestre el número de dígitos del número introducido.
25. Elabore una aplicación con un "Edit" y en la misma cree un módulo recursivo que devuelva el número de dígitos enteros de un número real. Programe el evento "onKeyPress" del edit de manera que sólo permita in-

troducir números reales y que al pulsar la tecla "Enter" muestre el número de dígitos enteros del número real.

26. Elabore una aplicación y en la misma cree un módulo recursivo que calcular el Chebyshev enésimo ("n") de un número real "x", evitando que un mismo valor sea calculado dos o más veces ($C_n(x) = 2xC_{n-1}(x) - C_{n-2}(x)$; $C_0(x) = 1$; $C_1(x) = x$). Pruebe el módulo en el evento "onClick" de la forma con "n=0, x=3"; "n=1, x=4.3" y "n=4, x=5.5", mostrando los resultados con "TextOut".
27. Elabore una aplicación y en la misma cree un módulo recursivo que calcule, con 14 dígitos de precisión, la función "J" de Bessel de primera especie y orden "n": $J_n(x) = (2n-2)J_{n-1}(x)/x - J_{n-2}(x)$; $J_0(x) = 1 - x^2/2^2 + x^4/(2^2 4^2) - x^6/(2^2 4^2 6^2) + x^8/(2^2 4^2 6^2 8^2) - \dots + x^{24}/(2^2 4^2 6^2 8^2 10^2 12^2 16^2 18^2 20^2 22^2 24^2)$; $J_1(x) = x/2 - x^3/(2^2 * 4) + x^5/(2^2 * 4^2 * 6) - x^7/(2^2 * 4^2 * 6^2 * 8) + x^9/(2^2 4^2 6^2 8^2 10) - \dots + x^{25}/(2^2 4^2 6^2 8^2 10^2 12^2 16^2 18^2 20^2 22^2 24^2 26)$.

12. LLAMADAS CIRCULARES O RECÍPROCAS

Las llamadas circulares o recíprocas constituyen otra forma de lograr iteración, donde se empleando dos módulos en lugar de uno.

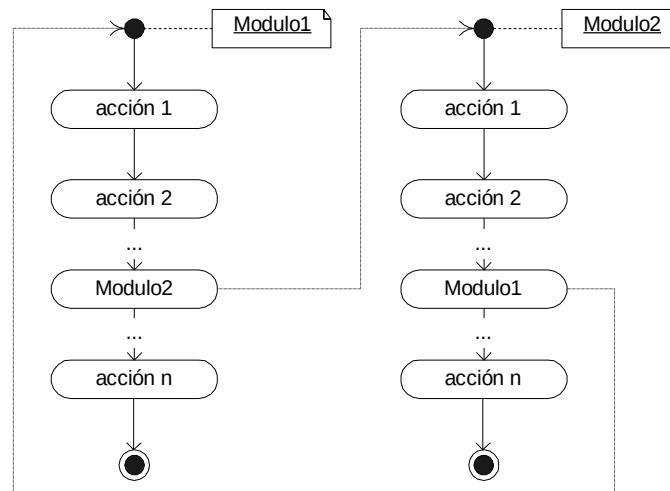
En las llamadas circulares o recíprocas, un módulo llama a otro y este a su vez llama al primero, formándose así un ciclo iterativo.

Si bien todos los problemas iterativos pueden ser resueltos con llamadas circulares o recíprocas, existen muy pocos casos en los cuales las llamadas circulares o recíprocas constituyan una ventaja sobre las estructuras iterativas estándar o la recursividad.

En la práctica sólo se deben emplear llamadas circulares o recíprocas en aquellos raros casos donde su aplicación sea de beneficio evidente, es decir cuando gracias a esta técnica se consigue una solución más sencilla y/o eficiente. En este capítulo, para adquirir práctica en esta técnica, emplearemos llamadas circulares o recíprocas aún cuando su aplicación no represente ningún beneficio.

Por lo tanto, el objetivo del presente capítulo, es que al concluir el mismo estén capacitados para resolver problemas iterativos empleando llamadas circulares o recíprocas.

La lógica involucrada en las llamadas circulares o recíprocas es la que se muestra en el siguiente diagrama de actividades:



Tal como sucede con las llamadas recursivas, en las llamadas circulares debe existir una condición de finalización, pues de lo contrario las llamadas se repetirían por siempre y el ciclo nunca terminaría (ciclo infinito).

Cada vez que un módulo llama a otro se crea una nueva copia del módulo en memoria (en realidad se crea una nueva copia de las variables del módulo), razón por la cual las llamadas circulares (al igual que las recursivas) consumen más memoria y recursos del sistema.

La codificación de las llamadas circulares se realiza casi siempre en dos submódulos el primero de los cuales llama al segundo y este al primero. Debido a que en *Pascal* sólo se puede llamar a un módulo si ha sido declarado previamente, se presenta un problema al momento de implementar las llamadas circulares, pues sin importar el orden en que se escriban los módulos, el primero siempre llama al segundo y como el mismo todavía no ha sido declarado se produce un error en tiempo de compilación.

Este problema se resuelve con la declaración adelantada de módulos. Para hacer una declaración adelantada de un procedimiento o función simplemente se escribe su cabecera seguida de la palabra reservada *Forward*. De esta manera se informa a *Pascal* sobre la existencia del módulo indicándole que el código respectivo se encuentra *más adelante (forward)*. La declaración adelantada de módulos no es necesaria si los mismos han sido declarados en la interfaz (interface) de la aplicación, pues en ese caso ya son conocidos por *Pascal* y pueden ser llamados en cualquier orden.

Las llamadas circulares o recíprocas constituyen simplemente otra forma de lograr iteración, pero no otra forma de pensar (como ocurre con la recursividad) y pueden ser empleadas tanto para el razonamiento directo como para el razonamiento recursivo.

12.1. Ejemplos

12.1.1. Cálculo del Impuesto al Valor Agregado (IVA)

Como primer ejemplo elaboraremos una aplicación para calcular el IVA.

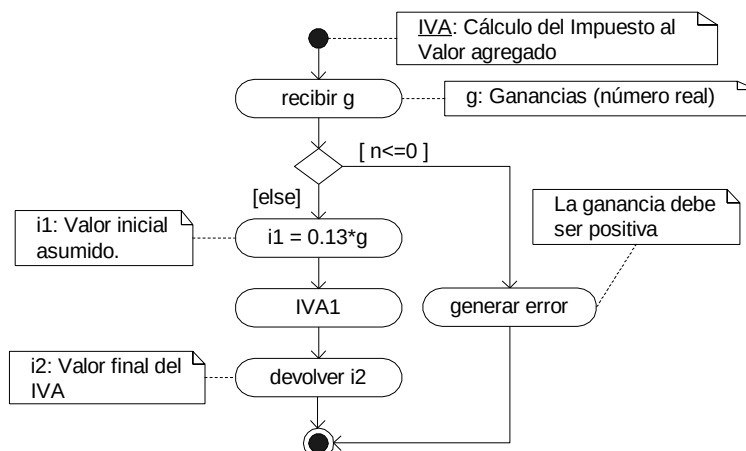
El IVA es el 13% de las ganancias. Si somos rigurosos el IVA debería ser calculado multiplicando por 0.13 las ganancias netas (sin incluir el IVA pues no es una ganancia). En consecuencia la ecuación para calcular el IVA es:

$$IVA2 = (Ganancia - IVA1) * 0.13$$

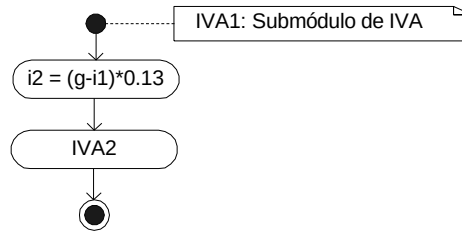
La lógica para resolver esta ecuación es similar a la de la raíz cuadrada o cúbica: comenzando con un valor inicial asumido "IVA1" (generalmente el 13% de las ganancias brutas) se calcula "IVA2", entonces si "IVA1" e "IVA2" son iguales en un determinado número de dígitos el proceso concluye, siendo la respuesta "IVA2", caso contrario "IVA1" toma el valor de "IVA2" y el proceso se repite.

Por lo tanto lógica para resolver el problema es directa, por lo que puede ser resuelto eficientemente con alguna de las estructuras iterativas estándar y eso es lo que haríamos en un caso real, no obstante en este capítulo, y sólo con el objetivo de adquirir práctica, resolveremos el problema empleando llamadas circulares. Para ello emplearemos dos submódulos en uno de los cuales se calculará el nuevo valor del IVA y en el otro se compararán los dos últimos valores calculados.

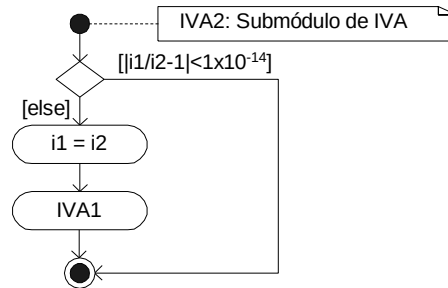
El diagrama de actividades del módulo principal (no circular) es el siguiente:



El diagrama de actividades del primer submódulo es:



Y el del segundo:



Para codificar estos módulos creamos una nueva aplicación (File -> New Application) y la unidad (File -> New Unit) y guardamos la aplicación (File -> Save Project As...) en el directorio "Circulares\IVA" con los siguientes nombres: "ufIVA" para "Unit1", "uIVA" para "Unit2" y "pIVA" para "Project1".

Ahora escribamos el código (incluyendo los módulos de lectura y escritura) en la unidad "uIVA":

```
unit uIVA;
```

```
interface
```

```
  uses StdCtrls, SysUtils, QDialogs, Math;
```

```
  function LeerGanancia(m:TMemo): currency;
```

```
  function IVA(g: currency): currency;
```

```
  function CalcularIVA(m: TMemo): currency;
```

```
  procedure MostrarIVA(m:TMemo;iva: currency);
```

```
implementation
```

```
  function LeerGanancia(m:TMemo): currency;
```

```
  begin
```

```
    try
```

```
      result:= StrToFloat(m.Text);
```

```
    except
```

```
      on EConvertError do begin
```

```
        ShowMessage('La ganancia está mal escrita');
```

```
        m.SetFocus; m.SelectAll; Abort end;
```

```
    end;
```

```
  end;
```

```
  function IVA(g: currency): currency;
```

```
  var i1,i2: currency;
```

```
    procedure IVA2; forward // declaración adelantada de IVA2
```

```
    procedure IVA1;
```

```
    begin i2:= (g-i1)*0.13; IVA2; end;
```

```

procedure IVA2;
begin if abs(i1/i2-1)<1E-14 then exit; i1:= i2; IVA1; end;
begin
  if g<=0 then raise EInvalidArgument.Create(
    'La ganancia debe ser positiva');
  i1:= g*0.13; IVA1; result:= i2;
end;

function CalcularIVA(m: TMemo): currency;
begin
  try
    result:= IVA(LeerGanancia(m));
  except
    on e: EInvalidArgument do begin
      ShowMessage(e.Message); m.SetFocus; m.SelectAll; Abort; end;
    end;
end;

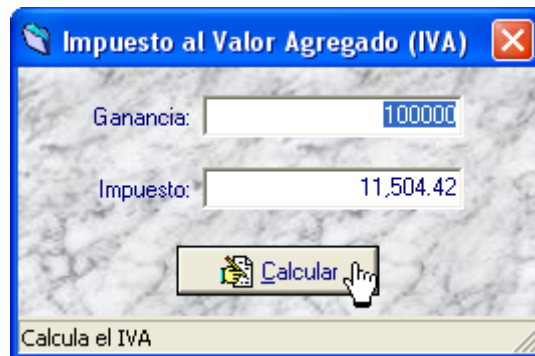
procedure MostrarIVA(m:TMemo;iva: currency);
begin
  m.Text:= FormatFloat('###,###,##0.00',iva);
end;

end.

```

En el módulo "MostrarIva" se ha empleado la función *FormatFloat*, la cual permite dar formato a un número real. Para ello en los lugares donde se quiere que aparezca un número se escribe un '0' o '#'. La diferencia entre estos dos caracteres es la siguiente: si se emplea '0' y no existe un número la función coloca un '0' en ese lugar, si se emplea '#' y no existe un número la función no coloca nada. En el ejemplo se ha colocado dos ceros después del punto, para que en todos los casos se muestre el resultado con dos dígitos después del punto (inclusive cuando el resultado no tenga parte fraccionaria). Observe también que se emplea la coma ',' para separar los miles y que el tipo de dato empleado es "currency" (monetario).

Elaboramos la interfaz de la aplicación de manera que, en ejecución, se vea aproximadamente como se muestra en la siguiente figura:



En esta aplicación existe un nuevo componente en la parte inferior de la pantalla: un TStatusBar (), el cual se encuentra en la pestaña Win32. Este componente se emplea para mostrar información adicional o de ayuda. En este ejemplo se emplea el *StatusBar* para mostrar la sigla de la materia y las pistas (*Hint*) escritas para los memos, el botón y la forma.

El *StatusBar* puede ser dividido en dos o más partes (paneles) para mostrar información diferente en cada uno de ellos, en esta aplicación, sin embargo, utilizaremos un solo panel. Para utilizar un solo panel se debe

colocar la propiedad *SimplePanel* en *True* y el texto que aparece en el panel se escribe en la propiedad *SimpleText*.

Para que las pistas (Hints) de los componentes aparezcan en el *StatusBar*, se debe colocar la propiedad *AutoHint* en *True*.

Otro aspecto nuevo en esta aplicación es que la ventana no cuenta con los iconos que permiten maximizarla y minimizarla (pues en realidad no son necesarios). Los iconos de la forma se añaden o quitan activando o desactivando elementos de la propiedad "BorderIcons" (que es un conjunto) de la forma. Para ello simplemente se coloca en *True* los elementos que se quieren activar y en *False* los que se quieren desactivar.

Las propiedades modificadas en los componentes de esta aplicación son:

Icono de la aplicación: '3D document.ico';

Form1: Name = 'fIVA'; Caption = 'Impuesto al Valor Agregado'; Height = 177; Width = 267; Position = poScreenCenter; BorderIcons = [biSystemMenu]; Hint = 'SIS101: Cálculo del IVA'.

Label1: Caption = 'Ganancia:'; Label2: Caption = 'Impuesto:'; Propiedades comunes: Transparent = True; Alignment = taRightJustify.

Mem1: Name = 'mGanancia'; Cursor = crIBeam; Hint = 'Ganancia bruta' Mem2: Name = 'mImpuesto'; Cursor = crArrow; ReadOnly = True; Hint = 'Impuesto IVA calculado'; Propiedades comunes: Height = 21; Width = 139, WantReturns = False; Alignment = taRightJustify; Color= clNavy.

BitBtn1: Name = 'bbCalcular'; Caption = '&Calcular'; Glyph= 'npadwrite.bmp', Cursor = crHandPoint; Default = True; Hint= 'Cálcula el IVA'.

StatusBar1: Name = 'sbIVA'; AutoHint = True; SimplePanel = True; SimpleText = 'SIS101: Cálculo del IVA'.

Asegúrese también que el orden de los componentes sea (Edit -> Tab Order): mGanancia, mImpuesto y bbCalcular.

Ahora asignamos la imagen de la brocha e inicializamos el texto de los memos en el evento "onCreate" de la forma:

```
procedure TfIVA.FormCreate(Sender: TObject);
begin
  fIVA.Brush.Bitmap:= TBitmap.Create;
  fIVA.Brush.Bitmap.LoadFromFile('Mármol blanco.bmp');
  MostrarIVA(mGanancia,0); MostrarIVA(mImpuesto,0);
end;
```

Como de costumbre, liberamos la imagen en el evento "onClose":

```
procedure TfIVA.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  fIVA.Brush.Bitmap.Free;
end;
```

Asignamos el foco y seleccionamos el texto del memo "mGanancia" en el evento "onActivate" de la forma:

```
procedure TfIVA.FormActivate(Sender: TObject);
begin
  mGanancia.SetFocus; mGanancia.SelectAll;
end;
```

Validamos la introducción de datos al memo "mGanancia" en el evento "onKeyPress":

```
procedure TfIVA.mGananciaKeyPress(Sender: TObject; var Key: Char);
```

```

begin
  case Key of
    #8, '0'..'9', '.', '-':
      else Beep; Abort;
    end;
end;

```

Finalmente damos funcionalidad a la aplicación en el evento "onClick" del BitBtn "bbCalcular":

```

procedure TfIVA.bbCalcularClick(Sender: TObject);
var i: currency;
begin
  i:= CalcularIVA(mGanancia);
  MostrarIVA(mImpuesto,i);
  mGanancia.SetFocus; mGanancia.SelectAll;
end;

```

Una vez corregidos los errores la aplicación deberá tener la apariencia y calcular el resultado mostrado en la figura correspondiente a la interfaz de la aplicación.

12.1.2. Inversión de los dígitos de un número entero

Como segundo ejemplo volveremos a resolver el problema de invertir los dígitos de un número entero positivo o negativo (resuelto en el capítulo de recursividad), en este caso sin embargo seguiremos un razonamiento directo en lugar del razonamiento recursivo.

La solución se basa en que el residuo de la división del número entero entre 10 nos devuelve el último dígito de dicho número, mientras que el cociente contiene los dígitos restantes. Así por ejemplo si el número es 12345, el residuo de 12345/10 es "5" y el cociente "1234". Ahora si volvemos a dividir el cociente entre 10: 1234/10, nuevamente el residuo nos devuelve el último dígito "4" y el cociente los dígitos restantes "123". Si proseguimos de esta manera 3 veces más obtenemos los residuos: "3", "2" y "1". Podemos ver entonces que los residuos obtenidos en cada una de estas divisiones corresponden en el orden correcto al número invertido: "5", "4", "3", "2" y "1".

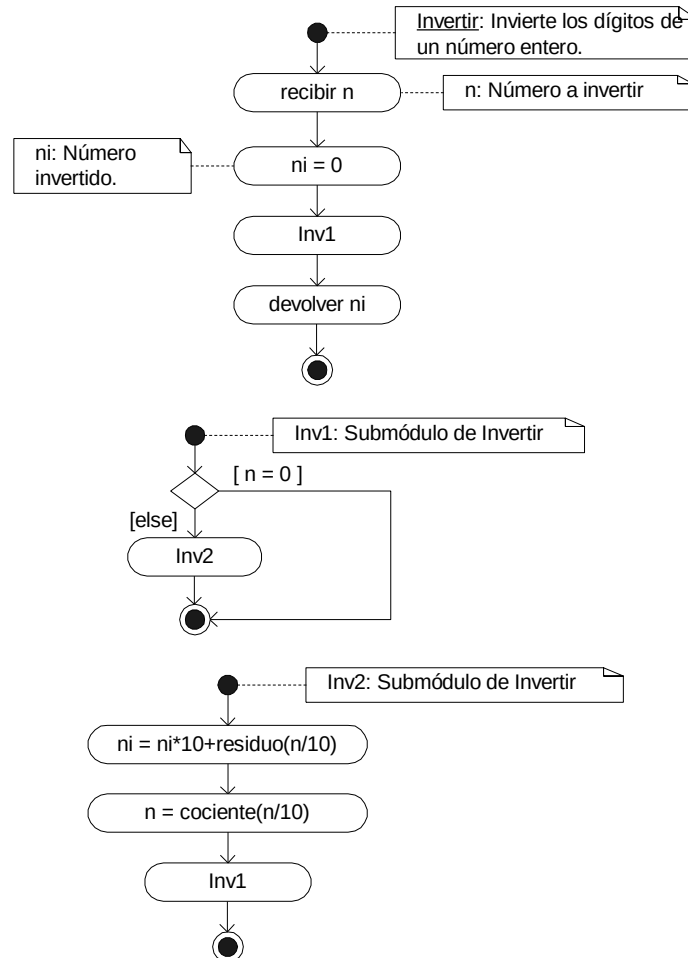
Ahora bien, para armar el número resultante invertido no podemos simplemente sumar los números, pues en este caso por ejemplo nos daría: $5+4+3+2+1 = 15$, que obviamente no es el número invertido. El procedimiento que se sigue es el siguiente: multiplicamos el residuo de la primera división (5) por 10: $5*10 = 50$ y le sumamos el residuo de la segunda división (4): $50+4 = 54$, entonces multiplicamos este resultado por 10 y le sumamos el residuo de la tercera división (3): $54*10+3 = 543$, este resultado volvemos a multiplicar por 10 y le sumamos el residuo de la cuarta división (2): $543*10+2 = 5432$, finalmente este resultado volvemos a multiplicar por 10 y le sumamos el residuo de la última división (1): $5432*10+1 = 54321$, obteniendo así el número invertido correcto.

Las divisiones consecutivas de los cocientes del número entre 10 se repiten hasta que el cociente es cero o hasta que el número es menor a 10 (es decir tiene un solo dígito), en cuyo caso el número invertido es obviamente el mismo número.

Para resolver el problema con llamadas circulares emplearemos una vez más dos submódulos, en uno de ellos simplemente se verificará si el número a invertir es cero, en cuyo caso la respuesta (el número invertido) es cero,

caso contrario se llama al segundo submódulo donde se calcula la respuesta en base al razonamiento antes descrito.

Los algoritmos del módulo principal y de los submódulos circulares son los siguientes:



Para codificar estos módulos creamos una nueva aplicación (File -> New Application) y la unidad (File -> New Unit) y guardamos la aplicación (File -> Save Project As...) en el directorio "Circulares\Invertir" con los siguientes nombres: "uInvertir" para "Unit1", "uInvertir" para "Unit2" y "pInvertir" para "Project1".

Ahora escribamos el código (incluyendo los módulos de lectura y escritura) en la unidad "uInvertir":

```
unit uInvertir;
```

```
interface
```

```
    uses SysUtils, StdCtrls, QDialogs;
```

```
    function LeerNumero(m: TMemo): int64;
```

```
    function Invertir(n: int64): int64;
```

```
    procedure MostrarNumInvertido(m: TMemo; ni: int64);
```

```
implementation
```

```

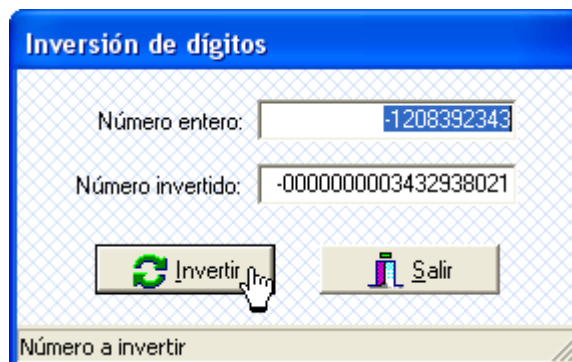
function LeerNumero(m: TMemo): int64;
begin
    try
        result:= StrToInt(m.Text);
    except
        on eConvertError do begin
            ShowMessage('El número ha invetir está mal escrito');
            m.SetFocus; m.SelectAll; Abort; end;
        end;
    end;

function Invertir(n: int64): int64;
var ni: int64;
    procedure Inv2; forward;
    procedure Inv1;
    begin
        if n=0 then exit;
        Inv2;
    end;
    procedure Inv2;
    begin
        ni:= ni*10+n mod 10;
        n:= n div 10;
        inv1;
    end;
begin
    ni:=0;
    Inv1;
    result:= ni;
end;

procedure MostrarNumInvertido(m: TMemo; ni: int64);
begin
    m.Text:= Format('%12.12d',[ni])
end;
end.

```

Elaboramos la interfaz de la aplicación de manera que, en ejecución, se vea aproximadamente como se muestra en la siguiente figura:



La interfaz de esta aplicación es muy similar a la del capítulo de recursividad, excepto que ahora existe un StatusBar y se han ocultado los iconos de la ventana. Las propiedades modificadas son las siguientes:

Icono de la aplicación: 'Fish.ico' (aun cuando no se ve en ejecución, pero sí aparece en la lista de los programas en ejecución).

Form1: Name = 'fInvertir'; Caption = 'Inversión de dígitos'; Height = 180; Width = 287; Position = poScreenCenter; BorderIcons = []; Hint = 'SIS101: Inversión de dígitos'.

Label1: Caption = 'Número entero:'; Label2: Caption = 'Número invertido:'; Propiedades comunes: Transparent = True; Alignment = taRightJustify.

Memo1: Name = 'mNumEntero'; Cursor = crIBeam; Hint = 'Número a invertir'
Memo2: Name = 'mNumInvertido'; Cursor = crArrow; ReadOnly = True; TabStop = False; Hint = 'Número invertido'; Propiedades comunes: Height = 21; Width = 130, WantReturns = False; Alignment = taRightJustify.

BitBtn1: Name = 'bbInvertir'; Caption = '&Invertir'; Kind= bkRetry; Hint = 'Invertir número'; Default = True; BitBtn2: Name = 'bbSalir'; Caption = '&Salir'; Kind= bkClose; Hint = 'Salir de la aplicación'; Propiedades comunes: Cursor = crHandPoint.

StatusBar1: Name = 'sbInvertir'; AutoHint = True; SimplePanel = True; SimpleText = 'SIS101: Inversión de dígitos'.

Asegúrese también que el orden de los componentes sea (Edit -> Tab Order): mNumEntero, mNumInvertido, bbInvertir y bbSalir.

Ahora cambiamos el estilo de la brocha e inicializamos el texto de los memos en el evento "onCreate" de la forma:

```
procedure TfInvertir.FormCreate(Sender: TObject);
begin
    fInvertir.Brush.Color:= clInactiveCaptionText;
    fInvertir.Brush.Style:= bsDiagCross;
    mNumEntero.Text:= '0'; mNumInvertido.Text:= '0';
end;
```

Asignamos el foco y seleccionamos el texto del memo "mNumEntero" en el evento "onActivate" de la forma:

```
procedure TfInvertir.FormActivate(Sender: TObject);
begin
    mNumEntero.SetFocus; mNumEntero.SelectAll;
end;
```

Validamos la introducción de datos al memo "mNumEntero" programando su evento "onKeyPress":

```
procedure TfInvertir.mNumEnteroKeyPress(Sender: TObject; var Key: Char);
begin
    case Key of
        #8, '0'..'9', '-':
            else Beep; Abort;
    end;
end;
```

Hacemos que cambie el texto del StatusBar "sbInvertir" cuando el foco está en el memo "mNumEntero" o "mNumInvertido", programando el evento "onEnter" del memo "mNumEntero":

```
procedure TfInvertir.mNumEnteroEnter(Sender: TObject);
begin
    sbInvertir.SimpleText:= (Sender as TMemo).Hint;
end;
```

Observe que en este caso se está empleando el operador "as" (como), el mismo que nos permite tratar un objeto "como" si fuera de otro tipo, en este caso tratamos el objeto "Sender" de tipo "TObject" como si fuere de tipo "TMemo", y sabemos que ello es posible porque el evento se está generando en

un objeto de tipo "TMemo", por lo tanto el objeto que envía (sender) el evento es en realidad de ese tipo. Esta es la forma preferida en programación orientada a objetos, mientras que en programación estructurada se emplea principalmente el moldeado de tipos, que ya se ha estudiado en capítulos anteriores, donde para tratar un objeto (o una variable) como si fuera de otro tipo simplemente se escribe el nombre del tipo de dato y entre paréntesis el nombre del objeto o de la variable, así la instrucción del anterior evento, empleando moldeado de tipos, sería:

```
sbInvertir.SimpleText:= TMemo(Sender).Hint;
```

Se debe tener en cuenta que tanto con el operador "as" como con el moldeado de tipos, la responsabilidad de tratar un objeto de un tipo como si fuera de otro, recae en el programador, pues en estos casos el compilador hace simplemente lo que se le ordena, aun cuando dicha orden no sea lógica. Por lo tanto es necesario ser meticuloso para no cometer errores y provocar fallos en la aplicación e inclusive en el sistema operativo.

Ahora hacemos un tratamiento similar al anterior cuando el foco está en uno de los botones "bbInvertir" o "bbSalir", programando el evento "onEnter" del BitBtn "bbInvertir" (recuerde que para que dos o más componentes compartan un mismo evento se programa dicho evento estando seleccionados los componentes o alternativamente se hace el programa para uno de los componentes y para los otros se selecciona el nombre del módulo programado en el inspector de objetos):

```
procedure TfInvertir.bbInvertirEnter(Sender: TObject);
begin
  sbInvertir.SimpleText:= (Sender as TBitBtn).Hint;
end;
```

Damos funcionalidad al BitBtn "bbSalir", programando su evento "onClick":

```
procedure TfInvertir.bbSalirClick(Sender: TObject);
begin
  fInvertir.Close;
end;
```

Finalmente damos funcionalidad a la aplicación programando el evento "onClick" del BitBtn "bbInvertir":

```
procedure TfInvertir.bbInvertirClick(Sender: TObject);
var n,ni: Int64;
begin
  n:= LeerNumero(mNumEntero);
  ni:= Invertir(n);
  MostrarNumInvertido(mNumInvertido,ni);
  mNumEntero.SetFocus; mNumEntero.SelectAll;
end;
```

Una vez corregidos los errores la aplicación deberá tener la apariencia y devolver el resultado que se muestra en la figura correspondiente a la interfaz de la aplicación.

12.1.3. Cálculo del Chebyshev

Como tercer ejemplo elaboraremos una aplicación para calcular el Chebyshev enésimo de un número real.

La ecuación de definición del Chebyshev es la siguiente:

$$T_{n+1}(x) - 2xT_n(x) + T_{n-1}(x) = 0; \quad T_0(x) = 1 \text{ y } T_1(x) = x$$

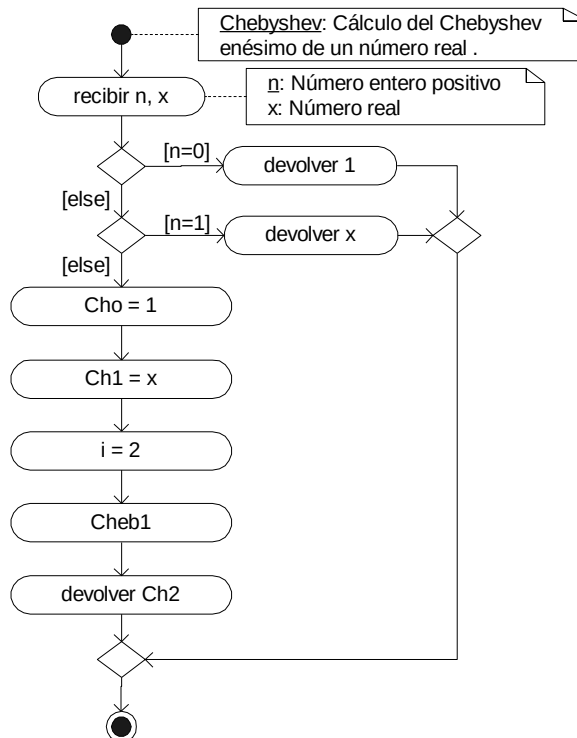
Que escrita en función de la variable " n " en lugar de " $n+1$ " es:

$$T_n(x) = 2xT_{n-1}(x) - T_{n-2}(x); \quad T_0(x) = 1 \text{ y } T_1(x) = x$$

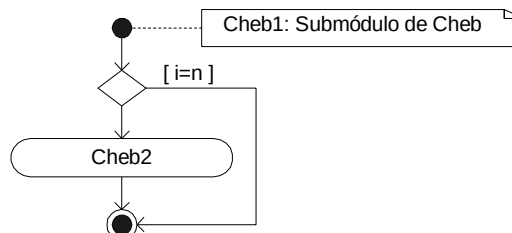
Como ya se explicó al estudiar la recursividad, la programación directa de la formula recursiva da lugar a una lógica errónea (igual que la fórmula del *Fibonacci*), razón por la cual este problema será resuelto siguiendo un razonamiento directo: Comenzando con los valores conocidos (T_0 y T_1) se calcula el *Chebyshev* de 2, luego con T_1 y T_2 se calcula T_3 , posteriormente con T_2 y T_3 se calcula T_4 y así sucesivamente hasta llegar al *Chebyshev* enésimo que se quiere calcular.

Al igual que en los ejemplos anteriores, se empleará un módulo principal y dos submódulos circulares, en uno de dichos submódulos simplemente se comprobará si ya se ha alcanzado el número cuyo *Chebyshev* se quiere calcular terminando el proceso de ser así y llamando al segundo, para que calcule el siguiente valor del *Chebyshev* en caso contrario.

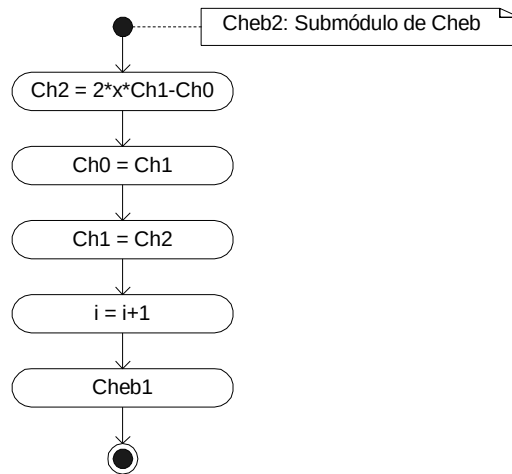
El algoritmo del módulo principal, donde se reciben los parámetros, se analizan los casos donde no es necesario realizar ningún cálculo pues ya se conoce la respuesta, se inicializan las variables y se llama al primer módulo circular y devolviendo el resultado calculado es el siguiente:



El algoritmo del primer submódulo circular, donde simplemente se verifica si ya se ha alcanzado el valor enésimo (n) es:



Y el algoritmo del segundo submódulo circular, donde se calcula el nuevo valor del Chebyshev es:



Para codificar estos módulos creamos una nueva aplicación (File -> New Application) y la unidad (File -> New Unit) y guardamos la aplicación (File -> Save Project As...) en el directorio "Circulares\Chebyshev" con los siguientes nombres: "ufChebyshev" para "Unit1", "uChebyshev" para "Unit2" y "pChebyshev" para "Project1".

Ahora escribamos el código (incluyendo los módulos de lectura y escritura) en la unidad "uChebyshev":

```
unit uCheb;
```

```
interface
```

```
    function Cheb(n: word; x: real): real;
```

```
implementation
```

```
    function Cheb(n: word; x: real): real;
```

```
    var Ch0,Ch1,Ch2: real; i: word;
```

```
    procedure Cheb2; Forward;
```

```
    procedure Cheb1;
```

```
    begin
```

```
        Ch2:= 2*x*Ch1-Ch0;
```

```
        Cheb2;
```

```
    end;
```

```
    procedure Cheb2;
```

```
    begin
```

```
        if i=n then exit;
```

```
        Ch0:= Ch1;
```

```
        Ch1:= Ch2;
```

```
        inc(i);
```

```
        Cheb1;
```

```
    end;
```

```
    begin
```

```
        case n of
```

```
            0: result:= 1;
```

```
            1: result:= x;
```

```
        else
```

```
            Ch0:= 1;
```

```
            Ch1:= x;
```



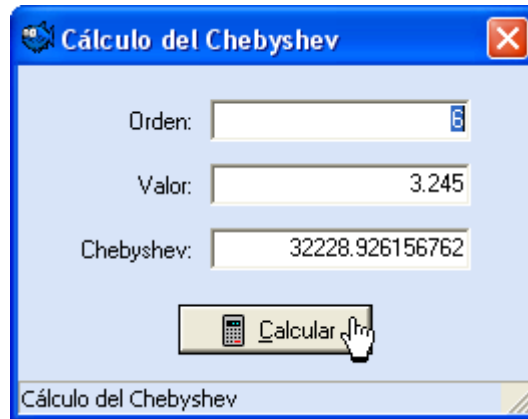
```

        i:= 2;
        Cheb1;
        result:= Ch2;
    end;
end;

end.

```

Elaboramos la interfaz de la aplicación de manera que, en ejecución, se vea aproximadamente como se muestra en la siguiente figura:



Las propiedades modificadas son las siguientes:

Icono de la aplicación: 'Fish Blue.ico'.

Form1: Name = 'fChebyshev'; Caption = 'Cálculo del Chebyshev'; Height = 208; Width = 265; Position = poScreenCenter; BorderIcons = [biSystemeMenu]; Hint = 'SIS101: Chebyshev'; Color = clInactiveCaptionText.

Label1: Caption = 'Orden:'; Label2: Caption = 'Valor:'; Label3: Caption = 'Chebyshev:'; Propiedades comunes: Transparent = True; Alignment = taRightJustify.

Memo1: Name = 'mOrden'; Cursor = crIBeam; Hint = 'Número entero positivo'; Memo2: Name = 'mValor'; Cursor = crIBeam; Hint = 'Número real'; Memo2: Name = 'mChebyshev'; Cursor = crArrow; ReadOnly = True; TabStop = False; Hint = 'Chebyshev'; Propiedades comunes: Height = 21; Width = 130, WantReturns = False; Alignment = taRightJustify.

BitBtn1: Name = 'bbCalcular'; Caption = '&Calcular'; Glyph = 'Calculat.bmp'; Hint = 'Cálculo del Chebyshev'; Default = True; Cursor = crHandPoint.

StatusBar1: Name = 'sbChebyshev'; AutoHint = True; SimplePanel = True; SimpleText = 'SIS101: Chebyshev'.

Asegúrese también que el orden de los componentes sea (Edit -> Tab Order): mOrden, mValor, mChebyshev y bbCalcular.

Ahora inicializamos el texto de los memos en el evento "onCreate" de la forma:

```

procedure TfChebyshev.FormCreate(Sender: TObject);
begin
    mOrden.Text:= '0'; mValor.Text:= '0'; mChebyshev.Text:= '0';
end;

```

Damos el foco y seleccionamos el texto del memo "mOrden" programando el evento "onActivate" de la forma:

```
procedure TfChebyshev.FormActivate(Sender: TObject);
begin
  mOrden.SetFocus; mOrden.SelectAll;
end;
```

Hacemos que cuando reciban el foco los memos "mOrden" y "mValor", se muestre su pista (Hint) y se seleccione el texto del memo, programando el evento "onEnter" del memo "mOrden" (empleando el moldeo de tipos):

```
procedure TfChebyshev.mOrdenEnter(Sender: TObject);
begin
  sbChebyshev.SimpleText:= TMemo(Sender).Hint;
  TMemo(Sender).SelectAll;
end;
```

Igualmente hacemos que se muestre la pista (Hint) del memo "mChebyshev", programando su evento "onEnter" (en este caso empleando el operador "as"):

```
procedure TfChebyshev.mChebyshevEnter(Sender: TObject);
begin
  sbChebyshev.SimpleText:= (Sender as TMemo).Hint;
end;
```

Hacemos lo mismo para el BitBtn "bbCalcular":

```
procedure TfChebyshev.bbCalcularEnter(Sender: TObject);
begin
  sbChebyshev.SimpleText:= bbCalcular.Hint;
end;
```

Validamos la introducción de datos (números enteros) al memo "mOrden", programando su evento "onKeyPress":

```
procedure TfChebyshev.mOrdenKeyPress(Sender: TObject; var Key: Char);
begin
  case Key of
    #8, '0'..'9':
      else Beep; Abort;
  end;
end;
```

Igualmente validamos la introducción de datos (números reales) al memo "mValor", programando su evento "onKeyPress":

```
procedure TfChebyshev.mValorKeyPress(Sender: TObject; var Key: Char);
begin
  case Key of
    #8, '0'..'9', '.', '+', '-', 'E', 'e':
      else Beep; Abort;
  end;
end;
```

Finalmente, damos funcionalidad a la aplicación programando el evento "onClick" del BitBtn "bbCalcular":

```
procedure TfChebyshev.bbCalcularClick(Sender: TObject);
var n: word; x, ch: real;
begin
  n:= LeerOrden(mOrden);
  x:= LeerValor(mValor);
  ch:= Chebyshev(n, x);
  MostrarChebyshev(mChebyshev, ch);
  mOrden.SetFocus; mOrden.SelectAll;
end;
```

Una vez corregidos los errores, la aplicación deberá verse y devolver el resultado que se muestra en la figura correspondiente a la interfaz de la aplicación.

12.2. Preguntas y ejercicios

1. Cree una aplicación con un memo y programe el evento "onKeyPress" del memo de manera que sólo permita escribir números reales y que al pulsar la tecla escape el número real escrito se muestre con separador de miles y tres dígitos después del punto.
2. Cree una aplicación con un edit y programe el evento "onKeyDown" del edit de manera que sólo permita escribir números reales y que al pulsar la tecla "F1" el número real escrito se muestre con 7 dígitos antes del punto y 4 dígitos después del punto (inclusive si el número escrito no tiene tantos dígitos).
3. Cree una aplicación con un ComboBox y programe el evento "onCreate" de la forma de manera que la lista del ComboBox se llene con 100 números reales generados aleatoriamente (con random) comprendidos entre -50000 y +50000. Los números del memo deben tener separadores de miles y 2 dígitos después del punto.
4. Cree una aplicación con un SpeedButton y un StatusBar. Programe el evento "onCreate" de la forma de manera que el StatusBar tenga un solo panel, que el texto del mismo sea su nombre completo y que muestre los Hints de los objetos. Asigne a la forma el Hint "Forma principal" y al BitBtn "SpeedButton de la aplicación".
5. Cree una aplicación con dos BitBtns y un StatusBar. Programe uno de los botones de manera que cierre la aplicación y el otro muestre el mensaje "Aplicación de prueba". En el evento onCreate de la forma haga que el StatusBar tenga un solo panel, que muestre los Hints de la aplicación, que la forma no tenga ningún icono y que los Hints de la forma, y botones sean: "Ventana Principal", "Botón de cierre" y "Botón de mensaje".
6. Cree una aplicación con tres memos y un StatusBar. Programe el evento "onCreate" de la forma de manera que el StatusBar tenga un solo panel, muestre los Hints de los objetos, que los memos no permitan añadir líneas, que al empezar la aplicación estén en blanco y que sus Hints sean "Memo 1", "Memo 2" y "Memo 3" respectivamente. Luego programe el evento "onEnter" de uno de ellos de manera que al ingresar a cualquiera de los memos se seleccione su contenido y se muestre en el panel el Hint respectivo.
7. Cree una aplicación con un RadioGroup y un StatusBar. Programe el evento "onCreate" de la forma de manera que el RadioGroup tenga 4 columnas, las opciones "Opción 1" a "Opción 16", estando seleccionada la primera opción, siendo su Hint "RadioGroup de la aplicación", haga que el StatusBar tenga un solo panel y que muestre las Hint de los objetos de la aplicación. Programe luego el evento "onClick" del RadioGroup de manera que muestre en el panel la opción elegida.
8. Cree una aplicación con un StatusBar y en la misma escriba un módulo circular para calcular la función "J" de Bessel de orden 0 con 16 dígitos de precisión. Luego pruebe el módulo en el evento "onClick" de la

forma para $x=4.6$ y $x=5.7$, mostrando los resultados en el StatusBar.
 $(J_0(x)=1-x^2/2^2+x^4/(2^24^2)-x^6/(2^24^26^2)+x^8/(2^24^26^28^2)-\dots\infty)$

9. Cree una aplicación con un StatusBar y en la misma escriba el código de un módulo circular para calcular el Legendre enésimo de un número real "x". Luego pruebe el módulo en el evento "onKeyDown" de la forma de manera que al pulsar la tecla "F2" muestre, en el StatusBar el valor del Legendre para $n=2$, $x=3.5$ y $n=4$, $x=4.7$. ($L_n(x)=(2-1/n)xL_{n-1}(x)-(1-1/n)L_{n-2}(x)$; $L_0(x)=1$; $L_1(x)=x$)
10. Cree una aplicación con un StatusBar y en la misma escriba el código de un módulo circular que calcule la raíz cuadrada de un número real con 12 dígitos de precisión. Pruebe el módulo en el evento "onKeyDown" de la forma de manera que al pulsar la tecla "F5" muestre en el StatusBar las raíces cuadradas de 0, 1, 4.5 y -8.2. ($x_2=(x_1+x/x_1)/2$)
11. Cree una aplicación con un StatusBar y en la misma escriba el código de un módulo circular, que siguiendo el razonamiento recursivo calcule la potencia entera de un número real. Pruebe luego el módulo en el evento "onKeyDown" de la forma de manera que al pulsar la tecla "F12" muestre en el StatusBar las potencias: 2.3^{40} , 3.45^{10} , 6.534^7 .
12. Cree una aplicación con un StatusBar y en la misma escriba el código de un módulo circular que calcule el exponente de un número real. Pruebe luego el módulo en el evento "onKeyDown" de manera que al pulsar las teclas "Ctrl+Alt+C" muestre en el StatusBar el exponente de 67 y -89. ($e^x=1+x+x^2/2!+x^3/3!+x^4/4!+\dots+\infty$)

13. EL COMANDO GOTO

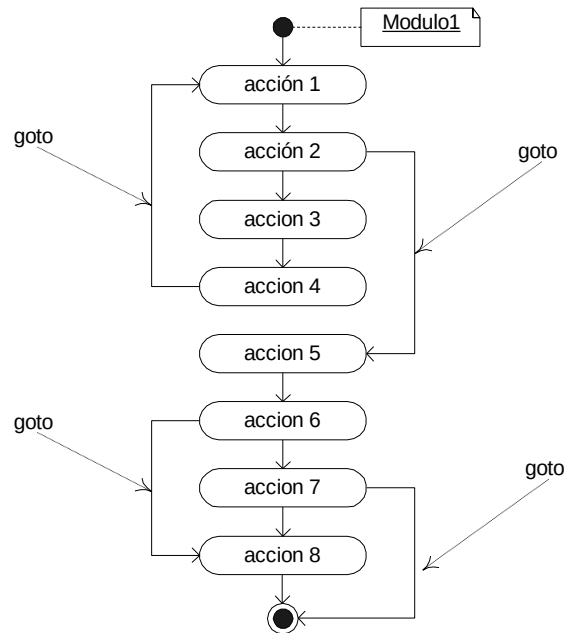
De las herramientas alternativas que estudiamos en esta materia para resolver problemas iterativos, el comando *goto* es sin lugar a dudas el menos estructurado de todos. Es más, como se señaló en la introducción a la programación estructurada, esta metodología surgió como una programación *anti goto*, debido a que el uso exagerado de este comando da lugar a una lógica enredada y confusa (programación tipo espagueti). Es por ello que se recomienda emplear este comando con moderación y sólo cuando su uso de lugar a una lógica más clara y en lo posible más eficiente que con las estructuras estándar o la recursividad.

El comando *goto* permite saltar desde un punto del programa hasta otro ubicado antes o después del punto a partir del cual se salta. En realidad la versatilidad de este comando no sólo le permite resolver problemas iterativos, sino que con él es posible emular (y también romper) cualquiera de las estructuras estándar.

Para evitar el abuso de este comando en los lenguajes estructurados, su alcance está limitado a un módulo, es decir el comando *goto* puede ser empleado para saltar a cualquier parte dentro de un módulo, pero nunca de un módulo a otro.

Por otra parte, si bien el comando *goto* permite saltar al interior de un ciclo iterativo estándar (*mientras, desde o hasta*), esto es algo que no se debe hacer nunca, pues lo único que se consigue con ello es una lógica confusa y un programa que se comporta de manera impredecible.

El comando *goto* no tiene una representación propia en un diagrama de actividades, pues cualquier salto no secuencial (hacia adelante o hacia atrás) puede ser codificado con un salto incondicional *goto*). Así por ejemplo los flujos no secuenciales de la siguiente figura pueden ser interpretados como comandos *goto*:



El objetivo del presente capítulo es que al concluir el mismo estén capacitados para resolver problemas empleando el comando "goto", tanto para lograr la iteración como para emular otras estructuras estándar. Sin embargo se debe recordar que en la práctica sólo se emplea este comando cuando con

ello se consigue una solución más clara que con las estructuras iterativas estándar y/o la recursividad.

El comando *goto* no implica otra forma de razonar, se trata sólo de otra forma más en la que se puede codificar un programa, sin embargo, su flexibilidad permite resolver los problemas siguiendo la lógica natural, pues no es necesario forzar la lógica a ninguna estructura.

Para codificar el comando *goto* en *Pascal* es necesario declarar previamente etiquetas. Las etiquetas son identificadores que marcan los puntos hacia los cuales se realizan los saltos y al igual que las variables pueden estar conformados por letras, números o una combinación de letras y números.

Para declarar las etiquetas se escribe la palabra reservada *Label* (etiqueta). Normalmente las etiquetas se declaran en el mismo segmento donde se declaran las variables y constantes. Por ejemplo la siguiente sentencia declara 5 etiquetas: *1, 300, salto, inicio y fin*:

```
Label 1, 300, salto, inicio, fin;
```

El punto al cual se salta se señala escribiendo el nombre de la etiqueta seguido de dos puntos (:). Por ejemplo en el siguiente segmento de código se marca un lugar de salto con la etiqueta *inicio*:

```
a:= 3*x+sqr(x)-y;
inicio:
  b:= sqrt(x*x+y*y);
```

Para saltar a una etiqueta se escribe el comando *goto* seguido del nombre de la etiqueta. Por ejemplo con la siguiente sentencia se salta hasta el lugar donde se encuentra la etiqueta *inicio*:

```
goto inicio;
```

Por lo tanto el programa continúa después de la etiqueta *inicio*: es decir ejecuta la instrucción "*b:= sqrt(x*x+y*y)*".

13.1. Ejemplos

13.1.1. Cálculo de la raíz cuadrada de un número real

Como primer ejemplo volveremos a resolver el problema de la raíz cuadrada, sólo que ahora escribiremos el código empleando el comando *goto*.

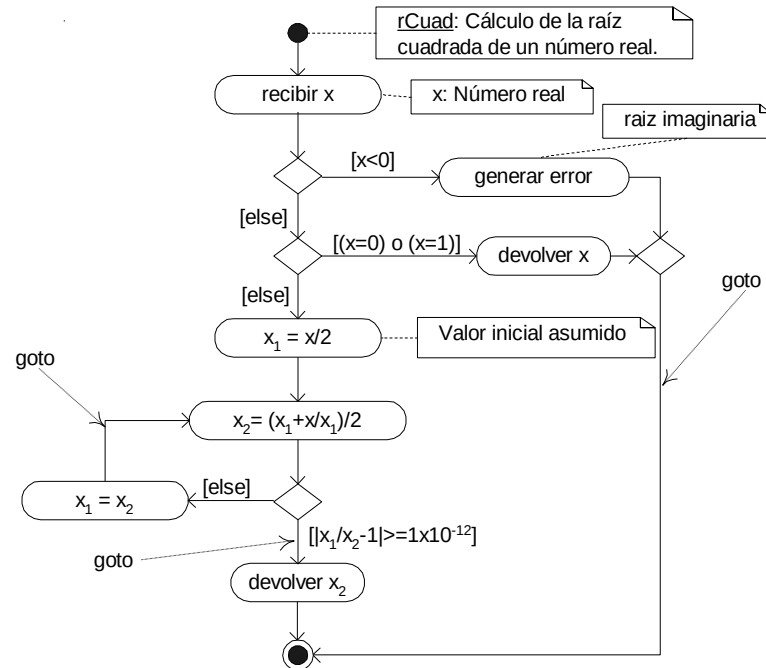
Recordemos que la raíz cuadrada de un número real se calcula con la ecuación de *Newton*:

$$x_2 = \frac{1}{2} \left(x_1 + \frac{x}{x_1} \right)$$

Donde comenzando con un valor inicial asumido para "*x₁*" (generalmente 1), se calculan valores sucesivos de "*x₂*" (empleando siempre el último valor calculado como valor asumido) hasta que los dos últimos valores son iguales en un determinado número de dígitos.

Por supuesto en la lógica es necesario analizar previamente aquellos casos donde no se puede calcular el resultado (porque el dato es incorrecto) o donde no es necesario calcular el resultado (porque ya es conocido). Dicho algoritmo se presenta en el diagrama de actividades de la siguiente página y en el mismo, sólo a manera de ilustración, se han señalado los flujos que pueden ser codificados con el comando "*goto*". Observe que algunos de dichos flujos parecen secuenciales y no saltos *goto*, por ejemplo el flujo que sigue a la condición donde se comparan los dos últimos valores calculados, no obs-

tante, como rompe un ciclo, se trata o de una modificación o de un salto `goto`. Igualmente no es secuencial el flujo que sigue al cambio de variables pues es se trata de un salto hacia atrás y como sabemos, los flujos secuenciales siempre son hacia delante.



Como de costumbre, para codificar estos módulos creamos una nueva aplicación (File -> New Application), una nueva unidad (File -> New Unit) y guardamos la aplicación (File -> Save Project As...) en el directorio "Goto\Raíz cuadrada" con los siguientes nombres: "ufRaizCuadrada" para "Unit1", "uRaizCuadrada" para "Unit2" y "pRaizCuadrada" para "Project1".

Ahora escribimos el código (incluyendo los módulos de lectura y escritura) en la unidad "uRaizCuadrada":

```
unit uRaizCuadrada;
```

```
interface
```

```
  uses StdCtrls, SysUtils, Math, QDialogs;
```

```
  function LeerNumero(m: TMemo): double;
```

```
  function RaizCuadrada(x: double):double;
```

```
  function CalcularRaiz(m: TMemo): double;
```

```
  procedure MostrarRaiz(m: TMemo; r: double);
```

```
implementation
```

```
  function LeerNumero(m: TMemo): double;
```

```
  begin
```

```
    try
```

```
      result:= StrToFloat(m.Text);
```

```
    except
```

```
      on eConvertError do begin
```

```
        ShowMessage('El número está mal escrito');
```

```
        m.SetFocus; m.SelectAll; Abort; end;
```

```
    end;
```

```
  end;
```

```

function RaizCuadrada(x: double):double;
Label fin, ciclo, salir;
var x1, x2: double;
begin
  if x<0 then raise EInvalidArgument.Create('Raiz Imaginária');
  if (x=0) or (x=1) then begin result:= x; goto fin end;
  x1:= x/2;
  ciclo:
    x2:= (x1+x/x1)/2;
    if abs(x1/x2-1)<1E-14 then
      begin result:= x2; goto fin end;
    x1:= x2;
  goto ciclo;
fin:
end;

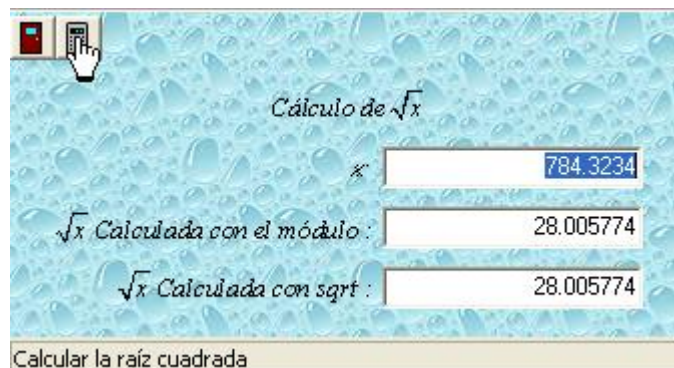
function CalcularRaiz(m: TMemo): double;
begin
  try
    result:= RaizCuadrada(LeerNumero(m));
  except
    on e: EInvalidArgument do begin
      ShowMessage(e.Message); m.SetFocus; m.SelectAll; Abort; end;
  end;
end;

procedure MostrarRaiz(m: TMemo; r: double);
begin
  m.Text:= FormatFloat('#####.000000',r);
end;

end.

```

Ahora elaboramos la interfaz de la aplicación de manera que, en ejecución, se vea aproximadamente como se muestra en la figura:



En esta aplicación existe un componente visible nuevo un *ToolBar*: , que es donde se encuentran los botones de esta aplicación (en la parte superior izquierda). Este componente se encuentra en la pestaña *Win32*, no obstante para su utilización se requiere de al menos un componente más, un *ImageList*: , el cual también se encuentra en la pestaña *Win32*, pero sólo es visible durante el diseño, razón por la cual no aparece en la figura.

Un *ToolBar* es la barra de botones que aparece en la parte superior de casi todas las aplicaciones de *Windows*. Se utiliza para colocar los botones de uso más frecuente de la aplicación (botones de acceso rápido).

Cuando se coloca el *ToolBar* en la forma se acomoda en la parte superior de la ventana. Para añadir en tiempo de diseño botones al *ToolBar* se hace clic en sobre él con el botón derecho del *mouse* y en el menú emergente se elige la opción *New Button*. Deberá llevar a cabo este procedimiento dos veces para añadir los dos botones que tiene la aplicación.

Para añadir botones mediante código se deben crear objetos "tToolButton" y fijar su propiedad "Parent" en el nombre del "ToolBar", por ejemplo el siguiente código añade 10 botones al ToolBar "ToolBar1":

```
for i:=1 to 10 do begin
  tb:= tToolButton.Create(ToolBar1);
  tb.Parent:= ToolBar1;
end;
```

Sin embargo y como podrá observar, los botones añadidos no tienen imágenes ni las propiedades *Glyph* o *Kind* que sirven para añadir imágenes en un *BitBtn* o un *SpeedButton*. Para añadir las imágenes de los botones de un *ToolBar* se requiere de un *ImageList*.

Como su nombre sugiere, un *ImageList* es un componente que almacena una lista de imágenes las cuales pueden ser empleadas luego en otros componentes como los botones de un *ToolBar* o las opciones de un menú. Puesto que su propósito es el de almacenar imágenes pero no mostrarlas, este componente sólo es visible en tiempo de diseño pero no en tiempo de ejecución.

Para añadir imágenes a un *ImageList* en tiempo de diseño, coloque el componente en la forma, haga doble click sobre él y en la ventana que aparece haga click en la opción *Add*. Entonces elija la o las imágenes que desea añadir y haga clic en *Aceptar*.

Por defecto, las imágenes que se añaden al *ImageList* deben ser de 16x16 pixeles. Si son más grandes *ImageList* le informará al respecto y le pedirá autorización para dividir la imagen en dos o más imágenes de manera que se ajusten al tamaño predefinido, esto ocurre por ejemplo cuando se eligen las imágenes de botones de *Delphi*, pues los mismos tienen una tamaño de 16x32 pixeles, de manera que son divididos en dos, el dibujo por defecto (en colores) y el dibujo para los botones deshabilitados (en gris).

Para esta aplicación añada los botones *DoorShut.bmp* y *Calculat.bmp*, aceptando la división de la imagen cuando *ImageList* le pida autorización para ese fin. Luego borre las imágenes en gris (con la opción *Delete*) y deje sólo las dos imágenes en colores.

Finalmente para que las imágenes aparezcan en los botones debe seleccionar *ImageList1* en la propiedad *Images* del *ToolBar1*.

Para cargar imágenes a un *ImageList* en código, se debe emplear la propiedad "Add(Imagen, Máscara)", donde "Imagen" y "Máscara" debe ser objetos de tipo "tBitmap". Cuando la propiedad "Masked" del *ImageList* está en "True", entonces la máscara se combina con la imagen, cuando está en "False" la máscara es ignorada y la imagen aparece sin combinar, lo mismo ocurre si en lugar de la máscara se escribe el valor nulo (Nil). Además, para que la máscara aparezca en el botón, se debe asignar a cada uno de los botones un número de imagen en la propiedad "ImageList". Para borrar una imagen se puede emplear la propiedad "Delete(N° de imagen)". Por ejemplo el siguiente código añade 10 imágenes de botones al *ImageList* "ImageList1", borrando las imágenes en gris, luego añade 10 botones al *ToolBar* "ToolBar1", asignando a cada uno de ellos una de las imágenes del *ImageList*:

```

procedure TForm1.FormCreate(Sender: TObject);
var tb:tToolButton; i:byte; Imagen:tBitmap;
const imagenes:array [1..10] of string=('Abort.bmp',
    'ArrowLR.bmp', 'BulbOn.bmp', 'Calculat.bmp', 'Calendar.bmp',
    'Check.bmp', 'Clock.bmp', 'DoorOpen.bmp', 'DoorShut.bmp',
    'FCabOpen.bmp');
    Path='C:\Archivos de Programa\Archivos Comunes\'+'
    'Borland Shared\Images\Buttons\';
begin
    Imagen:=tBitmap.Create;
    ImageList1.Clear;
    for i:=1 to 10 do begin
        Imagen.LoadFromFile(Path+imagenes[i]);
        ImageList1.Add(Imagen,Nil);
        ImageList1.Delete(i);
    end;
    ToolBar1.Images:=ImageList1;
    for i:=1 to 10 do begin
        tb:=tToolButton.Create(ToolBar1);
        tb.Parent:=ToolBar1;
        tb.ImageIndex:= i-1;
    end;
    Imagen.Free;
end;

```

En realidad existen tres tipos de imágenes que se pueden asignar a los botones de un ToolBar: *DisabledImages*, para los botones inhabilitados; *HotImages* que aparecen cuando se coloca el cursor del mouse sobre el botón e *Images* que son las imágenes que parecen por defecto en los botones. Las imágenes para cada uno de estos tipos deben encontrarse en un *ImageList* diferente, de manera que si quiere asignar los tres tipos de imágenes deberá emplear tres *ImageList*.

Otra propiedad que se ha modificado en el *ToolBar1* es la propiedad *Transparent* que ha sido colocada en *True* para que la barra de botones sea transparente (por esta razón sólo se ven los botones en la parte superior, pero no la barra de botones).

Otra propiedad que se ha modificado en esta aplicación es la propiedad "*BorderStyle*" de la forma. Esta propiedad permite definir el tipo y comportamiento del borde de la ventana de la forma. Las opciones posibles son: "*bsDialog*": Borde de una ventana de diálogo, no permite cambiar el tamaño de la ventana; "*bsSingle*": borde con una línea simple, no permite cambiar el tamaño de la ventana; "*bsNone*", sin borde, no permite cambiar el tamaño de la ventana; "*bsSizeable*", borde estándar, permite cambiar el tamaño de la ventana; "*bsToolWindow*", igual que "*bsSingle*", pero con una letra más pequeña y "*bsSizeToolWin*", igual que "*bsSizeable*" pero con una letra más pequeña. En la presente aplicación se ha colocado el estilo del borde (*BorderStyle*) en *bsNone* y es por esta razón que no aparece el borde y que no es posible cambiar el tamaño de la ventana cuando la aplicación corre (haga la prueba).

Además en esta aplicación se han empleado tres *TImage*, para escribir el símbolo de la raíz cuadrada, que en realidad es un símbolo gráfico y que por lo tanto no puede ser escrito en un *Label*. Para ello los tres textos que tienen el símbolo de la raíz cuadrada han sido escritos en "*MathType*", guardados como imágenes de tipo "*wmf*" (File->Save Copy As...), en el directorio "Goto\Raíz cuadrada" con los nombres "*raizc.wmf*", "*raizc2.wmf*", "*raizc3.wmf*" y cargados en los tres *TImage* de la aplicación.

Las propiedades modificadas en los componentes de esta aplicación son:

Icono de la aplicación: 'Brain Ram 2.ico'. Este icono no aparece en la ventana, pues no tiene bordes, pero aparece en el archivo ejecutable.

Form1: Name = 'fRaizCuadrada'; Caption = 'Raíz Cuadrada'; Height = 217; Width = 345; Position = poScreenCenter; Hint= 'Cálculo de la raíz cuadrada'; BorderStyle = bsNone.

Label1: Caption = 'x: '; Transparent = True; Alignment = taRightJustify.

Image1: Name = 'iRaiz', Picture = 'Raizc.wmf'; Image2: Name = 'iRaizCalculada', Picture = 'Raizc2.wmf'; Image1: Name = 'iRaizConocida', Picture = 'Raizc3.wmf'; Propiedades comunes: Autosize = True; Transparent = True.

Memo1: Name = 'mNumero'; Cursor = crIBeam; Hint = 'Nº cuya raíz se quiere calcular'; Memo2: Name = 'mRaizCalculada'; Cursor = crArrow; ReadOnly = True; TabStop = False; Hint = 'Raiz calculada con el módulo'; Memo3: Name = 'mRaizConocida'; Cursor = crArrow; ReadOnly = True; TabStop = False; Hint = 'Raíz calculada con la función sqrt'; Propiedades comunes: Height = 21; Width = 130, WantReturns = False; Alignment = taRightJustify.

ToolBar: Name = 'tbRaizCuadrada'; Images = ilRaizCuadrada; Transparent = true.

ToolButton1: Name = 'tbSalir'; 'Hint= Salir de la aplicación'; ImageIndex = 0; ToolButton2: Name = 'tbCalcular'; 'Hint= Calcular la raíz cuadrada'; ImageIndex = 1; Propiedades comunes: Cursor = crHandPoint.

StatusBar1: Name = 'sbRaizCuadrada'; AutoHint = True; SimplePanel = True; SimpleText='Cálculo de la raíz cuadrada';

ImageList1: Name = 'ilRaizCuadrada'; Imágenes cargadas: DoorShut.bmp, Calculat.bmp.

Asegúrese también que el orden de los componentes sea (Edit -> Tab Order): mNumero, mRaizCalculada, mRaizConocida, tbRaizCuadrada y sbRaizCuadrada.

Asignamos entonces una imagen a la brocha e inicializamos el texto de los memos en el evento "onCreate" de la forma:

```
procedure TfRaizCuadrada.FormCreate(Sender: TObject);
```

```
begin
```

```
    fRaizCuadrada.Brush.Bitmap := TBitmap.Create;
```

```
    fRaizCuadrada.Brush.Bitmap.LoadFromFile('Gotas de agua.bmp');
```

```
    mNumero.Text:= '0'; mRaizCalculada.Text:= '0'; mRaizConocida.Text:= '0';
```

```
end;
```

Cerramos la aplicación y liberamos el "Bitmap" en el evento "onClick" del ToolButton "tbSalir":

```
procedure TfRaizCuadrada.tbSalirClick(Sender: TObject);
```

```
begin
```

```
    fRaizCuadrada.Brush.Bitmap.Free;
```

```
    fRaizCuadrada.Close;
```

```
end;
```

Damos el foco y seleccionamos el texto del memo "mNumero" en el evento "onActivate" de la forma:

```
procedure TfRaizCuadrada.FormActivate(Sender: TObject);
```

```
begin
```

```
    mNumero.SetFocus;
```

```
    mNumero.SelectAll;
```

```
end;
```

Hacemos que la pista (Hint) de los tres memos se muestre, cuando tienen el foco, programando el evento "onEnter" del memo "mNumero":

```
procedure TfRaizCuadrada.mNumeroEnter(Sender: TObject);
begin
  sbRaizCuadrada.SimpleText:= (Sender as TMemo).Hint;
end;
```

Validamos la introducción de números reales en el evento "onKeyPress" del memo "mNumero", haciendo que la tecla "Enter" actúe como el evento "onClick" del ToolButton "tbCalcular":

```
procedure TfRaizCuadrada.mNumeroKeyPress(Sender: TObject; var Key: Char);
begin
  case Key of
    #8, '.', '0'..'9', '-', 'E', 'e': ;
    #13: tbCalcular.Click;
    else Beep; Abort;
  end;
end;
```

Finalmente damos funcionalidad a la aplicación programando el evento "onClick" del ToolButton "tbCalcular":

```
procedure TfRaizCuadrada.tbCalcularClick(Sender: TObject);
var x: double;
begin
  x:= CalcularRaiz(mNumero);
  MostrarRaiz(mRaizCalculada,x);
  MostrarRaiz(mRaizConocida,Sqrt(LeerNumero(mNumero)));
  mNumero.SetFocus; mNumero.SelectAll;
end;
```

Una vez corregidos los errores la aplicación deberá tener la apariencia y devolver el resultado que se muestra en la figura correspondiente a la interfaz de la aplicación.

13.1.2. Cálculo del seno de un ángulo en radianes

Como segundo ejemplo elaboraremos una aplicación para calcular el seno de un ángulo en radianes. La serie de *Taylor* que permite calcular el seno es:

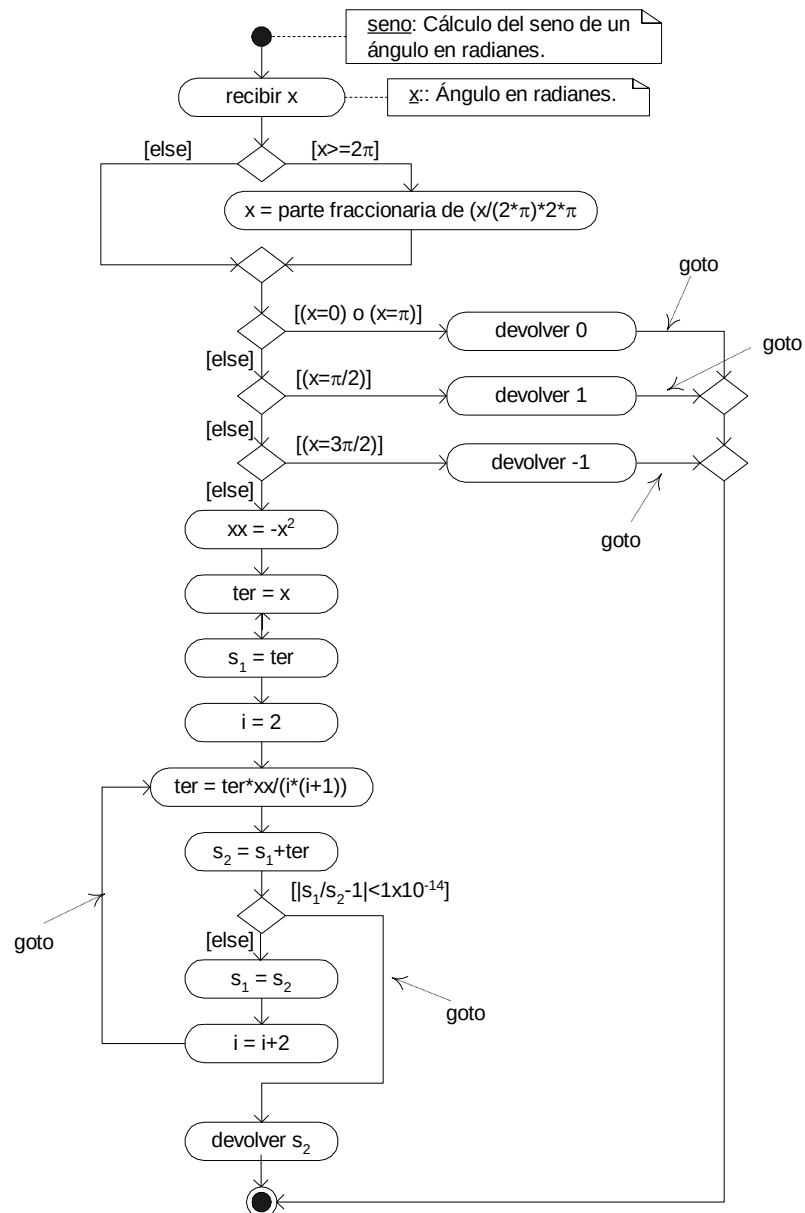
$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots, \quad -\infty < x < \infty$$

Siguiendo la lógica natural, primero se analizan los casos donde o se puede producir un error o no se requiere calcular el resultado pues son conocidos, luego se inicializan las variables y se procede a calcular valores sucesivos de la serie hasta que los dos últimos valores calculados son aproximadamente iguales.

El algoritmo que resuelve el problema se presenta en la siguiente página.

Como de costumbre, para codificar estos módulos creamos una nueva aplicación (File -> New Application), una nueva unidad (File -> New Unit) y guardamos la aplicación (File -> Save Project As...) en el directorio "Goto\Seno" con los siguientes nombres: "ufSeno" para "Unit1", "uSeno" para "Unit2" y "pSeno" para "Project1".

Ahora escribimos el código (incluyendo los módulos de lectura y escritura) en la unidad "uSeno":



```
unit uSeno;
```

```
interface
```

```
uses StdCtrls, ExtCtrls, SysUtils, QDialogs;
```

```
function LeerAngulo(m: TMemo; rg: TRadioGroup): extended;
```

```
function seno(x: extended): extended;
```

```
procedure MostrarSeno(m: TMemo; s: extended);
```

```
implementation
```

```
function LeerAngulo(m: TMemo; rg: TRadioGroup): extended;
```

```
begin
```

```
try
```

```
if rg.ItemIndex=1 then result:= StrToFloat(m.Text)
```

```
else result:= StrToFloat(m.Text)*pi/180;
```

```

except
  on EConvertError do begin
    MessageDlg('El ángulo está mal escrito',mtError,[mbOK],0);
    m.SetFocus; m.SelectAll; Abort; end;
  end;
end;

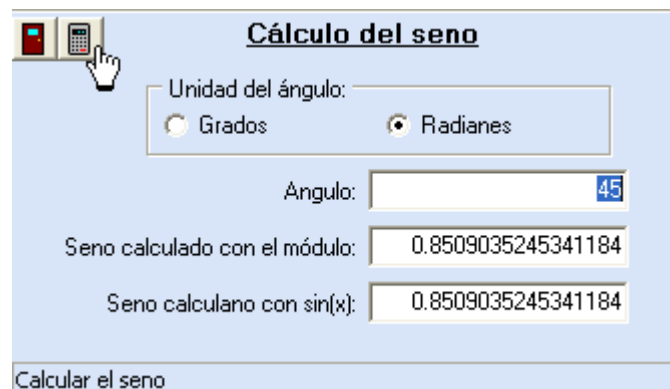
function seno(x: extended): extended;
var xx, ter, s1, s2: extended; i: word;
    label salir, resultado, ciclo;
begin
  if x>=2*pi then x:= frac(x/2/pi)*2*pi;
  if (x=0) or (x=pi) then
    begin result:=0; goto salir; end;
  if (x=pi/2) then
    begin result:= 1; goto salir; end;
  if (x=3*pi/2) then
    begin result:= -1; goto salir; end;
  xx:= -sqr(x);
  ter:= x;
  s1:= ter;
  i:= 2;
  ciclo:
    ter:= ter*xx/i/(succ(i));
    s2:= s1+ter;
    if abs(s1/s2-1)<1E-14 then goto resultado;
    s1:= s2;
    inc(i,2);
  goto ciclo;
  resultado:
    result:= s2;
  salir:
end;

procedure MostrarSeno(m: TMemo; s: extended);
begin
  m.Text:= Format('%18.16g',[s]);
end;

end.

```

Ahora elaboramos la interfaz de la aplicación de manera que, en ejecución, se vea aproximadamente de la siguiente forma:



En esta aplicación no existen componentes nuevos y las propiedades modificadas de los componentes son:

Icono de la aplicación: CDR0M01.ico. Este icono no aparece en la ventana, pues no tiene bordes, pero si en el archivo ejecutable.

Form1: Name = 'fSeno'; Caption = 'Seno de un ángulo en radianes'; Height = 227; Width = 343; Position = poScreenCenter; Hint= 'Seno de un ángulo'; BorderStyle = bsNone; Color = clInactiveCaptionText.

Label1: Caption = 'Angulo: '; Label2: Caption = 'Seno calculado con el módulo: '; Label3: Caption = 'Seno calculado con sin(x): '; Label4: Caption = 'Cálculo del seno'; Font.Size = 10; Font.Style = [fsBold, fsUnderline]; Propiedades comunes: Transparent = True; Alignment = taRightJustify.

Memo1: Name = 'mAngulo'; Cursor = crIBeam; Hint = 'Ángulo'; Memo2: Name = 'mSenoCalculado'; Cursor = crArrow; ReadOnly = True; TabStop = False; Hint = 'Seno calculado con el módulo' Memo3: Name = 'mSenoConocido'; Cursor = crArrow; ReadOnly = True; TabStop = False; Hint = 'Seno calculado con la función Sin'; Propiedades comunes: Height = 21; Width = 130, WantReturns = False; Alignment = taRightJustify.

ToolBar: Name = 'tbSeno'; Images = ilSeno; Transparent = True;

ToolButton1: Name = 'tbSalir'; 'Hint= Salir de la aplicación'; ImageIndex = 0; ToolButton2: Name = 'tbCalcular'; 'Hint= Calcular el seno'; ImageIndex = 1; Propiedades comunes: Cursor = crHandPoint.

StatusBar1: Name = 'sbSeno'; AutoHint = True; SimplePanel = True; SimpleText='Seno de un ángulo'; ParentColor = True;

ImageList1: Name = 'ilSeno'; Imágenes cargadas: DoorShut.bmp, Calculat.bmp.

Asegúrese también que el orden de los componentes sea (Edit -> Tab Order): mAngulo, mSenoCalculado, mSenoConocido, tbRaizCuadrada, rgAngulo y sbSeno.

Inicializamos el texto de los memos en el evento "onCreate" de la forma:

```
procedure TfSeno.FormCreate(Sender: TObject);
begin
    mAngulo.Text:= '0';
    mSenoCalculado.Text:= '0';
    mSenoConocido.Text:= '0';
end;
```

Cerramos la aplicación en el evento "onClick" del ToolButton "tbSalir":

```
procedure TfSeno.tbSalirClick(Sender: TObject);
begin
    fSeno.Close;
end;
```

Damos el foco y seleccionamos el texto del memo "mAngulo" en el evento "onActivate":

```
procedure TfSeno.FormActivate(Sender: TObject);
begin
    mAngulo.SetFocus;
    mAngulo.SelectAll;
end;
```

Validamos la introducción de números reales en el evento "onKeyPress" del memo "mAngulo":

```
procedure TfSeno.mAnguloKeyPress(Sender: TObject; var Key: Char);
begin
    case Key of
```

```

#8,'0'..'9','.', '+', '-', 'e', 'E': ;
#13: tbCalcular.Click;
else
  Beep; Abort;
end;
end;
end;

```

Finalmente damos funcionalidad a la aplicación programando el evento "on-Click" del ToolButton "tbCalcular":

```

procedure TfSeno.tbCalcularClick(Sender: TObject);
var x,s: extended;
begin
  x:= LeerAngulo(mAngulo,rgAngulo);
  s:= Seno(x);
  MostrarSeno(mSenoCalculado,s);
  MostrarSeno(mSenoConocido,sin(x));
  mAngulo.SetFocus; mAngulo.SelectAll;
end;

```

Una vez corregidos los errores la aplicación debe tener la apariencia y devolver el resultado mostrado en la figura correspondiente a la interfaz de la aplicación.

13.1.3. Cálculo de la función "J" de Bessel

Como tercer ejemplo elaboraremos una aplicación para calcular la función "J" de Bessel de orden "n".

La ecuación que permite calcular el valor de esta función es:

$$j_n(x) = \sum_{k=0}^{\infty} \frac{(-1)^k * \left(\frac{x}{2}\right)^{n+2*k}}{k!(n+k)!}$$

Que simplemente es una forma más compacta de escribir una serie. Para facilitar la comprensión y solución de esta serie es conveniente desarrollarla:

$$j_n(x) = \frac{(-1)^0 * \left(\frac{x}{2}\right)^{n+2*0}}{0!(n+0)!} + \frac{(-1)^1 * \left(\frac{x}{2}\right)^{n+2*1}}{1!(n+1)!} + \frac{(-1)^2 * \left(\frac{x}{2}\right)^{n+2*2}}{2!(n+2)!} + \frac{(-1)^3 * \left(\frac{x}{2}\right)^{n+2*3}}{3!(n+3)!} + \dots$$

Finalmente realizando algunas operaciones queda en una forma más familiar:

$$j_n(x) = \frac{\left(\frac{x}{2}\right)^n}{n!} - \frac{\left(\frac{x}{2}\right)^{n+2}}{1!(n+1)!} + \frac{\left(\frac{x}{2}\right)^{n+4}}{2!(n+2)!} - \frac{\left(\frac{x}{2}\right)^{n+6}}{3!(n+3)!} + \dots$$

Que como se puede observar es muy parecida a una serie de Taylor. Por consiguiente la lógica para resolver este problema es esencialmente la misma que se sigue en cualquier serie de Taylor.

El único aspecto adicional que debemos tomar en cuenta en este caso es que el primer término no es un valor simple, razón por la cual debe ser calculado antes de comenzar el proceso iterativo.

Analicemos entonces como se genera el segundo término una vez que se tiene el primero, es decir averigüemos cuáles son las operaciones que debemos realizar para obtener el numerador y el denominador del segundo término:

$$\frac{\left(\frac{x}{2}\right)^n}{n!} \frac{?}{?} = -\frac{\left(\frac{x}{2}\right)^{n+2}}{1!(n+1)!}$$

Vemos que para obtener el numerador del segundo término debemos multiplicar el numerador del primero por $-(x/2)^2$, pues $(x/2)^n \cdot -(x/2)^2$ nos da $-(x/2)^{n+2}$. En el denominador debemos multiplicar el denominador del primer término por 1 y por $(n+1)$, pues $0! \cdot 1$ nos da $1!$ y $n! \cdot (n+1)$ nos da $(n+1)!$, es decir:

$$\frac{\left(\frac{x}{2}\right)^n}{n!} \frac{* \left(-\left(\frac{x}{2}\right)^2\right)}{*(1*(n+1))} = -\frac{\left(\frac{x}{2}\right)^{n+2}}{1!(n+1)!}$$

Veamos ahora como podemos obtener el tercer término a partir del segundo:

$$-\frac{\left(\frac{x}{2}\right)^{n+2}}{1!(n+1)!} \frac{?}{?} = \frac{\left(\frac{x}{2}\right)^{n+4}}{2!(n+2)!}$$

Una vez más para obtener el nuevo numerador debemos multiplicar el anterior por $-(x/2)^2$, pues $-(x/2)^{n+2} \cdot -(x/2)^2$ nos da $+(x/2)^{n+4}$. En el denominador debemos multiplicar por 2 y por $n+2$ pues $1! \cdot 2$ nos da $2!$ y $(n+1)! \cdot (n+2)$ nos da $(n+2)!$, es decir:

$$-\frac{\left(\frac{x}{2}\right)^{n+2}}{1!(n+1)!} \frac{* \left(-\left(\frac{x}{2}\right)^2\right)}{*(2*(n+2))} = \frac{\left(\frac{x}{2}\right)^{n+4}}{2!(n+2)!}$$

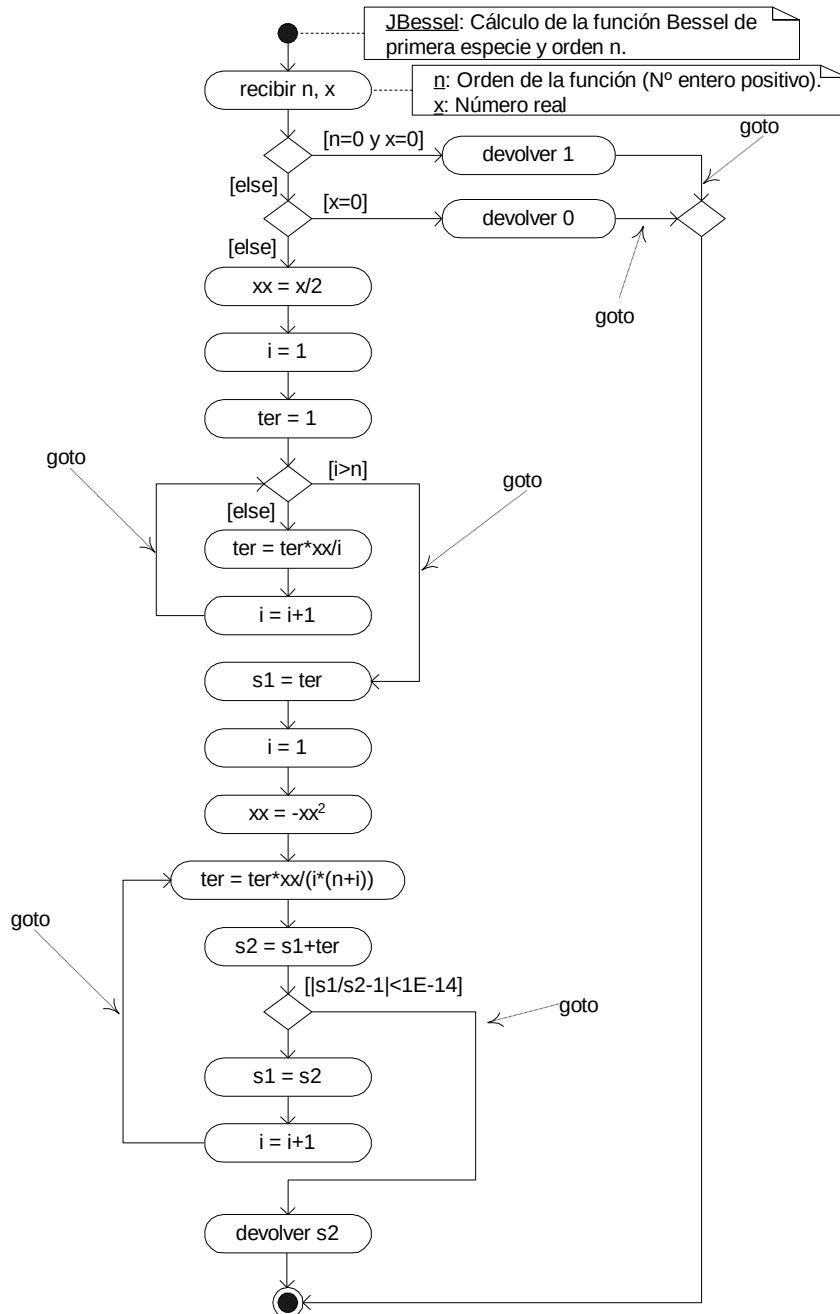
Podemos concluir entonces que para calcular un nuevo término debemos multiplicar el numerador del término anterior por $-(x/2)^2$ y multiplicar el denominador del término anterior por un contador y por "n" más ese contador. Dicho contador debe inicializar en 1 e ir incrementando de 1 en 1. Es decir el factor por el que debemos multiplicar el término anterior es:

$$\frac{-\left(\frac{x}{2}\right)^2}{i*(n+i)} \quad \{i=1,2,3,\dots\}$$

Una vez que sabemos cómo generar un nuevo término en base al anterior, la lógica para resolver el problema es la misma que se sigue al resolver una serie de Taylor: primero se analizan los casos donde se puede producir un error o para los cuales no es necesario realizar el cálculo pues ya se sabe la respuesta, luego se procede a resolver la serie calculando el primer término, inicializando variables y calculando valores sucesivos de la serie hasta que los dos últimos valores calculados son iguales en un determinado número de dígitos.

La lógica antes descrita se presenta en el diagrama de actividades de la siguiente página.

Como de costumbre, para codificar estos módulos creamos una nueva aplicación (File -> New Application), una nueva unidad (File -> New Unit) y guardamos la aplicación (File -> Save Project As...) en el directorio "Goto\Bessel" con los siguientes nombres: "ufBessel" para "Unit1", "uBessel" para "Unit2" y "pBessel" para "Project1".



Ahora escribimos el código (incluyendo los módulos de lectura y escritura) en la unidad "uBessel":

```
unit uBessel;
```

```
interface
```

```
  uses StdCtrls, SysUtils, QDialogs;
```

```
  function LeerOrden(m: TMemo): word;
```

```
  function LeerNumReal(m: TMemo): double;
```

```
  function JBessel(n: word; x: double):double;
```

```
  procedure MostrarBessel(m: TMemo; b: double);
```

implementation

```

function LeerOrden(m: TMemo): word;
begin
  try
    result:= StrToInt(m.Text);
  except
    on EConvertError do begin
      ShowMessage('El orden está mal escrito');
      m.SetFocus; m.SelectAll; Abort; end;
    end;
end;

function LeerNumReal(m: TMemo): double;
begin
  try
    result:= StrToFloat(m.Text);
  except
    on EConvertError do begin
      ShowMessage('El número real está mal escrito');
      m.SetFocus; m.SelectAll; Abort; end;
    end;
end;

function JBessel(n: word; x: double):double;
Label ciclo1, finciclo1, ciclo2, resultado, salir;
var xx,ter,s1,s2: double; i: word;
begin
  if (n=0) and (x=0) then
    begin result:= 1; goto salir; end;
  if x=0 then
    begin result:= 0; goto salir; end;
  xx:= x/2;
  i:= 1;
  ter:= 1;
  ciclo1:
    if i>n then goto finciclo1;
    ter:= ter*xx/i;
    inc(i);
  goto ciclo1;
  finciclo1:
  s1:= ter;
  i:= 1;
  xx:= -sqr(xx);
  ciclo2:
    ter:= ter*xx/i/(n+i);
    s2:= s1+ter;
    if abs(s1/s2-1)<1E-14 then goto resultado;
    s1:= s2;
    inc(i);
  goto ciclo2;
  resultado:
    result:= s2;
    goto salir;
  salir:
end;

procedure MostrarBessel(m: TMemo; b: double);

```

```
begin
    m.Text:= Format('%14.12n',[b]);
end;
```

```
end.
```

Ahora elaboramos la interfaz de la aplicación de manera que, en ejecución, se vea aproximadamente como se muestra en la siguiente figura:



Algunas de las propiedades modificadas en los componentes de esta aplicación son:

Icono de la aplicación: AngelFish.ico. Este icono no aparece en la ventana, pues no tiene bordes, pero si en el archivo ejecutable.

Form1: Name = 'fBessel'; Caption = 'Función J de Bessel'; Height = 216; Width = 332; Position = poScreenCenter; Hint= 'Función "j" de Bessel'; BorderStyle = bsNone.

Label1: Caption = 'Orden:'; Label2: Caption = 'Número real:'; Label3: Caption = 'Valor de la función:'; Label4: Caption = 'Función "j" de Bessel'; Font.Size = 12; Font.Style = [fsBold, fsItalic]; Propiedades comunes: Transparent = True; Alignment = taRightJustify.

Memo1: Name = 'mOrden'; Cursor = crIBeam; Hint = 'Orden de la función'; Memo2: Name = 'mNumReal'; Cursor = crIBeam; Hint = 'Valor para el cual se calcula la función'; Memo3: Name = 'mBessel'; Cursor = crArrow; ReadOnly = True; TabStop = False; Hint = 'Valor de la función "j" de Bessel'; Propiedades comunes: Height = 21; Width = 130, WantReturns = False; Alignment = taRightJustify.

ToolBar: Name = 'tbBessel'; Images = ilBessel; Transparent = True;

ToolButton1: Name = 'tbSalir'; 'Hint= Salir de la aplicación'; ImageIndex = 0; ToolButton2: Name = 'tbCalcular'; 'Hint= Calcular la función Bessel'; ImageIndex = 1; Propiedades comunes: Cursor = crHandPoint.

StatusBar1: Name = 'sbBessel'; AutoHint = True; SimplePanel = True; SimpleText='Función "J" de Bessel'; ParentColor = True;

ImageList1: Name = 'ilBessel'; Imágenes cargadas: DoorShut.bmp,.bmp, CompPC2.bmp.

Asegúrese también que el orden de los componentes sea (Edit -> Tab Order): mAngulo, mSenoCalculado, mSenoConocido, tbRaizCuadrada, rgAngulo y sbSeno.

Ahora cambiamos el color de la forma, el estilo de la brocha e inicializamos el texto de los memos en el evento "onCreate" de la forma:

```
procedure TfBessel.FormCreate(Sender: TObject);
begin
```

```

    fBessel.Color:= clActiveBorder;
    fBessel.Brush.Style:= bsFdiagonal;
    mOrden.Text:= '0';
    mNumReal.Text:= '0';
    mBessel.Text:= '1';
end;

    Terminamos la aplicación, programando el evento "onClick" del ToolButton
    "tbSalir":

procedure TfBessel.tbSalirClick(Sender: TObject);
begin
    fBessel.Close;
end;

    Damos el foco al memo "mOrden" y seleccionamos su texto en el evento
    "onActivate" de la forma:

procedure TfBessel.FormActivate(Sender: TObject);
begin
    mOrden.SetFocus;
    mOrden.SelectAll;
end;

    Validamos la introducción de números enteros en el evento "onKeyPress"
    del memo "mOrden":

procedure TfBessel.mOrdenKeyPress(Sender: TObject; var Key: Char);
begin
    case Key of
        #8, '0'..'9': ;
        #13: tbCalcular.Click;
        else
            Beep; Abort;
    end;
end;

    Validamos la introducción de números reales en el evento "onKeyPress" del
    memo "mNumReal":

procedure TfBessel.mNumRealKeyPress(Sender: TObject; var Key: Char);
begin
    case Key of
        #8, '.', '0'..'9', '-', 'E', 'e': ;
        #13: tbCalcular.Click;
        else
            Beep; Abort;
    end;
end;

    Hacemos que se seleccione el texto de los memos cuando reciben el foco y
    que se muestre su "Hint" en la barra de estado, programando el evento "onEn-
    ter" del memo "mOrden":

procedure TfBessel.mOrdenEnter(Sender: TObject);
begin
    with sender as TMemo do
        begin
            SelectAll;
            sbBessel.SimpleText:= Hint;
        end;
end;

```

Finalmente damos funcionalidad a la aplicación programando el evento "on-Click" del ToolButton "tbCalcular":

```
procedure TfBessel.tbCalcularClick(Sender: TObject);
var n: word; x,b: double;
begin
  n:= LeerOrden(mOrden);
  x:= LeerNumReal(mNumReal);
  b:= JBessel(n,x);
  MostrarBessel(mBessel,b);
  mOrden.SetFocus; mOrden.SelectAll;
end;
```

Como puede observar, en el evento "OnEnter" del memo "mOrden", se ha empleado el operador **with**, el cual ya hemos estudiado y empleado previamente, sin embargo, para refrescar la memoria recordemos que el operador "With" nos permite acceder a las propiedades o métodos de un objeto sin necesidad de escribir por delante el nombre del objeto.

Así por ejemplo en el siguiente segmento de código se emplea el operador "with" para evitar escribir en cada una de las instrucciones el nombre del objeto "SatusBar1":

```
with StatusBar1 do
begin
  SimplePanel:= True;
  SimpleText:= 'Panel de ayuda';
  AutoHint:= True;
  Color:= clYellow;
  Cursor:= crHandPoint;
  Font.Color:= clBlue;
  Font.Size:= 10;
  Font.Style:= Font.Style+[fsBold,fsItalic];
end;
```

Sin el operador "with" las mismas instrucciones se escriben de la siguiente manera:

```
StatusBar1.SimplePanel:= True;
StatusBar1.SimpleText:= 'Panel de ayuda';
StatusBar1.AutoHint:= True;
StatusBar1.Color:= clYellow;
StatusBar1.Cursor:= crHandPoint;
StatusBar1.Font.Color:= clBlue;
StatusBar1.Font.Size:= 10;
StatusBar1.Font.Style:= StatusBar1.Font.Style+[fsBold,fsItalic];
```

En esta aplicación se ha empleado el operador "With" por dos motivos: a) para evitar escribir el nombre del objeto (Sender) en cada una de las instrucciones y b) para instruir, una sola vez, que el objeto *Sender* sea tratado como (as) un *Memo*. Sin el operador "With", a más de escribir dos veces el nombre del objeto, se tiene que dar dos veces la instrucción de tratar al objeto *Sender* como (as) un memo, tal como se muestra en el siguiente código (que funcionalmente es el equivalente al presentado anteriormente).

```
procedure TfBessel.mOrdenEnter(Sender: TObject);
begin
  (Sender as TMemor).SelectAll;
  sbBessel.SimpleText:= (Sender as TMemor).Hint;
end;
```

Recordemos que el operador "With" sólo tiene este propósito, es decir el reducir la cantidad de código que se escribe, pero que no afecta a la eficiencia del programa, es significa que un programa no es más o menos eficiente por emplear el operador "With", en consecuencia el uso o no de este operador depende de las preferencias del programador y sobre todo de cómo resulte más comprensible el código final.

Para comprender mejor cómo funciona el comando *goto*, se recomienda hacer correr paso a paso los programas, teniendo a la vista los valores de las variables locales (View->Debug Windows->Local Variables).

Una vez corregidos los errores, la aplicación deberá tener la apariencia y devolver el resultado que se muestra en la figura correspondiente a la interfaz de la aplicación.

13.1.4. Ejercicios

1. Cree una aplicación con un *ToolBar* y un *ImageList* y programe el evento "onCreate" de la misma de manera que el *ImageList* tenga 7 imágenes (eliminando las imágenes en gris) y el *ToolBar* tenga 7 botones con las imágenes del *ImageList*.
2. Cree una aplicación con un *ToolBar* y un *ImageList* y programe el evento *onCreate* de la forma de manera que el *ToolBar* tenga dos botones con 2 imágenes en color del *ImageList*. Que el *Hint* del primer botón sea "Salir de la aplicación" y del segundo "Mostrar Mensaje". Programe también el evento *onClick* del primer botón de manera que cierre la aplicación y el evento *onClick* del segundo botón de manera que muestre en una ventana de mensaje su nombre completo.
3. Cree una aplicación con un *ToolBar* y dos *ImageList* y programe el evento *onCreate* de la forma de manera que el *ToolBar* tenga 5 botones con 5 imágenes en color del primer *ImageList* y 5 imágenes en gris del segundo *ImageList*. Programe también el evento *onClick* de todos los botones de manera que al hacer click en los mismos el botón se inhabilite (mostrando la imagen en gris) y al volver a hacer click se vuelva a habilitar (mostrando la imagen en color).
4. Cree una aplicación con un *ToolBar*, un *ImageList* y un *StatusBar*. En el evento *onCreate* de la forma haga que el *ToolBar* tenga 1 botón con una imagen y el *Hint* "Salir de la aplicación", que el *StatusBar* tenga un solo panel y que en el mismo se muestren los *Hint* de la aplicación y que la ventana de la aplicación no tenga bordes. Programe también el evento *onClick* del *ToolButton* de manera que al hacer click sobre el mismo la aplicación se cierre.
5. Cree una aplicación con un *ToolBar*, un *ImageList* y un *StatusBar*. En el evento *onCreate* de la forma haga que el *ToolBar* tenga 4 botones con 4 imágenes, que el *StatusBar* tenga un solo panel y en el mismo esté su nombre completo y que la ventana no permita modificar, maximizar o minimizar su tamaño. Programe también el evento *onClick* del *ToolBar* de manera que al hacer click sobre él, cambien las imágenes de los botones rotando las mismas hacia la derecha.
6. Cree una aplicación con un *ToolBar*, dos *ImageList* y un *StatusBar*. En el evento *onCreate* de la forma haga que el *ToolBar* sea transparente y tenga 5 botones con 5 imágenes en color y 5 en gris, que el fondo de la forma sea la figura "gotas de agua.bmp", que el *StatusBar* tenga un solo panel y que no se pueda cambiar el tamaño de la ventana pero que sí

pueda ser maximizada o minimizada. Programe el evento *onClick* de los 4 primeros botones de manera que al hacer click sobre los mismos el botón se inhabilite (quedando la imagen en gris) y muestre en el *StatusBar* "El ToolButton x ha sido inhabilitado" y que al volver a hacer click sobre el mismo el botón se habilite (quedando la imagen en color) y muestre en el *StatusBar* el mensaje "El ToolButton x ha sido habilitado". Programe también el evento *onClick* del último botón de manera que al hacer click en el mismo la aplicación se cierre.

7. Cree una aplicación y en la misma escriba un módulo que empleando el comando Goto, calcule el valor de la siguiente productoria $\prod (x+i)$ $\{i=2,3,6,...n\}$. Pruebe la aplicación el evento *onKeyDown* de la forma (tecla F10) con $x=5.43$, $n=10$; $x=2.32$, $n=40$.
8. Cree una aplicación y en la misma escriba un módulo que empleando el comando Goto calcule el coseno de un ángulo en radianes con 15 dígitos de precisión. Recuerde reducir los ángulos mayores a 2π a su equivalente comprendido entre 0 y 2π y tomar en cuenta todos los posibles casos. Pruebe la aplicación en el evento "onClick" de la forma con $x=0$, π , $\pi/2$, 54, 567 y 4323.323. $(\cos(x)=1-x^2/2!+x^4/4!-x^6/6!+...\infty)$.
9. Cree una aplicación y en la misma escriba un módulo que empleando el comando Goto calcule el Legendre enésimo de un número real "x". En la resolución del problema debe tomar en cuenta todos los posibles casos. Pruebe la aplicación en el evento *onContextPopUp* de la aplicación con $n=3$, $x=6.5$; $n=6$, $x=4.2$; $n=0$, $x=4.1$; $n=1$, $x=3.2$. $(L_n(x)=(2-1/n)xL_{n-1}(x)-(1-1/n)L_{n-2}(x); L_0(x)=1; L_1(x)=x)$
10. Cree una aplicación y en la misma escriba un módulo que empleando el comando Goto calcule una de las soluciones de la ecuación cúbica: $ax^3+bx^2+cx+d=0$. Tanto el valor inicial (x_1) como los coeficientes deben ser parámetros (datos) del módulo y x_1 debe tener un valor por defecto igual a 1.1. En la resolución del problema debe contar el número de iteraciones efectuadas y si las mismas llegan a 50 suspender el proceso iterativo generando un error con el mensaje "Valor inicial asumido erróneo". Pruebe el módulo en el evento *onKeyDown* (tecla F1) con $x^3+2x^2+3x+4=0$ y $3x^3-2x^2+x-8=0$. $(x_2=(2ax_1^3+bx_1^2-d)/(3ax_1^2+2bx_1+c))$
11. Cree una aplicación y en la misma escriba un módulo que empleando Goto invierta los dígitos de un número entero. Pruebe el módulo en el evento *onKeyDown* (teclas Ctrl+Shift+C) con 123456789 y -123456789.
12. Cree una aplicación y en la misma escriba un módulo que empleando Goto integre funciones con una incógnita con la regla del trapecio: $\int f(x)dx = (h/2)(f(x_1)+2*\sum f(x_i)\{i=2,3,4..n+f(x_{n+1})\}; x_1=a; x_{n+1}=b; h=(b-a)/n; a=\text{límite inferior}; b=\text{límite superior}; n=\text{número de segmentos}$. Luego pruebe la función calculando el valor de la integral: $\int ((2x^{4.1}+3x^3-2x^{1.45})^{1/2}/(6.2x^{3.7}-2x^{3.7}+4)^{1/3})dx$ {límites: $a=1.2$; $b=8.45$, con $n=100$ en el evento *onKeyDown* (teclas Alt+Shift+I).

14. MATRICES ESTÁTICAS

14.1. Vectores estáticos

En este capítulo comenzaremos el estudio de los datos estructurados. *Un dato es estructurado si está conformado por dos o más datos de tipo simple o estructurado.*

Un *vector* es un dato estructurado porque está conformado por dos o más datos del mismo tipo.

En este y en los siguientes capítulos emplearemos indistintamente los términos *vector*, *matriz* o *array*, aunque el último término corresponde a la denominación inglesa de matrices o arreglos, sin embargo, dado que prácticamente todos los lenguajes de programación están en inglés es de utilidad acostumbrarse a este término.

Un *vector* es un caso especial de una *matriz* (*array*) pues básicamente se trata de una matriz con una sola fila (o una sola columna). Esta es la razón por la cual los vectores son conocidos también como *matrices unidimensionales* (o *arrays unidimensionales*).

14.1.1. Declaración de vectores

En *Pascal* se puede trabajar con vectores de diferentes formas: *como vectores estáticos, parámetros abiertos, vectores dinámicos y punteros.*

En este capítulo trabajaremos con *vectores estáticos*, los cuales existen en *Pascal* desde sus inicios. Los *vectores estáticos* se caracterizan por tener un *número fijo de elementos*, es decir que una vez declarados no es posible incrementar ni disminuir el número de elementos. Como veremos luego, esta característica constituye una desventaja, sobre todo cuando se elaboran módulos de propósito general. Esta limitante ha hecho que los *vectores estáticos* vayan siendo relegados en las aplicaciones a favor de los *vectores dinámicos* y los *punteros*.

Los *vectores estáticos* se declaran de acuerdo a la siguiente sintaxis:

Nombre: **array** [*límite inferior* .. *límite superior*] **of** *tipo de dato*;

Donde *Nombre* es el nombre del vector o una lista de nombres separadas con comas; *límite inferior* y *límite superior* son los índices del primer y último elemento respectivamente; *tipo de dato* es el tipo de dato de cada uno de los elementos del vector.

Así en el siguiente ejemplo se declaran dos vectores (*a* y *b*), donde el índice del primer elemento es 1 y el del último 50 siendo todos los elementos de tipo *doble*:

```
var a,b: array [1..50] of double;
```

Como en este ejemplo el primer índice es 1 y el último 50 el vector tiene 50 elementos (los elementos 1, 2, 3, hasta 50).

En *Pascal*, los índices de un vector deben ser siempre valores ordinales, que como sabemos son aquellos que tienen un *orden* predefinido (como los números enteros, letras o tipos definidos por el usuario). Otra condición que deben cumplir los índices es que el límite inferior debe ser siempre menor al límite superior. De acuerdo a lo anterior las siguientes declaraciones son correctas:

```
var x,y: array ['a'..'z'] of integer;  
    u,v,w: array [-20..40] of extended;
```

```

r: array [lunes..viernes] of string;
z: array ['D'..'R'] of double;

```

Mientras que las siguientes declaraciones son incorrectas (encuentre las razones para ello):

```

var c,d,e: array [2.34..4.67] of Longint;
    f,g: array [20..1] of char;
    h,i,l: array ['lunes'..'viernes'] of double;
    m: array ['Z'..'L'] of extended;

```

Si bien las variables pueden ser declaradas directamente, tal como se ha mostrado en los anteriores ejemplos, es recomendable declarar primero un tipo de dato y luego declarar variables de ese tipo. Esto es algo que se debe hacer inevitablemente cuando se pasan vectores como parámetros a procedimientos o funciones.

Así, sería recomendable hacer las anteriores declaraciones (las declaraciones correctas) de la siguiente manera:

```

type tx = array ['a'..'z'] of integer;
      tu = array [-20..40] of extended;
      tr = array [lunes..viernes] of string;
      tz = array ['D'..'R'] of double;
var x,y: tx;
    u,v,w: tu;
    r: tr;
    z: tz;

```

Puesto que resolvemos los problemas en módulos (procedimientos o funciones), emplearemos casi exclusivamente esta forma, es decir declaramos tipos de datos antes de declarar variables o parámetros tipo array.

14.1.2. Vectores constantes

Si los valores de un vector no varían, como cuando corresponden a una tabla o lista de constantes, es más eficiente declararlos como constantes, de acuerdo a la siguiente sintaxis (donde "x" es el nombre de la variable):

```
const x: array [li..ls] of tipo de dato = (valores separados por comas);
```

Así por ejemplo para almacenar los datos de la siguiente tabla en el vector "tabla".

Nº	Valor
1	6.9302
2	8.3212
3	10.2122
4	12.8943
5	20.9092
6	25.3821
7	27.3321
8	30.5432
9	33.8752
10	37.1782

Escribimos lo siguiente:

```
var tabla : array [1..10] of real = (6.9302,8.3212,10.2122,12.8943,20.9092,
                                     25.3821,27.3321,30.5432,33.8752,37.1782);
```

Por supuesto podemos declarar también un tipo y luego la constante de ese tipo:

```

type ttabla = array [1..10] of real;
var tabla : ttabla = (6.9302,8.3212,10.2122,12.8943,20.9092,25.3821,
                      27.3321,30.5432,33.8752,37.1782);

```

14.1.3. Acceso a los elementos de un vector

Como ya se dijo, los vectores son matrices unidimensionales, por lo tanto para acceder a sus elementos sólo se requiere una *dimensión* o *índice*. Para trabajar con cualquier elemento de un vector se escribe el nombre del vector seguido del índice entre corchetes. Así por ejemplo `b[3]`, es el elemento correspondiente al índice 3 del vector "b", mientras que `c['d']` es el elemento correspondiente al índice 'd' del vector "c".

Con excepción de su notación cada uno de los elementos de un vector se trata exactamente igual que cualquier variable simple del mismo tipo. Por ejemplo, con los siguientes vectores (que son de tipo real):

```

var a,b: array [1..50] of double;

```

Podemos escribir las siguientes expresiones:

```

a[2] := sqrt(b[1])+3*exp(b[3]);
a[3] := (2*sqr(a[1])+3*a[1]*ln(a[1]))/a[2];
a[50]:= sin(a[5])+cos(a[4]-a[3])+sqr(a[7]);

```

Donde, como se puede observar, se trabaja con los elementos del vector igual que con cualquier variable de tipo real.

Cuando se trabaja con vectores lo más frecuente es realizar la misma operación con cada uno de los elementos del vector, entonces en lugar de trabajar con valores literales para los índices (como se ha hecho en los anteriores ejemplos) se utilizan variables ordinales (del mismo tipo que el índice). Por ejemplo para calcular la siguiente sumatoria:

$$s = \sum_{i=1}^n x_i^2$$

Se emplea una estructura *For*, donde la variable de control "i", que va desde 1 hasta n, se convierte en el índice de los elementos:

```

s:=0;
for i:=1 to n do s:= s+sqr(x[i]);

```

La memoria que ocupa un vector se calcula multiplicando su número de elementos por el tamaño que ocupa cada elemento (es decir el tamaño del tipo de dato). Así por ejemplo el siguiente vector:

```

var a: array [1..50] of double;

```

Ocupa $50 \times 8 = 400$ bytes de memoria.

14.2. Matrices Estáticas

Denominaremos matrices a aquellos arrays que tengan 2 o más dimensiones. Aun cuando en Pascal las matrices pueden ser declaradas con un número prácticamente ilimitado de dimensiones, la administración de las misma se hace muy difícil y confusa cuando son más de dos, por esta razón en la práctica las matrices generalmente tienen dos dimensiones y muy rara vez 3.

Un array multidimensional se declara separando los límites de cada uno de los índices con comas. Por ejemplo la siguiente instrucción declara una matriz de tipo LongInt con cuatro dimensiones:

```

type

```

```
tmulti = array [1..3,2..4,0..3,10..20] of LongInt;
```

Las matrices también pueden ser declaradas como un vector de vectores, así la anterior declaración puede ser escrita como:

```
type
```

```
  tMulti = array [1..3] of array [2..4] of array [0..3] of array [10..20] of  
          LongInt;
```

Para acceder a los elementos de una matriz, se escribe su nombre y se especifica la posición del elemento con los índices separados con comas. Por ejemplo en el siguiente segmento de código se asignan dos números enteros a dos elementos de una matriz de tipo *tMulti*.

```
var x : tMulti;
```

```
begin
```

```
  x[1,2,3,15] := 12345;
```

```
  x[1,3,1,17] := 56789;
```

Es posible también acceder a los elementos de una matriz tratándola como un vector de vectores. En ese caso el índice correspondiente a cada dimensión va entre corchetes, así el anterior segmento de código puede ser escrito también de la siguiente forma:

```
var x : tMulti;
```

```
begin
```

```
  x[1][2][3][15] := 12345;
```

```
  x[1][3][1][17] := 56789;
```

14.2.1. Matrices constantes

Al igual que sucede con los vectores, en ocasiones es útil almacenar en una matriz los datos correspondientes a una tabla, con dos o más columnas, en forma de una constante. Para ello se procede igual que con los vectores constantes, sólo que en este caso cada fila se escribe entre paréntesis y las diferentes filas se separan con comas.

Así por ejemplo, para guardar los datos de la siguiente tabla:

Año	Ganancia
1990	45784.32
1992	60223.43
1994	90214.65
1995	120328.95
1996	140324.88
1997	160324.32

Escribimos:

```
type ttabla = array [1..6,1..2] of real;
```

```
const
```

```
  tabla : ttabla = ((1990,45784.32),(1992,60223.43),(1994,90214.65),  
                    (1995,120328.95),(1996, 140324.88),(1997,160325.32));
```

Si en lugar de trabajar con constantes empleamos variables, el proceso resulta mucho más moroso, porque en ese caso los valores deben ser asignados elemento por elemento, como se muestra a continuación:

```
var
```

```
  tabla : ttabla;
```

```
...
```

```
tabla[1,1]:=1990; tabla[1,2]:=45784.32;
```

```
tabla[2,1]:=1992; tabla[2,2]:=60223.43;
```

```

tabla[3,1]:=1994; tabla[3,2]:=90214.65;
tabla[4,1]:=1995; tabla[4,2]:=120328.95;
tabla[5,1]:=1996; tabla[5,2]:=140324.88;
tabla[6,1]:=1997; tabla[6,2]:=160324.32;

```

Al igual que sucede con los vectores, la mayoría de las operaciones con matrices se llevan a cabo en ciclos "For" (uno por cada dimensión), así por ejemplo el siguiente segmento de código suma todos los elementos de la matriz tabla:

```

s:= 0;
for i:=1 to 6 do
  for j:=1 to 2 Do
    s:= s+tabla[i,j];

```

14.3. Ejemplos

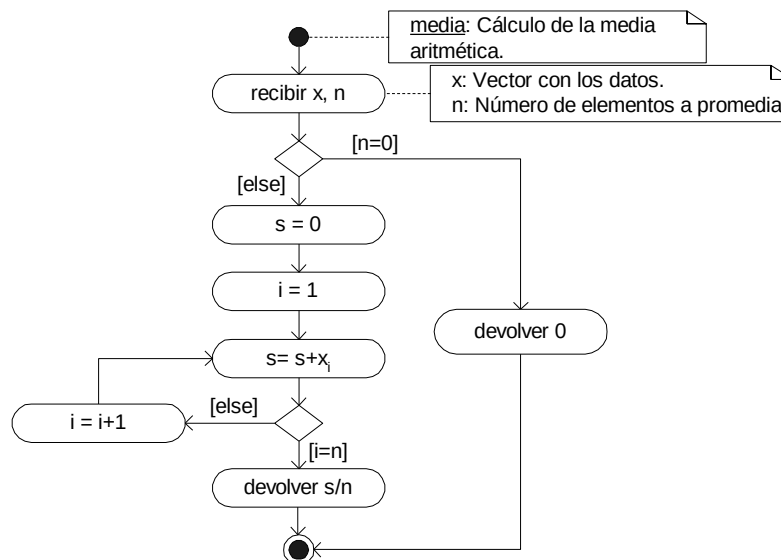
14.3.1. Cálculo de la media aritmética

Como primer ejemplo elaboraremos una aplicación para calcular la media aritmética de "n" números reales.

La ecuación de definición de la media aritmética es la siguiente:

$$\bar{x} = \frac{\sum_{i=1}^n x_i}{n}$$

Donde "x" es el vector que contiene los datos y "n" es el número de elementos a promediar. La lógica para resolver este problema es muy sencilla, el único problema radica en el cálculo de la sumatoria y la misma consiste simplemente de un ciclo "for" que va desde 1 hasta "n", donde en cada repetición del ciclo se suma el valor de x_i . El algoritmo, en forma de diagrama de actividades, es el siguiente:



Al momento de codificar el algoritmo nos vemos en la necesidad de fijar de antemano el número de elementos del vector (porque como ya se dijo, los vectores estáticos tienen un número fijo de elementos). Sin embargo, al ser un módulo de carácter general, no se sabe de antemano cuántos elementos deberán ser promediados, por lo tanto no nos queda otra alternativa que fijar un número de elementos que sea adecuado para la mayoría de los casos.

En este ejemplo fijaremos el número de elementos en 500, de esta manera estaremos en condiciones de calcular el promedio de hasta 500 valores. Ello significa que si mandamos a promediar 10 valores quedarán sin utilizar 490 elementos, por el contrario si queremos calcular el promedio de 510 elementos, ello no será posible porque sólo se ha reservado espacio para 500 elementos. Este es un problema que siempre se presenta con los vectores estáticos y esa es la razón por la cual actualmente no son muy utilizados.

Lo anterior también es la razón por la cual en el módulo se reciben dos datos: el vector con los valores a promediar "x" y el número de valores a promediar "n". Porque aún cuando se ha reservado espacio para cierto número de elementos (500 en nuestro caso), muy rara vez querremos promediar ese número de valores.

Cuando se trabaja con vectores es conveniente contar con módulos que generen un determinado número de elementos al azar, porque al momento de probar un módulo resulta muy moroso introducir manualmente un número elevado de elementos (por ejemplo 500).

Para generar un número al azar se emplea la función "random", la cual devuelve un número real comprendido entre 0 y 1. Para generar en base a este número otro comprendido entre ciertos límites se emplea la siguiente fórmula:

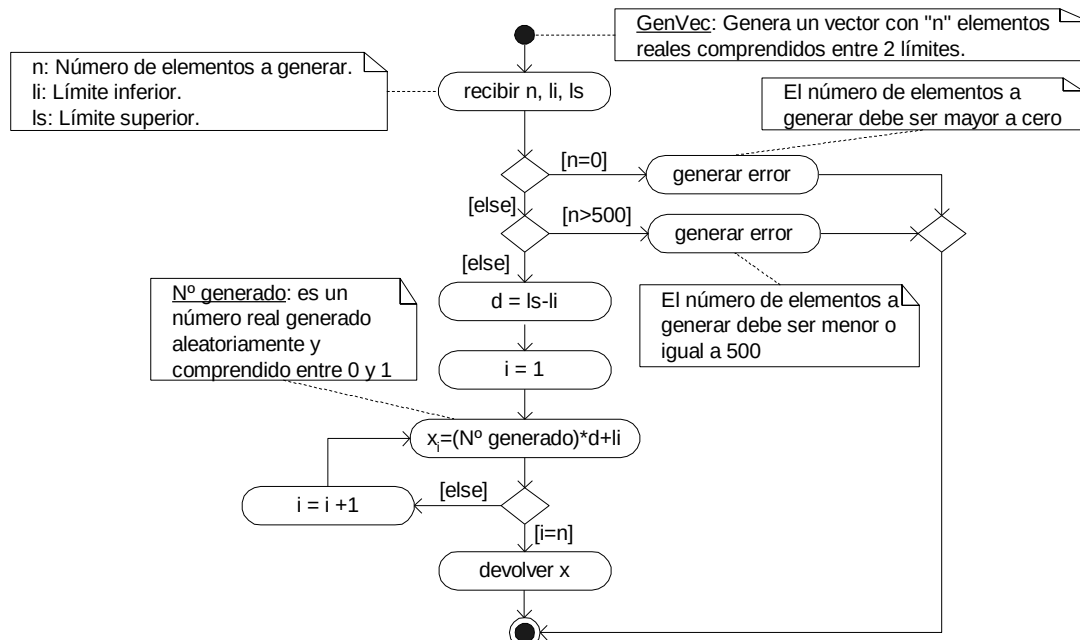
$$N^{\circ} \text{ aleatorio comprendido entre } li \text{ y } ls = (N^{\circ} \text{ aleatorio generado}) * (ls - li) + li$$

Donde "li" es el límite inferior y "ls" el límite superior. Así por ejemplo para generar un número aleatorio comprendidos entre 10 y 50, la fórmula es:

$$N^{\circ} \text{ aleatorio comprendido entre } 10 \text{ y } 50 = (N^{\circ} \text{ aleatorio generado}) * (50 - 10) + 10$$

De esa manera si el número generado es 0, se convierte en 10 ($0 * 40 + 10 = 10$) que corresponde el límite inferior, por el contrario si el número generado es 1 se convierte en 50 ($1 * 40 + 10 = 50$) que corresponde al límite superior.

El algoritmo del módulo que genera un vector con "n" números aleatorios comprendidos entre dos límites dados es el siguiente:



Como de costumbre, para codificar estos módulos creamos una nueva aplicación (File -> New Application), una nueva unidad (File -> New Unit) y guardamos la aplicación (File -> Save Project As...) en el directorio "Vectores y matrices estáticas\Promedio" con los siguientes nombres: "uMedia" para "Unit1", "ufMedia" para "Unit2" y "pMedia" para "Project1".

En la unidad "uMedia" escribimos el código de los módulos "Promedio" y "GenerarVector", así como el código de los módulos de lectura y escritura:

```
unit uMedia;
```

```
interface
```

```
  uses StdCtrls, SysUtils, Math, QDialogs;
```

```
  type tv500 = array [1..500] of extended;
```

```
  function LeerVector(m: TMemo): tv500;
```

```
  function Promedio(const x: tv500; n: word): extended;
```

```
  function GenerarVector(n: word; li, ls: extended): tv500;
```

```
  procedure MostrarVector(x: tv500; n: word; m: TMemo);
```

```
  procedure MostrarPromedio(p: extended; m: TMemo);
```

```
implementation
```

```
  function LeerVector(m: TMemo): tv500;
```

```
  var i, l: word; x: tv500; p, n: integer;
```

```
  begin
```

```
    try
```

```
      //Supresión de las líneas en blanco al final del memo
```

```
      n:= m.Lines.Count-1;
```

```
      while (m.Lines[n]='') and (n>=0) do begin
```

```
        m.Lines.Delete(n); dec(n); end;
```

```
      //Si el memo no tiene datos se genera un error
```

```
      inc(n); //Número de filas con datos
```

```
      if n=0 then raise EInvalidArgument.Create(
```

```
        'No se ha introducido ningún dato para calcular el promedio');
```

```
      //Lectura de los valores del memo
```

```
      p:= 0; //p: Posición del cursor en el memo
```

```
      for i:=1 to n do begin
```

```
        p:=p+Length(m.Lines[i-1])+2;
```

```
        x[i]:=StrToFloat(m.Lines[i-1]); end;
```

```
      result:= x;
```

```
    except
```

```
      on EConvertError do begin
```

```
        MessageDlg('El número está mal escrito', mtError, [mbOk], 0);
```

```
        l:= Length(m.Lines[i-1]); //Longitud del número mal escrito
```

```
        m.SelStart:= p-1-2; //Posición del primer carácter mal escrito
```

```
        m.SelLength:= 1; //Selección del número mal escrito
```

```
        m.SetFocus; Abort; end;
```

```
      on e:EInvalidArgument do begin
```

```
        MessageDlg(e.Message, mtError, [mbOk], 0);
```

```
        m.SetFocus; m.SelStart:= 0; Abort; end;
```

```
    end;
```

```
  end;
```

```
  function Promedio(const x: tv500; n: word): extended;
```

```
  var s: extended; i: word;
```

```
  begin
```

```

    if n=0 then begin result:=0; exit; end;
    s:=0;
    for i:=1 to n do s:=s+x[i];
    result:=s/n;
end;

function GenerarVector(n: word; li,ls:extended): tv500;
var i: word; d: extended;
begin
    if n=0 then raise EInvalidArgument.Create(
        'El número de elementos a generar debe ser mayor a cero');
    if n>500 then raise EInvalidArgument.Create(
        'El número de elementos a generar debe er menor o igual a 500');
    randomize; d:=ls-li;
    for i:=1 to n do result[i]:=random*d+li;
end;

procedure MostrarVector(x: tv500; n: word; m: TMemo);
var i:word;
begin
    try
        m.Lines.BeginUpdate;//Se deshabilita la actualización del memo
        m.Lines.Clear;
        for i:=1 to n do m.Lines.Append(FloatToStr(x[i]));
    finally
        m.Lines.EndUpdate;//Se rehabilita la actualización del memo
    end;
end;

procedure MostrarPromedio(p: extended; m:TMemo);
begin
    m.Text:= Format('%16.4f',[p]);
end;

end.

```

Observe que en el módulo "Promedio", el vector se recibe en forma de un parámetro constante para evitar modificar inadvertidamente los datos originales.

En el módulo "GenerarVector" aparece una llamada al procedimiento *randomize*, algo que no ha sido considerado en el algoritmo. Esta llamada no aparece en el algoritmo porque el procedimiento *randomize* es específico de Pascal y puede no existir o no ser necesario en otros lenguajes. Su función es la de asegurar que se genere una serie diferente de números aleatorios con la función *random*. Si no se hace una llamada al procedimiento *randomize*, la función *random* siempre generará la misma serie de números aleatorios, por lo tanto los números generados no serán realmente aleatorios sino pseudoaleatorios (algo que puede ser de utilidad en algunos problemas). Por el contrario si se llama a la función *randomize* (ante de llamar a la función *random*) los números generados serán realmente aleatorios.

Sin embargo, no se debe llamar al procedimiento *randomize* cada vez que se utiliza la función *random*, pues si se procede así se consigue justamente el efecto contrario, es decir se genera el mismo número. Esto sucede porque el procedimiento *randomize* inicializa una nueva serie tomando como base la hora del sistema (en segundos) y con la velocidad actual de las computadoras pueden generarse cientos o miles de números aleatorios antes de que pase un segundo. Por ejemplo si en el módulo *GenVec* se coloca la llamada a *randomize*

en el interior del ciclo *For*, todos los números generados resultan iguales (haga la prueba).

En la función "*LeerVector*" se han empleado algunos métodos de la propiedad "*Lines*", que como ya vimos anteriormente es un objeto de tipo "*Tstrings*", dichas propiedades son "*Count*", que devuelve el número de líneas existentes en el memo; "*Clear*", que borra todas las líneas del memo y "*Delete*", que elimina el número de línea especificado. Se ha empleado también la propiedad "*Lines*", que permite modificar o leer el texto de cualquiera de las líneas del memo.

Se han empleado también las propiedades "*SelStart*" y "*SelLength*". "*SelStart*" nos permite fijar o leer el inicio del texto seleccionado y "*SelLength*" nos permite fijar o leer el número de caracteres seleccionados (siendo su valor cero cuando no está seleccionado nada). Es necesario emplear estas propiedades debido a que el memo no cuenta con un método que nos permita seleccionar directamente una fila.

En el procedimiento "*MostrarVector*" se han empleado también otros métodos de la propiedad *Lines*: "*Append*", que nos permite añadir líneas de texto al final del memo y los métodos "*BeginUpdate*" y "*EndUpdate*".

El método "*BeginUpdate*" cancela temporalmente la actualización visual del componente que contiene la propiedad (en nuestro caso el memo), es decir impide que cada modificación de la propiedad "*Lines*" sea vista en el componente, algo que es útil en este caso porque los cambios ocurren tan rápidamente que, si son visibles, da la impresión de que algo está fallando porque los cambios se ven como un titileo de la pantalla. Por otra parte al impedir que el memo se actualice con cada nuevo cambio se consume mucho menos tiempo, algo que siempre es conveniente en un programa.

El método "*EndUpdate*" vuelve a activar la actualización visual del componente, no obstante, si después de llamar al método *BeginUpdate* se produce un error y no se ejecuta el método *EndUpdate*, el memo queda congelado y no se actualiza más ni puede ser modificado manualmente, por esta razón, para que inclusive en caso de error el memo quede activo, la llamada a *EndUpdate* se la realiza entre ***finally*** y ***end***

Como recordarán, la estructura ***try...finally...end***, se emplea para asegurar que el código que se encuentra entre ***finally*** y ***end*** sea ejecutado siempre, incluso cuando se produce un error. Recordemos también que el código normal se escribe entre ***try*** y ***finally***.

Ahora elaboremos la interfaz de la aplicación de manera que, en ejecución, se vea aproximadamente como se muestra en la figura de la siguiente página.

En esta aplicación no existen componentes nuevos, el componente central, donde se introducen los datos a promediar es un memo, sólo que en este caso su tamaño ha sido ampliado a 305 pixeles de alto por 155 de ancho y se ha mantenido su propiedad *WantReturns* en *True* para que sea posible añadir más de una línea. Adicionalmente se ha asignado al memo una barra de desplazamiento vertical cambiando la propiedad *ScrollBar*s a *ssVertical*. Como este memo permite la introducción de varias líneas trabajamos con la propiedad *Lines*, como ya hemos visto anteriormente, en lugar de la propiedad *Text*.

Las propiedades modificadas en los componentes de esta Interfaz son:

Icono de la aplicación: "*Blue unicorn.ico*". Este icono no aparece en la ventana, pues no tiene bordes, pero aparece en el archivo ejecutable.

Form1: Name = "fMedia; Caption = 'Cálculo de la media aritmética'; Color = clInactiveCaptionText; Height =513; Width = 296; Position = poScreen-

Center; Hint = Media aritmética; BorderStyle = bsNone.

Datos a promediar:

- 33.026744753588
- 44.9083061609417
- 33.2938291667961
- 43.8380513852462
- 45.7823351421393
- 29.8203008435667
- 40.0962819042616
- 21.9240398192778
- 20.8875416708179
- 20.2558584045619
- 44.4877099222504
- 33.2794590992853
- 46.3064246461727
- 38.8111592829227
- 45.8284099469893
- 39.1027655126527
- 42.5161287863739
- 42.7962434757501
- 37.7901639952324
- 45.8607792994007
- 44.8903060727753
- 36.7771247960627
- 44.1918106633238

Calcular Promedio

Promedio: 36.3496

Generar Datos Borrar Datos

Calcular promedio

Label1: Caption = 'Datos a promedia:'; Label2: Caption = 'Promedio:';
Propiedades comunes: Transparent = True.

Memo1: Name = 'mDatos'; Cursor = crIBeam; Hint = 'Datos a promediar';
 WantReturns = True; ScrollBars = ssVertical; Height = 305; Width = 155;
Memo2: Name = 'mPromedio'; Cursor = crNo; ReadOnly = True; TabStop = False;
 Hint = 'Promedio calculado'; Height = 21; Width = 130, WantReturns = False;
Propiedades comunes: Alignment = taRightJustify; Font.Color = clBlue.

ToolBar: Name = 'tbSalir'; Images = ilMedia; Transparent = True.

ToolButton1: Name = 'tbSalir'; Hint = 'Salir de la aplicación', ImageIndex = 0;
ToolButton2: Name = 'tbPromedio'; Hint = 'Calcular el promedio'; ImageIndex = 1;
ToolButton3: Name = 'tbGenerar'; Hint = 'Generar datos'; Glyph = Retry.bmp;
ToolButton4: Name = 'tbBorrar'; Hint = 'Borrar Datos'; ImageIndex = 3;
Propiedades comunes: Cursor = crHandPoint.

BitBtn1: Name = 'bbPromedio'; Default = True; Hint = 'Calcular el promedio';
 Glyph = Sum.bmp; BitBtn2: Name = 'bbGenerar'; Hint = 'Generar datos';
 Glyph = Retry.bmp; BitBtn3: Name = 'bbBorrar'; Hint = 'Borrar Datos';
 Glyph = Clear.bmp; Propiedades comunes: Cursor = crHandPoint.

StatusBar1: Name = 'sbMedia'; AutoHint = True; SimplePanel = True; SimpleText = 'Media aritmética'; Hint = 'Media aritmética'.

ImgeList1: Name = 'ilMedia'; Imágenes cargadas: DoorOpen.bmp, Sum.bmp, Retry.bmp, Clear.bmp (sólo imágenes en color).

Orden de los componentes: mDatos, bbPromedio, mPromedio, bbGenerar, bbBorrar, sbMedia, tbMedia.

Para generar los datos es necesario introducir los límites inferior, superior y el número de elementos a generar. En esta aplicación, la introducción de estos datos se hará en una nueva ventana (una nueva forma). Para añadir una una nueva forma en una aplicación se elige la opción *New Form* del menú *File* (*File* -> *New Form*).

En ejecución la nueva ventana tendrá la apariencia que se muestra en la siguiente figura:

Como se puede observar en esta ventana tampoco existen componentes nuevos. Algunas de las propiedades modificadas en los componentes de esta ventana son:

Form1: Name = 'fGenerar'; Color = clInactiveCaptionText; Position = poScreenCenter; BorderStyle: bsDialog; Hint = 'Generar Datos'; Height = 187; Width = 280.

Memo1: Name = 'mLimInf'; Hint = 'Límite inferior'; Memo2: Name = 'mLimSup'; Hint = 'Límite superior'; Memo3: Name = 'mNumEle'; Hint = 'Número de elementos'; Propiedades comunes: Height = 21; Width = 130; WantReturns = False; Alignment = taRightJustify; Font.Color = clBlue; Cursor = crIBeam.

Label1: Caption = 'Límite inferior: '; Label2: Caption = 'Límite superior: '; Label3: Caption = 'Nº de elementos'; Propiedades comunes: Transparent = True; Alignment = taRightJustify.

BitBtn1: Name = 'bbAceptar'; Default = True; Kind = kbOK; Caption = '&Aceptar'; Hint = 'Generar datos'; BitBtn2: Name = 'bbCancelar'; Cancel = True; Kind = bkCancel; Caption = '&Cancelar'; Hint = 'Cancelar la generación de datos'; Propiedades comunes: Cursor = crHandPoint.

Esta ventana debe aparecer cuando se hace clic en el BitBtn *bbGenerar* (o en el ToolButton *tbGenerar*). Existen dos métodos que se pueden emplear para hacer aparecer una ventana en una aplicación: los métodos *Show* y *ShowModal*.

La diferencia entre estos dos métodos es la siguiente: el método *Show* muestra la ventana y permite trabajar tanto en la ventana mostrada como en otras ventanas de la aplicación, por el contrario, el método *ShowModal* muestra la ventana y sólo permite trabajar en dicha ventana hasta que se cierra. Se emplea el método *ShowModal* cuando la aplicación tiene una lógica secuencia y para continuar se debe esperar los resultados o acciones de la ventana mostrada.

Para cerrar una ventana que se ha hecho visible con *ShowModal*, se debe asignar algún valor entero, diferente de cero, a su propiedad *ModalResult*, el valor asignado a esta propiedad es el resultado devuelto por el método *ShowModal* (que es una función).

Dicho valor puede ser asignado directamente escribiendo código en algún evento de la aplicación o alternativamente a través de la propiedad *ModalResult* de los *BitBtns*. Cuando se elige un tipo de *BitBtn*, mediante la propiedad "Kind", se asigna también un valor a la propiedad *ModalResult*, así cuando se elige el tipo *mbOk*, la propiedad *ModalResult* toma el valor *mrOk*, cuando se elige el tipo *mbCancel*, la propiedad *ModalResult* toma el valor *mrCancel*; con *bkAbort*, *ModalResult* toma el valor *mrAbort*; con *bkRetry* *mrRetry*, con *mbIgnore* *mrIgnore*; con *mbYes* *mrYes* y con *mbNo* *mrNo*.

El valor por defecto de la propiedad *ModalResult* es *mrNone* (0) y mientras se mantiene este valor la ventana no puede ser cerrada.

Puesto que en esta aplicación los elementos se generan o no en función a los datos y acciones de la ventana *fGenerar*, la aplicación tiene una lógica secuencial y en consecuencia la ventana debe ser mostrada con *ShowModal*.

Dado que algunos de los módulos de esta ventana se emplearán desde la ventana principal (*fMedia*), dichos módulos deben ser escritos en el sector público de la clase "tfGenerar":

type

```
TfGenerar = class(TForm)
  Label1: TLabel;
  Label2: TLabel;
  Label3: TLabel;
  bbCancelar: TBitBtn;
  bbAceptar: TBitBtn;
  mLimInf: TMemo;
  mLimSup: TMemo;
  mNumEle: TMemo;
  procedure FormShow(Sender: TObject);
  procedure FormCreate(Sender: TObject);
  procedure FormActivate(Sender: TObject);
  procedure mLimInfKeyPress(Sender: TObject; var Key: Char);
  procedure mNumEleKeyPress(Sender: TObject; var Key: Char);
  procedure mLimSupExit(Sender: TObject);
  procedure mLimInfExit(Sender: TObject);
  procedure mLimInfEnter(Sender: TObject);
  procedure mNumEleExit(Sender: TObject);
private
  { Private declarations }
public
  function LeerLimiteInferior: extended;
  function LeerLimiteSuperior: extended;
  function LeerNumeroElementos: word;
end;
```

El código de estos tres módulos se escribe en el sector de implementación, precediendo el nombre de los mismos con el nombre de la clase: *tfGenerar*:

```
function tfGenerar.LeerLimiteInferior: extended;
begin
  try
    result:= StrToFloat(mLimInf.Text);
  except
```

```
    on EConvertError do begin ShowMessage('Límite inferiro mal escrito');
        mLimInf.SetFocus; mLimInf.SelectAll; Abort; end;
    end;
end;
```

```
function tfGenerar.LeerLimiteSuperior: extended;
begin
    try
        result:= StrToFloat(mLimSup.Text);
    except
        on EConvertError do begin ShowMessage('Límite superior mal escrito');
            mLimSup.SetFocus; mLimSup.SelectAll; Abort; end;
        end;
    end;
end;
```

```
function tfGenerar.LeerNumeroElementos: word;
begin
    try
        result:= StrToInt(mNumEle.Text);
    except
        on EConvertError do begin ShowMessage('N° de elementos mal escrito');
            mNumEle.SetFocus; mNumEle.SelectAll; Abort; end;
        end;
    end;
end;
```

Ahora escribimos los eventos de esta forma comenzando con el evento "on-Create", donde damos valores iniciales a los memos:

```
procedure TfGenerar.FormCreate(Sender: TObject);
begin
    mLimInf.Text:= '20';
    mLimSup.Text:= '50';
    mNumEle.Text:= '100';
end;
```

Hacemos que el memo "mLimInf" reciba el foco en el evento "onShow", el cual ocurre cada vez que la forma se hace visible:

```
procedure TfGenerar.FormShow(Sender: TObject);
begin
    mLimInf.SetFocus; mLimInf.SelectAll;
end;
```

Hacemos que el texto de los memos se seleccione al ingresar a los misos programando el evento "onEnter" del memo "mLimInf" y asignando también dicho evento a los memos "mLimSup" y "mNumEle":

```
procedure TfGenerar.mLimInfEnter(Sender: TObject);
begin
    (Sender as TMemo).SelectAll;
end;
```

Validamos la introducción de números reales a los memos "mLimInf" y "mLimSup" programando el evento "onKeyPress" del memo "mLimInf" y asignando también dicho evento al memo "mLimSup".

```
procedure TfGenerar.mLimInfKeyPress(Sender: TObject; var Key: Char);
begin
    case Key of
        '0'..'9', '.', '+', '-', #8;;
    else
        Beep; Abort;
    end;
end;
```

```
end;
end;
```

Validamos la introducción de números enteros al memo "mNumEle", programando su evento "onKeyPress":

```
procedure TfGenerar.mNumEleKeyPress(Sender: TObject; var Key: Char);
begin
  case Key of
    '0'..'9', #8:
      else
        Beep; Abort;
  end;
end;
```

Verificamos que el número introducido en el memo "mLimInf" sea menor al número existente en el memo "mLimSup", programando el evento "onExit" del memo "mLimInf" (este evento ocurre cuando el componente pierde el foco). Si el límite inferior es mayor al superior asignamos un número una unidad mayor al límite superior:

```
procedure TfGenerar.mLimInfExit(Sender: TObject);
var x: extended;
begin
  x:= LeerLimiteInferior;
  if x>LeerLimiteSuperior then mLimSup.Text:=FloatToStr(x+1);
end;
```

Verificamos que el número introducido en el memo "mLimSup" sea mayor al número existente en el memo "mLimInf", programando su evento "onExit":

```
procedure TfGenerar.mLimSupExit(Sender: TObject);
begin
  try
    if LeerLimiteSuperior<LeerLimiteInferior then
      raise EInvalidArgument.Create(
        'El límite superior no puede ser menor al límite inferior');
  except
    on e: EInvalidArgument do begin
      MessageDlg(e.Message, mtError, [mbOk], 0);
      mLimSup.SetFocus; mLimSup.SelectAll; Abort; end;
  end;
end;
```

Finalmente verificamos que el número de elementos introducido en el memo "mNumEle" sea mayor a cero programando su evento "onExit":

```
procedure TfGenerar.mNumEleExit(Sender: TObject);
begin
  try
    if LeerNumeroElementos=0 then raise EInvalidArgument.Create(
      'El número de elementos a generar debe ser mayor a cero');
  except
    on e: EConvertError do begin
      MessageDlg(e.Message, mtError, [mbOK], 0);
      mNumEle.SetFocus; mNumEle.SelectAll; Abort; end;
  end;
end;
```

No programamos los eventos de ninguno de los BitBtns, porque los mismos al ser de las clases "mbOK" y "mbCancel", asigna automáticamente los valores

"mrOk" y "mrCancel", respectivamente, a la propiedad "ModalResult" de la forma, con lo que la misma (al ser modal) se cierra y devuelve ese valor.

Ahora programamos los eventos de la forma "fMedia" recordando incluir (File->Use unit...) las unidades "uMedia" y "ufGenerar".

```
uses ufGenerar, uMedia;
```

Comenzamos con el evento "onActivate" en el cual simplemente simulamos el evento "onClick" del botón "bbBorrar" que es donde se borran los memos y se da el foco al memo "mDatos":

```
procedure TfMedia.FormActivate(Sender: TObject);
begin
    bbBorrar.Click;
end;
```

El evento "bbBorrar", al cual se llama desde el evento "onActivate" es el siguiente:

```
procedure TfMedia.bbBorrarClick(Sender: TObject);
begin
    mPromedio.Clear;
    mDatos.Lines.Clear;
    mDatos.SetFocus;
end;
```

Validamos la introducción de datos al memo "mDatos", cuidando de permitir la tecla "Enter" (#13) para posibilitar la adición de líneas en el memo:

```
procedure TfMedia.mDatosKeyPress(Sender: TObject; var Key: Char);
begin
    case Key of
        '0'..'9', '.', '+', '-', 'E', 'e', #13, #8:
            else
                Beep; Abort;
    end;
end;
```

Programamos el evento "onClick" del BitBtn "bbGenerar" de manera que muestre la ventana "fGenerar" (en forma modal) y si se sale de la misma haciendo click en el BitBtn "bbAceptar" (OK), se genera y muestra el vector en el memo "mDatos", caso contrario (si se sale de la ventana "fGenerar" haciendo click en el botón "bbCancelar") no se hace nada:

```
procedure TfMedia.bbGenerarClick(Sender: TObject);
var x: tv500; li,ls: extended; ne: word;
begin
    if fGenerar.ShowModal = mrOk then begin
        li:= fGenerar.LeerLimiteInferior;
        ls:= fGenerar.LeerLimiteSuperior;
        ne:= fGenerar.LeerNumeroElementos;
        x:= GenerarVector(ne,li,ls);
        MostrarVector(x,ne,mDatos);
    end;
end;
```

Programamos el evento "onClick" del BitBtn "bbPromedio" de manera que calcule y muestre el promedio de los datos existentes en el memo "mDatos":

```
procedure TfMedia.bbPromedioClick(Sender: TObject);
var x: tv500; ne: word; media: extended;
begin
    x:= LeerVector(mDatos);
```

```

ne:= mDatos.Lines.Count;
media:= Promedio(x,ne);
MostrarPromedio(media,mPromedio);
end;

```

Programamos también el evento "onClick" del ToolButton "tbSalir" de manera que al hacer click en el mismo se salga de la aplicación:

```

procedure TfMedia.tbSalirClick(Sender: TObject);
begin
    fMedia.Close;
end;

```

En los eventos "onClick" de los otros tres ToolButtons, simplemente se llama al evento "onClick" de los BitBtns respectivos.

```

procedure TfMedia.tbPromedioClick(Sender: TObject);
begin
    bbPromedio.Click;
end;

```

```

procedure TfMedia.tbGenerarClick(Sender: TObject);
begin
    bbGenerar.Click;
end;

```

```

procedure TfMedia.tbBorrarClick(Sender: TObject);
begin
    bbBorrar.Click;
end;

```

Una vez escrito el código y corregidos los errores, la aplicación deberá tener la apariencia mostrada en la figura correspondiente a la interfaz de la aplicación. Debe probar también que la aplicación responda correctamente a todos los posibles casos de error, introduciendo números y valores erróneos.

14.3.2. Cálculo de la desviación estándar

Como segundo ejemplo elaboraremos una aplicación para calcular la desviación estándar de una serie de datos.

La desviación estándar se calcula con la siguiente expresión:

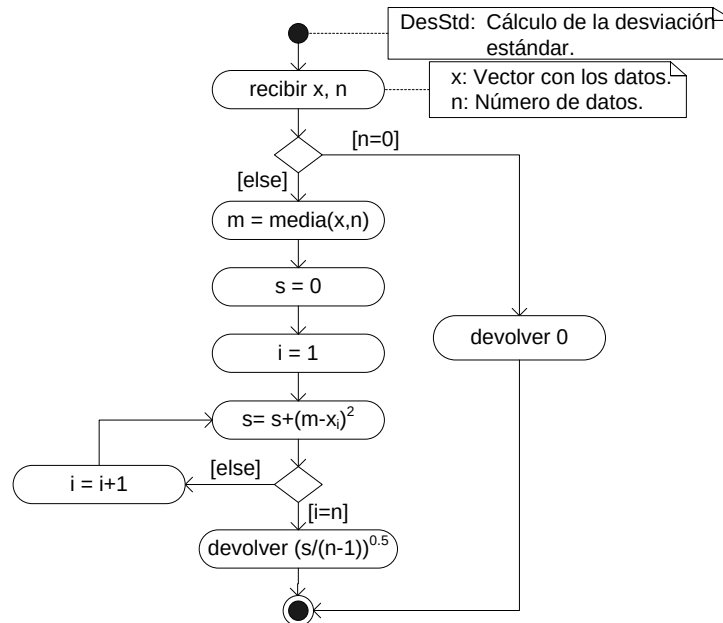
$$\text{desviación estándar} = \sqrt{\frac{\sum_{i=1}^n (\bar{x} - x_i)^2}{n-1}}$$

Donde " $\bar{x}$ " es el promedio de los datos y " n " es el número de datos.

Como se puede observar, en este caso es necesario calcular previamente el promedio, algo que ya se ha hecho en el ejemplo anterior, para aprovechar la solución del anterior ejemplo se puede copiar el módulo en la unidad donde se codifique el módulo para la desviación estándar o alternativamente se puede incluir la unidad donde se encuentra dicho módulo (la unidad uMedia) en el módulo donde se codifique la desviación estándar.

Aparte de la anterior consideración, la lógica para resolver este problema es muy sencilla, simplemente se trata de un ciclo "for" que se repite desde 1 hasta " n " y donde en cada repetición del ciclo se suma el valor de $(\bar{x} - x_i)^2$.

El algoritmo que resuelve este problema es el siguiente:



Para su implementación se elaborará una aplicación muy similar a la del ejemplo anterior, con la diferencia de que en lugar del promedio se calcula la desviación estándar.

Lo más eficiente en este caso es hacer una copia de la carpeta "Promedio" (que es la carpeta donde se guardó la anterior aplicación), cambiarle el nombre a "Desviación Estándar", abrir el proyecto en Delphi y guardar (File -> Save As...) las unidades y programas con los siguientes nombres: "ufDesviacionEstandar" para "ufMedia" y "pDesviacionEstandar" para "pMedia", dejando las unidades "uMedia" y "fGenerar" sin modificar. Posteriormente, en la carpeta "Desviación Estándar" se deben borrar todos los archivos cuyos nombres comienzan con "ufMedia", y "pMedia".

Escribimos el código del módulo "DesviacionEstandar", así como el módulo "MostrarDesviacion" en la unidad "uDesviacionEstandar" (File -> New Unit; y luego: File -> Save As... -> uDesviacionStandar), no necesitamos volver a escribir el módulo "Promedio" y el módulo "GenerarVector", porque los mismos se encuentra en la unidad "uMedia" y dicha unidad será incorporada en la unidad "uDesviacionEstandar":

```
unit uDesviacionEstandar;
```

```
interface
```

```
  uses uMedia, StdCtrls, SysUtils;
```

```
  function DesviacionEstandar(const x:tv500; n:word):extended;
```

```
  procedure MostrarDesviacion(ds: extended; m: TMemo);
```

```
implementation
```

```
  function DesviacionEstandar(const x:tv500; n:word):extended;
```

```
  var m,s: extended; i: word;
```

```
  begin
```

```
    if n=0 then begin result:=0; exit; end;
```

```
    m:= Promedio(x,n);
```

```

s:=0;
for i:=1 to n do s:=s+sqr(m-x[i]);
result:= sqrt(s/n-1);
end;

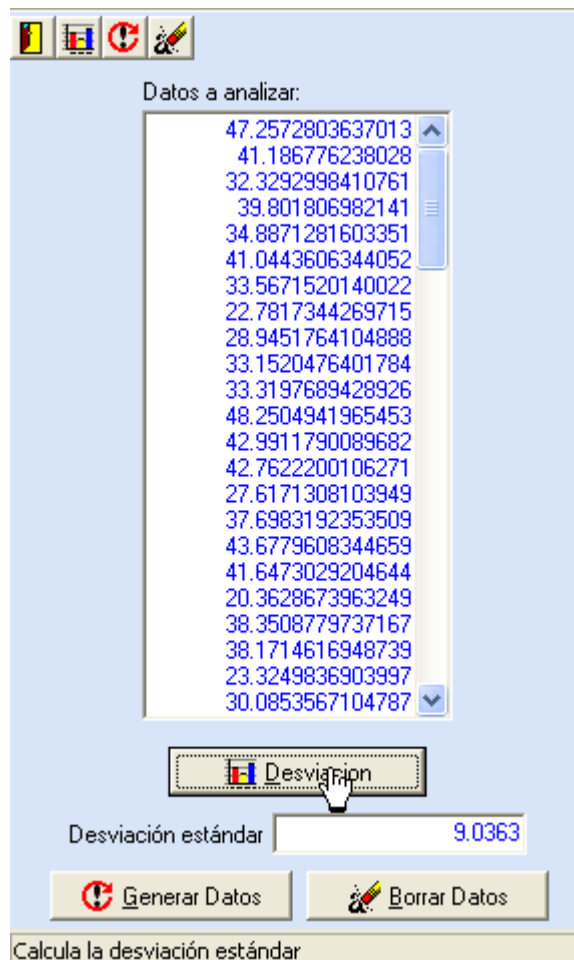
procedure MostrarDesviacion(ds: extended; m: TMemo);
begin
m.Text:= Format('%14.4f',[ds]);
end;

end.

```

Observe que tampoco se declara el tipo "tv500" porque el mismo está declarado en la unidad "uMedia".

La interfaz de la aplicación es esencialmente la misma que la del anterior ejemplo:



A más de los cambios obvios en las etiquetas, se han cambiado unas cuantas propiedades de acuerdo al siguiente detalle:

Icono de la aplicación: Chart 8.ico.

Form1: Name = fDesviacionEstandar; Hint = 'Desviación estándar'.

Memo1: Hint = 'Datos a analizar'.

Memo2: Name = mDesviacion; Hint = 'Desviación estándar calculada'.

ToolButton2: Name=tbDesviacion; Hint='Calcular la desviación estándar'.

BitBtn1: Name = bbDesviacion; Caption = '&Desviacion'; Hint= 'Calcular la desviación estándar'; Glyph = 'GrphBar.bmp'.

StatusBar1: SimpleText = 'Desviación estándar'; Hint = 'Desviación estándar'.

ImageList1: Name = ilDesviacion; Se reemplaza la figura 'Sum.bmp' por 'GrphBar.bmp'.

En esta ventana prácticamente sólo se necesita modificar el evento "onClick" del BitBtn "bbDesviacion" y los nombres en los eventos "onClick" del ToolButton "tbDesviación" y del evento "onClose" de la forma:

```
procedure TfDesviacionEstandar.bbDesviacionClick(Sender: TObject);
```

```
var x: tv500; ne: word; ds: extended;
```

```
begin
```

```
  x:= LeerVector(mDatos);
```

```
  ne:= mDatos.Lines.Count;
```

```
  ds:= DesviacionEstandar(x,ne);
```

```
  MostrarDesviacion(ds,mDesviacion);
```

```
end;
```

```
procedure TfDesviacionEstandar.tbSalirClick(Sender: TObject);
```

```
begin
```

```
  fDesviacionEstandar.Close;
```

```
end;
```

```
procedure TfDesviacionEstandar.tbDesviacionClick(Sender: TObject);
```

```
begin
```

```
  bbDesviacion.Click;
```

```
end;
```

Una vez hechas las modificaciones y corregidos los errores, la aplicación deberá tener la apariencia correspondiente a la interfaz de la aplicación.

14.3.3. Posición del mayor valor en un vector

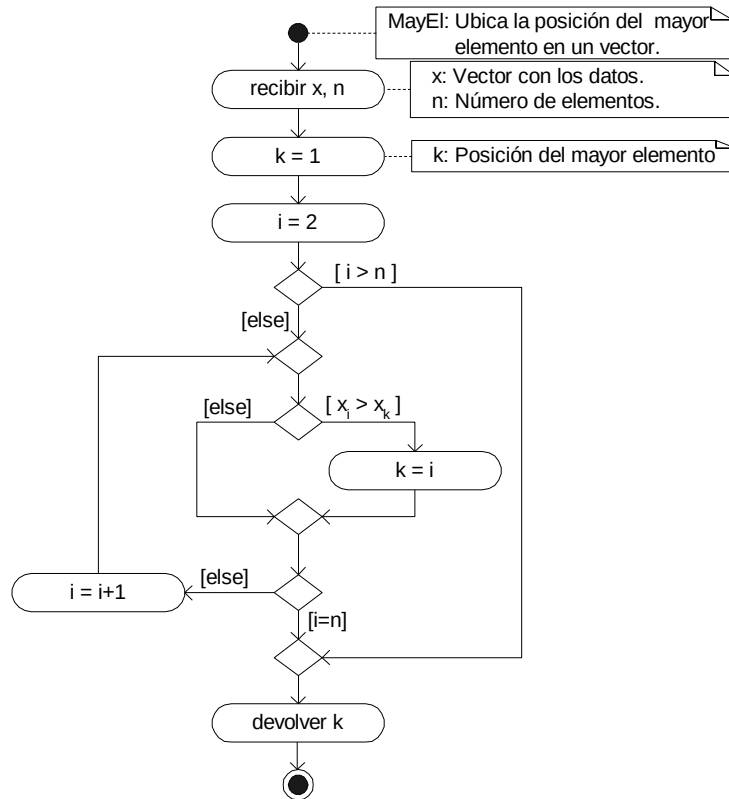
Como tercer ejemplo elaboraremos una aplicación para ubicar la posición del mayor valor en un vector.

La lógica para resolver este problema es muy sencilla: La posición del mayor valor se guarda en una variable (la cual se inicializa en 1), entonces se recorre la lista comparando cada elemento del vector con el valor que se encuentra en dicha posición, si el elemento es mayor, entonces se guarda en la variable la posición de dicho elemento, de esa manera, cuando ya no quedan más elemento que comparar, en la variable queda la posición del mayor elemento.

El algoritmo que implementa el anterior razonamiento se presenta en la siguiente página. En este ejemplo, para implementar el algoritmo emplearemos un vector de números enteros en lugar de un vector de números reales. Se contará también con un módulo para llenar un vector con "n" números enteros generados al azar.

Dado que el algoritmo para genera números enteros es prácticamente el mismo que para números reales (sólo que se trabaja con números enteros en lugar de números reales), se elaborará directamente el código del mismo.

En Pascal la función que genera números enteros aleatorios se llama igualmente *random*, pero en el caso de los números enteros recibe como parámetro un número entero y devuelve un número entero aleatorio comprendido entre 0 y el número introducido menos uno.



Por lo tanto la expresión que nos permite generar un número entero aleatorio comprendido entre dos límites dados es:

Nº aleatorio comprendido entre "li" y "ls" = random(ls-li+1)+li

Una vez más la aplicación tendrá una apariencia similar a la aplicación del primer ejemplo, por lo que lo más práctico es copiar la carpeta "Promedio", cambiar su nombre a "Mayor elemento", abrir la aplicación den Delphi y guardar el proyecto (File -> Save Project as...) con el nombre pMayEl, la unidad "uMedia" (File -> Save as...) con el nombre "uMayEl", la unidad "ufMedia" con el nombre "ufMayEl" y conservar la unidad "ufGenerar" porque sólo se modificarán algunos eventos de la misma. Posteriormente se debe borrar de la carpeta "Mayor elemento" todos los archivos cuyos nombres comiencen con "uMedia", "ufMedia" y "pMedia".

Entonces en la unidad "uMayEl" escribimos el código del módulo "MayEl", el código del módulo "GenVec" (que genera el vector de números enteros), así como el código de los módulos de lectura y escritura:

```
unit uMayEl;
```

```
interface
```

```
  uses StdCtrls, SysUtils, Math, QDialogs;
```

```
  type tvel1000 = array [1..1000] of integer;
```

```
  function LeerVector(m: TMemo): tvel1000;
```

```
  function GenVec(n: word; li, ls: integer): tvel1000;
```

```
  function MayEl(const x: tvel1000; n: word): word;
```

```
  procedure MostrarVector(x: tvel1000; n: word; m: TMemo);
```

```
  procedure MostrarMayor(may: word; m: TMemo);
```

implementation

```

function LeerVector(m: TMemo): tvel000;
var i,l: word; x: tvel000; p,n: integer;
begin
  try
    //Supresión de las líneas en blanco al final del memo
    n:= m.Lines.Count-1;
    while (m.Lines[n]='') and (n>=0) do begin
      m.Lines.Delete(n); dec(n); end;
    //Si el memo no tiene datos se genera un error
    inc(n); //Número de filas con datos
    if n=0 then raise EInvalidArgument.Create(
      'No se ha introducido ningún dato para calcular el promedio');
    //Lectura de los valores del memo
    p:= 0; //p: Posición del cursor en el memo
    for i:=1 to n do begin
      p:=p+Length(m.Lines[i-1])+2;
      x[i]:=StrToInt(m.Lines[i-1]); end;
    result:= x;
  except
    on EConvertError do begin
      MessageDlg('El número está mal escrito',mtError,[mbOk],0);
      l:= Length(m.Lines[i-1]); //Longitud del número mal escrito
      m.SelStart:= p-1-2; //Posición del primer carácter mal escrito
      m.SelLength:= 1; //Selección del número mal escrito
      m.SetFocus;Abort; end;
    on e:EInvalidArgument do begin
      MessageDlg(e.Message,mtError,[mbOk],0);
      m.SetFocus; m.SelStart:= 0; Abort; end;
  end;
end;

function GenVec(n: word; li,ls:integer): tvel000;
var x: tvel000; i: word; d: integer;
begin
  if n=0 then raise EInvalidArgument.Create(
    'El número de elementos a generar debe ser mayor a cero');
  if n>1000 then raise EInvalidArgument.Create(
    'El número de elementos a generar debe ser menor o igual a 1000');
  randomize;
  d:=ls-li+1;
  for i:=1 to n do
    x[i]:=random(d)+li;
  result:= x;
end;

function MayEl(const x: tvel000; n: word): word;
var i,k: word;
begin
  k:= 1;
  for i:=2 to n do
    if x[i]>x[k] then k:= i;
  result:=k;
end;

procedure MostrarVector(x: tvel000; n: word; m: TMemo);
var i:word;

```

```

begin
  try
    m.Lines.BeginUpdate; //Se deshabilita la actualización del memo
    m.Lines.Clear;
    for i:=1 to n do m.Lines.Append(IntToStr(x[i]));
  finally
    m.Lines.EndUpdate; //Se rehabilita la actualización del memo
  end;
end;

```

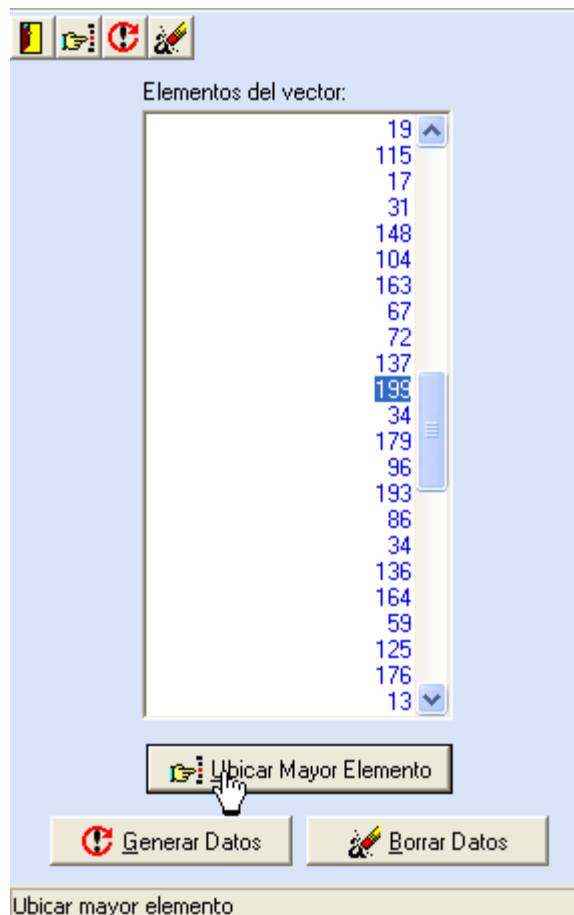
```

procedure MostrarMayor(may: word; m: TMemo);
var i, p: integer; //Selecciona la fila "may" del memo "m"
begin
  p:=0;
  for i:=0 to may-2 do
    p:=p+Length(m.Lines[i])+2;
  m.SetFocus;
  m.SelStart:= p;
  m.SelLength:= Length(m.Lines[may-1]);
end;

```

end.

La interfaz de la aplicación tiene la apariencia que se muestra en la siguiente figura:



A más de las modificaciones obvias en los títulos (captions), algunas de las propiedades modificadas, con relación al ejemplo 1, son:

Icono de la aplicación: *Eyes 2.ico* (que aunque no se ve en la ventana de la aplicación, si se ve en el ejecutable de la aplicación).

Form1: Name = 'fMayEl'; Hint = 'Mayor elemento'.

ImageList1: Name = 'ilMayEl'; Imágenes: DoorOpen.bmp; Find.bmp; Retry.bmp; Clear.bmp.

Memor1: Hint = 'Elementos del vector'.

Label1: Caption = 'Elementos del vector'.

ToolBar1: Images = ilMayEl.

ToolButton2: Name = tbUbicar'; Hint = 'Ubicar mayor elemento'.

BitBtn1: Name = 'bbUbicar'; Caption = '&Ubicar Mayor Elemento'; Glyph = 'Find.bmp', Hint = 'Ubicar mayor elemento'.

StatusBar1: SimpleText = 'Mayor elemento'; Hint = 'Mayor elemento'.

En los ToolButtons "tbUbicar", "tbGenerar" y "tbBorrar" de esta aplicación, en lugar de simular los eventos "onClick" de los BitBtns con el método "Click" (como se ha hecho en el ejemplo del promedio), se asignan directamente (en el inspector de objetos) los eventos "onClick" de los BitBtns: *bbUbicar*, *bbGenerar* y *bbBorrar*.

Al igual que en el primer ejemplo, en la unidad de esta ventana se deben incluir las unidades "ufGenerar" y "uMayEl", pues "ufMayEl" emplea los tipos y módulos de ambas unidades.

uses ufGenerar, uMayEl;

Se debe cambiar también el evento "onKeyPress" del memo "mDatos" porque en este caso solo se debe permitir la introducción de números enteros:

```
procedure TfMayEl.mDatosKeyPress(Sender: TObject; var Key: Char);
begin
  case Key of
    #8, #13, '0'..'9', '-':
      else
        Beep; Abort;
  end;
end;
```

Con excepción de los tipos, el evento "onClick" del BitBtn "bbGenerar" es esencialmente el mismo que en el ejemplo 1:

```
procedure TfMayEl.bbGenerarClick(Sender: TObject);
var x: tvel1000; li, ls: integer; ne: word;
begin
  if fGenerar.ShowModal = mrOk then begin
    li:= fGenerar.LeerLimiteInferior;
    ls:= fGenerar.LeerLimiteSuperior;
    ne:= fGenerar.LeerNumeroElementos;
    x:= GenVec(ne, li, ls);
    MostrarVector(x, ne, mDatos);
  end;
end;
```

El evento "onClick" del BitBtn "bbUbicar", que es donde se ubica y muestra el mayor elemento del memo, es el siguiente:

```
procedure TfMayEl.bbUbicarClick(Sender: TObject);
var x: tvel1000; ne, may: word;
begin
```

```

x:= LeerVector(mDatos);
ne:= mDatos.Lines.Count;
may:= MayEl(x,ne);
MostrarMayor(may,mDatos);
end;

```

Igualmente debemos cambiar el evento "onClose" del ToolButton "tbSalir", porque ha cambiado el nombre de la ventana:

```

procedure TfMayEl.tbSalirClick(Sender: TObject);
begin
    fMayEl.Close;
end;

```

Por otra parte, la ventana para la generación de los elementos es la misma que en el primer ejemplo, excepto que ahora los límites de los números generados sólo pueden ser número enteros, por lo tanto cambia tanto la declaración como el código de los módulos que leen los límites:

```

type
    TfGenerar = class(TForm)
        Label1: TLabel;
        Label2: TLabel;
        . . .
        procedure mNumEleExit(Sender: TObject);
    private
        { Private declarations }
    public
        function LeerLimiteInferior: integer;
        function LeerLimiteSuperior: integer;
        function LeerNumeroElementos: word;
    end;

```

Siendo el código respectivo:

```

function tfGenerar.LeerLimiteInferior: integer;
begin
    try
        result:= StrToInt(mLimInf.Text);
    except
        on EConvertError do begin
            MessageDlg('Límite inferior erróneo',mtError,[mbOK],0);
            mLimInf.SetFocus; mLimInf.SelectAll; end;
        end;
    end;

function tfGenerar.LeerLimiteSuperior: integer;
begin
    try
        result:= StrToInt(mLimSup.Text);
    except
        on EConvertError do begin
            MessageDlg('Límite superior erróneo',mtError,[mbOK],0);
            mLimSup.SetFocus; mLimSup.SelectAll; end;
        end;
    end;

```

El módulo que lee el número de elementos no cambia porque el tipo de dato devuelto sigue siendo entero.

También cambia el módulo que valida la introducción de los límites, porque ahora dichos límites son enteros:


```

procedure TfGenerar.mLimInfKeyPress(Sender: TObject; var Key: Char);
begin
  case Key of
    #8, '0'..'9', '.', '-':;
    else
      Beep; Abort;
    end;
end;

```

Por la misma razón, cambia también el módulo del evento "onExit" del límite inferior:

```

procedure TfGenerar.mLimInfExit(Sender: TObject);
var x: integer;
begin
  x:= LeerLimiteInferior;
  if x>LeerLimiteSuperior then mLimSup.Text:=IntToStr(x+1);
end;

```

Los otros eventos permanecen sin cambios porque en los mismos no se trabaja con los tipos de datos.

Una vez hechos estos cambios y corregidos los errores, la aplicación deberá tener en ejecución la apariencia de la figura mostrada previamente.

14.3.4. Multiplicación de dos matrices estáticas

Como cuarto ejemplo elaboraremos un módulo para multiplicar dos matrices estáticas de números reales. La ecuación que permite obtener la matriz resultante es la siguiente:

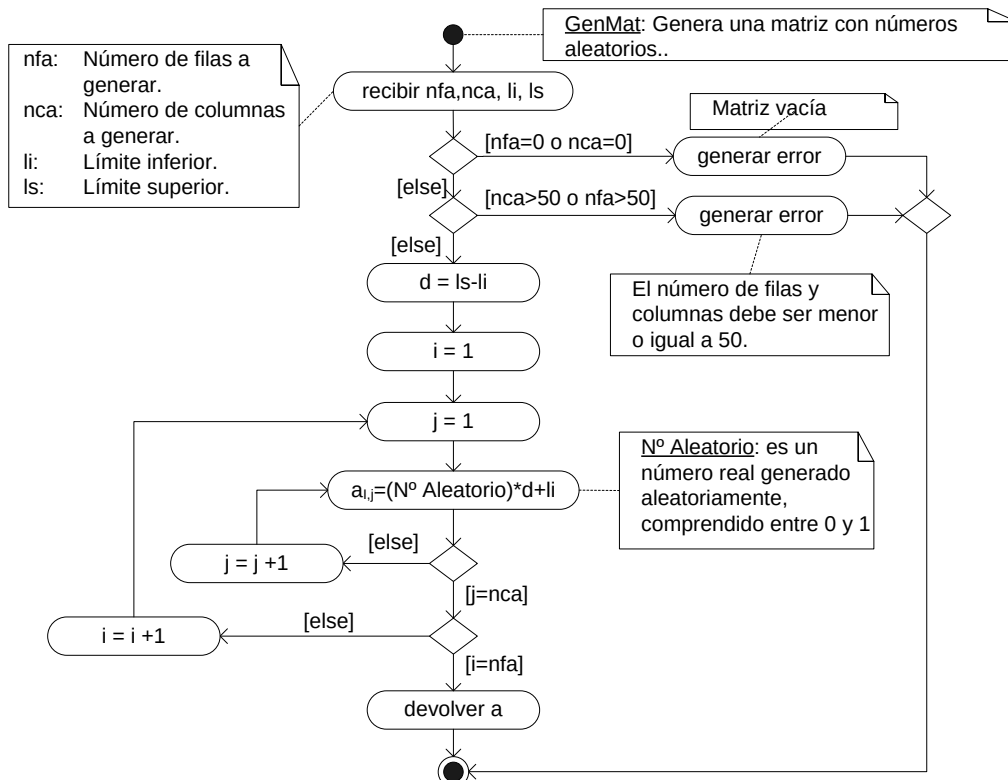
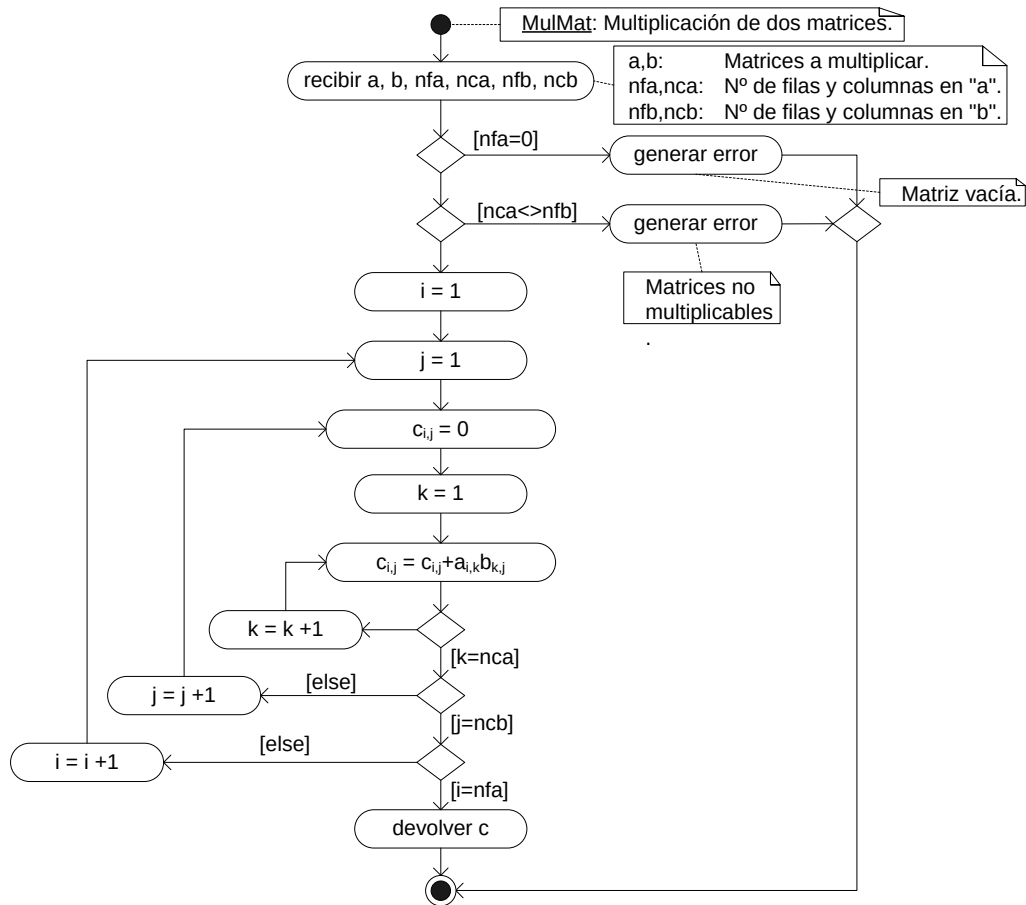
$$c_{ij} = \sum_{k=1}^{nca} a_{ik} * b_{kj} \quad \begin{cases} i=1 \rightarrow nfa \\ j=1 \rightarrow ncb, \text{ para cada valor de } i \end{cases}$$

Donde "a" y "b" son las matrices a multiplicar, "c" es la matriz resultante, "nfa" es el número de filas de la matriz "a", "nca" es el número de columnas de la matriz "a" y "ncb" es el número de columnas de la matriz "b".

Como se puede ver en la ecuación, para multiplicar las dos matrices son necesarios tres ciclos: el primer ciclo, con el contador "i", va desde 1 hasta el número de filas de "a", el segundo, con el contador "j", está dentro del primero y va desde 1 hasta el número de columnas de "b" y el tercero, con el contador "k", está dentro del segundo ciclo y va desde 1 hasta el número de columnas de "a".

Antes de efectuar la multiplicación se debe verificar primero si las matrices son multiplicables: dos matrices son multiplicables si el número de columnas de la matriz "a" (nca) es igual al número de filas de la matriz "b" (nfb), de no cumplirse esta condición las matrices no pueden ser multiplicadas. Por otra parte, dado que en la matriz "c", todos sus elementos son el resultado de una sumatoria, los mismos deben ser inicializados en cero. El algoritmo elaborado tomando en cuenta las anteriores consideraciones se presenta en el diagrama de actividades de la siguiente página.

Al igual que ocurre con los vectores, es útil contar con un módulo que genere una matriz de números aleatorios comprendidos entre ciertos límites. La lógica para generar esta matriz es básicamente la misma que para generar vectores, sólo que ahora se emplean dos ciclos: uno para las filas y otro para las columnas, tal como se muestra en el algoritmo de la siguiente página.



Para codificar este algoritmo elaboraremos una nueva aplicación (File -> New Application), una nueva unidad (File -> New Unit) y guardaremos la aplicación (File -> Save Project As...) en el subdirectorío "Multiplicación" dentro del directorío "Vectores y matrices estáticas", con los nombres "ufMulMat" para "Unit1", "uMulMat" para "Unit2" y "pMulMat" para "Project1".

En esta aplicación emplearemos un componente nuevo, un "StringGrid", el cual estudiaremos con más detalle cuando elaboremos la interfaz aplicación, sin embargo, por el momento debemos saber que en un "StringGrid", los datos se guardan en la propiedad "Cells" (en forma de cadenas de texto), de manera similar a una matriz, sólo que los índices se escriben al revés: primero el índice correspondiente a la columna y luego el índice correspondiente a la fila. Por otra parte, como ocurre con todos los componentes indexados de Delphi, el primer índice, tanto de las filas como de las columnas, es siempre cero.

El número de filas de un "StringGrid" se puede leer o fijar con la propiedad "RowCount" y el número de columnas con la propiedad "ColCount", la fila activa se puede leer o fijar con la propiedad "Row" y la columna activa con la propiedad "Col".

Para codificar este diagrama de flujo con matrices estáticas debemos decidir (como sucede con los vectores estáticos) el número de elementos que tendrá la matriz. En este caso asumiremos que las matrices a multiplicar no tendrán más de 50 filas y 50 columnas cada una, lo que implica $50 \times 50 = 2500$ elementos (250000 bytes si los elementos son de tipo extended).

Ahora escribimos el código de los módulos "MulMat" y "GenMat", así como el código de los módulos de lectura y escritura, en la unidad uMulMat:

```
unit uMulMat;
```

```
interface
```

```
  uses Grids, SysUtils, QDialogs, Math;
```

```
  type tMat = array [1..50,1..50] of extended;
```

```
  function LeerMatriz(s:tStringGrid):tMat;
```

```
  function GenMat(nfa,nca:byte; li,ls:extended):tMat;
```

```
  function MulMat(a,b:tMat; nfa,nca,nfb,ncb:byte):tMat;
```

```
  procedure MostrarMatriz(a:tMat; nfa,nca:byte; s:TStringGrid);
```

```
implementation
```

```
  function LeerMatriz(s:tStringGrid):tMat;
```

```
  var i,j,nfa,nca: byte; a: tMat;
```

```
  begin
```

```
    nfa:=s.RowCount; nca:=s.ColCount;
```

```
    try
```

```
      if nfa=0 then raise EInvalidArgument.Create('Matriz vacía');
```

```
      for i:=1 to nfa do
```

```
        for j:=1 to nca do
```

```
          a[i,j]:=StrToFloat(s.Cells[j-1,i-1]);
```

```
      result:=a;
```

```
    except
```

```
      on EConvertError do begin
```

```
        MessageDlg('Número mal escrito',mtError,[mbOK],0);
```

```
        s.SetFocus; s.Row:=i-1; s.Col:=j-1; Abort; end;
```

```
    on e:EInvalidArgument do begin
```

```
      MessageDlg(e.Message,mtError,[mbOK],0);
```

```

        s.SetFocus; Abort; end;
    end;
end;

function GenMat(nfa,nca:byte; li,ls:extended):tMat;
var i,j: byte; d:extended; a:tMat;
begin
    if (nfa=0) or (nca=0) then raise
        EInvalidArgument.Create(';Matriz vacía!');
    d:=ls-li; randomize;
    for i:=1 to nfa do
        for j:=1 to nca do
            a[i,j]:=random*d+li;
        result:=a;
    end;

function MulMat(a,b:tMat; nfa,nca,nfb,ncb:byte):tMat;
var i,j,k:byte; c:tMat;
begin
    if nfa=0 then raise EInvalidArgument.Create(';Matriz vacía!');
    if nca<>nfb then raise
        EInvalidArgument.Create('Matrices no multiplicables');
    for i:=1 to nfa do
        for j:=1 to ncb do begin
            c[i,j]:=0;
            for k:=1 to nca do
                c[i,j]:=c[i,j]+a[i,k]*b[k,j]; end;
            result:=c;
        end;

procedure MostrarMatriz(a:TMat; nfa,nca:byte; s:TStringGrid);
var i,j: byte;
begin
    s.RowCount:=nfa; s.ColCount:=nca;
    for i:=1 to nfa do
        for j:=1 to nca do
            s.Cells[j-1,i-1]:=Format('%12.2f',[a[i,j]]);
        end;
end.

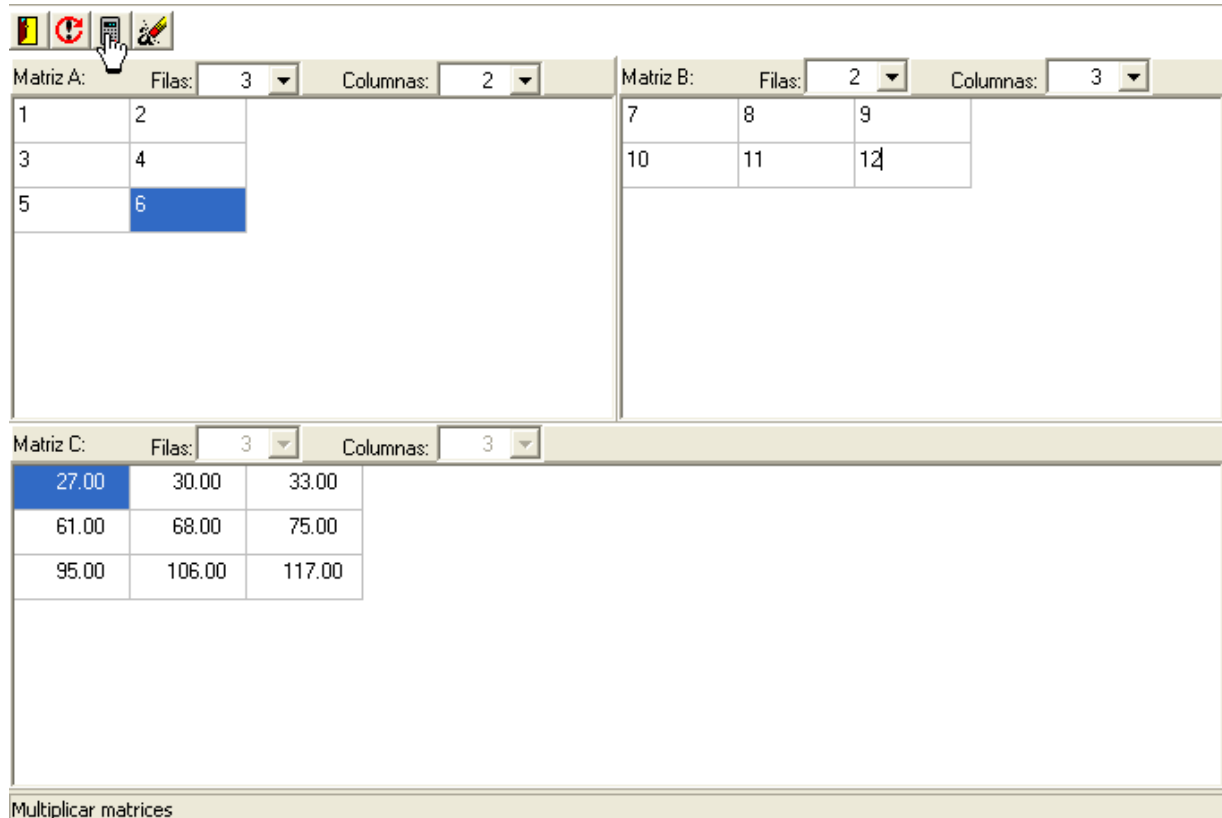
```

Como ya se dijo, en esta aplicación se emplea un nuevo componente, un "StringGrid" , que es una tabla que nos permite mostrar y editar texto (strings) y se encuentra en la página "Additional". En esta aplicación se emplean tres StringGrids: "sgMatrizA" para los datos de la matriz "a"; "sgMatrizB" para los datos de la matriz "b" y "sgMatrizC" para los datos de la matriz resultante "c".

A más de las propiedades antes mencionadas y empleadas en los módulos de la unidad "uMulMat", algunas de las más usuales son: "DefaultColWidth" para establecer o leer el ancho por defecto de las columnas; "DefaultRowHeight" para establecer o leer el alto por defecto de las filas; "FixedCols" para establecer o leer el número de columnas fija, que son aquellas que no se mueven con las barras de desplazamiento y que aparecen en color gris en la parte izquierda del "StringGrid", "FixedRows", igual que "FixedCols" pero con las filas; "ColWidths" para fijar o leer el ancho de cada una de las columnas (en forma de vector), estas dos últimas propiedades no se encuentran en el inspector de objetos, razón por la cual sólo pueden ser modifi-

cadass mediante código, así por ejemplo, si se quiere establecer el ancho de la cuarta columna del "StringGrid" "sgMatrizA" en 100 puntos, se debe escribir: `sgMatrizA.ColWidths[3]:=100`.

En ejecución, la interfaz de la aplicación tiene la siguiente apariencia:



Otra propiedad importante es Options, la cual nos permite cambiar una serie de opciones del StringGrid. Esta propiedad es de tipo conjunto y algunos de sus elementos que más frecuentemente se cambian en la misma son "goEditing": que cuando se coloca en verdadero (es decir cuando se incluye en el conjunto "Options"), permite modificar (editar) los datos del StringGrid; "goAlwaysShowEditor": que cuando se coloca en verdadero, permite editar las celdas sin necesidad de pulsar las teclas Enter o F2; "goDrawFocusSelected": que cuando es colocado en verdadero muestra la celda seleccionada con un color diferente; "goRowSizing" y "goColSizing": que cuando son colocados en verdadero permiten cambiar el alto y ancho de las filas y columnas del StringGrid; "goRowMoving" y "goColMoving": que cuando son colocados en verdadero permiten mover las filas y columnas del StringGrid; "goTabs": que cuando es colocado en verdadero permite navegar a través de las celdas con la tecla "Tab"; "goThumbTracking": que cuando es colocado en verdadero actualiza el StringGrid a medida que se mueve la barra de desplazamiento (y no sólo cuando la misma se suelta).

Además de los StringGrids, en esta aplicación se han empleado también seis paneles (Panel: ). Este componente se encuentra en la página estándar y se emplea generalmente con dos propósitos: a) Crear una barra de títulos y/o de componentes y b) para agrupar objetos relacionados permitiendo una organización más clara y eficiente de los mismos.

En esta aplicación tres de los paneles se han empleado con el primer propósito: Se han creado barras de títulos, con el nombre de la matriz (escrito

en la propiedad Caption) y con las etiquetas y ComboBoxes para fijar el número de filas y columnas de las mismas.

Los otros tres paneles, que no son visibles, pues están tapados por los StringGrids y los paneles visibles, se han empleado con el segundo propósito, para guardar objetos de manera más organizada. En cada uno de estos tres paneles se ha colocado un panel de títulos y un StringGrid. Para que los paneles de título queden alineados en la parte superior de los paneles contenedores se ha colocado su propiedad "align" en "alTop" y para que los StringGrids ocupen el resto del panel contenedor, se ha colocado su propiedad "Align" en "alClient".

Además en esta aplicación dos de los ToolButtons: "tbGenerar" y "tbBotrar", sólo se habilitan cuando el foco está sobre los StringGrids "sgMatrizA" o "sgMatrizB" (o uno de sus ComboBoxes), porque sólo en esos casos tiene sentido generar una matriz y/o borrar sus elementos. Para ello en esta aplicación se emplean dos ImageList: "ilMulMat1" e "ilMulMat2", la primera contiene las imágenes en color y es asignada a la propiedad "Images" del ToolBar "tbMulMat" y la segunda tiene las imágenes en gris y es asignada a la propiedad "DisabledImages" del ToolBar "tbMulMat".

Los otros componentes de la aplicación son conocidos.

Las propiedades modificadas en esta aplicación son las siguientes:

Icono de la aplicación: TechNlgy.ico

Form1: Name= fMulMat; color= clWhite; BorderStyle= bsNone; Position= poScreenCenter; Height =492; Width=688; Hint= 'Multiplicación de matrices'.

Panel1: Name=pMatrizA; Align=alLeft; Hint='Datos de la matriz A'; Width=340; Panel2:Name=pMtrizB; Align=alRight; Hint= 'Datos de la matriz B'; Width=340; Panel3: Name=pMatrizC; Align=alBottom; Hint= 'Resultados de la multiplicación'; Width=681; Propiedades comunes:Height=205; Caption = ''.

Panel4: Name=pTitMatA; Caption= 'MatrizA:'; Panel5: Name= pTitMatB; Caption= 'Matriz B'; Panel6: Name=pTitMatC; Caption= 'MatrizC:'; Propiedades comunes: Align= alTop; Height=21; Alignment=taLeftJustify.

ComboBox1: Name=cbFilasA; ComboBox2: Name=cbColumnasA; ComboBox3: Name=cbFilasB; ComboBox4: Name= cbColumnasB; ComboBox5: Name=cbFilasC; Enabled=False; ComboBox6: Name=cbColumnasC; Enabled=False; Propiedades comunes:Style=csDropDownList; Width=60.

StringGrid1: Name= 'sgMatrizA'; Options= [goEditing]; StringGrid2:Name= 'sgMatrizB'; Options= [goEditing]; StringGrid3: Name= 'sgMatrizC'; Propiedades comunes: Align=alClient; ColCount=2; RowCount=2; FixedRows=0; FixedCols=0; Options=[goDrawFocusSelected, goTabs, goThumbTracking].

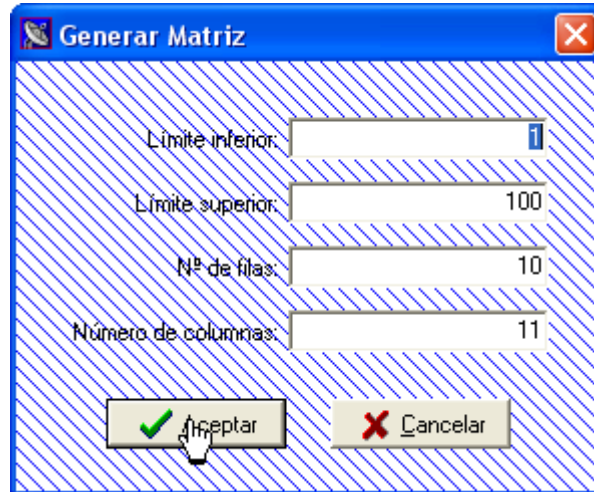
ToolBar1: Name=tbMulMat; AutoSize=True; EdgeInner=esNone; EdgeOuter=esNone; Images=ilMulMat1; DisabledImages=ilMulMat2.

ToolButton1: Name=tbSalir; ImageIndex=0; Hint= 'Salir de la aplicación'; ToolButton2: Name= tbGenerar; ImageIndex=1; Hint= 'Generar Matriz'; Enabled=False; ToolButton3: Name= tbMultiplicar; ImageIndex=2; Hint= 'Multiplicar matrices'; ToolButton4: Name=tbBorrar; ImageIndex=3; Hint= 'Borrar datos de la matriz'; Enabled = False; Propiedades comunes: Cursor=crHandPoint.

ImageList1: Name=ilMulMat1; Imágenes en color de: DoorOpen.bmp, Retry.bmp, Calculat.bmp, Clear.bmp; ImageList2: Name= ilMulMat2; Imágenes en gris de: DoorOpen.bmp, Retry.bmp, Calculat.bmp, Clear.bmp.

```
StatusBar1: Name=sbMulMat; SimplePanel=True; SimpleText='Multiplicación de matrices'; AutoHint=True.
```

Cuando se hace click en el ToolButton "tbGenerar" debe aparecer una nueva ventana donde se introducen los datos para generar una matriz al azar. Para ello y al igual que en el primer ejemplo, añadimos una nueva forma a la aplicación (File -> NewForm). La unidad correspondiente a esta nueva forma debe ser guardada (File -> Save As...) con el nombre 'ufGenerar'. Esta forma es muy similar a la forma "fGenerar" del primer ejemplo, sólo que ahora en lugar del número de elementos se introduce el número de filas y el número de columnas. La apariencia de esta ventana en ejecución es la que se muestra en la siguiente figura:



Como se puede observar, en esta ventana no existen nuevos componentes, siendo las propiedades modificadas las siguientes:

```
Form2: Name=fGenerar; Caption= 'Generar Matriz'; Width=300; Height=248; Color=clWhite; BorderStyle=bsSingle; bsBorderIcons=[biSystemMenu]; Position=poScreenCenter.
```

```
Label1: Caption= 'Límite inferior:'; Label2: Caption= 'Límite superior:'; Label3: Caption= 'Número de filas'; Label4: Caption= 'Número de columnas:'; Propiedades comunes: Width=130; Height=21; Alignment=taRightJustify; Cursor=crIBeam.
```

```
Memo1: Name=mLimInf; Memo2: Name=mLimSup; Memo3:Name=mNumFil; Memo4: Name=mNumCol; Propiedades comunes:Width=140; Height=21; Alignment=taRightJustify; Cursor=crIBeam.
```

```
BitBtn1: Name=bbAceptar; Kind=bkOK; Caption= '&Aceptar'; BitBtn2: Name=bbCancelar; Kind=bkCancel; Caption= '&Cancelar'; Propiedades comunes: Cursor=crHandPoint; Width=90.
```

Dado que la ventana tiene un comportamiento muy similar a la ventana "fGenerar" del primer ejemplo, presentamos el código de los módulos y eventos sin mucha explicación:

type

```
TfGenerar = class (TForm)
    Label1: TLabel;
    mLimInf: TMemo;
```

```
    . . .
```

```

    procedure mNumFilKeyPress(Sender: TObject; var Key: Char);
private
    { Private declarations }
public
    function LeerLimiteInferior: extended;
    function LeerLimiteSuperior: extended;
    function LeerNumeroFilas: byte;
    function LeerNumeroColumnas: byte;
end;

var
    fGenerar: TfGenerar;

implementation

{$R *.dfm}

uses Math, ufMulMat;

function tfGenerar.LeerLimiteInferior: extended;
begin
    try
        result:= StrToFloat(mLimInf.Text);
    except
        on EConvertError do begin
            MessageDlg('Límite inferior mal escrito',mtError,[mbOK],0);
            mLimInf.SetFocus; mLimInf.SelectAll; Abort; end;
        end;
    end;
end;

function tfGenerar.LeerLimiteSuperior: extended;
begin
    try
        result:= StrToFloat(mLimSup.Text);
    except
        on EConvertError do begin
            MessageDlg('Límite superior mal escrito',mtError,[mbOK],0);
            mLimSup.SetFocus; mLimSup.SelectAll; Abort; end;
        end;
    end;
end;

function tfGenerar.LeerNumeroFilas: byte;
begin
    try
        result:=StrToInt(mNumFil.Text);
    except
        on EconvertError do begin
            MessageDlg('Número de filas mal escrito',mtError,[mbOK],0);
            mNumFil.SetFocus; mNumFil.SelectAll; Abort; end;
        end;
    end;
end;

function tfGenerar.LeerNumeroColumnas: byte;
begin
    try
        result:= StrToInt(mNumCol.Text);
    except
        on EConvertError do begin

```



```
        MessageDlg('Número de columnas mal escrito',mtError,[mbOK],0);
        mNumCol.SetFocus; mNumCol.SelectAll; Abort; end;
    end;
end;
```

```
procedure TfGenerar.FormCreate(Sender: TObject);
begin
    fGenerar.Brush.Color:= clBlue;
    fGenerar.Brush.Style:= bsFDiagonal;
    mLimInf.Text:= '1';
    mLimSup.Text:= '100';
    mNumFil.Text:= '10';
    mNumCol.Text:= '11';
end;
```

Para que el texto de los memos quede seleccionado al ingresar a los mismos programamos el evento "onEnter" del memo mLimInfEnter y luego asignamos este procedimiento a los otros memos.

```
procedure TfGenerar.mLimInfEnter(Sender: TObject);
begin
    (Sender as TMemo).SelectAll;
end;
```

```
procedure TfGenerar.mLimInfExit(Sender: TObject);
begin
    if LeerLimiteInferior>LeerLimiteSuperior then
        mLimSup.Text:= FloatToStr(LeerLimiteInferior+1);
end;
```

```
procedure TfGenerar.mLimSupExit(Sender: TObject);
begin
    try
        if LeerLimiteSuperior<LeerLimiteInferior then raise
            EInvalidArgument.Create(
                'El límite superiro debe ser mayor al inferior');
    except
        on e: EInvalidArgument do begin
            MessageDlg(e.Message,mtError,[mbOK],0);
            mLimSup.SetFocus; mLimSup.SelectAll; Abort; end;
        end;
end;
```

```
procedure TfGenerar.mNumFileExit(Sender: TObject);
begin
    try
        if LeerNumeroFilas=0 then raise EInvalidArgument.Create(
            'El número de filas debe ser mayor a cero');
    except
        on e: EInvalidArgument do begin
            MessageDlg(e.Message,mtError,[mbOK],0);
            mNumFil.SetFocus; mNumFil.SelectAll; Abort; end;
        end;
end;
```

```
procedure TfGenerar.mNumColExit(Sender: TObject);
begin
    try
        if LeerNumeroColumnas=0 then raise EInvalidArgument.Create(
```

```

        'El número de columnas debe ser mayor a cero');
    except
    on e: EInvalidArgument do begin
        MessageDlg(e.Message, mtError, [mbOK], 0);
        mNumCol.SetFocus; mNumCol.SelectAll; Abort; end;
    end;
end;

```

```

procedure TfGenerar.mLimInfKeyPress(Sender: TObject; var Key: Char);
begin
    if not (Key in [#8, '.', '0'..'9', '+', '-', 'e', 'E']) then
        begin Beep; Abort; end;
end;

```

El anterior procedimiento tiene que ser asignado también al evento onKeyPress del memo "mLimSup". Lo mismo ocurre con el siguiente (que debe ser asignado al evento onKeyPress del memo "mNumCol").

```

procedure TfGenerar.mNumFilKeyPress(Sender: TObject; var Key: Char);
begin
    if not (Key in [#8, '0'..'9']) then begin Beep; Abort; end;
end;

```

Ahora escribimos los eventos de la forma principal (fMulMat), comenzando con el evento "onCreate", donde inicializamos las listas (Items) de los ComboBox con números comprendidos entre 1 y 50 y seleccionamos el segundo item:

```

uses ufGenerar, uMulMat;

```

```

{$R *.dfm}

```

```

procedure TfMulMat.FormCreate(Sender: TObject);
var i: byte;
begin
    for i:=1 to 50 do cbFilasA.Items.Append(Format('%8d', [i]));
    cbColumnasA.Items.Assign(cbFilasA.Items);
    cbFilasB.Items.Assign(cbFilasA.Items);
    cbColumnasB.Items.Assign(cbFilasA.Items);
    cbFilasC.Items.Assign(cbFilasA.Items);
    cbColumnasC.Items.Assign(cbFilasA.Items);
    cbFilasA.ItemIndex:= 1;
    cbColumnasA.ItemIndex:= 1;
    cbFilasB.ItemIndex:= 1;
    cbColumnasB.ItemIndex:= 1;
    cbFilasC.ItemIndex:= 1;
    cbColumnasC.ItemIndex:= 1;
end;

```

Programamos el evento "onClick" del ToolButton "tbSalir":

```

procedure TfMulMat.tbSalirClick(Sender: TObject);
begin
    Close;
end;

```

Programamos el evento "onClick" del ToolButton "tbGenerar", llenando los valores generados en el StringGrid "sgMatrizA", si el mismo tiene el foco (es decir si está activo) o en el StringGrid "sgMatrizB" en caso contrario:

```

procedure TfMulMat.tbGenerarClick(Sender: TObject);
var li,ls: extended; nf,nc: byte; a:tMat; foco: boolean;
begin

```

```

if fGenerar.ShowModal=mrOK then begin
  li:=fGenerar.LeerLimiteInferior;
  ls:=fGenerar.LeerLimiteSuperior;
  nf:=fGenerar.LeerNumeroFilas;
  nc:=fGenerar.LeerNumeroColumnas;
  a:=GenMat(nf,nc,li,ls);
  foco:= sgMatrizA.Focused;
  if foco then begin
    cbFilasA.ItemIndex:=nf-1;
    cbColumnasA.ItemIndex:=nc-1;
    MostrarMatriz(a,nf,nc,sgMatrizA); end
  else begin
    cbFilasB.ItemIndex:=nf-1;
    cbColumnasB.ItemIndex:=nc-1;
    MostrarMatriz(a,nf,nc,sgMatrizB); end;
  end;
end;

  Habilitamos los ToolButtons "tbGenerar" y "tbBorrar" programando el evento
  "onEnter" del StringGrid "sgMatrizA" y asignando también este módulo a
  los eventos "onEnter" del StringGrid "sgMatrizB" y de los ComboBox "cbFilasA",
  "cbColumnasA", "cbFilasB" y "cbColumnasC":

procedure TfMulMat.sgMatrizAEnter(Sender: TObject);
begin
  tbGenerar.Enabled:=True;
  tbBorrar.Enabled:=True;
end;

  Deshabilitamos los ToolButtons "tbGenerar" y "tbBorrar" programando el
  evento "onExit" del StringGrid "sgMatrizA" y asignando también este módulo
  al evento "onExit" del StringGrid "sgMatrizB" y de los ComboBox "cbFilasA",
  "cbColumnasA", "cbFilasB" y "cbColumnasC":

procedure TfMulMat.sgMatrizAExit(Sender: TObject);
begin
  tbGenerar.Enabled:=False;
  tbBorrar.Enabled:=False;
end;

  Cambiamos el número de filas o columnas de la matriz "A", cuando cambia
  el número de filas o columnas en el ComboBox "cbFilasA" o "cbColumnasA",
  programando sus eventos "onChange":

procedure TfMulMat.cbFilasAChange(Sender: TObject);
begin
  sgMatrizA.RowCount:= cbFilasA.ItemIndex+1;
end;

procedure TfMulMat.CbColumnasAChange(Sender: TObject);
begin
  sgMatrizA.ColCount:=cbColumnasA.ItemIndex+1;
end;

  Hacemos lo mismo para los ComboBox "cbFilasB" y "cbColumnasB":

procedure TfMulMat.cbFilasBChange(Sender: TObject);
begin
  sgMatrizB.RowCount:= cbFilasB.ItemIndex+1;
end;

procedure TfMulMat.cbColumnasBChange(Sender: TObject);

```

begin

sgMatrizB.ColCount:=cbColumnasB.ItemIndex+1;

end;

Borramos los elementos de la matriz "A" o "B" programando el evento "onClick" del ToolButton "tbBorrar":

procedure TfMulMat.tbBorrarClick(Sender: TObject);

var sg:TStringGrid; i:byte;

begin

if sgMatrizA.Focused **then** sg:=sgMatrizA **else** sg:=sgMatrizB;

for i:=0 **to** sg.RowCount-1 **do** sg.Rows[i].Clear;

end;

Finalmente multiplicamos las matrices programando el evento "onClick" del ToolButton "tbMultiplicar":

procedure TfMulMat.tbMultiplicarClick(Sender: TObject);

var a,b,c:tMat; nfa,nfb,nca,ncb:byte;

begin

a:=LeerMatriz(sgMatrizA);

b:=LeerMatriz(sgMatrizB);

nfa:=cbFilasA.ItemIndex+1;

nca:=cbColumnasA.ItemIndex+1;

nfb:=cbFilasB.ItemIndex+1;

ncb:=cbColumnasB.ItemIndex+1;

c:=MulMat(a,b,nfa,nca,nfb,ncb);

MostrarMatriz(c,nfa,ncb,sgMatrizC);

cbFilasC.ItemIndex:=nfa-1;

cbColumnasC.ItemIndex:=ncb-1;

end;

Una vez escrito el código y corregido los errores, la aplicación deberá tener la apariencia y devolver el resultado mostrado en la figura correspondiente a la interfaz de la aplicación.

14.3.5. Resolución de un sistema de ecuaciones lineales. Matrices estáticas

Como quinto ejemplo, elaboraremos una aplicación para resolver un sistema de ecuaciones lineales por el método de Gauss.

Las ecuaciones del método de eliminación de Gauss son las siguientes:

$$\left. \begin{aligned} a_{p,j} &= \frac{a_{p,j}}{a_{p,p}} \quad \left\{ \begin{array}{l} j=p+1 \rightarrow m, \text{ para cada valor de } p \\ \end{array} \right. \\ a_{i,j} &= a_{i,j} - a_{i,p} * a_{p,j} \quad \left\{ \begin{array}{l} i=p+1 \rightarrow n, \text{ para cada valor de } p, \text{ excepto para } p=n \\ j=p+1 \rightarrow m, \text{ para cada valor de } i \end{array} \right. \end{aligned} \right\} p=1 \rightarrow n$$

$$x_i = a_{i,m} - \sum_{k=i+1}^n a_{ik} * x_k \quad \left\{ \begin{array}{l} x_n = a_{n,m} \\ i=n-1 \rightarrow 1 \end{array} \right.$$

Donde "n" es el número de ecuaciones en el sistema, "m", es el número de columnas (n+1), "a" es la matriz de los coeficientes aumentada con la columna de las constantes, "x" es el vector con los resultados.

Analizando las ecuaciones vemos que las dos primeras ecuaciones están dentro de un ciclo cuyo contador "p" va desde 1 hasta "n". La primera ecuación a su vez se resuelve con otro ciclo cuyo contador "j" va desde 1 hasta "n", mientras que la segunda se resuelve con dos ciclos, el primero cuyo contador "i" va desde "p+1" hasta n y dentro de este el segundo cuyo contador "j" va

desde " $p+1$ " hasta " m ". Los resultados se obtienen con la tercera ecuación que se resuelve a su vez con dos ciclos, el primero cuyo contador " i " va desde " $n-1$ " bajando hasta 1 y dentro de este un segundo ciclo, donde se calcula la sumatoria y cuyo contador " k " va desde " $i+1$ " hasta n .

El algoritmo elaborado en base a este análisis se presenta en el diagrama de actividades de la siguiente página. Se debe anotar que no se trata de una versión rigurosa del *método de eliminación de Gauss*, pues para ello sería necesario efectuar al menos un *pivotaje total* o por lo menos *parcial*.

Para implementar el código de este módulo creamos una nueva aplicación (File -> New Application), una nueva unidad (File -> New Unit) y guardamos la aplicación (File -> Save Project As...) en el directorio "Método de Gauss", con los nombres "ufGauss" para "Unit1", "uGauss" para "Unit2" y "pGauss" para "Project1".

En la unidad "uGauss", escribimos el código del módulo Gauss, así como el código de los módulos de lectura y escritura. Una vez más (debido a que trabajamos con matrices estáticas) debemos decidir el número de filas que reservaremos para la matriz aumentada. En este caso reservaremos 100 filas y 101 columnas, lo que nos permitirá resolver un sistema de hasta 100 ecuaciones lineales con 100 incógnitas.

El código escrito en la unidad "uGauss" es el siguiente:

```
unit uGauss;
```

```
interface
```

```
uses Grids, SysUtils, QDialogs, Math;
```

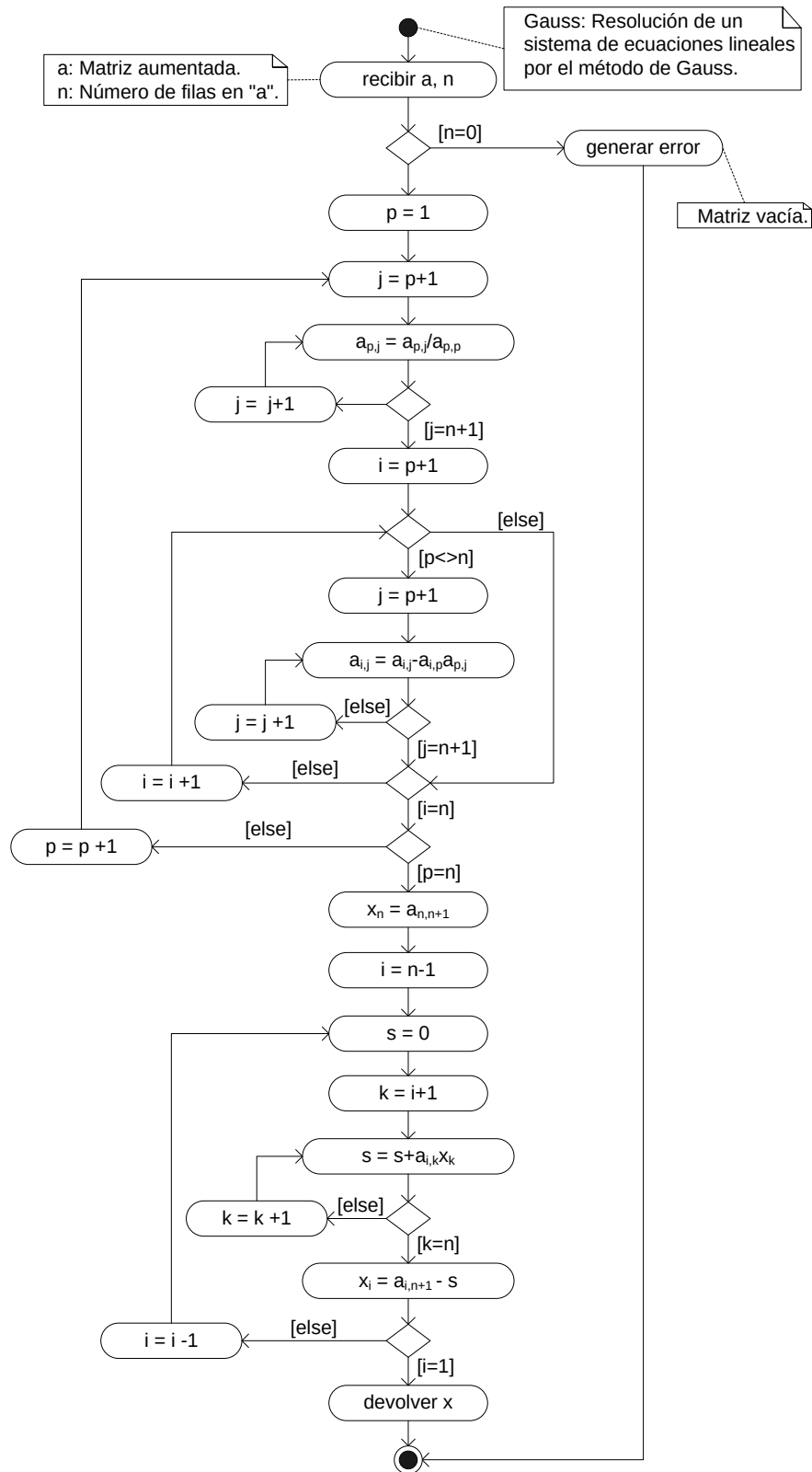
```
type tMat2 = array[1..100,1..101] of extended;
      tVec2 = array[1..100] of extended;
```

```
function LeerMatriz(sg:TStringGrid):tMat2;
function GenerarMatriz(li,ls:extended; n:byte):tMat2;
function Gauss(a:Tmat2; n:byte):tVec2;
procedure MostrarSoluciones(x:tVec2; n:byte; sg:tStringGrid);
procedure MostrarFilas(nf:byte; sg:TStringGrid);
procedure MostrarMatriz(a:tMat2; nf:byte; sg:tStringGrid);
```

```
implementation
```

```
function LeerMatriz(sg:TStringGrid):tMat2;
var i,j,n: byte; a:tMat2;
begin
  try
    n:=sg.RowCount-1;
    for i:=1 to n do
      for j:=1 to n+1 do
        a[i,j]:=StrToFloat(sg.Cells[j,i]);
    result:=a;
  except
    on EConvertError do begin
      ShowMessage('Número mal escrito');
      sg.SetFocus; sg.Row:=i; sg.Col:=j; Abort; end;
  end;
end;

function GenerarMatriz(li,ls:extended; n:byte):tMat2;
```



```

var i,j:byte; d:extended; a:tMat2;
begin
  randomize; d:=ls-li;
  for i:=1 to n do

```

```

        for j:=1 to n+1 do
            a[i,j]:=random*d+li;
        result:=a;
    end;

function Gauss(a:Tmat2; n:byte):tVec2;
var i,j,k,p:byte; s:extended; x:tVec2;
begin
    if n=0 then raise EInvalidArgument.Create(';Matriz Vacía!');
    for p:=1 to n do begin
        for j:=p+1 to n+1 do a[p,j]:=a[p,j]/a[p,p];
        for i:=p+1 to n do
            if p<>n then
                for j:=p+1 to n+1 do a[i,j]:=a[i,j]-a[i,p]*a[p,j]; end;
        x[n]:=a[n,n+1];
        for i:=n-1 downto 1 do begin
            s:=0;
            for k:=i+1 to n do s:=s+a[i,k]*x[k];
            x[i]:=a[i,n+1]-s; end;
        result:=x;
    end;

procedure MostrarSoluciones(x:tVec2; n:byte; sg:tStringGrid);
var i:byte;
begin
    sg.RowCount:=n+1;
    for i:=1 to n do begin
        sg.Cells[0,i]:=Format('    x[%d]',[i]);
        sg.Cells[1,i]:=Format('%12.8g',[x[i]]); end;
    end;

procedure MostrarFilas(nf:byte; sg:TStringGrid);
var i,j: byte;
begin
    sg.RowCount:=nf+1; sg.ColCount:=nf+2;
    for i:=1 to nf do sg.Cells[0,i]:=Format('%10d',[i]);
    for j:=1 to nf+1 do sg.Cells[j,0]:=Format('%10d',[j]);
    end;

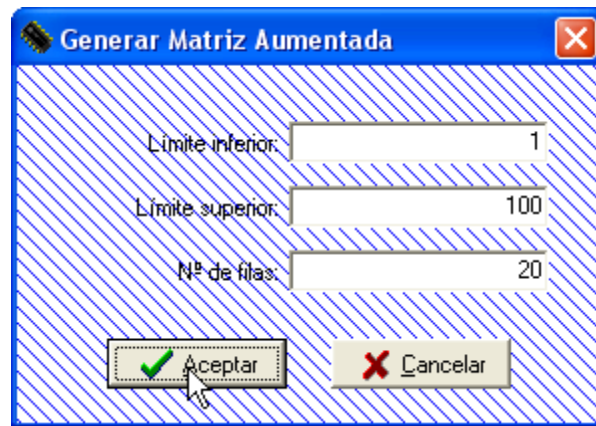
procedure MostrarMatriz(a:tMat2; nf:byte; sg:tStringGrid);
var i,j: byte;
begin
    MostrarFilas(nf,sg);
    for i:=1 to nf do
        for j:=1 to nf+1 do
            sg.Cells[j,i]:= Format('%12.2f',[a[i,j]]);
        end;
    end;
end.

```

Dado que la ventana para generar la matriz con valores aleatorio sólo difiere de la ventana "fGenerar" del anterior ejemplo en que ahora sólo se requiere el número de filas y no el número de columnas, lo más práctico es modificar dicha ventana. Para ello se añade la unidad "ufGenerar" (que como sabemos corresponde a la ventana "fGenerar") al proyecto, eligiendo la opción "Add to Project..." del menú "Project" (o pulsando las teclas Shift+F11). En la ventana que aparece se va al directorio "Multiplicación" y se abre el archivo "ufGenerar", con ello la ventana "fGenerar" es añadida al proyecto, sin embargo, y antes de realizar ninguna modificación en dicha ventana, de-

bemos guardar una copia de la misma en el directorio "Método de Gauss, para ello elegimos la opción "Save As..." del menú "File", vamos al directorio "Método de Gauss" y guardamos el archivos con el mismo nombre. Ello es necesario porque de lo contrario se estaría trabajando con la unidad "ufGenerar" de la aplicación "pMultiplicar", con lo que la misma quedaría modificada. (En el siguiente capítulo veremos otra forma en la que podemos reutilizar componentes de otras aplicaciones).

La ventana "fGenerar", con las modificaciones hechas, tiene la siguiente apariencia:



Donde como se puede observar sólo se ha cambiado el título (caption) de la forma y se ha borrado la etiqueta y el memo correspondiente a las columnas.

En el código igualmente se debe eliminar la declaración y el código del módulo "LeerNumeroColumnas", así como el evento "onExit" del memo "mNumCol".

Finalmente se debe modificar el evento "onCreate" eliminando al inicialización del memo "mNumCol":

```
procedure TfGenerar.FormCreate(Sender: TObject);
begin
  fGenerar.Brush.Color:= clBlue;
  fGenerar.Brush.Style:= bsFDiagonal;
  mLimInf.Text:= '1';
  mLimSup.Text:= '100';
  mNumFil.Text:= '10';
end;
```

Ahora elaboremos la interfaz de la aplicación, que en ejecución, tendrá la apariencia que se muestra en la figura de la siguiente página.

En esta aplicación no existen componentes nuevos, siendo las propiedades modificadas las siguientes:

Icono de la aplicación: "Chip.ico".

Form1: Name= 'fGauss'; Caption= 'Resolución de un sistema de ecuaciones lineales por el método de Gauss'; BorderStyle= bsSingle; BorderIcons= [biSystemMenu]; Position=poScreenCenter; Width=683; Height=487.

Paneles ocultos: Panel1: Name= PmatAumentada; Align= alLeft; Caption=''; Width=525; Panel2: Name= pSoluciones; Align= alRight; Caption= ''; Width= 150.

Paneles visibles: Panel3: Name= pTitMatAumentada; Align= alTop; Height=21; Panel4: Name= pTitSoluciones; Align= alTop; Height= 21; Panel5: Name= pBotones; Align= alBottom; Height= 41.

Resolución de un sistema de ecuaciones lineales por el método de Gauss

MATRIZ AUMENTADA N° de ecuaciones: 3

A	1	2	3	4
1	2	8	2	14
2	1	6	-1	13
3	2	-1	2	5

SOLUCIONES

Variable	Valor
x[1]	5
x[2]	1
x[3]	-2

Resolver el sistema Generar Matriz Aumentada

```
StringGrid1: Name= sgMatAumentada; Options= [goEditing, goAl-
waysShowEditor]; StringGrid2: Name= sgSoluciones; Propiedades comunes:
Cursor= crHandPoint; Width= 210.
```

```
ComboBox1: Name=cbFilas; Style= csDropDownList; Width=60.
```

No se debe olvidar incluir en la unidad "ufGauss" las unidades "ufGenerar" y "uGauss":

```
uses ufGenerar, uGauss;
```

En el evento "onCreate" de la forma escribimos el texto de las celdas que no cambian con el número de filas y asignamos los valores a la lista (items) del ComboBox "cbFilas":

```
procedure TfGauss.FormCreate(Sender: TObject);
var i: byte;
begin
  sgMatAumentada.Cells[0,0]:=Format('%10s',['A']);
  for i:=2 to 100 do cbFilas.Items.Append(Format('%6d',[i]));
  cbFilas.ItemIndex:=0;
  MostrarFilas(2,sgMatAumentada);
  sgSoluciones.ColCount:=2;
  sgSoluciones.RowCount:=2;
  sgSoluciones.Cells[0,0]:=Format('%10s',['Variable']);
  sgSoluciones.Cells[1,0]:=Format('%10s',['Valor']);
end;
```

Cambiamos el número de filas de la matriz (y mostramos las celdas con los títulos) cuando cambia el número de filas del ComboBox "cbFilas", programando su evento "onChange":

```

procedure TfGauss.cbFilasChange(Sender: TObject);
var i,nf:byte;
begin
  nf:=cbFilas.ItemIndex+2;
  for i:=1 to nf do sgMatAumentada.Rows[i].Clear;
  MostrarFilas(nf,sgMatAumentada);
  sgMatAumentada.Row:=1;
  sgMatAumentada.Col:=1;
  sgMatAumentada.SetFocus;
  sgSoluciones.RowCount:=2;
  sgSoluciones.Cells[1,1]:='';
end;

```

Mostramos la ventana "fGenerar" y generamos la matriz con los límites y número de filas escritos en la misma, programando el evento "onClick" del BitBtn "bbGenerar":

```

procedure TfGauss.bbGenerarClick(Sender: TObject);
var li,ls: extended; nf:byte; a:tMat2;
begin
  if fGenerar.ShowModal=mrOk then begin
    li:=fGenerar.LeerLimiteInferior;
    ls:=fGenerar.LeerLimiteSuperior;
    nf:=fGenerar.LeerNumeroFilas;
    a:=GenerarMatriz(li,ls,nf);
    cbFilas.ItemIndex:=nf-2;
    MostrarMatriz(a,nf,sgMatAumentada); end;
end;

```

Finalmente resolvemos el sistema de ecuaciones programando el evento "onClick" del BitBtn "bbResolver":

```

procedure TfGauss.bbResolverClick(Sender: TObject);
var a:tMat2; x:tVec2; nf:byte;
begin
  a:=LeerMatriz(sgMatAumentada);
  nf:=cbFilas.ItemIndex+2;
  x:=Gauss(a,nf);
  MostrarSoluciones(x,nf,sgSoluciones);
end;

```

Una vez escrito el código y corregidos los errores, la aplicación deberá tener la apariencia y devolver el resultado que se muestra en la figura correspondiente a la interfaz de la aplicación, en la cual se ha resuelto el siguiente sistema de ecuaciones lineales:

$$\begin{aligned}
 2x_1 + 8x_2 + 2x_3 &= 14 \\
 x_1 + 6x_2 - x_3 &= 13 \\
 2x_1 - x_2 + 2x_3 &= 5
 \end{aligned}$$

14.4. Preguntas y ejercicios

1. Cree una aplicación con un StringGrid. En el evento "onCreate" de la forma haga que el StringGrid tenga 301 filas y dos columnas, siendo los títulos de las columnas: "N° de fila" y "N° real". Llene la primera celda de cada fila con números consecutivos del 1 al 300 y la segunda con números enteros aleatorios comprendidos entre -1000 y 1000.
2. Cree una aplicación con un Memo, un Edit y un BitBtn. En el evento "onCreate" de la forma haga que el memo tenga 100 puntos de ancho, 300 de alto y que tenga una barra de desplazamiento vertical. Elabore un módu-

lo para que genere un vector con hasta 500 números enteros aleatorios comprendidos entre 1 y 10000 y otro para que muestre en el memo el vector generado. Programe el evento "onClick" del BitBtn de manera que genere y muestre un vector con el número de elementos escrito en el Edit.

3. Cree una aplicación con un Memo, un Edit y un BitBtn. En el evento "onCreate" de la forma haga que el memo tenga 140 puntos de ancho, 250 de alto, que su fondo sea azul, las letras amarillas y que tenga una barra de desplazamiento vertical, llene el memo con 500 filas con el siguiente texto: "Fila número N°", donde "N°" es un número consecutivo que va desde 1 hasta 500. Escriba un módulo que reciba un número y un memo y seleccione en el memo el número de fila recibido, finalmente programe el evento "onClick" del BitBtn de manera que se seleccione el número de fila escrito en el Edit.
4. Cree una aplicación con un ToolBar, un ImageList, un Memo y un StatusBar. En el evento "onCreate" de la forma haga que la forma esté centrada en la pantalla y no tenga bordes, que el ToolBar tenga 3 botones, que el ImageList tenga las imágenes en color de "DoorOpen.bmp", "Retry.bmp" y "Clear.bmp", que las imágenes del ToolBar sean las del ImageList, que el Hint del primer ToolButton sea 'Salir de la aplicación', del segundo sea 'LlenarMemo' y del tercero 'Borrar Memo'; que el Memo tenga 130 puntos de ancho, 250 de alto, que su fondo sea de color amarillo y tenga una barra de desplazamiento vertical, siendo su Hint 'Números enteros consecutivos'; que el StatusBar tenga un solo panel, que muestre automáticamente los hint de los objetos y que el texto inicial y Hint del mismo sea su nombre completo. Programe el evento "onClick" del ToolButton1 de manera que la aplicación se cierre, del ToolButton2, de manera que el memo se llene con 200 números pares y del ToolButton3 de manera que se borre el texto del memo.
5. Cree una aplicación con dos paneles, un StringGrid, dos Label y dos ComboBox. En el evento "onCreate" de la aplicación haga que el primer panel esté alineado en la parte superior de la forma y tenga 30 puntos de alto, que el segundo panel esté alineado en toda la forma, que el StringGrid pertenezca al Panel2, esté alineado en todo el panel y tenga 2 filas y dos columnas; que los Label pertenezcan al Panel1, tengan los títulos 'Filas:' y 'Columnas:', sean transparentes, estén ubicados a 8 puntos de la parte superior y 30 y 130 puntos de la izquierda; que los ComboBox pertenezcan al Panel1, tengan 60 puntos de ancho, estén ubicados a 5 puntos de la parte superior y a 5 puntos a la derecha de los Label, que su lista sean los números del 1 al 100 y que esté seleccionado el número 2. Programe el evento "onChange" de los ComboBox de manera que al cambiar el mismo cambie el número de filas o columnas respectivo en el StringGrid.
6. Cree una aplicación con dos paneles, un StringGrid, dos Label, dos ComboBox y un BitBtn. Escriba un módulo que genere una matriz de hasta 100 filas y 100 columnas de números enteros aleatorios comprendidos entre -1000 y 1000 y otro que muestre los elementos de una matriz de números enteros en un StringGrid, con los números de fila y columna en la primera columna y fila del StringGrid. En el evento "onCreate" de la aplicación haga que el primer panel esté alineado en la parte superior de la forma y tenga 40 puntos de alto, que el segundo panel esté alineado en toda la forma, que el StringGrid pertenezca al Panel2, esté alineado en todo el panel, tenga tres filas y tres columnas, que se actualice a medida que se mueven las barras de desplazamiento; que los Label perte-

- nezcan al Panel1, tengan los títulos 'Filas:' y 'Columnas:', sean transparentes, estén ubicados a 13 puntos de la parte superior y 30 y 130 puntos de la izquierda; que los ComboBox pertenezcan al Panel1, tengan 60 puntos de ancho, estén ubicados a 10 puntos de la parte superior y a 5 puntos a la derecha de los Label, que su lista sean los números del 2 al 100 y que esté seleccionado el número 2; que el BitBtn tenga la figura "Calculat.bmp", el título "Generar" y que al hacer clic sobre el mismo se genere y muestre una matriz aleatoria con el número de filas y columnas establecidos en los ComboBox.
7. Cree una aplicación con un Memo, dos BitBtn y dos Edit. Escriba un módulo recursivo que calcule el promedio de un vector de números reales, otro que genere un vector de hasta 250 números reales comprendidos entre 1 y 100, otro que muestre en un memo un vector de hasta 250 números reales y otro que lea de un memo un vector de hasta 250 números reales. En el evento "onCreate" de la forma haga que el memo tenga 120 puntos de ancho, 200 de alto, la letra sea de color azul y tenga una barra de desplazamiento vertical; el primer BitBtn tenga la imagen "Calculat.bmp", el título "&Calcular" y que al hacer clic sobre el mismo lea y calcule el promedio de los números del memo y se muestre el resultado en el Edit1; que el segundo BitBtn tenga la imagen "Retry.bmp", el título "&Generar" y que al hacer clic sobre el mismo se genere y muestre el número de elementos especificado en el Edit2.
 8. Cree una aplicación con un StringGrid, dos BitBtn y un Edit. Escriba un módulo recursivo que reciba un vector de hasta 500 números enteros y devuelva un vector con los números impares del vector recibido, otro que genere un vector de hasta 500 números enteros comprendidos entre 1 y 1000, otro que muestre en un StringGrid un vector de hasta 500 números enteros y otro que lea de un StringGrid un vector de hasta 500 números enteros. En el evento "onCreate" de la forma haga que el memo tenga 130 puntos de ancho, 250 de alto, la letra sea de color azul y tenga una barra de desplazamiento vertical; el primer BitBtn tenga la imagen "Report.bmp", el título "&Impares" y que al hacer clic sobre el mismo lea los elementos del memo y deje y muestre sólo los números impares; que el segundo BitBtn tenga la imagen "BulbOn.bmp", el título "&Generar" y que al hacer clic sobre el mismo se genere y muestre el número de elementos especificado en el Edit1.
 9. Cree una aplicación con un StringGrid, un BitBtn y un Edit. Escriba un módulo que reciba un número entero comprendido entre 1 y 50 y devuelva un vector con esa cantidad de números primos y otro que muestre el vector en un StringGrid. En el evento "onCreate" de la forma haga que el StringGrid tenga dos columnas, 140 puntos de ancho y 200 de alto; el BitBtn tenga la imagen "Table.bmp", el título "&Primos" y que al hacer clic sobre el mismo genere y muestre la cantidad de números primos especificado en el Edit.
 10. Cree una aplicación con 2 Panel, 2 Label, 2 ComboBox, 2 BitBtn y 1 StringGrid. Programe un módulo que reciba una matriz de hasta 200 filas y 200 columnas de números enteros y devuelva su transpuesta, otro que lea la matriz de un StringGrid, otro que muestre la matriz en un StringGrid y otro que genere la matriz de números enteros. En el evento "onCreate" de la forma haga que el primer panel esté alineado en la parte superior de la forma, tenga 30 puntos de alto y no tenga título; que el segundo esté alineado en toda la forma y no tenga título; que el StringGrid pertenezca al Panel2, esté alineado en todo el panel, sus celdas tengan 50 puntos de ancho y tenga 2 filas y 2 columnas; que los

Label pertenezcan al Panel1, tengan los títulos "Filas:" y "Columnas:", que sean transparentes, estén a 8 puntos del margen superior y 20 y 140 puntos del margen izquierdo; que los ComboBox pertenezcan al Panel1, tengan 60 puntos de ancho, estén a 5 puntos del margen superior y a 5 puntos de los Label, siendo su lista los números del 1 al 200, estando seleccionado el número 2; que el primer BitBtn tenga la figura "DockStack.bmp", el título "&Generar", esté a 250 puntos del margen izquierdo y a 5 del superior, tenga 90 puntos de ancho y que al hacer clic sobre el mismo se genere y muestre una matriz con los números de fila y columna de los ComboBox; que el segundo BitBtn tenga la figura "Export.bmp", el título "&Transponer", esté a 350 puntos del margen izquierdo y a 5 del superior, tenga 90 puntos de ancho y que al hacer clic sobre el mismo transponga y muestre la matriz tranpuesta.

11. Cree una aplicación con 4 Panel, 2 Label, 2 ComboBox, 2 BitBtn y 3 StringGrid. Programe un módulo que reciba 2 matrices de hasta 150 filas y 150 columnas de números reales y devuelva la suma de las mismas, otro que lea la matriz de un StringGrid, otro que muestre la matriz en un StringGrid y otro que genere la matriz de números reales. En el evento "onCreate" de la forma haga que el primer panel esté alineado en la parte superior de la forma, tenga 30 puntos de alto y no tenga título; que el segundo esté alineado a la izquierda, tenga un ancho igual a la tercera parte del ancho disponible en la forma y no tenga título; que el tercero esté alineado a la izquierda, tenga un ancho igual a la tercera parte del ancho disponible en la forma y no tenga título; que el cuarto esté alineado en la forma y no tenga título; que el primer StringGrid pertenezca al Panel2, el segundo al Panel3 y el tercero al Panel4, estén alineados en todo el panel, no tenga filas ni columnas fijas y que tengan una fila y una columna; que los Label pertenezcan al Panel1, tengan los títulos "Filas:" y "Columnas:", que sean transparentes, estén a 8 puntos del margen superior y 20 y 140 puntos del margen izquierdo; que los ComboBox pertenezcan al Panel1, tengan 60 puntos de ancho, estén a 5 puntos del margen superior y a 5 puntos de los Label, siendo su lista los números del 1 al 150, estando seleccionado el número 2; que el primer BitBtn tenga la figura "Edit.bmp", el título "&Generar", esté a 250 puntos del margen izquierdo y a 5 del superior, tenga 80 puntos de ancho y que al hacer clic sobre el mismo se generen y muestren dos matrices, en los dos StringGrid, con el número de filas y columnas de los ComboBox; que el segundo BitBtn tenga la figura "Sum.bmp", el título "&Sumar", esté a 340 puntos del margen izquierdo y a 5 del superior, tenga 80 puntos de ancho y que al hacer clic sobre el mismo lea las matrices de los StringGrid 1 y 2, sume sus elementos y muestre la matriz resultante en el tercer StringGrid.
12. Cree una aplicación con 2 Panel, 1 Label, 1 ComboBox y 1 Memo. Programe un módulo que reciba un número entero comprendido entre 1 y 100 y devuelva una matriz con ese número de filas con los elementos del triángulo de Pascal, otro que muestre las filas de la matriz en un Memo. En el evento "onCreate" de la forma haga que el primer panel esté alineado en la parte superior de la forma, tenga 30 puntos de alto y no tenga título; que el segundo esté alineado en toda la forma y no tenga título; que el Memo pertenezca al Panel2, esté alineado en toda la forma, tengan barras de desplazamiento vertical y horizontal; que el Label pertenezcan al Panel1, tengan el título "Filas:", que sean transparentes, esté a 8 puntos del margen superior y 30 del margen izquierdo; que el ComboBox pertenezcan al Panel1, tengan 50 puntos de ancho, esté a 5 puntos del margen superior y a 5 puntos del Label, siendo su lista los números del 1 al 100, estando seleccionado el número 1; que el BitBtn

tenga la figura "BulbOn.bmp", el título "&Generar" y que al hacer clic sobre él se genere y muestre en el Memo el triángulo de Pacal con el número de filas especificado en el ComboBox (Las 6 primeras filas del triángulo de Pascal son: 1; 1,1; 1,2,1; 1,3,3,1; 1,4,6,4,1; 1,5,10,10,5,1; 1,6,15,20,15,6,1).

15. PARÁMETROS ABIERTOS Y VECTORES DINÁMICOS

Como se vio en el anterior capítulo, cuando se emplean vectores estáticos es necesario fijar de antemano el número de elementos, algo que evidentemente es una desventaja cuando se crean módulos de carácter general. Afortunadamente *Pascal* (como otros lenguajes) nos permite evitar esta limitante recurriendo a parámetros abiertos, vectores dinámicos y punteros. Comenzaremos el estudio de estas formas alternativas a las matrices estáticas con los parámetros abiertos y vectores dinámicos.

Los parámetros abiertos son parámetros vector que no tienen límites y que en consecuencia pueden recibir (o devolver) vectores con cualquier número de elementos. Por esta razón, los módulos creados con parámetros abiertos tienen un carácter general.

Un parámetro abierto se declara igual que un vector (empleando las palabras *array of*) pero sin especificar los límites. Por ejemplo en la siguiente declaración el parámetro "z" es un parámetro abierto de tipo doble:

```
function sum(var z: array of double): double;
```

Como se puede ver en este ejemplo, el parámetro abierto ha sido declarado por referencia (con *var*). Los parámetros abiertos pueden ser declarados por valor, por referencia o como constantes. En general es recomendable declararlos por referencia o como constantes para ahorrar memoria y tiempo, recordemos que los parámetros por valor crean una copia de los datos originales lo que consume tanto memoria como tiempo, algo que es más notorio aún en el caso de los vectores que, al ser datos estructurados, pueden contar con cientos o miles de elementos. No obstante, existen ocasiones donde los parámetros abiertos deben ser declarados por valor, por ejemplo, cuando el módulo debe recibir tanto valores literales como variables.

Sin importar cuales sean los índices del vector original, una vez que es recibido como un parámetro abierto, su primer índice (al interior del módulo) siempre será cero.

Para determinar el número de elementos de un parámetro abierto se emplean las funciones *Length* o *High*. La función *Length* tiene la siguiente sintaxis:

Length(vector o parámetro abierto)

Y devuelve el número de elementos del vector o parámetro abierto. La función *High* tiene la siguiente sintaxis:

High(vector o parámetro abierto)

Y devuelve el índice más alto del vector o parámetro abierto. Puesto que todos los parámetros abiertos comienzan en cero, el índice más alto es siempre el número de elementos menos uno. Si el vector no tiene elementos, *High* devuelve -1.

La función complementaria a *High* es *Low*, cuya sintaxis es la siguiente:

Low(vector o parámetro abierto)

Y devuelve el primer índice de un vector o parámetro abierto, sin embargo, esta función no es de utilidad con los parámetros abiertos, pues como ya se dijo, el primer índice de un parámetro abierto es siempre cero.

Como su nombre indica, los parámetros abiertos son sólo eso parámetros (datos de una función o procedimiento) y en consecuencia no pueden ser empleados en otro lugar que no sea en el interior de un procedimiento o función.

15.1. VECTORES DINÁMICOS

Como ya se explicó en el anterior acápite, los parámetros abiertos son solo eso: parámetros, por lo tanto no pueden ser empleados en la declaración de variables, tipos de datos o en la recepción y devolución de resultados de una función. Los vectores dinámicos por el contrario son tipos de datos y como tales pueden ser empleados en cualquier parte de un programa.

Los vectores dinámicos se caracterizan por no tener límites definidos. Se declaran de manera similar a un vector estático excepto que no se incluyen los límites, tal como se muestra a continuación:

```
var Nombre de la variable : array of tipo de dato;
```

Donde "Nombre de la variable" es cualquier nombre válido en Pascal o una lista de nombres separados por comas.

Al igual que en los vectores estáticos es recomendable declarar primero un tipo de dato y luego variables de ese tipo, de acuerdo al siguiente formato:

```
type Nombre del tipo: array of tipo de dato;
```

```
var Nombre de la variable: Nombre del tipo;
```

Sólo de esta manera es posible emplear los vectores dinámicos como parámetros y resultados de los procedimientos y funciones.

En el interior de un módulo los vectores dinámicos se comportan prácticamente de la misma forma que los parámetros abiertos: su primer índice es cero, el número de elementos se determina con la función *Length* y el índice del último elemento con *High*. Sin embargo, los vectores dinámicos son en realidad *punteros* (direcciones de memoria) y como tales tiene el comportamiento característico de estos. Así por ejemplo, una vez que se declara una variable como vector dinámico, se le debe asignar memoria antes de poder emplearla.

En el caso de los vectores dinámicos de *Delphi* la memoria se reserva con el procedimiento *SetLength* de acuerdo a la siguiente sintaxis:

```
SetLength(Nombre del vector dinámico, Número de elementos);
```

Analicemos brevemente el por qué es necesario reservar memoria para las variables que son vectores dinámicos. Como se dijo, estas variables son en realidad punteros y un puntero (como veremos con más detalle en un tema posterior) es una variable que almacena una dirección de memoria y una dirección de memoria en la arquitectura de 32 bits ocupa 4 bytes (y en la de 64 bits, 8 bytes). Por lo tanto cuando se declara una variable como vector dinámico sólo se está reservando memoria para 4 bytes (la dirección de memoria), pero no para los elementos del vector. Esta es la razón por la cual es necesario reservar explícitamente la memoria que será empleada para guardar los elementos del vector.

Lo que hace *SetLength* es reservar un bloque continuo de memoria para el número de elementos especificado, entonces guardar la dirección de ese bloque en la variable. Por lo tanto una variable dinámica contiene sólo la dirección de memoria donde se encuentran los elementos, pero no los elementos en sí, es necesario tener en mente este hecho para evitar errores cuando se trabaja con variables diámicas (y punteros). Por ejemplo, después de ejecutar el siguiente segmento de código:

```
type tvreal = array of real;
var a,b: tvreal; i: byte;
begin
```



```

SetLength(a,5);
for i:=0 to 4 do a[i]:= i+1;
b:= a;
for i:=0 to 4 do b[i]:= b[i]+10;
...
end;

```

Los elementos del vector "a" son: 11, 12, 13, 14 y 15, y no 1, 2, 3, 4 y 5 como sucede cuando se trabaja con vectores estáticos. Esto es así porque la asignación `b:= a`, no copia los elementos del vector dinámico "a" en el vector dinámico "b" (como ocurre con los vectores estáticos), sino que simplemente copia la dirección de memoria almacenada en "a" en la variable "b". De esa manera tanto "a" como "b" tienen la misma dirección de memoria (apuntan al mismo lugar), es decir ambas trabajan con los mismos datos, en otras palabras es como si a una misma variables se le hubiera dado dos nombres. Por lo tanto cualquier cambio que se haga a los datos del vector "b" es también un cambio a los datos del vector "a" y viceversa.

Para copiar los elementos de un vector dinámico en otro, se debe reservar memoria para ambos vectores y copiar los elementos con un ciclo `for` o con la función `Copy` que tiene la siguiente sintaxis:

Copy(vector dinámico, índice inicial, número de elementos)

Esta función reserva un bloque de memoria y copia del "vector dinámico" el "número de elementos" especificado comenzando en el "índice inicial" dado. La función devuelve la dirección de memoria donde se ha realizado la copia. Así si en el anterior ejemplo, la intención era copiar los elementos y no la dirección de memoria, debería haberse escrito:

`b:= Copy(a,0,Length(a));`

En lugar de `b:=a`.

La función `Copy` puede emplearse también para truncar los elementos de un vector dinámico, así por ejemplo para truncar el vector "a" en el tercer elemento se escribe:

`a:= Copy(a,0,3);`

Aunque normalmente se prefiere emplear el procedimiento `SetLength`:

`SetLength(a,3).`

Al trabajar con variables dinámicas se debe tener cuidado también en las operaciones relacionales, pues sólo se comparan las direcciones de memoria y no los valores a los que apuntan. Ello explica el porqué al final del siguiente segmento de código, la variable `result` termina con el valor `False` y no `True` (como ocurriría si se trabajara con vectores estáticos):

```

type tvinteger = array of integer;
var a,b : tvinteger;
begin
  SetLength(a,1);
  SetLength(b,1);
  a[0]:= 5;
  b[0]:= 5;
  result:= a=b;
end;

```

Como los vectores dinámicos no tienen límites, los índices no son controlados por el compilador, así si en el anterior código se escribe `a[10]:= 3`; no se produce ningún error al momento de compilar el programa, pero al momento de ejecutarlo es muy probable que se produzca un error, ocasionando la

falla del programa o inclusive del sistema operativo. Esto se debe a que se está escribiendo en sectores de memoria no reservados y por lo tanto es muy probable que se esté sobre escribiendo algún dato empleado por el mismo programa, otro programa o el sistema operativo.

Una vez utilizado el vector dinámico en las operaciones requeridas, es necesario liberar la memoria ocupada, pues de lo contrario, seguiría reservada evitando así que pueda ser empleada por otras aplicaciones o por el mismo programa.

La memoria reservada para un vector dinámico se libera con el procedimiento *Finalize*, de acuerdo a la siguiente sintaxis:

```
Finalize(Nombre del vector dinámico)
```

Dado que al interior de un módulo los vectores dinámicos se comportan de manera muy similar a los parámetros abiertos, pueden ser empleados en lugar de los parámetros abiertos, así, la siguiente función:

```
function sum(var z: array of double): double;
```

Que emplea parámetros abiertos, puede ser declarada también de la siguiente manera:

```
type tvdouble = array of double;
```

```
function sum(z: tvdouble): double;
```

Observe sin embargo que en este caso es necesario declarar previamente un tipo de dato (*tvdouble*) y que el parámetro se recibe por valor. El parámetro podría ser recibido también por referencia, pero con ello sólo se ahorrarían 4 bytes (una dirección de memoria), porque como ya se dijo, los vectores dinámicos son en realidad punteros y un puntero es sólo una dirección de memoria.

Puesto que se recibe la dirección de memoria donde se encuentran los datos, al interior del módulo se trabaja realmente con los datos originales, por lo tanto, cuando se trabaja con vectores dinámicos, cualquier modificación que se haga a los datos modifica en realidad los datos originales, algo que se debe tomar muy en cuenta para evitar resultados inesperados. En consecuencia, cuando no se desea modificar los datos originales, se debe trabajar con parámetros abiertos en lugar de vectores dinámicos.

Otro aspecto a tomar en cuenta es que cuando en un módulo se recibe un vector empleando vectores dinámicos, sólo se puede mandar al mismo otro vector dinámico (del mismo tipo), mientras que cuando se recibe un vector en forma de parámetro abierto, se le puede mandar un vector de valores, un vector estático o un vector dinámico, con la única condición de que sus elementos sean del tipo especificado en el parámetro. Por lo tanto los módulos creados con parámetros abiertos tienen un carácter más general que los creados con vectores dinámicos.

Al trabajar con vectores dinámicos, no se debe olvidar que en realidad son punteros y que en consecuencia se comportan como tales.

15.2. Ejemplos

En los siguientes ejemplos las interfaces gráficas de las aplicaciones no tienen componentes nuevos con relación a los del anterior tema, razón por la cual no se darán mayores explicaciones con relación a las mismas.

1. Elabore una aplicación con dos Memos, una etiqueta y un BitBtn y en la misma escriba los módulos que empleando vectores dinámicos generen, muestren y lean los elementos de un vector de números enteros desde un

Memo. Escriba también un módulo para seleccionar una fila determinada en un memo. La forma debe estar centrada en la pantalla, tener 400 puntos de ancho, 450 de alto, con el título "Vectores" y los bordes de un cuadro de diálogo. Los memos deben tener 150 puntos de ancho, 300 de alto, estar centrado horizontalmente (separados con 10 puntos), a 25 puntos del margen superior, contar con una barra de desplazamiento vertical y su texto debe estar alineado a la derecha. La etiqueta debe tener el título "Elementos del vector", ser transparente y estar a 15 puntos por encima del Memo1 (alineado con el mismo a la izquierda). El BitBtn debe tener 160 puntos de ancho, estar centrado a 10 puntos por debajo del Memo1, tener el título "Probar módulos dinámicos", el Glyph "Retry.bmp" y el puntero "crHandPoint". El evento "onClick" del BitBtn debe ser programado de manera que genere un vector con 300 números enteros comprendidos entre 1 y 1000, muestre dichos elementos en el primer memo, lea los elementos del memo en otro vector, muestren dichos elementos en el segundo memo y seleccione la fila 100 del mismo.

El código de los módulos (programados en una unidad) es el siguiente:

```
unit uDinamicos;

interface

uses StdCtrls, SysUtils, QDialogs;

type tvInteger = array of integer;

procedure SeleccionarFila(f:cardinal; m:TMemo);
function GenerarVEnteros(n:cardinal; li,ls:integer):tvInteger;
procedure MostrarVEnteros(x:tvInteger; m:TMemo);
function LeerVEnteros(m:TMemo):tvInteger;

implementation

procedure SeleccionarFila(f:cardinal; m:TMemo);
var sc:Integer; i:cardinal;
begin
    sc:=0;
    for i:=0 to f-1 do sc:=sc+Length(m.Lines[i])+2;
    m.SelStart:=sc;
    m.SelLength:=Length(m.Lines[f]);
    m.SetFocus;
end;

function GenerarVEnteros(n:cardinal; li,ls:integer):tvInteger;
var i:cardinal; x:tvInteger; d:integer;
begin
    SetLength(x,n); d:=ls-li+1;
    for i:=0 to n-1 do x[i]:=Random(d)+li;
    result:=x;
end;

procedure MostrarVEnteros(x:tvInteger; m:TMemo);
var i,n: cardinal;
begin
    n:=Length(x);
    try
        m.Lines.BeginUpdate;
        m.Lines.Clear;
```

```

    for i:=0 to n-1 do m.Lines.Append(IntToStr(x[i]));
  finally
    m.Lines.EndUpdate;
  end;
end;

function LeerVEnteros(m:TMemo):tvInteger;
var i,n: cardinal; x:tvInteger;
begin
  try
    n:=m.Lines.Count;
    SetLength(x,n);
    for i:=0 to n-1 do x[i]:=StrToInt(m.Lines[i]);
    result:=x;
  except
    on EConvertError do begin
      ShowMessage('El número está mal escrito');
      m.SetFocus; SeleccionarFila(i,m);
    end;
  end;
end;
end.

```

Observe que en los módulos "GenerarVector" y "LeerVector", antes de generar o leer valor alguno, se reserva la memoria necesaria con `SetLength`. Aún en estos sencillos ejemplos, la ventaja de emplear vectores dinámicos es evidente: se pueden generar vectores con prácticamente cualquier número de elementos. En realidad en estos ejemplos el límite es de 4294967295 elementos debido a que el número de elementos a generar se recibe en una variable de tipo cardinal.

En el módulo "MostrarVector" se ha empleado el bloque "Try-Finally", para asegurar que el memo quede activo incluso si se produce un error (pues es desactivado con el método "BeginUpdate"), en este módulo no se controlan los errores de conversión (`EConvertError`) porque en el vector de números enteros no puede existir otra cosa que números enteros y cualquier número entero puede ser convertido siempre en un String.

En el módulo "LeerVEnteros" se ha empleado el bloque Try- Except, para controlar los errores de conversión (`EConvertError`), porque los números pueden estar mal escritos en el memo. En este módulo se emplea el método "SeleccionarFila", para seleccionar la fila donde se encuentra el número mal escrito.

El evento "onCreate" de la aplicación es el siguiente:

```

procedure TForm1.FormCreate(Sender: TObject);
begin
  Form1.Width:= 400;
  Form1.Height:= 450;
  Form1.Caption:= 'Vectores';
  Form1.BorderStyle:= bsDialog;
  Form1.Position:= poScreenCenter;
  Memo1.Lines.Clear;
  Memo1.Width:= 150;
  Memo1.Height:= 300;
  Memo1.Left:= (Form1.ClientWidth-Memo1.Width*2-10) div 2;
  Memo1.Top:= 25;
  Memo1.ScrollBars:= ssVertical;
  Memo1.Alignment:=taRightJustify;
  Memo2.Lines.Clear;

```

```

Memo2.Width:= Memo1.Width;
Memo2.Height:= Memo1.Height;
Memo2.Left:= Memo1.Left+Memo1.Width+10;
Memo2.Top:= Memo1.Top;
Memo2.ScrollBars:= ssVertical;
Memo2.Alignment:=taRightJustify;
Label1.Caption:= 'Elementos del vector: ';
Label1.Transparent:= True;
Label1.Left:= Memo1.Left;
Label1.Top:= Memo1.Top-15;
BitBtn1.Width:= 160;
BitBtn1.Left:= (Form1.Width-BitBtn1.Width) div 2;
BitBtn1.Top:= Memo1.Top+Memo1.Height+10;
BitBtn1.Caption:= '&Probar módulos dinámicos';
BitBtn1.NumGlyphs:= 2;
BitBtn1.Glyph.LoadFromFile('C:\Archivos de programa\Archivos Comunes\
    Borland Shared\Images\Buttons\Retry.bmp');
BitBtn1.Cursor:= crHandPoint;
end;

```

Y el evento "onClick" del BitBtn (donde se prueban los módulos) es:

```

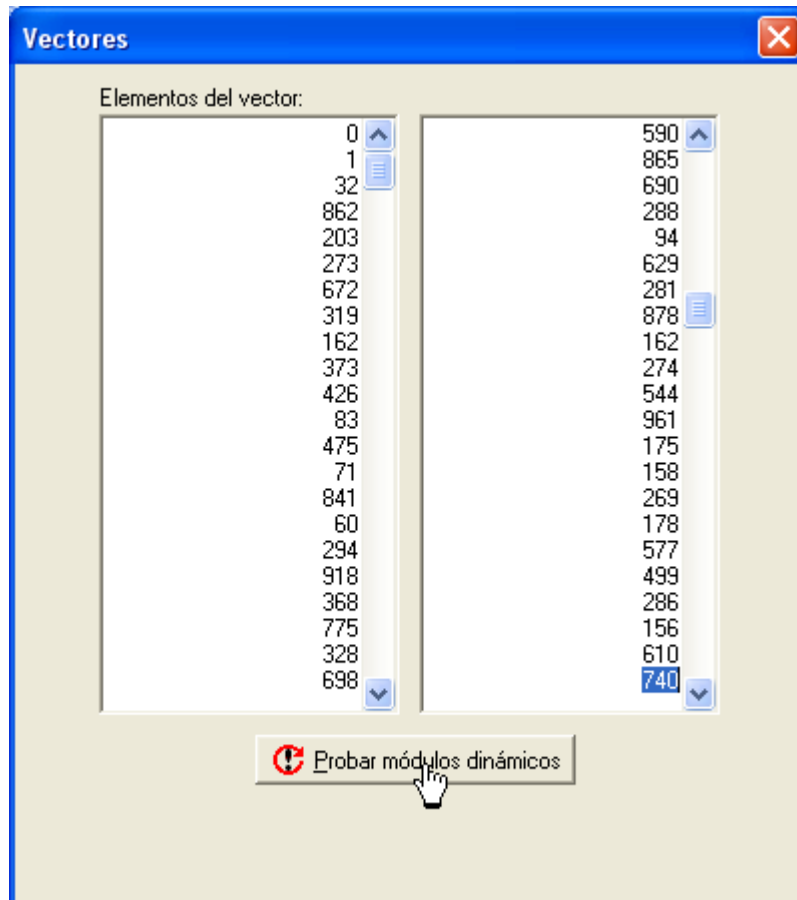
procedure TForm1.BitBtn1Click(Sender: TObject);
var x,y:tvInteger;
begin
    x:=GenerarVEnteros(300,1,1000);
    MostrarVEnteros(x,Memo1);
    Finalize(x);
    y:=LeerVEnteros(Memo1);
    MostrarVEnteros(y,Memo2);
    Finalize(y);
    SeleccionarFila(100,Memo2);
end;

```

En ejecución la aplicación tiene la apariencia que se muestra en la figura de la siguiente página.

2. Elabore una aplicación con un ToolBar, un StatusBar, un StringGrid, un Memo y un Label. En la aplicación debe crear un módulo que genere un vector con "n" números reales aleatorios comprendidos entre 0 y 10000, otro que muestre un vector de números reales en un StringGrid con dos columnas, mostrando en la primera columna el número de elemento y en la segunda el número real con dos dígitos después del punto, otro que lea los elementos de la segunda columna de un StringGrid, otro que calcule el promedio de un vector de números reales y otro que muestre un número real en un memo con dos dígitos después del punto. La forma debe tener 300 puntos de ancho y 400 de alto, estar centrada en la pantalla, tener el color "clInactiveCaptionText" y no tener bordes. El ToolBar debe tener tres botones con las imágenes en color de "DoorOpen.bmp", "BulbOn.bmp" y "Calculat.bmp", con los Hint: "Salir de la aplicación", "Generar vector" y "Calcular promedio", siendo la forma del puntero una mano apuntando. El StatusBar debe tener un solo panel con el texto "Cálculo del promedio" y debe mostrar los Hint de manera automática. El StringGrid debe tener 2 columnas (de 60 y 120 puntos), estar centrado horizontalmente y a 30 puntos del margen superior, estando fijas una fila y una columna, los títulos de las columnas deben ser "N°" y "Valor", y tener una barra de desplazamiento vertical. El memo debe tener 130 puntos de ancho, 21 de alto, con el texto alineado a la derecha, estar centrado horizontalmente (conjuntamente el label), a 10 puntos por debajo StringGrid y ser sólo de lectura. El label debe tener el tí-

tulo "Promedio:", ser transparente, estar a 5 puntos a la izquierda del memo y a 3 puntos por debajo del borde superior del memo. Al hacer clic en el primer ToolButton la aplicación debe cerrarse, al hacer clic en el segundo ToolButton, se debe pedir el número de elementos a generar y generar dicho número de elementos mostrándolos en el StringGrid y al hacer clic en el tercer ToolButton, se debe calcular el promedio de los elementos del StringGrid y mostrar dicho promedio en el memo.



La lógica para el cálculo del promedio y la generación de los elementos del vector es esencialmente la misma que con vectores estáticos, sólo que con parámetros abiertos los módulos son más generales, pues no está limitado a un número fijo de elementos.

$$x = \frac{\sum_{i=1}^n x_i}{n}$$

En este ejemplo recibiremos los vectores como parámetros abiertos y devolveremos los resultados como vectores dinámicos. El código de la unidad donde se escriben los módulos es el siguiente:

```
unit uParamAbiertos;
```

```
interface
```

```
  uses Math, QDialogs, SysUtils, StdCtrls, Grids;
```

```
  type tvReal = array of real;
```

```
function GenerarVReales(n: cardinal):tvReal;  
procedure MostrarVReales(const x: array of real; s:TStringGrid);  
function LeerVReales(s:TStringGrid):tvReal;  
function Mediapa(const x: array of real):real;  
procedure MostrarReal(x:real; m:TMemo);
```

implementation

```
function GenerarVReales(n: cardinal):tvReal;  
var i:cardinal; x:tvReal;  
begin  
  SetLength(x,n);  
  for i:=0 to n-1 do x[i]:= random*10000;  
  result:=x;  
end;  
  
procedure MostrarVReales(const x: array of real; s:TStringGrid);  
var i,n:cardinal;  
begin  
  n:=Length(x);  
  s.RowCount:=n+1;  
  for i:=0 to n-1 do begin  
    s.Cells[0,i+1]:=Format('%12d',[i+1]);  
    s.Cells[1,i+1]:=Format('%20.2f',[x[i]]);  
  end;  
end;  
  
function LeerVReales(s:TStringGrid):tvReal;  
var i,n:cardinal; x:tvReal;  
begin  
  try  
    n:=s.RowCount-1;  
    SetLength(x,n);  
    for i:=0 to n-1 do x[i]:=StrToFloat(s.Cells[1,i+1]);  
    result:=x;  
  except  
    on EConvertError do begin  
      ShowMessage('Número real mal escrito');  
      s.SetFocus; s.Row:= i;  
    end;  
  end;  
end;  
  
function Mediapa(const x: array of real):real;  
var i,n:Cardinal; s:real;  
begin  
  n:=Length(x);  
  if n=0 then raise EInvalidArgument.Create(  
    'El vector a promediar está vacío');  
  s:=0;  
  for i:=0 to n-1 do s:=s+x[i];  
  result:= s/n;  
end;  
  
procedure MostrarReal(x:real; m:TMemo);  
begin  
  m.Text:= Format('%12.2f',[x]);
```

end;

end.

En la unidad principal (la unidad de la forma) añadimos en el sector público de la clase, los módulos que se ejecutarán en el evento "onClick" de los ToolButtons: "tb1Click", "tb2Click" y "tb3Click". Cambiamos las propiedades de los objetos en el evento "onCreate" de la forma y damos funcionalidad a la aplicación escribiendo el código de los procedimientos "tb1Click", "tb2Click" y "tb3click":

type

```
TForm1 = class(TForm)
  ToolBar1: TToolBar;
  StatusBar1: TStatusBar;
  StringGrid1: TStringGrid;
  Memo1: TMemo;
  ImageList1: TImageList;
  Label1: TLabel;
  procedure FormCreate(Sender: TObject);
private
  { Private declarations }
public
  procedure tb1Click(Sender:TObject);
  procedure tb2Click(Sender:TObject);
  procedure tb3Click(Sender:TObject);
end;
```

var

```
Form1: TForm1;
```

implementation

uses uParamAbiertos;

```
{ $R *.dfm }
```

```
procedure TForm1.FormCreate(Sender: TObject);
var tb:TToolButton; i:byte; bm:TBitmap; path:string;
const boton: array [1..3] of string=('Calculat.bmp',
  'BulbOn.bmp', 'DoorOpen.bmp');
  pista: array [1..3] of string=('Salir de la aplicación',
  'Generar vector', 'Calcular Promedio');
```

begin

```
Form1.Position:=poScreenCenter;
Form1.Color:=clInactiveCaptionText;
Form1.BorderStyle:=bsNone;
Form1.Width:= 300;
Form1.Height:=400;
for i:=1 to 3 do begin
  tb:=TToolButton.Create(ToolBar1);
  tb.Parent:=ToolBar1;
  tb.ImageIndex:= i-1;
  tb.Hint:= Pista[i];
end;
ToolBar1.Buttons[0].OnClick:=tb1Click;
ToolBar1.Buttons[1].OnClick:=tb2Click;
ToolBar1.Buttons[2].OnClick:=tb3Click;
ToolBar1.Images:=ImageList1;
```



```
ToolBar1.Cursor:=crHandPoint;
path:='C:\Archivos de programa\Archivos comunes\Borland Shared\
  Images\Buttons\';
bm:=TBitmap.Create;
for i:=1 to 3 do begin
  bm.LoadFromFile(path+boton[i]);
  ImageList1.Add(bm,Nil);
  ImageList1.Delete(i);
end;
bm.Free;
StatusBar1.SimplePanel:=True;
StatusBar1.SimpleText:='Cálculo del promedio';
StatusBar1.AutoHint:=True;
StringGrid1.ColCount:=2;
StringGrid1.ColWidths[0]:=60;
StringGrid1.ColWidths[1]:=120;
StringGrid1.Width:=StringGrid1.ColWidths[0]+StringGrid1.ColWidths[1]+25;
StringGrid1.Height:=300;
StringGrid1.Left:=(Form1.ClientWidth-StringGrid1.Width) div 2;
StringGrid1.Top:= 30;
StringGrid1.FixedCols:=1;
StringGrid1.FixedRows:=1;
StringGrid1.Cells[0,0]:=Format('%14s', ['N°']);
StringGrid1.Cells[1,0]:=Format('%20s', ['Valor']);
StringGrid1.Options:=StringGrid1.Options+[goThumbTracking];
Mem1.Width:=130;
Mem1.Height:=21;
Mem1.Top:=StringGrid1.Top+StringGrid1.Height+10;
Mem1.Alignment:=taRightJustify;
Mem1.Text:= '0';
Mem1.ReadOnly:=True;
Label1.Caption:='Promedio: ';
Label1.Transparent:=True;
Label1.Left:=(Form1.ClientWidth-Mem1.Width-Label1.Width-5) div 2;
Mem1.Left:=Label1.Left+Label1.Width+5;
Label1.Top:= Mem1.Top+3;
end;

procedure TForm1.tb1Click(Sender:TObject);
begin
  Close;
end;

procedure TForm1.tb2Click(Sender:TObject);
var n:integer; x:tvReal; s:String;
begin
  s:=InputBox('Generación de elementos','N° Elementos a generar:', '100');
  n:=StrToInt(s);
  x:=GenerarVReales(n);
  MostrarVReales(x,StringGrid1);
  Finalize(x);
end;

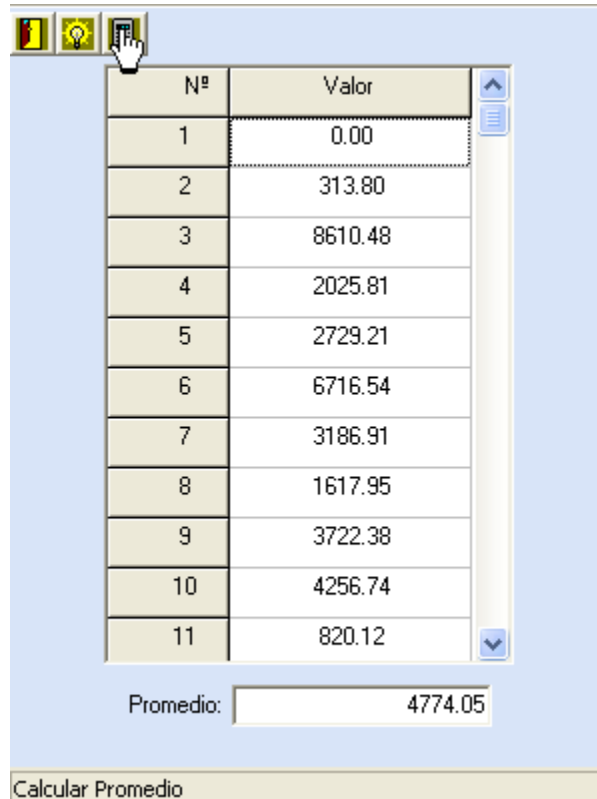
procedure TForm1.tb3Click(Sender:TObject);
var x:tvReal; p:real;
begin
  x:=LeerVReales(StringGrid1);
  p:=Mediapa(x);
```

```

MostrarReal(p,Memol);
Finalize(x);
end;

```

En ejecución, la aplicación tiene la siguiente apariencia:



- Como tercer ejemplo repetiremos el ejemplo anterior pero empleando sólo vectores dinámicos.

Los únicos módulos que varían con relación al ejemplo anterior son los que muestran el vector de números reales y el que calcula el promedio, razón por la cual son los únicos que se muestran:

```
unit uDinamicos;
```

```
interface
```

```
uses Math, QDialogs, SysUtils, StdCtrls, Grids;
```

```
type tvReal = array of real;
```

```
...
```

```
procedure MostrarVReales(x:tvReal; s:TStringGrid);
```

```
function Mediavd(x: tvReal):real;
```

```
...
```

```
implementation
```

```
...
```

```
procedure MostrarVReales(x:tvReal; s:TStringGrid);
```

```

var i,n:cardinal;
begin
  n:=Length(x);
  s.RowCount:=n+1;
  for i:=0 to n-1 do begin
    s.Cells[0,i+1]:=Format('%12d',[i+1]);
    s.Cells[1,i+1]:=Format('%20.2f',[x[i]]);
  end;
end;

. . .

function Mediavd(x: tvReal):real;
var i,n:Cardinal; s:real;
begin
  n:=Length(x);
  if n=0 then raise EInvalidArgument.Create(
    'El vector a promediar está vacío');
  s:=0;
  for i:=0 to n-1 do s:=s+x[i];
  result:= s/n;
end;

. . .

end.

```

Como se puede ver en estos módulos lo único que varía es la declaración de los parámetros. Sin embargo, estos módulos sólo funcionan con vectores dinámicos, mientras que los del anterior ejemplo funcionan tanto con vectores dinámicos como con estáticos, por lo que en realidad tienen un carácter más general.

El código de la unidad principal (y la interfaz de la aplicación) es el mismo que la del ejemplo anterior, lo único que cambia es que ahora se llama a "mediavd" en lugar de "mediapa", razón por la cual no se presenta dicho código.

4. Elabore una aplicación con un ToolBar, un StatusBar y un StringGrid. En la aplicación debe crear un módulo que determine si un número es o no primo, otro que genere un vector con los primeros "n" números primos y otro que muestre un vector de números enteros en un StringGrid con dos columnas, mostrando en la primera columna el número de elemento y en la segunda el número primo. La forma debe tener 250 puntos de ancho y 410 de alto, estar centrada en la pantalla, tener el fondo con líneas diagonales de color azul, el Hint "Números primos" y no tener bordes. El ToolBar debe tener dos botones con las imágenes en color de "DoorOpen.bmp" y "BulbOn.bmp", con los Hint: "Salir de la aplicación" y "Generar Números Primos", siendo la forma del puntero una mano apuntando. El StatusBar debe tener un solo panel con el texto "Números primos" y debe mostrar los Hint de manera automática. El StringGrid debe tener 300 puntos de alto, 2 columnas (de 60 y 80 puntos con una barra de desplazamiento vertical), estar centrado horizontalmente y a 30 puntos del margen superior, estando fijas una fila y una columna, los títulos de las columnas deben ser "Nº" y "Primo". Al hacer clic en el primer ToolButton la aplicación debe cerrarse, al hacer clic en el segundo ToolButton, se debe pedir el número de primos a generar, generar dicho número y mostrarlos en el StringGrid.

Los módulos que determinan, generan y muestran el vector de números primos son los siguientes:

```
unit uDinamicos;
```

```
interface
```

```
    uses Grids, SysUtils, Math;
```

```
    type tvCardinal = array of cardinal;
```

```
    function EsPrimo(n:cardinal):Boolean;
```

```
    function Primos(n:cardinal):tvCardinal;
```

```
    procedure MostrarPrimos(const x:array of cardinal; sg:TStringGrid);
```

```
implementation
```

```
    function EsPrimo(n:cardinal):Boolean;
```

```
    var i:cardinal;
```

```
    begin
```

```
        if n=0 then raise EInvalidArgument.Create(
            'El número debe ser mayor a cero');
```

```
        for i:=2 to n div 2 do
```

```
            if n mod i = 0 then begin
```

```
                result:=False; exit;
```

```
            end;
```

```
        result:=True;
```

```
    end;
```

```
    function Primos(n:cardinal):tvCardinal;
```

```
    var i,j:cardinal; x:tvCardinal;
```

```
    begin
```

```
        SetLength(x,n);
```

```
        i:=0; j:=1;
```

```
        while i<n do begin
```

```
            if EsPrimo(j) then begin
```

```
                x[i]:=j; inc(i);
```

```
            end;
```

```
            inc(j);
```

```
        end;
```

```
        result:=x;
```

```
    end;
```

```
    procedure MostrarPrimos(const x:array of cardinal; sg:TStringGrid);
```

```
    var i,n:cardinal;
```

```
    begin
```

```
        n:=Length(x);
```

```
        sg.RowCount:=n;
```

```
        for i:=0 to n-1 do begin
```

```
            sg.Cells[0,i+1]:=Format('%12d',[i+1]);
```

```
            sg.Cells[1,i+1]:=Format('%14d',[x[i]]);
```

```
        end;
```

```
    end;
```

```
end.
```

El código de la unidad principal (la unidad de la forma) es el siguiente:

```
unit ufPrimos;
```

interface**uses**

Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
Dialogs, ImgList, Grids, ComCtrls, ToolWin;

type

```
TForm1 = class(TForm)
    ToolBar1: TToolBar;
    StatusBar1: TStatusBar;
    StringGrid1: TStringGrid;
    ImageList1: TImageList;
    procedure FormCreate(Sender: TObject);
private
    { Private declarations }
public
    procedure tb1Click(sender: TObject);
    procedure tb2Click(sender: TObject);
end;
```

var

Form1: TForm1;

implementation

uses uDinamicos;

{ \$R *.dfm }

```
procedure TForm1.FormCreate(Sender: TObject);
var tb:tToolButton; i:byte; bm:tBitmap; path:string;
const botones: array [1..2] of string=('BulbOn.bmp','DoorOpen.bmp');
    hints: array [1..2] of string=('Generar Números Primos',
    'Salir de la aplicación');
begin
    Form1.Width:=250;
    Form1.Height:=410;
    Form1.Position:=poScreenCenter;
    Form1.Brush.Style:=bsFDiagonal;
    Form1.Brush.Color:=clBlue;
    Form1.Hint:='Números primos';
    Form1.BorderStyle:=bsNone;
    bm:=tBitmap.Create;
    path:='C:\Archivos de programa\Archivos comunes\Borland Shared\'+'
    'Images\Buttons\';
    for i:=1 to 2 do begin
        bm.LoadFromFile(Path+botones[i]);
        ImageList1.Add(bm,Nil);
        ImageList1.Delete(i);
    end;
    for i:=1 to 2 do begin
        tb:=tToolButton.Create(ToolBar1);
        tb.Parent:=ToolBar1;
        tb.Hint:=hints[i];
        tb.ImageIndex:=i-1;
        tb.Cursor:=crHandPoint;
    end;
    ToolBar1.Buttons[0].OnClick:=tb1Click;
```

```

ToolBar1.Buttons[1].OnClick:=tb2Click;
ToolBar1.Images:=ImageList1;
StatusBar1.SimplePanel:=True;
StatusBar1.SimpleText:='Números Primos';
StatusBar1.AutoHint:=True;
StringGrid1.Height:=350;
StringGrid1.ColCount:=2;
StringGrid1.ColWidths[0]:=60;
StringGrid1.ColWidths[1]:=80;
StringGrid1.Width:=StringGrid1.ColWidths[0]+StringGrid1.ColWidths[1]+25;
StringGrid1.Left:=(Form1.ClientWidth-StringGrid1.Width) div 2;
StringGrid1.Top:=35;
StringGrid1.FixedRows:=1;
StringGrid1.FixedCols:=1;
StringGrid1.Cells[0,0]:=Format('%12s', ['N°']);
StringGrid1.Cells[1,0]:=Format('%14s', ['Primo']);
StringGrid1.Options:=StringGrid1.Options+[goThumbTracking];
end;

procedure TForm1.tb1Click(sender: TObject);
begin
    Close;
end;

procedure TForm1.tb2Click(sender: TObject);
var n:cardinal; x:tvCardinal;
begin
    n:=StrToInt(InputBox('Generar Primos','N° de primos a generar:','100'));
    x:=Primos(n);
    MostrarPrimos(x,StringGrid1);
    Finalize(x);
end;

end.

```

En ejecución la aplicación tiene apariencia que se muestra en la figura de la siguiente página:

15.3. Ejercicios

1. Cree una aplicación con un Memo, un Edit y un BitBtn. En el evento "on-Create" de la forma haga que el memo tenga 100 puntos de ancho, 300 de alto y una barra de desplazamiento vertical. Elabore un módulo que empleando vectores dinámicos genere un vector con números enteros aleatorios comprendidos entre 1 y 10000 y otro que empleando parámetros abiertos muestre en el memo el vector generado. Programe el evento "onClick" del BitBtn de manera que genere y muestre un vector con el número de elementos escrito en el Edit.
2. Cree una aplicación con un ToolBar, un ImageList, un Memo y un StatusBar. Escriba un módulo que empleando vectores dinámicos genere un vector con "n" números pares y otro que empleando parámetros abiertos muestre un vector de números enteros en un memo. En el evento "onCreate" de la forma haga que la forma esté centrada en la pantalla y no tenga bordes, que el ToolBar tenga 3 botones, que el ImageList tenga las imágenes en color de "DoorOpen.bmp", "Retry.bmp" y "Clear.bmp", que las imágenes del ToolBar sean las del ImageList, que el Hint del primer ToolButton sea 'Salir de la aplicación', del segundo sea 'LlenarMemo' y

del tercero 'Borrar Memo'; que el Memo tenga 130 puntos de ancho, 250 de alto, que su fondo sea de color amarillo y tenga una barra de desplazamiento vertical, siendo su Hint 'Números enteros consecutivos'; que el StatusBar tenga un solo panel, que muestre automáticamente los hint de los objetos y que el texto inicial y Hint del mismo sea su nombre completo. Programe el evento "onClick" del ToolButton1 de manera que la aplicación se cierre, del ToolButton2, de manera que el memo se llene con 200 números pares y del ToolButton3 de manera que se borre el texto del memo.



3. Cree una aplicación con un Memo, dos BitBtn y dos Edit. Escriba un módulo recursivo que empleando parámetros abiertos calcule el promedio de un vector de números reales, otro que empleando vectores dinámicos genere un vector de números reales comprendidos entre 1 y 100, otro que empleando parámetros abiertos muestre en un memo un vector de números reales y otro que empleando vectores dinámicos lea de un memo un vector de números reales. En el evento "onCreate" de la forma haga que el memo tenga 120 puntos de ancho, 200 de alto, la letra sea de color azul y tenga una barra de desplazamiento vertical; el primer BitBtn tenga la imagen "Calculat.bmp", el título "&Calcular" y que al hacer clic sobre el mismo lea y calcule el promedio de los números del memo y se muestre el resultado en el Edit1; que el segundo BitBtn tenga la imagen "Retry.bmp", el título "&Generar" y que al hacer clic sobre el mismo se genere y muestre el número de elementos especificado en el Edit2.
4. Cree una aplicación con un StringGrid, dos BitBtn y un Edit. Escriba un módulo recursivo que empleando parámetros abiertos y vectores dinámicos reciba un vector de números enteros y devuelva un vector con los núme-

ros impares del vector recibido, otro que empleando vectores dinámicos genere un vector de números enteros comprendidos entre 1 y 1000, otro que empleando parámetros abiertos muestre en un StringGrid un vector de números enteros y otro que lea de un StringGrid un vector de números enteros. En el evento "onCreate" de la forma haga que el memo tenga 130 puntos de ancho, 250 de alto, la letra sea de color azul y tenga una barra de desplazamiento vertical; el primer BitBtn tenga la imagen "Report.bmp", el título "&Impares" y que al hacer clic sobre el mismo lea los elementos del memo y deje y muestre sólo los números impares; que el segundo BitBtn tenga la imagen "BulbOn.bmp", el título "&Generar" y que al hacer clic sobre el mismo se genere y muestre el número de elementos especificado en el Edit1.

5. Cree una aplicación con un StringGrid, un BitBtn y un Edit. Escriba un módulo que empleando vectores dinámicos reciba un número entero comprendido entre 1 y 50 y devuelva un vector con esa cantidad de números primos y otro que empleando parámetros abiertos muestre el vector en un StringGrid. En el evento "onCreate" de la forma haga que el StringGrid tenga dos columnas de 140 puntos de ancho y 200 de alto; el BitBtn tenga la imagen "Table.bmp", el título "&Primos" y que al hacer clic sobre el mismo genere y muestre la cantidad de números primos especificado en el Edit.

16. MATRICES DINÁMICAS

Los procedimientos empleados para vectores dinámicos pueden ser extendidos fácilmente para trabajar con matrices dinámicas, para ello simplemente se reserva memoria adicional para las otras dimensiones. Por ejemplo, en el siguiente segmento de código se crea la matriz dinámica "ma" con 10 filas y 20 columnas:

```
type tm= array of array of double;
var ma: tm;
begin
  SetLength(ma,10,20);
```

Al igual que ocurre con los vectores dinámicos, en las matrices dinámicas los primeros índices son siempre 0, así el primer elemento de la matriz "ma" es ma[0,0] y el último ma[9,19]. A diferencia de las matrices estáticas, las matrices dinámicas pueden no ser rectangulares, es decir pueden tener diferentes números de elementos en cada una de sus filas. Por ejemplo en el siguiente segmento de código se crea, llena y muestra una matriz que tiene una columna en la primera fila, dos en la segunda, tres en la tercera y así sucesivamente hasta la vigésima fila:

```
type treal= array of array of real;

procedure TForm1.FormCreate(Sender:Tobject);
var ma: treal; i,j: byte;
begin
  SetLength(ma,20);
  for i:=0 to 19 do
    SetLength(ma[i],i+1);
  for i:=0 to 19 do
    for j:=0 to i do ma[i,j]:=i+1;
  Memo1.Lines.Clear;
  for i:=0 to 19 do begin
    Memo1.Lines.Append('');
    for j:=0 to i do
      Memo1.Lines[i]:=Memo1.Lines[i]+FloatToStr(ma[i,j])+ ' ';
  end;
```

Para liberar la memoria se sigue empleando el procedimiento Finalize (Finalize(Variable)):

16.1. Ejemplos

En los siguientes ejemplos las interfaces gráficas de las aplicaciones no tienen componentes nuevos con relación a los del anterior tema, razón por la cual no se darán mayores explicaciones con relación a las mismas.

1. Elabore una aplicación con un ToolBar, un StatusBar, seis paneles, seis Labels, seis ComboBox, tres StringGrids y un ImageList. En la aplicación escriba el código de los siguientes módulos: uno que muestre los elementos de una matriz dinámica de números reales en un StringGrid, otro que lea los elementos de una matriz dinámica de números reales desde un StringGrid, otro que genere una matriz dinámica con "m" filas y "n" columnas de números reales aleatorios comprendidos entre dos límites dados y otro que multiplique dos matrices dinámicas de números reales. La forma debe tener 700 puntos de ancho, 500 de alto, estar centrada en la pantalla y no tener bordes. El ToolBar debe tener cuatro botones con las figuras de "Opendoor.bmp", "Retry.bmp", "Calculat.bmp"

y "Clear.bmp", con los hints: "Salir de la aplicación", "Generar Matriz", "Multiplicar Matrices" y "Borrar Matriz". El primer panel debe estar alineado a la izquierda y tener un ancho igual a la mitad del ancho disponible en la **forma**, el segundo panel debe estar alineado a la derecha y tener el mismo ancho que el primero, el tercer panel debe estar alineado al fondo y tener un alto igual a la mitad del alto disponible en la **forma** (sin contar el ToolBar y el StatusBar). El StatusBar debe tener un solo panel con el texto "Multiplicación de matrices" y en el mismo se deben mostrar automáticamente los hint de los objetos. En el primer panel se debe insertar el cuarto panel alineado en la parte superior, con 30 puntos de alto, el título "Matriz A" alineado a la izquierda y en el mismo se deben insertar los **dos** primeros Labels con los títulos "Filas:" y "Columnas" y los **dos** primeros ComboBox con 60 puntos de ancho y los números del 1 al 500, los Label y los ComboBox deben estar centrados verticalmente en ese panel y los Labels deben estar al lado de los ComboBox respectivos, en el panel se debe insertar el primer StringGrid que debe estar alineado en todo el panel, tener columnas de 60 puntos de ancho, el Hint "Matriz A" y no tener filas ni columnas fijas. En el segundo panel se debe proceder igual que en el primero, sólo que **ahora** se inserta el quinto panel con el título "Matriz B", los Labels 3 y 4, los ComboBoxs 3 y 4 y el StringGrid 2 con el hint "Matriz B". En el tercer panel se procede igual que en el primero sólo que se inserta el sexto panel con el título "Matriz C", los Labels 5y 6, los ComboBox 5 y 6 y el StringGrid3 con el hint "Matriz C". Al hacer clic sobre el primer ToolButton la aplicación debe cerrarse, al hacer clic sobre el segundo ToolButton se debe generar una matriz con el número de filas y columnas especificado en los ComboBoxs, al hacer clic sobre el tercer ToolButton se deben multiplicar las matrices "A", "B" y mostrar la matriz resultante "C" y al hacer clic sobre el cuarto ToolButton se deben **borrar** las matrices **dejando** una sola fila y columna.

El código de los módulos **donde** se generan, leen, muestran, **borran** y multiplican las matrices es el siguiente:

```
unit uDinamicos;
```

```
interface
```

```
    uses Grids, SysUtils, QDialogs;
```

```
    type tmReal = array of array of real;
```

```
    procedure MostrarMatriz(a:tmReal; sg:TStringGrid);
```

```
    function LeerMatriz(sg:TStringGrid):tmReal;
```

```
    function GenerarMatriz(m,n:cardinal; li,ls:real):tmReal;
```

```
    function MulMat(a,b:tmReal):tmReal;
```

```
    procedure BorrarStringGrid(sg:TStringGrid);
```

```
implementation
```

```
    procedure MostrarMatriz(a:tmReal; sg:TStringGrid);
```

```
    var i,j,m,n:cardinal;
```

```
    begin
```

```
        m:=Length(a);
```

```
        n:=Length(a[0]);
```

```
        sg.RowCount:=m;
```

```
        sg.ColCount:=n;
```

```
        for i:=0 to m-1 do
```

```
            for j:=0 to n-1 do
```

```

        sg.Cells[j,i]:=Format('%10.2f',[a[i,j]]);
    end;

function LeerMatriz(sg:TStringGrid):tmReal;
var i,j,m,n:cardinal; a:tmReal;
begin
    try
        m:=sg.RowCount;
        n:=sg.ColCount;
        SetLength(a,m,n);
        for i:=0 to m-1 do
            for j:=0 to n-1 do
                a[i,j]:=StrToFloat(sg.Cells[j,i]);
            result:=a;
        except
            on EConvertError do begin
                ShowMessage('Número real mal escrito');
                sg.SetFocus; sg.Row:=i; sg.Col:=j;
            end;
        end;
    end;

function GenerarMatriz(m,n:cardinal; li,ls:real):tmReal;
var i,j:cardinal; a:tmReal; d:real;
begin
    SetLength(a,m,n);
    d:=ls-li;
    for i:=0 to m-1 do
        for j:=0 to n-1 do
            a[i,j]:=random*d+li;
        result:=a;
    end;

function MulMat(a,b:tmReal):tmReal;
var i,j,k,nfa,nca,nfb,ncb:cardinal; c:tmReal;
begin
    nfa:=Length(a); nca:=Length(a[0]);
    nfb:=Length(b); ncb:=Length(b[0]);
    if (nfa=0) or (nfb=0) then raise EInvalidOp.Create(
        'La matriz a multiplicar está vacía');
    if nca<>nfb then raise EInvalidOp.Create(
        'Las matrices no son multiplicables');
    SetLength(c,nfa,ncb);
    for i:=0 to nfa-1 do
        for j:=0 to ncb-1 do begin
            c[i,j]:=0;
            for k:=0 to nca-1 do c[i,j]:=c[i,j]+a[i,k]*b[k,j];
        end;
    result:=c;
end;

procedure BorrarStringGrid(sg:TStringGrid);
begin
    sg.RowCount:=1;
    sg.ColCount:=1;
    sg.Cells[0,0]:=Format('%10.2f',[0.0]);
end;

```

end.

El código de la unidad principal (la unidad de la **forma**) **donde** se modifican las propiedades de los componentes y se programan los evento "onClick" de los ToolButton es el siguiente:

unit ufMulMat;

interface

uses

Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, **Forms**, Dialogs, ImgList, Grids, ExtCtrls, ComCtrls, ToolWin, StdCtrls;

type

```
TForm1 = class(TForm)
  ToolBar1: TToolBar;
  StatusBar1: TStatusBar;
  Panel1: TPanel;
  Panel2: TPanel;
  Panel3: TPanel;
  StringGrid1: TStringGrid;
  StringGrid2: TStringGrid;
  StringGrid3: TStringGrid;
  ImageList1: TImageList;
  Panel4: TPanel;
  Panel5: TPanel;
  Panel6: TPanel;
  Label1: TLabel;
  Label2: TLabel;
  Label3: TLabel;
  Label4: TLabel;
  Label5: TLabel;
  Label6: TLabel;
  ComboBox1: TComboBox;
  ComboBox2: TComboBox;
  ComboBox3: TComboBox;
  ComboBox4: TComboBox;
  ComboBox5: TComboBox;
  ComboBox6: TComboBox;
  procedure FormCreate(Sender: TObject);
private
  { Private declarations }
public
  procedure tb1Click(sender: TObject);
  procedure tb2Click(sender: TObject);
  procedure tb3Click(sender: TObject);
  procedure tb4Click(sender: TObject);
end;
```

var

Form1: TForm1;

implementation

uses uDinamicos;

{ \$R *.dfm }

```
procedure TForm1.FormCreate(Sender: TObject);
var tb:tToolButton; bm:tBitmap; path:string; i:word;
const images: array [1..4] of string = ('Clear.bmp',
    'Calculat.bmp', 'Retry.bmp', 'DoorOpen.bmp');
    hints: array [1..4] of string = ('Borrar Matriz',
    'Multiplicar Matrices', 'Generar Matriz', 'Salir de la aplicación');
begin

    Form1.Width:=700;
    Form1.Height:=500;
    Form1.Position:=poScreenCenter;
    Form1.BorderStyle:=bsNone;

    bm:=tBitmap.Create;
    path:='C:\Archivos de programa\Archivos comunes\Borland Shared\'+'
        'Images\Buttons\';
    for i:=1 to 4 do begin
        bm.LoadFromFile(path+images[i]);
        ImageList1.Add(bm,Nil);
        ImageList1.Delete(i);
    end;
    bm.Free;

    for i:=1 to 4 do begin
        tb:=tToolButton.Create(ToolBar1);
        tb.Hint:=hints[i];
        tb.ImageIndex:=i-1;
        tb.Cursor:=crHandPoint;
        tb.Parent:=ToolBar1;
    end;
    ToolBar1.Images:=ImageList1;
    ToolBar1.Buttons[0].OnClick:=tb1Click;
    ToolBar1.Buttons[1].OnClick:=tb2Click;
    ToolBar1.Buttons[2].OnClick:=tb3Click;
    ToolBar1.Buttons[3].OnClick:=tb4Click;

    Panel1.Align:=alLeft;
    Panel1.Caption:='';
    Panel1.Width:=Form1.ClientWidth div 2;
    Panel2.Align:=alRight;
    Panel2.Caption:='';
    Panel2.Width:=Panel1.Width;
    Panel3.Width:=Form1.ClientWidth;
    Panel3.Height:=(Form1.ClientHeight-ToolBar1.Height-StatusBar1.Height)
        div 2;

    StatusBar1.SimplePanel:=True;
    StatusBar1.SimpleText:='Multiplicación de Matrices';
    StatusBar1.AutoHint:=True;
    Panel3.Align:=alBottom;

    Panel4.Parent:=Panel1;
    Panel4.Caption:='Matriz A';
    Panel4.Alignment:=taLeftJustify;
    Panel4.Align:=alTop;
    Panel4.Height:=30;

    Panel5.Parent:=Panel2;
```

```
Panel5.Caption:='Matriz B';
Panel5.Alignment:=taLeftJustify;
Panel5.Align:=alTop;
Panel5.Height:=30;

Panel6.Parent:=Panel3;
Panel6.Caption:='Matriz c';
Panel6.Alignment:=taLeftJustify;
Panel6.Align:=alTop;
Panel6.Height:=30;

Label1.Parent:=Panel4;
Label1.Caption:='Filas: ';
Label1.Left:=70;
Label1.Top:=(Panel4.Height-Label1.Height) div 2;
Label2.Parent:=Panel4;
Label2.Caption:='Columnas: ';
Label2.Left:=170;
Label2.Top:=Label1.Top;

Label3.Parent:=Panel5;
Label3.Caption:='Filas: ';
Label3.Left:=70;
Label3.Top:=Label1.Top;
Label4.Parent:=Panel5;
Label4.Caption:='Columnas: ';
Label4.Left:=170;
Label4.Top:=Label1.Top;

Label5.Parent:=Panel6;
Label5.Caption:='Filas: ';
Label5.Left:=70;
Label5.Top:=Label1.Top;
Label6.Parent:=Panel6;
Label6.Caption:='Columnas: ';
Label6.Left:=170;
Label6.Top:=Label1.Top;

ComboBox1.Parent:=Panel4;
ComboBox1.Width:=60;
for i:=1 to 500 do ComboBox1.Items.Append(IntToStr(i));
ComboBox1.ItemIndex:=0;
ComboBox1.Left:=Label1.Left+Label1.Width+5;
ComboBox1.Top:=(Panel4.Height-ComboBox1.ClientHeight) div 2;
ComboBox1.Style:=csDropDownList;
ComboBox2.Parent:=Panel4;
ComboBox2.Width:=60;
ComboBox2.Items.Assign(ComboBox1.Items);
ComboBox2.ItemIndex:=0;
ComboBox2.Left:=Label2.Left+Label2.Width+5;
ComboBox2.Top:=ComboBox1.Top;
ComboBox2.Style:=csDropDownList;

ComboBox3.Parent:=Panel5;
ComboBox3.Width:=60;
ComboBox3.Items.Assign(ComboBox1.Items);
ComboBox3.ItemIndex:=0;
ComboBox3.Left:=Label3.Left+Label3.Width+5;
```

```
ComboBox3.Top:=ComboBox1.Top;
ComboBox3.Style:=csDropDownList;
ComboBox4.Parent:=Panel5;
ComboBox4.Width:=60;
ComboBox4.Items.Assign(ComboBox1.Items);
ComboBox4.ItemIndex:=0;
ComboBox4.Left:=Label4.Left+Label4.Width+5;
ComboBox4.Top:=ComboBox1.Top;
ComboBox4.Style:=csDropDownList;

ComboBox5.Parent:=Panel6;
ComboBox5.Width:=60;
ComboBox5.Items.Assign(ComboBox1.Items);
ComboBox5.ItemIndex:=0;
ComboBox5.Left:=Label5.Left+Label5.Width+5;
ComboBox5.Top:=ComboBox1.Top;
ComboBox5.Style:=csDropDownList;
ComboBox6.Parent:=Panel6;
ComboBox6.Width:=60;
ComboBox6.Items.Assign(ComboBox1.Items);
ComboBox6.ItemIndex:=0;
ComboBox6.Left:=Label6.Left+Label6.Width+5;
ComboBox6.Top:=ComboBox1.Top;
ComboBox6.Style:=csDropDownList;

StringGrid1.Parent:=Panel1;
StringGrid1.FixedCols:=0;
StringGrid1.FixedRows:=0;
StringGrid1.Options:=StringGrid1.Options+[goThumbTracking,goEditing];
StringGrid1.DefaultColWidth:=60;
StringGrid1.RowCount:=1;
StringGrid1.ColCount:=1;
StringGrid1.Align:=alClient;
StringGrid1.Hint:='Matriz A';

StringGrid2.Parent:=Panel2;
StringGrid2.FixedCols:=0;
StringGrid2.FixedRows:=0;
StringGrid2.Options:=StringGrid1.Options+[goThumbTracking,goEditing];
StringGrid2.DefaultColWidth:=60;
StringGrid2.RowCount:=1;
StringGrid2.ColCount:=1;
StringGrid2.Align:=alClient;
StringGrid2.Hint:='Matriz B';

StringGrid3.Parent:=Panel3;
StringGrid3.FixedCols:=0;
StringGrid3.FixedRows:=0;
StringGrid3.Options:=StringGrid1.Options+[goThumbTracking,goEditing];
StringGrid3.DefaultColWidth:=60;
StringGrid3.RowCount:=1;
StringGrid3.ColCount:=1;
StringGrid3.Align:=alClient;
StringGrid3.Hint:='Matriz C';
```

end;

procedure TForm1.tb1Click(sender: TObject);

```

begin
    Close;
end;

procedure TForm1.tb2Click(sender: TObject);
var nfa,nca,nfb,ncb:cardinal; a,b:tmReal;
begin
    nfa:=ComboBox1.ItemIndex+1;
    nca:=ComboBox2.ItemIndex+1;
    a:=GenerarMatriz(nfa,nca,0,1000);
    MostrarMatriz(a,StringGrid1);
    nfb:=ComboBox3.ItemIndex+1;
    ncb:=ComboBox4.ItemIndex+1;
    b:=GenerarMatriz(nfb,ncb,0,1000);
    MostrarMatriz(b,StringGrid2);
    Finalize(a); Finalize(b);
end;

```

Matriz A Filas: 3 Columnas: 4

0.00	31.38	861.05	202.58
272.92	671.65	318.69	161.80
372.24	425.67	82.01	474.79

Matriz B Filas: 4 Columnas: 2

70.57	840.85
59.72	293.30
917.28	367.91
774.66	327.93

Matriz c Filas: 3 Columnas: 2

948628.58	392424.72
477038.85	596788.04
494716.94	623717.20

Multiplicar Matrices

```

procedure TForm1.tb3Click(sender: TObject);
var a,b,c:tmReal;
begin
    a:=LeerMatriz(StringGrid1);
    b:=LeerMatriz(StringGrid2);
    c:=MulMat(a,b);
    MostrarMatriz(c,StringGrid3);
    ComboBox5.ItemIndex:=Length(a)-1;
    ComboBox6.ItemIndex:=Length(b[0])-1;

```



```

    Finalize(a); Finalize(b); Finalize(c);
end;

procedure TForm1.tb4Click(sender: TObject);
begin
    BorrarStringGrid(StringGrid1);
    BorrarStringGrid(StringGrid2);
    BorrarStringGrid(StringGrid3);
end;

end.

```

En ejecución la aplicación tiene la apariencia que se muestra en la figura de la anterior página.

- Un cuadrado mágico es una matriz cuadrada con un número impar de filas y columnas, que cumple con la condición de que la suma de sus filas y columnas, así como la de las **dos** diagonales principales es la misma. Por ejemplo, el siguiente es un cuadrado mágico de 5 filas y 5 columnas:

15	8	1	24	17
16	14	7	5	23
22	20	13	6	4
3	21	19	12	10
9	2	25	18	11

El algoritmo para construir estos cuadrados, lo popularizó S. de la Loubere en 1687, y su regla de generación es: iniciar en la celda central de la primera fila, entonces subir a la izquierda diagonalmente (como si estuviera enrollando) hasta asignar "n" celdas:

		1		
			5	
				4
3				
	2			

A continuación bajar una fila en la misma columna del último valor llenado y continuar, como en el primer paso, hasta completar la tabla:

	8	1		
		7	5	
			6	4
3				10
9	2			

⇒

15	8	1		17
	14	7	5	
		13	6	4
3			12	10
9	2			11

15	8	1		17
16	14	7	5	
	20	13	6	4
3		19	12	10
9	2		18	11

⇒

15	8	1	24	17
16	14	7	5	23
22	20	13	6	4
3	21	19	12	10
9	2	25	18	11

Elabore una aplicación con un módulo que, empleando matrices dinámicas, genere un cuadrado mágico con el número de filas (y columnas) especificado. La aplicación constará de 2 Panel, 1 StringGrid, 1 ComboBox y 1 Label, cuyas propiedades serán fijadas en el evento "onCreate" de la forma. El cuadrado debe ser generado al elegir el número de filas y columnas en el ComboBox.

La unidad **donde** se programa el módulo que genera el cuadrado mágico, así como el módulo que muestra el cuadrado en un StringGrid es la siguiente:

```
unit Unit2;
```

```
interface
```

```
    uses Grids, SysUtils;
```

```
    type tm= array of array of word;
```

```
    function CuadradoMagico(n:word):tm;
```

```
    procedure MostrarCuadrado(a:tm; s:tStringGrid);
```

```
implementation
```

```
    function CuadradoMagico(n:word):tm;
```

```
    var a:tm; i,j,k: word;
```

```
    begin
```

```
        if not odd(n) then raise EInvalidOp.Create(
```

```
            'El número de filas y columnas debe ser impar');
```

```
        SetLength(a,n,n);
```

```
        i:=0;
```

```
        j:=n div 2;
```

```
        for k:=1 to sqr(n) do begin
```

```
            a[i,j]:=k;
```

```
            if k mod n=0 then
```

```
                if i=n-1 then
```

```
                    i:=0
```

```
                else
```

```
                    inc(i)
```

```
            else begin
```

```
                if i>0 then
```

```
                    dec(i)
```

```
                else
```

```
                    i:=n-1;
```

```
                if j>0 then
```

```
                    dec(j)
```

```
                else
```

```
                    j:=n-1;
```

```
            end;
```

```
        end;
```

```
        result:=a;
```

```
    end;
```

```
    procedure MostrarCuadrado(a:tm; s:tStringGrid);
```

```
    var i,j,n: byte;
```

```
    begin
```

```
        n:=Length(a);
```

```
        s.RowCount:=n;
```

```
        s.ColCount:=n;
```

```
        for i:=0 to n-1 do
```

```
            for j:=0 to n-1 do
```

```
                s.Cells[j,i]:=IntToStr(a[i,j]);
```

```
    end;
```

```
end.
```

El código de los eventos "onCreate" de la **forma** y "onChange" del "ComboBox1" son los siguientes:

```
procedure TForm1.FormCreate(Sender: TObject);
var i,k:word;
begin

    Form1.Position:=poScreenCenter;
    Form1.BorderStyle:=bsDialog;
    Form1.Caption:='Cuadrados Mágicos';

    Panel1.Caption:='';
    Panel1.Align:=alTop;
    Panel1.Height:=40;
    Panel1.Color:=clInactiveCaptionText;

    Label1.Parent:=Panel1;
    Label1.Caption:='Número de filas y columnas: ';
    Label1.Transparent:=True;
    Label1.Top:=(Panel1.ClientHeight-Label1.Height) div 2;
    Label1.Left:=10;

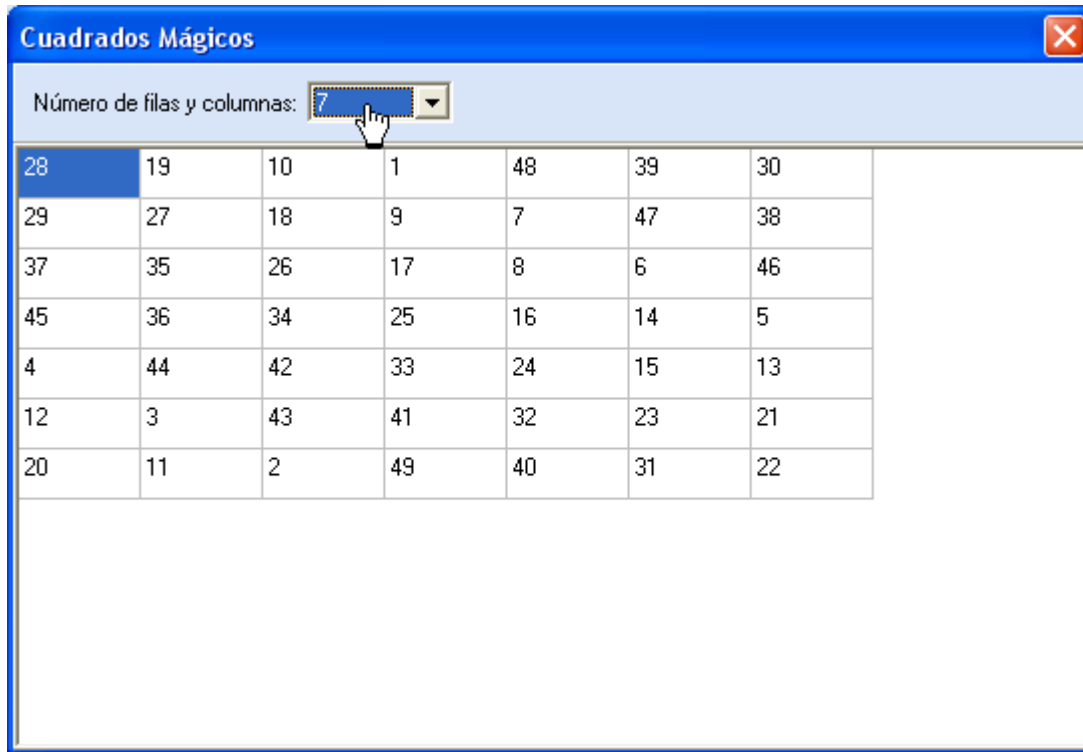
    ComboBox1.Parent:=Panel1;
    ComboBox1.Top:=(Panel1.ClientHeight-ComboBox1.Height) div 2;
    ComboBox1.Left:=Label1.Left+Label1.Width+5;
    ComboBox1.Style:=csDropDownList;
    ComboBox1.Cursor:=crHandPoint;
    k:=3;
    ComboBox1.Items.Clear;
    for i:=1 to 50 do begin
        ComboBox1.Items.Append(IntToStr(k));
        inc(k,2);
    end;
    ComboBox1.ItemIndex:=0;
    ComboBox1.Change(Self);

    Panel2.Align:=alLeft;
    Panel2.Width:=Form1.ClientWidth;
    Panel2.Caption:='';
    StringGrid1.Parent:=Panel2;
    StringGrid1.RowCount:=3;
    StringGrid1.ColCount:=3;
    StringGrid1.FixedCols:=0;
    StringGrid1.FixedRows:=0;
    StringGrid1.DefaultColWidth:=60;
    StringGrid1.Align:= alClient;

end;

procedure TForm1.ComboBox1Change(Sender: TObject);
var n:word; a:tm;
begin
    n:=StrToInt(ComboBox1.Text);
    a:=CuadradoMagico(n);
    MostrarCuadrado(a,StringGrid1);
    Finalize(a);
end;
```

En ejecución la aplicación tiene la apariencia que se muestra en la siguiente figura:



3. En una aplicación se requieren módulos para intercambiar los elementos de las filas y columnas de una matriz de números enteros. Elabore una aplicación con **dos** módulos que lleven a cabo dicha operación. La aplicación debe permitir generar una matriz con números enteros aleatorios, mostrar y leer la matriz de un StringGrid, intercambiar las filas o columnas en función a la opción elegida en un StringGrid y los números de filas (o columnas) elegidos de 2 ComboBox.

Todas las propiedades de los componentes de esta aplicación: 2 Panel, 2 BitBtn, 2 Label, 2 ComboBox, 1 RadioGroup y 1 StringGrid se fijan en el evento "onCreate" de la forma "Form1". La forma "Form2" donde se introducen los datos para generar la matriz de números enteros se crea dinámicamente en el módulo "GenerarMatriz", por lo que no debe ser añadida manualmente al proyecto.

El código de la unidad donde se han escrito todos los módulos requeridos (incluido el módulo para genera la matriz) es el siguiente:

```
unit Unit2;
```

```
interface
```

```
uses Forms, StdCtrls, Buttons, Grids, Controls, SysUtils, Math;
```

```
type tmInteger = array of array of integer;
```

```
function LeerMatriz(s:tStringGrid):tmInteger;
```

```
procedure MostrarMatriz(a:tmInteger; s:tStringGrid);
```

```
function MatrizAleatoria(nf,nc:word; li,ls:integer):tmInteger;
```

```
function GenerarMatriz():tmInteger;
```

```
procedure IntercambiarFilas(a:tmInteger;f1,f2:word);
```

```
procedure IntercambiarColumnas(a:tmInteger;c1,c2:word);

implementation

uses unit1;

function LeerMatriz(s:tStringGrid):tmInteger;
var nf,nc,i,j: word; a:tmInteger;
begin
    nf:=s.RowCount;
    nc:=s.ColCount;
    SetLength(a,nf,nc);
    for i:=0 to nf-1 do
        for j:=0 to nc-1 do
            a[i,j]:=StrToInt(s.Cells[j,i]);
    result:=a;
end;

procedure MostrarMatriz(a:tmInteger; s:tStringGrid);
var nf,nc,i,j: word;
begin
    nf:=Length(a);
    nc:=Length(a[0]);
    s.RowCount:=nf;
    s.ColCount:=nc;
    for i:=0 to nf-1 do
        for j:=0 to nc-1 do
            s.Cells[j,i]:=IntToStr(a[i,j]);
end;

function MatrizAleatoria(nf,nc:word; li,ls:integer):tmInteger;
var i,j: word; d:integer; a:tmInteger;
begin
    randomize;
    SetLength(a,nf,nc);
    d:=ls-li+1;
    for i:=0 to nf-1 do
        for j:=0 to nc-1 do
            a[i,j]:=random(d)+li;
    result:=a;
end;

function GenerarMatriz():tmInteger;
var Label1,Label2,Label3,Label4:tLabel;
    Edit1,Edit2,Edit3,Edit4:tEdit;
    BitBtn1,BitBtn2:tBitBtn;
    Form2:tForm;
    li,ls:integer;
    nf,nc:word;
begin
    try

        Form2:=TForm.Create(Form1);
        Form2.BorderStyle:=bsDialog;
        Form2.Position:=poScreenCenter;
        Form2.Caption:='Generar números enteros';

        Label1:=TLabel.Create(Form2);
```

```

Label1.Parent:=Form2;
Label1.Left:=50;
Label1.Top:=20;
Label1.Caption:='Límite inferior: ';

Edit1:=TEdit.Create(Form2);
Edit1.Parent:=Form2;
Edit1.Left:=Label1.Left+Label1.Width+5;
Edit1.Top:=Label1.Top-(Edit1.Height-Label1.Height) div 2;
Edit1.Text:='1';

Label2:=tLabel.Create(Form2);
Label2.Parent:=Form2;
Label2.Caption:='Límite superior: ';
Label2.Left:=Label1.Left+Label1.Width-Label2.Width;
Label2.Top:=Label1.Top+Label1.Height+20;

Edit2:=tEdit.Create(Form2);
Edit2.Parent:=Form2;
Edit2.Left:=Edit1.Left;
Edit2.Top:=Label2.Top-(Edit2.Height-Label2.Height) div 2;
Edit2.Text:='100';

Label3:=tLabel.Create(Form2);
Label3.Parent:=Form2;
Label3.Caption:='Número de filas: ';
Label3.Left:=Label1.Left+Label1.Width-Label3.Width;
Label3.Top:=Label2.Top+Label2.Height+20;

Edit3:=tEdit.Create(Form2);
Edit3.Parent:=Form2;
Edit3.Left:=Edit1.Left;
Edit3.Top:=Label3.Top-(Edit3.Height-Label3.Height) div 2;
Edit3.Text:='6';

label4:=tLabel.Create(Form2);
Label4.Parent:=Form2;
Label4.Caption:='Número de columnas: ';
Label4.Top:=Label3.Top+Label3.Height+20;
Label4.Left:=Label1.Left+Label1.Width-Label4.Width;

Edit4:=TEdit.Create(Form2);
Edit4.Parent:=Form2;
Edit4.Left:=Edit1.Left;
Edit4.Top:=Label4.Top-(Edit4.Height-Label4.Height) div 2;
Edit4.Text:='7';

Form2.ClientWidth:=Edit4.Left+Edit4.Width+20;

BitBtn1:=tBitBtn.Create(Form2);
BitBtn1.Parent:=Form2;
BitBtn1.Kind:=bkOk;
BitBtn1.Cursor:=crHandPoint;
BitBtn1.Width:=80;
BitBtn1.Caption:='&Aceptar';
BitBtn1.Top:=Edit4.Top+Edit4.Height+20;
BitBtn1.Left:=(Form2.ClientWidth-(BitBtn1.Width*2+30)) div 2;

```

```
    BitBtn2:=tBitBtn.Create(Form2);
    BitBtn2.Parent:=Form2;
    BitBtn2.Kind:=bkCancel;
    BitBtn2.Cursor:=crHandPoint;
    BitBtn2.Width:=BitBtn1.Width;
    BitBtn2.Caption:='&Cancelar';
    BitBtn2.Top:=BitBtn1.Top;
    BitBtn2.Left:=BitBtn1.Left+BitBtn1.Width+30;

    Form2.ClientHeight:=BitBtn1.Top+BitBtn1.Height+20;

    if Form2.ShowModal=mrOk then begin
        li:=StrToInt(Edit1.Text);
        ls:=StrToInt(Edit2.Text);
        nf:=StrToInt(Edit3.Text);
        nc:=StrToInt(Edit4.Text);
        result:=MatrizAleatoria(nf,nc,li,ls); end
    else
        result:=nil;

    finally
        Form2.Free;
    end;
end;

procedure IntercambiarFilas(a:tmInteger;f1,f2:word);
var nf,nc,j: word; aux:integer;
begin
    nf:=Length(a);
    nc:=Length(a[0]);
    if f1>nf then raise EInvalidArgument.Create(
        'En la matriz no existe la fila '+IntToStr(f1));
    if f2>nf then raise EInvalidArgument.Create(
        'En la matriz no existe la fila '+IntToStr(f2));
    for j:=0 to nc-1 do begin
        aux:=a[f1,j];
        a[f1,j]:=a[f2,j];
        a[f2,j]:=aux;
    end;
end;

procedure IntercambiarColumnas(a:tmInteger;c1,c2:word);
var nf,nc,i:word; aux:integer;
begin
    nf:=Length(a);
    nc:=Length(a[0]);
    if c1>nc then raise EInvalidArgument.Create(
        'En la matriz no existe la columna: '+IntToStr(c1));
    if c2>nc then raise EInvalidArgument.Create(
        'En la matriz no existe la columna: '+IntToStr(c2));
    for i:=0 to nf-1 do begin
        aux:=a[i,c1];
        a[i,c1]:=a[i,c2];
        a[i,c2]:=aux;
    end;
end;

end.
```

Y los eventos escritos en la unidad principal (el de la forma **Form1**) son:

```
uses Unit2;

procedure TForm1.FormCreate(Sender: TObject);
var i: word;
begin

    Form1.Caption:='Intercambiar Filas y Columnas';
    Form1.BorderStyle:=bsDialog;
    Form1.Position:=poScreenCenter;

    Panel1.Caption:='';
    Panel1.Align:=alTop;
    Panel1.Height:=40;
    Panel1.BevelInner:=bvLowered;

    BitBtn1.Parent:=Panel1;
    BitBtn1.Width:=100;
    BitBtn1.NumGlyphs:=2;
    BitBtn1.Cursor:=crHandPoint;
    BitBtn1.Kind:=bkRetry;
    BitBtn1.Caption:='&Generar';
    BitBtn1.Left:=20;
    BitBtn1.Top:=(Panel1.Height-BitBtn1.Height) div 2;

    RadioGroup1.Parent:=Panel1;
    RadioGroup1.Color:=Panel1.Color;
    RadioGroup1.Columns:=2;
    RadioGroup1.Width:=150;
    RadioGroup1.Height:=30;
    RadioGroup1.Caption:='Intercambiar: ';
    RadioGroup1.Items.Add('Filas');
    RadioGroup1.Items.Add('Columnas');
    RadioGroup1.Left:=BitBtn1.Left+BitBtn1.Width+20;
    RadioGroup1.Top:=(Panel1.Height-RadioGroup1.Height) div 2;
    RadioGroup1.ItemIndex:=0;

    Label1.Parent:=Panel1;
    Label1.Transparent:=True;
    Label1.Alignment:=taRightJustify;
    Label1.Caption:='Fila 1: ';
    Label1.Left:=RadioGroup1.Left+RadioGroup1.Width+40;
    Label1.Top:=(Panel1.Height-Label1.Height) div 2;

    ComboBox1.Width:=40;
    ComboBox1.Style:=csDropDownList;
    ComboBox1.Cursor:=crHandPoint;
    ComboBox1.Items.Clear;
    for i:=1 to 100 do
        ComboBox1.Items.Append(IntToStr(i));
    ComboBox1.Top:=(Panel1.Height-ComboBox1.Height) div 2;
    ComboBox1.Left:=Label1.Left+Label1.Width+5;
    ComboBox1.ItemIndex:=2;

    Label2.Parent:=Panel1;
    Label2.Transparent:=True;
    Label2.Alignment:=taRightJustify;
    Label2.Caption:='Fila 2:';
```



```
Label2.Left:=ComboBox1.Left+ComboBox1.Width+40;
Label2.Top:=Label1.Top;

ComboBox2.Width:=40;
ComboBox2.Style:=csDropDownList;
ComboBox2.Cursor:=crHandPoint;
ComboBox2.Items.Assign(ComboBox1.Items);
ComboBox2.Top:= ComboBox1.Top;
ComboBox2.Left:=Label2.Left+Label2.Width+5;
ComboBox2.ItemIndex:=4;

BitBtn2.Parent:=Panel1;
BitBtn2.Width:=100;
BitBtn2.NumGlyphs:=2;
BitBtn2.Cursor:=crHandPoint;
BitBtn2.Kind:=bkOk;
BitBtn2.Cursor:=crHandPoint;
BitBtn2.Caption:='&Intercambiar';
BitBtn2.Left:=ComboBox2.Left+ComboBox2.Width+30;
BitBtn2.Top:=(Panel1.Height-BitBtn2.Height) div 2;

Panel2.Caption:='';
Panel2.Align:=alLeft;
Panel2.Width:=Form1.ClientWidth;

StringGrid1.Parent:=Panel2;
StringGrid1.Align:=alClient;
StringGrid1.FixedCols:=0;
StringGrid1.FixedRows:=0;
StringGrid1.RowCount:=ComboBox1.ItemIndex+1;
StringGrid1.ColCount:=ComboBox2.ItemIndex+1;

end;

procedure TForm1.BitBtn1Click(Sender: TObject);
var a:tmInteger;
begin
  a:=GenerarMatriz();
  if a<>nil then begin
    MostrarMatriz(a,StringGrid1);
    Finalize(a);
  end;
end;

procedure TForm1.RadioGroup1Click(Sender: TObject);
begin
  if RadioGroup1.ItemIndex=0 then begin
    Label1.Caption:='Fila 1:';
    Label2.Caption:='Fila 2:'; end
  else begin
    Label1.Caption:='Columna 1:';
    Label2.Caption:='Columna 2:';
  end;
end;

procedure TForm1.BitBtn2Click(Sender: TObject);
var a:tmInteger;
begin
```

```

a:=LeerMatriz(StringGrid1);
case RadioGroup1.ItemIndex of
  0: IntercambiarFilas(a,ComboBox1.ItemIndex,ComboBox2.ItemIndex);
  1: IntercambiarColumnas(a,ComboBox1.ItemIndex,ComboBox2.ItemIndex);
end;
MostrarMatriz(a,StringGrid1);
end;

```

En ejecución la aplicación tiene la apariencia que se muestra en la siguiente figura:

56	9	63	17	57	96	87
55	39	74	80	97	57	12
7	15	64	64	80	20	2
98	44	77	2	33	57	43
14	88	76	32	36	55	34
96	35	100	22	11	97	18

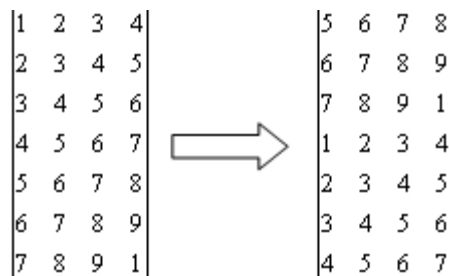
Y la forma donde se introducen los datos para genera la matriz (que aparece al hacer clic sobre el BitBtn1) la siguiente:

16.2. Ejercicios

1. Cree una aplicación con dos paneles, un StringGrid, dos Label, dos ComboBox y un BitBtn. Escriba un módulo que empleando vectores dinámicos cree una matriz de números enteros aleatorios comprendidos entre -1000 y 1000 y otro que muestre los elementos de una matriz de números enteros en un StringGrid, con los números de fila y columna en la primera columna y fila del StringGrid. En el evento "onCreate" de la aplicación haga que el primer panel esté alineado en la parte superior de la forma y tenga 40 puntos de alto, que el segundo panel esté alineado en toda la forma, que el StringGrid pertenezca al Panel2, esté alineado en todo el panel, tenga tres filas y tres columnas, que se actualice a medida que se mueven las barras de desplazamiento; que los Label pertenezcan al Panel1, tengan los títulos 'Filas:' y 'Columnas:', sean transparentes, estén ubicados a 13 puntos de la parte superior y 30 y 130 puntos de la izquierda; que los ComboBox pertenezcan al Panel1, tengan 60 puntos de ancho, estén ubicados a 10 puntos de la parte superior y a 5 puntos a la derecha de los Label, que su lista sean los números del 2 al 100 y que esté seleccionado el número 2; que el BitBtn pertenezca al panel1, tenga la figura "Calculat.bmp", el título "Generar" y que al hacer clic sobre el mismo se genere y muestre una matriz aleatoria con el número de filas y columnas establecidos en los ComboBox.
2. Cree una aplicación con 2 Panel, 2 Label, 2 ComboBox, 2 BitBtn y 1 StringGrid. Programe un módulo que empleando matrices dinámicas reciba una matriz de números enteros y devuelva su transpuesta, otro que lea la matriz desde un StringGrid, otro que muestre la matriz en un StringGrid y otro que genere la matriz de números enteros. En el evento "onCreate" de la forma haga que el primer panel esté alineado en la parte superior de la forma, tenga 30 puntos de alto y no tenga título; que el segundo esté alineado en toda la forma y no tenga título; que el StringGrid pertenezca al Panel2, esté alineado en todo el panel, sus celdas tengan 50 puntos de ancho y tenga 2 filas y 2 columnas; que los Label pertenezcan al Panel1, tengan los títulos "Filas:" y "Columnas:", que sean transparentes, estén a 8 puntos del margen superior y 20 y 140 puntos del margen izquierdo; que los ComboBox pertenezcan al Panel1, tengan 60 puntos de ancho, estén a 5 puntos del margen superior y a 5 puntos de los Label, siendo su lista los números del 1 al 200, estando seleccionado el número 2; que los BitBtn pertenezcan al panel1, el primero tenga la figura "DockStack.bmp", el título "&Generar", esté a 250 puntos del margen izquierdo y a 5 del superior, tenga 90 puntos de ancho y que al hacer clic sobre el mismo se genere y muestre una matriz con los números de fila y columna de los ComboBox; que el segundo tenga la figura "Export.bmp", el título "&Transponer", esté a 350 puntos del margen izquierdo y a 5 del superior, tenga 90 puntos de ancho y que al hacer clic sobre el mismo transponga y muestre la matriz transpuesta.
3. Cree una aplicación con 4 Panel, 2 Label, 2 ComboBox, 2 BitBtn y 3 StringGrid. Programe un módulo que reciba 2 matrices dinámicas de números reales y devuelva la suma de las mismas, otro que lea la matriz desde un StringGrid, otro que muestre la matriz en un StringGrid y otro que genere la matriz de números reales. En el evento "onCreate" de la forma haga que el primer panel esté alineado en la parte superior de la forma, tenga 30 puntos de alto y no tenga título; que el segundo esté alineado a la izquierda, tenga un ancho igual a la tercera parte del ancho disponible en la forma y no tenga título; que el tercero esté alineado a la izquierda, tenga un ancho igual a la tercera parte del ancho disponible en la forma y no tenga título; que el cuarto esté ali-

neado en la **forma** y no tenga título; que el primer StringGrid pertenezca al Panel2, el segundo al Panel3 y el tercero al Panel4, estén alineados en todo el panel, no tenga filas ni columnas fijas y que tengan una fila y una columna; que los Label pertenezcan al Panel1, tengan los títulos "Filas:" y "Columnas:", que sean transparentes, estén a 8 puntos del margen superior y 20 y 140 puntos del margen izquierdo; que los ComboBox pertenezcan al Panel1, tengan 60 puntos de ancho, estén a 5 puntos del margen superior y a 5 puntos de los Label, **siendo** su lista los números del 1 al 150, **estando** seleccionado el número 2; que los BitBtn pertenezcan al Panel1, el primero tenga la figura "Edit.bmp", el título "&Generar", esté a 250 puntos del margen izquierdo y a 5 del superior, tenga 80 puntos de ancho y que al hacer clic sobre el mismo se generen y muestren **dos** matrices, en los **dos** StringGrid, con el número de filas y columnas de los ComboBox; que el segundo BitBtn tenga la figura "Sum.bmp", el título "&Sumar", esté a 340 puntos del margen izquierdo y a 5 del superior, tenga 80 puntos de ancho y que al hacer clic sobre el mismo lea las matrices de los StringGrid 1 y 2, sume sus elementos y muestre la matriz resultante en el tercer StringGrid.

4. Cree una aplicación con 2 Panels, 1 Label, 1 ComboBox y 1 Memo. Programe un módulo que reciba un número entero comprendido entre 1 y 100 y devuelva una matriz dinámica con ese número de filas con los elementos del triángulo de Pascal (empleando sólo el número de columnas necesarios en cada fila) y otro que muestre las filas de la matriz en un Memo. En el evento "onCreate" de la **forma** haga que el primer panel esté alineado en la parte superior de la **forma**, tenga 30 puntos de alto y no tenga título; que el segundo esté alineado en toda la **forma** y no tenga título; que el Memo pertenezca al Panel2, esté alineado en toda la **forma**, tengan barras de desplazamiento vertical y horizontal; que el Label pertenezcan al Panel1, tengan el título "Filas:", que sean transparentes, esté a 8 puntos del margen superior y 30 del margen izquierdo; que el ComboBox pertenezcan al Panel1, tengan 50 puntos de ancho, esté a 5 puntos del margen superior y a 5 puntos del Label, **siendo** su lista los números del 1 al 100, **estando** seleccionado el número 1; que el BitBtn pertenezca al Panel1, tenga la figura "BulbOn.bmp", el título "&Generar" y que al hacer clic sobre él se genere y muestre en el Memo el triángulo de Pascal con el número de filas especificado en el ComboBox (Las 6 primeras filas del triángulo de Pascal son: 1; 1,1; 1,2,1; 1,3,3,1; 1,4,6,4,1; 1,5,10,10,5,1; 1,6,15,20,15,6,1).
5. En una aplicación se necesita de un módulo que rote (hacia abajo) las filas de una matriz de números enteros, un determinado número de posiciones, un ejemplo del resultado que se debe lograr **cuando** se rota una matriz 3 posiciones, se presenta al final de la pregunta. **Elabore** aplicación que empleando **vectores** dinámicos lleven a cabo dicha operación.



6. Se quiere intercambiar las diagonales adyacentes a la diagonal principal de una matriz cuadrada, así **por** ejemplo con la siguiente matriz se debe obtener:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 5 & 3 & 4 \\ 2 & 6 & 10 & 8 \\ 9 & 7 & 11 & 15 \\ 13 & 14 & 12 & 16 \end{bmatrix}$$

Elabore una aplicación que empleando vectores dinámicos lleve a cabo dicha tarea.

17. PUNTEROS

Ya hemos dicho que los vectores dinámicos son en realidad punteros, sin embargo, muchas de sus operaciones están automatizadas, por lo que en general pueden ser empleados de forma muy similar a las matrices y vectores estáticos (con excepción de los aspectos comentados en el capítulo 15). En este tema estudiaremos los punteros propiamente y comenzaremos recordando que un puntero es una variable que almacena una dirección de memoria.

Cuando se trabaja con punteros usualmente se dice que un puntero apunta a alguna dirección de memoria donde se encuentra almacenado algún tipo de información, en consecuencia un puntero puede apuntar (tener la dirección de memoria) de cualquier variable, estructura, módulo o espacio de memoria. Por ello, cuando se trabaja con punteros es posible acceder a cualquier sector de memoria, sin importar el lugar donde se encuentre ni el tipo de información que guarde.

Esta característica hace que el trabajo con punteros requiera más cuidado y planificación por parte del programador. Un error con punteros puede ocasionar no sólo resultados incorrectos, sino también pérdida de información, desorganización de la memoria e incluso el colapso del sistema operativo.

Como ya se dijo previamente un puntero es simplemente una variable que almacena una dirección de memoria y una dirección de memoria en Windows (con una arquitectura de 32 bits) ocupa 4 bytes, por consiguiente una variable puntero siempre ocupa 4 bytes (u 8 en la arquitectura de 64 bits), sin importar a qué apunte (qué dirección de memoria tenga), esto es algo que siempre se debe tener presente cuando se trabaja con punteros.

17.1. DECLARACIÓN DE VARIABLES PUNTERO

No obstante y a pesar de que todas las variables puntero son direcciones de memoria (y ocupan 4 u 8 bytes), Pascal, al igual que otros lenguajes permite declarar dos tipos de variables puntero: *punteros con tipo* y *punteros sin tipo* o punteros puros.

Los punteros con tipo, como su nombre sugiere, almacenan una dirección de memoria donde se espera encontrar un tipo específico de información. Estos punteros se declaran igual que una variable normal, pero precediendo el tipo de dato con un circunflejo (^). Por ejemplo las siguientes sentencias declaran 8 punteros con tipo:

```
var
  i1,i2: ^LongInt;
  s1,s2: ^String;
  x: ^Extended;
  y: ^Boolean;
  v,w: ^Array [1..50] of char;
```

En las direcciones almacenadas en "i1" e "i2" se espera encontrar datos de tipo *LongInt*, en las direcciones almacenadas en "s1" y "s2" se espera encontrar datos de tipo *string*, en la dirección almacenada en "x" un dato de tipo *extended*, en la dirección almacenada en "y" un dato de tipo *Boolean* y en las direcciones almacenadas en "v" y "w" arrays de tipo *char* con 50 elementos.

Los punteros con tipo al esperar un determinado tipo de dato en la dirección a la que apuntan ayudan a prevenir y evitar algunos errores frecuentes. No obstante es necesario recalcar que al trabajar con punteros el control del programa y de los errores queda principalmente en manos del programador.

Es posible declarar también tipos de datos puntero y luego variables (puntero) de ese tipo. Procediendo de esa manera las anteriores declaraciones serían escritas de la siguiente forma:

type

```
pLongInt = ^LongInt;
pString = ^String;
pExtended = ^Extended;
pBoolean = ^Boolean;
pArray50 = ^Array [1..50] of char;
```

var

```
i1,i2: pLongInt;
s1,s2: pString;
x: PExtended;
y: PBoolean;
v,w: pArray50;
```

Borland Pascal cuenta también con algunos tipos punteros predefinidos: PSingle, PDouble, PExtended, PCurrency, PInteger, PString, PAnsiString, PShortString, etc., los cuales por supuesto apuntan a datos del tipo que aparece en el nombre.

El segundo tipo de punteros: los punteros sin tipo, se declaran empleando la palabra reservada *pointer*. Por ejemplo la siguiente sentencia declara 3 variables puntero sin tipo:

var

```
p1, p2, p3 : pointer;
```

Puesto que en la dirección de memoria almacenada por un puntero sin tipo no se espera encontrar ningún tipo de dato o información en particular, es poco lo que el compilador puede hacer para ayudar a prevenir errores, por lo que al trabajar con punteros sin tipo se debe tener más cuidado aún que con los punteros con tipo. Un puntero sin tipo puede recibir la dirección de cualquier otro puntero, recíprocamente su dirección puede ser asignada a cualquier variable puntero, por lo que suele ser de utilidad cuando se quiere intercambiar información entre dos direcciones de memoria.

17.2. ASIGNACIÓN DE DIRECCIONES A LAS VARIABLES PUNTERO

En general existen dos formas de asignar direcciones de memoria a un puntero: a) asignar la dirección de una variable, constante o módulo existente y b) reservar un segmento de memoria y asignar la dirección de ese segmento a un puntero (variables dinámicas). Estudiaremos en este acápite la primera forma y la segunda la estudiaremos en el acápite de variables dinámicas.

Para asignar la dirección de memoria de una variable, constante, procedimiento o función, a un puntero se puede emplear el operador @ (arroba) o la función Addr (address = dirección). En el primer caso se debe preceder el nombre de la variable, constante, procedimiento o función con el operador @ y en el segundo se debe mandar como parámetro el nombre de la variable, constante, procedimiento o función. Por ejemplo en el siguiente segmento de código se asigna la dirección de memoria de dos variables reales a dos punteros con tipo y a uno sin tipo:

var

```
x1: extended; x2: single;
p1: ^extended; p2: ^single; p: pointer;
```

begin

```
p1:= Addr(x1);
p2:= @x2;
```



```
p:= p1;
```

Después de esta asignación tanto *p1* como *p* apunta a *x1*.

En la mayoría de los casos tanto *@* como *Addr* son equivalentes, sin embargo existe una ligera diferencia, *Addr* siempre devuelve un puntero sin tipo, mientras que *@* devuelve un puntero con tipo cuando la directiva de compilación *TypedAddress* está activada, esto es cuando en el programa se ha incluido la directiva *{T+}* o *{\$TYPEDADDRESS ON}*. Si esta directiva no está incluida ambos métodos son equivalentes.

Es posible también, aunque no recomendable, asignar directamente una dirección de memoria a un puntero, por ejemplo con los anteriores punteros se podrían efectuar las siguientes asignaciones:

```
p:= Pointer(122433); p1:= Pointer(229382); p2:= Pointer(34233);
```

17.3. LA DIRECCIÓN NULA

Para especificar que un puntero no apunta a nada, es decir que contiene una dirección de memoria nula, se emplea la constante *nil*. Con frecuencia se emplea esta constante para señalar el final de una estructura. La constante *nil* es equivalente a inicializar en cero una variable de tipo numérica.

17.4. ACCESO A LA INFORMACIÓN APUNTADA POR UN PUNTERO

Para acceder a la información apuntada por un puntero con tipo, simplemente se escribe el nombre del puntero seguido de un circunflejo (^). Con esta sintaxis se pueden efectuar todas las operaciones que son válidas para el tipo de dato al que apunta el puntero. Por ejemplo el siguiente segmento de código muestra en pantalla "El valor de n es: 777":

```
var
  n: LongInt;
  p: ^LongInt;
begin
  p:= @n;
  p^:= 777;
  ShowMessage('El valor de n es : ',IntToStr(n));
end.
```

Como se puede ver no se modifica directamente el valor de "n", pero dado que "p" apunta a "n", al modificar el valor al que apunta "p" se modifica el valor de "n".

Aunque normalmente los punteros sin tipo no se emplean para acceder directamente a la información a la que apuntan, es posible emplearlos con este fin mediante un moldeado de tipos, recordemos que para el moldeado de tipos se escribe el nombre del tipo y entre paréntesis el valor que se quiere moldear. Por ejemplo el siguiente segmento de código hace uso de esta técnica y muestra en pantalla los valores 5.5, 2.5 y 9.0:

```
var
  x1,x2,x3: double;
  p1,p2,p3: pointer;
begin
  p1:= @x1; p2:= @x2; p3:= @x3;
  pdouble(p1)^:= 5.5;
  pdouble(p2)^:= 2.5;
  pdouble(p3)^:= sqr(pdouble(p1)^-pdouble(p2)^);
  ShowMessage(Format('x1 = %8.2f x2 = %8.2f x3= %8.2f',[x1,x2,x3]));
end.
```

Una vez más se trabaja con los punteros (*p1*, *p2* y *p3*) y no con las variables (*x1*, *x2* y *x3*), sin embargo dado que los punteros apuntan a estas variables, al modificar su valor se modifica el valor de las variables.

17.5. PUNTEROS Y VECTORES

Aunque en algunos lenguajes como "C" los punteros pueden ser tratados directamente como vectores (de manera similar a los vectores dinámicos y parámetros abiertos), en Pascal se debe recurrir a los procedimientos "Inc" y "Dec".

Con punteros los procedimientos "Inc" y "Dec", se comportan de manera diferente que como lo hacen con los tipos ordinales. Con punteros "Inc" incrementa la dirección de memoria de manera que apunte al siguiente dato del mismo tipo, por el contrario "Dec" disminuye la dirección de memoria de manera que apunte al anterior dato del mismo tipo. Por supuesto se puede incrementar o disminuir la dirección de memoria en 2 o más posiciones, por ejemplo *Inc(p,5)* (siendo "p" un puntero), incrementa la dirección de "p" de manera que apunte al sexto dato del mismo tipo.

En el siguiente segmento de código se calcula la sumatoria de los elementos de un vector:

```
const
  x : array [1..20] of LongInt = (1,4,5,6,3,2,3,4,2,3,5,6,2,2,3,2,4,2,5,7);
var
  p: pLongInt; s: LongInt; i: byte;
begin
  p:= @x; s:= 0;
  for i:=1 to 20 do
    begin s:=s+p^; inc(p) end;
  ShowMessage(Format('La sumatoria es: %6d',[s]));
end;
```

Como se puede observar cada vez que se repite el ciclo se suma a "s" el valor al que apunta "p". Al comenzar el ciclo "p" apunta al primer elemento del array (*p=@x*) y antes de repetir el ciclo se hace que "p" apunte al siguiente dato (con *Inc(p)*), de esa manera se suman todos los elementos del vector.

Observe que al finalizar este código "p" queda apuntando al dato que existe después del vector (del cual no se tiene ninguna información). Para realizar una nueva operación con el vector, sería necesario asignar nuevamente la dirección al puntero (*p:=@x*), pues de lo contrario se trabajaría con datos desconocidos. Este es un error que se comete frecuentemente cuando se trabaja con punteros (no se vuelve a apuntar al primer dato) por lo que debe ser tomado muy en cuenta.

Igualmente se debe recordar que el compilador no controla el número de datos de un vector. Así en el anterior programa, si en lugar de finalizar el ciclo en 20 (que corresponde al número de datos con los que realmente se cuenta) se finaliza en 200, el compilador no reportará ningún error e incluso el programa correrá (aunque por supuesto se obtendrá un resultado erróneo). Cuando como en este caso lo que se hace es leer los datos simplemente se obtiene un resultado erróneo, pero si lo que se hace es escribir datos, se corrompe la memoria con lo que puede colapsar el programa e inclusive el sistema operativo.

Otro error frecuente al trabajar con punteros son los "punteros no inicializados". Por ejemplo el siguiente segmento de código, aún cuando compila sin reportar ningún error, es erróneo:

```

var
  x: double;
  p: pdouble;
begin
  x:=3.45;
  p^:=sqr(x);
  ShowMessage(Format('El valor al que apunta p es : %12.8g',[p^]));
end;

```

En este código se asigna el cuadrado de 3.45 a alguna dirección de memoria desconocida, pues "p" no ha sido inicializado y en consecuencia apunta a alguna dirección de memoria aleatoria. Si el lugar al que apunta "p" es parte de la memoria utilizada por el sistema operativo (u otra aplicación), esta simple asignación puede hacer colgar la aplicación e inclusive el sistema operativo.

Al trabajar con punteros, el compilador tampoco controla que los valores almacenados sean del tipo correcto, por lo tanto es posible acceder a información de un tipo como si fuera de otro. Si bien en algunos casos es posible aprovechar esta característica normalmente es una fuente de error, por ejemplo al ejecutar el siguiente segmento de código:

```

var
  x: double; i: integer;
  p1: pdouble;
  p2: pInteger;
begin
  x:=123.45; i:=12345;
  p1:=@i; p2:=@x;
  ShowMessage(Format('El valor al que apunta p1 es: %8.6g',[p1^]));
  ShowMessage(Format('El valor al que apunta p2 es: %8d',[p2^]));
end;

```

No se muestra en pantalla los números 123.45 y 12345, como era de esperar, sino un número aleatorio (182693E-307) y -858993459. Esto sucede porque al leer el valor al que apunte "p1" el compilador lee 8 bytes (el tamaño de un número double) cuando en esa dirección sólo se han almacenado 4 bytes (el tamaño de un integer), por lo tanto los otros 4 bytes son los valores aleatorios que se encuentran en ese momento en ese sector de memoria. Por el contrario al leer el valor al que apunta "p2" el compilador lee sólo 4 bytes (el tamaño de un integer), cuando en esa dirección se han almacenado 8 bytes (el tamaño de un double), en consecuencia sólo se lee la mitad de la información, la que a su vez es interpretada como un entero.

17.6. PUNTEROS Y MATRICES

Aunque en realidad los punteros sólo pueden ser tratados como vectores, internamente toda matriz (y en realidad todo array multidimensional) es almacenado como un vector. En el caso de las matrices los elementos de la segunda fila están a continuación del último elemento de la primera fila, los elementos de la tercera fila están a continuación del último elemento de la segunda fila y así sucesivamente, no existe ningún carácter, elemento o símbolo que separe una fila (o dimensión) de otra. En consecuencia un array de n filas y m columnas se almacena internamente como un vector con $n*m$ elementos. Este hecho hace posible acceder a los elementos de una matriz empleando punteros.

Por ejemplo si se tienen una matriz con 7 filas y 9 columnas, como la que se presenta a continuación (cuyos índices comienzan en cero):

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$	$a_{0,5}$	$a_{0,6}$	$a_{0,7}$	$a_{0,8}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$	$a_{1,6}$	$a_{1,7}$	$a_{1,8}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$	$a_{2,6}$	$a_{2,7}$	$a_{2,8}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$	$a_{3,6}$	$a_{3,7}$	$a_{3,8}$
$a_{4,0}$	$a_{4,1}$	$a_{4,2}$	$a_{4,3}$	$a_{4,4}$	$a_{4,5}$	$a_{4,6}$	$a_{4,7}$	$a_{4,8}$
$a_{5,0}$	$a_{5,1}$	$a_{5,2}$	$a_{5,3}$	$a_{5,4}$	$a_{5,5}$	$a_{5,6}$	$a_{5,7}$	$a_{5,8}$
$a_{6,0}$	$a_{6,1}$	$a_{6,2}$	$a_{6,3}$	$a_{6,4}$	$a_{6,5}$	$a_{6,6}$	$a_{6,7}$	$a_{6,8}$

Se almacena internamente como un vector con 63 elementos ($7 * 9$). Dicho vector sería el siguiente:

a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
a_9	a_{10}	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{16}	a_{17}
a_{18}	a_{19}	a_{20}	a_{21}	a_{22}	a_{23}	a_{24}	a_{25}	a_{26}
a_{27}	a_{28}	a_{29}	a_{30}	a_{31}	a_{32}	a_{33}	a_{34}	a_{35}
a_{36}	a_{37}	a_{38}	a_{39}	a_{40}	a_{41}	a_{42}	a_{43}	a_{44}
a_{45}	a_{46}	a_{47}	a_{48}	a_{49}	a_{50}	a_{51}	a_{52}	a_{53}
a_{54}	a_{55}	a_{56}	a_{57}	a_{58}	a_{59}	a_{60}	a_{61}	a_{62}

Donde los elementos han sido ordenados en filas sólo para ver con mayor claridad la relación que existe entre los elementos de la matriz (con dos subíndices) y los elementos del vector (con un solo subíndice).

Observando la matriz y el vector, se deduce que para acceder a cualquier elemento de la matriz, en forma de vector, el índice del vector debe ser calculado con la siguiente relación:

$$iv = if * nc + ic$$

Donde "iv" es el índice del vector, "if" es el índice de la fila en la matriz, "ic" es el índice de la columna en la matriz y "nc" es el número de columnas en la matriz.

Así en la matriz del ejemplo para acceder al elemento $a_{2,3}$ en forma de vector, el índice sería:

$$iv = 2 * 9 + 3 = 21$$

Entonces es posible acceder al elemento $a_{2,3}$, en forma de vector, haciendo referencia al elemento a_{21} , lo que puede ser fácilmente corroborado observando la matriz y el vector mostrados anteriormente.

De igual manera, para acceder a los elementos $a_{0,7}$, $a_{4,0}$, $a_{5,8}$, $a_{6,3}$ y $a_{6,7}$, en forma de vector, los elementos serían: $a_{0*9+7} = a_7$; $a_{4*9+0} = a_{36}$; $a_{5*9+8} = a_{53}$; $a_{6,3} = a_{6*9+3} = a_{57}$; $a_{6,8} = a_{62}$. Lo que puede ser corroborado una vez más mediante la inspección visual de las matrices.

Añadiendo estos índices a la dirección del primer elemento, se puede acceder al valor de cualquier elemento. Por ejemplo si "x" apunta al primer elemento de la matriz ($a_{0,0}$) y se desea trabajar con el elemento $a_{5,6}$, la dirección del puntero debería ser cambiada de la siguiente forma:

$$inc(x, 5*9+6);$$

Se debe tomar en cuenta que después de este incremento "x" ya no apunta más al primer elemento, por lo que si se quiere acceder a otro elemento, empleando esta fórmula, sería necesario volver a asignar la dirección del primer elemento a la variable "x". Por esta razón cuando se trabaja con punteros, usualmente se emplean al menos dos punteros, uno que apunta al primer elemento de la matriz y otro, u otros, que cambian su dirección según sea necesario. El no guardar la dirección del primer elemento, es otro de los

errores que se comete frecuentemente cuando se trabaja con punteros, por lo que es un aspecto que debe ser tomado muy en cuenta.

17.7. VARIABLES DINÁMICAS

La principal aplicación de los punteros está en la creación, lectura, mantenimiento y destrucción de *variables dinámicas*, sin embargo y a pesar de que su principal aplicación radica en las variables dinámicas, es necesario tomar en cuenta que los punteros no son variables dinámicas, sino y como se ha recalcado repetidamente *son variable que almacenan direcciones de memoria*.

Una variable dinámica es una variable que es creada y destruida en tiempo de ejecución, esto significa que la memoria requerida por una variable dinámica se reserva y libera mientras el programa está corriendo, contrariamente a las variables estáticas para las cuales la memoria se reserva al momento de compilar el programa y se libera sólo cuando el programa finaliza. En consecuencia las variables dinámicas permiten un uso más eficiente y racional de la memoria disponible.

En realidad, aún cuando no las administramos directamente, ya hemos estado utilizando variables dinámicas: todas las variables locales que se declaran dentro de un procedimiento o función son dinámicas (excepto por supuesto las variables declaradas explícitamente como estáticas). La memoria que requieren las variables locales se reserva cuando se llama al procedimiento o función y se libera cuando el procedimiento o función concluye. La diferencia con las variables que estudiaremos en el presente acápite es que ahora está a nuestro cargo la reserva y liberación de la memoria requerida (tarea de la que se encarga el compilador con las variables locales de los procedimientos o funciones).

En general existen dos formas en las que podemos reservar memoria para las variables dinámicas: a) mediante los procedimientos *GetMem* y *ReAllocMem* y b) mediante el procedimiento *New*.

El procedimiento *GetMem* tiene la siguiente sintaxis:

GetMem(Variable puntero, Número de bytes);

Donde "*Variable puntero*" es el nombre de la variable puntero en la cual el procedimiento *GetMem* almacena la dirección de memoria del lugar donde reserva el *Número de bytes requeridos*. Si no existe memoria disponible para la variable, entonces el procedimiento *GetMem* devuelve la constante *nil* en la variable puntero.

El procedimiento *ReAllocMem* tienen la siguiente sintaxis.

ReAllocMem(Variable puntero, Número de bytes);

Donde los parámetros son los mismos que en el procedimiento *GetMem*, excepto que ahora la variable puntero tiene que estar inicializada con una dirección de memoria válida o tener el valor nulo *nil*. Este procedimiento permite cambiar el tamaño de la memoria reservada. Si en la dirección actual existe espacio suficiente, entonces simplemente amplía (o reduce) el número de bytes reservados, caso contrario busca una nueva dirección de memoria con espacio suficiente, reserva espacio en este lugar, copia el contenido de la dirección actual a la nueva dirección y guarda en la *variable puntero* la nueva dirección de memoria. Si la *variable puntero* es inicialmente nula (*nil*) *ReAllocMem* es equivalente al procedimiento *GetMem*. Si no existe suficiente memoria disponible *ReAllocMem* genera un error en tiempo de ejecución.

Con *GetMem* y *FreeMem*, la variable puntero puede ser de cualquier tipo o incluso puede no tener ningún tipo.

El procedimiento *New* tiene el siguiente formato:

New(*Variable puntero*);

En este caso *variable puntero* tiene que ser necesariamente un puntero con tipo. *New* reserva memoria suficiente para el tipo de dato al que apunta y guarda la dirección de la memoria en la *variable puntero*. Al igual que *FreeMem* asigna *nil* a la *variable puntero* si no existe suficiente memoria disponible.

Para liberar la memoria se tienen dos métodos: a) *FreeMem*, que libera la memoria reservada con *GetMem* o *ReAllocMem* y b) *Dispose* que libera la memoria reservada con *New*.

El procedimiento *FreeMem* tiene la siguiente sintaxis:

FreeMem(*variable puntero*);

FreeMem acepta opcionalmente el número de bytes a liberar, el cual, si se escribe, debe coincidir con el número de bytes reservados con *GetMem* o *ReAllocMem*, por lo que en general este parámetro no se emplea en la práctica.

El procedimiento *Dispose* tiene la siguiente sintaxis:

Dispose(*variable puntero*);

Al trabajar con vectores y matrices se reserva memoria con *GetMem* (o *ReAllocMem*) y se la libera con *FreeMem*, pues no se reserva espacio para un valor, sino para varios (el número de elementos que tiene el vector o la matriz).

17.8. EJEMPLOS

En los siguientes ejemplos las interfaces gráficas de las aplicaciones no tienen componentes nuevos con relación a los del anterior tema, razón por la cual no se darán mayores explicaciones sobre las mismas.

1. Elabore una aplicación con dos Memos, una etiqueta y un BitBtn y en la misma escriba los módulos que empleando vectores dinámicos generen, muestren y lean los elementos de un vector de números enteros desde un Memo. Escriba también un módulo para seleccionar una fila determinada en un memo. La forma debe estar centrada en la pantalla, tener 400 puntos de ancho, 450 de alto, con el título "Vectores" y los bordes de un cuadro de diálogo. Los memos deben tener 150 puntos de ancho, 300 de alto, estar centrado horizontalmente (separados con 10 puntos), a 25 puntos del margen superior, contar con una barra de desplazamiento vertical y su texto debe estar alineado a la derecha. La etiqueta debe tener el título "Elementos del vector", ser transparente y estar a 15 puntos por encima del Memo1 (alineado con el mismo a la izquierda). El BitBtn debe tener 160 puntos de ancho, estar centrado a 10 puntos por debajo del Memo1, tener el título "Probar módulos dinámicos", el Glyph "Retry.bmp" y el puntero "crHandPoint". El evento "onClick" del BitBtn debe ser programado de manera que genere un vector con 300 números enteros comprendidos entre 1 y 1000, muestre dichos elementos en el primer memo, lea los elementos del memo en otro vector, muestren dichos elementos en el segundo memo y seleccione la fila 100 del mismo.

Este ejemplo se resuelve en la misma aplicación que en el ejemplo del tema 15, el código de los módulos (programados en una unidad) es el siguiente:

El código de los módulos añadido a la aplicación del anterior ejemplo en la unidad "uPunteros" es:

```

unit uPunteros;

interface

  uses StdCtrls, SysUtils, QDialogs, uDinamicos;

  function GenerarVEnterosp(n:cardinal; li,ls:integer):pInteger;
  procedure MostrarVEnterosp(x:pInteger; n:Cardinal; m:TMemo);
  procedure LeerVEnterosp(m:TMemo; out x:pInteger; out n:cardinal);

implementation

  function GenerarVEnterosp(n:cardinal; li,ls:integer):pInteger;
  var i:cardinal; p,x:pInteger; d:integer;
  begin
    GetMem(x,n*SizeOf(x^));p:=x;
    d:=ls-li+1;
    for i:=1 to n do begin p^:=Random(d)+li; inc(p); end;
    result:=x;
  end;

  procedure MostrarVEnterosp(x:pInteger; n:Cardinal; m:TMemo);
  var i:cardinal;
  begin
    try
      m.Lines.BeginUpdate;
      m.Lines.Clear;
      for i:=1 to n do begin m.Lines.Append(IntToStr(x^)); inc(x); end;
    finally
      m.Lines.EndUpdate;
    end;
  end;

  procedure LeerVEnterosp(m:TMemo; out x:pInteger; out n:cardinal);
  var i:cardinal; p:pInteger;
  begin
    try
      n:=m.Lines.Count;
      GetMem(x,n*SizeOf(x^)); p:=x;
      for i:=0 to n-1 do begin p^:=StrToInt(m.Lines[i]); inc(p); end;
    except
      on EConvertError do begin
        ShowMessage('El número está mal escrito');
        m.SetFocus; SeleccionarFila(i,m);
      end;
    end;
  end;
end.

```

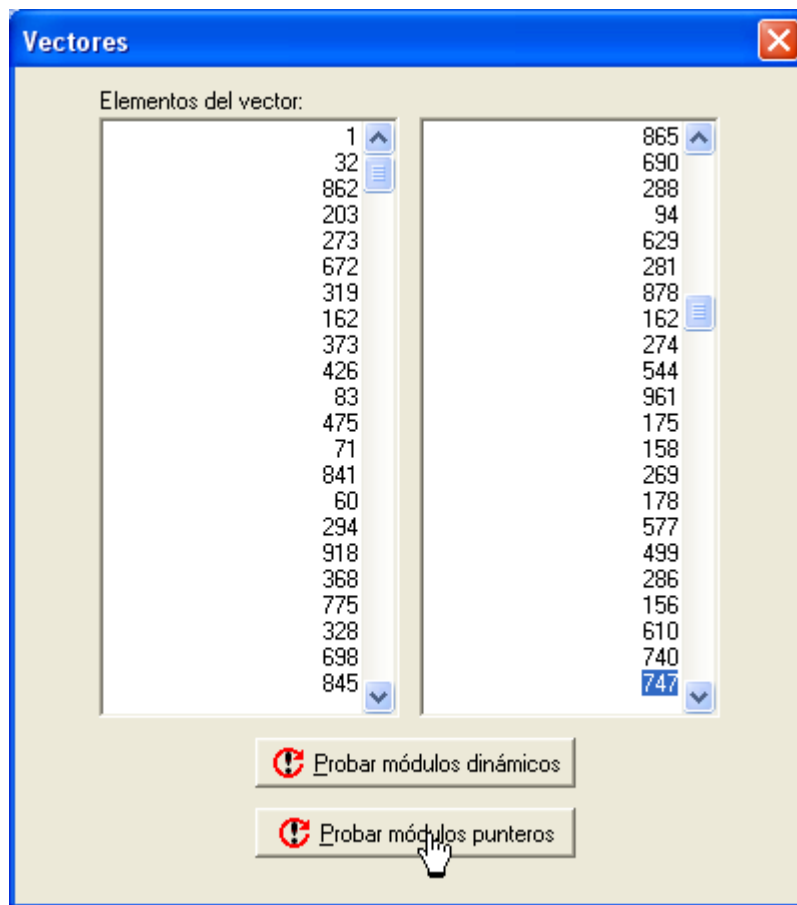
Como se puede ver, en esta unidad se ha incluido la unidad "uDinamicos" del anterior tema porque el módulo "LeerVEnterosp" utiliza el módulo "SeleccionarFila" que se encuentra en dicha unidad.

En el módulo "MostrarVEnterosp" a más del puntero se debe mandar el número de elementos, porque en el caso de los punteros no es posible averiguar el número de elementos. Igualmente, y por la misma razón, cuando se leen los

elementos de un vector se devuelve el puntero y el número de elementos leídos (como parámetros out).

A la aplicación se le debe añadir un BitBtn y el código para el mismo (añadido en el evento "onCreate") es:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    ...
    BitBtn2.Width:= 160;
    BitBtn2.Left:= (Form1.Width-BitBtn1.Width) div 2;
    BitBtn2.Top:= BitBtn1.Top+BitBtn1.Height+10;
    BitBtn2.Caption:= '&Probar módulos punteros';
    BitBtn2.NumGlyphs:= 2;
    BitBtn2.Glyph.LoadFromFile('C:\Archivos de programa\Archivos Comunes\
        Borland Shared\Images\Buttons\Retry.bmp');
    BitBtn2.Cursor:= crHandPoint;
end;
```



El evento "onClick" de dicho botón es:

```
procedure TForm1.BitBtn2Click(Sender: TObject);
var x,y:pInteger; n:cardinal;
begin
    n:=300;
    x:=GenerarVEnterosp(n,1,1000);
    MostrarVEnterosp(x,n,Memo1);
    FreeMem(x);
    LeerVEnterosp(Memo1,y,n);
    MostrarVEnterosp(y,n,Memo2);
```



```

FreeMem(y);
SeleccionarFila(100,Memo2);
end;

```

En ejecución la aplicación tiene la apariencia que se muestra en la figura de la página anterior.

2. Elabore una aplicación con un *ToolBar*, un *StatusBar*, un *StringGrid*, un *Memo* y un *Label*. En la aplicación debe crear un módulo que genere un vector con "n" números reales aleatorios comprendidos entre 0 y 10000, otro que muestre un vector de números reales en un *StringGrid* con dos columnas, mostrando en la primera columna el número de elemento y en la segunda el número real con dos dígitos después del punto, otro que lea los elementos de la segunda columna de un *StringGrid*, otro que calcule el promedio de un vector de números reales y otro que muestre un número real en un *memo* con dos dígitos después del punto. La forma debe tener 300 puntos de ancho y 400 de alto, estar centrada en la pantalla, tener el color "clInactiveCaptionText" y no tener bordes. El *ToolBar* debe tener tres botones con las imágenes en color de "DoorOpen.bmp", "BulbOn.bmp" y "Calculat.bmp", con los Hint: "Salir de la aplicación", "Generar vector" y "Calcular promedio", siendo la forma del puntero una mano apuntando. El *StatusBar* debe tener un solo panel con el texto "Cálculo del promedio" y debe mostrar los Hint de manera automática. El *StringGrid* debe tener 2 columnas (de 60 y 120 puntos), estar centrado horizontalmente y a 30 puntos del margen superior, estando fijas una fila y una columna, los títulos de las columnas deben ser "N°" y "Valor", y tener una barra de desplazamiento vertical. El *memo* debe tener 130 puntos de ancho, 21 de alto, con el texto alineado a la derecha, estar centrado horizontalmente (conjuntamente el *label*), a 10 puntos por debajo *StringGrid* y ser sólo de lectura. El *label* debe tener el título "Promedio:", ser transparente, estar a 5 puntos a la izquierda del *memo* y a 3 puntos por debajo del borde superior del *memo*. Al hacer clic en el primer *ToolButton* la aplicación debe cerrarse, al hacer clic en el segundo *ToolButton*, se debe pedir el número de elementos a generar y generar dicho número de elementos mostrándolos en el *StringGrid* y al hacer clic en el tercer *ToolButton*, se debe calcular el promedio de los elementos del *StringGrid* y mostrar dicho promedio en el *memo*.

Igualmente en este caso se escribe el código en la misma aplicación del ejemplo del tema 15. El código de los módulos cambia tal como se muestra en la siguiente unidad:

```
unit uPunteros;
```

```
interface
```

```
uses Math, QDialogs, SysUtils, StdCtrls, Grids;
```

```
type pReal = ^Real;
```

```
function GenerarVReales(n: cardinal):pReal;
```

```
procedure MostrarVReales(x:pReal; n:cardinal; s:TStringGrid);
```

```
procedure LeerVReales(s:TStringGrid; out x:pReal; out n:cardinal);
```

```
function Media(x: pReal; n:cardinal):real;
```

```
procedure MostrarReal(x:real; m:TMemo);
```

```
implementation
```

```
function GenerarVReales(n: cardinal):pReal;
```

```

var i:cardinal; x,p:pReal;
begin
  GetMem(x,n*SizeOf(x^));
  p:=x;
  for i:=1 to n do begin
    p^:=random*10000; inc(p);
  end;
  result:=x;
end;

procedure MostrarVReales(x:pReal; n:cardinal; s:TStringGrid);
var i:cardinal;
begin
  s.RowCount:=n+1;
  for i:=1 to n do begin
    s.Cells[0,i]:=Format('%12d',[i]);
    s.Cells[1,i]:=Format('%20.2f',[x^]);
    inc(x);
  end;
end;

procedure LeerVReales(s:TStringGrid; out x:pReal; out n:cardinal);
var i:cardinal; p:pReal;
begin
  try
    n:=s.RowCount-1;
    GetMem(x,n*SizeOf(x^));
    p:=x;
    for i:=1 to n do begin
      p^:=StrToFloat(s.Cells[1,i]);
      inc(p);
    end;
  except
    on EConvertError do begin
      ShowMessage('Número real mal escrito');
      s.SetFocus; s.Row:= i;
    end;
  end;
end;

function Media(x: pReal; n:cardinal):real;
var i:Cardinal; s:real;
begin
  if n=0 then raise EInvalidArgument.Create(
    'El vector a promediar está vacío');
  s:=0;
  for i:=1 to n do begin
    s:=s+x^; inc(x);
  end;
  result:= s/n;
end;

procedure MostrarReal(x:real; m:TMemo);
begin
  m.Text:= Format('%12.2f',[x]);
end;

end.

```

En la unidad principal (la unidad de la **forma**) solo cambian los módulos correspondientes a los eventos "onClick" de los ToolButtons 2 y 3:

```
procedure TForm1.tb2Click(Sender:TObject);
var n:integer; x:pReal; s:String;
begin
  s:=InputBox('Generación de elementos','Nº Elementos a generar:', '100');
  n:=StrToInt(s);
  x:=GenerarVReales(n);
  MostrarVReales(x,n,StringGrid1);
  FreeMem(x);
end;

procedure TForm1.tb3Click(Sender:TObject);
var x:pReal; p:real; n:cardinal;
begin
  LeerVReales(StringGrid1,x,n);
  p:=Media(x,n);
  MostrarReal(p,Memo1);
  FreeMem(x);
end;
```

En ejecución, la aplicación tiene la misma apariencia y comportamiento que la aplicación del tema 15.

3. Elabore una aplicación con un ToolBar, un StatusBar y un StringGrid. En la aplicación debe crear un módulo que determine si un número es o no primo, otro que genere un vector con los primeros "n" números primos y otro que muestre un vector de números enteros en un StringGrid con dos columnas, mostrando en la primera columna el número de elemento y en la segunda el número primo. La forma debe tener 250 puntos de ancho y 410 de alto, estar centrada en la pantalla, tener el fondo con líneas diagonales de color azul, el Hint "Números primos" y no tener bordes. El ToolBar debe tener dos botones con las imágenes en color de "DoorOpen.bmp" y "BulbOn.bmp", con los Hint: "Salir de la aplicación" y "Generar Números Primos", siendo la forma del puntero una mano apuntando. El StatusBar debe tener un solo panel con el texto "Números primos" y debe mostrar los Hint de manera automática. El StringGrid debe tener 300 puntos de alto, 2 columnas (de 60 y 80 puntos con una barra de desplazamiento vertical), estar centrado horizontalmente y a 30 puntos del margen superior, estando fijas una fila y una columna, los títulos de las columnas deben ser "Nº" y "Primo". Al hacer clic en el primer ToolButton la aplicación debe cerrarse, al hacer clic en el segundo ToolButton, se debe pedir el número de primos a generar, generar dicho número y mostrarlos en el StringGrid.

Este ejemplo también se resuelve en la misma aplicación del tema 15. La unidad con los módulos que generan y muestran el vector con los números primos es la siguiente (donde el módulo "EsPrimo" no cambia):

```
unit uPunteros;

interface

  uses Grids, SysUtils, Math;

  function EsPrimo(n:cardinal):Boolean;
  function Primos(n:cardinal):pCardinal;
  procedure MostrarPrimos(x:pcardinal; n:cardinal; sg:TStringGrid);
```

implementation

```

function EsPrimo(n:cardinal):Boolean;
var i:cardinal;
begin
  if n=0 then raise EInvalidArgument.Create(
    'El número debe ser mayor a cero');
  for i:=2 to n div 2 do
    if n mod i = 0 then begin
      result:=False; exit;
    end;
  result:=True;
end;

function Primos(n:cardinal):pCardinal;
var i,j:cardinal; p,x:pCardinal;
begin
  GetMem(x,n*SizeOf(x^));
  i:=0; j:=1; p:=x;
  while i<n do begin
    if EsPrimo(j) then begin
      p^:=j; inc(p); inc(i);
    end;
    inc(j);
  end;
  result:=x;
end;

procedure MostrarPrimos(x:pcardinal; n:cardinal; sg:TStringGrid);
var i:cardinal;
begin
  sg.RowCount:=n;
  for i:=1 to n do begin
    sg.Cells[0,i]:=Format('%12d',[i]);
    sg.Cells[1,i]:=Format('%14d',[x^]);
    inc(x);
  end;
end;

end.

  En la unidad principal (la unidad de la forma) sólo cambia el módulo correspondiente al evento "onClick" del ToolButton2:

procedure TForm1.tb2Click(sender: TObject);
var n:cardinal; x:pCardinal;
begin
  n:=StrToInt(InputBox('Generar Primos','N° de primos a generar:','100'));
  x:=Primos(n);
  MostrarPrimos(x,n,StringGrid1);
  FreeMem(x);
end;

  En ejecución, la aplicación tiene la misma apariencia y comportamiento que en el mencionado ejemplo.

```

4. Elabore una aplicación con un ToolBar, un StatusBar, seis paneles, seis Labels, seis ComboBox, tres StringGrids y un ImageList. En la aplicación escriba el código de los siguientes módulos: uno que muestre los

elementos de una matriz de números reales en un StringGrid, otro que lea los elementos de una matriz de números reales desde un StringGrid, otro que genere una matriz con "m" filas y "n" columnas de números reales aleatorios comprendidos entre dos límites dados y otro que multiplique dos matrices de números reales. La forma debe tener 700 puntos de ancho, 500 de alto, estar centrada en la pantalla y no tener bordes. El ToolBar debe tener cuatro botones con las figuras de "Opendoor.bmp", "Retry.bmp", "Calculat.bmp" y "Clear.bmp", con los hints: "Salir de la aplicación", "Generar Matriz", "Multiplicar Matrices" y "Borrar Matriz". El primer panel debe estar alineado a la izquierda y tener un ancho igual a la mitad del ancho disponible en la forma, el segundo panel debe estar alineado a la derecha y tener el mismo ancho que el primero, el tercer panel debe estar alineado al fondo y tener un alto igual a la mitad del alto disponible en la forma (sin contar el ToolBar y el StatusBar). El StatusBar debe tener un solo panel con el texto "Multiplicación de matrices" y en el mismo se deben mostrar automáticamente los hint de los objetos. En el primer panel se debe insertar el cuarto panel alineado en la parte superior, con 30 puntos de alto, el título "Matriz A" alineado a la izquierda y en el mismo se deben insertar los dos primeros Labels con los títulos "Filas:" y "Columnas" y los dos primeros ComboBox con 60 puntos de ancho y los números del 1 al 500, los Label y los ComboBox deben estar centrados verticalmente en ese panel y los Labels deben estar al lado de los ComboBox respectivos, en el panel se debe insertar el primer StringGrid que debe estar alineado en todo el panel, tener columnas de 60 puntos de ancho, el Hint "Matriz A" y no tener filas ni columnas fijas. En el segundo panel se debe proceder igual que en el primero, sólo que ahora se inserta el quinto panel con el título "Matriz B", los Labels 3 y 4, los ComboBoxs 3 y 4 y el StringGrid 2 con el hint "Matriz B". En el tercer panel se procede igual que en el primero sólo que se inserta el sexto panel con el título "Matriz C", los Labels 5 y 6, los ComboBox 5 y 6 y el StringGrid3 con el hint "Matriz C". Al hacer clic sobre el primer ToolButton la aplicación debe cerrarse, al hacer clic sobre el segundo ToolButton se debe generar una matriz con el número de filas y columnas especificado en los ComboBoxs, al hacer clic sobre el tercer ToolButton se deben multiplicar las matrices "A", "B" y mostrar la matriz resultante "C" y al hacer clic sobre el cuarto ToolButton se deben borrar las matrices dejando una sola fila y columna.

La aritmética de punteros, sobre todo cuando se trabaja con matrices, requiere cierto cuidado y planificación, por lo que es conveniente realizar un esquema donde se visualice la forma en la que deben ir cambiando las direcciones de los punteros. Por ejemplo, en este caso, al multiplicar las matrices los punteros quedan apuntando a elementos intermedios, por lo que es necesario conservar las direcciones originales en otros punteros, siendo esta la razón por la cual en el módulo "MulMat" se guardan las direcciones originales de los punteros "a", "b" y "c" en "a0", "b0" y "c0" respectivamente.

La unidad con los módulos que leen, muestran, generan, borran y multiplican las matrices, empleando punteros, es:

```
unit uPunteros;

interface

uses Grids, SysUtils, QDialogs;

type pReal = ^Real;
```

```

procedure MostrarMatriz(a:pReal; m,n:cardinal; sg:TStringGrid);
procedure LeerMatriz(sg:TStringGrid; out a:pReal; out m,n:cardinal);
function GenerarMatriz(m,n:cardinal; li,ls:real):pReal;
function MulMat(a,b:pReal;nfa,nca,nfb,ncb:cardinal):pReal;
procedure BorrarStringGrid(sg:TStringGrid);

```

implementation

```

procedure MostrarMatriz(a:pReal; m,n:cardinal; sg:TStringGrid);
var i,j:cardinal;
begin
    sg.RowCount:=m;
    sg.ColCount:=n;
    for i:=1 to m do
        for j:=1 to n do begin
            sg.Cells[j-1,i-1]:=Format('%10.2f',[a^]);
            inc(a);
        end;
    end;

procedure LeerMatriz(sg:TStringGrid; out a:pReal; out m,n:cardinal);
var i,j:cardinal; p:pReal;
begin
    try
        m:=sg.RowCount;
        n:=sg.ColCount;
        GetMem(a,m*n*SizeOf(a^));
        p:=a;
        for i:=1 to m do
            for j:=1 to n do begin
                p^:=StrToFloat(sg.Cells[j-1,i-1]);
                inc(p);
            end;
        except
            on EConvertError do begin
                ShowMessage('Número real mal escrito');
                sg.SetFocus; sg.Row:=i-1; sg.Col:=j-1;
            end;
        end;
    end;

function GenerarMatriz(m,n:cardinal; li,ls:real):pReal;
var i,j:cardinal; a,p:pReal; d:real;
begin
    GetMem(a,m*n*SizeOf(a^));
    d:=ls-li;
    p:=a;
    for i:=1 to m do
        for j:=1 to n do begin
            p^:=random*d+li;
            inc(p);
        end;
    result:=a;
end;

function MulMat(a,b:pReal;nfa,nca,nfb,ncb:cardinal):pReal;
var i,j,k:cardinal; a0,b0,c0,c:pReal;

```

```

begin
  if (nfa=0) or (nfb=0) then raise EInvalidOp.Create(
    'La matriz a multiplicar está vacía');
  if nca<>nfb then raise EInvalidOp.Create(
    'Las matrices no son multiplicables');
  GetMem(c,nfa*ncb*SizeOf(c^));
  a0:=a; b0:=b; c0:=c;
  for i:=1 to nfa do begin
    for j:=1 to ncb do begin
      c^:=0;
      for k:=1 to nca do begin
        c^:=c^+a^*b^;
        inc(a); inc(b,ncb);
      end;
      a:=a0; b:=b0; inc(b,j); inc(c);
    end;
    inc(a0,nca); a:=a0; b:=b0;
  end;
  result:=c0;
end;

procedure BorrarStringGrid(sg:TStringGrid);
begin
  sg.RowCount:=1;
  sg.ColCount:=1;
  sg.Cells[0,0]:=Format('%10.2f',[0.0]);
end;

```

end.

La unidad principal (la unidad de la forma) sólo difiere de la elaborada en el tema 16, en los eventos de los ToolButton 2 y 3 y en que ahora se hace uso de la unidad “uPunteros” en lugar de “uDinamicos”:

```

procedure TForm1.tb2Click(sender: TObject);
var nfa,nca,nfb,ncb:cardinal; a,b:pReal;
begin
  nfa:=ComboBox1.ItemIndex+1;
  nca:=ComboBox2.ItemIndex+1;
  a:=GenerarMatriz(nfa,nca,0,1000);
  MostrarMatriz(a,nfa,nca,StringGrid1);
  nfb:=ComboBox3.ItemIndex+1;
  ncb:=ComboBox4.ItemIndex+1;
  b:=GenerarMatriz(nfb,ncb,0,1000);
  MostrarMatriz(b,nfb,ncb,StringGrid2);
  FreeMem(a); FreeMem(b);
end;

procedure TForm1.tb3Click(sender: TObject);
var a,b,c:pReal; nfa,nca,nfb,ncb:cardinal;
begin
  LeerMatriz(StringGrid1,a,nfa,nca);
  LeerMatriz(StringGrid2,b,nfb,ncb);
  c:=MulMat(a,b,nfa,nca,nfb,ncb);
  MostrarMatriz(c,nfa,ncb,StringGrid3);
  ComboBox5.ItemIndex:=nfa-1;
  ComboBox6.ItemIndex:=ncb-1;
  FreeMem(a); FreeMem(b); FreeMem(c);
end;

```

En ejecución la aplicación tiene la apariencia y comportamiento de la aplicación presentada en el tema 16.

5. Como otro ejemplo volveremos a resolver el problema de generar los cuadrados mágicos, resuelto en el ejemplo 2 del tema 16, empleando punteros.

La interfaz de la aplicación es la misma que la del mencionado ejemplo, sólo que ahora empleamos punteros para llevar a cabo las operaciones. La unidad con los módulos modificados para trabajar con punteros es la siguiente:

```

unit Unit2;

interface

uses Grids, SysUtils;

type pWord = ^word;

function CuadradoMagico(n:word):pWord;
procedure MostrarCuadrado(a:pWord; n:Word; s:tStringGrid);

implementation

function CuadradoMagico(n:word):pWord;
var a,a0:pWord; i,j,k: word;
begin
  if not odd(n) then raise EInvalidOp.Create(
    'El número de debe ser impar');
  GetMem(a,n*n*SizeOf(a^));
  a0:=a; i:=1; j:=(n+1) div 2;
  inc(a,j-1);
  for k:=1 to sqr(n) do begin
    a^:=k;
    if k mod n = 0 then
      begin inc(a,n); inc(i); end
    else
      if j=1 then begin dec(a);dec(i);j:=n end
      else
        if i>1 then begin dec(a,n+1);dec(i);dec(j); end
        else begin inc(a,(n-1)*n-1); i:=n; dec(j); end;
      end;
    result:=a0;
  end;

procedure MostrarCuadrado(a:pWord; n:Word; s:tStringGrid);
var i,j: byte;
begin
  s.RowCount:=n;
  s.ColCount:=n;
  for i:=0 to n-1 do
    for j:=0 to n-1 do begin
      s.Cells[j,i]:=IntToStr(a^);
      inc(a);
    end;
  end;

end.

```


Se debe modificar también el código del evento onChange del "ComboBox1":

```
procedure TForm1.ComboBox1Change(Sender: TObject);
var n:word; a:pWord;
begin
  n:=StrToInt(ComboBox1.Text);
  a:=CuadradoMagico(n);
  MostrarCuadrado(a,n,StringGrid1);
  FreeMem(a);
end;
```

En ejecución la aplicación tiene la misma apariencia y funcionalidad que en el ejemplo del tema 16.

17.9. EJERCICIOS

1. Cree una aplicación con un Memo, un Edit y un BitBtn. En el evento "onCreate" de la forma haga que el memo tenga 100 puntos de ancho, 300 de alto y que tenga una barra de desplazamiento vertical. Elabore un módulo que empleando punteros genere un vector con números enteros aleatorios comprendidos entre 1 y 10000 y otro que muestre en el memo el vector generado. Programe el evento "onClick" del BitBtn de manera que genere y muestre un vector con el número de elementos escrito en el Edit.
2. Cree una aplicación con un ToolBar, un ImageList, un Memo y un StatusBar. Escriba un módulo que empleando punteros genere un vector con "n" números pares y otro que muestre un vector de números enteros en un memo. En el evento "onCreate" de la forma haga que la forma esté centrada en la pantalla y no tenga bordes, que el ToolBar tenga 3 botones, que el ImageList tenga las imágenes en color de "DoorOpen.bmp", "Retry.bmp" y "Clear.bmp", que las imágenes del ToolBar sean las del ImageList, que el Hint del primer ToolButton sea 'Salir de la aplicación', del segundo sea 'LlenarMemo' y del tercero 'Borrar Memo'; que el Memo tenga 130 puntos de ancho, 250 de alto, que su fondo sea de color amarillo y tenga una barra de desplazamiento vertical, siendo su Hint 'Números enteros consecutivos'; que el StatusBar tenga un solo panel, que muestre automáticamente los hint de los objetos y que el texto inicial y Hint del mismo sea su nombre completo. Programe el evento "onClick" del ToolButton1 de manera que la aplicación se cierre, del ToolButton2, de manera que el memo se llene con 200 números pares y del ToolButton3 de manera que se borre el texto del memo.
3. Cree una aplicación con dos paneles, un StringGrid, dos Label, dos ComboBox y un BitBtn. Escriba un módulo que empleando punteros cree una matriz de números enteros aleatorios comprendidos entre -1000 y 1000 y otro que muestre los elementos de una matriz de números enteros en un StringGrid, con los números de fila y columna en la primera columna y fila del StringGrid. En el evento "onCreate" de la aplicación haga que el primer panel esté alineado en la parte superior de la forma y tenga 40 puntos de alto, que el segundo panel esté alineado en toda la forma, que el StringGrid pertenezca al Panel2, esté alineado en todo el panel, tenga tres filas y tres columnas, que se actualice a medida que se mueven las barras de desplazamiento; que los Label pertenezcan al Panel1, tengan los títulos 'Filas:' y 'Columnas:', sean transparentes, estén ubicados a 13 puntos de la parte superior y 30 y 130 puntos de la izquierda; que los ComboBox pertenezcan al Panel1, tengan 60 puntos de ancho, estén ubicados a 10 puntos de la parte superior y a 5 puntos a la derecha de los Label, que su lista sean los números del 2 al 100 y

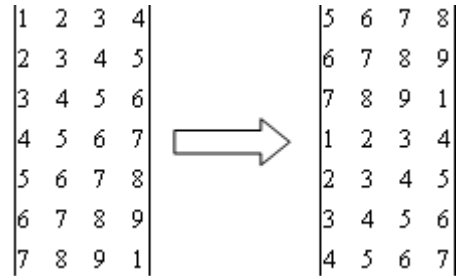
que esté seleccionado el número 2; que el BitBtn pertenezca al panel1, tenga la figura "Calculat.bmp", el título "Generar" y que al hacer clic sobre el mismo se genere y muestre una matriz aleatoria con el número de filas y columnas establecidos en los ComboBox.

4. Cree una aplicación con un Memo, dos BitBtn y dos Edit. Escriba un módulo recursivo que empleando punteros calcule el promedio de un vector de números reales, otro que genere un vector de números reales comprendidos entre 1 y 100, otro que muestre en un memo un vector de números reales y otro que lea de un memo un vector de números reales. En el evento "onCreate" de la forma haga que el memo tenga 120 puntos de ancho, 200 de alto, la letra sea de color azul y tenga una barra de desplazamiento vertical; el primer BitBtn tenga la imagen "Calculat.bmp", el título "&Calcular" y que al hacer clic sobre el mismo lea y calcule el promedio de los números del memo y se muestre el resultado en el Edit1; que el segundo BitBtn tenga la imagen "Retry.bmp", el título "&Generar" y que al hacer clic sobre el mismo se genere y muestre el número de elementos especificado en el Edit2.
5. Cree una aplicación con un StringGrid, dos BitBtn y un Edit. Escriba un módulo recursivo que empleando punteros reciba un vector de números enteros y devuelva un vector con los números impares del vector recibido, otro que genere un vector de números enteros comprendidos entre 1 y 1000, otro que muestre en un StringGrid un vector de números enteros y otro que lea de un StringGrid un vector de números enteros. En el evento "onCreate" de la forma haga que el memo tenga 130 puntos de ancho, 250 de alto, la letra sea de color azul y tenga una barra de desplazamiento vertical; el primer BitBtn tenga la imagen "Report.bmp", el título "&Impares" y que al hacer clic sobre el mismo lea los elementos del memo y deje y muestre sólo los números impares; que el segundo BitBtn tenga la imagen "BulbOn.bmp", el título "&Generar" y que al hacer clic sobre el mismo se genere y muestre el número de elementos especificado en el Edit1.
6. Cree una aplicación con un StringGrid, un BitBtn y un Edit. Escriba un módulo que empleando punteros reciba un número entero comprendido entre 1 y 50 y devuelva un vector con esa cantidad de números primos y otro que muestre el vector en un StringGrid. En el evento "onCreate" de la forma haga que el StringGrid tenga dos columnas, 140 puntos de ancho y 200 de alto; el BitBtn tenga la imagen "Table.bmp", el título "&Primos" y que al hacer clic sobre el mismo genere y muestre la cantidad de números primos especificado en el Edit.
7. Cree una aplicación con 2 Panel, 2 Label, 2 ComboBox, 2 BitBtn y 1 StringGrid. Programe un módulo que empleando punteros reciba una matriz de números enteros y devuelva su transpuesta, otro que lea la matriz desde un StringGrid, otro que muestre la matriz en un StringGrid y otro que genere la matriz de números enteros. En el evento "onCreate" de la forma haga que el primer panel esté alineado en la parte superior de la forma, tenga 30 puntos de alto y no tenga título; que el segundo esté alineado en toda la forma y no tenga título; que el StringGrid pertenezca al Panel2, esté alineado en todo el panel, sus celdas tengan 50 puntos de ancho y tenga 2 filas y 2 columnas; que los Label pertenezcan al Panel1, tengan los títulos "Filas:" y "Columnas:", que sean transparentes, estén a 8 puntos del margen superior y 20 y 140 puntos del margen izquierdo; que los ComboBox pertenezcan al Panel1, tengan 60 puntos de ancho, estén a 5 puntos del margen superior y a 5 puntos de los Label, siendo su lista los números del 1 al 200, estando seleccionado el

- número 2; que los BitBtn pertenezcan al panel1, el primero tenga la figura "DockStack.bmp", el título "&Generar", esté a 250 puntos del margen izquierdo y a 5 del superior, tenga 90 puntos de ancho y que al hacer clic sobre el mismo se genere y muestre una matriz con los números de fila y columna de los ComboBox; que el segundo tenga la figura "Export.bmp", el título "&Transponer", esté a 350 puntos del margen izquierdo y a 5 del superior, tenga 90 puntos de ancho y que al hacer clic sobre el mismo transponga y muestre la matriz tranpuesta.
8. Cree una aplicación con 4 Panel, 2 Label, 2 ComboBox, 2 BitBtn y 3 StringGrid. Programe un módulo que reciba 2 punteros a 2 matrices de números reales y devuelva la suma de las mismas, otro que lea la matriz desde un StringGrid, otro que muestre la matriz en un StringGrid y otro que genere la matriz de números reales. En el evento "onCreate" de la forma haga que el primer panel esté alineado en la parte superior de la forma, tenga 30 puntos de alto y no tenga título; que el segundo esté alineado a la izquierda, tenga un ancho igual a la tercera parte del ancho disponible en la forma y no tenga título; que el tercero esté alineado a la izquierda, tenga un ancho igual a la tercera parte del ancho disponible en la forma y no tenga título; que el cuarto esté alineado en la forma y no tenga título; que el primer StringGrid pertenezca al Panel2, el segundo al Panel3 y el tercero al Panel4, estén alineados en todo el panel, no tenga filas ni columnas fijas y que tengan una fila y una columna; que los Label pertenezcan al Panel1, tengan los títulos "Filas:" y "Columnas:", que sean transparentes, estén a 8 puntos del margen superior y 20 y 140 puntos del margen izquierdo; que los ComboBox pertenezcan al Panel1, tengan 60 puntos de ancho, estén a 5 puntos del margen superior y a 5 puntos de los Label, siendo su lista los números del 1 al 150, estando seleccionado el número 2; que los BitBtn pertenezcan al Panel1, el primero tenga la figura "Edit.bmp", el título "&Generar", esté a 250 puntos del margen izquierdo y a 5 del superior, tenga 80 puntos de ancho y que al hacer clic sobre el mismo se generen y muestren dos matrices, en los dos StringGrid, con el número de filas y columnas de los ComboBox; que el segundo BitBtn tenga la figura "Sum.bmp", el título "&Sumar", esté a 340 puntos del margen izquierdo y a 5 del superior, tenga 80 puntos de ancho y que al hacer clic sobre el mismo lea las matrices de los StringGrid 1 y 2, sume sus elementos y muestre la matriz resultante en el tercer StringGrid.
9. Cree una aplicación con 2 Panels, 1 Label, 1 ComboBox y 1 Memo. Programe un módulo que reciba un número entero comprendido entre 1 y 100 y devuelva un puntero a una matriz con ese número de filas con los elementos del triángulo de Pascal (empleando sólo el número de columnas necesarios en cada fila) y otro que muestre las filas de la matriz en un Memo. En el evento "onCreate" de la forma haga que el primer panel esté alineado en la parte superior de la forma, tenga 30 puntos de alto y no tenga título; que el segundo esté alineado en toda la forma y no tenga título; que el Memo pertenezca al Panel2, esté alineado en toda la forma, tengan barras de desplazamiento vertical y horizontal; que el Label pertenezcan al Panel1, tengan el título "Filas:", que sean transparentes, esté a 8 puntos del margen superior y 30 del margen izquierdo; que el ComboBox pertenezcan al Panel1, tengan 50 puntos de ancho, esté a 5 puntos del margen superior y a 5 puntos del Label, siendo su lista los números del 1 al 100, estando seleccionado el número 1; que el BitBtn pertenezca al Panel1, tenga la figura "BulbOn.bmp", el título "&Generar" y que al hacer clic sobre él se genere y muestre en el Memo el triángulo de Pascal con el número de filas especificado en el Combo-

Box (Las 6 primeras filas del triángulo de Pascal son: 1; 1,1; 1,2,1; 1,3,3,1; 1,4,6,4,1; 1,5,10,10,5,1; 1,6,15,20,15,6,1).

10. En una aplicación se necesita de un módulo que rote (hacia abajo) las filas de una matriz de números enteros, un determinado número de posiciones, un ejemplo del resultado que se debe lograr cuando se rota una matriz 3 posiciones, se presenta al final de la pregunta. Elabore aplicación que empleando punteros lleven a cabo dicha operación.



11. Se quiere intercambiar las diagonales adyacentes a la diagonal principal de una matriz cuadrada, así por ejemplo con la siguiente matriz se debe obtener:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 5 & 3 & 4 \\ 2 & 6 & 10 & 8 \\ 9 & 7 & 11 & 15 \\ 13 & 14 & 12 & 16 \end{bmatrix}$$

Elabore una aplicación que empleando punteros lleve a cabo dicha tarea.

18. ORDENACIÓN: BURBUJA, SELECCIÓN E INSERCIÓN

Como una aplicación de los vectores, en este tema estudiaremos tres métodos de ordenación. El propósito es que al concluir el tema estén capacitados para crear aplicaciones capaces de ordenar vectores de números o cadenas.

Ordenar es arreglar los elementos de un vector (o lista) de acuerdo a un determinado criterio. Por ejemplo para ordenar ascendentemente números reales, el criterio es que todos los elementos anteriores a un elemento dado sean menores o iguales a dicho elemento y que los elementos posteriores sean mayores o iguales al mismo. Si el objetivo es ordenar descendentemente los elementos de una vector con nombres, el criterio sería que todos los elementos anteriores a un elemento dado sean alfabéticamente, mayores o iguales a dicho elemento y que todos los elementos posteriores sean menores o iguales al mismo.

En la mayoría de los casos prácticos los elementos de un vector son ordenados para que sus elementos estén en orden ascendente o descendente. Es por ello que los métodos que estudiaremos en este capítulo nos permitirán ordenar los vectores en una de estas dos formas.

En general los métodos de ordenación suelen clasificarse en dos grupos: a) Los métodos directos, que ordenan comparando elementos consecutivos y b) los métodos indirectos, que ordenan comparando elementos que están separados entre sí por un cierto número de elementos.

En este tema estudiaremos los métodos directos de *Burbuja*, *Selección* e *Inserción*. Los métodos del segundo grupo serán estudiados en el siguiente tema.

Con listas pequeñas (de hasta unos 1000 elementos) no existen diferencia importantes entre los distintos métodos, pero a medida que incrementa el número de elementos la diferencia entre el tiempo consumido por los métodos directos e indirectos también incrementa, hasta que con listas grandes (con cientos de miles o millones de elementos) las diferencias son tan grandes que los métodos directos pueden requerir días y hasta semanas para ordenar una lista mientras que los métodos indirectos pueden ordenarlas en cuestión de minutos.

En todos los métodos se explicará el procedimiento que se sigue para ordenar los elementos de manera ascendente pues para el orden inverso sólo es necesario cambiar la condición o pregunta (invirtiéndola).

18.1 MÉTODO DE BURBUJA

El fundamento de ese método consiste en llevar el mayor valor del vector a la última posición. Para ello se comparan pares consecutivos de elementos (comenzando con los dos primeros) y se realiza un intercambio si el valor del primer elemento es mayor que el segundo.

Una vez que se aplica el anterior procedimiento el mayor valor está ya en su posición correcta (la última), por lo tanto el último elemento está ya ordenado. Entonces cada vez que se aplica el procedimiento queda ordenado un elemento (el último), por lo tanto, para ordenar los "n" elementos de un vector, se debe repetir el procedimiento "n-1" veces, reduciendo en cada repetición el número de elementos a ordenar en uno.

Veamos por ejemplo como se ordena el siguiente vector aplicando el método:

x_1	x_2	x_3	x_4	x_5
5	4	2	1	3

Comenzamos comparando los dos primeros valores (5 con 4) y puesto que el primero es mayor que el del segundo, se efectúa un intercambio:

x_1	x_2	x_3	x_4	x_5
~~5~~ 4	~~4~~ 5	2	1	3

Ahora comparamos el segundo con el tercer elemento (5 con 2) y como el primero es mayor al segundo realizamos otro intercambio:

x_1	x_2	x_3	x_4	x_5
4	~~5~~ 2	~~2~~ 5	1	3

Continuamos con el tercer y cuarto elemento (5 con 1) y puesto que una vez más el primero es mayor que el segundo realizamos otro intercambio:

x_1	x_2	x_3	x_4	x_5
4	2	~~5~~ 1	~~1~~ 5	3

Finalmente comparamos el cuarto y quinto elemento (5 con 3) y como una vez más el primero es mayor que el segundo realizamos otro intercambio:

x_1	x_2	x_3	x_4	x_5
4	2	1	~~5~~ 3	~~3~~ 5

Con ello terminamos el procedimiento y como se puede observar, el mayor valor está ya en su posición correcta (la última):

x_1	x_2	x_3	x_4	x_5
4	2	1	3	5

Ahora repetimos el mismo procedimiento pero sin tomar en cuenta el último elemento ($x_5=5$), pues el mismo ya está ordenado:

x_1	x_2	x_3	x_4	x_5
~~4~~ 2	~~2~~ 4	~~1~~ 4	~~3~~ 4	5

Con lo que ahora queda ordenado el penúltimo elemento:

x_1	x_2	x_3	x_4	x_5
2	1	3	4	5

Repetimos ahora el proceso pero sin tomar en cuenta los dos últimos elementos (pues ya están ordenados):

x_1	x_2	x_3	x_4	x_5
~~2~~ 1	~~1~~ 2	3	4	5

Ahora el antepenúltimo elemento está en su posición correcta:

x_1	x_2	x_3	x_4	x_5
1	2	3	4	5

Repetimos una vez más el procedimiento pero sin tomar en cuenta los tres últimos elementos (que ya están ordenados):

x_1	x_2	x_3	x_4	x_5
1	2	3	4	5

En este caso no se ha realizado ningún intercambio pues el primer elemento no es mayor que el segundo. Dado que ahora sólo nos queda un elemento, la lista ya está ordenada:

x_1	x_2	x_3	x_4	x_5
1	2	3	4	5

Al aplicar el método de burbuja puede suceder que la lista quede ordenada antes de repetir el proceso $n-1$ veces (inclusive la lista puede estar ordenada desde el principio). En el método de burbuja se sabe que la lista está ordenada cuando al aplicar el proceso no se produce ningún intercambio.

Por ejemplo si la lista a ordenar es la siguiente:

x_1	x_2	x_3	x_4	x_5	x_6	x_7
1	2	7	3	4	5	6

Aplicando el procedimiento de burbuja una vez se tiene:

x_1	x_2	x_3	x_4	x_5	x_6	x_7
1	2	~~7~~	~~3~~	~~4~~	~~5~~	~~6~~
		3	4	5	6	7

Con lo que el vector queda ordenado:

x_1	x_2	x_3	x_4	x_5	x_6	x_7
1	2	3	4	5	6	7

Si aplicamos el procedimiento de burbuja una vez más:

x_1	x_2	x_3	x_4	x_5	x_6	x_7
1	2	3	4	5	6	7

No se produce ningún intercambio, lo que como dijimos ocurre en el método de burbuja cuando la lista ya está ordenada. En consecuencia si en una iteración del método no se realiza ningún intercambio es porque la lista ya está ordenada.

Todos los métodos que estudiaremos en este capítulo pueden ser empleados para ordenar vectores con cualquier tipo de datos. Los algoritmos que desarrollaremos en el presente capítulo están pensados en datos simples como números reales, enteros y cadenas (en realidad las cadenas no son datos simples, pero pueden ser tratados como tales). Además, en todos los casos el método será programado empleando parámetros abiertos (o vectores dinámicos) y punteros.

18.1.1 Algoritmo y código empleando vectores dinámicos

El algoritmo del método de Burbuja, empleando vectores dinámicos o parámetros abiertos es el siguiente:

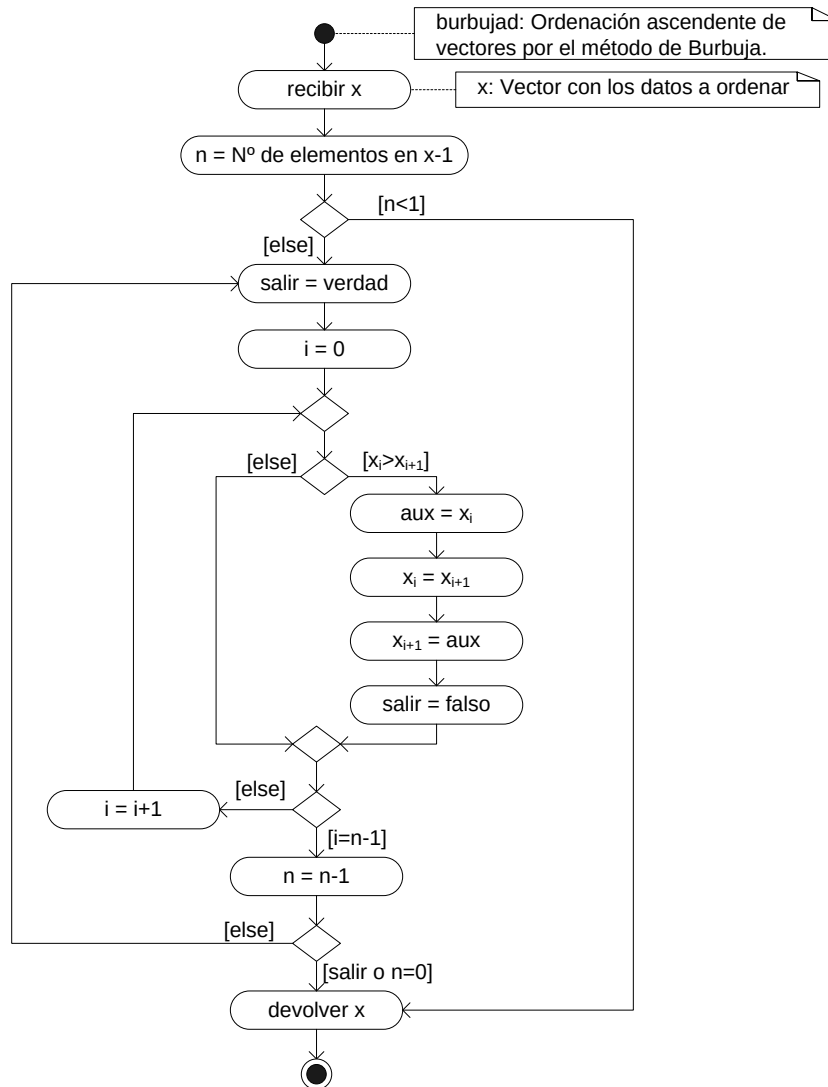


Figura 18.1. Algoritmo para el método de Burbuja (vectores dinámicos)

Para probar este algoritmo se elaborará una aplicación con un StringGrid, un Memo, un Label y dos BitBtns. A más del módulo para el método de burbuja, se programarán módulos para: Generar un vector de números enteros comprendidos entre 1 y 10 veces el número de elementos a generar; Mostrar un vector de números enteros en un StringGrid con dos columnas, mostrando en la primera columna el número de elemento y en la segunda el valor del elemento; Leer un vector desde un StringGrid con dos columnas, estando el vector en la segunda columna; Devolver el tiempo transcurrido entre dos llamadas consecutivas a una función en segundos (con fracciones hasta los milisegundos). La forma debe estar centrada en la pantalla, tener el fondo "GreenBar.bmp", una altura de 500 puntos, un ancho de 340, el título "Método de Burbuja (vectores dinámicos)" y el borde de un cuadro de diálogo. El StringGrid debe estar alineado a la izquierda, tener dos columnas con las cabeceras: "N°" y "Valor", siendo el ancho de la primera columna 60 puntos y el de la segunda 100, el ancho del StringGrid debe ser igual al ancho de sus columnas más 25

puntos para el ScrollBar y su contenido debe ser actualizado dinámicamente con el movimiento de la barra de desplazamiento. Los BitBtns deben tener 100 puntos de ancho y su cursor debe ser una mano apuntando. El primero debe ser de la clase "Retry", tener el título "Generar", estar ubicado a 50 puntos del margen superior y a 20 puntos a la derecha del TStringGrid. El segundo BitBtn debe tener la figura "Sort.bmp", el título "Ordenar", estar ubicado a 100 puntos del margen superior y a 20 puntos a la derecha del TStringGrid. El Memo debe tener 21 puntos de alto y 100 de ancho, estar alineado a la izquierda con los BitBtn, a 170 puntos del margen superior y no permitir la adición de líneas con la tecla "Enter". El Label debe tener el título "Tiempo empleado:", ser transparente, estar alineado a la izquierda con el memo y a 15 puntos por encima del mismo.

La unidad donde se han escrito los módulos antes indicados es:

```
unit uBurbujaD;
```

```
interface
```

```
  uses Grids, StdCtrls, Math, DateUtils, SysUtils;
```

```
  type tvEnteros= array of Integer;
```

```
  function GenerarVector(n:cardinal):tvEnteros;
  procedure MostrarVector(x: array of Integer; s:TStringGrid);
  function LeerVector(s:TStringGrid):tvEnteros;
  procedure Burbuja(var x: array of Integer);
  function Tiempo:real;
```

```
implementation
```

```
  function GenerarVector(n:cardinal):tvEnteros;
  var i,d: cardinal; x:tvEnteros;
  begin
    if n=0 then raise EInvalidArgument.Create(
      'El número de elementos debe ser mayor a cero');
    d:=10*n;
    SetLength(x,n);
    for i:=0 to n-1 do
      x[i]:=Random(d)+1;
    result:=x;
  end;

  procedure MostrarVector(x: array of Integer; s:TStringGrid);
  var i,n:cardinal;
  begin
    n:=Length(x);
    s.RowCount:=n+1;
    for i:=1 to n do begin
      s.Cells[0,i]:= Format('%15d',[i]);
      s.Cells[1,i]:= Format('%20d',[x[i-1]]);
    end;
  end;

  function LeerVector(s:TStringGrid):tvEnteros;
  var i,n:cardinal; x:tvEnteros;
  begin
    n:=s.RowCount-1;
    if n=0 then raise EInvalidArgument.Create(
```

```

    'El StringGrid no tiene datos');
    SetLength(x,n);
    for i:=1 to n do
        x[i-1]:=StrToInt(s.Cells[1,i]);
    result:=x;
end;

procedure Burbuja(var x: array of Integer);
var i,n,aux:integer; salir:boolean;
begin
    n:=High(x);
    if n<1 then exit;
    repeat
        salir:=True;
        for i:=0 to n-1 do
            if x[i]>x[i+1] then begin
                aux:=x[i];
                x[i]:=x[i+1];
                x[i+1]:=aux;
                salir:=False;
            end;
        dec(n);
    until salir or (n=0);
end;

function Tiempo:real;
const t:TDateTime=0;
begin
    if t=0 then begin
        t:=Now; result:=0;
    end
    else begin
        result:=MillisecondsBetween(Now,t)/1000;
        t:=0;
    end;
end;

end.

```

El código del evento "onCreate" de la forma (donde se da formato a los componentes de la misma) y de los eventos "onClick" de los botones, donde se genera y ordena la lista de números enteros, son los siguientes:

implementation

```
uses uBurbujaD;
```

```
{ $R *.dfm }
```

```

procedure TForm1.FormCreate(Sender: TObject);
begin
    Form1.Position:=poScreenCenter;
    Form1.Brush.Bitmap:=TBitmap.Create;
    Form1.Brush.Bitmap.LoadFromFile('C:\Archivos de programa\'+
        'Archivos comunes\Borland Shared\Images\Backgrnd\GreenBar.bmp');
    Form1.Height:=500;
    Form1.Width:=340;
    Form1.Caption:='Método de Burbuja (vectores dinámicos)';
    Form1.BorderStyle:=bsDialog;

```

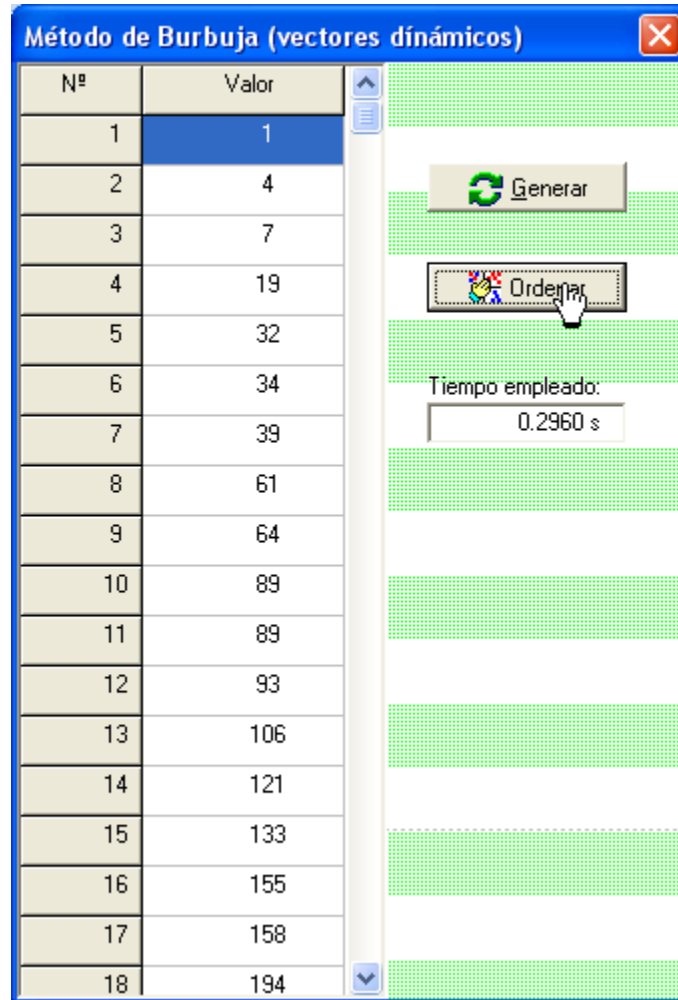
```
StringGrid1.Align:=alLeft;
StringGrid1.ColCount:=2;
StringGrid1.Cells[0,0]:=Format('%8s', ['N°']);
StringGrid1.Cells[1,0]:=Format('%18s', ['Valor']);
StringGrid1.ColWidths[0]:=60;
StringGrid1.ColWidths[1]:=100;
StringGrid1.Width:=StringGrid1.ColWidths[0]+StringGrid1.ColWidths[1]+25;
StringGrid1.Options:=StringGrid1.Options+[goThumbTracking];
BitBtn1.Width:=100;
BitBtn1.Kind:=bkRetry;
BitBtn1.Caption:='&Generar';
BitBtn1.Left:=StringGrid1.Width+20;
BitBtn1.Top:=50;
BitBtn2.Width:=100;
BitBtn1.Cursor:=crHandPoint;
BitBtn2.Glyph.LoadFromFile('C:\Archivos de programa\Archivos comunes\'+'
  'Borland Shared\Images\Buttons\Sort.bmp');
BitBtn2.NumGlyphs:=2;
BitBtn2.Caption:='&Ordenar';
BitBtn2.Left:=BitBtn1.Left;
BitBtn2.Top:=100;
BitBtn2.Cursor:=crHandPoint;
Memo1.Height:=21;
Memo1.Width:=100;
Memo1.WantReturns:=False;
Memo1.Text:='0';
Memo1.Left:= BitBtn1.Left;
Memo1.Top:=170;
Label1.Caption:='Tiempo empleado:';
Label1.Transparent:=True;
Label1.Left:=Memo1.Left;
Label1.Top:=Memo1.Top-15;
end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  Form1.Brush.Bitmap.Free;
end;

procedure TForm1.BitBtn1Click(Sender: TObject);
var x: tvEnteros;
begin
  x:=GenerarVector(10000);
  MostrarVector(x,StringGrid1);
  Finalize(x);
end;

procedure TForm1.BitBtn2Click(Sender: TObject);
var x:tvEnteros;
begin
  x:=LeerVector(StringGrid1);
  tiempo;
  Burbuja(x);
  Memo1.Text:=Format('%20.4f s', [tiempo]);
  MostrarVector(x,StringGrid1);
  Finalize(x);
end;
end.
```

En ejecución, la aplicación tiene la apariencia que se muestra en la siguiente figura:



13.1.2 Algoritmo y código empleando punteros

El algoritmo del método de Burbuja cuando se trabaja con punteros se presenta en el diagrama de actividades de la siguiente página.

La interfaz de la aplicación es la misma que del acápite anterior, por lo que no volveremos a mostrar el código del evento "onCreate" ni "onClose" de la misma.

La unidad con los mismos módulos del acápite anterior, pero empleando punteros (excepto en la función tiempo) es la siguiente:

```
unit uBurbujaP;
```

```
interface
```

```
  uses Grids, StdCtrls, Math, DateUtils, SysUtils;
```

```
  type pEnteros = ^Integer;
```

```
  function GenerarVector(n: cardinal): pEnteros;
```

```
  procedure MostrarVector(x: pEnteros; n: cardinal; s: TStringGrid);
```

```
  procedure LeerVector(s: TStringGrid; out x: pEnteros; out n: cardinal);
```

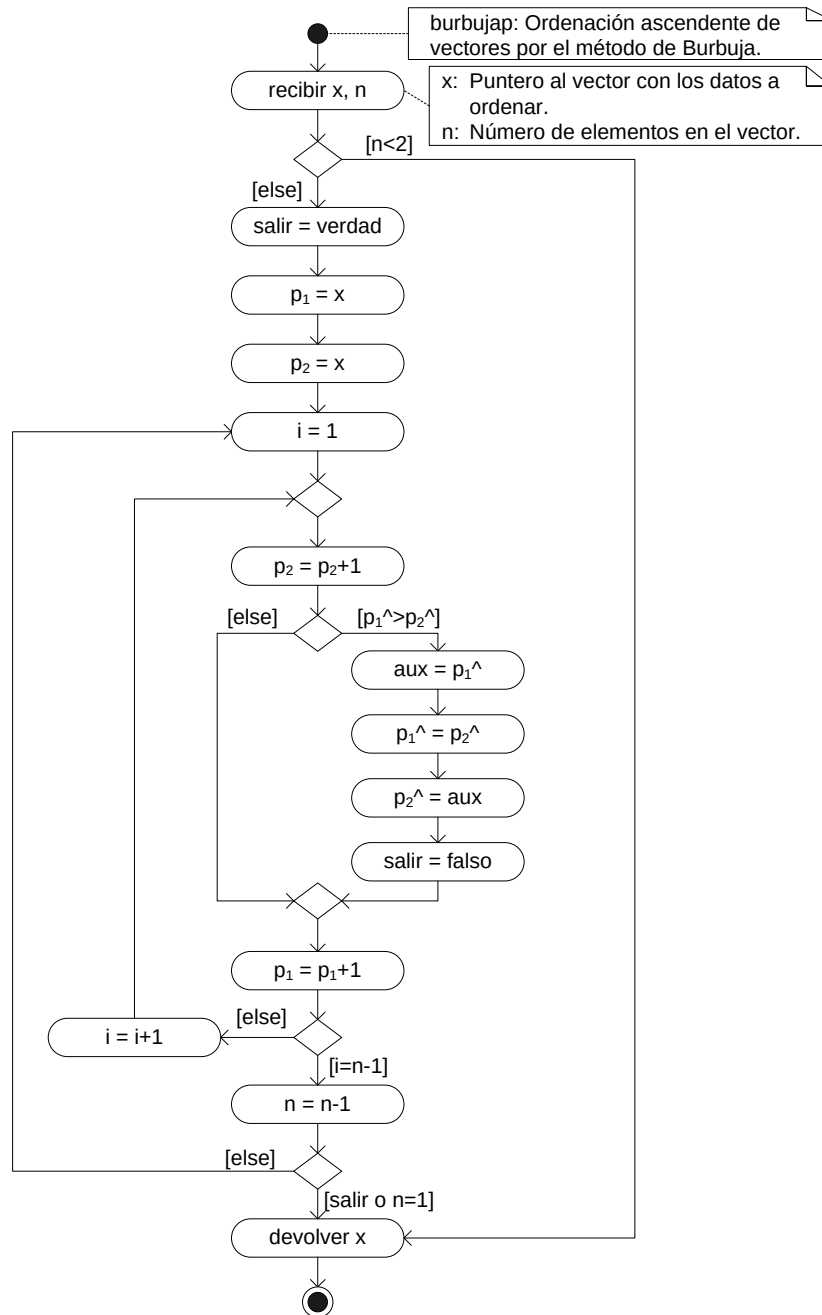


Figura 18.2. Algoritmo para el método de Burbuja (punteros)

```

procedure Burbuja(x:pEnteros; n:cardinal);
function Tiempo:real;

```

implementation

```

function GenerarVector(n:cardinal):pEnteros;
var i,d: cardinal; p,x:pEnteros;
begin
  if n=0 then raise EInvalidArgument.Create(
    'El número de elementos debe ser mayor a cero');
  d:=10*n;
  GetMem(x,n*SizeOf(x^));

```

```

    p:=x;
    for i:=1 to n do begin
        p^:=Random(d)+1;
        inc(p);
    end;
    result:=x;
end;

procedure MostrarVector(x: pEnteros; n:cardinal; s:TStringGrid);
var i:cardinal;
begin
    s.RowCount:=n+1;
    for i:=1 to n do begin
        s.Cells[0,i]:= Format('%12d',[i]);
        s.Cells[1,i]:= Format('%20d',[x^]);
        inc(x);
    end;
end;

procedure LeerVector(s:TStringGrid; out x:pEnteros; out n:cardinal);
var i:cardinal; p:pEnteros;
begin
    n:=s.RowCount-1;
    if n=0 then raise EInvalidArgument.Create(
        'El StringGrid no tiene datos');
    GetMem(x,n*SizeOf(x^));
    p:=x;
    for i:=1 to n do begin
        p^:=StrToInt(s.Cells[1,i]);
        inc(p);
    end;
end;

procedure Burbuja(x:pEnteros; n:cardinal);
var i,aux:integer; p1,p2:pEnteros; salir:boolean;
begin
    if n<2 then exit;
    repeat
        salir:=True;
        p1:=x;
        p2:=x;
        for i:=1 to n-1 do begin
            inc(p2);
            if p1^>p2^ then begin
                aux:=p1^;
                p1^:=p2^;
                p2^:=aux;
                salir:=False;
            end;
            inc(p1);
        end;
        dec(n);
    until salir or (n=1);
end;

function Tiempo:real;
const t:TDateTime=0;
begin

```

```

    if t=0 then begin
        t:=Now; result:=0;
    end
    else begin
        result:=MillisecondsBetween(Now,t)/1000;
        t:=0;
    end;
end;

end.

```

En la unidad principal (la unidad de la forma) sólo cambian los eventos "onClick" de los botones y la instrucción "uses":

```
uses uBurbujaP;
```

```

procedure TForm1.BitBtn1Click(Sender: TObject);
var x: pEnteros; const n=10000;
begin
    x:=GenerarVector(n);
    MostrarVector(x,n,StringGrid1);
    FreeMem(x);
end;

```

```

procedure TForm1.BitBtn2Click(Sender: TObject);
var x:pEnteros; n:cardinal;
begin
    LeerVector(StringGrid1,x,n);
    tiempo;
    Burbuja(x,n);
    Memo1.Text:=Format('%20.4f s',[tiempo]);
    MostrarVector(x,n,StringGrid1);
    FreeMem(x);
end;

```

En ejecución la aplicación tiene la misma apariencia que en el anterior acápite (con la única diferencia de que el título de la forma deberá ser ahora: "Método de Burbuja (punteros)").

18.2 MÉTODO DE SELECCIÓN

En este método se selecciona primero el mayor valor y se intercambia con el último elemento, de esta manera (al igual que sucedía con el método de *Burbuja*), el mayor valor queda siempre en la última posición por lo tanto el último elemento queda con el valor correcto.

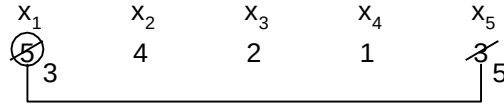
Luego se vuelve a aplicar el procedimiento pero sin tomar en cuenta el último elemento (pues ya está con el valor correcto), entonces el penúltimo elemento queda también con el valor correcto. Como cada vez que se aplica el procedimiento queda ordenado un elemento, el mismo se repite $n-1$ veces tomando en cada repetición un elemento menos.

Este método suele ser un tanto más eficiente que el método de Burbuja, pues requiere menos intercambios, sin embargo, en este método no existe ningún indicador que nos permita determinar si la lista ya está ordenada, por lo que siempre se repite $n-1$ veces inclusive si la lista ya está ordenada.

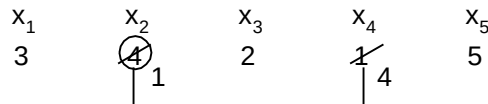
Para comprender mejor el método ordenemos la siguiente lista aplicando el método.

x_1	x_2	x_3	x_4	x_5
5	4	2	1	3

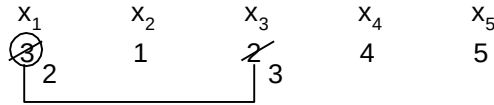
Seleccionamos el mayor de valor de la lista (el número 5) e intercambiamos el mismo con el último elemento:



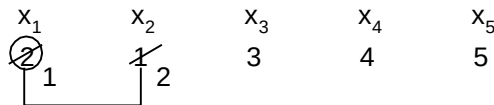
Ahora repetimos el procedimiento sin tomar en cuenta el último elemento (pues ya está ordenado):



Repetimos el procedimiento sin tomar en cuenta ahora los dos últimos elementos (que ya están ordenados):



Finalmente repetimos una vez más el procedimiento sin tomar en cuenta los tres últimos elementos (que ya están ordenados):



Con lo que el vector queda ordenado:

x_1	x_2	x_3	x_4	x_5
1	2	3	4	5

18.2.1 Algoritmo y código empleando vectores dinámicos

El algoritmo que automatiza el procedimiento, tomando en cuenta vectores dinámicos o parámetros abiertos se presenta en la Figura 18.3.

Para todos los métodos que se estudien en este capítulo las aplicaciones serán esencialmente las mismas que en el primer método, por lo que sólo se mostrarán los módulos que presentan algún cambio, así en este caso, el único módulo que tiene cambios importantes es el del método:

```

procedure Seleccion(var x: array of Integer);
var i,k,n,aux:integer;
begin
  n:=Length(x);
  if n<2 then exit;
  repeat
    k:=0;
    for i:=1 to n-1 do
      if x[i]>x[k] then k:=i;
    if k<>n-1 then begin
      aux:=x[k];
      x[k]:=x[n-1];

```


$x[n-1] := aux;$

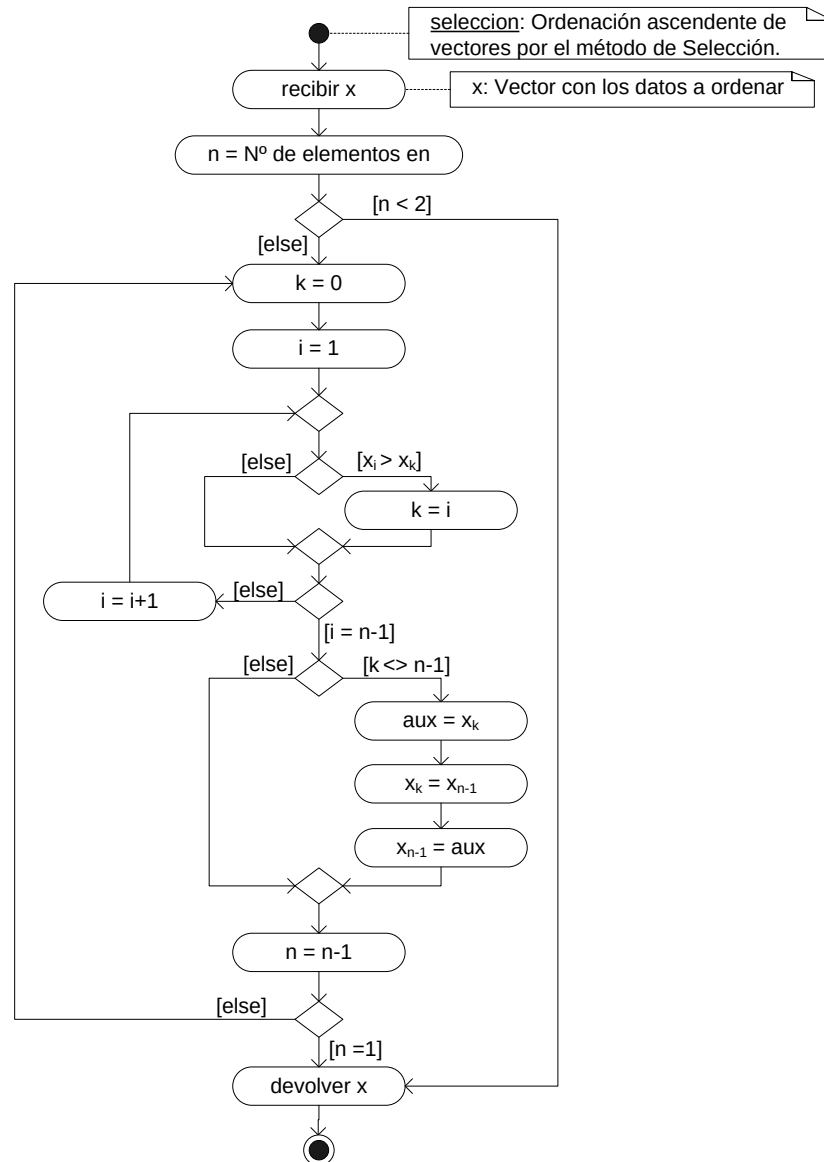


Figura 18.3. Algoritmo del método de Selección (Vectores dinámicos)

```

end;
dec(n);
until n=1;
end;

```

Los otros dos módulos que tienen ligeros cambios son: el evento "onClick" del BitBtn 2 (Ordenar), donde en lugar de llamar al método de Burbuja se llama al método de Selección y el evento "onCreate" de la forma, donde se cambia el título (Caption) a "Método de Selección (vectores dinámicos)".

18.2.2 Algoritmo y código empleando punteros

Al ser los métodos de ordenación iterativos, todos ellos pueden ser programados también empleando la recursividad. Así el algoritmo de este método, empleando punteros y llamadas recursivas, es el que se muestra en la siguiente página:

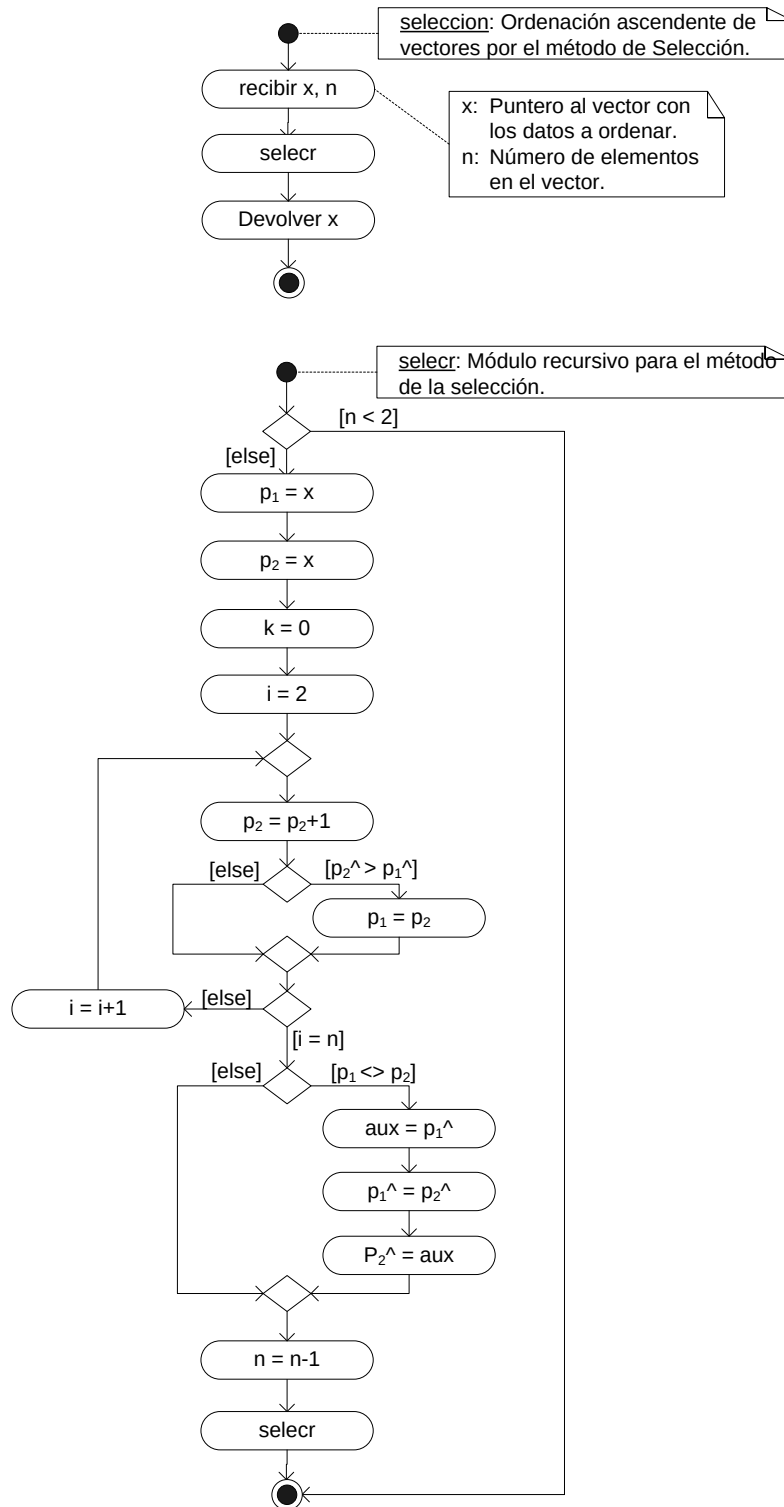


Figura 18.4. Algoritmo del método de Selección (Punteros-Recursivo).

Observe que el módulo que realmente implementa el método en forma recursiva es *Selecr*, el cual es un submódulo de *Seleccion*. Observe también que las llamadas recursivas se emplean sólo para repetir el método, pero en realidad no existe un razonamiento recursivo.

El código correspondiente es el siguiente:

```

procedure Seleccion(x:pEnteros; n:cardinal);
var aux:integer; p1,p2:pEnteros;
procedure Selecr;
var i:cardinal;
begin
  if n<2 then exit;
  p1:=x;
  p2:=x;
  for i:=2 to n do begin
    inc(p2);
    if p2^>p1^ then p1:=p2;
  end;
  if p1<>p2 then begin
    aux:=p1^;
    p1^:=p2^;
    p2^:=aux;
  end;
  dec(n);
  selecr;
end;
begin
  selecr;
end;

```

Al igual que en el caso de los vectores dinámicos, la aplicación es prácticamente la misma que la elaborada para el método de Burbuja con punteros, razón por la cual no se vuelve a mostrar los otros módulos.

18.3 MÉTODO DE INSERCIÓN

En este método se inserta un elemento en su posición correcta con relación a los elementos que se encuentran antes del mismo. Para ello es necesario que los elementos ubicados antes del elemento estén ordenados, por lo tanto se inserta primero el segundo elemento (para que queden ordenados los dos primeros), luego el tercero (para que queden ordenados los tres primeros) y así sucesivamente hasta insertar el último elemento del vector.

En la práctica este método se emplea para insertar nuevos elementos en vectores ya ordenados y no para ordenar todo el vector, no obstante, en este capítulo implementaremos el método de manera que permita ordenar todo el vector.

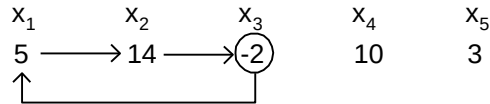
Para comprender mejor el método ordenemos el siguiente vector aplicando el procedimiento:

x_1	x_2	x_3	x_4	x_5
5	14	-2	10	3

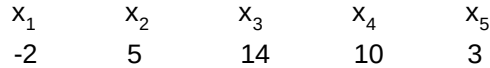
Comenzamos con el segundo elemento (que tiene el valor 14) y como es mayor al primero está en su posición correcta, razón por la cual no es necesario efectuar ningún intercambio:

x_1	x_2	x_3	x_4	x_5
5	(14)	-2	10	3

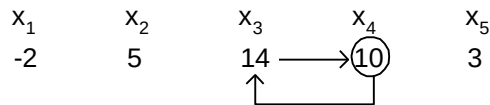
Ahora pasamos el tercer elemento (que tiene el valor -2) y como es menor a los dos elementos anteriores (que ya están ordenados), el tercer elemento debe ser insertado en la primera posición, para ello es necesario desplazar los elementos anteriores una posición hacia la derecha:



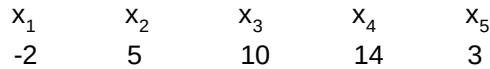
Quedando los tres primeros elementos ordenados:



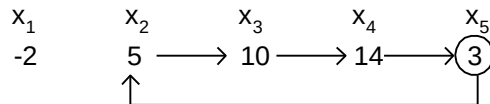
Ahora pasamos al cuarto elemento (que tiene el valor 10) y como este elemento es menor al tercero, pero mayor al segundo, debe ser insertado en la tercera posición, siendo necesario desplazar el tercer elemento una posición hacia la derecha:



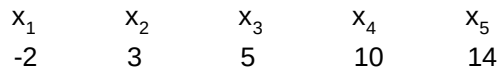
Quedando los cuatro primeros elementos ordenados:



Finalmente pasamos al quinto elemento (que tiene el valor 3) y como este elemento es menor al segundo elemento, pero mayor al primero, debe ser insertado en la segunda posición, siendo necesario desplazar los elementos 2 al 4 una posición a la derecha:



Quedando así los 5 elementos ordenados:



18.3.1 Algoritmo y código empleando vectores dinámicos

El algoritmo para el método de inserción se presenta en la Figura 18.5, siendo el código del módulo respectivo el siguiente:

```

procedure Insercion(var x: array of Integer);
var n, ne, j, aux: integer;
begin
  ne:= High(x);
  if ne<1 then exit;
  n:= 1;
  repeat
    j:= n;
    aux:= x[n];
    while (j>0) and (aux<x[j-1]) do
      begin
        x[j]:= x[j-1];
        dec(j);
      end;
  end;

```

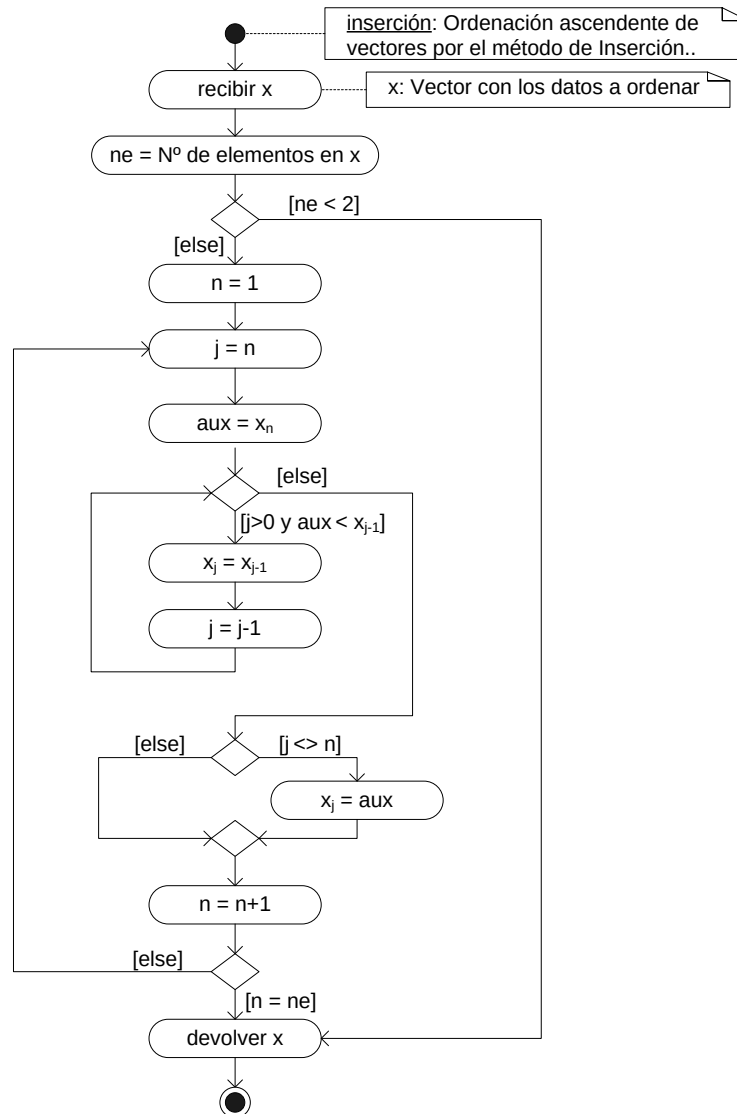


Figura 18.5. Algoritmo del método de Inserción (Vectores Dinámicos)

```

if j<>n then x[j]:= aux;
  inc(n);
until n>ne;
end;

```

Como en el método anterior, la aplicación es esencialmente la misma que para el método de burbuja, razón por la cual no se presentan los otros módulos de la misma.

18.3.2 Algoritmo y código empleando punteros

El algoritmo del método trabajando con punteros y llamadas recursivas se presenta en la siguiente página y el código del módulo respectivo es el siguiente:

```

procedure Insercion(x:pEnteros; ne:cardinal);
var aux:integer; p1,p2,p3:pEnteros;
procedure inserr(p:pEnteros);
begin

```

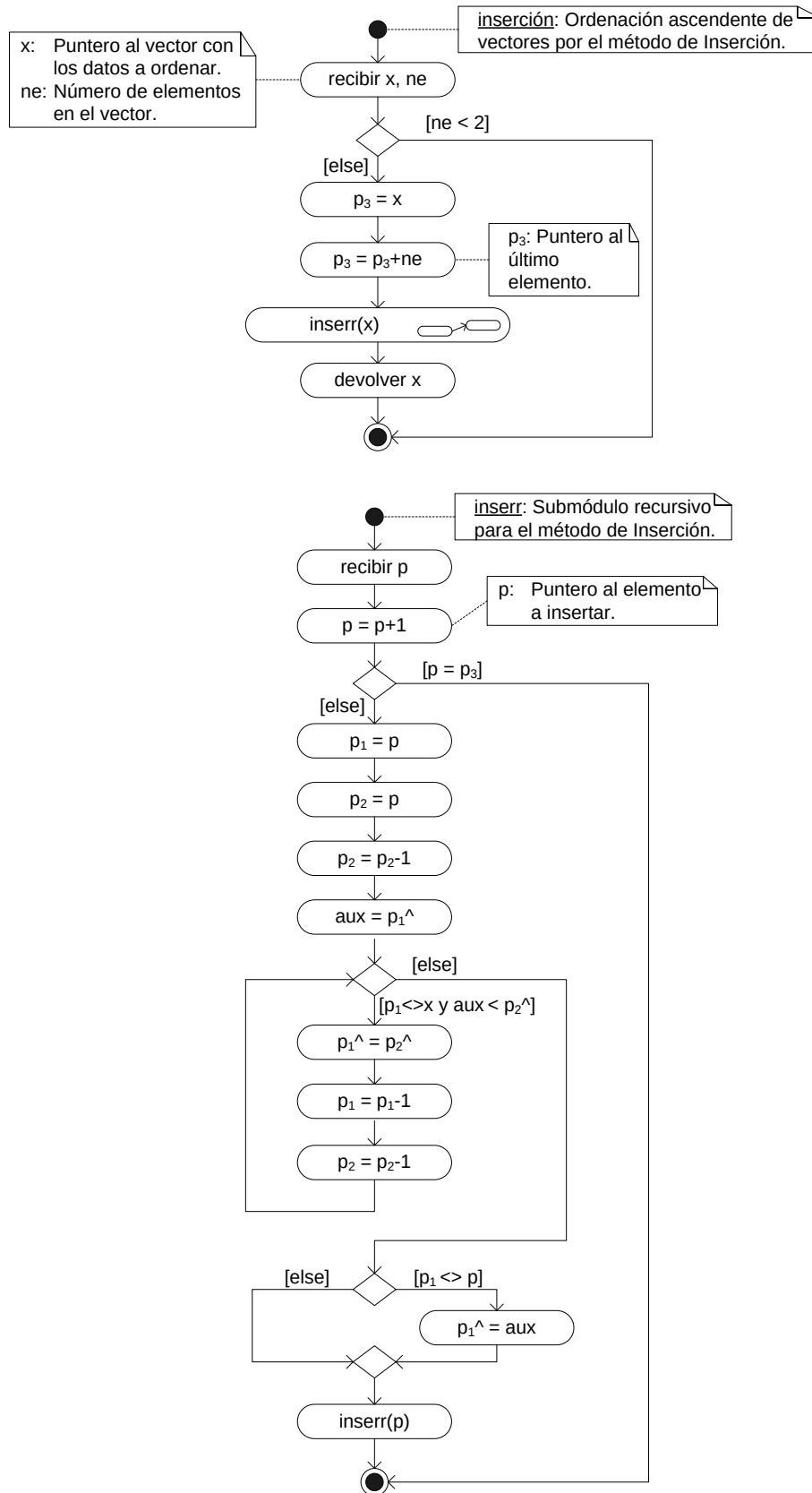


Figura 18.6. Algoritmo del método de Inserción (Punteros)

```
    inc(p);  
    if p=p3 then exit;  
    p1:=p;  
    p2:=p;  
    dec(p2);  
    aux:=p1^;  
    while (p1<>x) and (aux<p2^) do begin  
        p1^:=p2^;  
        dec(p1);  
        dec(p2);  
    end;  
    if p1<>p2 then p1^:=aux;  
    inserr(p);  
end;  
begin  
    p3:=x;  
    inc(p3,ne);  
    inserr(x);  
end;
```

Una vez más, la aplicación es esencialmente la misma que para el método de burbuja, razón por la cual no se presentan los otros módulos de la misma.

18.4 EJERCICIOS

1. Elabore una aplicación para ordenar vectores de números reales por el método de Burbuja, trabajando con punteros y empleando la recursividad.
2. Elabore una aplicación para ordenar vectores de números reales por el método de Selección, trabajando con vectores dinámicos (y/o parámetros abiertos) y empleando la recursividad.
3. Elabore una aplicación para ordenar vectores de números reales por el método de Inserción, trabajando con punteros (no recursivo).

19. ORDENACIÓN: SHELL Y QUICK SORT

Como ya se habíamos mencionado en el anterior tema, ahora estudiaremos el grupo de los métodos indirectos (es decir aquellos en los que no se comparan valores sucesivos) y de este grupo estudiaremos dos métodos: Shell y Quick Sort.

19.3 MÉTODO SHELL

El método *Shell*, denominado así en honor a su creador *D. L. Shell*, es el primero de los métodos indirectos que estudiaremos en este capítulo.

En este método se comparan valores que están separados entre sí por un determinado número de elementos. Al igual que en el método de burbuja si el primer valor es mayor que el segundo se lleva a cabo un intercambio.

Al número de elementos que separan los valores a comparar se conoce con el nombre de *salto* (o *paso*) e inicialmente es igual a la mitad (entera) del número de elementos existentes.

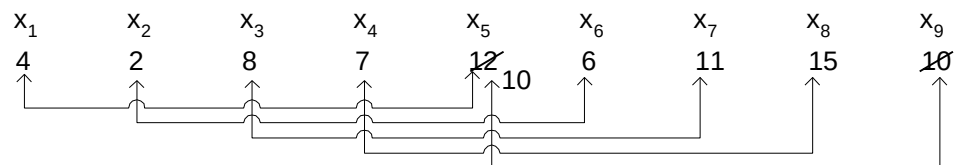
Si k es el paso o salto, en el proceso se comparan el primer elemento con el elemento que se encuentra $k+1$ elementos más adelante, luego se compara el segundo elemento con el elemento que se encuentra $k+2$ elementos más adelante y así sucesivamente hasta que se compara el último elemento del vector (es decir se realizan $k-p$ comparaciones)

El anterior procedimiento se repite hasta que no ocurre ningún intercambio, entonces el *paso* o *salto* es reducido a la mitad y el procedimiento se vuelve a repetir (con el nuevo *paso*) hasta que una vez más no existen más intercambio. Cuando esto ocurre se vuelve a reducir el valor del *paso* a la mitad y se continúa de esta manera hasta que el *paso* se reduce a 1.

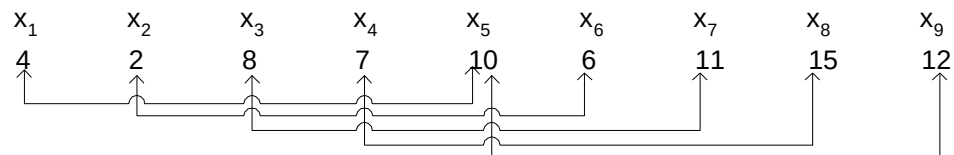
Para comprender mejor el procedimiento ordenaremos el siguiente vector siguiendo el método Shell:

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
4	2	8	7	12	6	11	15	10

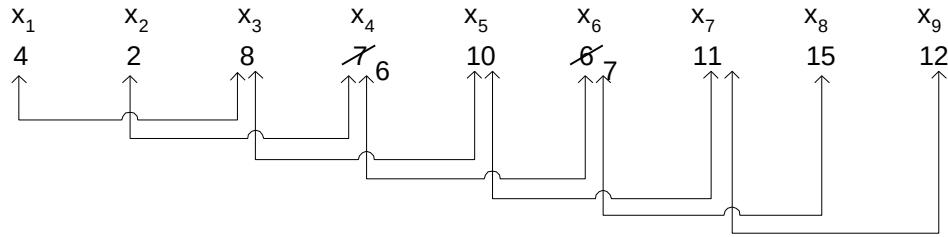
Como el número de elementos es 9, el paso inicial es $9/2=4$ (división entera). Entonces comparamos el primer elemento con el quinto, el segundo con el sexto y así sucesivamente:



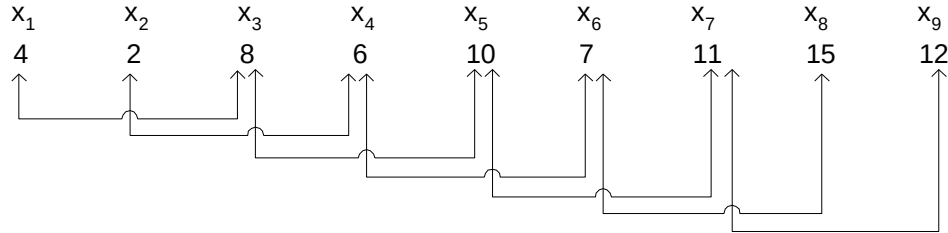
Y como se ha producido un intercambio repetimos el procedimiento:



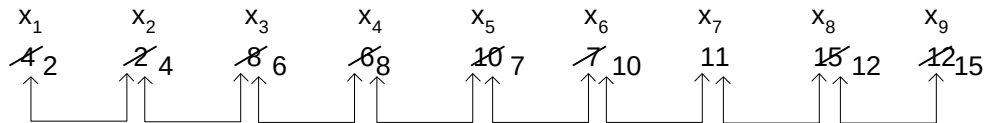
Como ahora no se produce ningún intercambio, reducimos el paso a la mitad de su valor. Por lo tanto el nuevo paso será: $4/2=2$, entonces comparamos x_1 con x_3 , x_2 con x_4 y así sucesivamente:



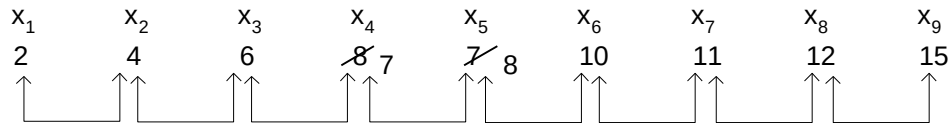
Como se ha producido un intercambio repetimos el proceso:



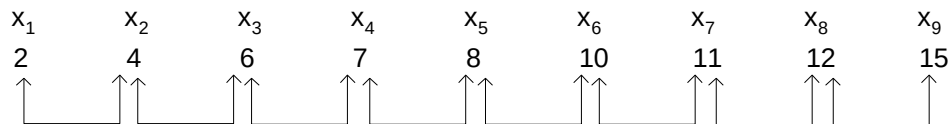
Y dado que no se produce ningún intercambio, reducimos el paso a la mitad: $2/2=1$. Entonces comparamos elementos sucesivos: el primero con el segundo, el segundo con el tercero, etc.:



Como han existido intercambios, se repite el procedimiento:



Como se ha producido un intercambio se sigue repitiendo el procedimiento:



Puesto que ahora no se produce ningún intercambio y dado que el paso es ya 1, el proceso concluye, estando la lista ya ordenada.

Este método es más eficiente que los métodos estudiados hasta ahora porque requiere un menor número de intercambios. Así, inclusive con el vector de tan sólo 9 elementos del ejemplo, se requieren 7 intercambios, mientras que con *Burbuja* habrían sido necesarios al menos 9. Esta diferencia es más grande cuanto mayor es el número de elementos del vector.

19.3.1 Algoritmo y código empleando vectores dinámicos

El algoritmo del método se presenta en el diagrama de actividades de la siguiente página (Figura 19.1).

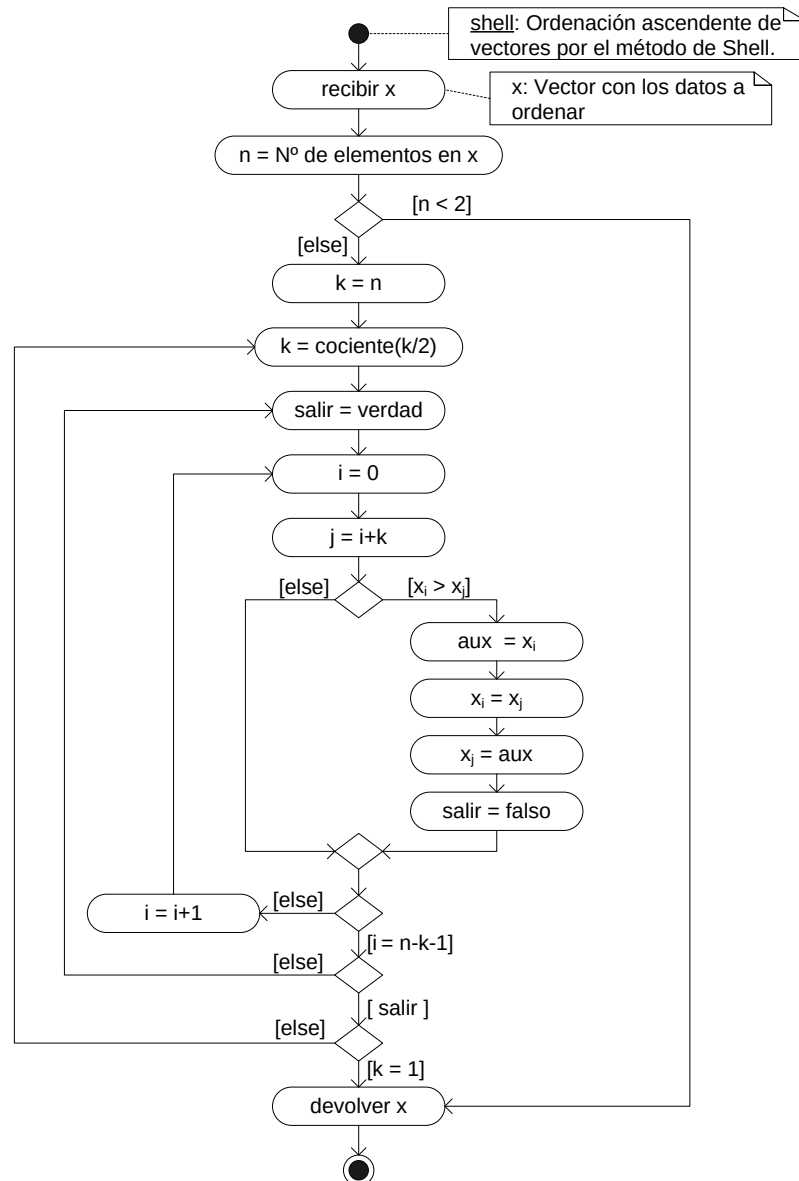


Figura 19.1. Algoritmo del método Shell (Vectores Dinámicos)

El código del módulo respectivo es el siguiente:

```

procedure Shell(var x: array of Integer);
var i,j,k,n:cardinal; aux:integer; salir: boolean;
begin
  n:= Length(x);
  if n<2 then exit;
  k:=n;
  repeat
    k:=k div 2;
    repeat
      salir:= true;
      for i:=0 to n-k-1 do begin
        j:= i+k;
        if x[i]>x[j] then begin
          aux:= x[i];
          x[i]:= x[j];
          x[j]:= aux;
        end
      end
      if salir then break;
    until k=1;
  until k=1;
end

```

```

        x[j]:= aux;
        salir:= false;
    end;
end;
until salir;
until k=1;
end;

```

Una vez más, la aplicación es esencialmente la misma que para el método de burbuja (presentada en el tema anterior), razón por la cual no se presentan los otros módulos de la misma.

19.3.2 Algoritmo y código empleando punteros

El algoritmo del método *Shell*, cuando se trabaja con punteros se presenta en la siguiente página y el código respectivo es el siguiente:

```

procedure Shell(x:pEnteros; n:cardinal);
var i,k:cardinal; aux:integer; p1,p2:pEnteros; salir:boolean;
begin
    if n<2 then exit;
    k:=n;
    repeat
        k:=k div 2;
        repeat
            p1:=x;
            p2:=x;
            inc(p2,k);
            salir:=True;
            for i:=1 to n-k do begin
                if p1^>p2^ then begin
                    aux:=p1^;
                    p1^:=p2^;
                    p2^:=aux;
                    salir:=False;
                end;
                inc(p1);
                inc(p2);
            end;
        until salir;
    until k=1;
end;

```

Una vez más, la aplicación es esencialmente la misma que para el método de burbuja, razón por la cual no se presentan los otros módulos de la misma.

19.4 MÉTODO QUICK SORT (HOARE)

Este método fue inventado por C. R. Hoare y es el método de ordenación más eficiente de todos los estudiados en este y en el anterior tema.

Básicamente el método consiste en dividir el vector en dos: uno con elementos menores o iguales a un valor de referencia denominado *pívote* y otro con elementos mayores o iguales a dicho valor.

El anterior procedimiento se repite con cada uno de los dos vectores resultantes y luego con cada uno de los vectores resultantes de estos y así sucesivamente hasta que cada vector queda con un solo elemento.

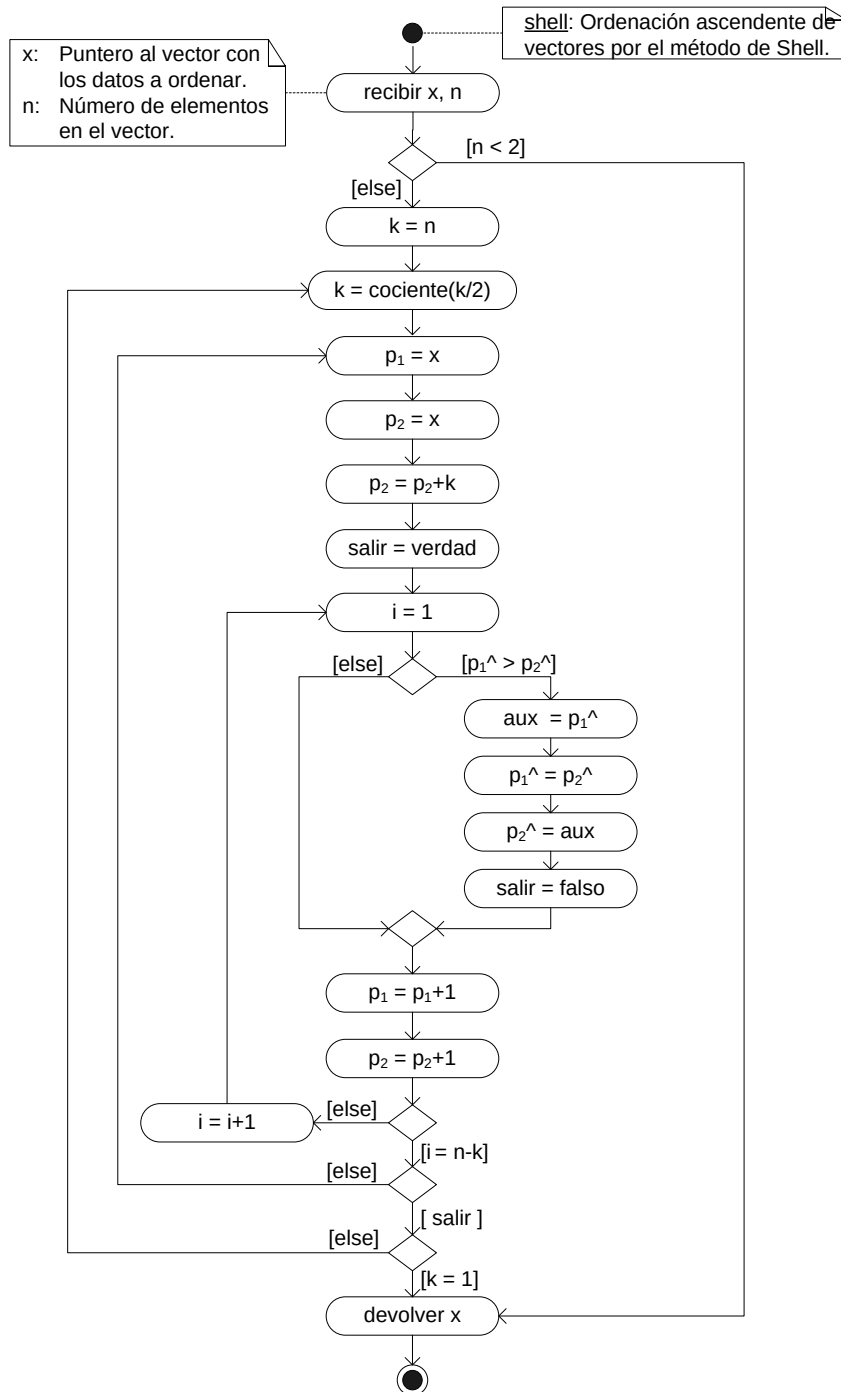


Figura 19.2. Algoritmo del método Shell (Punteros)

El valor de referencia, el valor *pivote*, puede ser cualquiera de los elementos del vector, aunque frecuentemente se elige el elemento central, el primer elemento o el último elemento. Al finalizar la división, el *pivote* queda en uno de los dos vectores resultantes o al medio de los dos, en cuyo caso se encuentra en ya en la posición correcta.

Para comprender mejor el procedimiento ordenaremos la siguiente lista empleando el método *Quick Sort*:

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
15	21	30	7	20	18	10	14	28

Elegiremos como *pivote* el primer elemento, por lo tanto nuestro *pivote* será 15 (note que el pivote es el valor del primer elemento, no el elemento en sí, porque en el proceso su valor puede ir cambiando). Ahora debemos dividir la lista en 2, uno con elementos mayores a 15 y otro con elementos menores a 15. Para ello empleamos dos contadores: uno que comienza en el índice más bajo del vector (en nuestro caso 1) y otro en el índice más alto (en nuestro caso 9):

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	
15	21	30	7	20	18	10	14	28	pivote = 15
$i=1$									$j=9$

Ahora incrementamos el valor del contador i mientras x_i sea menor al valor pivote. Como en este caso x_i es igual al pivote, no incrementamos su valor. Entonces pasamos a disminuir el valor del contador j , en este caso mientras x_j sea mayor al pivote, por lo tanto el contador j disminuye hasta x_8 , donde el valor (14) es menor al pivote:

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	
15	21	30	7	20	18	10	14	28	pivote = 15
$i=1$							$j=8$		$j=9$

Entonces se intercambian x_i con x_j , (x_1 con x_8) y se incrementa el contador i en uno y se disminuye el contador j en uno:

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	
14	21	30	7	20	18	10	15	28	pivote = 15
$i=1$	$i=2$						$j=8$		$j=7$

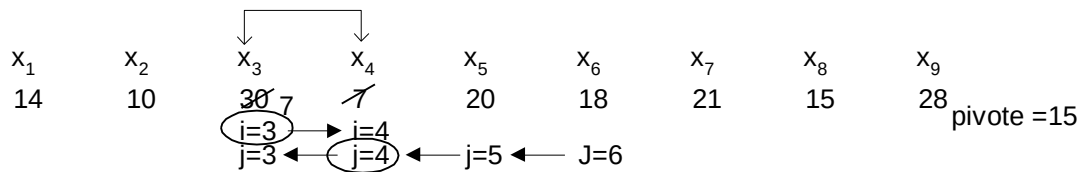
Ahora volvemos a repetir el proceso, es decir incrementamos el contador i mientras x_i sea menor al pivote y disminuimos j mientras x_j sea mayor al pivote. En este caso x_i es ya mayor al pivote y x_j menor, por lo tanto no se incrementa ni se disminuyen sus valores:

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	
14	21	30	7	20	18	10	15	28	pivote = 15
	$i=2$					$j=7$			

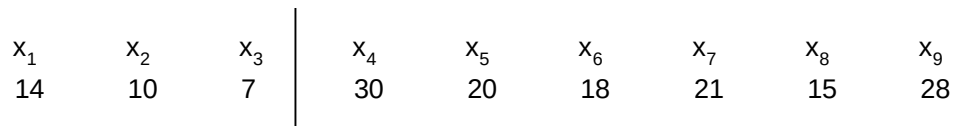
Ahora que los contadores han quedado fijos, intercambiamos x_i con x_j , luego incrementamos i en uno y disminuimos j en uno:

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	
14	10	30	7	20	18	21	15	28	pivote = 15
	$i=2$	$i=3$				$J=7$			$j=6$

Ahora volvemos a repetir el procedimiento: incrementar i mientras x_i sea menor al pivote, disminuir j mientras x_j sea mayor al pivote, intercambiar variables, incrementar i y disminuir j :

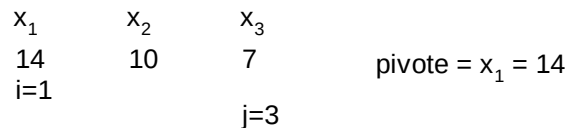


Como vemos en este caso el contador i queda con un valor mayor que el contador j . Cuando esto sucede el proceso concluye y el vector queda dividido en dos: el vector izquierdo que va desde el primer elemento (en nuestro caso 1) hasta j (3) y el derecho que va desde i (4) hasta el último elemento (en nuestro caso 9):

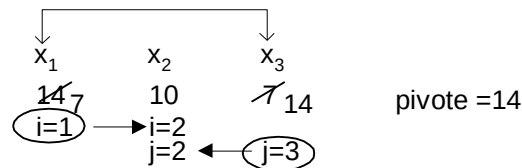


De esta manera se consigue dividir el vector en dos: el izquierdo con los elementos menores al pivote y el derecho con los elementos mayores o iguales al pivote.

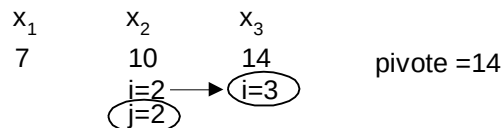
Ahora se vuelve a repetir el mismo procedimiento con los dos vectores resultantes. Apliquemos primero el procedimiento al vector izquierdo:



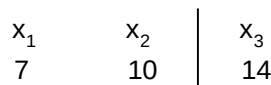
Como se ve ahora los contadores comienzan en 1 y 3 respectivamente y el pivote es el número 14. Aplicando el procedimiento una vez resulta:

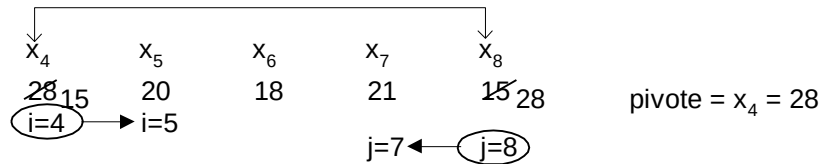


Y repitiendo el procedimiento una vez más se tiene:

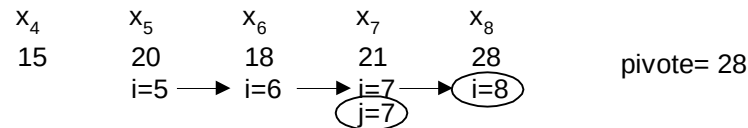


Ahora que los contadores han quedado fijos deberíamos intercambiar los valores de x_i y x_j , pero como el contador i es mayor al contador j , no se realiza el intercambio y el vector queda dividido en dos: el izquierdo (hasta $j=2$) con los elementos menores al pivote y el derecho (desde $i=3$) con los elementos mayores o iguales al pivote:

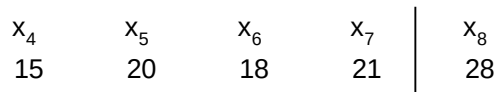




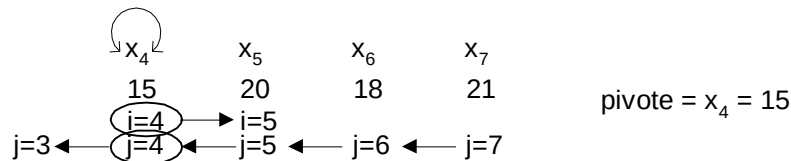
Puesto que i sigue siendo menor a j volvemos a aplicar el procedimiento:



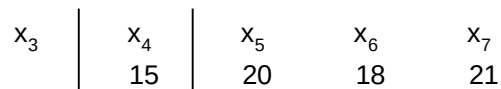
Una vez más no se realiza ningún intercambio pues el contador i es mayor al contador j . Por lo tanto el proceso concluye y el vector queda dividido en dos:



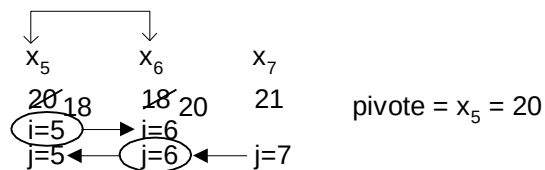
Como el vector derecho tiene un solo elemento (x_8), está ya ordenado. El vector izquierdo sin embargo tiene más de un elemento, por lo que debemos volver a aplicar el procedimiento al mismo:



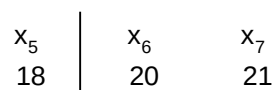
Ahora i es mayor a j , por lo que el vector queda dividido en 2: el vector izquierdo que no tiene elementos y el vector derecho que tiene tres elementos. Al medio queda un elemento que como sabemos debe estar ya con el valor correcto:



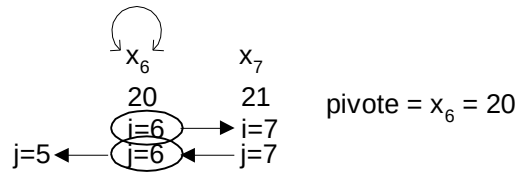
Como el vector derecho tiene más de un elemento, aplicamos el procedimiento al mismo:



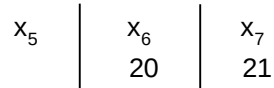
Entonces el vector queda dividido en 2:



Como el vector izquierdo tiene un solo elemento está ya ordenado. Pero como el vector derecho tiene aún 2 elementos debemos aplicar el procedimiento al mismo:



Una vez más el vector queda dividido en 3:



Puesto que el vector izquierdo no tiene elementos, el derecho sólo uno y al medio queda un elemento, todos los vectores tienen un solo elemento y en consecuencia están ya ordenados y como no quedan vectores con más de un elemento, todo el vector está ahora ordenado:

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
7	10	14	15	18	20	21	28	30

Seguramente al estudiar este ejemplo se preguntarán si no ha existido un error al presentar este método como el más eficiente de todos los estudiados, porque desde el punto de vista del esfuerzo humano parecería ser todo lo contrario, sin embargo, en todos los casos se repite siempre el mismo procedimiento mecánico, algo en lo que los humanos no somos buenos, pero que las computadora realizan con bastante eficiencia. Por el contrario, algo que si consume tiempo de computación, y que en consecuencia es conveniente reducir, son los intercambios de variables, y en ese sentido el método Quick - Sort reduce considerablemente el número de intercambios. Aún en un vector tan pequeño como el del ejemplo, con Quick Sort se requieren 7 intercambios, comparados con los 19 que se requieren para el método de *Burbuja*. A medida que incrementa el número de elementos la diferencia se incrementa y cuando el vector es muy grande (con millones o cientos de millones de elementos) lo que a *Burbuja* le toma días a Quick Sort le toma minutos.

19.4.1 Algoritmo y código empleando vectores dinámicos

Este método es por naturaleza recursivo: el método debe llamarse a si mismo primero con todo el vector, luego, cuando el vector queda dividido en dos, con el vector izquierdo y posteriormente el derecho hasta que los vectores quedan con un solo elemento. El algoritmo del módulo principal es:

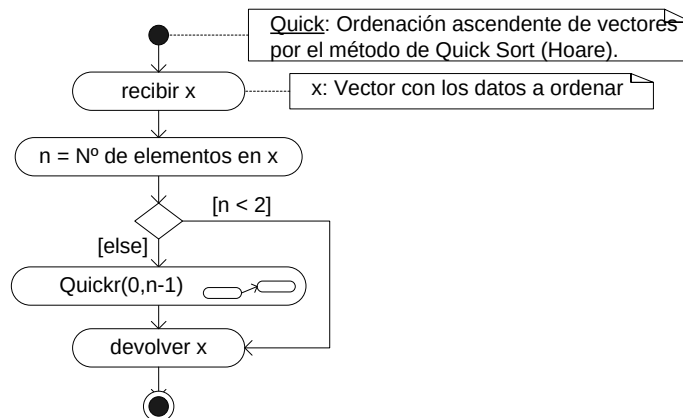


Figura 19.3. Algoritmo del módulo principal del método Quick Sort

Y el algoritmo del módulo recursivo, que es donde en realidad se resuelve el problema es el siguiente:

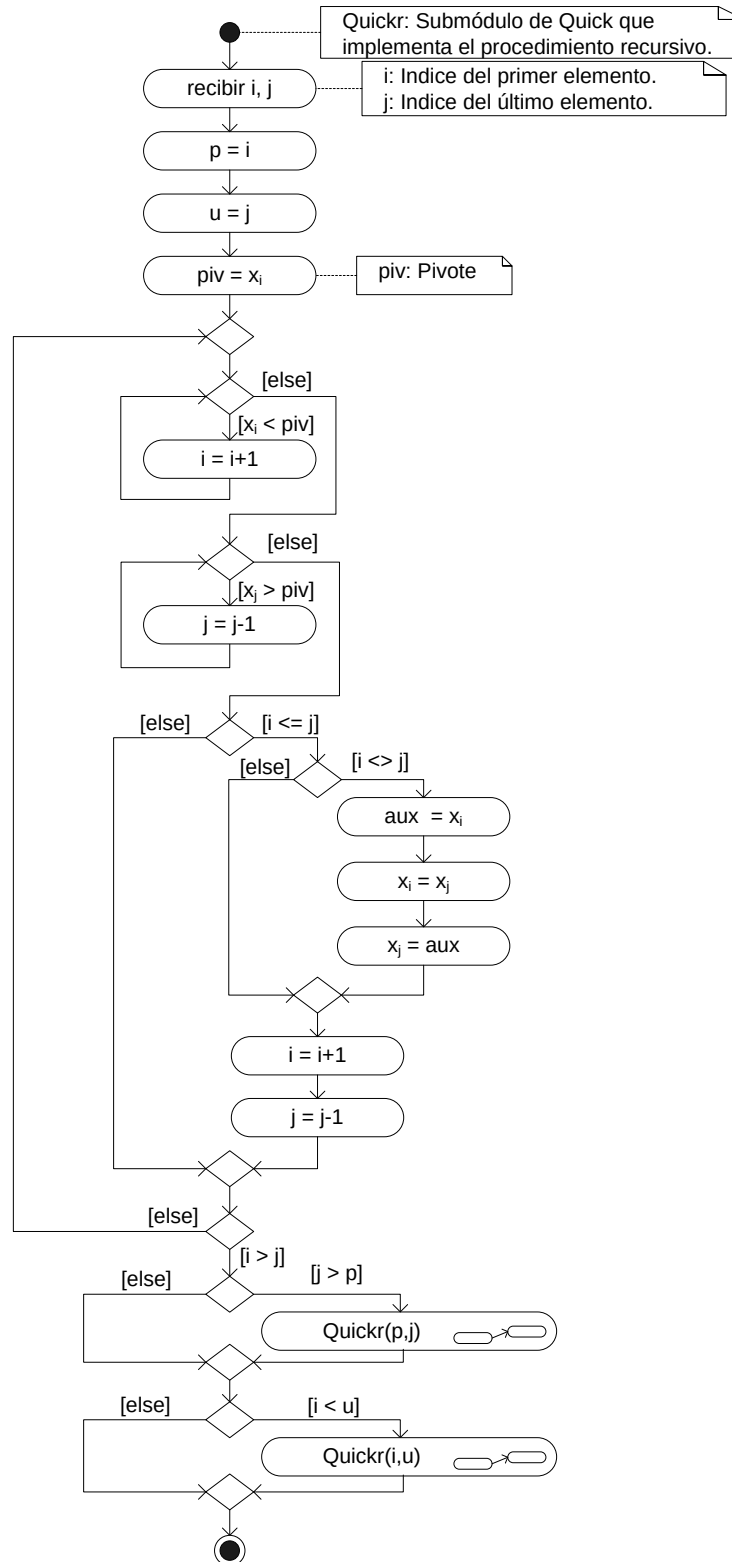


Figura 19.4. Algoritmo del submódulo recursivo del método Quick-Sort.

En este módulo, una vez que el vector queda dividido en dos (aplicando el método), se verifica si existe más de un elemento en el vector izquierdo (lo que ocurre cuando "j" es mayor al primer índice) y de ser así se llama a sí mismo para ordenar ese vector. Cuando no existen más elementos en el vector izquierdo se verifica si existe más de un elemento en el vector derecho (lo que ocurre cuando "i" es menor al último índice) y de ser así se llama a sí mismo para ordenar ese vector. Este proceso recursivo se repite hasta que ya no quedan vectores con más de un elemento ni a la izquierda ni a la derecha de la división.

El código del módulo respectivo es el siguiente:

```

procedure Quick(var x: array of Integer);
var n,piv,aux: integer;
procedure Quickr(i,j: integer);
var p,u: integer;
begin
  p:=i;
  u:=j;
  piv:=x[i];
  repeat
    while x[i]<piv do Inc(i);
    while x[j]>piv do Dec(j);
    if i<=j then begin
      if i<>j then begin
        aux:= x[i];
        x[i]:= x[j];
        x[j]:= aux;
      end;
      inc(i);
      dec(j);
    end;
  until i>j;
  if j>p then Quickr(p,j);
  if i<u then Quickr(i,u);
end;
begin
  n:= Length(x);
  if n<2 then exit;
  Quickr(0,n-1);
end;

```

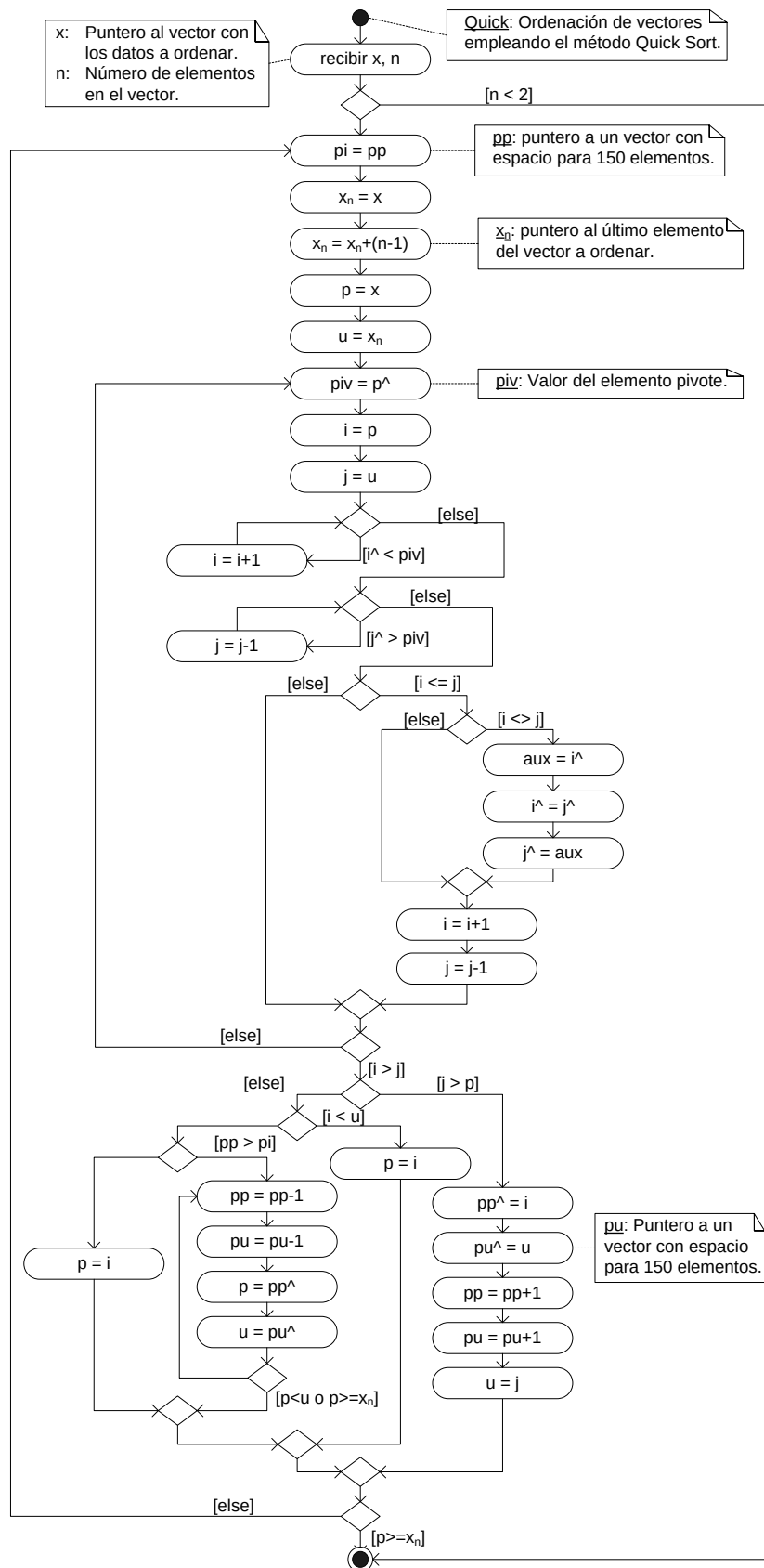
Una vez más, la aplicación es esencialmente la misma que para el método de burbuja, razón por la cual no se presentan los otros módulos de la misma.

19.4.2 Algoritmo y código empleando punteros

Como vimos antes, este método es por naturaleza recursivo, por lo tanto la manera más sencilla de implementar el método es aprovechando la propiedad recursiva, no obstante y al igual que todo procedimiento iterativo puede ser programado en forma recursiva, todo proceso recursivo puede ser programado también de forma iterativa y eso es lo que haremos en este caso.

Antes debemos recordar que en Delphi sólo se puede preguntar si dos punteros son iguales o diferentes, por lo tanto en todas las operaciones relacionales de desigualdad o igualdad, se debe emplear moldeado de tipos (con el tipo Integer) para convertir las direcciones de memoria en números enteros y poder compararlas entre sí.

El algoritmo no recursivo del método se presenta en la siguiente página.



Como se puede ver, el método en sí se encuentra dentro del ciclo *repetir* interno (el cual concluye cuando $i > j$). Este ciclo, al constituir el método, no cambia sin importar que la implementación sea o no recursiva. En el algoritmo las sentencias antes y después del ciclo simplemente logran que el método se repita con la lista (o sublista) adecuada.

El puntero al primer elemento de la lista es "p" y el puntero al último elemento es "u", los punteros "i" y "j" se emplean (al igual que en la explicación) para dividir la lista en dos. Inicialmente "p" apunta al primer elemento de toda la lista ("x") y "u" al último elemento de toda la lista ("x_n").

Cuando la lista queda dividida en dos (al aplicar el método), se verifica si existen elementos en la lista izquierda (preguntando si $j > p$) y de ser así se almacenan los punteros del primer y último elemento de la lista derecha (para poder volver a la misma al terminar con la lista izquierda) y se aplica el método a la lista izquierda (haciendo que u sea igual a j).

Para almacenar los punteros de la lista derecha se emplean dos vectores de punteros (punteros a punteros): "pp" que almacena punteros a los primeros elementos y "pu" que almacena punteros a los últimos elementos.

Cuando ya no existen elementos en la lista izquierda (j no es mayor a p), entonces se verifica si existen elementos en la lista derecha (preguntando si $i < u$) y de ser así se repite el método con la lista derecha (haciendo que p sea igual a i).

Finalmente si no existen elementos ni en la lista izquierda ni en la lista derecha, se verifica si existen elementos a la derecha de la lista actual (preguntando si $pp > pi$, siendo pi un puntero al primer elemento de la lista de punteros pp), de ser así se recuperan los punteros de la lista derecha (en p y u) hasta encontrar una lista que tenga más de un elemento ($p < u$) o hasta que ya no existan más listas a la derecha ($p \geq x_n$).

Todo el proceso se repite hasta que no existan más elementos en la lista (lo que se verifica preguntando si $p \geq x_n$).

Al implementar el código es necesario reservar memoria para los vectores a punteros (punteros a punteros) "pp" y "pu", el problema es que no se sabe de antemano cuantos elementos tendrán estos vectores. Como se está trabajando con punteros es posible ir incrementando la memoria según sea necesario (empleando *ReAllocMem*), sin embargo en caso de que no exista memoria suficiente, ello implicaría un trabajo extra para el procesador (pues se tendría que reservar memoria en otro lugar y copiar los valores almacenados en la nueva dirección). A fin de evitar este trabajo extra se ha optado por reservar memoria para un número fijo de elementos (150) esto tomando en cuenta que el número de veces que una lista se subdivide a sí misma es relativamente pequeño. Por ejemplo para ordenar 10 millones de elementos generalmente las listas se dividen a sí mismas menos de 100 veces).

El código del módulo es el siguiente:

```
procedure Quick(x:pEnteros; n:cardinal);
var aux,piv:Integer; i,j,p,u,xn:pEnteros; pi,pp,pu: ^pEnteros;
begin
  if n<2 then exit;
  GetMem(pp,150*SizeOf(Integer));
  GetMem(pu,150*SizeOf(Integer));
  pi:=pp;
  xn:=x;
  inc(xn,n-1);
  p:=x;
```

```

u:=xn;
repeat
  piv:=p^;
  i:=p;
  j:=u;
  repeat
    while i<piv do inc(i);
    while j>piv do dec(j);
    if Integer(i)<=Integer(j) then begin
      if i<>j then begin
        aux:= i^;
        i^:= j^;
        j^:= aux;
      end;
      inc(i);
      dec(j);
    end;
  until Integer(i)>Integer(j);
  if Integer(j)>Integer(p) then begin
    pp^:=i;
    pu^:=u;
    inc(pp);
    inc(pu);
    u:=j; end
  else
    if Integer(i)<Integer(u) then p:=i
    else
      if Integer(pp)>Integer(pi) then
        repeat
          dec(pp);
          dec(pu);
          p:= pp^;
          u:= pu^;
        until (Integer(p)<Integer(u)) or (Integer(p)>=Integer(xn))
      else
        p:= i;
    until Integer(p)>=Integer(xn);
    FreeMem(pp);
    FreeMem(pu);
  end;
end;

```

Una vez más, la aplicación es esencialmente la misma que para el método de burbuja, razón por la cual no se presentan los otros módulos de la misma.

19.5 Ejercicios

1. Elabore una aplicación para ordenar vectores de números reales por el método Shell, trabajando con vectores dinámicos (y/o parámetros abiertos) y empleando la recursividad.
2. Elabore una aplicación para ordenar vectores de números reales por el método Quick Sort, trabajando con punteros y la recursividad.

20. BÚSQUEDA E INTERCALACIÓN

Como otra aplicación de los vectores estudiaremos en este tema dos métodos de búsqueda y un método de intercalación. La búsqueda se emplea para ubicar un determinado valor dentro de una lista. Cuando se trabaja con vectores, ubicar un valor implica determinar la posición (índice o dirección de memoria) del elemento que contiene ese valor.

De los diferentes métodos de búsqueda que existen estudiaremos los métodos de *búsqueda secuencial* y *búsqueda binaria*.

20.1 MÉTODO DE BÚSQUEDA SECUENCIAL

Este método, conocido también como búsqueda lineal, se emplea cuando los elementos del vector no están ordenados. Al estar los datos desordenados, no se puede aplicar ningún algoritmo para acelerar la búsqueda, por lo que la búsqueda secuencial lo único que se hace es recorrer el vector, elemento por elemento, hasta ubicar el valor buscado o hasta que ya no existen más elementos en la lista.

Como en una lista pueden existir valores repetidos, una vez ubicado un elemento, se puede continuar la búsqueda una posición después del elemento ubicado.

A pesar de su simplicidad, el método de búsqueda secuencial es un método de utilidad práctica, porque mucha de la información existente, como textos y documentos, se encuentra desordenada, no existiendo otra alternativa para la búsqueda que la secuencial.

20.1.1 Algoritmo y código empleando vectores dinámicos

El algoritmo del método considerando el uso de parámetros abiertos o vectores dinámicos se presenta en la siguiente figura (Figura 20.1). En dicho algoritmo si el valor buscado no se encuentra en la lista el resultado es -1, es decir una posición imposible (porque como se sabe el menor índice en un vector dinámico o parámetro abierto es cero).

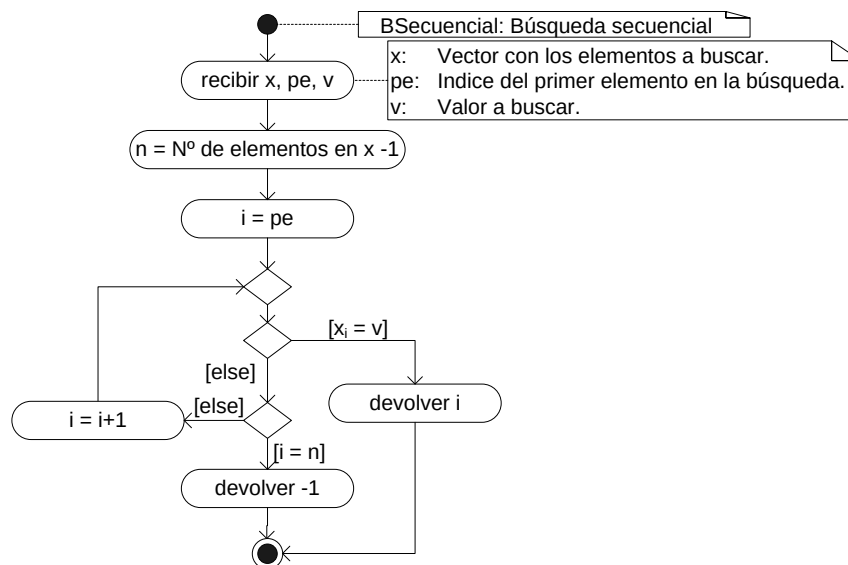


Figura 20.1. Método de búsqueda secuencial (Vectores dinámicos)

Para probar este método se elaborará una aplicación de prueba con un StringGrid, dos BitBtns, un Memo y un Label. El código del método, así como el de los módulos para leer un número entero del memo y generar un vector de números enteros entre dos límites dados, es el siguiente:

```
unit uSecuencialD;
```

```
interface
```

```
    uses SysUtils, StdCtrls, QDialogs, uQuickSortD;
```

```
    function bSecuencial(x:array of integer; pe:cardinal; v:integer):integer;
```

```
    function LeerEntero(m:TMemo):integer;
```

```
    function GenerarVEnteros(n:cardinal; li,ls:integer):tvEnteros;
```

```
implementation
```

```
    function bSecuencial(x:array of integer; pe:cardinal; v:integer):integer;
```

```
    var i:integer; n:cardinal;
```

```
    begin
```

```
        n:=High(x);
```

```
        for i:=pe to n do
```

```
            if x[i]=v then begin
```

```
                result:=i;
```

```
                exit;
```

```
            end;
```

```
        result:=-1;
```

```
    end;
```

```
    function LeerEntero(m:TMemo):integer;
```

```
    begin
```

```
        try
```

```
            result:=StrToInt(m.Text);
```

```
        except
```

```
            on EConvertError do begin
```

```
                result:=0;
```

```
                ShowMessage('El número ha buscar está mal escrito');
```

```
            end;
```

```
        end;
```

```
    end;
```

```
    function GenerarVEnteros(n:cardinal; li,ls:integer):tvEnteros;
```

```
    var x:tvEnteros; d:integer; i:cardinal;
```

```
    begin
```

```
        SetLength(x,n);
```

```
        d:=ls-li+1;
```

```
        for i:=0 to n-1 do
```

```
            x[i]:=Random(d)+1;
```

```
        result:=x;
```

```
    end;
```

```
end.
```

Como se puede ver, en esta unidad se ha incluido la unidad uQuickSortD, y para ello se ha copiado el archivo "uQuickSortD.dcu" de la carpeta donde se ha elaborado la aplicación para el método de Quick Sort (con vectores dinámicos) a la carpeta donde se ha guardado la presente aplicación.

En ejecución la aplicación tiene la apariencia que se muestra en la siguiente figura.

Nº	Valor
3121	425
3122	336
3123	725
3124	434
3125	33
3126	673
3127	205
3128	405
3129	618
3130	690
3131	892
3132	573
3133	707
3134	314
3135	640
3136	45
3137	794

Generar

Número a buscar: 45

Buscar

Para lograr esta apariencia se han modificado algunas de las propiedades de los componentes en el evento "onCreate" de la forma:

```
uses uSecuencialD,uQuickSortD;
```

```
{ $R *.dfm }
```

```
procedure TForm1.FormCreate(Sender: TObject);
```

```
begin
```

```
    Form1.Position:=poScreenCenter;
```

```
    Form1.Caption:='Búsqueda Secuencial (Dinámicos)';
```

```
    Form1.BorderStyle:=bsDialog;
```

```
    Form1.Brush.Bitmap:=tBitmap.Create;
```

```
    Form1.Brush.Bitmap.LoadFromFile('C:\Archivos de programa\'+  
        'Archivos Comunes\Borland Shared\Images\BackGrnd\Writing.bmp');
```

```
    StringGrid1.Align:=alLeft;
```

```
    StringGrid1.ColCount:=2;
```

```
    StringGrid1.ColWidths[0]:=80;
```

```
    StringGrid1.ColWidths[1]:=100;
```

```
    StringGrid1.Width:=StringGrid1.ColWidths[0]+StringGrid1.ColWidths[1]+25;
```

```
    StringGrid1.Cells[0,0]:=Format('%12s', ['Nº']);
```

```
    StringGrid1.Cells[1,0]:=Format('%15s', ['Valor']);
```

```
    StringGrid1.Options:=StringGrid1.Options+[goThumbTracking];
```

```
    BitBtn1.Width:=100;
```

```
    BitBtn1.Left:=StringGrid1.Width+20;
```

```
    BitBtn1.Top:=50;
```

```

BitBtn1.Glyph.LoadFromFile('C:\Archivos de programa\Archivos Comunes\' +
  'Borland Shared\Images\Buttons\DocStack.bmp');
BitBtn1.NumGlyphs:=2;
BitBtn1.Caption:='&Generar';
BitBtn1.Cursor:=crHandPoint;
BitBtn2.Width:=100;
BitBtn2.Left:=BitBtn1.Left;
BitBtn2.Top:=180;
BitBtn2.Glyph.LoadFromFile('C:\Archivos de programa\Archivos Comunes\' +
  'Borland Shared\Images\Buttons\ZoomIn.bmp');
BitBtn2.NumGlyphs:=2;
BitBtn2.Caption:='&Buscar';
BitBtn2.Cursor:=crHandPoint;
BitBtn2.Default:=True;
Mem1.Width:=100;
Mem1.Height:=21;
Mem1.Left:=BitBtn1.Left;
Mem1.Top:=120;
Mem1.WantReturns:=False;
Mem1.Alignment:=taRightJustify;
Mem1.Text:='';
Label1.Left:=Mem1.Left;
Label1.Top:=Mem1.Top-15;
Label1.Transparent:=True;
Label1.Caption:='Número a buscar: ';
Form1.Width:=BitBtn1.Left+BitBtn1.Width+20;
end;

```

Para liberar el Bitmap de la forma se ha programado también el evento "onClose" de la misma:

```

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  Form1.Brush.Bitmap.Free;
end;

```

Se ha validado la introducción de datos al Mem1, de manera que sólo permita escribir números enteros, programando su evento "onKeyPress":

```

procedure TForm1.Mem1KeyPress(Sender: TObject; var Key: Char);
begin
  If not (Key in [#8,'0'..'9']) then Abort;
end;

```

Se ha programado el evento "onClick" del BitBtn1 de manera que genere un vector con 10000 números enteros comprendidos entre 1 y 1000 (para que existan números repetidos), mostrándolos en el StringGrid1:

```

procedure TForm1.BitBtn1Click(Sender: TObject);
var x:tvEnteros;
begin
  x:=GenerarVEnteros(10000,1,1000);
  MostrarVector(x,StringGrid1);
  Mem1.SetFocus;
  StringGrid1.Tag:=1;
  Finalize(x);
end;

```

Como se puede observar, en este módulo se ha asignado el número "1" a la propiedad "Tag" del StringGrid. La propiedad "Tag" es una propiedad de tipo entero (común a todos los componentes visibles) creada para el uso de los

programadores. En esta aplicación se emplea para guardar el estado del StringGrid: Si ha sido actualizado, esta propiedad es colocada en "1", caso contrario mantiene su valor por defecto: "0".

Finalmente se ha programado el evento "onClick" del BitBtn2 de manera que lea el número escrito en el Memo1, lea el vector del StringGrid1 y busque el número en el vector (empleando el método de búsqueda Secuencial). Si el número buscado no existe se muestra un mensaje y se selecciona la primera fila del StringGrid, caso contrario se muestra el elemento encontrado y se incrementa el índice "pe" para que la siguiente búsqueda continúe después del elemento encontrado. Antes de realizar la búsqueda se verifica si el StringGrid1 ha sido modificado (es decir si su propiedad "Tag" está en "1") y de ser así se lee su contenido, se reinicializa el índice "pe" en cero y se coloca la propiedad "Tag" del StringGrid en "0" (no modificado):

```

procedure TForm1.BitBtn2Click(Sender: TObject);
var v:integer;
const x:tvEnteros=nil; pe:Integer=0;
begin
  if StringGrid1.Tag=1 then begin
    StringGrid1.Tag:=0;
    Finalize(x);
    pe:=0;
    StringGrid1.Row:=0;
    x:=LeerVector(StringGrid1);
  end;
  v:=LeerEntero(Memo1);
  pe:=bSecuencial(x,pe,v);
  if pe<0 then begin
    ShowMessage('No existe el número: '+IntToStr(v)+' en el vector');
    StringGrid1.Row:=1; end
  else begin
    inc(pe);
    StringGrid1.Row:=pe;
  end;
end;

```

20.1.2 Algoritmo y código empleando punteros

El algoritmo del método secuencial, trabajando con punteros es el que se muestra en la siguiente figura:

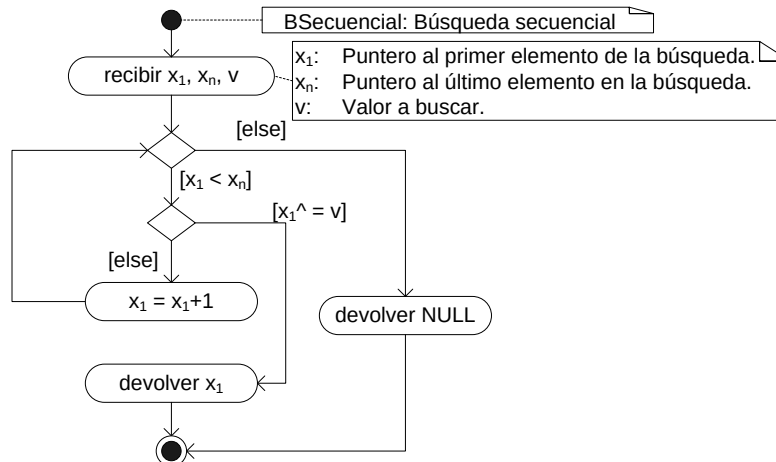


Figura 20.2. Método de búsqueda secuencial (Punteros)

La aplicación de prueba para este método es muy similar a la del anterior acápite y el código del método así como el de los módulos para leer un número entero del memo y generar un vector de números enteros entre dos límites dados, empleando punteros, es el siguiente:

```
unit uSecuencialP;
```

```
interface
```

```
    uses SysUtils, StdCtrls, QDialogs, uQuickSortP;
```

```
    function bSecuencial(x1,xn:pEnteros; v:integer):pEnteros;
```

```
    function LeerEntero(m:TMemo):integer;
```

```
    function GenerarVEnteros(n:cardinal; li,ls:integer):pEnteros;
```

```
implementation
```

```
    function bSecuencial(x1,xn:pEnteros; v:integer):pEnteros;
```

```
    var i:integer; n:cardinal;
```

```
    begin
```

```
        while Integer(x1)<Integer(xn) do begin
```

```
            if x1^=v then begin
```

```
                result:=x1;
```

```
                exit;
```

```
            end;
```

```
            inc(x1);
```

```
        end;
```

```
        result:=nil;
```

```
    end;
```

```
    function LeerEntero(m:TMemo):integer;
```

```
    begin
```

```
        try
```

```
            result:=StrToInt(m.Text);
```

```
        except
```

```
            on EConvertError do begin
```

```
                result:=0;
```

```
                ShowMessage('El número ha buscar está mal escrito');
```

```
            end;
```

```
        end;
```

```
    end;
```

```
    function GenerarVEnteros(n:cardinal; li,ls:integer):pEnteros;
```

```
    var x,p:pEnteros; d:integer; i:cardinal;
```

```
    begin
```

```
        GetMem(x,n*SizeOf(x^));
```

```
        p:=x;
```

```
        d:=ls-li+1;
```

```
        for i:=1 to n do begin
```

```
            p^:=Random(d)+1;
```

```
            inc(p);
```

```
        end;
```

```
        result:=x;
```

```
    end;
```

```
end.
```

El evento "onCreate" de la aplicación sólo cambia en el título (caption) de la forma, por lo que no se mostrará el mismo. El evento "onClose" es el

mismo y aunque el evento "onKeyPress" también puede ser el mismo se ha cambiado empleando la estructura "case" en lugar de "if":

```
procedure TForm1.Mem1KeyPress(Sender: TObject; var Key: Char);
begin
  case Key of
    #8, '0'..'9':;
    else Abort;
  end;
end;
```

Los eventos que sí cambian son los "onClick" de los BitBtns:

```
procedure TForm1.BitBtn1Click(Sender: TObject);
var x:pEnteros; const n=10000;
begin
  x:=GenerarVEnteros(n,1,1000);
  MostrarVector(x,n,StringGrid1);
  StringGrid1.Tag:=1;
  Mem1.SetFocus;
  FreeMem(x);
end;

procedure TForm1.BitBtn2Click(Sender: TObject);
var v:integer; n:cardinal;
const pe:pEnteros=nil; x:pEnteros=nil; xn:pEnteros=nil;
begin
  if pe=nil then pe:=x;
  if StringGrid1.Tag=1 then begin
    FreeMem(x);
    LeerVector(StringGrid1,x,n);
    xn:=x;
    inc(xn,n-1);
    StringGrid1.Tag:=0;
    pe:=x;
  end;
  v:=LeerEntero(Mem1);
  pe:=bSecuencial(pe,xn,v);
  if pe=nil then begin
    ShowMessage('No existe el número: '+IntToStr(v)+' en el vector');
    StringGrid1.Row:=1;
    pe:=x end
  else begin
    n:=(Integer(pe)-Integer(x)) div SizeOf(x^);
    StringGrid1.Row:=n+1;
    inc(pe);
  end;
end;
```

20.1.3 Algoritmo y código con Strings y vectores dinámicos

Aunque el método de búsqueda secuencial (al igual que los otros métodos) no cambia con el tipo de dato que se busque, cuando se trabaja con cadenas, frecuentemente sólo se quiere buscar una parte de la misma. Por ejemplo cuando se busca un nombre con frecuencia sólo se escriben las primeras letras del mismo (y no el nombre completo). En esos casos el método debe ser capaz de encontrar el primer nombre cuyas primeras letras coincida con las letras escritas. Entonces el algoritmo cambia en el sentido de que el valor buscado se compara sólo con las primeras letras de los elementos y no con el

valor en completo (como sucede con los números). El algoritmo que toma en cuenta dicha modificación es el siguiente:

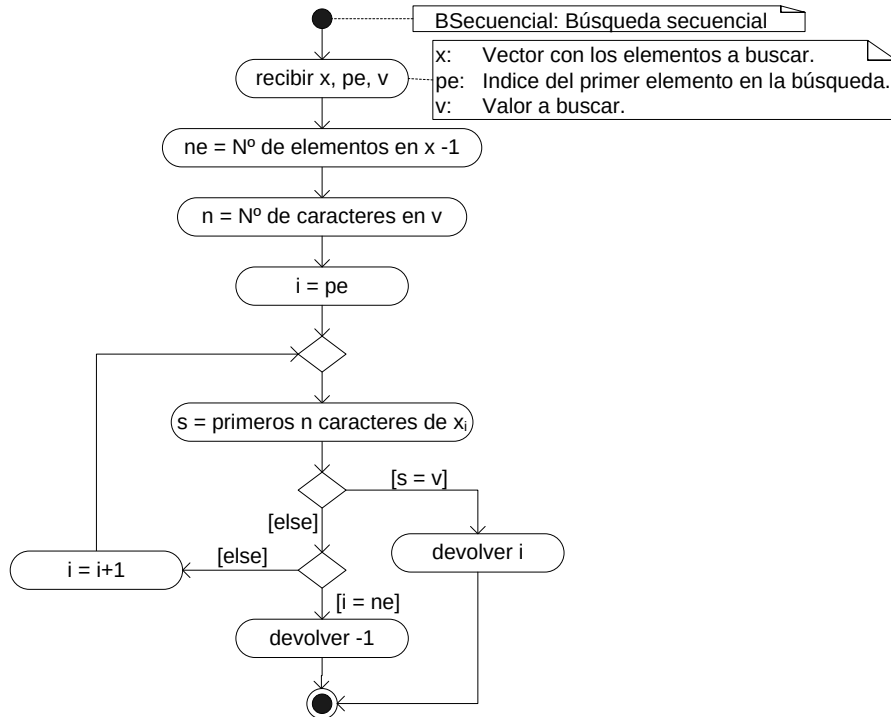


Figura 20.3. Método de búsqueda secuencial (Cadenas)

En cuanto al código, el principal inconveniente que surge cuando se trabaja con cadenas es el generar los valores de prueba, pues no existen funciones que generen nombres u otro tipo de cadenas. En consecuencia debemos crear dichas funciones.

En este ejemplo generaremos un vector con nombres de personas conformados por dos apellidos y uno o dos nombres. Para ello crearemos dos funciones que devuelvan al azar un nombre, femenino o masculino, de una lista de nombres, otra que devuelva un apellido de una lista de apellidos y otra que devuelva el nombre completo empleando las otras funciones. El código de dichas funciones (escrito en la unidad uNombres) es el siguiente:

```
unit uNombres;
```

```
interface
```

```
function GenerarNombreF:string;
function GenerarNombreM:string;
function GenerarApellido:string;
function GenerarNombreCompleto:string;
```

```
implementation
```

```
function GenerarNombreF:string;
const nombre:array[0..75] of string = ('ANA','AMALIA','ANDREA','ADELA',
'ARIANA','ANABEL','ADRIANA','BETTY','BELINDA','CARMEN','CARLA','CECILIA',
'CINTIA','DANITZA','DIANA','DORA','DANIELA','ELENA','ELMI','ESTER','ELY',
'EDMI','FABIOLA','FERNANDA','CABRIELA','GLADYS','GEORGINA','GLORIA',
'HILDA','ILSEN','ISABEL','JANETH','JANINA','JULIA','JUDITH','KARINA',
'LIDIA','LILIANA','LIZETH','LUPE','LUZ','LOURDES','MARIA','MARIANA',
```



```

'MARITZA', 'MARCELA', 'MIRIAN', 'MAGDALENA', 'MARTA', 'NOEMI', 'NANET', 'NOELIA',
'OLIMPIA', 'OFELIA', 'PATRICIA', 'PAULINA', 'PAMELA', 'ROMINA', 'ROSSEMARY',
'RUTH', 'ROSA', 'SANDRA', 'SUSANA', 'SELENA', 'SILVINA', 'SCARLET', 'TERESA',
'TALIA', 'VANIA', 'VERONICA', 'VIVIANA', 'VIRGINIA', 'XIMENA', 'YARA', 'YANESI',
'ZULEMA');

begin
    result:=nombre[random(76)];
end;

function GenerarNombreM:string;
const nombre:array[0..71] of string = ('ALEJANDRO', 'ALFREDO', 'ALVARO',
'ANTONIO', 'ANGEL', 'ALCIDES', 'ALBERTO', 'BENIGNO', 'BETO', 'BORIS', 'CARLOS',
'CESAR', 'CECILIO', 'DAVID', 'DANIEL', 'ENRIQUE', 'ERNESTO', 'EDWIN', 'EDSON',
'ELIAS', 'FERNANDO', 'FABRICIO', 'FREDDY', 'FELIX', 'FELICIANO', 'GASTON',
'GILMAR', 'GONZALO', 'GERSON', 'GUIDO', 'HENRY', 'HERNAN', 'HORACIO', 'IGNACIO',
'ISAAC', 'IVES', 'JAVIER', 'JHON', 'JOSE', 'JUAN', 'JORGE', 'JAIME', 'LUIS',
'LIMBERTH', 'MARCELO', 'MICHEL', 'MAURICIO', 'MAX', 'NELSON', 'NESTOR',
'ORLANDO', 'PEDRO', 'PABLO', 'PAUL', 'RICHARD', 'ROBERTO', 'RICARDO', 'RAFAEL',
'RENE', 'SANDRO', 'SANTOS', 'TITO', 'ULISES', 'VICENTE', 'VICTOR', 'VLADIMIR',
'WILSON', 'WILLY', 'WILFREDO', 'YVANOX', 'YAMIL', 'ZENON');

begin
    result:=nombre[random(72)];
end;

function GenerarApellido:string;
const apellido:array[0..175] of string = ('AGUILAR', 'ALBINO', 'ALVAREZ',
'ARANCIBIA', 'AVILA', 'ANDRADE', 'ARANDIA', 'ARAUZ', 'ARDUZ', 'ARIAS', 'BARRON',
'BALLIVIAN', 'BARRIENTOS', 'BEJARANO', 'BEDOYA', 'BELLIDO', 'BELTRAN',
'BENAVIDEZ', 'BORDA', 'BOBARIN', 'BORJA', 'CABA', 'CAMPOS', 'CARDOZO', 'CARMONA',
'CARREON', 'CARRASCO', 'CARRILLO', 'CARVALLO', 'CASTAÑOS', 'CHOQUERIVE',
'CEPEDA', 'CONDORI', 'COCA', 'CORTEZ', 'CRESPO', 'CRUZ', 'DALENCE',
'DAVALOS', 'DAVEZIES', 'DAVILA', 'DAZA', 'DELGADILLO', 'DULON', 'DURAN',
'ECHALAR', 'ECHEVERRIA', 'EID', 'ESCALANTE', 'ENRIQUEZ', 'ESPADA', 'FERNANDEZ',
'FLORES', 'FONSECA', 'FRIAS', 'FUERTES', 'GALLARDO', 'GALVAN', 'GARCIA', 'GOMEZ',
'GORENA', 'GRIMALDOS', 'HERRERA', 'HINOJOSA', 'HURTADO', 'IBÁÑEZ', 'IBARRA',
'ILLANES', 'INCHAUSTI', 'JALDIN', 'JARA', 'JIMENEZ', 'KAWANO', 'KOMAREK',
'LAGRAVA', 'LAGUNA', 'LAIME', 'LARA', 'LAZCANO', 'LEAÑO', 'LEYTON', 'LLANOS',
'LLAVE', 'LLOBET', 'MAMANI', 'MARTINEZ', 'MEDINA', 'MENDIENTA', 'MEZA', 'MOLINA',
'MUÑOZ', 'NAVA', 'NOGALES', 'NOYA', 'NAVARRO', 'ORGAZ', 'ORIAS', 'OROZCO',
'ORTIZ', 'ORTUSTE', 'OTONDO', 'OVANDO', 'PACHECO', 'PALACIOS', 'PADILLA',
'PEMINTEL', 'PERALTA', 'PEREIRA', 'PEREZ', 'PLANTARROSA', 'POPPE', 'PRADEL',
'QUEVEDO', 'QUINTEROS', 'QUIÑONES', 'QUIROGA', 'QUIROZ', 'QUENTA',
'RADA', 'RAMIREZ', 'RENDON', 'REYNAGA', 'REYNOLDS', 'RENTERIA', 'RIOS', 'RIVERA',
'ROCAVADO', 'ROCHA', 'ROJAS', 'ROMERO', 'RUIZ', 'SAAVEDRA', 'SALAMANCA',
'SALAS', 'SALAZAR', 'SANCHEZ', 'SANDI', 'SANTOS', 'SERRANO', 'STUMVOLL',
'TARDIO', 'TABOADA', 'TERAN', 'TIRADO', 'TOLEDO', 'TORRES', 'TORRICO',
'TORREJON', 'UGRINOVIC', 'URIONA', 'URQUIDI', 'URQUIZU', 'URRIOLAGOITIA',
'URIETA', 'VACA', 'VACAGUZMAN', 'VALDEZ', 'VALENCIA', 'VARGAS', 'VALVERDE',
'VENTURA', 'VELASQUEZ', 'VILLAVICENCIO', 'VILLEGAS', 'WAYAR', 'WILLIAMS',
'YAÑEZ', 'YUGAR', 'ZAMORA', 'ZARATE', 'ZELAYA', 'ZEGADA', 'ZEBALLOS', 'ZULETA',
'ZORRILLA', 'ZURITA');

begin
    result:=apellido[random(176)];
end;

function GenerarNombreCompleto:string;
begin
    result:=GenerarApellido+' '+GenerarApellido+' ';
end;

```

```

if random(100)<50 then begin
  result:=result+GenerarNombreM;
  if random(100)>50 then
    result:=result+' '+GenerarNombreM; end
else begin
  result:=result+GenerarNombreF;
  if random(100)>50 then
    result:=result+' '+GenerarNombreF;
end;
end;

end.

```

Ahora podemos escribir el código del módulo que genera un vector con nombres de personas y el código del método Secuencial:

```

unit uSecuencialD;

interface

  uses SysUtils, StdCtrls, QDialogs, UNombres;

  type tvCadenas=array of string;

  function bSecuencial(x:array of string; pe:cardinal; v:string):integer;
  function GenerarNombres(n:cardinal):tvCadenas;

implementation

  function bSecuencial(x:array of string; pe:cardinal; v:string):integer;
  var i:integer; ne:cardinal; n:byte; s:string;
  begin
    ne:=High(x);
    n:=Length(v);
    for i:=pe to ne do begin
      s:=Copy(x[i],1,n);
      if s=v then begin
        result:=i;
        exit;
      end;
    end;
    result:=-1;
  end;

  function GenerarNombres(n:cardinal):tvCadenas;
  var x:tvCadenas; i:cardinal;
  begin
    SetLength(x,n);
    for i:=0 to n-1 do
      x[i]:=GenerarNombreCompleto;
    result:=x;
  end;

end.

```

Puesto que en el anterior capítulo los programas han sido elaborados para trabajar con números enteros, debemos modificar la unidad "uQuickSortD" para que trabaje con cadenas y aun cuando para este método, no se requiere el módulo de ordenación, el mismo también ha sido modificado a manera de ejemplo y como recurso para futuras aplicaciones:

```
unit uQuickSortD;
```

```
interface
```

```
    uses Grids, StdCtrls, Math, DateUtils, SysUtils, uSecuencialD;
```

```
    procedure MostrarVector(x: array of string; s:TStringGrid);
```

```
    function LeerVector(s:TStringGrid):tvCadenas;
```

```
    procedure Quick(var x: array of string);
```

```
implementation
```

```
    procedure MostrarVector(x: array of string; s:TStringGrid);
```

```
    var i,n:cardinal;
```

```
    begin
```

```
        n:=Length(x);
```

```
        s.RowCount:=n+1;
```

```
        for i:=1 to n do begin
```

```
            s.Cells[0,i]:=Format('%15d',[i]);
```

```
            s.Cells[1,i]:=x[i-1];
```

```
        end;
```

```
    end;
```

```
    function LeerVector(s:TStringGrid):tvCadenas;
```

```
    var i,n:cardinal; x:tvCadenas;
```

```
    begin
```

```
        n:=s.RowCount-1;
```

```
        if n=0 then raise EInvalidArgument.Create(  
            'El StringGrid no tiene datos');
```

```
        SetLength(x,n);
```

```
        for i:=1 to n do
```

```
            x[i-1]:=s.Cells[1,i];
```

```
        result:=x;
```

```
    end;
```

```
    procedure Quick(var x: array of string);
```

```
    var n:integer; piv,aux: string;
```

```
    procedure Quickr(i,j: integer);
```

```
    var p,u: integer;
```

```
    begin
```

```
        p:=i;
```

```
        u:=j;
```

```
        piv:=x[i];
```

```
        repeat
```

```
            while x[i]<piv do Inc(i);
```

```
            while x[j]>piv do Dec(j);
```

```
            if i<=j then begin
```

```
                if i<>j then begin
```

```
                    aux:= x[i];
```

```
                    x[i]:= x[j];
```

```
                    x[j]:= aux;
```

```
                end;
```

```
                inc(i);
```

```
                dec(j);
```

```
            end;
```

```
        until i>j;
```

```
        if j>p then Quickr(p,j);
```

```
        if i<u then Quickr(i,u);
```

```

end;
begin
  n:= Length(x);
  if n<2 then exit;
  Quickr(0,n-1);
end;

```

end.

Para probar este algoritmo se ha elaborado una aplicación muy similar a la de los anteriores ejemplos, sólo que ahora la segunda columna debe ser más ancha (250 puntos) y cambiar su título a "Nombre" (formateado con 40 espacios):

Nº	Nombre
1	AGUILAR ANDRADE DORA
2	RADA GALLARDO GILMAR
3	BEDOYA LLOBET SANTOS
4	FERNANDEZ VELASQUEZ ROBERTO
5	REYNOLDS UGRINOVIC FERNANDA
6	GALVAN LLAVE SANDRO
7	MAMANI CARRASCO EDMY YANESI
8	MEDINA VALDEZ AMALIA
9	CARREON MEZA LUIS
10	SANDI QUIÑONES OFELIA DORA
11	RAMIREZ PADILLA MARTA ELMY
12	RAMIREZ FLORES ROBERTO
13	VACAGUZMAN MOLINA XIMENA GLADYS
14	ALBINO RIOS ROSSEMARY DORA
15	CARREON VENTURA MARITZA OLIMPIA
16	ESPADA BELTRAN EDMY CINTIA
17	ESCALANTE ORGAZ DANITZA

A más de la apariencia se debe modificar el evento "onKeyPress" del Memo1 para que permita escribir sólo espacios y letras en mayúsculas y los eventos "onClick" de los BitBtns para trabajar con un vector de cadenas con 1000 nombres:

```

procedure TForm1.Memo1KeyPress(Sender: TObject; var Key: Char);
begin
  Key:=AnsiUpperCase(Key)[1];
  if not (Key in [#8,'A'..'Z','Ñ',' ']) then begin
    Beep; Abort;
  end;
end;

```

```

procedure TForm1.BitBtn1Click(Sender: TObject);
var x:tvCadenas; const n=1000;
begin
    x:=GenerarNombres(n);
    MostrarVector(x,StringGrid1);
    Mem1.SetFocus;
    StringGrid1.Tag:=1;
    Finalize(x);
end;

procedure TForm1.BitBtn2Click(Sender: TObject);
var v:string;
const x:tvCadenas=nil;
const pe:Integer=0;
begin
    if StringGrid1.Tag=1 then begin
        StringGrid1.Tag:=0;
        Finalize(x);
        pe:=0;
        StringGrid1.Row:=1;
        x:=LeerVector(StringGrid1);
    end;
    v:=Mem1.Text;
    pe:=bSecuencial(x,pe,v);
    if pe<0 then begin
        ShowMessage('No existe el nombre: '+v+' en el vector');
        StringGrid1.Row:=1; end
    else begin
        inc(pe);
        StringGrid1.Row:=pe;
    end;
end;

```

20.2 MÉTODO DE BÚSQUEDA BINARIA

El método de búsqueda binaria se emplea cuando los elementos del vector están ordenados. En este capítulo asumiremos que los datos están ordenados ascendentemente.

En este método se compara el valor buscado con el elemento central del vector. Si el valor buscado es igual al elemento central el proceso concluye, caso contrario se divide el vector en dos y se repite el procedimiento en la mitad donde es posible que se encuentre el valor buscado. Este procedimiento se repite (dividiendo el vector en dos en cada iteración) hasta que el valor es encontrado o hasta que la mitad donde se realiza la búsqueda queda sin elementos.

Se sabe en cuál de las mitades se encuentre el valor buscado comparándolo con el elemento central: Si es mayor al elemento central entonces se encuentra en la mitad derecha (o superior) y si es menor en la mitad izquierda (o inferior).

Cuando en el vector existen dos o más elementos con el valor buscado, el método de búsqueda binaria ubica uno de esos elementos, pero no garantiza que dicho elemento sea el primero. Por consiguiente, una vez ubicado el elemento, es necesario recorrer el vector hacia atrás hasta ubicar el primer elemento con el valor buscado.

Para comprender mejor el procedimiento que se sigue en el método ubiquemos el número 5 en el siguiente vector:

x_1	x_2	x_3	x_4	x_5	x_6	x_7
1	3	5	9	11	21	22

En el proceso emplearemos 3 variables: "i" para el índice del primer elemento que se toma en cuenta en la búsqueda; "j" para el índice del último elemento que se toma en cuenta en la búsqueda y "k" para el índice del elemento central. El valor de "k" se calcula como el cociente de la división de $i+j$ entre 2:

x_1	x_2	x_3	x_4	x_5	x_6	x_7
1	3	5	9	11	21	22
↑			↑			↑
i=1			k = cociente(i+j)/2=4			j=7

Como el elemento central es mayor al valor buscado (5) se repite el procedimiento en la mitad izquierda. El último índice a ser tomado en cuenta (j) es 3 (k-1):

x_1	x_2	x_3
1	3	5
↑	↑	↑
i=1	k = 2	j=3

Ahora el elemento central es menor al valor buscado (5), entonces se repite el procedimiento en la mitad derecha de este vector, por lo tanto nos queda un vector con un solo elemento:

x_3
5
↑
i=3 k=3 j=3

Y como ese elemento es igual al valor buscado el proceso concluye habiéndose ubicado el número 5 en el elemento número 3 (el valor de k).

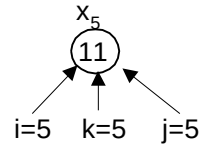
Ahora empleemos el método para ubicar el número 11. Como siempre el proceso comienza tomando en cuenta todo el vector:

x_1	x_2	x_3	x_4	x_5	x_6	x_7
1	3	5	9	11	21	22
↑			↑			↑
i=1			k = cociente(i+j)/2=4			j=7

Puesto que el elemento central es menor al valor buscado (11) se repite el procedimiento en el vector derecho:

x_5	x_6	x_7
11	21	22
↑	↑	↑
i=5	k=6	j=7

Dado que el elemento central es mayor al valor buscado (11) se repite el procedimiento con el vector izquierdo, quedándonos una vez más un vector con un solo elemento:



Y como es igual al valor buscado el proceso concluye habiéndose ubicado el valor en el quinto elemento (el valor de k).

Veamos que sucede cuando en el vector no existe el valor buscado. Por ejemplo si en lugar del número 11 buscamos el número 12. Aplicando el procedimiento se llega al mismo vector de un solo elemento (x_5) y como el elemento central es menor al valor buscado (12) deberíamos proseguir la búsqueda en el vector derecho, pero como ya no existen más elementos al lado derecho el proceso concluye. Por lo tanto el valor buscado no existe cuando al realizar la búsqueda el vector de búsqueda se queda sin elementos y ello ocurre cuando el valor de la variable " i " es mayor al de la variable " j ".

20.2.1 Algoritmo y código empleando vectores dinámicos

Aun cuando el método de búsqueda binaria es por naturaleza recursivo puede ser programado sin dificultad de manera directa:

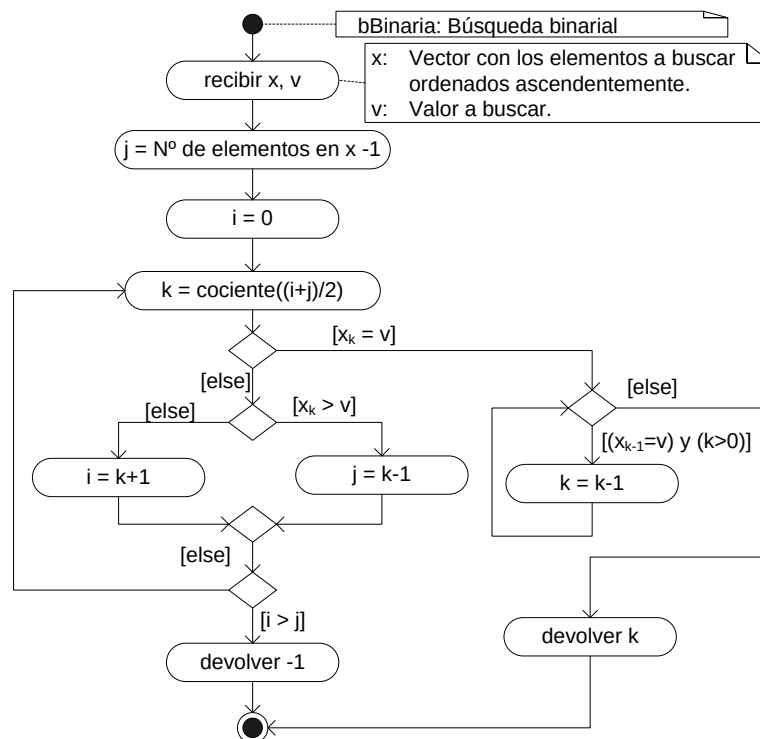


Figura 20.4. Algoritmo para el método de búsqueda binaria

Como se puede ver, una vez ubicado el valor buscado ($x_k=v$) se recorre la lista hacia atrás (con un ciclo While) para ubicar el primer elemento en caso de que existan valores repetidos.

El código elaborado empleando parámetros abiertos, vectores dinámicos y trabajando con vectores de números enteros es el siguiente:

```
unit uBinariaD;
```

interface

```

uses SysUtils, StdCtrls, QDialogs, uQuickSortD;

function bBinaria(x:array of integer; v:integer):integer;
function LeerEntero(m:TMemo):integer;
function GenerarVEnteros(n:cardinal; li,ls:integer):tvEnteros;

```

implementation

```

function bBinaria(x:array of integer; v:integer):integer;
var i,j,k:integer;
begin
    j:=High(x);
    i:=0;
    repeat
        k:=(i+j) div 2;
        if x[k]=v then begin
            while (x[k-1]=v) and (k>0) do dec(k);
            result:=k;
            exit;
        end;
        if x[k]>v then
            j:=k-1
        else
            i:=k+1;
        until i>j;
        result:=-1;
    end;

function LeerEntero(m:TMemo):integer;
begin
    try
        result:=StrToInt(m.Text);
    except
        on EConvertError do begin
            result:=0;
            ShowMessage('El número ha buscar está mal escrito');
        end;
    end;
end;

function GenerarVEnteros(n:cardinal; li,ls:integer):tvEnteros;
var x:tvEnteros; d:integer; i:cardinal;
begin
    SetLength(x,n);
    d:=ls-li+1;
    for i:=0 to n-1 do
        x[i]:=Random(d)+1;
    result:=x;
end;

end.

```

Como se puede ver en la figura de la siguiente página, la interfaz de la aplicación es esencialmente la misma que para el método de selección, sólo que en este caso al hacer click en el primer botón no sólo se genera, sino que también se ordena el vector de números enteros (por el método Quick Sort).

Nº	Valor
5832	5779
5833	5779
5834	5780
5835	5781
5836	5782
5837	5783
5838	5784
5839	5784
5840	5784
5841	5784
5842	5785
5843	5786
5844	5787
5845	5787
5846	5787
5847	5790
5848	5795

Los eventos que difieren de la aplicación elaborada para el método Secuencial son los eventos "onClick" de los BitBtns (no se debe olvidar incluir las unidades respectivas):

```
uses uBinariaD,uQuickSortD;
```

```
procedure TForm1.BitBtn1Click(Sender: TObject);
```

```
var x:tvEnteros;
```

```
begin
```

```
  x:=GenerarVEnteros(10000,1,10000);
```

```
  Quick(x);
```

```
  MostrarVector(x,StringGrid1);
```

```
  Mem1.SetFocus;
```

```
  StringGrid1.Tag:=1;
```

```
  Finalize(x);
```

```
end;
```

```
procedure TForm1.BitBtn2Click(Sender: TObject);
```

```
var v,pe:integer;
```

```
const x:tvEnteros=nil;
```

```
begin
```

```
  if StringGrid1.Tag=1 then begin
```

```
    StringGrid1.Tag:=0;
```

```
    Finalize(x);
```

```
    StringGrid1.Row:=0;
```

```
    x:=LeerVector(StringGrid1);
```

```

end;
v:=LeerEntero(Memol);
pe:=bBinaria(x,v);
if pe<0 then begin
  ShowMessage('No existe el número: '+IntToStr(v)+' en el vector');
  StringGrid1.Row:=1; end
else
  StringGrid1.Row:=pe+1;
end;
end;

```

20.2.2 Algoritmo y código empleando punteros

Como ya se dijo, el método de búsqueda binaria es por naturaleza recursivo por lo que puede ser programado haciendo uso de la recursividad. El algoritmo del módulo principal, considerando el uso de punteros, se presenta en la Figura 20.5 y el algoritmo del submódulo recursivo en la Figura 20.6.

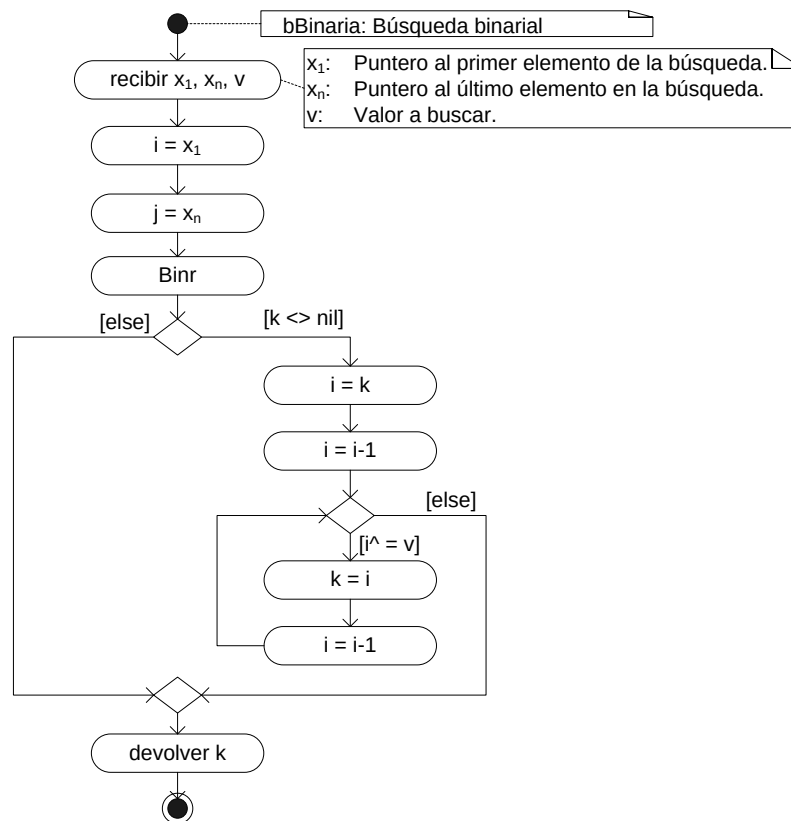


Figura 20.5. Búsqueda binaria - Módulo principal

El código elaborado en base a este algoritmo es el siguiente:

```
unit uBinariaP;
```

```
interface
```

```
uses SysUtils, StdCtrls, QDialogs, uQuickSortP;
```

```
function bBinaria(x1,xn:pEnteros; v:integer):pEnteros;
```

```
function LeerEntero(m:TMemo):integer;
```

```
function GenerarVEnteros(n:cardinal; li,ls:integer):pEnteros;
```

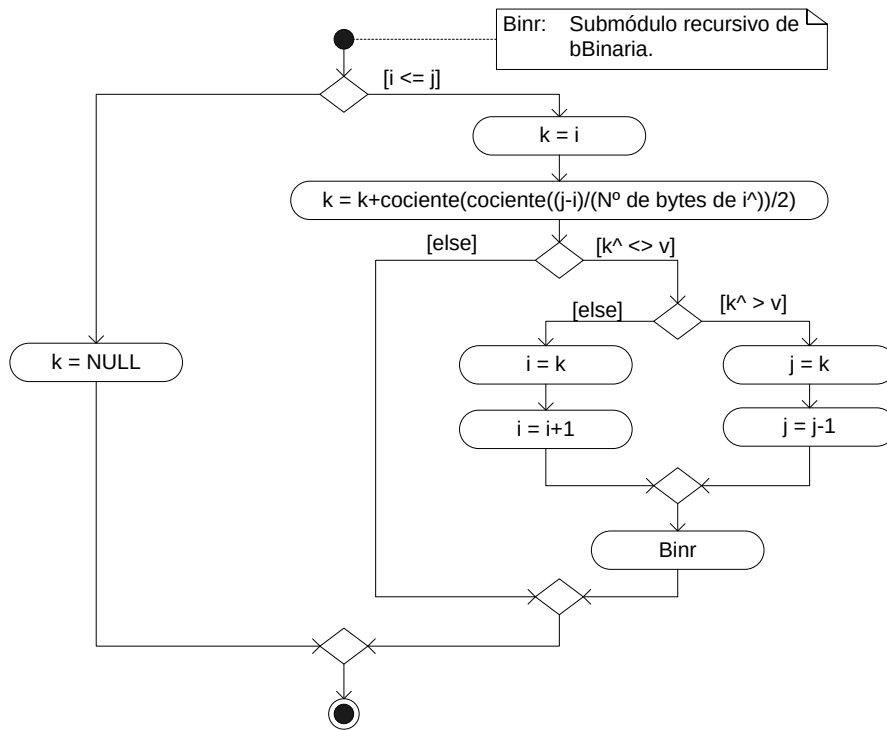


Figura 20.6. Búsqueda Binaria - Submódulo recursivo

implementation

```

function bBinaria(x1,xn:pEnteros; v:integer):pEnteros;
var i,j,k:pEnteros;
procedure Binr;
begin
  if Integer(i)<=Integer(j) then begin
    k:=i;
    inc(k,((Integer(j)-Integer(i)) div SizeOf(i^)) div 2);
    if k^<>v then begin
      if k^>v then begin
        j:=k;
        dec(j);
      end
      else begin
        i:=k;
        inc(i);
      end;
      Binr;
    end;
  else
    k:=nil;
  end;
begin
  i:=x1;
  j:=xn;
  Binr;
  if k<>nil then begin
    i:=k;
    dec(i);
  end;
end;

```

```

        while i^=v do begin
            k:=i;
            dec(i);
        end;
    end;
    result:=k;
end;

function LeerEntero(m:TMemo):integer;
begin
    try
        result:=StrToInt(m.Text);
    except
        on EConvertError do begin
            result:=0;
            ShowMessage('El número ha buscar está mal escrito');
        end;
    end;
end;

function GenerarVEnteros(n:cardinal; li,ls:integer):pEnteros;
var x,p:pEnteros; d:integer; i:cardinal;
begin
    GetMem(x,n*SizeOf(x^));
    p:=x;
    d:=ls-li+1;
    for i:=1 to n do begin
        p^:=Random(d)+1;
        inc(p);
    end;
    result:=x;
end;

end.

```

El programa de prueba es esencialmente el mismo que el del anterior acápite (y la del método secuencial). Los módulos que sufren alguna modificación importante son los correspondientes a los eventos "onClick" de los BitBtn:

```

uses uBinariaP,uQuickSortP;

procedure TForm1.BitBtn1Click(Sender: TObject);
var x:pEnteros; const n=10000;
begin
    x:=GenerarVEnteros(n,1,n);
    Quick(x,n);
    MostrarVector(x,n,StringGrid1);
    StringGrid1.Tag:=1;
    Memol.SetFocus;
    FreeMem(x);
end;

procedure TForm1.BitBtn2Click(Sender: TObject);
var v:integer; n:cardinal; pe:pEnteros;
const x:pEnteros=nil; xn:pEnteros=nil;
begin
    if StringGrid1.Tag=1 then begin
        FreeMem(x);
    end;
end;

```

```

    LeerVector(StringGrid1,x,n);
    xn:=x;
    inc(xn,n-1);
    StringGrid1.Tag:=0;
end;
v:=LeerEntero(Mem01);
pe:=bBinaria(x,xn,v);
if pe=nil then begin
    ShowMessage('No existe el número: '+IntToStr(v)+' en el vector');
    StringGrid1.Row:=1;
end
else begin
    n:=(Integer(pe)-Integer(x)) div SizeOf(x^);
    StringGrid1.Row:=n+1;
end;
end;
end;

```

20.3 INTERCALACIÓN

La intercalación es la operación por la cual se unen dos listas ordenadas para formar una tercera igualmente ordenada. Por supuesto las listas podrían ser simplemente añadidas y posteriormente ordenadas con uno de los métodos de ordenación estudiados, sin embargo, el proceso de ordenación consume más tiempo que el de intercalación y dicha operación implica la repetición innecesaria de operaciones (se vuelven a ordenar listas que ya estaban ordenadas).

La intercalación puede ser empleada también para ordenar listas muy grandes. En este caso se ordenan segmentos de la lista empleando uno de los métodos de ordenación y luego se intercalan los segmentos ordenados, logrando así ordenar toda la lista.

En la intercalación los elementos de las listas se van añadiendo a la lista resultante de forma tal que la misma siempre queda ordenada. Para ello simplemente se compara un elemento de una de las listas con otro elemento de la otra (comenzando con el primer elemento de cada una de las listas) y se añade a la lista resultante el menor de ellos. Esta operación se repite (sin tomar en cuenta los elementos añadidos) hasta que no quedan más elementos en una de las listas. Cuando esto sucede, los elementos de la otra lista (la que quedó con elementos) se añaden al final de la lista resultante.

Por ejemplo si queremos intercalar las siguientes listas:

i	1	2	3	4	5	6
x	<u>6</u>	15	23	30	34	40
y	5	<u>10</u>	25	26	30	

Comenzamos comparando los primeros elementos de ambas listas (6 con 5) y puesto que 5 es menor que 6, añadimos este valor a la lista resultante (z):

i	1	2	3	4	5	6
x	<u>6</u>	15	23	30	34	40
y	<u>5</u>	<u>10</u>	25	26	30	
z	5					

Entonces comparamos 6 con 10 y puesto que 6 es menor que 10, dicho valor es añadido a la lista:

i	1	2	3	4	5	6
x	<u>6</u>	<u>15</u>	23	30	34	40
y	<u>5</u>	<u>10</u>	25	26	30	
z	5	6				

Ahora comparamos 15 con 10 y dado que 10 es menor a 15 añadimos 10 a la lista resultante:

i	1	2	3	4	5	6
x	~~6~~	15	23	30	34	40
y	~~5~~	~~10~~	25	26	30	
z	5	6	10			

Proseguimos de esa manera: comparamos 15 con 25 (añadimos el número 15), luego 23 con 25 (añadimos el número 23), 30 con 25 (añadimos el número 25), 30 con 26 (añadimos el número 26), 30 con 30 (como 30 no es menor a 30, añadimos el número 30 de la segunda lista). En este punto se acaban los elementos de la lista "y":

i	1	2	3	4	5	6	7	8	9	10	11
x	~~6~~	~~15~~	~~23~~	30	34	40					
y	~~5~~	~~10~~	~~25~~	~~26~~	~~30~~						
z	5	6	10	15	23	25	26	30			

Entonces los elementos restantes de la lista "x" (desde x_4 hasta x_6) son añadidos al final de la lista resultante, con lo que obtenemos la lista intercalada ("z"):

i	1	2	3	4	5	6	7	8	9	10	11
x	~~6~~	~~15~~	~~23~~	~~30~~	~~34~~	~~40~~					
y	~~5~~	~~10~~	~~25~~	~~26~~	~~30~~						
z	5	6	10	15	23	25	26	30	30	34	40

20.3.1 Algoritmo y código empleando vectores dinámicos

El algoritmo para la intercalación de dos vectores ordenados ascendentemente, considerando el uso de vectores dinámicos o parámetros abiertos, se presenta en el diagrama de actividades de la Figura 20.7. El código elaborado en base a dicho algoritmo, trabajando con vectores de números enteros, es el siguiente:

```
unit uIntercalarD;
```

```
interface
```

```
uses uQuickSortD;
```

```
function Intercalar(x,y: array of integer):tvEnteros;
```

```
function GenerarVEnteros(n:cardinal; li,ls:integer):tvEnteros;
```

```
implementation
```

```
function Intercalar(x,y: array of integer):tvEnteros;
```

```
var i,j,k,nx,ny:integer; z:tvEnteros;
```

```
begin
```

```
  i:=0;
```

```
  j:=0;
```

```
  k:=0;
```

```
  nx:=Length(x);
```

```
  ny:=Length(y);
```

```
  SetLength(z,nx+ny);
```

```
  while (i<nx) and (j<ny) do begin
```

```
    if x[i]<y[j] then begin
```

```
      z[k]:=x[i];
```

```
      inc(i); end
```

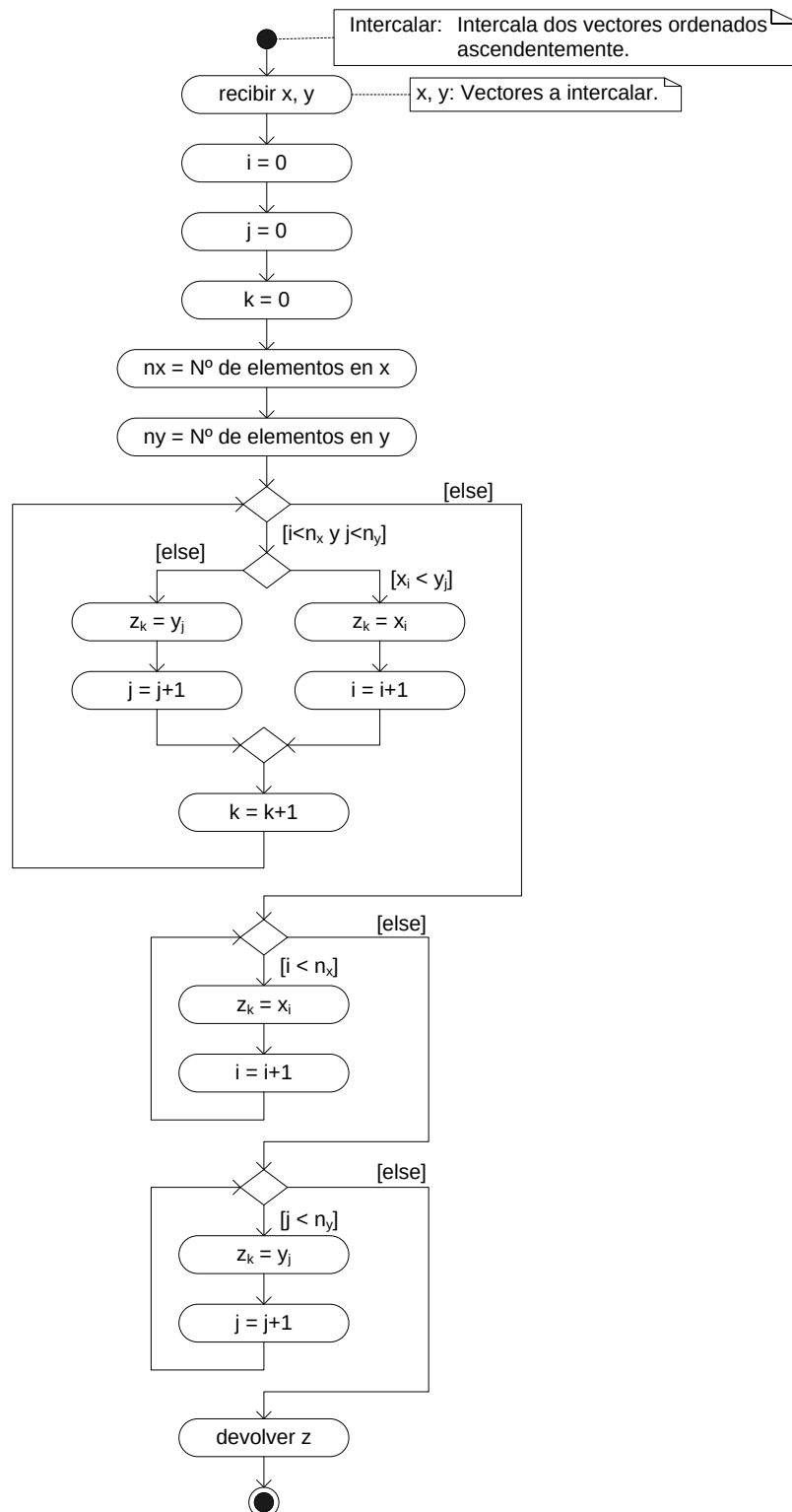


Figura 20.7. Intercalación - Vectores dinámicos

```

else begin
    z[k]:=y[j];
    inc(j);
end;

```

```

    inc(k);
end;
while i<nx do begin
    z[k]:=x[i];
    inc(k); inc(i);
end;
while j<ny do begin
    z[k]:=y[j];
    inc(k);inc(j);
end;
result:=z;
end;

function GenerarVEnteros(n:cardinal; li,ls:integer):tvEnteros;
var x:tvEnteros; d:integer; i:cardinal;
begin
    SetLength(x,n);
    d:=ls-li+1;
    for i:=0 to n-1 do x[i]:=Random(d)+1;
    result:=x;
end;

```

end.

Para probar el código se ha elaborado una aplicación con 4 Panels, 3 Labels, 3 StringGrids y 2 BitBtns, cuya apariencia es la siguiente:

INTERCALACIÓN - VECTORES DINÁMICOS

VECTOR X:		VECTOR Y:		VECTOR Z:	
Nº	Valor	Nº	Valor	Nº	Valor
1	1	1	20	1	1
2	4	2	29	2	4
3	7	3	36	3	7
4	38	4	38	4	20
5	39	5	47	5	29
6	51	6	48	6	36
7	58	7	52	7	38
8	66	8	75	8	38
9	77	9	91	9	39
10	97	10	97	10	47
11	97	11	121	11	48
12	103	12	134	12	51
13	113	13	135	13	52
14	115	14	147	14	58

Las propiedades de los componentes han sido modificadas en el evento "on-Create" de la forma:

```
uses uIntercalarD, UQuickSortD;
```

```
procedure TForm1.FormCreate(Sender: TObject);
```

```
begin
```

```
    Form1.Position:=poScreenCenter;
    Form1.Caption:='INTERCALACIÓN - VECTORES DINÁMICOS';
    Form1.BorderStyle:=bsDialog;
    Panel1.Align:=alBottom;
    Panel1.Height:=40;
    Panel1.Caption:='';
    StringGrid1.ColCount:=2;
    StringGrid1.RowCount:=2;
    StringGrid1.Cells[0,0]:=Format('%15s', ['N°']);
    StringGrid1.Cells[1,0]:=Format('%15s', ['Valor']);
    StringGrid1.ColWidths[0]:=80;
    StringGrid1.ColWidths[1]:=100;
    StringGrid1.Width:=StringGrid1.ColWidths[0]+StringGrid1.ColWidths[1]+25;
    Panel2.Align:=alLeft;
    Panel2.Caption:='';
    Panel2.Alignment:=taLeftJustify;
    Panel2.Width:=StringGrid1.Width+2;
    StringGrid1.Parent:=Panel2;
    StringGrid1.Top:=25;
    StringGrid1.Left:=0;
    StringGrid1.Height:=Panel2.ClientHeight-25;
    StringGrid1.Options:=StringGrid1.Options+[goThumbTracking];
    Label1.Parent:=Panel2;
    Label1.Caption:='VECTOR X: ';
    Label1.Font.Style:=[fsBold];
    Label1.Left:=(Panel2.ClientWidth-Label1.Width) div 2;
    Label1.Top:=(25-Label1.Height) div 2;
    StringGrid2.ColCount:=2;
    StringGrid2.RowCount:=2;
    StringGrid2.Cells[0,0]:=Format('%15s', ['N°']);
    StringGrid2.Cells[1,0]:=Format('%15s', ['Valor']);
    StringGrid2.ColWidths[0]:=80;
    StringGrid2.ColWidths[1]:=100;
    StringGrid2.Width:=StringGrid2.ColWidths[0]+StringGrid2.ColWidths[1]+25;
    Panel3.Left:=Panel2.Left+Panel2.Width;
    Panel3.Top:=0;
    Panel3.Height:=Panel2.Height;
    Panel3.Caption:='';
    Panel3.Width:=StringGrid2.Width+2;
    StringGrid2.Parent:=Panel3;
    StringGrid2.Top:=25;
    StringGrid2.Left:=0;
    StringGrid2.Height:=Panel3.ClientHeight-25;
    StringGrid2.Options:=StringGrid2.Options+[goThumbTracking];
    Label2.Parent:=Panel3;
    Label2.Caption:='VECTOR Y: ';
    Label2.Font.Style:=[fsBold];
    Label2.Left:=(Panel2.ClientWidth-Label1.Width) div 2;
    Label2.Top:=(25-Label1.Height) div 2;
    StringGrid3.ColCount:=2;
    StringGrid3.RowCount:=2;
    StringGrid3.Cells[0,0]:=Format('%18s', ['N°']);
```

```

StringGrid3.Cells[1,0]:=Format('%25s',['Valor']);
StringGrid3.ColWidths[0]:=80;
StringGrid3.ColWidths[1]:=100;
StringGrid3.Width:=StringGrid3.ColWidths[0]+StringGrid3.ColWidths[1]+25;
Panel4.Left:=Panel3.Left+Panel3.Width;
Panel4.Top:=0;
Panel4.Height:=Panel2.Height;
Panel4.Caption:='';
Panel4.Width:=StringGrid3.Width+2;
StringGrid3.Parent:=Panel4;
StringGrid3.Left:=0;
StringGrid3.Top:=25;
StringGrid3.Height:=Panel4.ClientHeight-25;
StringGrid3.Options:=StringGrid3.Options+[goThumbTracking];
Label3.Parent:=Panel4;
Label3.Caption:='VECTOR Z: ';
Label3.Font.Style:=[fsBold];
Label3.Left:=(Panel2.ClientWidth-Label1.Width) div 2;
Label3.Top:=(25-Label1.Height) div 2;
Form1.Width:=Panel2.Width+Panel3.Width+Panel4.Width;
BitBtn1.Parent:=Panel1;
BitBtn1.Width:=140;
BitBtn1.Left:=(Panel1.ClientWidth-BitBtn1.Width*2-100) div 2;
BitBtn1.Top:=(Panel1.ClientHeight-BitBtn1.Height) div 2;
BitBtn1.Glyph.LoadFromFile('C:\Archivos de programa\'+
  'Archivos comunes\Borland Shared\Images\Buttons\Sort.bmp');
BitBtn1.NumGlyphs:=2;
BitBtn1.Caption:='&Generar y ordenar';
BitBtn1.Cursor:=crHandPoint;
BitBtn2.Parent:=Panel1;
BitBtn2.Width:=140;
BitBtn2.Left:= 300;
BitBtn2.Top:=BitBtn1.Top;
BitBtn2.Left:=BitBtn1.Left+BitBtn1.Width+100;
BitBtn2.Glyph.LoadFromFile('C:\Archivos de programa\'+
  'Archivos comunes\Borland Shared\Images\Buttons\Calculat.bmp');
BitBtn2.NumGlyphs:=2;
BitBtn2.Caption:='&Intercalar';
BitBtn2.Cursor:=crHandPoint;
end;

```

En el evento "onClick" del BitBtn1 se generan dos vectores, uno con 1000 y otro con 1200 elementos, son ordenados con el método Quick Sort y mostrados en los StringGrids 1 y 2 respectivamente:

```

procedure TForm1.BitBtn1Click(Sender: TObject);
var x,y: tvEnteros;
begin
  x:=GenerarVEnteros(1000,1,10000);
  Quick(x);
  MostrarVector(x,StringGrid1);
  y:=GenerarVEnteros(1200,1,10000);
  Quick(y);
  MostrarVector(y,StringGrid2);
  Finalize(x);
  Finalize(y);
end;

```

En el evento "onClick" del BitBtn2, se leen los vectores de los StringGrids 1 y 2, se intercalan y se muestra el vector resultante en el

StringGrid3:

```
procedure TForm1.BitBtn2Click(Sender: TObject);
var x,y,z:tvEnteros;
begin
  x:=LeerVector(StringGrid1);
  y:=LeerVector(StringGrid2);
  z:=Intercalar(x,y);
  MostrarVector(z,StringGrid3);
  Finalize(x);
  Finalize(y);
  Finalize(z);
end;
```

20.3.2 Algoritmo y código empleando punteros

El algoritmo de intercalación, trabajando con punteros, se presenta en la Figura 20.8 y el código elaborado en base al mismo es el siguiente:

```
unit uIntercalarP;

interface

  uses uQuickSortP;

  function Intercalar(x1,xn,y1,yn:pEnteros):pEnteros;
  function GenerarVEnteros(n:cardinal; li,ls:integer):pEnteros;

implementation

  function Intercalar(x1,xn,y1,yn:pEnteros):pEnteros;
  var z,z1:pEnteros; nx,ny:integer;
  begin
    nx:=(Integer(xn)-Integer(x1)) div SizeOf(x1^)+1;
    ny:=(Integer(yn)-Integer(y1)) div SizeOf(y1^)+1;
    GetMem(z, (nx+ny)*SizeOf(z^));
    z1:=z;
    while (Integer(x1)<=Integer(xn)) and (Integer(y1)<=Integer(yn)) do begin
      if x1^<y1^ then begin
        z^:=x1^;
        inc(x1);
      end
      else begin
        z^:=y1^;
        inc(y1);
      end;
      inc(z);
    end;
    while Integer(x1)<=Integer(xn) do begin
      z^:=x1^;
      inc(x1);
      inc(z);
    end;
    while Integer(y1)<=Integer(yn) do begin
      z^:=y1^;
      inc(y1);
      inc(z);
    end;
    result:=z1;
```

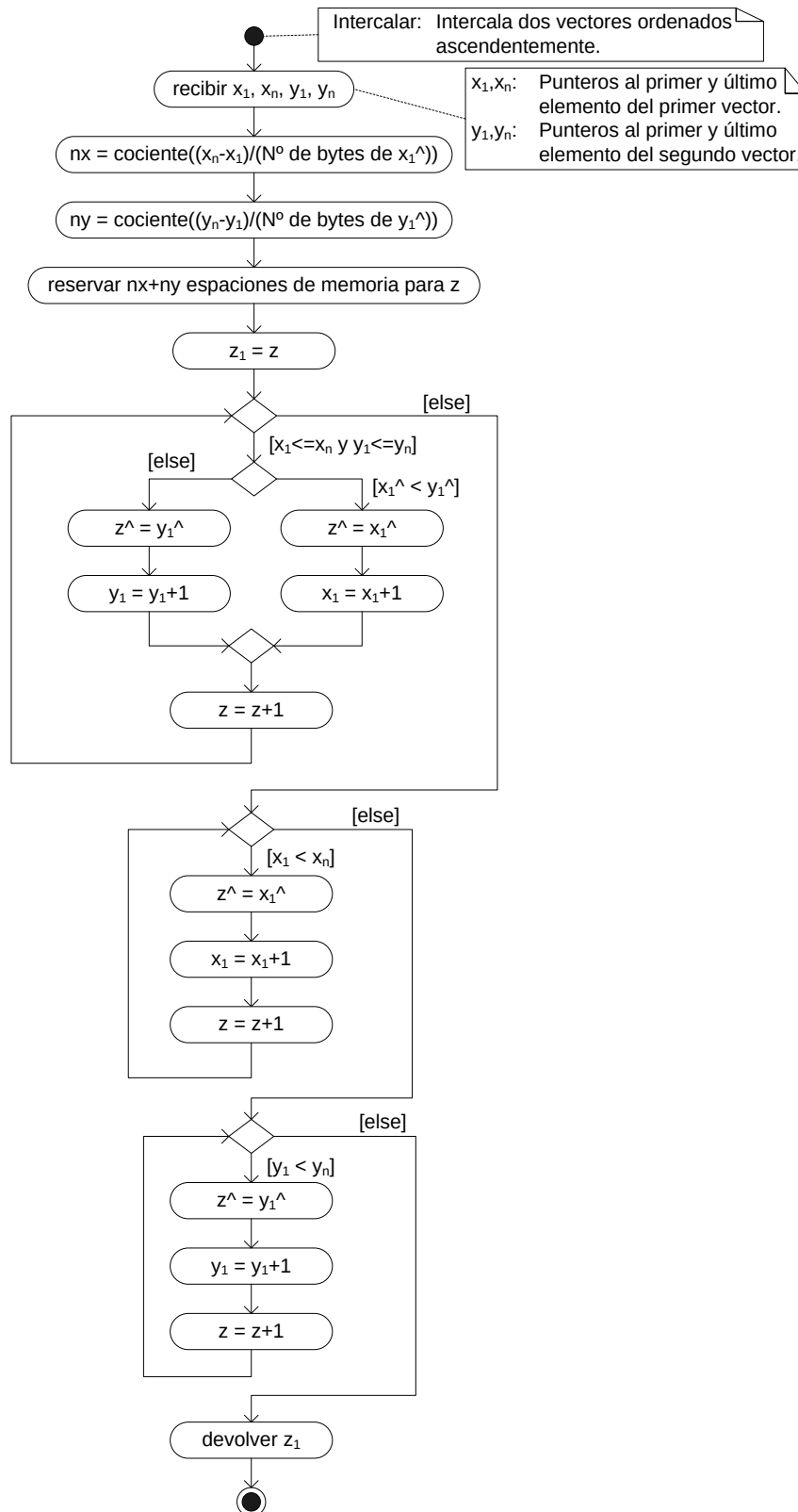


Figura 20.8. Intercalación - Punteros

end;

```

function GenerarVEnteros(n:cardinal; li,ls:integer):pEnteros;
var x,p:pEnteros; d:integer; i:cardinal;
begin
  GetMem(x,n*SizeOf(x^));
  p:=x;
  d:=ls-li+1;
  for i:=1 to n do begin
    p^:=Random(d)+1;
    inc(p);
  end;
  result:=x;
end;

```

end.

La aplicación de prueba es esencialmente la misma que la del anterior acápite. Los módulos que difieren con relación al mismo son los correspondientes a los eventos "onClick" de los BitBtns 1 y 2:

```

procedure TForm1.BitBtn1Click(Sender: TObject);
var x,y: pEnteros;
const nx=1000; ny=1200;
begin
  x:=GenerarVEnteros(nx,1,10000);
  Quick(x,nx);
  MostrarVector(x,nx,StringGrid1);
  y:=GenerarVEnteros(ny,1,10000);
  Quick(y,ny);
  MostrarVector(y,ny,StringGrid2);
  FreeMem(x);
  FreeMem(y);
end;

```

```

procedure TForm1.BitBtn2Click(Sender: TObject);
var x,xn,y,yn,z:pEnteros; nx,ny,nz:cardinal;
begin
  LeerVector(StringGrid1,x,nx);
  xn:=x;
  inc(xn,nx-1);
  LeerVector(StringGrid2,y,ny);
  yn:=y;
  inc(yn,ny-1);
  z:=Intercalar(x,xn,y,yn);
  nz:=nx+ny;
  MostrarVector(z,nz,StringGrid3);
  FreeMem(x);
  FreeMem(y);
  FreeMem(z);
end;

```

20.4 Ejercicios

1. Elabore una aplicación que trabajando con punteros encuentre un nombre, en un vector desordenado de nombres completos, empleando el método de búsqueda Secuencial. Para buscar el nombre es suficiente que estén escritas las primeras letras del mismo.

2. Elabore una aplicación que trabajando con vectores dinámicos, parámetros abiertos y la recursividad, encuentre un nombre, en un vector ordenado con nombres completos, empleando el método de búsqueda binaria. Para buscar el nombre es suficiente que estén escritas las primeras letras del mismo.
3. Elabore una aplicación que trabajando con punteros (sin emplear la recursividad) encuentre un nombre, en un vector ordenado con nombres completos, empleando el método de búsqueda binaria. Para buscar el nombre es suficiente que estén escritas las primeras letras del mismo.
4. Elabore una aplicación que trabajando con vectores dinámicos y/o parámetros abiertos, intercale dos listas con 2500 y 3200 nombres completos generados al azar.
5. Elabore una aplicación que trabajando con punteros, intercale dos listas con 3050 y 2700 nombres completos generados al azar.

21. CONJUNTOS

Los conjuntos constituyen el segundo tipo de dato estructurado que estudiaremos en esta materia. Los conjuntos son datos estructurados porque están conformados por dos o más datos (elementos) de tipo simple (ordinal).

En Pascal, un conjunto es un grupo de 0 a 256 elementos de tipo ordinal. El tipo ordinal puede ser byte, char, enumerado o subrango (siempre que el subrango esté comprendido entre 0 y 255). Al igual que en teoría de conjuntos, en los conjuntos de Pascal no pueden existir elementos duplicados.

Un conjunto se declara empleando las palabras reservadas *set of* seguidas del tipo de dato ordinal, por ejemplo:

```
var c1,c2 : set of char;
```

Como sucede con los otros tipos de datos, las variables conjuntos se pueden declarar directamente, como en el anterior ejemplo o declarando primero un tipo y después variables de ese tipo, por ejemplo:

```
type
  tletras = set of 'A'..'Z';
var
  vocales, consonantes : tletras;
```

Es posible también declarar constantes de conjuntos, por ejemplo:

```
const vocales = ['A','E','I','O','U']
      TecladoExtendido = [#72,#80,#87,#85,#123]
```

Igualmente se pueden declarar variables inicializadas (o variables locales estáticas) empleando conjuntos, por ejemplo:

```
const
  Aprobados : set of byte = [51,80,75,90];
  Reprobados : set of byte = [10,15,20,25];
  Vocales : set of char = ['A','E','I','O','U'];
```

El conjunto vacío (un conjunto sin elementos) se asigna escribiendo los corchetes sin nada adentro, por ejemplo, dada las variables conjuntos *con1* y *con2*, las siguientes sentencias les asignan conjuntos vacíos:

```
con1 := [];
con2 := [];
```

El conjunto universo es el conjunto que contiene todos los elementos posibles dentro del dominio. Se asigna colocando los límites inferior y superior separados por dos puntos, así por ejemplo si *con3* es una variable conjunto de tipo *byte*, la siguiente sentencia le asigna el conjunto universo:

```
con3 := [0..255];
```

21.3 OPERACIONES CON CONJUNTOS

Con los conjuntos se pueden realizar las siguientes operaciones:

21.3.1 Pertenencia (*in*)

La operación de pertenencia es probablemente la operación más utilizada en conjuntos, se emplea para verificar si un determinado elemento pertenece o no a un conjunto. Por ejemplo, la siguiente sentencia verifica si el día *lunes* está en el conjunto *laborables*:

```
if lunes in laborables then
```

```
ShowMessage('es un día laborable');
```

21.3.2 Unión (+)

La unión (+) de dos conjuntos devuelve un conjunto que contiene todos los elementos que existen en ambos conjuntos (recuerde que un conjunto no puede tener elementos duplicados).

Por ejemplo, dado los conjuntos

```
A := [lunes, martes, viernes];
```

```
B := [lunes, miércoles, viernes, sábado];
```

La operación A+B devuelve:

```
A+B => [lunes, martes, miércoles, viernes, sábado].
```

21.3.3 Intersección (*)

La intersección (*) de dos conjuntos devuelve un conjunto que contiene los elementos que se encuentran tanto en el primer como en el segundo conjunto.

Por ejemplo, dado los conjuntos:

```
A := [enero, marzo, abril, julio];
```

```
B := [enero, febrero, mayo, julio, agosto];
```

La operación A*B devuelve:

```
A*B => [enero, julio]
```

21.3.4 Diferencia (-)

La diferencia (-) de dos conjuntos devuelve un conjunto que contiene todos los elementos que se encuentran en el primer conjunto pero que no se encuentran en el segundo.

Por ejemplo, dado los conjuntos:

```
A := [rojo, verde, azul];
```

```
B := [rojo, amarillo, verde, rosado, marrón];
```

La operación A-B devuelve:

```
A-B => [azul]
```

21.3.5 Igualdad (=)

El operador relacional de igualdad (=) compara dos conjuntos y devuelve verdadero si los dos conjuntos tienen los mismos elementos y falso en caso contrario.

Por ejemplo el siguiente segmento de código compara los conjuntos *con1* y *con2* y muestra un mensaje indicando si los mismos son iguales o diferentes:

```
if con1 = con2 then
  Edit1.Text := 'Los conjuntos son iguales'
else
  Edit1.Text := 'Los conjuntos son diferentes';
```


21.3.6 Desigualdad ($\neq$)

Este operador relacional compara dos conjuntos y devuelve verdadero si los dos conjuntos difieren en al menos un elemento.

Por ejemplo dados los conjuntos:

```
A := ['A','E','I'];
B := ['A','E','I','O'];
C := ['A','E','I'];
```

La operación $A \neq B$ devuelve verdadero, mientras que la operación $A \neq C$ devuelve falso.

21.3.7 Subconjunto ($\subseteq$)

Este operador relacional compara dos conjuntos y devuelven verdadero si el primer conjunto es un subconjunto del segundo, es decir si el segundo conjunto contiene todos los elementos del primero.

Por ejemplo dado los conjuntos:

```
A := [1, 2, 3];
B := [1, 2, 3, 4, 5];
```

La operación $A \subseteq B$ devuelve verdadero, mientras que $B \subseteq A$ devuelve falso.

21.3.8 Superconjunto ($\supseteq$)

Este operador es el complemento del anterior. Compara dos conjuntos y devuelve verdadero si el primero es un superconjunto del segundo, es decir si el primer conjunto contiene todos los elementos del primero.

Por ejemplo dados los conjuntos:

```
A := [1, 2, 3];
B := [1, 2, 3, 4, 5];
```

La operación $A \supseteq B$ devuelve falso, mientras que $B \supseteq A$ devuelve verdadero.

21.4 Procedimientos que operan con conjuntos

Con los conjuntos se pueden emplear los procedimientos *include* y *exclude*.

21.4.1 Include

El procedimiento *Include* tiene la siguiente sintaxis:

```
Include(variable conjunto, elemento)
```

Este procedimiento añade el *elemento* a la *variable conjunto*, es equivalente a la sentencia:

```
Variable conjunto := Variable conjunto + [elemento]
```

Sólo que genera un código más eficiente.

Por ejemplo dado el conjunto:

```
A := [1,4,7,8,9];
```

Include(A,5) devuelve el conjunto [1,4,5,7,8,9] e Include(A,7) devuelve el mismo conjunto ([1,4,7,8,9]) pues el elemento 7 ya pertenecía al conjunto y como ya se dijo: un conjunto no pueden contener elementos repetidos.

21.4.2 Exclude

El procedimiento *Exclude* tiene la siguiente sintaxis:

Exclude(variable conjunto, elemento)

Este procedimiento excluye un *elemento* de la *variable conjunto*, es equivalente a la sentencia:

Variable conjunto := Variable conjunto - [elemento]

Sólo que genera un código más eficiente.

Por ejemplo, dado el conjunto:

A := [1,3,7,9];

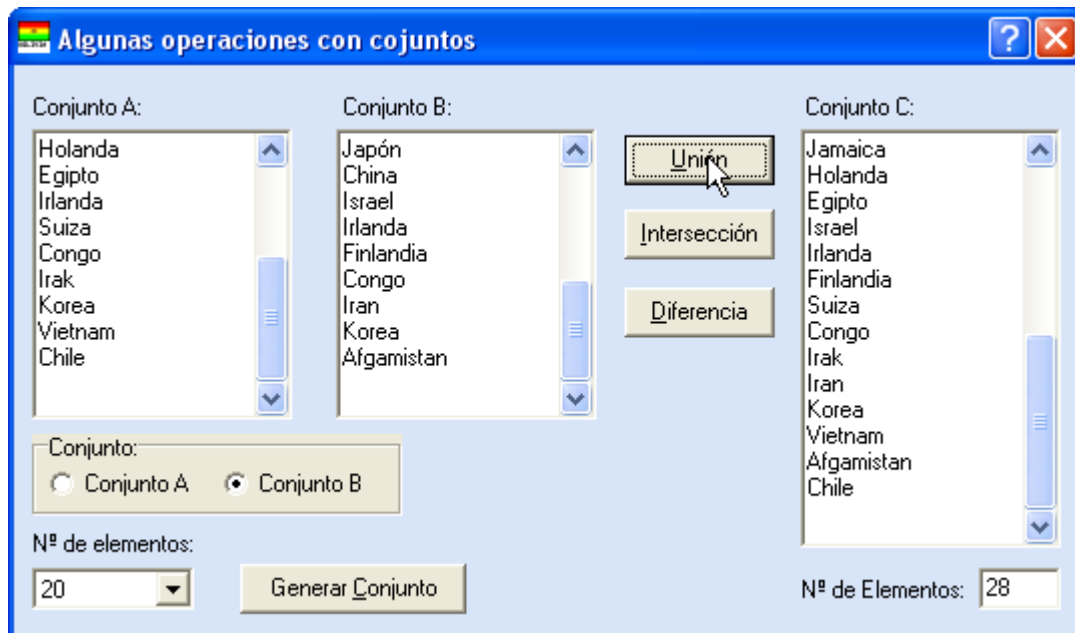
Exclude(A,3) devuelve el conjunto [1,7,9] y Exclude(A,2), devuelve el mismo conjunto ([1,3,7,9]) pues el elemento 2 no pertenece al conjunto y no se puede excluir de un conjunto un elemento que no existe.

21.5 EJEMPLOS

21.5.1 Ejemplo 1

Como primer ejemplo elaboraremos una aplicación donde generaremos al azar dos conjuntos con los nombres de algunos países y con los mismos se podrán llevar a cabo las operaciones de unión, intersección y diferencia.

En ejecución la aplicación tiene la apariencia que se muestra la siguiente figura:



La misma, como se puede ver, consta de 3 Memos, 4 Botones, un RadioGroup, un ComboBox, un Edit y los respectivos Label (con su propiedad Transparent en True). No se explica nada con relación a la elaboración de esta interfaz porque no existen componentes nuevos y en consecuencia puede ser construida fácilmente en base a la figura.

Para generar conjuntos con los nombres de algunos países, primero debemos crear un tipo enumerado con algunos de dichos nombres en una nueva unidad:

```
unit Unit2;

interface

uses StdCtrls;

type
  tpais=(Bolivia,Argentina,Paraguay,Uruguay,Ecuador,
    Peru,Colombia,Brasil,Mexico,Canada,Espana,
    Alemania,Inglaterra,Francia,Japon,Polonia,
    China,Jamaica,Ghana,Antillas,Holanda,India,
    Egipto,Israel,Irlanda,Finlandia,Suiza,Congo,
    Irak,Iran,Korea,Vietnam,Afgamistan,Pakistan,
    Chile);
  tcpais = set of tpais;
```

Con este tipo ordinal se puede trabajar con conjuntos (recuerde que los conjuntos sólo pueden operar con datos de tipo ordinal), sin embargo, los tipos enumerados al no ser estándar, no pueden ser visualizados directamente, por lo que es necesario implementar también alguna forma de visualizarlos. Frecuentemente lo más sencillo y eficiente es visualizarlos en forma de cadena y para que exista una correspondencia directa entre el tipo de dato y la cadena, es conveniente crear un vector cuyos elementos sean los mismos que el del tipo enumerado, pero por supuesto en forma de cadenas:

```
const pais:array [Bolivia..Chile] of string[10]= (
  'Bolivia','Argentina','Paraguay','Uruguay',
  'Ecuador','Perú','Colombia','Brasil','México',
  'Canadá','España','Alemania','Inglaterra',
  'Francia','Japón','Polonia','China','Jamaica',
  'Ghana','Antillas','Holanda','India',
  'Egipto','Israel','Irlanda','Finlandia','Suiza',
  'Congo','Irak','Irán','Korea','Vietnam',
  'Afgamistán','Pakistán','Chile');
```

Así el valor de país[Uruguay] es la cadena 'Uruguay', el de país[Iran] es 'Irán', el de país[Francia] es 'Francia', etc.

De esa manera con los tipos "tpais", "tcpais" y con el vector constante "pais", se pueden crear las funciones para generar, mostrar, leer y contar el número de elementos en un conjunto con nombres de países:

```
function GenerarCPaises(ne:byte):tcpais;
procedure MostrarCPaises(c:tcpais; m:TMemo);
function LeerCPaises(m:TMemo):tcpais;
function ContarCPaises(c:tcpais):byte;
```

implementation

```
function GenerarCPaises(ne:byte):tcpais;
var e:TPais; i:byte; c:tcpais;
begin
  i:=0;
  c:=[];
  while i<ne do begin
    e:=TPais(Random(35));
    if not (e in c) then begin
      include(c,e);
```

```

        inc(i);
    end;
end;
result:=c;
end;

procedure MostrarCPaises(c:tcpais; m:TMemo);
var i:tpais;
begin
    m.Lines.Clear;
    for i:=Bolivia to Chile do
        if i in c then m.Lines.Append(pais[i]);
    end;

function LeerCPaises(m:TMemo):tcpais;
var i:tpais; k:byte;
begin
    result:=[];
    for k:=0 to m.Lines.Count-1 do
        for i:=Bolivia to Chile do
            if m.Lines[k]=pais[i] then
                begin include(result,i); break; end;
        end;
    end;

function ContarCPaises(c:tcpais):byte;
var i:tpais;
begin
    result:=0;
    for i:=Bolivia to Chile do
        if i in c then inc(result);
    end;

end.

    Los eventos programados son los siguientes:
procedure TForm1.FormCreate(Sender: TObject);
var i:byte;
begin
    Memo1.Clear;
    Memo2.Clear;
    Memo3.Clear;
    Form1.Brush.Color:=clInactiveCaptionText;
    ComboBox1.Clear;
    for i:=1 to 35 do
        ComboBox1.Items.Append(IntToStr(i));
    ComboBox1.ItemIndex:=0;
end;

procedure TForm1.Button1Click(Sender: TObject);
var c:tcpais; ne:byte;
begin
    ne:=StrToInt(ComboBox1.Text);
    c:=GenerarCPaises(ne);
    if RadioGroup1.ItemIndex=0 then
        MostrarCPaises(c,Memo1)
    else
        MostrarCPaises(c,Memo2);
end;

```

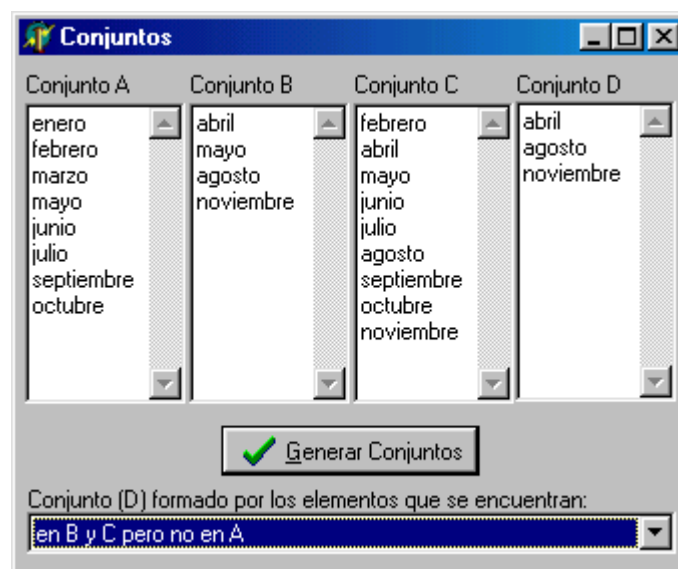
```

procedure TForm1.Button2Click(Sender: TObject);
var a,b,c: tcpais; boton:byte;
begin
    a:=LeerCPaises (Memo1);
    b:=LeerCPaises (Memo2);
    boton:=Ord(TButton(Sender).Name='Button2')+
            Ord(TButton(Sender).Name='Button3')*2+
            Ord(TButton(Sender).Name='Button4')*3;
    case boton of
        1: c:=a+b;
        2: c:=a*b;
        else
            c:=a-b;
    end;
    MostrarCPaises(c,Memo3);
    Edit1.Text:=IntToStr(ContarCPaises(c));
end;

```

21.5.2 Ejemplo 2

Como segundo ejemplo elaboraremos una aplicación donde se realizarán algunas operaciones con conjuntos. La interfaz de la aplicación tiene la siguiente apariencia:



Donde se emplean 4 memos para mostrar los elementos de los conjuntos (con su propiedad *ScrollBars* en *ssVertical*). En esta aplicación se realizan algunas operaciones con los conjuntos "A", "B" y "C" y el resultado de dichas operaciones se muestra en el conjunto "D".

Los elementos de los conjuntos son generados aleatoriamente al hacer click en el botón *Generar Conjuntos*.

Las operaciones a realizar se eligen en el *ComboBox* y son las siguientes: devolver el conjunto formado por los elementos que se encuentra en A o B; en A y B; en B o C; en B y C; en A y B pero no en C; en A y B o en A y C; en A y B pero no en A y C; en A o C pero no en B; en B y C pero no en A.

En esta aplicación, los elementos de los conjuntos son los meses del año. Entonces comenzamos declarando, en el sector de implementation, los tipos y variables que emplearemos en la aplicación:

type

```

    tMeses = (enero, febrero, marzo, abril, mayo,
              junio, julio, agosto, septiembre,
              octubre, noviembre, diciembre);
    tcMeses = set of tMeses;

```

const

```

    Meses: array[enero..diciembre] of string[10] =
        ('enero','febrero','marzo','abril','mayo',
         'junio','julio','agosto','septiembre',
         'octubre','noviembre','diciembre');

```

```

var A,B,C,D: tcMeses;

```

Al igual que en el anterior ejemplo es necesario declarar previamente un tipo enumerado: *tMeses* y un vector constante *Meses*, el cual almacena los meses del año en forma de cadenas de caracteres.

El procedimiento para generar los conjuntos es:

```

procedure GenerarConjunto(var c : tcMeses);
var i,n : integer; m : tMeses;
begin
    n := random(12)+1;
    i := 0; c := [];
    while i<n do
    begin
        m := tMeses(random(12));
        if not (m in c) then
        begin
            inc(i);
            include(c,m);
        end;
    end;
end;

```

Como se puede observar, el número de elementos del conjunto (*n*) es generado al azar, lo mismo que los elementos del conjunto, para esto último el número generado, comprendido entre 0 y 11, es convertido a su mes equivalente mediante el tipo *TMeses*, mediante *moldeo de tipos* (es decir empleando el tipo de dato *TMeses* como una función para convertir un entero en *Meses*).

Los elementos generados al azar se incluyen en el conjunto dentro de un ciclo *while*. En este caso no se puede emplear un ciclo *For*, pues aun cuando sabemos el número de elementos que debe contener el conjunto (*n*), los elementos se generan al azar y en consecuencia es probable que se generen dos o más valores repetidos y como un conjunto no puede contener elementos repetidos, es necesario descartar estos valores y generar otros nuevos, con lo que incrementa el número de veces que se repite el proceso.

Para mostrar en un Memo los elementos de un conjunto escribimos el siguiente procedimiento:

```

procedure MostrarConjunto(var c: tcMeses; var Memo: TMemo);
var m : tMeses;
begin
    Memo.Lines.Clear;
    for m:=enero to diciembre do
        if m in c then Memo.Lines.Add(Meses[m]);
end;

```

En este procedimiento se recibe como parámetros el conjunto (*c*) y el memo en el cual se mostrarán los elementos del conjunto (*Memo*). La sentencia *Memo.Lines.Add(Meses[m])* añade una línea con el mes "*m*" al memo, es en esta sentencia donde se emplea el array *Meses* para convertir el mes "*m*" (que es un tipo enumerado) en un string (que puede ser visualizado en el memo).

Los elementos de los conjuntos A, B y C se generan cuando se hace click sobre el botón Generar Conjuntos (BitBtn1). Entonces en el evento *OnClick* de dicho botón escribimos el siguiente código:

```
procedure TForm1.BitBtn1Click(Sender: TObject);
begin
    GenerarConjunto(A);
    MostrarConjunto(A,Memo1);
    GenerarConjunto(B);
    MostrarConjunto(B,Memo2);
    GenerarConjunto(C);
    MostrarConjunto(C,Memo3);
end;
```

Para llenar el conjunto D se elige una de las opciones disponibles en el ComboBox. Por lo tanto se debe escribir el código que efectúa las operaciones necesarias y llenar el Memo4 en el evento *OnChange* del ComboBox1:

```
procedure TForm1.ComboBox1Change(Sender: TObject);
begin
    case ComboBox1.ItemIndex of
        0 : D := A+B;
        1 : D := A*B;
        2 : D := B+C;
        3 : D := B*C;
        4 : D := A*B-C;
        5 : D := (A*B)+(A*C);
        6 : D := (A*B)-(A*C);
        7 : D := (A+C)-B;
        8 : D := (B*C)-A;
    end;
    MostrarConjunto(D,Memo4);
end;
```

Finalmente, para que al ejecutar la aplicación aparezca elegida una de las opciones del ComboBox, escribimos el siguiente código en el evento *OnCreate* de la forma:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    ComboBox1.ItemIndex := 0;
end;
```

21.5.3 Ejemplo 3

Como tercer ejemplo elaboraremos una aplicación para modificar algunas de las opciones de un StringGrid. Las opciones que se modificarán serán: *goFixedVertLine*, *goFixedHorzLine*, *goVertLine*, *goHorzLine*, *goRowMoving*, *goColMoving* y *goThumbTracking*.

Esta es una de las posibles aplicaciones de los conjuntos pues, como ya se informó previamente, la propiedad "Options" del "StringGrid" es un conjunto.

La aplicación sólo consta de un StringGrid, un GroupBox y 7 CheckBox (dentro del GroupBox). En ejecución tiene la apariencia que se muestra en la siguiente figura:

Posición	Número	Cuadrado
1	0.00	0.00
2	31.38	984.70
3	861.05	741404.46
4	202.58	41039.05
5	272.92	74486.02
6	671.65	451119.66
7	318.69	101564.13
8	161.80	26177.77
9	372.24	138561.40
10	425.67	181198.16

Opciones:

- ☒ FixedVertLine
- ☒ FixedHorzLine
- ☒ VertLine
- ☒ HorzLine
- ☐ RowMoving
- ☐ ColMoving
- ☐ ThumbTracking

Las propiedades de la forma y de los componentes han sido modificadas en el evento "onCreate" de la forma. Sólo para contar con valores de prueba el StringGrid ha sido llenado con 100 filas de números generadas al azar:

```

procedure TForm1.FormCreate(Sender: TObject);
var i,ic,t: integer; cb: TCheckBox;
    r: double;
const Captions: array [1..7] of string = (
    'FixedVertLine','FixedHorzLine','VertLine',
    'HorzLine','RowMoving','ColMoving','ThumbTracking');
begin
    Application.Icon.LoadFromFile('C:\Archivos de programa\Archivos comunes\''+
    'Borland Shared\Images\Icons\Chip.ico');
    Form1.Caption:='Opciones de un StringGrid';
    Form1.BorderStyle:= bsDialog;
    //Propiedades del StringGrid
    StringGrid1.ColCount:=3;
    StringGrid1.RowCount:=101;
    StringGrid1.DefaultColWidth:=90;
    StringGrid1.Width:=StringGrid1.DefaultColWidth*3+25;
    StringGrid1.Align:=alLeft;
    StringGrid1.Font.Name:='Courier New';
    //Ancho y alto de la forma
    Form1.Width:=StringGrid1.Width+150;
    Form1.Height:=300;
    //Propiedades del GroupBox
    GroupBox1.Align:=alRight;
    GroupBox1.Width:=Form1.ClientWidth-StringGrid1.Width;
    GroupBox1.Caption:='Opciones: ';
    //Propiedades de los CheckBox
    ic:=CheckBox1.ComponentIndex-1;
    t:=25;
    for i:=1 to 7 do begin
        cb:= Components[i+ic] as TCheckBox;
        cb.Caption:=Captions[i];
        cb.Left:=10;
        cb.Top:=t;
        inc(t,25);
    end

```



```

    if i<5 then cb.Checked:=True
    else cb.Checked:=False;
end;
//Datos del StringGrid
StringGrid1.Cells[0,0]:=Format('%10s', ['Posición']);
StringGrid1.Cells[1,0]:=Format('%10s', ['Número']);
StringGrid1.Cells[2,0]:=Format('%10s', ['Cuadrado']);
for i:=1 to 100 do begin
    StringGrid1.Cells[0,i]:=Format('%8d', [i]);
    r:=random*1000;
    StringGrid1.Cells[1,i]:=Format('%10.2f', [r]);
    StringGrid1.Cells[2,i]:=Format('%10.2f', [sqr(r)]);
end;
end;

```

Luego se ha procedido a programar los eventos "onClick" de cada uno de los "CheckBox", de manera que si ha sido chequeado se añada la propiedad correspondiente a las opciones del "StringGrid" y en caso contrario se quite la propiedad de las opciones:

```

procedure TForm1.CheckBox1Click(Sender: TObject);
begin
    if CheckBox1.Checked then
        StringGrid1.Options:=StringGrid1.Options+[goFixedVertLine]
    else
        StringGrid1.Options:=StringGrid1.Options-[goFixedVertLine];
end;

```

```

procedure TForm1.CheckBox2Click(Sender: TObject);
begin
    if CheckBox2.Checked then
        StringGrid1.Options:=StringGrid1.Options+[goFixedHorzLine]
    else
        StringGrid1.Options:=StringGrid1.Options-[goFixedHorzLine];
end;

```

```

procedure TForm1.CheckBox3Click(Sender: TObject);
begin
    if CheckBox3.Checked then
        StringGrid1.Options:=StringGrid1.Options+[goVertLine]
    else
        StringGrid1.Options:=StringGrid1.Options-[goVertLine];
end;

```

```

procedure TForm1.CheckBox4Click(Sender: TObject);
begin
    if CheckBox4.Checked then
        StringGrid1.Options:=StringGrid1.Options+[goHorzLine]
    else
        StringGrid1.Options:=StringGrid1.Options-[goHorzLine];
end;

```

```

procedure TForm1.CheckBox5Click(Sender: TObject);
begin
    if CheckBox5.Checked then
        StringGrid1.Options:=StringGrid1.Options+[goRowMoving]
    else
        StringGrid1.Options:=StringGrid1.Options-[goRowMoving];
end;

```

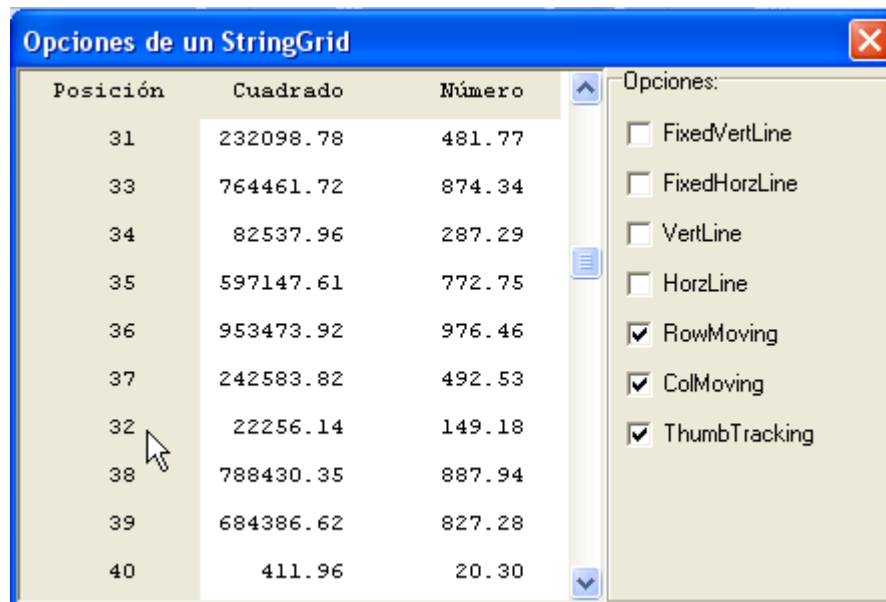
```

procedure TForm1.CheckBox6Click(Sender: TObject);
begin
    if CheckBox6.Checked then
        StringGrid1.Options:=StringGrid1.Options+[goColMoving]
    else
        StringGrid1.Options:=StringGrid1.Options-[goColMoving];
end;

procedure TForm1.CheckBox7Click(Sender: TObject);
begin
    if CheckBox7.Checked then
        StringGrid1.Options:=StringGrid1.Options+[goThumbTracking]
    else
        StringGrid1.Options:=StringGrid1.Options-[goThumbTracking];
end;

```

Cambiando las propiedades en los CheckBoxes, se ve el efecto en el StringGrid, así por ejemplo, en la siguiente pantalla se han eliminado las líneas y se ha habilitado el movimiento de las filas y columnas, es así como se ha cambiado el orden de las columnas y se ha llevado la fila 32 después de la fila 37. Como ya vimos previamente ThumbTracking hace que al mover la barra de desplazamiento se mueva el contenido del StringGrid (si está activado) o no (si no está activado).



21.6 EJERCICIOS

1. Elabore una aplicación similar a la desarrollada en el presente capítulo, pero donde los conjuntos estén formados por hasta 15 países (nombres de los países). El conjunto D debe estar formado por: Los elementos que se encuentran en A o B o C; los elementos que se encuentra en A y B y C; los elementos que se encuentra en A y C o en A y B; los elementos que se encuentra en A y C pero no en B; los elementos que se encuentra en A o B pero no en B y C; los elementos que se encuentra en B o C pero no en A o B; los elementos que se encuentra en A o B y en A o C; los elementos que se encuentra en A o C y En B o C.

2. Elabore una aplicación para modificar algunas de las opciones de un StringGrid. Las opciones que se deberán poder modificar son: `goRowSizing`, `goColSizing`, `goEditing`, `goTabs`, `goRowSelected`, `goAlwaysShowEditor`, `goColMoving`, `goRowMoving` y `goThumbTracking`.

22. REGISTROS PLANOS

En este tema comenzaremos el estudio del tercer tipo de dato estructurado: **Los Registros**. *Un registro es una colección de datos relacionados* y a diferencia de los array cuyos datos tienen que ser todos del mismo tipo, en un registro los datos pueden (y generalmente son) de diferentes tipos. Cada uno de los datos que conforman un registro se conoce con el nombre de **campo**. Los campos pueden ser de cualquier tipo válido en Delphi, inclusive otro registro.

Dos o más registros del mismo tipo conforman una **tabla** y dos o más tablas constituyen una **base de datos**. En los sistemas informáticos las bases de datos se almacenan generalmente en dispositivos externos como cintas o discos donde toman el nombre genérico de **archivos**.

Para declarar una variable registro se emplea la palabra **Record**, y la declaración puede ser efectuada de una de las siguientes formas:

- a) Definir primero un tipo y luego declarar la variable (o variables) de ese tipo:

Type

```
Tipo_Registro = Record
  Nombre_campo1 : tipo_campo1;
  Nombre_campo2 : tipo_campo2;
  ....
  Nombre_campon : tipo_campon;
End;
```

...

Var

```
Nombre_de_las_variables : Tipo_Registro;
```

- b) Declarar directamente la o las variables registro:

Var

```
Nombre_de_las_variables : Record
  Nombre_campo1 : tipo_campo1;
  Nombre_campo2 : tipo_campo2;
  ....
  Nombre_campon : tipo_campon;
End;
```

La primera forma es la que se emplea con más frecuencia y es la única que pueden ser empleada cuando se trabajan con módulos (procedimientos y funciones) que reciben parámetros de tipo registro.

Por ejemplo, para almacenar los datos personales de los alumnos de un colegio se podría definir el siguiente registro:

Type

```
Talumno = Record
  Nombre : String[30];
  Direccion : String[20];
  Telefono : LongInt;
  Edad : Byte;
  Lug_Nac : String[15];
  Pais : String[15];
End;
```

Entonces es posible declarar variables como las siguientes:

Var

```
Nuevo,Egresado,Regular : Talumno;
```

Alternativamente las variables pueden ser declaradas directamente:

```
Nuevo, Egresado, Regular : Record
  Nombre : String[30];
  Direccion : String[20];
  Telefono : LongInt;
  Edad : Byte;
  Lug_Nac : String[15];
  Pais : String[15];
End;
```

Los campos de un registro pueden ser modificados o utilizados en una de las siguientes formas:

- a) De forma explícita: escribiendo el nombre de la variable registro seguida de un punto y el nombre del campo al cual se quiere acceder, por ejemplo el siguiente código asigna datos a la variable nuevo en forma explícita:

```
nuevo.nombre := 'Juan Perez';
nuevo.direccion := 'La Paz 234';
nuevo.telefono := 43254;
nuevo.edad := 17;
nuevo.lug_nac := 'Cochabamba'
nuevo.pais := 'Bolivia';
```

- b) De forma implícita: empleando la palabra reservada **With**. Por ejemplo el anterior código escrito de manera implícita es:

```
With nuevo Do
Begin
  nombre := 'Juan Perez';
  direccion := 'La Paz 234';
  telefono := 43254;
  edad := 17;
  lug_nac := 'Cochabamba'
  pais := 'Bolivia';
End;
```

Los campos de un registro, con la excepción de su escritura, se emplean como cualquier variable de Delphi. Por ejemplo, dada la siguiente declaración:

```
Type
TSueldo = Record
  Item : Word;
  Sal_basico : Single;
  Antiguedad : Byte;
  Bono_Antig : Single;
  Des_Iva : Single;
  Liq_Pag : Single;
End;
Var
  Grupo100, Grupo200 : TSueldo;
```

Se pueden escribir con ellas las siguientes sentencias:

```
With Grupo100 Do
Begin
  Item := 845;
  Sal_Basico := 1200;
  Antiguedad := 10;
  Bono_Antig := Sal_Basico*Antiguedad*5/100;
```

```

Des_Iva := (Sal_Basico+Bono_Antig)*0.13;
Liq_Pag := (Sal_Basico+Bono_Antig)-Des_Iva;
End;
Grupo200 := Grupo100

```

En la última sentencia se asignan todos los valores de la variable grupo100 a la variable grupo200.

22.3 MEMORIA EMPLEADA POR LOS REGISTROS

Una variable registro emplea una cantidad de memoria igual a la suma de la memoria que requieren cada uno de sus campos. Por ejemplo las variables: nuevo, egresado y regular, requieren 89 bytes (31+21+4+1+16+16). Las variables grupo100 y grupo200 requieren 20 bytes (2+4+2+4+4+4).

22.4 EJEMPLOS

22.4.1 Generación aleatoria de registros con nombres completos de personas

Puesto que en los registros frecuentemente se trabaja con datos alfanuméricos como los nombres de personas, libros, direcciones, etc., y dado que para probar una aplicación la introducción de tales datos consumiría demasiado tiempo, como primer ejemplo elaboraremos una aplicación con un modulo que genere aleatoriamente registros y listas de registros con los nombres completos de personas, permitiendo ordenarlos y realizar búsquedas por los campos apellido paterno, materno y nombres.

Problemas similares al presente ya han sido resueltos al trabajar con cadenas en los métodos de ordenación y búsqueda, sólo que en este caso en lugar de generar una cadena con los nombres completos (apellidos y nombre), se generarán registros donde los nombres y los apellidos se encuentren en sus respectivos campos, además, en este caso, aunque no se mostrarán por ocupar demasiado campo, se incluirán listas de nombres y apellidos mucho más extensas, de manera que sea posible generar cientos de miles o millones de registros.

Los módulos que permiten generar de manera aleatoria los apellidos y nombres, tanto masculinos como femeninos han sido implementados en la unidad uNombres:

```

unit uNombres;

interface

function GenerarNombreF:string;
function GenerarNombreM:string;
function GenerarApellido:string;

implementation

function GenerarNombreF:string;
const nombre:array[0..1005] of string[15] = (
  'ABRIL', 'ACACIA', 'ADA', 'ADABELA', 'ADELIA', 'ADELA', 'ADELAIDA', 'ADELINA',
  ...
  'ZAIRA', 'ZARA', 'ZENOBIA', 'ZITA', 'ZOE', 'ZORAIDA', 'ZULEICA');
begin
  result:=nombre[random(1006)];
end;

function GenerarNombreM:string;

```

```

const nombre:array[0..922] of string[15] = (
  'AARÓN', 'AHARÓN', 'ABEL', 'ABELARDO', 'ABRAHAM', 'ABSALÓN', 'ADALBERTO', 'ADÁN',
  ...
  'Yael', 'YAGO', 'YAMIL');
begin
  result:=nombre[random(923)];
end;

function GenerarApellido:string;
const apellido:array[0..5325] of string[15] =(
  'ABAD', 'ABADIA', 'ABAJAS', 'ABALLE', 'ABALOS', 'ABANCENS', 'ABAR', 'ABARCA',
  ...
  'ZURI', 'ZURIARRAIN', 'ZURICA', 'ZURITA', 'ZUYA'
);
begin
  result:=apellido[random(5326)];
end;

end.

```

Donde por razones obvias de espacio no se han copiado los casi 2000 nombres y los más de 5000 apellidos a partir de los cuales se generan los valores.

Los módulos que generan los registros (mediante llamadas a los módulos de la unidad uNombres) se han implementado en la unidad uRegistros:

```

unit uRegistros;

interface

uses uNombres, Grids, SysUtils;

type
  tNombre = record
    Paterno: string[20];
    Materno: string[20];
    Nombres: string[30];
  end;
  tvNombre = array of tNombre;

function GenerarRegistro: tNombre;
function GenerarVRegistro(n:cardinal):tvNombre;
procedure MostrarVRegistro(r:tvNombre; s:tStringGrid);
function LeerVRegistro(s:tStringGrid):tvNombre;
procedure OrdenarPorPaterno(r:tvNombre);
procedure OrdenarPorMaterno(r:tvNombre);
procedure OrdenarPorNombres(r:tvNombre);
function BuscarPaterno(v:string;r:tvNombre):integer;
function BuscarMaterno(v:string;r:tvNombre):integer;
function BuscarNombres(v:string;r:tvNombre):integer;

implementation

function GenerarRegistro: tNombre;
var s:string; femenino,dosnombres:boolean;
begin
  femenino:= Random(100)>=49;
  dosnombres:= Random(100)>=49;
  result.Paterno:=GenerarApellido;

```



```
result.Materno:=GenerarApellido;
if femenino then begin
  s:=GenerarNombreF;
  if dosnombres then
    s:=s+ ' '+GenerarNombreF;
end else begin
  s:=GenerarNombreM;
  if dosnombres then
    s:=s+ ' '+GenerarNombreM;
end;
result.Nombres:=s;
end;

function GenerarVRegistro(n:cardinal): tvNombre;
var i: cardinal;
begin
  if n=0 then exit;
  SetLength(result,n);
  for i:=0 to n-1 do
    result[i]:=GenerarRegistro;
end;

procedure MostrarVRegistro(r:tvNombre; s:tStringGrid);
var i,n: cardinal;
begin
  n:=Length(r);
  s.RowCount:=n+1;
  for i:=1 to n do begin
    s.Cells[0,i]:=Format('%14d',[i]);
    s.Cells[1,i]:=r[i-1].Paterno;
    s.Cells[2,i]:=r[i-1].Materno;
    s.Cells[3,i]:=r[i-1].Nombres;
  end;
end;

function LeerVRegistro(s:tStringGrid):tvNombre;
var v:tvNombre; i,n:integer;
begin
  n:=s.RowCount-1;
  SetLength(v,n);
  for i:=0 to n-1 do begin
    v[i].Paterno:=s.Cells[1,i+1];
    v[i].Materno:=s.Cells[2,i+1];
    v[i].Nombres:=s.Cells[3,i+1];
  end;
  result:=v;
end;

procedure OrdenarPorPaterno(r:tvNombre);
var pivote:string; aux:tNombre; i,j:integer;
procedure Quickr(p,u:integer);
begin
  i:=p;j:=u;pivote:=r[i].Paterno;
  repeat
    while r[i].Paterno<Pivote do inc(i);
    while r[j].Paterno>pivote do dec(j);
    if i<=j then begin
      if i<>j then begin
```

```

        aux:=r[i];
        r[i]:=r[j];
        r[j]:=aux;
    end;
    inc(i);dec(j)
end;
until i>j;
if j>p then Quickr(p,j);
if i<u then Quickr(i,u);
end;
begin
    Quickr(0,High(r));
end;

procedure OrdenarPorMaterno(r:tvNombre);
var pivote:string; aux:tNombre; i,j:integer;
procedure Quickr(p,u:integer);
begin
    i:=p;j:=u;pivote:=r[i].Materno;
    repeat
        while r[i].Materno<Pivote do inc(i);
        while r[j].Materno>pivote do dec(j);
        if i<=j then begin
            if i<>j then begin
                aux:=r[i];
                r[i]:=r[j];
                r[j]:=aux;
            end;
            inc(i);dec(j)
        end;
    until i>j;
    if j>p then Quickr(p,j);
    if i<u then Quickr(i,u);
end;
begin
    Quickr(0,High(r));
end;

procedure OrdenarPorNombres(r:tvNombre);
var pivote:string; aux:tNombre; i,j:integer;
procedure Quickr(p,u:integer);
begin
    i:=p;j:=u;pivote:=r[i].Nombres;
    repeat
        while r[i].Nombres<Pivote do inc(i);
        while r[j].Nombres>pivote do dec(j);
        if i<=j then begin
            if i<>j then begin
                aux:=r[i];
                r[i]:=r[j];
                r[j]:=aux;
            end;
            inc(i);dec(j)
        end;
    until i>j;
    if j>p then Quickr(p,j);
    if i<u then Quickr(i,u);
end;

```

```

begin
    Quickcr(0,High(r));
end;

function BuscarPaterno(v:string;r:tvNombre):integer;
var i,j,k,nc:integer;
begin
    i:=0; j:=High(r); nc:=Length(v);
    repeat
        k:=(i+j) div 2;
        if Copy(r[k].Paterno,1,nc)=v then begin
            while Copy(r[k-1].Paterno,1,nc)=v do dec(k);
            result:=k; exit;
        end;
        if v>r[k].Paterno then i:=k+1 else j:=k-1;
    until i>j;
    result:=-1;
end;

function BuscarMaterno(v:string; r:tvNombre):integer;
var i,j,k,nc:integer;
begin
    i:=0; j:=High(r); nc:=Length(v);
    repeat
        k:=(i+j) div 2;
        if Copy(r[k].Materno,1,nc)=v then begin
            while Copy(r[k-1].Materno,1,nc)=v do dec(k);
            result:=k; exit;
        end;
        if v>r[k].Materno then i:=k+1 else j:=k-1;
    until i>j;
    result:=-1;
end;

function BuscarNombres(v:string; r:tvNombre):integer;
var i,j,k,nc:integer;
begin
    i:=0; j:=High(r); nc:=Length(v);
    repeat
        k:=(i+j) div 2;
        if Copy(r[k].Nombres,1,nc)=v then begin
            while Copy(r[k-1].Nombres,1,nc)=v do dec(k);
            result:=k; exit;
        end;
        if v>r[k].Nombres then i:=k+1 else j:=k-1;
    until i>j;
    result:=-1;
end;

initialization

    randomize;

end.

```

Donde como se puede ver los registros son mostrados y leído en un StringGrid, que se supone cuenta con una fila de encabezados y una columna con los números de registros. Los registros son ordenados con el método

Quick-Sort y los tres módulos de ordenación sólo difieren entre sí por el campo que se emplea en las operaciones relaciones. De manera similar los módulos de búsqueda, que emplean el método de búsqueda binaria, sólo difieren entre sí por el campo que se comparación que se emplea en las operaciones relacionales.

Observe también que en esta unidad se ha empleado el sector de inicialización para iniciar la semilla del generador de números aleatorios.

En ejecución la aplicación tiene la apariencia que se muestra en la siguiente figura:

Nº	Paterno	Materno	Nombres
3545	VITORES	PELON	ASUNTA ALTAGRACIA
3546	ESTRAMIANA	PEON	LAURENTINO MELQUISEDI
3547	GABILONDO	PERAL	NÉLIDA
3548	GREGORIO	PERAL	TEODOMIRO JUSTO
3549	TARAN	PERALES	EMILY
3550	SEBER	PERALES	MILENA
3551	SIERRA	PERALES	EDUARDO MARCELO
3552	HERAS	PERALTA	AMANDA PRISCILA
3553	GOIRI	PERALTA	SEAN ABEL
3554	REGUERO	PERCAZ	ESTÍBALIZ JOSUÉ
3555	DIAZ	PERDIZ	QUINTÍN
3556	VIDARTE	PERDIZ	PETRONA
3557	ESPINAREDA	PEREA	PACIENTE GILBERTO
3558	AJAMIL	PERECEIRA	EVARISTO

Y como se puede observar la misma consta de dos ComboBox, un Edit, un BitBtn y un StringGrid (además de las etiquetas).

Los eventos programados en esta aplicación, e incluidos en la unidad fNombres son los siguientes:

```

procedure TForm1.FormCreate(Sender: TObject);
var i:cardinal; ancho:integer;
begin
    ComboBox1.Clear;
    for i:=1 to 10000 do
        ComboBox1.Items.Append(IntToStr(i));
    ComboBox1.ItemIndex:=0;
    StringGrid1.ColWidths[0]:=60;
    StringGrid1.ColWidths[1]:=120;
    StringGrid1.ColWidths[2]:=120;

```

```
StringGrid1.ColWidths[3]:=140;
StringGrid1.Cells[0,0]:=Format('%10s', ['N°']);
StringGrid1.Cells[1,0]:=Format('%16s', ['Paterno']);
StringGrid1.Cells[2,0]:=Format('%16s', ['Materno']);
StringGrid1.Cells[3,0]:=Format('%20s', ['Nombres']);
ancho:=0;
for i:=0 to 3 do
    ancho:=ancho+StringGrid1.ColWidths[i];
StringGrid1.Options:=StringGrid1.Options+
    [goThumbTracking,goRowSelect];
BitBtn1.Cursor:= crHandPoint;
Form1.BorderStyle:=bsDialog;
Form1.ClientWidth:=ancho+25;
end;

procedure TForm1.ComboBox1Change(Sender: TObject);
var r:tvNombre;
begin
    r:=GenerarVRegistro(ComboBox1.ItemIndex+1);
    case ComboBox2.ItemIndex of
        0: OrdenarPorPaterno(r);
        1: OrdenarPorMaterno(r);
    else
        OrdenarPorNombres(r);
    end;
    MostrarvRegistro(r,StringGrid1);
    Finalize(r);
end;

procedure TForm1.ComboBox2Change(Sender: TObject);
var x:tvNombre;
begin
    x:=LeerVRegistro(StringGrid1);
    case ComboBox2.ItemIndex of
        0: OrdenarPorPaterno(x);
        1: OrdenarPorMaterno(x);
        2: OrdenarPorNombres(x);
    end;
    MostrarVRegistro(x,StringGrid1);
    Finalize(x);
end;

procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);
begin
    key:=AnsiUpperCase(string(Key))[1];
    if not (Key in [#8,'A'..'Z','Ñ',' ']) then
        begin Beep; Abort; end;
end;

procedure TForm1.BitBtn1Click(Sender: TObject);
var x:tvNombre; v:string; k:integer;
begin
    x:=LeerVRegistro(StringGrid1);
    v:=Edit1.Text;
    case ComboBox2.ItemIndex of
        0: k:=BuscarPaterno(v,x);
        1: k:=BuscarMaterno(v,x);
    else
```

```

    k:=BuscarNombres(v,x);
end;
finalize(x);
StringGrid1.Row:=k+1;
StringGrid1.SetFocus;
end;

end.

```

Note que los nombres se generan y ordena en el evento "onChange" del ComboBox1, en función del orden elegido en el ComboBox2.

La búsqueda se ha programado en el evento "onClick" del BitBtn1 y al igual que la generación, el campo buscado depende del valor elegido en el ComboBox2.

En el ComboBox2, se ordenan los registros, en función al campo elegido.

22.4.2 Generación aleatoria de registros con direcciones de domicilios

Como segundo ejemplo elaboraremos una aplicación que genere aleatoriamente registros y listas de registros con direcciones de domicilios, los ordene alfabéticamente y permita buscar una dirección en los mismos.

Como en la anterior aplicación primero creamos en una unidad la función que permite generar aleatoriamente nombres de calles (en este empleando un array con sólo 60 nombres):

```
unit uDirecciones;
```

```
interface
```

```
    function GenerarCalle:string;
```

```
implementation
```

```

function GenerarCalle:string;
const Calle: array [0..59] of string[25] = (
    'Azurduy','Zudañes','San Alberto','Ravelo','Junín',
    'Potosí','Oruro','Padilla','La Paz','Bustillos',
    'Nicolás Ortiz','Perez','Dalence','Ayacucho','Calvo',
    'Arenales','Loa','Colón','Grau','Bolivar',
    'Aniceto Arce','Ballivián','Daniel Campos','Regimiento Campos',
    'Camargo',
    'Av. Jaime Mendoza','Daniel Bustamante','México','Canada','Guatemala',
    'Av. Ostría Gutierrez','Pasaje Rouma','Pilinco','España','Estudiantes',
    'Av. Hernando Siles','Perú','Colombia','Destacamento 111','Puerto Rico',
    'Vitorino Vega','San Martín','Serrano','Guillermo Loayza','Suipacha',
    'Av. German Mendoza','Antofagasta','Germán Bush','Calama','Arani',
    'Regimiento Carabineros','Adolfo Vilar','Ignacio Warmes','Tarija',
    'Avaroa',
    'Nataniel Aguirre','Wiracocha','Cotagaita','Rafael Gómez Reyes',
    '23 de Marzo'
);
begin
    result:=Calle[Random(60)];
end;

```

```
end.
```

El registro consta de dos campos: uno para el nombre de la calle y otro para el número. Los tipos de datos, variables y módulos creados para trabajar con este registro se han implementado en la unidad "uRDirecciones":

```

unit uRDirecciones;

interface

uses Grids, SysUtils, MaskUtils;

type
    tDireccion = record
        Calle: string[25];
        Numero: word;
    end;
    tvDireccion = array of tDireccion;

function GenerarDireccion:tDireccion;
function GenerarvDireccion(n:Cardinal):tvDireccion;
function LeerDirecciones(s:tStringGrid):tvDireccion;
procedure MostrarDirecciones(v:tvDireccion;s:tStringGrid);
procedure OrdenarDirecciones(x:tvDireccion);
function BuscarDireccion(d:tDireccion;x:tvDireccion):integer;

implementation

uses uDirecciones;

function GenerarDireccion:tDireccion;
begin
    with result do begin
        Calle:= GenerarCalle;
        Numero:= Random(9999)+1;
    end
end;

function GenerarvDireccion(n:Cardinal):tvDireccion;
var i: Cardinal;
begin
    SetLength(result,n);
    for i:=0 to n-1 do
        result[i]:=GenerarDireccion;
    end;

function LeerDirecciones(s:tStringGrid):tvDireccion;
var i,n:integer; x:tvDireccion;
begin
    n:=s.RowCount-1;
    SetLength(x,n);
    for i:=0 to n-1 do begin
        x[i].Calle:=s.Cells[1,i+1];
        x[i].Numero:=StrToInt(s.Cells[2,i+1]);
    end;
    result:=x;
end;

procedure MostrarDirecciones(v:tvDireccion;s:tStringGrid);
var i,n:integer;
begin

```

```

n:=Length(v);
s.RowCount:=n+1;
for i:=0 to n-1 do begin
    s.Cells[0,i+1]:=Format('%12d',[i]);
    s.Cells[1,i+1]:=v[i].Calle;
    s.Cells[2,i+1]:=IntToStr(v[i].Numero);
end;
end;

procedure OrdenarDirecciones(x:tvDireccion);
var i,j:integer; aux:tDireccion; pivote:string;
function Direccion(d:tDireccion):string;
begin
    result:=d.Calle;
    case d.Numero of
        0..9:result:=result+' '+IntToStr(d.Numero);
        10..99:result:=result+' '+IntToStr(d.Numero);
        100..999:result:=result+' '+IntToStr(d.Numero);
        1000..9999:result:=result+' '+IntToStr(d.Numero);
    end;
end;
procedure Quickr(p,u:integer);
begin
    i:=p; j:=u;
    pivote:=Direccion(x[i]);
    repeat
        while Direccion(x[i])<pivote do inc(i);
        while Direccion(x[j])>pivote do dec(j);
        if i<=j then begin
            if i<>j then begin
                aux:=x[i]; x[i]:=x[j]; x[j]:=aux;
            end;
            inc(i); dec(j);
        end;
    until i>j;
    if j>p then Quickr(p,j);
    if i<u then Quickr(i,u);
end;
begin
    Quickr(0,High(x));
end;

function BuscarDireccion(d:tDireccion;x:tvDireccion):integer;
var i,j,k: integer; s: string;
begin
    i:=0; j:=Length(x);
    repeat
        k:=(i+j) div 2;
        if x[k].Calle=d.Calle then begin
            while (x[k-1].Calle=d.Calle) and (k>0) do dec(k);
            i:=k;
            while (x[k+1].Calle=d.Calle) do inc(k);
            j:=k; k:=i;
            while i<=j do begin
                s:=IntToStr(d.Numero);
                if s=Copy(IntToStr(x[i].Numero),1,Length(s)) then
                    begin k:=i; break; end;
                inc(i);
            end;
        end;
    until i>j;
    return k;
end;

```



```

    end;
    result:=k;
    exit;
  end;
  if x[k].Calle>d.Calle then j:=k-1 else i:=k+1;
until i>j;
result:=-1;
end;

end.

```

Aun cuando en este caso el registro es más simple que el del anterior ejemplo, los campos de diferentes tipos: uno string y otro word, complican los métodos de ordenación y de búsqueda. Por esta razón con frecuencia se suelen tratar todos los datos como cadenas (strings).

La aplicación en ejecución tiene la siguiente apariencia:

Nº Registro	Direccion	Nº
813	Rafael Gómez Reyes	6684
814	Rafael Gómez Reyes	7528
815	Rafael Gómez Reyes	7838
816	Rafael Gómez Reyes	9862
817	Ravelo	963
818	Ravelo	1014
819	Ravelo	1402
820	Ravelo	1810
821	Ravelo	1939
822	Ravelo	2422
823	Ravelo	2838
824	Ravelo	2933
825	Ravelo	3444
826	Ravelo	4742
827	Ravelo	5077

Los eventos programados en esta aplicación son los siguientes:

```

procedure TForm1.FormCreate(Sender: TObject);
var i:integer;
begin
  Form1.Caption:='Registros: Lista de direcciones';
  Form1.BorderStyle:=bsDialog;
  Form1.Position:=poScreenCenter;
  Form1.ShowHint:=True;
  Form1.KeyPreview:=True;

```

```

Panel1.Caption:='';
StringGrid1.ColCount:=3;
StringGrid1.RowCount:=2;
StringGrid1.Cells[0,0]:=Format('%14s',['N° Registro']);
StringGrid1.Cells[1,0]:=Format('%45s',['Direccion']);
StringGrid1.Cells[2,0]:=Format('%10s',['N°']);
StringGrid1.ColWidths[0]:=80;
StringGrid1.ColWidths[1]:=250;
StringGrid1.ColWidths[2]:=60;
StringGrid1.Options:=StringGrid1.Options+
  [goRowSelect,goThumbTracking];
Form1.ClientWidth:=StringGrid1.ColWidths[0]+
  StringGrid1.ColWidths[1]+StringGrid1.ColWidths[2]+25;
ComboBox1.Style:=csDropDownList;
ComboBox1.Items.Clear;
for i:=1 to 1000 do
  ComboBox1.Items.Append(IntToStr(i));
ComboBox1.ItemIndex:=0;
Edit1.Text:='';
Edit2.Text:='';
SpeedButton1.NumGlyphs:=2;
SpeedButton1.Glyph.LoadFromFile('C:\Archivos de Programa\'+
  'Archivos Comunes\Borland Shared\Images\Buttons\Find.bmp');
SpeedButton1.Hint:='Buscar';
SpeedButton1.Cursor:=crHandPoint;
end;

procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);
begin
  case AnsiUpperCase(Key)[1] of
    #8,'0'..'9',' ','A'..'Z','Ñ':;
  else Beep; Abort;
  end;
end;

procedure TForm1.ComboBox1Change(Sender: TObject);
var x:tvDireccion; n:integer;
begin
  n:=ComboBox1.ItemIndex+1;
  x:=GenerarvDireccion(n);
  OrdenarDirecciones(x);
  MostrarDirecciones(x,StringGrid1);
  Finalize(x);
end;

procedure TForm1.SpeedButton1Click(Sender: TObject);
var x: tvDireccion; d:tDireccion; k:integer;
begin
  x:=LeerDirecciones(StringGrid1);
  d.Calle:=Edit1.Text;
  d.Numero:=StrToInt(Edit2.Text);
  k:=BuscarDireccion(d,x);
  if k=-1 then
    ShowMessage('Dirección no encontrada')
  else
    StringGrid1.Row:=k+1;
end;

```

```

procedure TForm1.FormKeyPress(Sender: TObject; var Key: Char);
begin
    if Key=#13 then SpeedButton1.Click;
end;

procedure TForm1.Edit2KeyPress(Sender: TObject; var Key: Char);
begin
    if not (key in [#8,'0'..'9']) then begin Beep; Abort; end;
end;

end.

```

Observe que en el Edit1 (nombre de la calle) se permite introducir no sólo letras, sino también números, debido a que los nombres de algunas calles constan de números.

Observe también que en este caso se ha empleado un SpeedButton y como el mismo no tiene la propiedad "Default" (para que sea activado al pulsar la tecla Enter), se ha activado la propiedad "KeyPreview" de la forma, para que cualquier evento del teclado sea procesado previamente por la forma, entonces se ha programado el evento "onKeyPress" de la forma de manera que si la tecla pulsada es la tecla Enter (o Intro, tecla #13) se llama al método "Click" del SpeedButton, es decir se emula el evento "onClick" en el mismo.

22.4.3 Generación aleatoria de registros con fechas de nacimiento

Como tercer ejemplo elaboraremos una aplicación que genere aleatoriamente registros y listas de registros con fechas de nacimiento comprendidas entre dos fechas dadas, permitiendo ordenarlas cronológicamente de forma ascendente o descendente y realizar búsquedas en las mismas.

En este caso no es necesario crear una función adicional, como en los dos ejemplos previos, porque las fechas son números y como tales pueden ser generadas directamente con la función Random.

En su forma de registro una fecha está conformada por tres campos: el día, el mes y el año. Los tipos, constantes y funciones creadas para esta aplicación son los siguientes:

```

unit urFechas;

interface

uses Grids, SysUtils;

type
    tFecha = record
        dia: 1..31;
        mes: 1..12;
        anio: word;
    end;

    tvFecha= array of tFecha;

    const meses : array [1..12] of string[10] =
        ('Enero', 'Febrero', 'Marzo', 'Abril', 'Mayo', 'Junio',
         'Julio', 'Agosto', 'Septiembre', 'Octubre', 'Noviembre',
         'Diciembre');

    function GenerarFecha: tFecha;

```

```

function GenerarvFecha(n:cardinal):tvFecha;
function LeerFechas(s:tStringGrid):tvFecha;
procedure MostrarFechas(x:tvFecha;s:tStringGrid);
function MesAByte(mes:string):byte;
procedure OrdenarFechasA(x:tvFecha);
procedure OrdenarFechasD(x:tvFecha);
function FechaACardinal(f:tvFecha):cardinal;
function BuscarFechaA(f:tvFecha;x:tvFecha):integer;
function BuscarFechaD(f:tvFecha;x:tvFecha):integer;

```

implementation

```

function GenerarFecha:tFecha;
begin
  result.mes:=Random(12)+1;
  case result.mes of
    1,3,5,7,8,10,12: result.dia:=Random(31)+1;
    4,6,9,11: result.dia:=Random(30)+1;
    2: result.dia:=Random(28)+1;
  end;
  result.anio:=Random(49)+1961;
end;

function GenerarvFecha(n:cardinal):tvFecha;
var i:cardinal;
begin
  SetLength(result,n);
  for i:=0 to n-1 do
    result[i]:=GenerarFecha;
  end;

function LeerFechas(s:tStringGrid):tvFecha;
var i,n:integer;
begin
  n:=s.RowCount-1;
  SetLength(result,n);
  for i:=0 to n-1 do begin
    result[i].dia:=StrToInt(s.Cells[1,i+1]);
    result[i].mes:=MesAByte(s.Cells[2,i+1]);
    result[i].anio:=StrToInt(s.Cells[3,i+1]);
  end;
end;

procedure MostrarFechas(x:tvFecha;s:tStringGrid);
var i,n:integer;
begin
  n:=Length(x);
  s.RowCount:=n+1;
  for i:=0 to n-1 do begin
    s.Cells[0,i+1]:=Format('%11d',[i+1]);
    s.Cells[1,i+1]:=Format('%8d',[x[i].dia]);
    s.Cells[2,i+1]:=Meses[x[i].mes];
    s.Cells[3,i+1]:=Format('%8d',[x[i].anio]);
  end;
end;

function MesAByte(mes:string):byte;
var i:byte;

```

```

begin
  result:=0;
  for i:=1 to 12 do
    if meses[i]=mes then result:=i;
  end;

procedure OrdenarFechasA(x:tvFecha);
var i,j:integer; pivote:cardinal; aux:tFecha;
procedure Quickr(p,u:integer);
begin
  i:=p; j:=u; pivote:=FechaACardinal(x[i]);
  repeat
    while FechaACardinal(x[i])<pivote do inc(i);
    while FechaACardinal(x[j])>pivote do dec(j);
    if i<=j then begin
      if i<>j then begin
        aux:=x[i]; x[i]:=x[j]; x[j]:=aux;
      end;
      inc(i); dec(j);
    end;
  until i>j;
  if j>p then Quickr(p,j);
  if i<u then Quickr(i,u);
end;
begin
  Quickr(0,High(x));
end;

procedure OrdenarFechasD(x:tvFecha);
var i,j:integer; pivote:cardinal; aux:tFecha;
procedure Quickr(p,u:integer);
begin
  i:=p; j:=u; pivote:=FechaACardinal(x[i]);
  repeat
    while FechaACardinal(x[i])>pivote do inc(i);
    while FechaACardinal(x[j])<pivote do dec(j);
    if i<=j then begin
      if i<>j then begin
        aux:=x[i]; x[i]:=x[j]; x[j]:=aux;
      end;
      inc(i); dec(j);
    end;
  until i>j;
  if j>p then Quickr(p,j);
  if i<u then Quickr(i,u);
end;
begin
  Quickr(0,High(x));
end;

function FechaACardinal(f:tFecha):cardinal;
begin
  result:=f.anio*10000+f.mes*100+f.dia;
end;

function BuscarFechaA(f:tFecha;x:tvFecha):integer;
var v:cardinal; i,j,k:integer;
begin

```

```

i:=0; j:=High(x); v:=FechaACardinal(f);
repeat
  k:=(i+j) div 2;
  if FechaACardinal(x[k])=v then begin
    while FechaACardinal(x[k-1])=v do dec(k);
    result:=k; exit;
  end;
  if FechaACardinal(x[k])>v then j:=k-1 else i:=k+1;
until i>j;
result:=-1;
end;

function BuscarFechaD(f:tFecha;x:tvFecha):integer;
var v:cardinal; i,j,k:integer;
begin
  i:=0; j:=High(x); v:=FechaACardinal(f);
  repeat
    k:=(i+j) div 2;
    if FechaACardinal(x[k])=v then begin
      while FechaACardinal(x[k-1])=v do dec(k);
      result:=k; exit;
    end;
    if FechaACardinal(x[k])<v then j:=k-1 else i:=k+1;
  until i>j;
  result:=-1;
end;

end.

```

Las fechas son generadas entre los años 1960 y 2009. Como se pudo observar para generar los días es necesario tomar en cuenta el mes, porque existen meses que pueden tener como máximo 31 días, otros 30 y el mes de febrero 28.

En esta aplicación los meses se muestran por su nombre, no por su número y para ello se ha creado el vector constante "meses".

Para ordenar las fechas, así como para buscar una fecha, se convierten los datos del registro en un número entero (con la función FechaACardinal) en el orden año+mes+día, debido a que no es posible comparar los registros como tales. Aun cuando conceptualmente una fecha consta realmente de los tres campos empleados en esta aplicación, en la práctica y para evitar los problemas mencionados se suele guardar la fecha (y la hora) como un solo dato de tipo numérico, generalmente real, o en su caso como una cadena de caracteres.

En ejecución la aplicación tiene la apariencia que se muestra en la figura de la siguiente página. Los eventos programados en la misma son los siguientes:

```

procedure TForm1.FormCreate(Sender: TObject);
var i, ancho:integer;
begin
  Form1.Caption:='Registros: Lista de Fechas';
  Form1.BorderStyle:=bsDialog;
  Form1.Position:=poScreenCenter;
  Panel1.Caption:='';
  ComboBox1.Items.Clear;
  for i:=1 to 10000 do
    ComboBox1.Items.Append(IntToStr(i));
  ComboBox1.ItemIndex:=0;

```

Registros: Lista de Fechas

Nº Registros: 1900

Buscar Fecha: 11/09/2003

Orden

☐ Ascendente

☒ Descendente

Nº Registro	Día	Mes	Año
232	11	Enero	2004
233	5	Enero	2004
234	27	Diciembre	2003
235	22	Diciembre	2003
236	21	Diciembre	2003
237	15	Diciembre	2003
238	16	Noviembre	2003
239	14	Noviembre	2003
240	4	Noviembre	2003
241	18	Octubre	2003
242	10	Octubre	2003
243	24	Septiembre	2003
244	11	Septiembre	2003
245	3	Septiembre	2003

```

MaskEdit1.EditMask:='00/00/0000;0;_';
MaskEdit1.Text:='01012001';
RadioGroup1.Caption:='Orden';
RadioGroup1.Items.Clear;
RadioGroup1.Columns:=1;
RadioGroup1.Items.Append('&Ascendente');
RadioGroup1.Items.Append('&Descendente');
RadioGroup1.OnClick:=nil;
RadioGroup1.ItemIndex:=0;
RadioGroup1.OnClick:=RadioGroup1Click;
StringGrid1.ColCount:=4;
StringGrid1.RowCount:=2;
StringGrid1.Cells[0,0]:=Format('%13s',['Nº Registro']);
StringGrid1.Cells[1,0]:=Format('%8s',['Día']);
StringGrid1.Cells[2,0]:=Format('%16s',['Mes']);
StringGrid1.Cells[3,0]:=Format('%8s',['Año']);
StringGrid1.ColWidths[0]:=70;
StringGrid1.ColWidths[1]:=50;
StringGrid1.ColWidths[2]:=100;
StringGrid1.ColWidths[3]:=50;
StringGrid1.Options:=StringGrid1.Options+
    [goRowSelect,goThumbTracking];
ancho:=0;
for i:=0 to 3 do
    ancho:=ancho+StringGrid1.ColWidths[i];
Form1.ClientWidth:=ancho+25;

```

```

end;

procedure TForm1.ComboBox1Change(Sender: TObject);
var x:tvFecha; n:integer;
begin
  n:=ComboBox1.ItemIndex+1;
  x:=GenerarvFecha(n);
  case RadioGroup1.ItemIndex of
    0: OrdenarFechasA(x);
    else OrdenarFechasD(x);
  end;
  MostrarFechas(x,StringGrid1);
  Finalize(x);
end;

procedure TForm1.RadioGroup1Click(Sender: TObject);
var x:tvFecha;
begin
  x:=LeerFechas(StringGrid1);
  if RadioGroup1.ItemIndex=0 then
    OrdenarFechasA(x)
  else
    OrdenarFechasD(x);
  MostrarFechas(x,StringGrid1);
  Finalize(x);
end;

procedure TForm1.MaskEdit1KeyPress(Sender: TObject; var Key: Char);
var f:tFecha; x:tvFecha; k:integer;
begin
  if Key=#13 then begin
    f.dia:=StrToInt(Copy(MaskEdit1.Text,1,2));
    f.mes:=StrToInt(Copy(MaskEdit1.Text,3,2));
    f.anio:=StrToInt(Copy(MaskEdit1.Text,5,4));
    x:=LeerFechas(StringGrid1);
    if RadioGroup1.ItemIndex=0 then
      k:=BuscarFechaA(f,x)
    else
      k:=BuscarFechaD(f,x);
    if k<0 then
      ShowMessage('No existe la fecha buscada')
    else
      StringGrid1.Row:=k+1;
      Finalize(x);
    end;
  end;
end;

end.

```

En todos los ejemplos presentados, cada vez que se realiza una operación con la lista de registros se lee la misma del StringGrid y si es necesario, una vez hechas las operaciones se vuelve a mostrar la lista en el StringGrid, eliminando finalmente la lista leída. Aun cuando ello permite mantener los módulos independientes (uno de los fundamentos de la programación estructurada), es bastante ineficiente, por lo que en la práctica se suele declarar una variable en el sector de implementación, de esa manera dicha variable es global a todos los procedimientos de la unidad (es decir es conocida por todos ellos), pero no es conocida por ningún procedimiento o programa externo, lo que permite mantener en cierto modo la modularidad.

22.5 EJERCICIOS

1. Vuelva a elaborar el ejemplo 1 pero empleando punteros en lugar de vectores dinámicos.
2. Vuelva a elaborar el ejemplo 2 pero empleando punteros en lugar de vectores dinámicos.
3. Vuelva a elaborar el ejemplo 3 pero empleando punteros en lugar de vectores dinámicos.
4. Elabore una aplicación que genere aleatoriamente, ordene y realice búsquedas de registros con direcciones de correos electrónicos.
5. Elabore una aplicación que genere aleatoriamente, ordene y realice búsqueda de registros de los artículos de una tienda de ropa: Tipo de vestimenta (pantalón, falda, chamarra, etc.), industria (japonesa, boliviana, brasilera, etc.), marca (Lewis, Lacoste, etc.), talla (pequeña, mediana, grande, etc.), color (azul marino, verde claro, etc.), precio (en bolivianos) y existencia (cuántos ejemplares existen en almacenes).
6. Elabore una aplicación que permita modificar, añadir y borrar la siguiente información correspondiente a una biblioteca: Código de libro; Título de la obra; Autor (o autores); Editorial; País; Año de Publicación; Número de Edición y Categoría (Social, Técnica, Religiosa, Medicina, Autosuperación, Entretenimiento, etc.). La aplicación debe mostrar la información ordenada por Título, por Autor por Editorial y por Año de publicación.
7. Elabore una aplicación que permita modificar, añadir y borrar la siguiente información correspondiente a una ferretería: Código del artículo; Nombre del artículo; Categoría (jardinería, electricidad, carpintería, construcción, etc.); Precio; Cantidad disponible; Unidad de medida (pieza, Kg, Litros, bolsa, metros, etc.); Proveedor; Marca y País. La aplicación debe mostrar la información ordenada por nombre, categoría, país y marca.

23. REGISTROS JERÁRQUICOS

Si uno o más de los campos de un registro es a su vez un registro, el registro recibe el nombre de *registro jerárquico*.

La siguiente declaración es un ejemplo de este tipo de registros:

Type

```
Tfecha = Record
  dia : 1..31;
  mes : 1..12;
  anio : Integer;
End;
Templeado = Record
  item : word;
  nombre : String[30];
  f_nacimiento : tfecha;
  f_contrato : tfecha;
End;
```

Var

```
empleado : empleado;
```

Para modificar o emplear los valores de los *subcampos* de un campo registro se puede emplear la nomenclatura explícita o implícita. Así por ejemplo en forma implícita se podrían realizar las siguientes asignaciones:

```
empleado.f_nacimiento.dia := 21;
empleado.f_nacimiento.mes := 3;
empleado.f_nacimiento.anio := 1959;
empleado.f_contrato.dia := 10;
empleado.f_contrato.mes := 1;
empleado.f_contrato.anio := 1980;
```

Y en forma *implícita*, para los subcampos del campo `f_nacimiento` sería:

```
With empleado.f_nacimiento Do
Begin
  dia:=21; mes:=3; anio:=1959;
End;
```

Que igualmente puede ser escrito de la siguiente forma:

```
With empleado Do
  With f_nacimiento Do
  Begin
    dia:=21; mes:=3; anio:=1959;
  End;
```

O también:

```
With empleado, f_nacimiento Do
Begin
  dia:=21; mes:=3; anio:=1959;
End;
```

La última forma, puede ser empleada no solo con registros jerárquicos, sino con registros de cualquier tipo cuando se trabaja con dos o más registros a la vez. En este caso, si dos o más registros tienen un campo con el mismo nombre, el valor que es válido dentro de la sentencia *with* es el correspondiente al último registro incluido en la sentencia. Para acceder a los campos de los otros registros se debe emplear la notación explícita.

Por ejemplo en la siguiente sentencia:

```

with empleado, f_nacimiento, f_contrato do
begin
  Edit1.Text := IntToStr(dia);
  Edit2.Text := IntToStr(mes);
  Edit3.Text := IntToStr(anio);
end;

```

Los valores que se asignan a los cuadros de edición son los correspondientes al campo `f_contrato`. Para asignar los valores del campo `f_nacimiento` sería necesario emplear notación explícita como la siguiente:

```
Edit4.Text := IntToStr(empleado.f_nacimiento.dia);
```

23.3 EJEMPLOS

23.3.1 Directorio

Como primer ejemplo elaboraremos un directorio con la siguiente información: Nombre completo, Fecha de nacimiento, Dirección, Celular. La aplicación debe permitir generar datos al azar (para realizar pruebas), añadir nuevos registros, eliminar registros existentes, ordenar los registros por los campos nombre completo y fecha de nacimiento, permitiendo buscar un nombre o una fecha de nacimiento dadas.

Para esta aplicación la información será organizada en forma de registros jerárquicos, donde los campos para el nombre completo, fecha de nacimiento y dirección serán a su vez registros. Aunque procederemos de esa forma en esta aplicación, se aclara que en la práctica no es muy eficiente organizar los registros de esa forma por la sobrecarga que se ocasiona al momento de buscar y organizar este tipo de archivos compuestos.

Como ejemplo elaboraremos una aplicación que permita modificar, añadir y borrar la siguiente información correspondiente a un directorio: Nombre completo: Apellido paterno, materno y nombres; Dirección del domicilio; Dirección del Trabajo; Teléfono del domicilio; Teléfono del trabajo; Celular; email y Fecha de Nacimiento. La aplicación debe poder mostrar la información ordenada por apellido paterno, por apellido materno, por nombres, por nombre completo o por fecha de nacimiento.

La aplicación tiene la apariencia que se muestra en la figura de la siguiente página.

En esta aplicación, los campos Teléfono Domicilio, Teléfono Trabajo, Celular y Fecha Nacimiento se introducen en componentes *MaskEdit*. Los botones de navegación son botones de un *ToolBar*. En este caso, como se puede observar se requieren ocho botones. Estos botones tienen las siguientes funciones: *Primer registro*, *Anterior registro*, *Siguiente registro*, *Ultimo registro*, *Nuevo registro*, *Eliminar registro*, *Confirmar cambios* y *Cancelar*, como ayuda para el usuario, la función que realiza cada uno de estos botones debe ser colocada en su propiedad *Hint*, colocando también la propiedad *ShowHint* en *True*.

Los botones de la barra de herramientas aparecen alineados a la izquierda, para que estén más al centro de la aplicación (como se ve en la figura 1), se debe añadir un separador. Un separador es también un botón que es

invisible, para añadir un separador se sigue el mismo procedimiento que para añadir un botón, sólo que se elige la opción *New Separator*. Arrastre este separador a la izquierda de la barra de herramientas y modifique su ancho de manera que los botones queden aproximadamente al centro de la aplicación.



Una vez creada la interfaz de la aplicación, comenzaremos declarando los tipos y variables que emplearemos en la misma. El siguiente código deberá ser introducido en el sector de *implementation* de la aplicación:

type

```
TNombre = record
  Paterno : string[15];
  Materno : string[15];
  Nombres : string[20];
end;
```

```
TFecha = record
  dia : byte;
  mes : byte;
  anio : word;
end;
```

```
TDirectorio = record
  Nombre : TNombre;
  DirDom : string[20];
  DirTra : string[20];
  TelDom : LongInt;
  TelTra : LongInt;
  Celular : LongInt;
  email : string[30];
  F_Nacimiento : TFecha;
```

end;

ti = **array of** LongInt;

var

x : **array of** TDirectorio;

Paterno : ti;

Materno : ti;

Nombres : ti;

Completo : ti;

Fecha : ti;

Indice : ti;

nr,ra,ia : LongInt;

modificado : boolean;

La variable *x* contendrá los registros de la aplicación. Las variables *Paterno*, *Materno*, *Nombres*, *Completo* y *Fecha*, se emplearán como arrays índice, es decir arrays con números de registro. Estos números de registro deben estar en un orden tal que al seguirlos los datos aparecen en un determinado orden. Así al seguir los números de registro de *Paterno*, los datos se aparecen ordenados alfabéticamente por apellido paterno, al seguir los números de registro de *Fecha*, los datos aparecen ordenados cronológicamente por fecha de nacimiento, etc.

La variable *Indice*, se empleará como una variable temporal para almacenar el nombre del array índice que se esté empleando en un momento dado.

La variable *nr* almacenará el número de registros existentes en el array *x*, la variable *ra*, almacenará el número de registro actualmente visualizado, la variable *ia* almacenará el índice actual dentro del array índice y la variable *modificado*, será verdadera cuando alguno de los datos de un registro sea modificado y falso en caso contrario.

En la introducción de datos a un registro es importante asegurarse de que los mismos tengan el menor número de errores posible, esto se consigue en gran parte con validaciones. En la presente aplicación realizaremos las siguientes validaciones (que por cierto no son completas):

En el caso del Apellido Paterno, Materno y Nombres, sólo se debe permitir la introducción de letras y espacios, además para dar uniformidad a los datos, sólo se permite la introducción de datos en mayúsculas. Para este fin se ha elaborado la siguiente función (que debe estar en el sector de implementación):

function Mayuscula(Key: Char):Char;

begin

if Key='ñ' then Mayuscula:= 'Ñ'

else Mayuscula:=UpCase(Key);

end;

Y la validación se la efectúa en el evento OnKeyPress de los cuadros de edición Edit1, Edit2 y Edit3:

procedure TForm1.Edit1KeyPress(Sender: TObject; **var** Key: Char);

begin

Key := Mayuscula(Key);

if not (Key in [#8, ' ', 'A'..'Z', 'Ñ']) then abort;

end;

En cuanto a la dirección de domicilio y de trabajo, la validación difiere sólo en que en este caso se permite además la introducción de números y el

símbolo de número °. La validación en el evento OnKeyPress de Edit4 y Edit5 es la siguiente:

```
procedure TForm1.Edit4KeyPress(Sender: TObject; var Key: Char);
begin
  Key := Mayuscula(Key);
  If not (Key in [#8, ' ', '.', '0'..'9', 'A'..'Z', 'Ñ', '°']) then abort;
end;
```

En lo referente al e-mail, ocurre lo contrario, pues las direcciones electrónicas tienen que estar en minúsculas, por ello en este caso se ha elaborado la siguiente función:

```
function Minuscula(Key: Char):Char;
begin
  if Key in ['A'..'Z'] then
    Minuscula := Chr(Ord(Key)+32)
  else
    Minuscula := Key;
end;
```

Y la validación, en el evento OnKeyPress de Edit6, la cual sólo permite letras en minúsculas y algunos símbolos especiales como el símbolo @, es la siguiente:

```
procedure TForm1.Edit6KeyPress(Sender: TObject; var Key: Char);
begin
  Key := Minuscula(Key);
  if not (Key in [#8, '.', '/', '0'..'9', '@', 'a'..'z', '~']) then abort;
end;
```

En los números telefónicos y en la fecha de nacimiento se emplean objetos MaskEdit, por lo que la validación (inicial) se la ha efectuado en la propiedad EditMask de estos objetos. Así para los teléfonos de domicilio y de trabajo se ha empleado la siguiente máscara:

0-0000;0;_

La cual puede ser escrita directamente en la propiedad EditMask o puede ser editada haciendo click en los 3 puntos que se encuentran al lado derecho de esta propiedad. El símbolo 0 de esta máscara sólo permite la introducción de un número en esta posición, el 0 después del ";" le indica a Delphi que no guarde los caracteres literales (el guión "-" en nuestro caso). Y el subrayado después del segundo ";" es el carácter que aparece en los espacios en blanco. Para aprender acerca de los diferentes símbolos que se pueden emplear en las máscaras, haga click en *Help* en el cuadro de edición de la máscara y luego click en *EditMask*.

En el celular se emplea una máscara similar:

000-00000;0;_

Y en la fecha de nacimiento se emplea la máscara:

99/99/9999;0;0

Al iniciar la aplicación se deben inicializar algunas variables, para ello empleamos el evento OnCreate de la forma:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  ReservarEspacio;
  ra := 0; ia := 0; nr := 0;
  Indice := Paterno;
  NoModificado;
```

end;

Donde los procedimientos *ReservarEspacio* y *NoModificado*, que deben estar en el sector de implementación, antes del evento, son los siguientes:

```
procedure ReservarEspacio;
begin
  SetLength(x,Length(x)+10);
  SetLength(Paterno,Length(Paterno)+10);
  SetLength(Materno,Length(Materno)+10);
  SetLength(Nombres,Length(Nombres)+10);
  SetLength(Completo,Length(Completo)+10);
  SetLength(Fecha,Length(Fecha)+10);
end;
```

```
procedure NoModificado;
begin
  Modificado := False;
  Form1.ToolButton7.Enabled := False;
  Form1.ToolButton8.Enabled := False;
end;
```

El primer procedimiento se encargan de reservar espacio extra para los arrays dinámicos y el segundo se encarga de colocar la variable modificado en falso e inhabilitar los botones confirmar cambios y cancelar.

Para que al modificar algún dato se coloque en verdadero la variable *Modificado*, se escribe el siguiente código en el evento *OnChange* de todos los *Edit* y de todos los *EditMask*. Esto se puede hacer, en un paso, seleccionando todos estos objetos y escribiendo el siguiente código en el evento *OnChange*:

```
procedure TForm1.Edit1Change(Sender: TObject);
begin
  Modificado := True;
  if not ToolButton8.Enabled then
    ToolButton8.Enabled := True;
  if not ToolButton9.Enabled then
    ToolButton9.Enabled := True;
end;
```

Que como se puede observar hace lo contrario del procedimiento *NoModificado*.

Para mostrar los datos de un registro en la forma se ha elaborado el siguiente procedimiento:

```
procedure MostrarRegistro;
begin
  with form1,x[ra],nombre,f_nacimiento do
  begin
    Edit1.Text := Paterno;
    Edit2.Text := Materno;
    Edit3.Text := Nombres;
    Edit4.Text := DirDom;
    Edit5.Text := DirTra;
    MaskEdit1.Text := IntToStr(TelDom);
    MaskEdit2.Text := IntToStr(TelTra);
    MaskEdit3.Text := IntToStr(Celular);
    Edit6.Text := email;
    MaskEdit4.Text := FechaACadena(f_nacimiento);
    NoModificado;
  end;
```


end;

La función FechaACadena que emplea este procedimiento y que como su nombre sugiere convierte una fecha en una cadena, es la siguiente:

```
function FechaACadena(var f: TFecha):string;
var fecha: LongInt; s: string;
begin
    fecha := f.dia*1000000+f.mes*10000+f.anio;
    s := IntToStr(fecha);
    if fecha<10000000 then s := '0'+s;
    FechaACadena := s;
end;
```

Esta función debe ser copiada antes de la función *MostrarRegistro* (en el sector de implementación).

Para leer los datos de la forma en un registro se ha elaborado el siguiente procedimiento:

```
procedure LlenarRegistro(var r: TDirectorio);
begin
    with Form1,r,nombre,f_nacimiento do
        begin
            Paterno := Edit1.Text;
            Materno := Edit2.Text;
            Nombres := Edit3.Text;
            DirDom := Edit4.Text;
            DirTra := Edit5.Text;
            if MaskEdit1.Text<>' ' then TelDom := StrToInt(MaskEdit1.Text);
            if MaskEdit2.Text<>' ' then TelTra := StrToInt(MaskEdit2.Text);
            if MaskEdit3.Text<>' ' then Celular := StrToInt(MaskEdit3.Text);
            email := Edit6.Text;
            if MaskEdit4.Text<>' ' then f_Nacimiento := CadenaAFecha(MaskEdit4.Text);
        end;
    end;
```

Donde la función CadenaAFecha, que como su nombre sugiere convierte una cadena en una fecha, es la siguiente:

```
function CadenaAFecha(s : string):TFecha;
var f : TFecha;
begin
    with f do
        begin
            dia := StrToInt(Copy(s,1,2));
            mes := StrToInt(Copy(s,3,2));
            anio := StrToInt(Copy(s,5,4));
        end;
    CadenaAFecha := f;
end;
```

Para borrar los campos del formulario y presentar un registro en blanco se ha elaborado la siguiente función:

```
procedure BorrarFormulario;
begin
    with form1 do
        begin
            Edit1.Clear;
            Edit2.Clear;
            Edit3.Clear;
```

```

    Edit4.Clear;
    Edit5.Clear;
    MaskEdit1.Clear;
    MaskEdit2.Clear;
    MaskEdit3.Clear;
    Edit6.Clear;
    MaskEdit4.Clear;
end;
end;

```

Para añadir un nuevo registro se debe hacer click sobre el botón Nuevo Registro (ToolButton5). El evento OnClick de este botón es el siguiente:

```

procedure TForm1.ToolButton5Click(Sender: TObject);
begin
    if Modificado then ConfirmarCambios;
    if nr>Length(x)-2 then ReservarEspacio;
    BorrarFormulario;
    inc(nr); ra := nr;
    MostrarRegistro;
    Form1.ActiveControl := Edit1;
end;

```

Donde el procedimiento *ConfirmarCambios* es el siguiente:

```

procedure ConfirmarCambios;
begin
    case MessageDlg('¿Quiere guardar las'+
                    'modificaciones efectuadas?',
                    mtConfirmation, [MBYES, MBNO, MBCANCEL], 0)
    of
        mrYes :
            begin
                LlenarRegistro(x[ra]);
                ActualizarIndices;
                ia := UbicarRegistro(Indice, ra);
            end;
        mrCancel : Abort;
    end;
    NoModificado;
end;

```

Como se puede observar, en este procedimiento se pregunta al usuario si desea guardar las modificaciones, en caso afirmativo se llena el registro con los datos del formulario y con el procedimiento *ActualizarIndices*, se actualizan los array índice (pues al cambiar los datos es muy probable que el orden también sufra un cambio). Una vez actualizado los índices se coloca el índice activo en el registro actual mediante la función *UbicarRegistro*.

La función *UbicarRegistro* (que debe ser escrita antes del primer procedimiento que lo utiliza) es la siguiente:

```

function UbicarRegistro(y:ti; reg:LongInt):LongInt;
var i : LongInt;
begin
    i := 0;
    while y[i]<>reg do inc(i);
    UbicarRegistro := i;
end;

```

Como se puede observar, en este procedimiento se ubica el registro (reg) empleando el método de búsqueda lineal (pues los números de registros no están ordenados).

El procedimiento para actualizar los array índice es el siguiente:

```
procedure ActualizarIndices;
begin
  if nr=0 then
    begin
      Paterno[0] := 0;
      Materno[0] := 0;
      Nombres[0] := 0;
      Completo[0] := 0;
      Fecha[0] := 0;
    end
  else
    begin
      ActualizarPaterno;
      ActualizarMaterno;
      ActualizarNombres;
      ActualizarCompleto;
      ActualizarFecha;
    end;
end;
```

Donde los procedimientos *ActualizarPaterno*, *ActualizarMaterno*, *ActualizarNombres*, *ActualizarCompleto* y *ActualizarFecha* son los siguientes (recuerde que en todos los casos estos procedimientos se escriben en el sector de implementación y que un procedimiento debe estar ubicado antes del procedimiento que lo utiliza):

```
procedure ActualizarPaterno;
var i,j,k : LongInt; aux,elemento : string;
begin
  if ra<>nr then
    begin
      k := UbicarRegistro(Paterno,ra);
      for i:=k to nr-1 do Paterno[i]:=Paterno[i+1];
    end;
  aux :=x[ra].Nombre.Paterno;
  j := nr-1;
  repeat
    elemento := x[Paterno[j]].Nombre.Paterno;
    if aux<elemento then Paterno[j+1] := Paterno[j]
    else break;
  dec(j);
  until j<0;
  Paterno[j+1] := ra;
end;
```

```
procedure ActualizarMaterno;
var i,j,k : LongInt; aux,elemento : string;
begin
  if ra<>nr then
    begin
      k := UbicarRegistro(Materno,ra);
      for i:=k to nr-1 do Materno[i]:=Materno[i+1];
    end;
  aux :=x[ra].Nombre.Materno;
```

```

j := nr-1;
repeat
  elemento := x[Materno[j]].Nombre.Materno;
  if aux<elemento then Materno[j+1] := Materno[j]
  else break;
  dec(j);
until j<0;
Materno[j+1] := ra;
end;

```

```

procedure ActualizarNombres;
var i,j,k : LongInt; aux,elemento : string;
begin
  if ra<>nr then
    begin
      k := UbicarRegistro(Nombres,ra);
      for i:=k to nr-1 do Nombres[i]:=Nombres[i+1];
    end;
  aux :=x[ra].Nombre.Nombres;
  j := nr-1;
  repeat
    elemento := x[Nombres[j]].Nombre.Nombres;
    if aux<elemento then Nombres[j+1] := Nombres[j]
    else break;
    dec(j);
  until j<0;
  Nombres[j+1] := ra;
end;

```

```

procedure ActualizarCompleto;
var i,j,k : LongInt; aux,elemento : string;
begin
  if ra<>nr then
    begin
      k := UbicarRegistro(Completo,ra);
      for i:=k to nr-1 do Completo[i]:=Completo[i+1];
    end;
  with x[ra],Nombre do
    aux :=Paterno+' '+Materno+' '+Nombres;
  j := nr-1;
  repeat
    with x[Completo[j]],Nombre do
      elemento := Paterno+' '+Materno+' '+Nombres;
      if aux<elemento then Completo[j+1]:=Completo[j]
      else break;
      dec(j);
    until j<0;
    Completo[j+1] := ra;
  end;

```

```

procedure ActualizarFecha;
var i,j,k : LongInt; aux,elemento : integer;
begin
  if ra<>nr then
    begin
      k := UbicarRegistro(Fecha,ra);
      for i:=k to nr-1 do Fecha[i]:=Fecha[i+1];
    end;

```

```

with x[ra],F_Nacimiento do
  aux := Anio*10000+Mes*100+Dia;
j := nr-1;
repeat
  with x[Fecha[j]],F_Nacimiento do
    elemento := Anio*10000+Mes*100+Dia;
    if aux<elemento then Fecha[j+1] := Fecha[j]
    else break;
  dec(j);
until j<0;
Fecha[j+1] := ra;
end;

```

Como puede observar en todos estos procedimientos se actualizan los índices empleando el método de inserción. Si el registro que ha sido modificado no se encuentra al final del array se elimina primero el número de registro del índice (y se reinserta luego empleando el método de inserción).

Para eliminar un registro se debe hacer click sobre el botón *Eliminar Registro* (ToolButton6). El evento OnClick de este botón es el siguiente:

```

procedure TForm1.ToolButton6Click(Sender: TObject);
begin
  if
    MessageDlg('¿Desea eliminar todos los datos'+
      de este registro?', mtConfirmation,
      [MBYES,MBCANCEL],0) = mrYes
  then
    begin
      BorrarFormulario;
      if nr=0 then
        FillChar(x[ra],SizeOf(x[ra]),0)
      else
        begin
          EliminarRegistro;
          EliminarDeIndices;
          if ia=nr then dec(ia);
          ra := Indice[ia];
          dec(nr);
        end;
      MostrarRegistro;
    end;
end;

```

Como se puede observar, en este procedimiento se pide confirmación al usuario para eliminar el registro. En caso afirmativo se borra la forma y si ya no existen más registros (nr=0) se borra el último registro, caso contrario se elimina el registro del array (con el procedimiento *EliminarRegistro*) y se elimina ese número de registro de los array índice (con el procedimiento *EliminarDeIndices*). Si el registro que se ha eliminado es el último, se coloca el índice actual (ia) en el registro anterior y se actualiza el número de registro actual (ra) y el número de registros (nr).

El procedimiento *EliminarRegistro*, que como su nombre sugiere Elimina el registro actual (ra) del array, es el siguiente:

```

procedure EliminarRegistro;
var i : LongInt;
begin
  if nr=0 then FillChar(x[ra],SizeOf(x[ra]),0);
  for i:=ra to nr-1 do x[i] := x[i+1];

```

end;

El procedimiento *EliminarDeIndices*, que como su nombre sugiere elimina el registro actual de los array índice, es el siguiente:

```
procedure EliminarDeIndices;
begin
  EliminarDeIndice(Paterno,ra);
  EliminarDeIndice(Materno,ra);
  EliminarDeIndice(Nombres,ra);
  EliminarDeIndice(Completo,ra);
  EliminarDeIndice(Fecha,ra);
end;
```

Donde el procedimiento *EliminarDeIndice*, que elimina el registro de un array índice específico, es el siguiente:

```
procedure EliminarDeIndice(y:ti;reg:LongInt);
var i,k : LongInt;
begin
  k := UbicarRegistro(y,reg);
  for i:=k to nr-1 do y[i]:=y[i+1];
  for i:=0 to nr do
    if y[i]>reg then y[i]:=y[i]-1;
end;
```

Para guardar los cambios efectuados en un registro sin añadir un nuevo registro o cambiar a otro registro, se debe hacer click sobre el botón Confirmar Cambios (ToolButton7). El procedimiento del evento OnClick de este botón es el siguiente:

```
procedure TForm1.ToolButton7Click(Sender: TObject);
begin
  LLenarRegistro(x[ra]);
  ActualizarIndices;
  ia := UbicarRegistro(Indice,ra);
  NoModificado;
end;
```

Para anular los cambios efectuados en un registro y recuperar los valores anteriores se debe hacer click sobre el botón Cancelar (ToolButton8). El evento OnClick de este botón es el siguiente:

```
procedure TForm1.ToolButton8Click(Sender: TObject);
begin
  MostrarRegistro;
  NoModificado;
end;
```

Para ir al primer registro de la lista (en el orden que se esté mostrando la lista actualmente) se debe hacer click sobre el botón Primer Registro (ToolButton1). El Evento OnClick de este botón es:

```
procedure TForm1.ToolButton1Click(Sender: TObject);
begin
  if Modificado then ConfirmarCambios;
  ia := 0;
  ra := Indice[ia];
  MostrarRegistro;
end;
```

En forma similar, para ir al último registro de la lista se debe hacer click sobre el botón Ultimo Registro (ToolButton4). El evento OnClick de dicho botón es el siguiente:

```
procedure TForm1.ToolButton4Click(Sender: TObject);  
begin  
    if Modificado then ConfirmarCambios;  
    ia := nr;  
    ra := Indice[ia];  
    MostrarRegistro;  
end;
```

Para pasar de un registro al registro anterior se debe hacer click sobre el botón Anterior Registro (ToolButton2). El evento OnClick de dicho botón es:

```
procedure TForm1.ToolButton2Click(Sender: TObject);  
begin  
    if Modificado then ConfirmarCambios;  
    if ia>0 then  
        begin  
            dec(ia);  
            ra := Indice[ia];  
            MostrarRegistro;  
        end  
    else  
        Beep;  
end;
```

En forma similar, para pasar de un registro al siguiente se debe hacer click sobre el botón Siguiente Registro (ToolButton3). El evento OnClick de dicho botón es:

```
procedure TForm1.ToolButton3Click(Sender: TObject);  
begin  
    if modificado then ConfirmarCambios;  
    if nr>ia then  
        begin  
            inc(ia);  
            ra := Indice[ia];  
            MostrarRegistro;  
        end  
    else  
        Beep;  
end;
```

Para cambiar el orden en que se muestran los registros, se debe elegir una de las opciones del menú *Orden*. Esto también constituye un evento OnClick. Los eventos OnClick de las opciones (a los cuales se accede simplemente haciendo click en la opción), para las opciones Apellido Paterno, Apellido Materno, Nombres, Nombre Completo y Fecha de Nacimiento son los siguientes:

```
procedure TForm1.ApellidoPaterno1Click(Sender: TObject);  
begin  
    Indice := Paterno;  
    ia := UbicarRegistro(Indice,ra);  
end;
```

```
procedure TForm1.ApellidoMaterno1Click(Sender: TObject);  
begin
```

```
Indice := Materno;
ia := UbicarRegistro(Indice,ra);
end;

procedure TForm1.Nombres1Click(Sender: TObject);
begin
    Indice := Nombres;
    ia := UbicarRegistro(Indice,ra);
end;

procedure TForm1.NombreCompleto1Click(Sender: TObject);
begin
    Indice := Completo;
    ia := UbicarRegistro(Indice,ra);
end;

procedure TForm1.FechadeNacimiento1Click(Sender: TObject);
begin
    Indice := Fecha;
    ia := UbicarRegistro(Indice,ra);
end;
```

El menú *Salir* lo único que hace es terminar la aplicación. Para ello se puede llamar al procedimiento *Application.Terminate*, o llamar al procedimiento *OnClose* de la forma principal (en nuestro caso la única forma disponible). Esta es la opción que se ha elegido para la presente caso, pues al cerrar la aplicación se realizan algunas tareas:

```
procedure TForm1.Salir1Click(Sender: TObject);
begin
    Close;
end;
```

Finalmente al cerrar la aplicación se libera la memoria empleada por los array dinámicos. Como en estos arrays la memoria es reservada por el programador y no por Delphi, Delphi no puede liberarla automáticamente, por lo que debe ser liberada manualmente. De no liberarse esta memoria, seguiría en uso aún después de terminar la aplicación disminuyendo progresivamente la memoria disponible en la computadora. El mejor lugar para liberar la memoria ocupada por estos arrays es al momento de cerrar la aplicación, por lo tanto en el procedimiento *OnClose* de la forma escribimos el siguiente código:

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
    Finalize(x);
    Finalize(Paterno);
    Finalize(Materno);
    Finalize(Nombres);
    Finalize(Completo);
    Finalize(Fecha);
end;
```

Con este último procedimiento la aplicación deberá ser funcional y capaz de realizar las tareas que se fijaron inicialmente.

Aun cuando la aplicación ahora es funcional y aparentemente se ha trabajado ya bastante en la misma, no está todavía completa. Por ejemplo no se puede realizar búsquedas en ningún campo. Las validaciones no están completas, por ejemplo al introducir la fecha de nacimiento controlamos que los datos introducidos sean números, pero no estamos controlando que dichos número sean coherentes, es decir el número de días no sea mayor a 31, que el

número de meses no sea mayor a 12 y que el número de año no sea mayor al año actual.

Sin embargo, la falla más importante de la aplicación es que los datos pueden ser empleados sólo mientras la aplicación está corriendo, una vez que se sale de la aplicación los datos se pierden definitivamente, por lo que en una siguiente corrida es necesario volver a introducir datos. Esta falla (y algunas otras) será corregida en el siguiente capítulo con el uso de archivos de disco.

23.4 EJERCICIOS

1. Elabore una aplicación que permita modificar, añadir y borrar la siguiente información correspondiente a una biblioteca: Código de libro; Título de la obra; Autor (o autores); Editorial; País; Año de Publicación; Número de Edición y Categoría (Social, Técnica, Religiosa, Medicina, Autosuperación, Entretenimiento, etc.). La aplicación debe mostrar la información ordenada por Título, por Autor por Editorial y por Año de publicación.
2. Elabore una aplicación que permita modificar, añadir y borrar la siguiente información correspondiente a una ferretería: Código del artículo; Nombre del artículo; Categoría (jardinería, electricidad, carpintería, construcción, etc.); Precio; Cantidad disponible; Unidad de medida (pieza, Kg, Litros, bolsa, metros, etc.); Proveedor; Marca y País. La aplicación debe mostrar la información ordenada por nombre, categoría, país y marca.

23. ARCHIVOS

Como se ha podido observar al trabajar con registros, los programas elaborados resultan de poca utilidad práctica si los datos no pueden ser recuperados cuando se hace correr el programa. Este inconveniente se resuelve con el empleo de *archivos*.

En informática se da el nombre de archivo o fichero (*file*) a la información almacenada en un dispositivo de almacenamiento secundario (cintas y discos magnéticos, discos ópticos, etc.). Así, toda la información existente en un disco duro: programas (incluido Turbo Pascal), datos, documentos, cartas, dibujos, etc., son archivos (ficheros).

En general **un archivo o fichero es una colección de registros relacionados**. Por ejemplo, los ficheros (archivos) de una biblioteca guardan fichas (registros) con información de cada uno de los libros existentes en la misma; los ficheros de un hospital guardan fichas con información de cada uno de los pacientes atendidos en el mismo; los ficheros de una empresa guardan fichas con información de cada uno de sus empleados; los ficheros de una unidad académica guardan fichas con información de cada uno de los alumnos que han estudiado o estudian en la misma, etc.

En Delphi y otros lenguajes como C++ Builder, el concepto de archivos ha sido ampliado de manera que un archivo se refiere no sólo a la información almacenada en un dispositivo de almacenamiento secundario, sino a cualquier dispositivo que pueda recibir o mandar datos a una computadora, así la pantalla, el teclado, la impresora, el modem, el mouse, el escaner, etc., son tratados en estos lenguajes como archivos.

En Borland Pascal de Delphi, podemos crear tres tipos de archivos: Archivos de texto o secuenciales, archivos de acceso directo o con tipo y archivos sin tipo.

23.3 ARCHIVOS DE TEXTO O SECUENCIALES

Los archivos de texto, como su nombre sugiere, permiten almacenar (y recuperar) información en forma de texto. Este tipo de archivos fue creado con el propósito de almacenar y recuperar información de cintas magnéticas. Debido a la forma de acceso que permiten las cintas magnéticas este tipo de archivos sólo permite la lectura o almacenamiento *secuencial* de información, es decir para leer un determinado dato almacenado en un archivo, se debe recorrer el archivo desde el principio hasta encontrar el dato buscado. En este tipo de archivos los datos no pueden ser modificados y sólo se puede añadir información adicional al final del archivo.

Como actualmente el principal medio de almacenamiento secundario son los discos magnéticos y ópticos (y no las cintas magnéticas) este tipo de archivos están en desuso.

Para almacenar datos en un archivo de texto se debe declarar primero una variable de este tipo, para ello se emplea la palabra reservada *TextFile*, por ejemplo la sentencia:

```
var ft : TextFile;
```

Declara la variable ft de tipo archivo de texto. Posteriormente se debe relacionar esta variable con algún archivo ubicado en algún dispositivo de almacenamiento externo (normalmente un disco), empleando para ello el procedimiento *AssignFile*, por ejemplo:

```
AssignFile(ft,'C:\Mis Documentos\Progs\datos.txt');
```

Relaciona la variable *ft* con el archivo *datos.txt*, ubicado en la unidad *C*, en el subdirectorío *Progs* de *Mis Documentos*. Una vez hecha esta asignación todos los datos que se lean o almacenen en *ft*, se estarán leyendo o almacenando en realidad en el archivo *datos.txt*.

Si el archivo con el que se relacionó la variable *archivo* no existe en disco puede ser creado empleando el procedimiento *Rewrite*, por ejemplo:

```
Rewrite(ft);
```

Crea el archivo *datos.txt* en la unidad *C*, en el subdirectorío *Progs* del directorío *Mis Documentos*. Este procedimiento debe ser empleado con mucho cuidado, pues si ya existía un archivo con el mismo nombre dicho archivo es eliminado definitivamente sin pedir ninguna confirmación.

Si el archivo ya existía puede ser abierto para leer información del mismo empleando el procedimiento *Reset*, por ejemplo:

```
Reset(ft);
```

Abre el archivo *datos.txt*, quedando el puntero al principio del archivo listo para comenzar a leer los datos. Para leer los datos del archivo se emplea el procedimiento *read* o *readln*, por ejemplo:

```
Read(ft,s);
```

Lee una cadena del archivo *datos.txt*, hasta que encuentra los códigos de fin de línea (#13 y #10) y asigna la cadena leída a la variable *s*. Después de leer la cadena el procedimiento *Read* no salta los códigos de fin de línea, de manera que los sucesivos *Read* devolverán el carácter nulo en forma indefinida, sin llegar nunca al fin del archivo.

Si los datos que se encuentran en el archivo son números el comando *Read*, lee la cadena hasta que encuentra un espacio en blanco o los códigos de fin de línea, transforma la cadena a número y almacena el número en la variable. Si se desea leer más de un número a la vez, se puede emplear una lista de variables separadas por comas.

Un procedimiento similar a *Read* es el procedimiento *Readln*, por ejemplo:

```
Readln(ft,s2)
```

Lee una cadena, hasta que encuentra los códigos de fin de línea y de salto de página y asigna la cadena leída a la variable *s2*, sin embargo, a diferencia de *Read*, *Readln* salta los códigos de fin de línea (#13 y #10) y deja el puntero después de estos códigos, de manera que el siguiente comando *Readln* (o *Read*) lee la cadena que se encuentra en la siguiente línea.

Para saber si se ha alcanzado o no el fin de un archivo se emplea la función *EOF*, por ejemplo

```
EOF(ft)
```

Devuelve verdadero si se ha llegado al final del archivo *datos.txt*, es decir si se ha encontrado el carácter #26. Generalmente esta función se encuentra dentro de un ciclo *while*. Por ejemplo el siguiente ciclo:

```
while not EOF(ft) do  
begin  
  Readln(ft,s);  
  Mem1.Lines.Add(s);  
end;
```

Lee el texto contenido en el archivo *datos.txt*, línea por línea, almacenando el texto leído en un *memo*, hasta que se alcanza el fin del archivo, es decir hasta que *EOF(ft)* devuelva *True*.

Una vez leído el contenido de un archivo el mismo puede ser cerrado empleando el procedimiento *CloseFile*, por ejemplo:

```
CloseFile(ft);
```

Finaliza la relación que existe entre la variable *ft* y el archivo *datos.txt*. Además de terminar la relación entre la variable y el archivo, el procedimiento *CloseFile* guarda en disco los datos que pertenecen al archivo pero que todavía se encuentran en memoria. Una vez finalizada la relación, la variable *ft* puede ser asociada a otro archivo.

Para añadir datos a un archivo de texto el mismo debe ser abierto empleando el procedimiento *Append*, por ejemplo:

```
Append(ft);
```

Abre el archivo *datos.txt* y coloca el puntero después del último dato existente, de manera que se pueden añadir datos al mismo. Para añadir dichos datos, se puede emplear el procedimiento *Write* o *Rewrite*. La diferencia entre estos dos procedimientos es la misma que entre los procedimientos *Read* y *Readln*. El procedimiento *Write* escribe él o los datos en el archivo y deja el puntero después del o los datos escritos, mientras que *Writeln* escribe él o los datos en el archivo y después añade los códigos de fin de línea (#13 y #10). Por ejemplo:

```
Write(ft,'Esta es una cadena de prueba');
```

Almacena el texto *Esta es una cadena de prueba* en el archivo '*datos.txt*', y deja el puntero después de la palabra *prueba*, mientras que

```
Writeln(ft,'Esta es una cadena de prueba');
```

Almacena el mismo texto en el archivo *datos.txt*, pero añade al final del mismo los códigos de fin de línea #13 y #10.

Como ya se mencionó previamente, cuando se emplea el procedimiento *rewrite*, para crear un nuevo archivo, si ya existe un archivo con el mismo nombre es eliminado sin pedir de confirmación. En Delphi existen varios procedimientos y funciones que permiten verificar la existencia de un archivo, una de esas funciones es *IOResult*, que devuelve el código de error de una operación de entrada (Input) o salida (Output). Si no se produce error alguno devuelve cero. Para emplear esta función se debe desactivar el control de errores de entrada y salida *{SI-}*, por ejemplo en el siguiente código:

```
{SI-} reset(ft); {SI+}  
if IOResult=0 then  
    ShowMessage('El archivo ya existe en disco')  
else  
    ShowMessage('El archivo no existe en disco');
```

Se intenta abrir el archivo *datos.txt*. Si el archivo es abierto con éxito (porque existe en disco) no se produce ningún error (*IOResult*=0), caso contrario (el archivo no existe en disco) se produce un código de error (*IOResult*<>0). Una función más específica para este fin es *FileExists*, que devuelve verdadero si el archivo existe y falso en caso contrario. Por ejemplo un código que emplea esta función y que hace lo mismo que el anterior es el siguiente:

```
if FileExists('C:\Mis Documentos\Progs\datos.txt')  
    then  
    ShowMessage('El archivo ya existe en disco')  
else  
    ShowMessage('El archivo no existe en disco');
```

Las operaciones de lectura y escritura en disco consumen mucho tiempo, debido a que son en parte operaciones mecánicas. Para mejorar la velocidad con la que se efectúan tales operaciones, se emplea parte de la memoria RAM de la computadora en lugar del disco, a esta memoria se conoce como *memoria Buffer*. Con la memoria Buffer, cuando se manda a leer un dato del disco, se lee el dato y toda la información que puede ser almacenada en la memoria Buffer, de manera que la siguiente vez que se manda a leer un dato de disco se busca primero en la memoria Buffer y sólo en caso de que dicho dato no se encuentre en el Buffer se recurre al disco. Igualmente, con la memoria Buffer cuando se manda a escribir un dato en disco se escribe dicho dato en la memoria Buffer y sólo cuando la memoria Buffer ha sido empleada completamente se pasa todo el contenido de la misma a disco. De esta manera se reduce el número de veces que es necesario leer o escribir en disco y con ello se logra un incremento en la velocidad. Para reservar memoria RAM como memoria Buffer se emplea el procedimiento *SetTextBuf*, por ejemplo en el siguiente código:

```
var bf: array [1..10000] of byte; f1: Text;
...
AssignFile(f1,'C:\datos2.txt');
SetTextBuf(f1,bf);
Reset(f1);
```

Relaciona el archivo *datos2.txt* con la variable *f1*. Asigna a dicho archivo un buffer (bf) de 10000 bytes y abre el archivo para lectura. Si no se asigna ningún buffer al archivo entonces Delphi le asigna un buffer de 128 bytes, el cual es muy pequeño para la mayoría de las aplicaciones.

El empleo de Buffers sin embargo tiene un inconveniente, si se corta la energía o se cuelga la aplicación, todos los datos que estaban en el Buffer se pierden. Por eso es que en las operaciones de escritura (cuando se abre el archivo con *rewrite* o *append*), se puede emplear el procedimiento *Flush*, el cual pasa los datos contenidos en el Buffer a disco y limpia el Buffer. Por ejemplo:

```
Flush(f1);
```

Pasa todos los datos contenidos en *bf* a disco.

23.4 ARCHIVOS CON TIPO O DE ACCESO DIRECTO

En los archivos con tipo todos los datos son registros del mismo tipo. La característica más importante de este tipo de archivos es el acceso directo a sus registros, es decir la posibilidad de leer o modificar cualquiera de sus registros con sólo conocer su posición (a diferencia de los archivos de texto donde es necesario recorrer el archivo desde el principio hasta ubicar el dato buscado).

Otra diferencia importante con relación a los archivos de texto es que la información se almacena empleando el mismo formato que en la memoria RAM, de manera que no es necesario realizar ningún tipo de conversión al leer o escribir registros en estos archivos.

Al igual que en los archivos de texto, para manejar archivos con tipo se debe declarar primero una variable archivo empleando este caso las palabras reservadas *file of*, así por ejemplo:

```
var ff : file of TDirectorio;
```

Declara la variable archivo *ff* que permitirá leer, guardar o modificar datos de tipo *Tdirectorio* (el tipo base puede ser un registro como en este caso o cualquier tipo válido en pascal).

Al igual que en los archivos de texto, una vez declarada la variable *archivo* se relaciona la misma con algún archivo físico empleando para ellos el procedimiento *AssignFile*, así:

```
AssignFile(ff,'c:\Directorio.dat');
```

Relaciona la variable *ff* con el archivo de disco *Directorio.dat*. Para abrir un archivo con tipo se emplea el procedimiento *Reset*, así:

```
Reset(ff);
```

Abre el archivo *Directorio.dat*. En este caso sin embargo, el archivo queda abierto no solo para lectura sino también para escritura. Igualmente para crear un archivo se emplea el procedimiento *Rewrite*, así:

```
Rewrite(ff);
```

Crea el archivo *Directorio.dat* quedando el archivo abierto para lectura o para escritura.

Para terminar la relación entre un archivo de disco y una variable *archivo* con tipo se emplea el procedimiento *CloseFile*, así:

```
CloseFile(ff);
```

Termina la relación entre el archivo *Directorio.dat* y la variable *ff*.

Para leer un registro de un archivo con tipo se emplea el procedimiento *read*, así:

```
Read(ff,d);
```

Lee un registro del archivo *Directorio.dat*, deja el puntero al principio del siguiente registro y almacena el registro leído en la variable *d* (que deberá ser de tipo *Tdirectorio*).

Para escribir un registro en un archivo con tipo se emplea el procedimiento *write*, así:

```
Write(ff,d);
```

Escribe el registro *d* en el archivo *Directorio.dat* y deja el cursor al final del registro escrito listo para escribir o leer otro registro. En los archivos con tipo no se pueden emplear los procedimientos *Readln* y *Writeln*, debido a que en estos archivos no existen líneas de texto, sino registros.

En los archivos con tipo, al igual que en un archivo de texto, se pueden emplear las funciones *EOF* (para determinar el fin de un archivo) y *IOResult* o *FileExists* (para verificar la existencia de un archivo).

Los archivos con tipo no emplean *buffers*, razón por la cual no se pueden emplear los procedimientos *SetTextBuf* y *Flush*. Igualmente no se puede emplear el procedimiento *Append*, pues los archivos con tipo quedan abiertos tanto para lectura como para escritura.

Adicionalmente en los archivos con tipo se pueden emplear los siguientes procedimientos y funciones: *Seek*, este procedimiento mueve el puntero a la posición especificada (el primer registro se encuentra en la posición 0). Así:

```
Seek(ff,10);
```

Mueve el puntero a la posición número 10, es decir al principio del registro número 11. La función *FilePos* devuelve la posición del registro actual, así, si se ha ejecutado la anterior sentencia:

```
FilePos(ff)
```

Devolvería la posición 10 que corresponde al registro número 11. La función *FileSize* devuelve el número de registros existentes en el archivo, así:

```
FileSize(ff)
```

Devuelve el número de registros que existen en el archivo *datos.dat*. El procedimiento *Truncate*, elimina los registros que se encuentran después de la posición actual del puntero, así, si las anteriores sentencias hubieran sido ejecutadas:

```
Truncate(ff);
```

Eliminaría todos los registros existentes después del registro actual, en nuestro caso el archivo quedaría con 10 registros.

23.5 ARCHIVOS SIN TIPO

Los archivos sin tipo permiten almacenar y leer información en y de cualquier tipo de archivos. Sin embargo generalmente se emplean conjuntamente los archivos con tipo para leer o escribir grandes volúmenes de información.

Como se señaló anteriormente, los archivos con tipo no emplean buffers, por lo que cada vez que se lee o escribe en estos archivos se lee o escribe efectivamente en disco, como esta es una operación que consume mucho tiempo, cuando el número de veces que se debe leer o escribir en estos archivos es elevado, no resulta eficiente el empleo directo de archivos con tipo, en su lugar se puede emplear archivos sin tipo los cuales permiten el empleo de buffers, con lo que (al igual que sucede con los archivos de texto) es posible acelerar considerablemente las operaciones de lectura y escritura. Esta es la principal aplicación de los archivos sin tipo.

Los archivos sin tipo emplean prácticamente los mismos procedimientos y funciones que los archivos con tipo, excepto en las operaciones de lectura y escritura, donde no se pueden emplear los procedimientos *Read* y *Write*. Por otra parte los procedimientos *Reset* y *Rewrite* admiten un segundo parámetro que corresponde al tamaño (en bytes) de los registros del archivo, así por ejemplo en el siguiente código:

```
var f : file;
...
AssignFile(f,'C:\Datos.dat');
Reset(f,SizeOf(TDirectorio));
```

Se declara la variable archivo sin tipo *f* (note que para declarar un archivo sin tipo se escribe la palabra reservada *file*), se relaciona dicha variable con el archivo *datos.dat* y se abre el archivo estableciendo la longitud de cada registro en la longitud del tipo de dato *TDirectorio*. Si en lugar de abrir el archivo, se quisiera crear el mismo, se emplearía en lugar del procedimiento *read*, el procedimiento *rewrite*:

```
Rewrite(f,SizeOf(TDirectorio));
```

Para leer o almacenar información en un archivo sin tipo se puede emplear, como se mencionó buffers, con este fin se declaran variables (que pueden ser de cualquier tipo) pero que deben tener un tamaño adecuado para los fines de la aplicación.

Para leer un bloque de información se emplea el procedimiento *BlockRead*(*Variable archivo*, *Variable Buffer*, *Número de registros a leer*, *Número de registros leídos*), donde el *número de registros a leer* es el número de registros que se desea leer, mientras que el *número de registros leídos* (que es opcional) devuelve en una variable de tipo entero el número de registros que realmente han sido leídos del archivo. Para la escritura de información

se emplea un procedimiento similar: *BlockWrite(Variable archivo, Variable Buffer, Número de registros a escribir, Número de registros escritos)*, donde el *número de registros a escribir* es el número de registros que se desea escribir en el archivo, mientras que el *número de registros escritos* (que es opcional) devuelve en una variable de tipo entero el número de registros que realmente han sido escritos en el archivo.

Por ejemplo, las siguientes sentencias:

```
var buf1 : array [1..1000] of TDirectorio;  
...  
BlockRead(f,buf1,1000,RegLei);  
...  
BlockWrite(f,buf1,1000,RegEsc);
```

Declara la variable *Bufl*, que será empleada como una memoria buffer con una capacidad de 1000 registros. Con *BlockRead* se intenta leer del archivo *datos.dat* en la variable *Bufl*, 1000 registros. Si se logran leer los 1000 registros, en la variable *RegLei* queda el número 1000, caso contrario queda un número menor correspondiente al número de registros que realmente se han leído. Con *BlockWrite* se intenta escribir en el archivo *datos.dat*, 1000 registros que deben estar guardados en la variable *buf1*. Si se escriben los 1000 registros con éxito en la variable *RegEsc* se guarda el número 1000, caso contrario en la variable *RegEsc* se guarda el número de registros que han sido escritos con éxito.

23.6 EJEMPLO

Como ejemplo modificaremos la aplicación creada en registros de manera que sea posible guardar y recuperar los datos de disco.

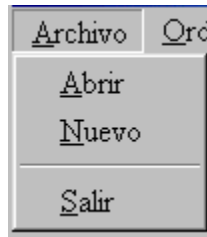
La interfaz de la aplicación tiene la siguiente apariencia:



The screenshot shows a Windows-style application window titled "Registros". It has a menu bar with "Archivo" and "Orden". Below the menu bar is a toolbar with icons for navigation and editing. The main area contains a form with the following fields and values:

Apellido Paterno:	PEREZ
Apellido Materno:	CORONADO
Nombres:	VICKY
Dirección Domicilio:	JUNIN 234
Dirección Trabajo:	AYACUCHO 430
Teléfono Domicilio:	5-6248
Teléfono Trabajo:	6-3258
Celular:	017-25895
e-mail:	vicky@hotmail.com
Fecha Nacimiento:	12/12/1966

Como se puede observar, en la aplicación se ha modificado el menú principal: Se ha quitado el menú *Salir* y se ha añadido el menú *Archivo*. Cuando se abre este menú presenta las siguientes opciones:



Donde, como recordará, la línea de división se crea simplemente colocando un guión (-) en la propiedad *Caption* de lo que vendría a ser el tercer *Item*. Después de esta línea se encuentra la opción *Salir* que en la anterior aplicación se encontraba como un menú.

Con el propósito de reducir el código que es necesario escribir en la aplicación, se han modificado los array índices y en lugar de emplear los 5 arrays: *Paterno*, *Materno*, *Nombres*, *Completo* y *Fecha* que se empleaban en registros, emplearemos ahora un array de arrays (*xi*). Con esta modificación es necesario cambiar también el tipo de la variable *Indice*, que ahora simplemente es de tipo entero. La declaración de estas variables (en el sector de implementación) es la siguiente:

```
xi : array [1..5] of ti; //Array de índices: 1=Paterno; 2=Materno;
                                3=Nombres; 4=Completo; 5=Fecha
Indice : LongInt; //índice actual elegido
```

Por supuesto deberán eliminarse las declaraciones de las variables *Paterno*, *Materno*, *Nombres*, *Completo* y *Fecha* así como reemplazar en los lugares que se empleaba el array *Paterno* por *xi[1]*, *Materno* por *xi[2]*, *Nombres* por *xi[3]*, *Completo* por *xi[4]* y *Fecha* por *xi[5]*.

Por otra parte en la declaración de variables se debe añadir:

```
f1 : file of TDirectorio;
f2 : array [1..5] of file of LongInt;
archivo : string;
```

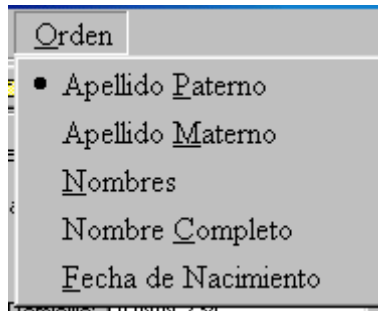
Donde *f1* es la variable archivo que se relacionará con el archivo que guarda los datos del directorio y *f2* son las variables archivos que se relacionarán con los archivos que guardan los índices. La variable *archivo* guardará el nombre del archivo de disco.

Los archivos se nombrarán de la siguiente manera: el archivo principal (el que contiene los registros del directorio) se guardará con la extensión *.dat*, los archivos índice se guardarán con el mismo nombre del archivo principal pero seguidos de un número (1 para el primer índice, 2 para el segundo, etc.) y tendrán la extensión *.idx*.

En el menú *Orden* es conveniente realizar la siguiente modificación: Ingresando al editor de menús seleccione todas las opciones del menú *Orden* y en el inspector de objetos coloque en verdadero (True) la propiedad *RadioItem*, con esta modificación el menú se comporta en forma similar a un *RadioGroup* y en consecuencia sólo es posible elegir una opción a la vez.

Para que una de las opciones aparezca seleccionada por defecto, se hace click en la opción y en el inspector de objetos se coloca la propiedad *Checked* en True. En la presente aplicación debe estar seleccionada por defecto la opción *Apellido Paterno*. Con esta modificación el menú tiene la apariencia que se muestra en la figura de la siguiente página, donde como se puede observar la opción elegida presenta un punto al lado izquierdo.

Al haberse llevado a cabo esta modificación y debido al cambio efectuado con los array índices los eventos *OnClick* de estas opciones cambian de la siguiente manera:



```
procedure TForm1.ApellidoPaterno1Click(Sender: TObject);
begin
    Indice := 1;
    ia := UbicarRegistro(xi[Indice],ra);
    ApellidoPaterno1.Checked := True;
end;
```

```
procedure TForm1.ApellidoMaterno1Click(Sender: TObject);
begin
    Indice := 2;
    ia := UbicarRegistro(xi[Indice],ra);
    ApellidoMaterno1.Checked := True;
end;
```

```
procedure TForm1.Nombres1Click(Sender: TObject);
begin
    Indice := 3;
    ia := UbicarRegistro(xi[Indice],ra);
    Nombres1.Checked := True;
end;
```

```
procedure TForm1.NombreCompleto1Click(Sender: TObject);
begin
    Indice := 4;
    ia := UbicarRegistro(xi[Indice],ra);
    NombreCompleto1.Checked := True;
end;
```

```
procedure TForm1.FechadeNacimiento1Click(Sender: TObject);
begin
    Indice := 5;
    ia := UbicarRegistro(xi[Indice],ra);
    FechadeNacimiento1.Checked := True;
end;
```

En todos estos procedimientos se cambia el valor de la variable *Indice*, se ubica el registro actual en el índice elegido y se hace visible la selección colocando la propiedad *Checked* en true.

El evento *OnCreate* de la forma cambia de la siguiente manera:

```
procedure TForm1.FormCreate(Sender: TObject);
var i : byte;
begin
    ra := 0; ia := 0; nr := 0; Indice := 1;
    NoModificado;
    Archivo := '';
    for i:=0 to 19 do
        Form1.Controls[i].Enabled := False;
```

end;

En este evento ya no se inicializan los arrays dinámicos pues el tamaño inicial de los mismos depende de que se abra un archivo existente o se cree uno nuevo. La variable *archivo* se deja inicialmente en blanco pues no existe archivo abierto y el ciclo *for* se emplea para inhabilitar los objetos de la forma con excepción del menú y la barra de herramientas.

Con la nueva declaración de los array índices, se puede cambiar también el procedimiento *ActualizarIndices* de la siguiente manera:

```
procedure ActualizarIndices;
var i : byte;
begin
  if nr=0 then
    for i:=1 to 5 do xi[i][0] := 0
  else
    begin
      ActualizarIndice(1,Paterno);
      ActualizarIndice(2,Materno);
      ActualizarIndice(3,Nombres);
      ActualizarIndice(4,Completo);
      ActualizarIndice(5,Fecha);
    end;
  end;
end;
```

Para que este procedimiento funcione se reemplazan los procedimientos *ActualizarPaterno*, *ActualizarMaterno*, *ActualizarNombres*, *ActualizarCompleto* y *ActualizarFecha* por el procedimiento *ActualizarIndice*:

```
procedure ActualizarIndice(ind: Integer; Campo:TCampo);
var i,j,k : LongInt; aux,elemento : string;
begin
  k := UbicarRegistro(xi[ind],ra);
  for i:=k to nr-1 do xi[ind][i]:=xi[ind][i+1];
  aux := Campo(x[ra]);
  j := nr-1;
  repeat
    elemento := Campo(x[xi[ind][j]]);
    if aux<elemento then xi[ind][j+1] := xi[ind][j]
    else break;
    dec(j);
  until j<0;
  xi[ind][j+1] := ra;
end;
```

En este procedimiento todos los datos son tratados como cadenas. El tipo *TCampo* (del parámetro *Campo*) debe ser declarado en el sector de implementación conjuntamente el tipo *ti*, su declaración es la siguiente:

```
TCampo = function (r:TDirectorio):string;
```

Como se puede observar, *TCampo* es un tipo función con un parámetro de tipo *TDirectorio* y un resultado de tipo *string*. Se emplea para obtener el campo (o campos) que se comparan al crear los diferentes índices. Como en la aplicación se emplean 5 índices, es necesario contar con 5 funciones de este tipo, estas funciones son:

```
function Paterno(r:TDirectorio):string;
begin
  Paterno := r.Nombre.Paterno;
end;
```

```

function Materno(r:TDirectorio):string;
begin
    Materno := r.Nombre.Materno;
end;

function Nombres(r:TDirectorio):string;
begin
    Nombres := r.Nombre.Nombres;
end;

function Completo(r:TDirectorio):string;
begin
    with r,Nombre do Completo := Paterno+' '+Materno+' '+Nombres;
end;

function Fecha(r:TDirectorio):string;
begin
    with r,f_nacimiento do Fecha := IntToStr(anio*10000+mes*100+dia);
end;

```

Cada una de estas funciones devuelve del registro *r*, en forma de cadena, el campo o combinación de campos que se comparan al obtener los índices por el método de inserción. El procedimiento *UbicarRegistro*, empleado en *ActualizarIndice*, debe igualmente ser modificado de la siguiente manera:

```

function UbicarRegistro(y:ti; reg:LongInt):LongInt;
var i : LongInt;
begin
    i := 0;
    while (y[i]<>reg) and (i<nr) do inc(i);
    UbicarRegistro := i;
end;

```

Donde básicamente se ha añadido una condición para controlar que no se pase el fin del registro.

El evento *OnClose* de la forma cambia de la siguiente manera:

```

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
    CerrarArchivos;
end;

```

Donde *CerrarArchivos* cierra, como su nombre sugiere, los archivos que hubieran sido abiertos. Su código es el siguiente:

```

procedure CerrarArchivos;
var i : byte;
begin
    if Archivo<>'' then
    begin
        CloseFile(f1);
        for i:=1 to 5 do CloseFile(f2[i]);
        LiberarArrays;
    end;
end;

```

En este procedimiento si el archivo ha sido creado o abierto (si la variable *Archivo* no está en blanco), se elimina la relación que existe entre los archivos de disco y las variables archivo, finalmente se liberan los array dinámicos. El procedimiento *LiberarArrays*, que libera la memoria ocupada por los arrays es el siguiente:

```

procedure LiberarArrays;
var i : byte;
begin
    Finalize(x);
    for i:=1 to 5 do Finalize(xi[i]);
end;

```

Para implementar la opción *Abrir* del menú *Archivo*, se añade a la forma el objeto *OpenDialog*, que está ubicado en la pestaña *Dialogs* y que tiene el siguiente icono:



Este objeto, que no es visible cuando la aplicación corre, permite mostrar el cuadro de diálogo estándar *Abrir* de *Windows*, de manera que resulte sencillo elegir el archivo que se quiere abrir.

Las propiedades de uso más frecuente en este objeto son:

Filter, esta propiedad permite definir el filtro que se empleará al abrir archivos, es decir él o los tipos de archivos que se mostrarán. El filtro puede ser escrito directamente o en el editor de filtros, el cual se activa haciendo *click* en los tres puntos (...) que se encuentran a la derecha de esta opción. Puesto que en la presente aplicación los archivos de directorios tendrán la extensión *.dat* el filtro a emplear es: *Archivo de directorios (*.dat)|*.DAT*, donde el texto a la izquierda de la barra (|) es la descripción de los archivos y el texto que se encuentra a la derecha es el filtro.

DefaultExt, esta propiedad define la extensión (las tres letras después del punto) que tendrá por defecto el archivo. Esta extensión se añade automáticamente al nombre del archivo. En nuestro caso la extensión es **.dat*.

Title, es el título que aparecerá en el cuadro de edición abrir, en nuestro caso colocaremos el título *Abrir archivos de directorio*.

InitialDir, es el directorio del cual se mostrarán los archivos, si se deja en blanco se muestran los archivos del directorio actual. En la presente aplicación puede especificar como directorio inicial el directorio en el cual se encuentra la aplicación, por ejemplo: *c:\sis101\archivos*.

El evento *OnClick* de la opción *Abrir* es el siguiente:

```

procedure TForm1.Abrir1Click(Sender: TObject);
var i : byte;
begin
    if Modificado then ConfirmarCambios;
    if OpenDialog1.Execute then
        begin
            if Archivo='' then
                for i:=0 to 19 do
                    Form1.Controls[i].Enabled := True;
                CerrarArchivos;
                Archivo := OpenDialog1.FileName;
                LeerArchivos;
            end;
        end;
end;

```

En este evento el código dentro de *if* se ejecuta sólo si el usuario elige un archivo y hace *click* en la opción *abrir*. Dentro de *if* se verifica si es la primera vez que se abre un archivo y en caso afirmativo se habilitan los cuadros de edición (mediante el ciclo *for*). Posteriormente se llama al procedimiento *CerrarArchivos*, se asigna el nombre del archivo elegido a la va-

riable *Archivo* y se lee el archivo de disco mediante el procedimiento *LeerArchivos*, cuyo código es el siguiente:

```
procedure LeerArchivos;
var i,j : byte; nom,arch : string;
begin
  AssignFile(f1,Archivo);
  Reset(f1);
  nr := FileSize(f1)-1;
  SetLength(x,nr+1);
  for i:=0 to nr do
    read(f1,x[i]);
  Nom := Copy(Archivo,1,Length(Archivo)-4);
  for i:=1 to 5 do
    begin
      Arch := Nom+IntToStr(i)+'.idx';
      AssignFile(f2[i],Arch);
      Reset(f2[i]);
      SetLength(xi[i],nr+1);
      for j:=0 to nr do
        read(f2[i],xi[i][j]);
    end;
  ra := nr;
  ia := UbicarRegistro(xi[Indice],ra);
  MostrarRegistro;
end;
```

En este procedimiento se asigna el archivo que contiene los datos del directorio a la variable *f1*, el número de registros existentes en el archivo se determina mediante la función *FileSize*. El número de elementos de los arrays dinámicos se fija en base al número de registros empleando la función *SetLength*. Los archivos índices se asignan a las variables *f2* (desde 1 a 5). Para generar los nombres de los archivos índice, se copia (con *copy*) el nombre del archivo principal (el directorio propiamente) sin su extensión, luego se le añade un número del 1 al 5 y finalmente se le añade la extensión *.idx*.

En este procedimiento se han empleado archivos con tipo, donde los datos deben ser leídos de disco uno por uno, habría sido más eficiente el empleo de archivos sin tipo, con los cuales podríamos haber leído todos los datos a la vez, la razón por la que no se ha procedido de esta manera es que los procedimientos *BlockRead* y *BlockWrite* no funcionan correctamente cuando se emplean arrays dinámicos como en el presente caso.

El procedimiento *ConfirmarCambios*, también ha sido modificado para que los cambios efectuados sean registrados en disco:

```
procedure ConfirmarCambios;
begin
  case
    MessageDlg('¿Quiere guardar las modificaciones efectuadas?',
      mtConfirmation,[MBYES,MBNO,MBCANCEL],0) of
    mrYes :
      begin
        LlenarRegistro(x[ra]);
        ActualizarIndices;
        GuardarCambios;
        ia := UbicarRegistro(xi[Indice],ra);
      end;
    mrCancel : Abort;
  end;
```

```
NoModificado;
end;
```

El procedimiento *GuardarCambios*, que es el encargado de guardar los cambios efectuados en disco tiene el siguiente código:

```
procedure GuardarCambios;
var i : byte; j : LongInt;
begin
  Seek(f1,ra);
  Write(f1,x[ra]);
  for i:=1 to 5 do
    begin
      Seek(f2[i],0);
      for j := 0 to nr do write(f2[i],xi[i][j]);
    end;
  end;
end;
```

En este procedimiento se busca en disco el registro modificado o añadido (empleando *seek*) y se escribe en esta posición el nuevo valor (*x[ra]*). En cuanto a los índices se escriben nuevamente todos los datos del archivo. Se ha adoptado esta alternativa por dos razones: a) Los archivos son pequeños (sólo contienen números) y b) No se ha almacenado en ninguna variable la posición en la cual ingresa el nuevo registro (o el registro modificado) en cada uno de los array índice. Una vez más recalcamos que en estos casos sería más eficiente el empleo de archivos sin tipo.

Para implementar el evento *OnClick* de la opción *Nuevo* se debe añadir el objeto *SaveDialog*. Al igual que *OpenDialog* este objeto se encuentra en la pestaña *Dialogs* y tiene el siguiente icono:



Este objeto permite presentar al usuario el cuadro de diálogo *Guardar Como*. *SaveDialog* funciona en forma similar a *OpenDialog* y ambos tienen las mismas propiedades. En la presente aplicación las propiedades de uso más frecuente de *SaveDialog1* se fijan igual que en *OpenDialog1*, excepto en la propiedad *Title* (título), que en *SaveDialog1* deberá ser cambiado a algo más adecuado como *Guardar Archivos de Directorio*.

El evento *OnClick* de la opción *Nuevo*, es el siguiente:

```
procedure TForm1.Nuevo1Click(Sender: TObject);
var i : byte;
begin
  if SaveDialog1.Execute then
    begin
      if Archivo='' then
        for i:=0 to 20 do Form1.Controls[i].Enabled := true
      else
        if Modificado then ConfirmarCambios;
        nr := 0; ra := 0; ia := 0;
        ReservarEspacio;
        CerrarArchivos;
        Archivo := SaveDialog1.FileName;
        CrearArchivos;
      end;
    end;
end;
```

Como se puede observar la lógica es similar a la del evento *OnClick* de la opción *Abrir*, sólo que en este caso se reserva espacio para los array dinámicos (mediante el procedimiento *ReservarEspacio*), se cierran los archivos

si están abiertos (mediante el procedimiento *CerrarArchivos*) y se crean los archivos con el nombre introducido en el cuadro de diálogo (mediante el procedimiento *CrearArchivos*).

El procedimiento *ReservarEspacio*, creado ya en la aplicación de registros, ha sido modificado para tomar en cuenta las nuevas declaraciones de los array índice:

```
procedure ReservarEspacio;
var i : byte;
begin
    SetLength(x,nr+10);
    for i:=1 to 5 do SetLength(xi[i],nr+10);
end;
```

El procedimiento *CrearArchivos*, que como su nombre sugiere permite crear los archivos de disco necesarios para manejar un directorio, es el siguiente:

```
procedure CrearArchivos;
var i : byte; Nom,Arch : string;
begin
    AssignFile(f1,Archivo);
    Rewrite(f1);
    Nom := Copy(Archivo,1,Length(Archivo)-4);
    for i:= 1 to 5 do
        begin
            Arch := Nom+IntToStr(i)+'.idx';
            AssignFile(f2[i],Arch);
            Rewrite(f2[i]);
        end;
    end;
```

El archivo principal, se crea directamente en base al nombre introducido por el usuario (almacenado en la variable *archivo*), mientras que los archivos índice se crean siguiendo la misma lógica que en el procedimiento *LeerArchivos*.

Para almacenar los cambios efectuados en disco se ha modificado también el evento *OnClick* de *ToolButton7* (Confirmar Cambios):

```
procedure TForm1.ToolButton7Click(Sender: TObject);
begin
    LLenarRegistro(x[ra]);
    ActualizarIndices;
    ia := UbicarRegistro(xi[Indice],ra);
    GuardarCambios;
    NoModificado;
end;
```

Igualmente se ha modificado el evento *OnClick* de *ToolButton6* (BorrarRegistro):

```
procedure TForm1.ToolButton6Click(Sender: TObject);
begin
    if
        MessageDlg('¿Desea eliminar todos los datos de este registro?',
            mtConfirmation,[MBYES,MBCANCEL],0) = mrYes
    then
        begin
            BorrarFormulario;
            if nr=0 then
```

```

        FillChar(x[ra],SizeOf(x[ra]),0)
    else
        begin
            EliminarRegistro;
            EliminarDeIndices;
            EliminarDeArchivo;
            if ia=nr then dec(ia);
            ra := xi[Indice][ia];
            dec(nr);
        end;
        MostrarRegistro;
        NoModificado;
    end;
end;

```

Donde el procedimiento *EliminarRegistro* ha sido también modificado de la siguiente manera:

```

procedure EliminarRegistro;
var i : LongInt;
begin
    for i:=ra to nr-1 do x[i] := x[i+1];
    FillChar(x[nr],SizeOf(x[nr]),0);
end;

```

Y el procedimiento *EliminarDeArchivo* que elimina el registro actual (ra) del archivo principal y de los archivos índices es el siguiente:

```

procedure EliminarDeArchivo;
var i,j : LongInt;
begin
    Seek(f1,ra);
    for i:=ra to nr-1 do write(f1,x[i]);
    truncate(f1);
    for i:=1 to 5 do
        begin
            Seek(f2[i],0);
            for j:=0 to nr-1 do write(f2[i],xi[i][j]);
            truncate(f2[i]);
        end;
    end;
end;

```

En el archivo principal, se ubica el archivo a eliminar (con *seek*) y a partir de esta posición se reemplaza el registro existente con el registro posterior, finalmente se elimina el registro sobrante con el procedimiento *truncate*. En los archivos índice se reemplazan todos los registros, menos el último, con los registros del array índice (el cual ya está actualizado).

Con estas modificaciones, la aplicación permite recuperar, guardar, borrar y modificar registros almacenados en archivos. Sin embargo y como se señaló en el anterior capítulo, la aplicación no está completa, no se tienen por ejemplo métodos de búsqueda.

23.7 EJERCICIOS

Modifique las aplicaciones del anterior capítulo de manera que permitan almacenar, recuperar, modificar y borrar los registros empleando archivos.