Introducción a la Programación usando Pascal como primer lenguaje

Capítulo 7
Bucles

Francisco J. Ballesteros

Bucles

7.1. Jugar a las 7½

El juego de las 7½ consiste en tomar cartas hasta que el valor total de las cartas que tenemos sea igual o mayor a 7½ puntos. Como mencionamos en el capítulo anterior, cada carta vale su número salvo las figuras (sota, caballo y rey), que valen ½ punto.

Ya vimos cómo implementar en un programa lo necesario para leer una carta e imprimir su valor. La pregunta es... ¿Cómo podemos implementar el juego entero? Necesitamos leer cartas e ir acumulando el valor total de las mismas hasta que dicho valor total sea mayor o igual a 7½. ¿Cómo lo hacemos?

Además de la secuencia o sentencias entre "**`begin`**'' y "**`end`**'' y la bifurcación condicional (todas las variantes de la sentencia *if*) y la sentencia *case*) tenemos otra estructura de control: los **bucles**.

Un bucle permite repetir una sentencia (o varias) el número de veces que sea preciso. Recordemos que las repeticiones son uno de los componentes básicos de los programas estructurados y, en general, son necesarios para implementar algoritmos.

Existen varias estructuras de control para implementar bucles. El primer tipo de las que vamos a ver se conoce como **bucle while** en honor a la palabra reservada **`while`** que se utiliza en esta estructura. Un *while* tiene el siguiente esquema:

```
inicializacion;
while condicion do begin  /* mientras ... repetir ... */
    sentencias;
end;
```

Donde *inicialización* es una o varias sentencias utilizadas para preparar el comienzo del bucle propiamente dicho, que es el código desde **`while`** hasta "**`end`**'' A la parse **`while...do`** se le llama normalmente **cabecera del bucle**. La cabecera del bucle determina si se ejecuta o no el cuerpo del bucle. Dicho cuerpo es el bloque de sentencias que sigue a la cabecera. Aunque nosotros usaremos siempre un bloque, es posible utilizar una única sentencia como cuerpo del bucle. En muchas ocasiones no es preciso preparar nada para el bucle y la inicialización se puede omitir. La *condición* determina si se ejecuta el cuerpo del bucle o no. En el caso del bucle *while*, el bucle se repite mientras se cumpla la condición, evaluando siempre dicha condición antes de ejecutar el cuerpo del bucle en cada ocasión. El flujo de control sigue el esquema mostrado en la figura 1.

Como de costumbre, el flujo de control entra por la parte superior del bucle y continúa

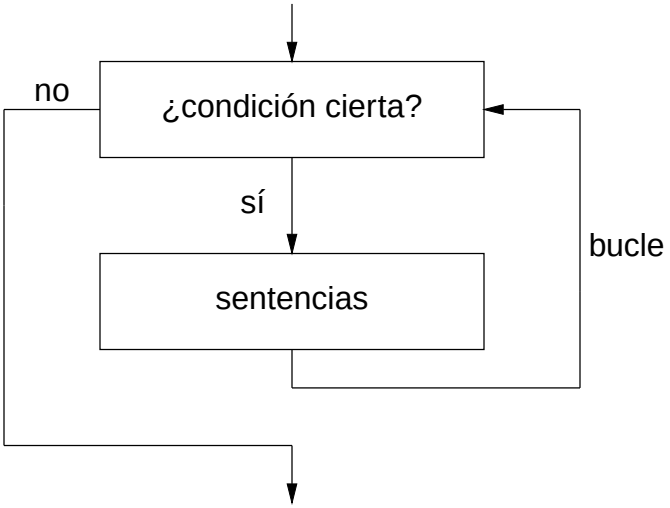

Fig. 1. Flujo de control en un bucle while.

(quizá dando algunas vueltas en el bucle) tras el mismo: un único punto de entrada y un único punto de salida. A ejecutar un bucle se lo denomina **iterar**. A cada ejecución del cuerpo del bucle se la suele denominar **iteración**. Aunque a veces se habla de *dar una pasada* del bucle, a pesar de que este término se considera muy coloquial (como podría resultar "tronco!'' para referirse a una persona).

Veamos ahora cómo jugar a las 7½. Inicialmente no tenemos puntos. Y mientras el número de puntos sea menor que 7½ tenemos que, repetidamente, leer una carta y sumar su valor al total de puntos:

```
puntos := 0.0;
while puntos < 7.5 do begin
    leercarta(c);
    puntos := puntos + valorcarta(c);
end;
```

Cuando el programa llega a la cabecera del *while* (a la línea del **`while`**) se evalúa la condición. En este caso la condición es:

```
puntos < 7.5
```

Si la condición es cierta, se entra en el bucle y se ejecuta el cuerpo del bucle. Una vez ejecutadas las sentencias que hay dentro del bucle se vuelve a la línea del **`while`**, y se vuelve a evaluar la condición de nuevo. Si la condición es falsa no se entra al bucle y se continúa con las sentencias escritas debajo del cuerpo del bucle. Pero si la condición es cierta se vuelve a repetir el bucle de nuevo.

Cuando se escribe un bucle es muy importante pensar en lo siguiente:

1. **¿Se va a entrar al bucle la primera vez?** Esto depende del valor inicial de la

condición.

2. **¿Qué queremos hacer en cada iteración?**(en cada ejecución del cuerpo del bucle).
3. **¿Cuándo se termina el bucle?** ¿Vamos a salir de el? Si nunca se sale del bucle, decimos que el bucle es un **bucle infinito**.

En la mayoría de los casos un bucle infinito es un error. Aunque en algunos casos es justo lo que se quiere.

En el ejemplo, la primera vez entramos dado que **`puntos`** está inicializado a 0.0, que es menor que 7.5. En cada iteración vamos a leer una carta de la entrada estándar y a sumar su valor, acumulándolo en la variable **`puntos`**. Como las cartas siempre valen algo, tarde o temprano vamos a salir del bucle: en cuanto tengamos acumuladas en **`puntos`** las suficientes cartas como para que la condición sea **`False`**.

Si **`puntos`** es menor que 7.5 y leemos y sumamos otra carta que hace que **`puntos`** sea mayor o igual que 7.5, no entraremos más al bucle. No leeremos ninguna carta más.

Comprobadas estas tres cosas parece que el bucle es correcto. Podríamos añadir a nuestro programa una condición para informar al usuario de si ha ganado el juego (tiene justo 7½ puntos) o si lo ha perdido (ha superado ese valor). Una vez terminado el bucle nunca podrá pasar que*puntos* sea menor , dado que en tal caso seguiríamos iterando.

```
puntos := 0.0;
while puntos < 7.5 do begin
    leercarta(c);
    puntos := puntos + valorcarta(c);
    escrcarta(c);
    writeln('puntos = ', puntos:0:2);
end;
if puntos > 7.5 then begin
    writeln('has perdido');
end
else begin
    writeln('has ganado');
end;
```

Este es el programa completo:

```
1
2 {
3     Jugar a las 7 y 1/2
4 }
5 program sieteymedia;
6
```

```
 7 type
 8     TipoPalo = (Oros, Bastos, Copas, Espadas);
 9     TipoValor = 1..12;
10     TipoCarta = record
11         palo: TipoPalo;
12         valor: TipoValor;
13     end;
14
15
16 procedure leercarta(var c: TipoCarta);
17 begin
18     read(c.valor);
19     read(c.palo);
20 end;
21
22 procedure escrcarta(c: TipoCarta);
23 begin
24     writeln(c.valor, ' de ', c.palo);
25 end;
26
27 function valorcarta(c: TipoCarta): real;
28 begin
29     if c.valor >= 10 then begin
30         result := 0.5;
31     end
32     else begin
33         result := c.valor
34     end;
35 end;
36
37 var
38     c: TipoCarta;
39     puntos: real;
40 begin
41     puntos := 0.0;
42     while puntos < 7.5 do begin
43         leercarta(c);
44         puntos := puntos + valorcarta(c);
45         escrcarta(c);
46         writeln('puntos = ', puntos:0:2);
47     end;
48     if puntos > 7.5 then begin
49         writeln('has perdido');
50     end
51     else begin
52         writeln('has ganado');
53     end;
54 end.
```

7.2. Contar

El siguiente programa escribe los números del 1 al 5:

```
numero := 1;
while numero <= 5 do begin
    writeln(numero);
    numero := numero + 1;
end;
```

Como puede verse, inicialmente se inicializa **numero** a **1**. Esto hace que se entre al **while**, dado que 1 es menor o igual que 5. Decir que se entra al bucle es lo mismo que decir que se ejecutan las sentencias del cuerpo del bucle: se imprime el número y se incrementa el número. Se sigue iterando (repitiendo, o dando vueltas) en el **while** mientras la condición sea cierta. ¡Por eso se llama **while**!

Se termina el bucle cuando el número pasa a valer 6, que no es menor o igual que 5. Fíjate que en la vuelta en la que el número pasa a valer 6, se escribe un 5 (ya que se escribe el número antes de incrementarlo). Después de esa iteración, se evalúa la condición, y como 6 no es menor o igual que 5, no se ejecuta el cuerpo del bucle y se continúa con las sentencias que hay después del bucle.

Es importante comprobar todo esto. Si el bucle hubiese sido

```
numero := 1;
while numero < 5 do begin
    writeln(numero);
    numero := numero + 1;
end;
```

no habríamos escrito el número 5, dado que incrementamos número al final del bucle y justo después comprobamos la condición. En este caso, en la última iteración se escribe el número 4.

Si hubiésemos programado en cambio

```
numero := 1;
while numero <= 5 do begin
    numero := numero + 1;
    writeln(numero);
end;
```

entonces no habríamos escrito el número 1.

Comprobando los tres puntos (entrada, qué se hace en cada iteración y cómo se sale) no debería haber problema; los errores introducidos por accidente deberían verse rápidamente.

Hay otro tipo de bucle que comprueba la condición al final de cada ejecución

del cuerpo, no al principio. Este bucle se suele conocer como **`repeat-until`** (en otros lenguajes encontramos una variante llamada **`do-while`**) y lo podemos escribir en Pascal como sigue:

```
repeat
    sentencias;
until condición;
```

En este caso se ejecutan las sentencias del cuerpo del bucle en primer lugar. ¡Sin evaluar condición alguna! Por último se evalúa la condición. Si esta es falsa, se termina la ejecución del bucle y se continúa tras él. Si esta es cierta, se sigue iterando.

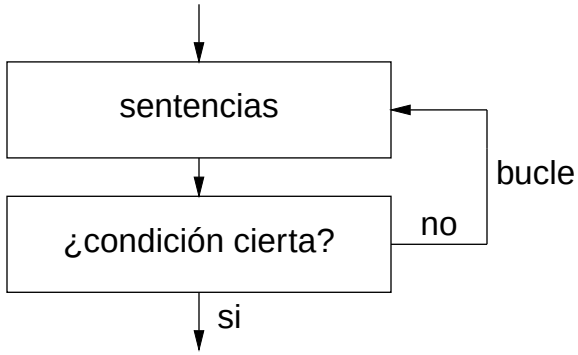

Fig. 2. Flujo de control en un bucle tipo repeat-until.

Por ejemplo, el siguiente bucle es más apropiado para escribir los números del uno al cinco, dado que sabemos que al menos vamos a escribir un número. Lo que es lo mismo, la primera vez siempre queremos entrar al bucle.

```
numero := 1;
repeat
    writeln(numero);
    numero := numero + 1;
until numero = 6;
```

Como puede verse, también ha sido preciso inicializar **`numero`** con el primer valor que queremos imprimir. Si en esta inicialización hubiésemos dado a **`numero`** un valor mayor o igual a 6 entonces habríamos creado un **bucle infinito**. Aunque no habría sido así si la condición del *repeat-until* hubiera utilizado "**`>=`**'' en lugar de "**`=`**'' ¿Ves por qué?

Hay que tener siempre cuidado de comprobar las tres condiciones que hemos mencionado antes para los bucles. Es muy útil también comprobar qué sucede en la primera iteración y qué va a suceder en la última. De otro modo es muy fácil que

iteremos una vez más, o menos, de las que precisamos debido a algún error. Si lo comprobamos justo cuando escribimos el bucle seguramente ahorremos mucho tiempo de depuración (que no suele ser un tiempo agradable).

Los dos tipos de bucle que hemos visto iteran un número variable de veces; siempre dependiendo de la condición: el bucle *while* comprobándola antes del cuerpo y el *repeat-until* comprobándola después.

Hay un tercer tipo de bucle que itera un número exacto de veces: ni una mas, ni una menos. Este bucle es muy popular dado que en muchas ocasiones sí sabemos el número de veces que queremos iterar. Nos referimos al bucle **for**.

El bucle *for* utiliza una variable, llamada **variable de control** del bucle, e itera siempre desde un valor inicial hasta un valor final. Por ejemplo, podría iterar de 1 a 5, de 0 a 4, etc. También puede iterar contando hacia atrás: de 5 a 1, de 10 a 4, etc. Dicho de otra forma, el bucle *for* itera haciendo que una variable tome los valores comprendidos en un rango.

La estructura del bucle *for* es como sigue:

```
for variable := valorInicial to valorFinal do begin
    sentencias;
end;
```

Igual que sucedía con el *while*, la parte *for...do* es la *cabecera* del bucle y el bloque de sentencias que sigue es el cuerpo del bucle. En Pascal podría ser una sóla sentencia, pero nosotros utilizamos un bloque.

La *inicialización* de la cabecera del *for* da un valor inicial a la variable de control del bucle. Por ejemplo,

```
for i := 1 to 10 do...
```

hace que *i* sea la variable de control del bucle y que empiece por el valor **1**. Dicho de otro modo, en la primera iteración, la variable **i** valdrá **1**.

Tras "**to**" indicamos el valor final, que ha de ser mayor o igual al inicial (o no ejecutará nunca el cuerpo del bucle).

Se verá claro con un ejemplo. Este bucle itera desde 1 hasta 5:

```
for i := 1 to 5 do begin
...
end;
```

Se suele leer este código diciendo: "desde **i** igual a 1 hasta 5, ...". Este bucle *for* ejecuta el cuerpo del bucle cinco veces. En la primera iteración **i** vale 1, en la segunda 2, y así hasta la última iteración, en que **i** vale 5.

También podemos escribir un bucle utilizando "**downto**" en lugar de "**to**" para contar hacia valores menores. Por ejemplo:

```
for i := 5 downto 1 do begin
...
end;
```

En este caso, el bucle va a ejecutar cinco veces. Pero cuenta desde 5 hasta 1.

La cabecera de un *for* hace que la variable se incremente (o decremente) automáticamente tras cada pasada del bucle, tras ejecutar el cuerpo del mismo.

Por decirlo de otro modo. Se suele decir que *la variable de control itera en un rango de valores*. Dicho rango está definido por la inicialización (el primer elemento del rango) y por la comparación (que indica el último elemento del rango).

La variable de control tiene que estar declarada antes del bucle, y no es preciso que sea un **`integer`** como en los ejemplos. Puede ser de cualquier tipo de datos **ordinal** (un tipo que se pueda contar). Por ejemplo, podemos utilizar enteros, caracteres, booleanos o cualquier otro tipo enumerado. Los bucles *for* ascendentes (de menor a mayor) hacen que la variable **`i`** pase a valer **`succ(i)`** tras cada iteración. Los descendentes hacen que la variable **`i`** pase a valer **`pred(i)`** tras cada iteración. Por ejemplo, este bucle imprime los caracteres que son letras minúsculas ('a'..'z'):

```
for c := 'a' to 'z' do begin
...
end;
```

Otra forma de escribir lo mismo es usar un rango y la palabra reservada "**`in`'' como en:

```
for c in 'a'..'z' do begin
...
end;
```

Este otro bucle imprime los números del 1 al 9:

```
for i in 1..9 do begin
    writeln(i);
end;
```

Igual que este:

```
for i := 1 to 9 do begin
    writeln(i);
end;
```

Este bucle escribe los números de 2 a 10 contados de dos en dos.

```
for i := 1 to 5 do begin
    writeln(2*i);
end;
```

Puede que no iteremos ninguna vez. Eso pasa en aquellas ocasiones en que el valor final está sobrepasado por el valor inicial.

Volvamos a ver nuestro ejemplo de antes. El siguiente bucle es el que

deberíamos escribir para imprimir los números del 1 al 5:

```
for i := 1 to 5 do begin
    writeln(i);
end;
```

Cuando vemos este bucle sabemos que el cuerpo ejecuta cinco veces. La primera vez **i** vale 1, la siguiente 2, luego 3, ... así hasta 5. Lo repetimos para que veas que este bucle es exactamente igual a este otro:

```
i := 1;
while i <= 5 do begin
    write(i);
    i := i + 1;    /* i = succ(i) */
end;
```

Pero el bucle *for* es más sencillo. Un bucle *for* se utiliza siempre que se conoce cuántas veces queremos ejecutar el cuerpo del bucle. En el resto de ocasiones lo normal es utilizar un *while*, salvo si sabemos que siempre vamos a entrar en el bucle la primera vez, en cuyo caso utilizamos un bucle *do-while*.

En muchas ocasiones la variable de control de un bucle *for* no se utiliza para nada mas en el programa. Para nada, salvo para conseguir que el bucle ejecute justo las veces que deseamos. Tal vez queramos hacer algo justo cinco veces, pero lo que hagamos sea siempre exactamente lo mismo, sin depender de cuál es valor de la variable cada vez. En tal caso, para hacer algo 5 veces, escribiríamos un bucle *for* como el de arriba, pero sustituyendo la invocación a **write** por las sentencias que queramos ejecutar 5 veces.

La variable de control no debe modificarse en el cuerpo del bucle *for*, ya que usamos un bucle de este tipo para poder iterar en un rango de valores definido en la cabecera del bucle, osea, para ejecutar una acción un número conocido de veces. El propio bucle incrementa o decrementa la variable de control para conseguir este comportamiento. Muchos lenguajes prohiben la asignación de valores a la variable de control dentro del cuerpo del bucle. ¿Qué hará Pascal? Del mismo modo, no debes depender de cuánto termina valiendo la variable de control una vez terminado el bucle.

Hay que tener en cuenta que una cosa son las sentencias que escribimos para controlar el bucle (cuándo entramos, cómo preparamos la siguiente pasada, cuándo salimos) y otra es lo que queremos hacer dentro del bucle. El bucle nos deja recorrer una serie de etapas. Lo que hagamos en cada etapa depende de las sentencias que incluyamos en el cuerpo del bucle.

Una última nota respecto a estos bucles. Es tradicional utilizar nombres de variable tales como **i**, **j** y **k** para controlar los bucles que iteran sobre enteros. Eso sí, cuando la variable de control del bucle realmente represente algo en nuestro programa (más allá de cuál es el número de la iteración) será mucho mejor darle un nombre más

adecuado; un nombre que indique lo que representa la variable. Aquí tienes mas ejemplos de bucles de este tipo.

```
 1
 2 {
 3     Jugar con bucles
 4 }
 5
 6 program bucles;
 7
 8 type
 9     TipoLetras = 'a'..'e';
10     TipoColor = (Rojo, Verde, Azul);
11 var
12     c: char;
13     col: TipoColor;
14
15 begin
16     for c := 'a' to 'e' do begin
17         writeln(c);
18     end;
19     for c in TipoLetras do begin
20         writeln(c);
21     end;
22     for col in TipoColor do begin
23         writeln(col);
24         { esto es ilegal: col := Rojo; }
25     end;
26     for col := low(TipoColor) to high(TipoColor) do begin
27         writeln(col);
28     end;
29     writeln(col);    { muy mala idea }
30 end.
```

7.3. Cuadrados

Queremos imprimir los cuadrados de los cinco primeros números positivos. Para hacer esto necesitamos repetir justo cinco veces las sentencias necesarias para elevar un número al cuadrado e imprimir su valor. Dado que queremos imprimir justo los cuadrados de los cinco primeros números podemos utilizar un bucle *for* que itere justo en ese rango y utilizar la variable de control del bucle como número para elevar al cuadrado. Este programa hace precisamente eso.

```
1 program cuadrados5;
```

```
 2
 3 const
 4     N: integer = 5;
 5 var
 6     i: integer;
 7
 8 begin
 9     for i := 1 to 5 do begin;
10         writeln(sqr(i));
11     end;
12 end.
```

Ahora hemos cambiado de opinión y deseamos imprimir los cuadrados menores o iguales a 100. Empezando por el 1 como el primer número a considerar. Esta vez no sabemos exactamente cuántas veces tendremos que iterar (podríamos hacer las cuentas pero es más fácil no hacerlo).

La estructura que determina el control sobre el bucle es el valor del cuadrado que vamos a imprimir. Hay que dejar de hacerlo cuando dicho cuadrado pase de 100. El programa completo podría quedar como sigue.

```
 1 {
 2     Cuadrados menores que 100
 3 }
 4
 5 program cuadrados;
 6
 7 const
 8     MaxCuad: integer = 100;
 9
10 var
11     num, cuad: integer;
12
13 begin
14     num := 1;
15     cuad := num * num;
16     while cuad < MaxCuad do begin
17         writeln(cuad);
18         num := num + 1;
19         cuad := num * num;
20     end;
21 end.
```

Podríamos haber escrito el bucle de este otro modo:

```
 1
 2 {
 3     Cuadrados de numeros menores que 100
 4 }
 5
 6 program cuadrados;
 7
 8 const
 9     MaxCuad: integer = 100;
10
11 var
12     num, cuad: integer;
13
14 begin
15     num := 1;
16     repeat
17         cuad := num * num;
18         if cuad < MaxCuad then begin
19             writeln(cuad);
20         end;
21         num := num + 1;
22     until cuad >= MaxCuad;
23 end.
```

Empezamos a contar en 1 y elevamos al cuadrado. Pasamos luego al siguiente número. Pero hemos de tener cuidado de no imprimir el cuadrado si hemos sobrepasado 100, por lo que es necesario un *if* que evite que se imprima un número cuando hemos pasado el límite.

Esta estructura, con un *if* para evitar que la acción suceda justo en la última iteración, es habitual si se utilizan bucles del estilo **`repeat-until`**.

7.4. Bucles anidados

Es posible anidar bucles (escribir uno dentro de otro) cuando queramos recorrer objetos que tengan varias dimensiones. Piensa que siempre es posible escribir sentencias dentro de sentencias compuestas tales como condicionales y bucles. Por ejemplo, el siguiente programa escribe los nombres de las casillas de un tablero de 10 x 10, donde la casilla (*i* , *j*) es la que está en la fila*i* y columna *j* .

```
 1
 2 {
 3     Matriz
 4 }
```

```
 5
 6 program matriz;
 7
 8 const
 9     DimX: integer = 4;
10     DimY: integer = 5;
11
12 var
13     i, j: integer;
14
15 begin
16     for i := 1 to DimX do begin
17         for j := 1 to DimY do begin
18             write(' [',i,',',j,']');
19         end;
20         writeln();
21     end;
22 end.
```

Esta es la salida de programa:

```
[1,1] [1,2] [1,3] [1,4] [1,5]
[2,1] [2,2] [2,3] [2,4] [2,5]
[3,1] [3,2] [3,3] [3,4] [3,5]
[4,1] [4,2] [4,3] [4,4] [4,5]
```

El bucle externo (el que comienza en la línea 16) se ocupa de imprimir cada una de las filas. Su cuerpo hace que se imprima la fila *i*. El b ucle interno se ocupa de imprimir las casillas de una fila. Su cuerpo se ocupa de imprimir la casilla *j*. Eso sí, su cuerpo hace uso del hecho de que estamos en la fila *i*. ¿P or qué hay un *writeln* tras el bucle más anidado?

7.5. Triángulos

Queremos dibujar un triángulo de altura dada como muestra la figura 3. Se trata de escribir empleando "*****" una figura en la pantalla similar al triángulo de la figura.

En este tipo de problemas es bueno dibujar primero (a mano) la salida del programa y pensar cómo hacerlo. Este es un ejemplo (hecho ejecutando el programa que mostramos luego):

```
*
**
***
****
*****
```

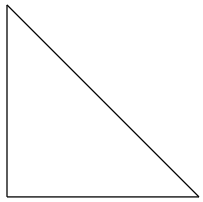

Fig. 3. Un triángulo rectángulo sobre un cateto.

Parece que tenemos que dibujar tantas líneas hechas con "*****" como altura nos dan (5 en este ejemplo). Por tanto el programa debería tener el aspecto

```
for fila := 1 to Alto do begin
    ...
end;
```

Donde *Alto* es una constante que determina la altura.

Fijándonos ahora en cada línea, hemos de escribir tantos "*****" como el número de línea en la que estamos. Luego el programa completo sería como sigue:

```
 1
 2 {
 3     Dibujar triangulo
 4 }
 5
 6 program triangulo;
 7
 8 const
 9     Tinta: char = '*';
10
11 procedure escrfilatriangulo(ancho: integer);
12 var
13     i: integer;
14 begin
15     for i := 1 to ancho do begin
16         write(Tinta);
17     end;
18     writeln();
19 end;
20
21 procedure escrtriangulo(alto: integer);
22 var
23     fila: integer;
24 begin
25     for fila := 1 to alto do begin
```

```
26          escrfilatriangulo(fila);
27      end;
28  end;
29
30  const
31      Alto: integer = 5;
32
33  begin
34      escrtriangulo(Alto);
35  end.
```

Ahora queremos dibujar dicho triángulo boca-abajo, como en la figura 4.

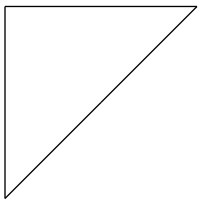

Fig. 4. Un triángulo invertido.

Podríamos pensar de nuevo como hacerlo. No obstante, la única diferencia es que ahora las líneas están al revés. Luego podríamos cambiar el programa anterior para que utilice un bucle de cuenta hacia atrás:

```
for fila := Alto downto 1 do begin
    ...
end;
```

El resultado de ejecutar este programa es como sigue:

```
*****
****
***
**
*
```

Aunque también es posible hacerlo de otro modo. El siguiente programa dibuja los dos triángulos.

```
1
```

```
 2 {
 3     Dibujar triangulo
 4 }
 5
 6 program triangulo;
 7
 8 const
 9     Tinta: char = '#';
10
11 procedure escrn(car: char; n: integer);
12 var
13     i: integer;
14 begin
15     for i := 1 to n do begin
16         write(car);
17     end;
18 end;
19
20 procedure escrfilatriangulo(ancho: integer);
21 var
22     i: integer;
23 begin
24     escrn(Tinta, ancho);
25     writeln();
26 end;
27
28 procedure escrfilatrianguloinv(fila, ancho: integer);
29 var
30     i: integer;
31 begin
32     escrn(Tinta, ancho-fila+1);
33     writeln();
34 end;
35
36 procedure escrtriangulo(alto: integer);
37 var
38     fila: integer;
39 begin
40     for fila := 1 to alto do begin
41         escrfilatriangulo(fila);
42     end;
43 end;
44
45 procedure escrtrianguloinv(alto: integer);
46 var
47     fila: integer;
48 begin
49     for fila := 1 to alto do begin
```

```
50          escrfilatrianguloinv(fila, alto);
51      end;
52  end;
53
54  const
55      Alto: integer = 5;
56
57  begin
58      escrtriangulo(Alto);
59      writeln();
60      escrtrianguloinv(Alto);
61  end.
```

Podríamos también centrar el triángulo sobre la hipotenusa, como muestra la figura 5. Este problema requiere pensar un poco más.

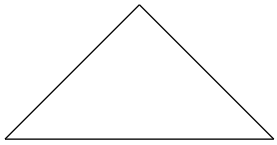

Fig. 5. Un triángulo sobre su hipotenusa.

De nuevo, hay que escribir tantas líneas como altura tiene el triángulo (que es el valor dado). Luego el programa sigue teniendo el aspecto

```
for fila := 1 to Alto do begin
    ...
end;
```

La diferencia es que ahora tenemos que escribir unos cuantos espacios en blanco al principio de cada línea, para que los "*****" queden centrados formando un triángulo. Después hay que escribir los "*****" para completar cada línea.

¿Cuántos espacios hay que escribir en cada línea? ¡Fácil! Hay que escribir un triángulo invertido de espacios en blanco. La primera linea tiene más, la siguiente menos... así hasta la última que no tendría ninguno.

¿Cuántos "*****" podemos escribir? Bueno, el problema sólo especificaba la altura. Lo más sencillo es imprimir el doble del número de línea menos uno. (En la primera línea uno, en la siguiente tres, etc.). Por lo tanto el programa queda de este modo:

```
1  {
2      Dibujar triangulo
```

```
 3 }
 4
 5 program triangulo;
 6
 7 const
 8     Tinta: char = '*';
 9     Papel: char = ' ';
10
11 procedure escrn(car: char; n: integer);
12 var
13     i: integer;
14 begin
15     for i := 1 to n do begin
16         write(car);
17     end;
18 end;
19
20 procedure escrfilatriangulo(fila, ancho: integer);
21 begin
22     escrn(Papel, (ancho-fila+1));
23     escrn(Tinta, 2*fila-1);
24     writeln();
25 end;
26
27 procedure escrtriangulo(alto: integer);
28 var
29     fila: integer;
30 begin
31     for fila := 1 to alto do begin
32         escrfilatriangulo(fila, alto);
33     end;
34 end;
35
36 const
37     Alto: integer = 5;
38
39 begin
40     escrtriangulo(Alto);
41 end.
```

Ejecutar el programa produce la siguiente salida:

```
    *
   ***
  *****
 *******
*********
```

Presta atención a como hemos inventado un procedimiento **escrn** para escribir n veces un carácter, y cómo el programa está dividido en problemas más simples (lo que hemos hecho conforme lo programamos, como siempre).

7.6. Primeros primos

Queremos imprimir los*n* primeros n úmeros primos. Lo que podemos hacer es ir recorriendo los números naturales e ir imprimiendo aquellos que son primos hasta tener *n*. El 1 es primo por definición, luego empezaremos a partir de ahí. Está todo hecho si pensamos en los problemas que hemos hecho, salvo ver si un número es primo o no. En tal caso, nos inventamos la función **esprimo** ahora mismo, lo que nos resuelve el problema. El programa tendría que tener el aspecto

```
var
    cuantos, n: integer;
begin
    cuantos := 0;
    n := 1;
    repeat
        if esprimo(n) then begin
            writeln(n);
            cuantos := cuantos + 1;
        end;
        n := n + 1;
    until cuantos = NumPrimos;
end.
```

Iteramos mientras el número de primos que tenemos sea menor que el valor deseado. Por tanto necesitamos una variable para el número por el que vamos y otra para el número de primos que tenemos por el momento.

¿Cuándo es primo un número? Por definición, 1 lo es. Además si un número sólo es divisible por él mismo y por 1 también lo es. Luego en principio podríamos programar algo como

```
function esprimo(n: integer): boolean;
var
    i: integer;
    loes: boolean;
begin
    loes := true;
    i := 2;
    while (i < n) and loes do begin
        if n mod i = 0 then begin
            loes := false;
```

```
            end
            else begin
                i := i + 1;
            end;
        end;
        result := loes;
    end;
```

Inicialmente decimos que es primo. Y ahora, para todos los valores entre 2 y el anterior al número considerado, si encontramos uno que sea divisible entonces no lo es.

Lo que sucede es que es una pérdida de tiempo seguir comprobando números cuando sabemos que no es un primo. Por tanto, podemos utilizar un *while* que sea exactamente como el *for* que hemos utilizado pero... ¡Que no siga iterando si sabe que no es primo! Por lo demás, el programa está terminado y podría quedar como sigue:

```
 1 {
 2     Escribir primeros primos
 3 }
 4
 5 program primos;
 6
 7 const
 8     NumPrimos: integer = 10;
 9
10 function esprimo(n: integer): boolean;
11 var
12     i: integer;
13     loes: boolean;
14 begin
15     loes := true;
16     i := 2;
17     while (i < n) and loes do begin
18         if n mod i = 0 then begin
19             loes := false;
20         end
21         else begin
22             i := i + 1;
23         end;
24     end;
25     result := loes;
26 end;
27
28 var
29     cuantos, n: integer;
```

```
30 begin
31     cuantos := 0;
32     n := 1;
33     repeat
34         if esprimo(n) then begin
35             writeln(n);
36             cuantos := cuantos + 1;
37         end;
38         n := n + 1;
39     until cuantos = NumPrimos;
40 end.
```

La salida del programa queda un poco fea, puesto que se imprime uno por línea. Podemos mejorar un poco el programa haciendo que cada cinco números primos se salte a la siguiente línea, y en otro caso separe el número con un espacio. Esto lo podemos hacer si modificamos el cuerpo del programa principal como sigue:

```
        cuantos := 0;
        n := 1;
        repeat
            if esprimo(n) then begin
                write(n);
                if cuantos mod 5 = 0 then begin
                    writeln();
                end
                else begin
                    write(' ');
                end;
                cuantos := cuantos + 1;
            end;
            n := n + 1;
        until cuantos = NumPrimos;
```

La idea es que cuando **`cuantos`** sea 5 queremos saltar de línea. Cuando sea 10 también. Cuando sea 15 también... Luego queremos que cuando **`cuantos`** sea múltiplo de 5 se produzca un salto de línea. Nótese que si este nuevo *if* lo ponemos fuera del otro *if* entonces saltaremos muchas veces de línea ¿Ves por qué?.

7.7. ¿Cuánto tardará mi programa?

Este último programa tiene dos bucles anidados. Por un lado estamos recorriendo números y por otro en cada iteración llamamos a **`esprimo`** que también recorre los números (hasta el que llevamos). Utilizar funciones para simplificar las cosas puede tal vez ocultar los bucles, pero hay que ser consciente de que están ahí.

En general, cuánto va tardar un programa en ejecutar es algo que no se puede saber. Como curiosidad, diremos también que de hecho resulta imposible saber automáticamente si un programa va a terminar de ejecutar o no. Pero volviendo al tiempo que requiere un programa... ¡Depende de los datos de entrada! (además de depender del algoritmo). Pero sí podemos tener una idea aproximada.

Tener una idea aproximada es importante. Nos puede ayudar a ver cómo podemos mejorar el programa para que tarde menos. Por ejemplo, hace poco cambiamos un **`for`** por un **`while`** precisamente para no recorrer todos los números de un rango innecesariamente. En lo que a nosotros se refiere, vamos a conformarnos con las siguientes ideas respecto a cuanto tardará mi programa:

- Cualquier sentencia elemental supondremos que tarda 1 en ejecutar. (Nos da igual si es un nanosegundo o cualquier otra unidad; sólo queremos ver si un programa va a tardar más o menos que otro). Pero... ¡Cuidado!, una sentencia que asigna un *record* de 15 campos tarda 15 veces más que asignar un entero a otro.
- Un bucle que recorre*n* elementos v a a tardar*n* unidades en ejecutar .
- Un bucle que recorre*n* elementos , pero que deja de iterar cuando encuentra un elemento, suponemos que tarda*n*/2 en ejecutar , por término medio.

Así pues, dos bucles *for* anidados suponemos que en general tardann^2. Tres bucles anidados tardaríann^3. Y así sucesivamente.

Aunque a nosotros nos bastan estos rudimentos, es importante aprender a ver cuánto tardarán los programas de un modo más preciso. A dichas técnicas se las conoce como el estudio de la **complejidad** de los algoritmos. Cualquier libro de algorítmica básica contiene una descripción razonable sobre cómo estimar la complejidad de un algoritmo. Aquí sólo estamos aprendiendo a programar de forma básica y esto nos basta: Lo más importante es que el programa sea correcto y se entienda bien. Lo siguiente más importante es que acabe cuanto antes. Esto es, que sea lo más eficiente posible en cuanto al tiempo que necesita para ejecutar. Lo siguiente más importante es que no consuma memoria de forma innecesaria.

7.8. Problemas

Recuerda que cuando encuentres un enunciado cuya solución ya has visto queremos que intentes hacerlo de nuevo sin mirar la solución en absoluto.

1. Calcular un número elevado a una potencia sin utilizar el operador de exponenciación de Pascal.
2. Escribir las tablas de multiplicar hasta el 11.
3. Calcular el factorial de un número dado.
4. Calcular un número combinatorio. Expresarlo como un tipo de datos e implementar una función que devuelva su valor.
5. Leer números de la entrada y sumarlos hasta que la suma sea cero.

6. Leer cartas de la entrada estándar y sumar su valor (según el juego de las 7 y media) hasta que la suma sea 7.5. Si la suma excede 7.5 el juego ha de comenzar de nuevo (tras imprimir un mensaje que avise al usuario de que ha perdido la mano).
7. Dibujar un rectángulo sólido con asteriscos dada la base y la altura.
8. Dibujar un rectángulo hueco con asteriscos dada la base y la altura.
9. Dibujar un tablero de ajedrez utilizando espacios en blanco para las casillas blancas, el carácter "**X**'' para las casillas negras, barras verticales y guiones para los bordes y signos "**+**'' para las esquinas.
10. Escribir un triángulo sólido en la salida estándar dibujado con asteriscos, dada la altura. El triángulo es rectángulo y está apoyado sobre su cateto menor con su cateto mayor situado a la izquierda.
11. Escribir en la salida estándar un triángulo similar al anterior pero cabeza abajo (con la base arriba).
12. Escribir un triángulo similar al anterior pero apoyado sobre la hipotenusa.
13. La serie de Fibonacci se define suponiendo que los dos primeros términos son 0 y 1. Cada nuevo término es la suma de los dos anteriores. Imprimir los cien primeros números de la serie de fibonacci.
14. Calcular los 10 primeros números primos.
15. Leer desde la entrada estándar fechas hasta que la fecha escrita sea una fecha válida.
16. Calcular el número de días entre dos fechas teniendo en cuenta años bisiestos.
17. Imprimir los dígitos de un número dado en base decimal.
18. Leer números de la entrada estándar e imprimir el máximo, el mínimo y la media de todos ellos.
19. Buscar el algoritmo de Euclides para el máximo común divisor de dos números e implementarlo en Pascal.
20. Escribir los factores de un número.
21. Dibuja la gráfica de una función expresada en Pascal empleando asteriscos.
22. Algunos de los problemas anteriores de este curso han utilizado múltiples sentencias en secuencia cuando en realidad deberían haber utilizado bucles. Localízalos y arréglalos.

Índice