Programación en Pascal. Bucles

Escuela Técnica Superior de Ingeniería de Telecomunicación
Universidad Rey Juan Carlos

gsyc-profes (arroba) gsyc.urjc.es

Diciembre de 2020

Contenidos

Bucles

Un bucle es una estructura que permite ejecutar una o más sentencias todas las veces necesarias. En Pascal, como en casi cualquier lenguaje de programación tenemos instrucciones para hacer bucles...

- Mientras se cumpla cierta condición

```
while CONDICION_DE_PERMANENCIA do
    SENTENCIAS
```

- Hasta que se cumpla cierta condición

```
repeat
    SENTENCIAS
until CONDICION_DE_SALIDA
```

- Un número predeterminado de veces

```
for VARIABLE := INICIO to FIN do
    SENTENCIAS
```

Antes de escribir un bucle, necesitamos tener claro:

- ¿Sabemos el número de veces que se ejecutará el bucle? (Es un valor en una variable, una constante, una expresión o una función)
  Sí $\Longrightarrow$ Lo más adecuado es `for`
  No:
  - ¿Se va a ejecutar la primera vez?
    Seguro que sí $\Longrightarrow$ Lo más adecuado es `repeat until`
    Tal vez no $\Longrightarrow$ La única opción es `while do`
- ¿Qué hay que hacer en cada iteración?
- ¿Cuál es la condición de salida?

Con `while do` podemos hacer cualquier tipo de bucle, aunque hay ocasiones donde `for` o `repeat until` es más adecuado (más sencillo)

Bucles while

Ejemplo de bucle while mínimo. Ejecutar algo 3 veces
(Más adelante veremos que `for` sería más adecuado, pero empezaremos por while, que sirve para cualquier tipo de bucle)

- Tendremos una variable que pasa por los estados 1, 2 y 3
- Como norma general las variables tienen que tener nombres completos y descriptivos, pero tradicionalmente en este caso se hace una excepción y se le suele llamar `i` (posiblemente por *indice*)
- Repite una sentencia mientras la condición de permanencia `(i <= 3)` sea cierta

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program while_00;
var
    i: integer;
begin
    i := 1;
    while i <= 3 do
        i := i + 1;
    writeln(i);   // Escribe 4
end.
```

- Este ejemplo es poco práctico, raramente repetiremos solamente una sola sentencia

Ejemplo más realista: repetir un bloque begin-end mientras la condición de permanencia sea cierta

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program while_01;
var
    i: integer;
begin
    i := 1;
    while i <= 3 do begin
        writeln( 'Probando bucles');
        i := i + 1;
    end;
end.
```

Resultado:

```
Probando bucles
Probando bucles
Probando bucles
```

En Pascal, la forma habitual de hacer algo `n` veces es contar desde 1 hasta n, como hemos visto
Pero también podemos contar desde 0 hasta n-1, que es equivalente (y lo habitual en otros lenguajes)

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program while_02;
var
    i: integer;
begin
    i := 0;
    while i < 3 do begin
        writeln( 'Probando bucles');
        i := i + 1;
    end;
end.
```

Resultado:

```
Probando bucles
Probando bucles
Probando bucles
```

- El error más habitual en este tipo de programas es confundir el 0 con el 1, el `n` con el `n-1`, el `<` con el `<=` ,etc

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program while_03;
var
    i: integer;
begin
    i := 1;
    while i < 3 do begin   // Cuidado, este son 2 ejecuciones
        writeln( 'Probando bucles');
        i := i + 1;
    end;
end.
```

1. Otro error frecuente es olvidar actualizar la condición de salida `i := i + 1`. En este caso el programa entraría en un *bucle infinito*. Tendríamos que abortarlo desde el terminal con `ctrl c`

- Observa que, con enteros, $i < 4 \Leftrightarrow i <= 3$.
  Podemos usar cualquiera de las dos expresiones

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program while_04;
var
    i: integer;
begin
    i := 1;
    while i < 4 do begin
        writeln( 'Probando bucles');
        i := i + 1;
    end;
end.
```

En los ejemplos anteriores hemos escrito `3` o `4` como *constantes literales*, esto es, como números *tal cual*

- Así, estos ejemplos han quedado más claros. Los humanos entedemos mejor `3` que `n`
- Pero en un programa normal no deberíamos hacerlo, esto sería un *número mágico*

Los números mágicos tienen (al menos) dos problemas

1. No queda claro *de dónde sale*. Ponerle un nombre (usar una constante) resulta más claro
2. Si necesitamos cambiarlo, basta modificar la definición de la constante, no es necesario buscarlo y cambiarlo por todo el código

El mismo bucle, sin números mágicos

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program while_05;
const
    N = 3;
var
    i: integer;
begin
    i := 1;
    while i <= N do begin
        writeln( 'Probando bucles');
        i := i + 1;
    end;
end.
```

Estos ejemplos donde la condición de permanencia en el bucle es la comparación de un contador con una constante, son buenos para ilustrar el uso de `while`

- Aunque veremos enseguida que `for` sería más adecuado

Para usar `while` de una manera más realista, necesitamos algo que el programador no conozca en el momento de escribir el código fuente.
Podría ser

- Una entrada del usuario
- La lectura de un fichero o cualquier otro dato
  - Realmente, para el programa leer la entrada del usuario es leer un fichero
- Una consulta al reloj
- Un número aleatorio
- ...

Generación de números aleatorios

En ocasiones necesitamos que un programa genere números aleatorios

- Un ordenador no es capaz de generar números verdaderamente aleatorios
- A menos que disponga de hardware específico para ello. Ejemplos: [1], [2], [3]

Normalmente nos basta usar números *pseudoaletorios*:

- A partir de un número inicial denominado *semilla*, de forma matemática se genera una serie de números casi aleatorios
- Es necesario cambiar la semilla en cada ejecución del programa, si no, la secuencia pseudoaleatoria siempre es la misma

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program random_00;
begin
    writeln(random);
    writeln(random);
    writeln(random);
end.
```

Este programa, como no cambia la semilla, siempre devuelve la misma serie

Para observar bien estos programas con números aleatorios, ejecutalos en tu ordenador. Puedes descargar todos los ejemplos en https://gsyc.urjc.es/~mortuno/fpi/tema07.zip

- La función `randomize` (sin argumentos) inicializa la semilla, usando la hora del sistema, en segundos
- La función `random` (sin argumentos) devuelve un número real pseudoaleatorio, $0 \leq numero < 1$

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program random_01;
begin
    randomize;  // Cambia la semilla del generador de números
                // pseudoaleatorios. Usa la hora, en segundos
    writeln(random);
end.
```

- Si necesitamos un número real entre 0 y n, basta multiplicar el valor que devuelve `random` por n
- Si necesitamos un entero, truncamos

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program random_02;
const
    N = 10;
begin
    randomize;

    writeln(random * N) ;
    // Nº real mayor o igual que 0, menor que N

    writeln( trunc( random * N) + 1 ) ;
    // Nº entero entre 1 y N
end.
```

Es muy común necesitar un número entero entre 0 y n-1,

- También se puede conseguir directamente pasando un argumento (**entero**) a `random()`
- `random(n)`, devuelve un número entero pseudoaleatorio, $0 \leq numero < n$

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program random_03;
const
    N = 6;
begin
    randomize;
    writeln( random(N) + 1);  // Número de 1 a 6
end.
```

Recuerda, esto es solo aplicable a los enteros. Para reales, `random * N`)

Para reutilizar este código, escribimos una función

```
1  {$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
2  program dado_01;
3
4  function tira_dado(caras_dado:integer):integer;
5  begin
6      result := random(caras_dado) + 1 ;
7  end;
8
9  const
10     Caras_dado = 6;
11 begin
12     randomize;
13     writeln( tira_dado(Caras_dado) );
14 end.
```

- En la línea 10 definimos la constante local `caras_dado`
- En la línea 13 llamamos a la función `dado` pasando 6 como argumento
- En el cuerpo de la función, línea 6, el parámetro `caras_dado` tiene el mismo nombre y el mismo valor (en este ejemplo) que la constante local `Caras_dado`, pero son dos cosas distintas

Un problema del uso de `randomize` se basa en la hora en segundos, por lo que todos los números generados el mismo segundo, tendrán la misma semilla

- Para evitarlo, usamos el procedimiento `delay()` que detiene la ejecución del programa el tiempo que indiquemos. Así nos aseguramos de que la semilla sea siempre distinta
  Si estamos seguros de que no vamos a pedir dos números aleatorios seguidos en el mismo segundo, no hace falta usar `delay()`
- Observa que tanto `delay` como `randomize` los ejecutamos una sola vez, en el cuerpo principal del programa. No en cada llamada a la función `tira_dado()`

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program dado_02;
uses crt;    // Necesario para delay

function tira_dado(caras_dado:integer):integer;
begin
    result := random( caras_dado ) + 1 ;
end;

const
    Caras_dado = 6;

begin
    delay(1000);   // 1000 milisegundos = 1 segundo
    randomize;
    writeln( tira_dado(Caras_dado) );
end.
```

Tiremos 3 dados

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program while_06;
uses crt;
function tira_dado(caras_dado:integer):integer;
begin
    result := random( caras_dado ) + 1 ;
end;

var    // Recuerda que si las variables se definen antes de
       // las funciones --> son globales --> suspenso seguro
    i: integer;
const
    Caras_dado = 6;
    N = 3;
begin
    delay(1000);
    randomize;
    i := 1;
    while i <= N  do begin
        writeln( tira_dado(Caras_dado));
        i := i + 1;
    end;
end.
```

Observa este problema:

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program while_07;
uses crt;

function tira_dado(caras_dado:integer):integer;
begin
    randomize;   //  ¡Mal!
    result :=  random( caras_dado ) + 1
end;

const
    Caras_dado = 6;
    N = 3;
var
    i: integer;
begin
    delay(1000);
    i := 1;
    while i <= N do begin
        writeln( tira_dado(Caras_dado));
        i := i + 1;
    end;
end.
```

Como en el parchís, tiramos dados hasta que salga 5. Este ya es un ejemplo donde es más adecuado `while` que `for`

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program salida_parchis_01; // Tira dados hasta que salga 5
uses crt;
function tira_dado(caras_dado:integer):integer;
begin
    result := random(  caras_dado ) + 1 ;
end;
const
    Caras_dado : integer = 6;
    Dado_salida : integer = 5;
var
    puntos: integer;
begin
    delay(1000);
    randomize;
    puntos := 0;  // Tenemos que iniciar puntos, con un valor
                  // que fuerce la primera ejecución del bucle
    while puntos <> Dado_salida do begin
        puntos := tira_dado(Caras_dado);
        writeln( puntos );
    end;
end.
```

Las chapas es un juego de azar donde los jugadores apuestan sobre el resultado del lanzamiento de dos monedas
https://es.wikipedia.org/wiki/Chapas_(juego_de_apuestas)

- Cada jugador apuesta por *cara* o *cruz* y lanza dos monedas
- Se considera *cara* si salen dos *caras*, se considera *cruz* si salen dos *cruces*
- En otro caso, se lanzan las monedas las veces que sean necesarias

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program chapas_01;
    // Tira dos monedas hasta que sean iguales
uses crt;

   // Definimos la función tira_dado() como en ejemplos anteriores

function tira_moneda(): string;
var
    valor : integer;
begin
    valor := tira_dado(2);
        // Permitimos este número mágico porque todas
        // las monedas tienen 2 caras
    if valor = 1 then
        result := 'cara'
    else
        result := 'cruz';
end;
```

```
var
    moneda1, moneda2 : string;
begin
    delay(1000);
    randomize;
    moneda1 := 'cara';  // Iniciamos la monedas, con un valor
    moneda2 := 'cruz';  // que fuerce la primera ejecución
    while moneda1 <> moneda2 do begin
        moneda1 := tira_moneda();
        moneda2 := tira_moneda();
        writeln( moneda1, ' ', moneda2);
    end;
end.
```

Bucles repeat

En los ejemplos del parchís y de las chapas, estábamos seguros de que era necesario ejecutar el bucle al menos una vez.
Antes de la sentencia while...

- Aún no teníamos ningun valor para los dados o las monedas
- Pero forzamos la condición de permanencia en el bucle para que la primera vez, siempre fuera cierto

Los bucles `repeat` son más cómodos para estos casos donde sabemos que el bucle se tiene que ejecutar al menos una vez, y evaluar luego la condición de salida

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program repeat_01;
var
    i: integer;

begin
    i := 1;
    repeat
        writeln( 'Bucle repeat');
        i := i + 1;
    until i = 4;
end.
```

Ejecución:

```
Bucle repeat
Bucle repeat
Bucle repeat
```

- Observa que `repeat` es la única sentencia que espera una lista de sentencias, no hace falta usar begin end

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program salida_parchis_repeat;
    // Tira dados hasta que salga 5
uses crt;


function tira_dado(caras_dado:integer):integer;
begin
    result := random(  caras_dado ) + 1 ;
end;

const
    Caras_dado: integer = 6;
var
    puntos: integer;
begin
    delay(1000);
    randomize;
    repeat
        puntos := tira_dado(Caras_dado);
        writeln( puntos );
    until puntos = 5;
end.
```

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program chapas_repeat;
    // Tira dos monedas hasta que sean iguales
uses crt;

// Definimos la función tira_moneda() como en los casos anteriores

var
    moneda1, moneda2 : string;

begin
    delay(1000);
    randomize;
    repeat
        moneda1 := tira_moneda();
        moneda2 := tira_moneda();
        writeln( moneda1, ' ', moneda2);
    until moneda1 = moneda2;
end.
```

Observa la versión de estos dos programas usando `repeat` y no `while do`

Ya no hace falta iniciar las variables antes del bucle, se les da valor dentro del cuerpo del bucle, antes de evaluar la expresión de salida

- La ejecución es equivalente
- Pero el código es más sencillo, más *elegante* $\Longrightarrow$ los errores del programador son menos probables

Observa también que

```
repeat
    sentencias
until condicion_salida
```

equivale a

```
{forzar condicion_salida = FALSE}
while not condicion_salida do begin
    sentencias
end
```

La condición de permanencia es un booleano. La condición de salida es otro booleano, la negación del anterior

- Ya sabemos aplicar doble negación, De Morgan, etc

En lenguaje natural sucede lo mismo
diviértete hasta que no puedas más
diviértete mientras puedas

Bucles repeat y lenguaje natural

No es raro que nos encontremos una especificación incorrecta en lenguaje natural parecida a la siguiente

```
Leer un valor
Si es erróneo, descartarlo y volver a leer
```

Una implementación ingenua, pero que cumple la letra de lo pedido sería

```
valor := lee_valor
if valor_erroneo(valor) then
    valor := lee_valor;
```

Difícilmente esto será lo deseado, puesto que si el segundo valor vuelve a ser erróneo, no se corrige.

Probablemente el autor de esa especificación esperaba que se sobreentendiera lo siguiente

```
Repite la lectura de un valor
    Hasta que no sea erróno
```

- Los *sobreentendidos* en ingeniería no son aceptables. Todo debe ser explícito
- Si (por algún extraño motivo) realmente se deseara corregir una vez y solo una, habría que dejarlo muy claro

Lectura desde teclado de número dentro de rango

Vamos a pedir al usuario que escriba un número entre 0 y 10

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program lectura_numero;

procedure lee_numero(var n:integer; limite_inf, limite_sup: integer);
var
    sal : boolean = FALSE;
    s : string;
    codigo : integer;
begin
    repeat
        write('Introduce un número entero entre ');
        write(limite_inf, ' y ' , limite_sup);
        writeln;
        readln(s) ;
        val(s, n, codigo);
        if (codigo = 0 ) and (n >= limite_inf) and (n <= limite_sup) then
            sal := True
        else
            writeln('Error, ',s,' no es un entero en el rango pedido');
    until sal;
end;
```

```
const
    LimiteInf = 0;
    LimiteSup = 10;
var
    numero: integer;

begin
    lee_numero(numero, LimiteInf, LimiteSup);
    writeln('Número indicado por el usuario: ',numero);
end.
```

Resultado:

```
Introduce un número entero entre 0 y 10
jj
Error, jj no es un entero en el rango pedido
Introduce un número entero entre 0 y 10
12
Error, 12 no es un entero en el rango pedido
Introduce un número entero entre 0 y 10
0
Número indicado por el usuario: 0
```

Observaciones

- El límite inferior y el superior se define en constantes locales al cuerpo del programa principal, que luego se pasan como parámetro al procedimiento
- El parámetro `n` se pasa por referencia, para que el valor generado dentro del procedimiento no se pierda al salir del procedimiento

Bucles for

Cuando se conoce de antemano el número de veces que se debe ejecutar el bucle, lo más adecuado es usar `for`

- Este *conocimiento* puede ser en el momento de escribir el programa, lo que se llama *en tiempo de compilación*. En ese caso, el número de iteraciones será una constante
- En otras ocasiones se conocerá al ejecutar el programa. Se llama *en tiempo de ejecución*. El valor será una variable, una expresión o el resultado de una función

```
for variable_de_control := valor_inicial to valor_final do
    sentencia;
```

o bien

```
for variable_de_control := valor_inicial to valor_final do begin
    sentencia_1
    sentencia_2
    ..
    sentencia_2
end
```

La variable de control puede ser de cualquier tipo *ordinal*, esto es, discreto, que tenga un número finito de elementos: enteros, caracteres, booleanos o cualquier enumerado

- La variable de control nunca puede sear real, intentarlo generaría un error de compilación

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program for_01;
const
    N = 3;
var
    i : integer;
begin
    for i := 1 to N do
        writeln( 'Bucle for');
end.
```

Resultado:

```
Bucle for
Bucle for
Bucle for
```

En un programa no debe haber *números mágicos*, pero considerar el número 1 como mágico seguramente es excesivo. Especialmente en Pascal, donde lo habitual es empezar a contar en 1 (y no en 0)

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program for_02;
const
    N = 3;
var
    i : integer;  // i, j, k son nombres tradicionales para índices:
                  // variables que iteran sobre enteros
begin
    for i := 1 to N do begin
        write( i, ' ' );
    end;
    writeln;  // Para añadir nueva línea al final
end.
```

Resultado:

```
1 2 3
```

```pascal
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program for_mal_01;
var
    i: real;  //  ¡Mal!  La variable ha de ser un tipo ordinal

begin
    for i := 1 to 3 do          // Error de compilación
                                // ordinal expression expected
        write( i, ' ' );
    writeln ;
end.
```

Para que el incremento de la variable de la variable de control sea negativo, en vez de `to` escribiremos `downto`

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program for_03;
const
    N = 3;
var
    i : integer;

begin
    for i := N downto 1 do
        write( i, ' ' );
    writeln;
end.
```

Resultado:

```
3 2 1
```

Si por error intentamos hacer un bucle decreciente sin usar *downto*, se ejecutará 0 veces

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program for_mal_02;
const
    N = 3;
var
    i: integer;

begin
    for i := N to 1 do
            // Error lógico. Se ejecuta 0 veces
        writeln( 'Holamundo de bucles for');
end.
```

Ejemplo con char:

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program for_04;
var
    c: char;

begin
    for c := 'a' to 'f' do
        write( c, ' ' );
    writeln;
end.
```

Resultado:

```
a b c d e f
```

Es muy habitual anidar bucles `for`

```
program matriz;
const
    Filas: integer = 4;
    Columnas: integer = 5;
var
    i, j: integer;
begin
    for i := 1 to Filas do begin
        for j := 1 to Columnas do
            write(' [',i,',',j,']');
        writeln();
    end;
end.
```

Resultado:

```
 [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]
```

- Observa que el primer bucle ahora tiene más de una sentencia, lo que obliga a usar un bloque begin-end

Máximo y mínimo

Busquemos el máximo y el mínimo de una serie de dados

- En una variable guardaremos el mínimo provisional. En otra, el máximo provisional
- En la primera iteración, el mínimo provisional será el valor obtenido, el único disponible. También será el máximo provisional
- En las siguientes iteraciones
  - Si el valor obtenido es menor que el mínimo provisional, ese valor pasará a ser el nuevo mínimo
  - Si el valor obtenido es mayor que el máximo provisional, ese valor pasará a ser el nuevo máximo
- Al final de todas las iteraciones, los valores provisionales pasan a ser definitivos

En una primera solución, usaremos dos sentencias condicionales anidadas

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program min_max_01;
uses crt;

function tira_dado(caras_dado:integer):integer;
begin
    result := random(caras_dado) + 1;
end;

procedure escribe_minmax(minimo,maximo:integer);
begin
    write('Valor mínimo: ',minimo);
    writeln(' Valor máximo: ',maximo);
end;
```

```
var
    i, resultado, minimo, maximo: integer;
const
    N = 6;
    CarasDado = 48;
begin
    delay(1000);
    randomize();
    for i := 1 to N  do begin
        resultado := tira_dado(CarasDado);
        write( resultado, ' ' );
        if i = 1 then begin
            minimo := resultado;
            maximo := resultado;
        end
        else begin
            if resultado > maximo then
                maximo := resultado;
            if resultado < minimo then
                minimo := resultado;
        end
    end;
    writeln;
    escribe_minmax(minimo,maximo);
end.
```

Podemos hacer una solución más elegante con un único condicional

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program min_max_02;
user crt;

// La función tira_dado(), el procedimiento escribe_minmax, la
↪  variables
// y las constantes son idénticas al ejemplo anterior
begin
    delay(1000);
    randomize();
    for i := 1 to N  do begin
        resultado := tira_dado(CarasDado);
        write( resultado, ' ' );
        if (i = 1) or (resultado < minimo) then
            minimo := resultado;
        if (i = 1) or (resultado > maximo) then
            maximo := resultado;
    end;
    writeln;
    escribe_minmax(minimo,maximo);
end.
```

Ejemplo: triángulo

Programa que escriba

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

- Bucle que se ejecuta tantas veces como la altura del triángulo
- Cada fila tiene tantos *puntos de tinta* como la variable de control

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program triangulo;
procedure escribe_fila(ancho: integer; tinta: string);
var
    i: integer;
begin
    for i := 1 to ancho do begin
        write(tinta);
    end;
    writeln();
end;
procedure escribe_triangulo(altura: integer; tinta: string);
var
    fila: integer;
begin
    for fila := 1 to altura do begin
        escribe_fila(fila, tinta);
    end
end;

const
    Alto = 5;
    Tinta: string = '* ';
begin
    escribe_triangulo(Alto, Tinta);
```

Ejemplo: triángulo invertido

Ahora queremos un programa que escriba

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

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

Para obtener la expresión del número de puntos de tinta del triángulo invertido, usamos la técnica habitual:

1. Poner los valores de unos cuantos casos particulares
2. Generalizar para `n`

```
Triangulo invertido, ancho 5

fila 1     5 puntos   (5 = 5-1+1)
fila 2     4 puntos   (4 = 5-2+1)
fila 3     3 puntos   (3 = 5-3+1)
fila 4     2 puntos   (2 = 5-4+1)
fila 5     1 puntos   (1 = 5-5+1)

                   ancho - fila + 1
```

https://gsyc.urjc.es/~mortuno/fpi/triangulos.pas

Ejemplo: triángulo isosceles

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

```
Espacios iniciales
fila 5    0 espacios  (5 - 5)
fila 4    1 espacios  (5 - 4)
fila 3    2 espacios  (5 - 3)
fila 2    3 espacios  (5 - 2)
fila 1    4 espacios  (5 - 1)
               altura - fila
```

```
Tinta
fila 1   1 puntos
fila 2   3 puntos
fila 3   5 puntos
fila 4   7 puntos
         (2 * fila) - 1
```

https://gsyc.urjc.es/~mortuno/fpi/triangulo_isosceles.pas