Introducción a la Programación usando Pascal como primer lenguaje

Capítulo 5
Acciones y procedimientos

Francisco J. Ballesteros

Acciones y procedimientos

5.1. Efectos laterales

Hasta el momento hemos utilizado constantes y funciones para realizar nuestros programas. Si programamos limpiamente, ambas cosas se comportarán como funciones para nosotros. Una constante es simplemente una función que no recibe argumentos y, como tal, siempre devuelve el mismo valor.

En este capítulo vamos a cambiar todo esto. Aunque parezca que a lo largo del capítulo estamos hablando de cosas muy diferentes, en realidad estaremos siempre hablando de acciones y variables.

Si nuestras funciones sólo utilizan sus parámetros como punto de partida para elaborar su resultado y no consultan ninguna fuente exterior de información ni producen ningún cambio externo en el programa (salvo devolver el valor que deban devolver), entonces las funciones se comportan de un modo muy similar a una función matemática (o a una constante, salvo por que el valor depende de los argumentos).

A esto se le denomina **transparencia referencial**. Esto es: cambiar en un programa una función y sus argumentos por el valor que produce esa función con esos argumentos no altera el resultado del programa.

Bastaría utilizar una sentencia **`write`** dentro de una función para romper la transparencia referencial. Por ejemplo, si la función **`media`** imprime un mensaje en lugar (o además) de devolver el valor correspondiente a la media de dos números, entonces ya no tendrá transparencia referencial. Ello es debido a que la función (con una llamada a **`write`**) produciría un efecto visible de forma externa (el mensaje). A eso se le denomina **efecto lateral** (Ver figura 1).

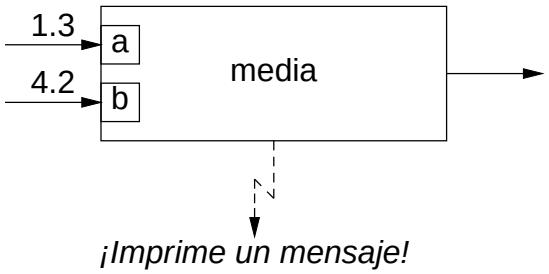

Fig. 1. En un efecto lateral una función provoca un cambio visible desde fuera.

Lo que importa de esto es que si tenemos una función que produce efectos

laterales (hace cosas que no debe) entonces *no* podremos olvidarnos del efecto lateral. Por ejemplo, si **`media`** escribe un mensaje, entonces no es igual llamar a esta función una que tres veces. ¡Pero debería dar igual si la queremos ver como una función matemática! Un fragmento de código como

```
if media(a, b) = 2 then
    ...
```

no parece imprimir ningún mensaje y, desde luego, no parece que cambie su comportamiento si volvemos a comprobar si la media es 3, por ejemplo, como en

```
if media(a, b) = 2 then
    ...
...
if media(a, b) = 3 then
    ...
...
```

El precio que pagamos por poner un efecto lateral es que tendremos que estar siempre pensando en qué hace dentro *media*, en lugar de poder olvidarnos por completo de ello y pensar sólo en lo que puede leerse viendo "**`media(a,b)`**" .

Además de las funciones, hay otro tipo de subprograma llamado **procedimiento**, o **`procedure`**, pensado justamente para producir efectos laterales. Esto es bueno, puesto que de otro modo todos nuestros programas serían autistas. Realmente ya conocemos varios procedimientos: **`write`** y **`writeln`** son procedimientos. Un procedimiento es básicamente una acción con un nombre, como veremos más adelante.

Pensando ahora en los datos, y no sólo en el código, además de las constantes tenemos otro tipo de entidades capaces de mantener un valor: las **variables**. Como su nombre indica, una variable es similar a una constante salvo porque podemos hacer que su valor varíe.

Al estilo de programación que hemos mantenido hasta el momento se lo denomina **programación funcional** (que es un tipo de programación **declarativa**). En este estilo estamos más preocupados por definir las cosas que por indicar cómo queremos que se hagan. Introducir procedimientos y variables hace que hablemos de **programación imperativa**, que es otro estilo de programación. En éste estamos más preocupados por indicar cómo queremos que se hagan las cosas que por definir funciones que lo hagan.

5.2. Variables

Un programa debe ser capaz de hacer cosas distintas cada vez que lo ejecutamos. Si todos los programas estuviesen compuestos sólo de funciones y constantes esto no sería posible.

El primer elemento que necesitamos es el concepto de **variable**. Una variable es en realidad un "nombre'' para un objeto o para un valor. Podemos crear cuantas

variables queramos. Para crear una variable hay que declararla indicando claramente su nombre (su identificador) y el tipo de datos para el valor que contiene. Las variables se declaran utilizando la palabra reservada "**`var`**" seguida de tantas declaraciones como queramos. Sólo podemos declarar variables para el cuerpo del programa principal o para el cuerpo de un procedimiento o función. En ambos casos vamos a hacer justo antes del "**`begin`**" del cuerpo principal. Por ejemplo:

```
var
    numero: integer;
```

declara una variable llamada **`numero`** de tipo **`integer`**. A partir del momento de la declaración, y hasta que termine la ejecución de la función o procedimiento donde se ha declarado la variable, dicha variable existirá como una cantidad determinada de espacio reservado en la memoria del ordenador a la que podemos referirnos simplemente mediante el nombre que le hemos dado (ver figura 2).

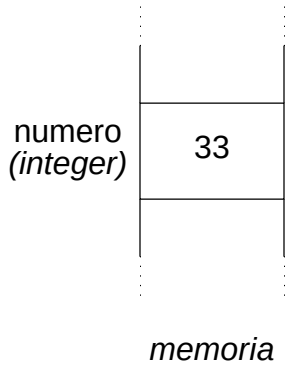

Fig. 2. Una variable es un nombre para una zona de memoria que guarda un valor de un tipo.

Como nombres de variable vamos a utilizar identificadores que están escritos en minúsculas, para distinguirlos de constantes. Es preciso utilizar **nombres cortos que resulten obvios** al verlos. Conviene que los nombres no sean muy largos para que no tengamos sentencias farragosas y difíciles de leer, pero tampoco hay que escatimar caracteres. El nombre más corto que indique de forma evidente a lo que se refiere es el mejor nombre posible. Es curioso, pero quienes no suelen programar (o no saben) normalmente insisten mucho en que los nombres de variables han de ser largos. Pero eso no es cierto. La realidad es que han de ser obvios. Cuanto más obvios y compactos, mejor.

Por ejemplo, si tenemos una variable en un subprograma que calcula el área de un círculo podríamos utilizar **`r`** como nombre para el radio, dado que resulta obvio lo que es **`r`** al verlo en una expresión como "**`Pi*sqr(r)`**" En este caso sería pésimo un

nombre algo como **valor_del_radio_del_circulo** dado que es mucho más largo de lo necesario y hará que las expresiones donde usemos la variable queden largas y pesadas. En cambio, si tenemos variables para el valor de una carta y el palo (de la baraja) de la misma en un subprograma, sería desastroso utilizar nombres como **v** y **p** para el valor y el palo respectivamente. No se puede esperar que nadie sepa qué es **v** y qué es **p**Lo lógico sería llamarles **valor** y **palo**.

Si en algún momento necesitamos variables para el número de peras en una cesta y para el número de manzanas en una cesta, podríamos declarar variables con nombres **numperas** y **nummanzanas**, que son claros. O tal vez **nperas** y **nmanzanas**. Pero si usasemos nombres como **n1** y **n2** o bien **np** y **nm** el código será ilegible. Si utilizamos nombres como **numerodeperasenlacesta** y **numerodemanzanasenlacesta** es muy posible que las sentencias sean también ilegibles (de lo largas que resultarán).

- Si tu compañero no sabe lo que son tus variables, entonces tus nombres están mal.
- Si tus nombres son tan largos que el código no te cabe en el editor y tienes que ensanchar la ventana, entonces tus nombres también están mal.

Si tienes tus nombres bien, te resultará muy fácil leer tu programa y será mucho más difícil que cometas errores al usarlos.

5.3. Asignación

Además de declararlas, hay dos operaciones básicas que podemos hacer con una variable: consultar su valor y cambiar su valor. Para consultar el valor de la variable basta con usar su nombre. Por ejemplo, si queremos escribir por la salida estándar el valor de la variable *numero*, podemos escribir esta sentencia:

```
write(numero);
```

Dicha sentencia **consulta** el valor de la variable **numero** y se lo pasa como argumento a **write**. Igualmente, si escribimos la expresión "**Pi*sqr(r)**" entonces Pascal consulta en la memoria el valor de **r** para calcular el valor de la expresión.

Para cambiar el valor que tiene una variable disponemos de una sentencia denominada **asignación**, que tiene la forma:

```
variable := valor;
```

Esta sentencia evalúa primero la expresión escrita a la derecha de "**:=**" y posteriormente guarda dicho valor en la variable cuyo nombre está a la izquierda de "**:=**" como por ejemplo, en:

```
numero := 3;
```

que cambia el valor de la variable **numero** y hace que en la memoria utilizada para almacenar dicha variable se guarde el valor 3, de tipo **integer** (el valor escrito en la parte derecha de la asignación).

A la izquierda de "**:=**" sólo pueden escribirse nombres de variable, dado que sólo tiene sentido utilizar la asignación para asignarle (darle) un valor a una variable

(guardar un valor en la variable). Por eso se dice que las variables son “L-values’’, o dicho de otro modo, que son elementos que pueden aparecer a la izquierda (la “L’’ es por “left’’) en una asignación.

A la derecha de “**:=**’’ sólo pueden escribirse expresiones cuyo valor sea del tipo de datos al que pertenece la variable. Esto es, una variable siempre tiene el mismo tipo de datos desde que nace (se declara) hasta que muere (su subprograma acaba).

Veamos lo que sucede si ejecutamos un programa que declara y utiliza una variable.

```
1 program vars;
2 var
3     numero: integer;
4 begin
5     writeln('voy a usar una variable');
6     numero := 3;
7     writeln('numero vale ', numero);
8 end.
```

Al ejecutar el programa se procesa primero la zona de declaraciones entre la cabecera del programa y su cuerpo. En este punto, al alcanzar la línea 3, se declara la variable **numero** y se reserva espacio en la memoria para mantener un entero. Una vez procesadas todas las declaraciones, Pascal ejecuta las sentencias del cuerpo del programa (líneas 4 a 8). En este punto la variable deja de existir dado que el programa donde se ha declarado ha dejado de existir.

Entre la línea 3 y la línea 6 *no sabemos cuanto vale la variable*. Esto quiere decir que *no debemos utilizar la variable* en ninguna expresión puesto que aún no le hemos dado un valor y el valor que tenga será lo que hubiese en la memoria del ordenador allí donde se le ha dado espacio a la variable. ¡Y no sabemos cuál es ese valor! Utilizar la variable antes de asignarle un valor tiene efectos impredecibles.

La línea 6 **inicializa** la variable. Esto es, le asigna un valor inicial. En este caso el valor 3 de tipo **integer**. A partir de este momento la variable guarda el valor 3 hasta que se le asigne alguna otra cosa.

La línea 7 consulta la variable como parte de la expresión **numero** que se utiliza en la sentencia **writeln**, que escribe el valor de la variable en la salida estándar.

Veamos otro ejemplo. Supongamos que tenemos el programa:

```
1 program vars;
2 var
3     x, y: integer;
4 begin
5     x := 3;
```

```
6     y := x + 1;
7     x := y div 2;
8  end.
```

La primera sentencia del cuerpo (línea 5) inicializa **x** a **3**. Antes de esta línea no sabemos ni cuánto vale **x** ni cuánto vale **y**, por lo que no deberíamos usarlas a la derecha de una asignación o como parte de una expresión. Pasada la línea 5, **x** está inicializada, pero **y** no lo está. La línea 6 evalúa "**x + 1**" (que tiene como valor 4, dado que **x** vale 3) y asigna este valor a la variable **y**, que pasa a valer 4. Igualmente, la línea 7 evalúa "**y div 2**" (cuyo valor es 2, dado que **y** vale 4) y asigna ese valor a **x**. El programa termina con **x** valiendo 2 y con **y** valiendo 4.

La siguiente secuencia de sentencias es interesante:

```
x := 3;
x := x + 1;
```

En la primera línea se le da el valor 3 a la variable **x**. En la segunda línea se utiliza **x** tanto en la parte derecha como en la parte izquierda de una asignación. Esto es perfectamente válido. **La asignación no es una ecuación matemática**. La asignación da un nuevo valor a una variable. El símbolo "**:=**" que vemos en la sentencia, es un símbolo de asignación que significa que a la parte izquierda se le da el valor de la parte derecha. No significa que la parte izquierda sea igual que la parte derecha. No hay que confundirlo con

```
x = x + 1
```

que es una expresión de tipo **boolean** cuyo valor es siempre **False** (puesto que **x** sólo puede tener un único valor a la vez).

Pero veamos lo que hace

```
x := x + 1;
```

Primero Pascal evalúa la parte derecha de la asignación: Esta es una expresión que calcula el valor de una suma de una variable **x** de tipo **integer** y de un literal "**1**" de tipo **integer**. El valor de la expresión es el resultado de sumar el valor de la variable y 1. Esto es, 4 en este ejemplo.

A continuación el valor calculado para la parte derecha de la asignación se le asigna a la variable escrita en la parte izquierda de la asignación. ¡Eso es justo lo que hace "**:=**" Como resultado, **x** pasa a tener el valor 4. Esto es,

```
x := x + 1;
```

ha incrementado el valor de la variable en una unidad. Si antes de la asignación valía 3, después valdrá 4. A esta sentencia se la conoce como**incremento**. Es una sentencia popular en variables destinadas a contar cosas (llamadas **contadores**).

Antes de seguir hay una cosa muy importante que es preciso decir. Una variable debería representar algún tipo de magnitud o entidad real y tener un valor que

corresponda con esa entidad. Por ejemplo, si hacemos un programa para jugar a las cartas y tenemos una variable **numcartas** que representa el número de cartas que tenemos en la mano, esa variable siempre debería tener como valor el número de cartas que tenemos en la mano, sin importar la línea del programa en la que consultemos la variable. De no hacerlo así, es fácil confundirse. Dicho de otro modo, las variables no deben mentir: si una variable es **numcartas**, debería tener siempre como valor el número de cartas que tenemos, ni una mas, ni una menos; sea cual sea la línea de código que estemos considerando.

5.4. Más sobre variables

Podemos tener variables de cualquier tipo de datos. Por ejemplo:

```
var
    c: char;
...

c := 'A';
```

declara la variable **c** de tipo **char** y le asigna el carácter cuyo literal es **'A'**

A partir de este momento podríamos escribir una sentencia como

```
c := chr(ord(c) + 1);
```

que haría que **c** pasase a tener el siguiente carácter en el código ASCII. Esto es, **B**. Esto es así puesto que, **c** tenía inicialmente el valor **A**, con lo que

```
ord(c)
```

es el número que corresponde a la posición de **A** en el juego de caracteres. Si sumamos una unidad a dicho número tenemos la siguiente posición en el juego de caracteres. Si ahora hacemos una conversión explícita a **char** para obtener el carácter con dicha posición, obtenemos el valor **B** de tipo **char**. Este valor se lo hemos asignado a **c**, con lo que **c** pasa a tener el valo **B** . Una forma más directa habría sido utilizar

```
c = succ(c);
```

que hace que **c** pase a tener el siguiente carácter de los presentes en el tipo **char**

En muchas ocasiones es útil utilizar una variable para almacenar un valor que el usuario escribe en la entrada estándar del programa. Así podemos hacer que nuestro programa lea datos de la entrada y calcule una u otra cosa en función de dichos datos.

Podemos pues inicializar una variable dándole un valor que leemos de la entrada el programa. Normalmente decimos que en tal caso **leemos** la variable de la entrada del programa (en realidad leemos un valor que se asigna a la variable). Esta acción puede realizarse llamando al procedimiento *read*. Por ejemplo, el siguiente programa lee un número y luego escribe el número al cuadrado en la salida.

```
 1 {
 2     jugar con variables
 3 }
 4 program vars;
 5
 6 var
 7     n: integer;
 8 begin
 9     read(n);
10     writeln(sqr(n));
11 end.
```

La sentencia **read** lee desde teclado un valor para la variable que se pasa como argumento. Osea, un entero en este caso. Una vez hecho eso, **read** asigna ese valor a la variable en cuestión. A partir de la línea 9 del programa tendremos en la variable **n** el número que haya escrito el usuario como entrada para el programa.

Hay otra sentencia para leer: **readln**. Esta es igual que **read** salvo porque, tras efectuar la lectura, se salta todo lo que exista hasta el final de la línea en la entrada.

Recuerda que algunos procedimientos y funciones predefinidos *built-in* en Pascal aceptan argumentos de distintos tipos (por ejemplo *write*, *writeln* y otros). Estos subprogramas son **polimórficos**. Ese es el caso de **read** y **readln**. Por ejemplo, si **x** es de tipo **char**

```
read(x);
```

lee un carácter, pero si **x** es de tipo **integer**, lee un entero. En nuestros programas no podemos definir subprogramas que actúen de esta forma. Eso sólo lo pueden hacer algunos **built-in** de Pascal. Salvo que aprendas más por tu cuenta sobre el compilador que usamos... ¡Inténtalo!

5.5. Ordenar dos números cualesquiera

El siguiente programa lee dos números de la entrada estándar y los escribe ordenados en la salida. Para hacerlo, se utilizan dos variables: una para almacenar cada número. Tras llamar dos veces a **read** para leer los números de la entrada (y almacenarlos en las variables), todo consiste en imprimirlos ordenados. Pero esto es fácil. Si el primer número es menor que el segundo entonces los escribimos en el orden en que los hemos leído. En otro caso basta escribirlos en el orden inverso.

```
1 {
2     ordenar 2 numeros
3 }
4 program vars;
```

```
 5
 6 uses math;
 7
 8 var
 9     n1, n2: integer;
10 begin
11     read(n1);
12     read(n2);
13     if n1 > n2 then begin
14         writeln(n2);
15         writeln(n1);
16     end
17     else begin
18         writeln(n1);
19         writeln(n2);
20     end;
21 end.
```

5.6. Procedimientos

Una función no puede modificar los argumentos que se le pasan, pero es muy útil hacer subprogramas (como **`read`**) que pueden modificar los argumentos o que pueden tener otros efectos laterales. Esos subprogramas se llaman **procedimientos** y en realidad ya los conocemos. El programa principal es, en realidad, un procedimiento. Tanto **`read`** como **`write`** son procedimientos.

Hace poco hemos conocido la sentencia de asignación. Esta y otras sentencias que hemos visto antes corresponden a acciones que realiza nuestro programa. Pues bien,

un procedimiento es una acción con nombre

Por ejemplo, **`write`** es en realidad un nombre para la acción o acciones que tenemos que realizar para escribir un valor. Igualmente, **`read`** es en realidad un nombre para la acción o acciones que tenemos que realizar para leer una variable.

Veamos un ejemplo. Supongamos que tenemos el siguiente programa, que lee un número e imprime su cuadrado realizando esto mismo dos veces.

```
1 {
2     cuadrados
3 }
4 program cuads;
5 uses math;    { para usar ** }
6 var
7     n: integer;
```

```
 8 begin
 9     read(n);
10     writeln(n ** 2);
11     read(n);
12     writeln(n ** 2);
13 end.
```

Como podemos ver, el cuerpo del programa realiza dos veces la misma secuencia:

```
read(n);
writeln(n ** 2);
```

Es mucho mejor inventarse un nombre para esta secuencia y hacer que el programa principal lo invoque dos veces. El resultado quedaría como sigue:

```
 1 {
 2     cuadrados
 3 }
 4 program cuads;
 5 uses math;    { para usar ** }
 6
 7 { leer un entero y escribir su cuadrado }
 8 procedure escrcuadrado();
 9 var
10     n: integer;
11 begin
12     read(n);
13     writeln(n ** 2);
14 end;
15
16 begin
17     escrcuadrado();
18     escrcuadrado();
19 end.
```

Aquí podemos ver que hemos definido un procedimiento llamado **`escrcuadrado`**. Esto es, nos hemos inventado un nombre para la acción consistente en imprimir el cuadrado de un número que se lee por teclado. Dicho nombre es **`escrcuadrado`**. Ahora, en el programa principal, podemos utilizar dicha acción sin mas que invocar su nombre (como el que invoca a un espíritu). ¡Podemos invocarlo cuantas veces queramos!

Las líneas 17 y 18 de nuestro programa invocan o llaman al procedimiento **`escrcuadrado`**. Es por eso que decimos, por ejemplo, que la línea 17 es una **llamada a procedimiento**. Como verás, la forma de definir un procedimiento es similar a la forma

de definir una función, salvo porque se emplea la palabra reservada **`procedure`** en lugar de **`function`** y por que un procedimiento nunca devuelve ningún valor. Y si nos fijamos... ¡El programa principal es como otro procedimiento más!

Si ejecutamos el programa sucederá lo de siempre: el programa comienza ejecutando en su programa principal, en la línea 16. Cuando el flujo de control del programa alcanza la línea 17, Pascal deja lo que estaba haciendo y llama al procedimiento **`escrcuadrado`** (igual que sucede cuando llamamos a una función). A partir de este momento se ejecuta el procedimiento **`escrcuadrado`**. Cuando se ejecuten las sentencias del procedimiento **`cuadrado`**, el procedimiento termina (decimos que **retorna**) y se continúa con lo que se estaba haciendo (en este caso, ejecutar el programa principal). Después, en la línea 18, sucede lo mismo. En ese punto se llama al procedimiento **`escrcuadrado`** de nuevo y se ejecutan las sentencias de dicho procedimiento. Cuando se acaban las sentencias del procedimiento, se retorna. Como esta es la última sentencia del programa principal, el programa acaba su ejecución. La figura 3 muestra esto de forma esquemática.

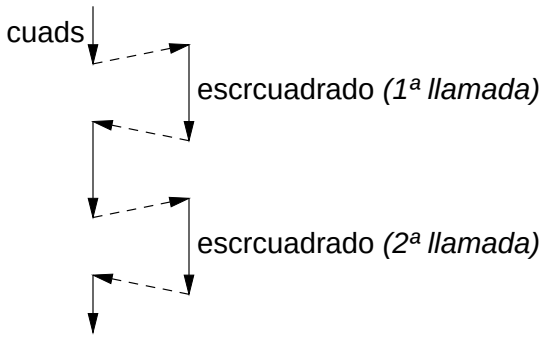

Fig. 3. llamada a procedimiento

Es bueno aprovechar ahora para ver algunas cosas respecto a la llamada a procedimiento: En la figura podemos ver cómo hay un único **flujo de control** (que va ejecutando una sentencia tras otra), representado por las flechas en la figura. Aun así, hay dos procedimientos en juego (tenemos el programa principal **`cuads`** y **`escrcuadrado`**). Cuando se llama a **`escrcuadrado`**, el procedimiento que hace la llamada deja de ejecutar hasta que el procedimiento llamado retorna. Podemos ver también que **`escrcuadrado`** no empieza a existir hasta que alguien lo llama. Pasa lo mismo con el programa principal, al que llama el sistema operativo cuando le pedimos que ejecute el p rograma (no existe hasta que lo ejecutamos). Podemos llamar al procedimiento más de una vez; y cada vez es una encarnación distinta (¿Será budista el sistema?). Por ejemplo, la variable **`n`** declarada en **`escrcuadrado`** no existe hasta la primera llamada, y deja de existir cuando esta termina. Además, la variable **`n`** que existe

en la segunda llamada es *otra variable distinta* a la que teníamos en la primera llamada. Recuerda que las declaraciones de un procedimiento sólo tienen efecto hasta que el procedimiento termina.

Esto mismo pasa cuando *escrcuadrado* llama a **`read`** y cuando **`escrcuadrado`** llama a **`writeln`**. La figura 4 muestra esto. Las llamadas a procedimiento se pueden **anidar** sin problemas (un procedimiento puede llamar a otro y así sucesivamente).

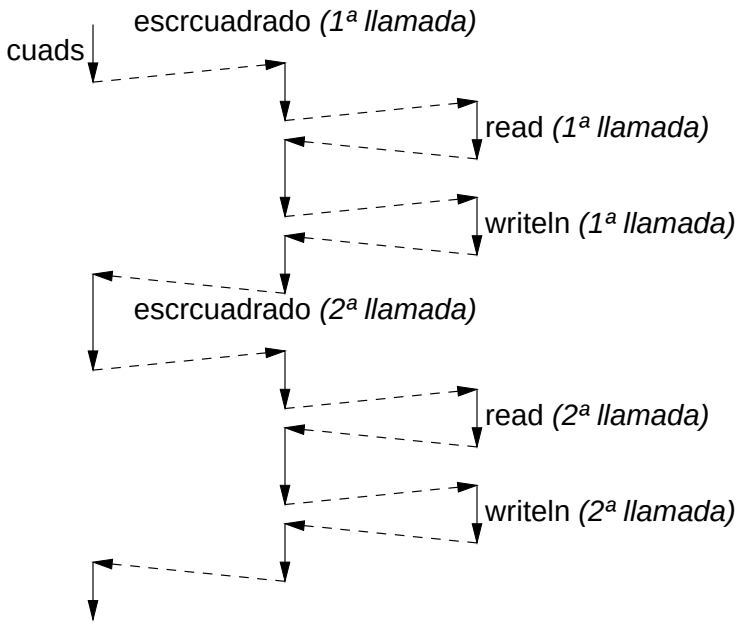

Fig. 4. Llamadas a procedimiento anidadas

Intenta seguir el flujo de control del programa mirando el código y prestando atención a la figura 4. Debería resultarte cómodo trazar a mano el flujo de ejecución de un programa sobre el código. Sólo tienes que seguir el rastro de las llamadas y recordar a dónde tienes que retornar cuando cada llamada termina. El ordenador utiliza una estructura de datos llamada **pila** para hacer esto de forma automática. Conforme se llaman a procedimientos y funciones, se apilan o comienzan a existir en la memoria las variables y datos que necesitan estos para ejecutar. Conforme se retorna de ellos, se desapilan (dejan de existir) de la memoria las variables y datos que necesitaban. También se guarda información extra en la pila, como por ejemplo en que dirección estaba ejecutando el programa cuando se hizo una llamada (o por dónde hay que seguir cuando esta termine).

5.7. Parámetros

Hemos visto cómo se producen las llamadas a procedimiento. Pero hemos omitido de la discusión el paso de parámetros.

Podemos declarar parámetros en los procedimientos del mismo modo que declaramos parámetros en una función. Por ejemplo, el procedimiento **escrnum** del siguiente programa escribe un entero tras un pequeño mensaje de tipo informativo y pasa a la siguiente línea tras escribir el entero. El programa principal lee dos números y escribe su suma.

```
 1 {
 2     escribir la suma de dos numeros
 3 }
 4 program sumar;
 5
 6 procedure escrnum(n: integer);
 7 begin
 8     write('el valor es ');
 9     write(n);
10     writeln();
11 end;
12
13 var
14     a, b: integer;
15     suma: integer;
16 begin
17     read(a);
18     read(b);
19     suma := a+b;
20     escrnum(suma);
21 end.
```

Las líneas 6 a 11 definen el procedimiento **escrnum**. Como este subprograma escribe en la salida estándar, tiene efectos laterales y ha de ser un procedimiento. Esta vez el procedimiento tiene un parámetro declarado, llamado **n**, para el que hay que suministrar un argumento (como sucede en la llamada de la línea 20 en el programa principal).

Cuando se produce la llamada a **escrnum** el procedimiento cobra vida. En este momento aparece una variable nueva: el parámetro **n**. El valor inicial de **n** es una copia del valor suministrado como argumento en la llamada (el valor de **suma** en el programa principal en el momento de realizar la llamada en la línea 20).

Cuando el procedimiento llega a su fin, el parámetro **n** (y cualquier variable que pueda tener declarada el procedimiento) deja de existir: su memoria deja de usarse y queda libre para otras necesidades futuras.

Una cosa curiosa es que podríamos haber llamado **suma** al parámetro del procedimiento (usando el mismo nombre que el de una variable declarada en el programa principal). En tal caso, nada varía con respecto a lo ya dicho: el parámetro del

procedimiento es una variable distinta a la variable empleada en el programa principal, una copia ¡Incluso si tienen el mismo nombre! Piensa que todo esto ya lo conoces: es exactamente lo mismo que sucede con los parámetros de las funciones.

Cuando declaramos parámetros de este modo, se copia el valor del argumento al parámetro, justo al principio de la llamada a subprograma. Pascal permite la modificación del valor del parámetro, pero esta modificación es local al procedimiento y no sale del mismo.

La figura 5 muestra el proceso de suministrar valores para los parámetros de un subprograma. A este proceso se lo conoce como **paso de parámetros por valor**, dado que se pasa un valor (una copia) como parámetro. Esto es, podemos utilizar este mecanismo para declarar e implementar un procedimiento como **`write`**, pero no podemos utilizarlo para implementar un procedimiento como **`read`** dado que el valor del argumento resultaría inalterado por la llamada.

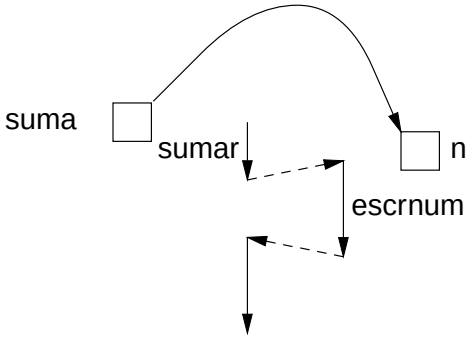

Fig. 5. Paso de parámetros por valor: se copia el valor del argumento al llamar.

Pero veamos ahora un procedimiento que incrementa el valor de una variable. El siguiente programa imprime **`2`** en su salida estándar.

```
 1 program incrementar;
 2
 3 procedure incr(var n: integer);
 4 begin
 5     n := n + 1;
 6 end;
 7
 8 const
 9     Num = 1;
10 var
11     x: integer;
12 begin
13     x := Num;
```

```
14      incr(x);
15      writeln(x);
16  end.
```

Comparando el procedimiento **incr** con el procedimiento **escrnum** programado antes, podrá verse que la principal diferencia es que el parámetro especificado en la cabecera tiene la palabra reservada **var** antes del nombre del parámetro.

Cuando un parámetro se declara como **var**, dicho parámetro corresponde en realidad a la variable suministrada como argumento. Esto quiere decir que **el procedimiento es capaz de modificar el argumento**. En nuestro programa de ejemplo, es como si el parámetro **n** fuese lo mismo que **numero** durante la llamada de la línea 16. La figura 6 muestra lo que sucede ahora durante la llamada: **n** se comporta como otro nombre para la variable **x** pasada como argumento.

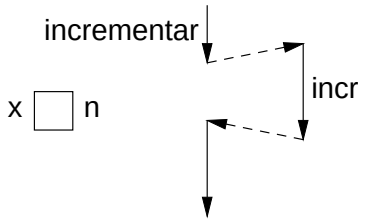

Fig. 6. Paso por referencia: durante la llamada el parámetro es lo mismo que el argumento

Como puede verse no se produce copia alguna de parámetros. En su lugar, resulta que el nombre **n** se refiere también al valor que almacena la variable **x** durante la llamada **incr(x)**Si el programa principal tuviese después una llamada de la forma **incr(y)**, entonces **n** sería lo mismo que **y** durante dicha llamada.

Dado que **n** se refiere en realidad al argumento que ha recibido (esto es, a **x**) la sentencia "**n := n + 1**'' en el procedimiento está en realidad incrementando **x**. A esta forma de pasar parámetros se la conoce como **paso por referencia**. A la forma de paso de parámetros que hemos utilizado hasta ahora se la conoce como **paso por valor**. Recapitulemos:

1. El paso por referencia permite modificar el argumento que se pasa. El paso por valor no.
2. El paso por referencia requiere una variable como argumento. El paso por valor no. (Cualquier expresión del tipo adecuado basta).
3. El paso por valor suele implicar la necesidad de copiar el valor utilizado como argumento. El paso por referencia no suele implicarlo. (Esto es importante si una variable ocupa mucha memoria).

En general, es bueno pensar que el paso de parámetros por referencia simplemente hace que los cambios al parámetro *se vean* tras el término del procedimiento; mientras que el paso de parámetros por valor hace que los cambios *sean locales* al procedimiento.

Por decir todo lo que hemos visto de otro modo: los parámetros que declaramos en las cabeceras de funciones y procedimientos son también variables. Si el paso es por valor, son variables que están declaradas automáticamente al comenzar la ejecución del procedimiento del que son parámetro, y son una copia del argumento que se ha pasado en la llamada, dejando de existir cuando el procedimiento (o función) termina.

Si el paso es por referencia, un parámetro es en realidad la variable pasada como argumento, no una copia. Cuando se termina un procedimiento, todo lo que se ha hecho sobre el parámetro tiene efecto en la variable pasada como argumento en la llamada.

Para no liarse, al principio, es conveniente dibujar en una hoja cajitas para las variables y parámetros y repasar mentalmente la ejecución del programa conforme se escribe, viendo qué valores se copian de qué variables a qué otras y qué variables se pasan por referencia.

En ocasiones hay que utilizar paso de parámetros por referencia en un procedimiento aún cuando no se desea modificar el parámetro. Cuando un parámetro tiene un tamaño considerable (si es una foto digital, por ejemplo) se suele utilizar paso de parámetros por referencia incluso si sólo se quiere consultar el parámetro. La razón es que, en general, el paso de parámetros por referencia evita hacer copia de los argumentos, consumiendo menos memoria.

En este curso seguiremos esta costumbre, de tal forma que en general si un parámetro se considera que tiene un tamaño excesivo se empleará **`var`** en su declaración, aunque sólo se desee consular éste.

Nótese que esto puede hacerse incluso en funciones, aunque *una función nunca debe tener parámetros por referencia* para evitar causar efectos laterales.

5.8. Variables globales

A las variables declaradas fuera de un subprograma se las denomina **variables globales** o **externas**. En este curso **no se permite el uso variables externas a un subprograma** (procedimiento o función), salvo cuando están declaradas justo antes del cuerpo del programa principal y se utilizan como variables locales del programa principal.

Naturalmente, en nuestros programas podemos utilizar constantes, funciones y procedimientos, y todo tipo de expresiones y estructuras de control (decisiones, etc.). Pero el diálogo entre el procedimiento y el resto del programa *debe quedar restringido a la cabecera del procedimiento*. Lo mismo queda dicho para las

funciones.

Veamos como queda el programa anterior si usa una variable global (evidentemente, este programa está mal escrito):

```
 1 {
 2     Mala idea! Y tiene un 0 como nota!
 3 }
 4 program incrementar;
 5
 6 var
 7     x: integer;
 8
 9 procedure incr();
10 begin
11     x := x + 1;
12 end;
13
14 const
15     Num = 1;
16 begin
17     x := Num;
18     incr();
19     writeln(x);
20 end.
```

En este caso, el procedimiento **`incr`** utiliza directamente la variable global **`x`**. Entonces, ese procedimiento sólo sirve para incrementar esa variable. ¡Ninguna otra!

Además, en programas mas largos sería fácil olvidar que un procedimiento está haciendo cosas más allá de lo que le permite su cabecera. De nuevo, eso hace que el programa tenga errores a no ser que seamos capaces de tener *todos* los detalles del programa en la cabeza. Y no lo somos.

Lo bueno de utilizar subprogramas y de hacer que un subprograma sólo pueda hablar con el resto del programa empleando su cabecera es que podemos **olvidarnos** por completo del resto del programa mientras programamos el subprograma. Como ya se ha visto al utilizar funciones, es muy útil emplear la idea del **refinamiento progresivo** y hacer el programa **top-down** (de arriba a abajo). Haciendo programas que llaman a otros más sencillos, que a su vez llaman a otros más sencillos, etc.

Programar consiste en inventar subprogramas que resuelven subproblemas del que tenemos entre manos, olvidándonos de cómo se hacen esos subproblemas inicialmente. Y luego, aplicar la misma técnica a los subproblemas hasta resolverlos. Aunque todavía no lo hemos dicho, también se hace lo mismo con los datos a la hora de organizarlos.

Las variables globales son útiles en ciertos casos (siempre que las use un programador experimentado) que quedan fuera del ámbito de este curso de programación. Y por cierto, los programadores experimentados intentan evitar el uso de variables globales.

Consideremos el siguiente programa.

```
 1 program ambitos;
 2
 3 {Nunca declarar variables aqui!. Esto es para jugar con ambitos}
 4 var
 5     n: integer; {1}
 6
 7 procedure incr(var n: integer);
 8 begin
 9     n := n + 1;
10 end;
11
12 procedure jugar(n: integer);
13 begin
14     incr(n);
15     writeln(n);
16 end;
17
18 begin
19     read(n);
20     incr(n);
21     writeln(n);
22     jugar(n);
23     writeln(n);
24 end.
```

Hay muchas variables y parámetros que se llaman **n**. Pero todas son distintas. Veamos. La variable que tiene un comentario marcándola como "**{1}**" existe durante la ejecución de todo el programa. Empieza a existir cuando se crean las variables del programa principal (justo antes de empezar a ejecutar el cuerpo del programa principal). Y sigue existiendo hasta que el flujo de control alcanza el final del programa principal, luego existe durante todo el programa. La llamada a **read** lee la variable de la que hablamos de la entrada.

Cuando llamamos a **incr** en la línea 20 aparece una **nueva** variable: el parámetro **n**. Esa nueva variable, también llamada **n**, es distinta de la variable declarada en la línea 5. En la llamada a **incr** la **n** del parámetro se refiere a la variable **n** del programa principal, por utilizar "**var**" en la declaración del parámetro (línea 7). Este parámetro comienza a existir al llamar a **incr** y dejará de existir cuando **incr** llegue a

su final.

Ahora seguimos tras retornar de la llamada en la línea 20, y llamamos a **writeln**. Si el usuario escribe un 1 a la entrada del programa y la línea 19 hizo que la variable **n** declarada en la línea 5 tuviera el valor 1, entonces **writeln** va a escribir "**2**'' en la salida.

LLamamos ahora al procedimiento **jugar** y compieza a existir otra variable, el parámetro de **jugar** declarado en la línea 12, llamado **n**. Esta vez, Pascal copia el valor de la variable **n** del programa principal al parámetro **n** del procedimiento **jugar**. Aunque esta nueva variable **n** se llama igual que las anteriores, es otra variable distinta. Ahora **jugar** llama a **incr** en la línea 14. Esta vez, aparece una nueva variable llamada **n** para el parámetro de **incr** que está declarado *por referencia*, con lo que el parámetro de la línea 7 ahora se refiere al argumento que se pasó (que es la **n** de **jugar**). Como el paso es por referencia, lo que **incr** haga con su **n** tiene efecto sobre la **n** de **jugar**. Cuando **incr** termine y la línea 15 llame a **writeln**, el programa escribirá 3, si a **jugar** se le pasó 2 como argumento.

Ahora bien, cuando **jugar** termine, el programa principal escribirá 2 en la línea 23 (suponiendo que ese fuese el valor de su variable al llamar a **jugar**).

El parámetro **n** de **incr** **oculta** la variable global del mismo nombre, dado que no puede haber dos variables que se llamen igual en el mismo bloque de sentencias. La variable global deja de ser visible hasta que el parámetro deje de existir.

Otra nota curiosa. Desde **jugar** podemos llamar a **incr**, pero no al contrario. Esto es así puesto que el código de **jugar** tiene declarado anteriormente el procedimiento **incr**, con lo que este procedimiento existe para el y se le puede llamar. En el punto del programa en que **incr** está definido no existe todavía ningún procedimiento **jugar**, por tanto no lo podemos llamar.

Se llama **ámbito** de una variable (o de un objeto en general) a la parte del programa en que dicha variable existe. Las variables declaradas en un procedimiento tienen como ámbito la parte del programa que va desde la declaración hasta el fin de ese procedimiento. Se dice que son **variables locales** de ese procedimiento. Cada llamada al procedimiento crea una copia nueva de dichas variables. Cada retorno (o fin) del procedimiento las destruye. Las variables externas se dice que son **variables globales** (y no se pueden usar durante este curso).

Otro concepto importante es el de **visibilidad**. Una variable es visible en un punto del programa si ese punto está dentro del ámbito de la variable y si además no hay ninguna otra variable que tenga la desfachatez de usar el mismo nombre y ocultarla. Por ejemplo, la global **n** de la línea 5 está dentro de su ámbito en el cuerpo del procedimiento **jugar** pero no es visible dado que el parámetro del procedimiento lo oculta. En este programa, la variable global tiene como ámbito el programa entero pero sólo es visible en el cuerpo del programa principal.

Declarar variables locales que oculten parámetros no es una buena práctica y

se debe evitar, aunque el lenguaje que estemos usando lo permita.

5.9. Ordenar puntos

Queremos ordenar dos puntos situados sobre una recta discreta, tras leerlos de la entrada estándar, según su distancia al origen. El programa debe escribir luego los puntos de menor a mayor distancia. Por ejemplo, si tenemos los puntos mostrados en la figura 7, el programa debería escribir en su salida estándar:

```
-2
3
```

Vamos a resolver este problema empleando las nuevas herramientas que hemos visto en este capítulo.

Fig. 7. Queremos ordenar dos puntos *a* y *b* según su distancia a *o*.

Sabemos que tenemos definido un punto con un valor entero. El problema consiste

1. leer dos puntos
2. ordenarlos de menor a mayor distancia
3. escribir ambos puntos.

Luego, suponiendo que tenemos disponibles cuantos procedimientos y funciones sean necesarios, podríamos escribir el programa como sigue:

```
{
    leer dos puntos y escribirlos ordenados por distancia
}
program ordpt;
    ...
var
    p1, p2: integer;
begin
    leerpunto(p1);
    leerpunto(p2);
    ordenarpordist(p1, p2);
    escrpunto(p1);
    escrpunto(p2);
end.
```

Hemos declarado dos variables locales en el procedimiento principal, **p1** y **p2**,

para almacenar los puntos. Después llamamos a un procedimiento **leerpunto** para leer el punto **p1** y de nuevo para leer el punto **p2**.

Como puede verse, para cada objeto que queremos que manipule el programa declaramos una variable (de tipo **integer** puesto que para nosotros un punto es un valor entero).

Igualmente, nos hemos inventado un procedimiento **leerpunto**, puesto que necesitamos leer un punto de la entrada. Debe ser un procedimiento y no una función, dado que corresponde a una acción, y además va la leer de la entrada, lo que es un efecto lateral. Al procedimiento hemos de pasarle una variable (para un punto) que debe quedar inicializada con el valor leído de la entrada. Ya veremos cómo lo hacemos.

Una vez tengamos leídos los dos puntos querremos ordenarlos (según sus distancias). De nuevo, suponemos que tenemos ya programado un procedimiento **ordenarpordist**, capaz de ordenar dos puntos. Para que los ordene, le tendremos que suministrar los dos puntos. Y supondremos que, tras la llamada a **ordenarpordist**, el primer argumento, **p1**, tendrá el punto con menor distancia y el segundo argumento, **p2**, tendrá el punto con mayor distancia. Como **p1** y **p2** son variables y no son constantes, no hay ningún problema en que cambien de valor. Para nosotros **p1** y **p2** son los dos puntos de nuestro programa, cuyo valor no se sabrá hasta que el programa ejecute y se lea de la entrada.

Nos resta escribir las distancias en orden, para lo que (de nuevo) nos inventamos un procedimiento **escrpunto** que se ocupe de escribir un punto.

Un detalle importante es que aunque nos hemos inventado estos subprogramas conforme los necesitamos, hemos intentado que sean siempre subprogramas útiles en general y no subprogramas que sólo hagan justo lo que debe hacer este programa. Por ejemplo, no hemos supuesto que tenemos un procedimiento **escrpuntos(a,b)** que escribe dos puntos. Sería raro que lo tuviéramos. Pero sí podría ser razonable que alguien hubiese programado antes un **escrpunto(a)** y lo pudiéramos utilizar ahora, por eso hemos supuesto directamente que ya lo tenemos.

Para leer un punto podríamos escribir un mensaje (para que el usuario sepa que el programa va a detenerse hasta que se escriba un número en la entrada y se pulse un *Enter*) y luego leer un entero. Ahora bien, el procedimiento **leerpunto** debe ser capaz de modificar su argumento. Dicho de otro modo, su parámetro debe pasarse por referencia. Sabiendo esto, podemos programar:

```
1 procedure leerpunto(var p: integer);
2 begin
3     write('punto? ');
4     readln(p);
5 end;
```

El procedimiento **readln** estará declarado de un modo similar, por eso es capaz de modificar **o** en la llamada de la línea 4.

El siguiente problema es cómo ordenar los puntos. Aquí tenemos dos problemas. Por un lado, el procedimiento debe ser capaz de modificar ambos argumentos (luego ambos deben pasarse por referencia). Por otro lado, tenemos que ver cómo los ordenamos.

Para ordenarlos, supondremos de nuevo que tenemos una función **`distancia`** que nos devuelve la distancia de un punto a un origen. Suponiendo esto, podemos ver si la distancia del primer punto es menor que la distancia del segundo punto. En tal caso ya los tendríamos ordenados y no tendríamos que hacer nada. En otro caso tendríamos que intercambiar ambos puntos para dejarlos ordenados. Pues bien, programamos justo esto:

```
procedure ordenarpordist(var p1, p2: integer);
begin
    if distancia(p1, Orig) > distancia(p2, Orig) then begin
        intercambiar(p1, p2);
    end;
end;
```

Ambos parámetros están declarados como "**`var`**" puesto que queremos modificar las variables que nos pasen como argumentos. Nos hemos inventado la constante **`Orig`** para representar el origen, la función **`distancia`**, y el procedimiento **`intercambiar`**. Este último ha de ser un procedimiento puesto que debe modificar sus argumentos y además no devuelve ningún valor.

La función **`distancia`** ya la teníamos programada antes, por lo que sencillamente la reutilizaremos en este programa (copiando su código).

El procedimiento **`intercambiar`** requiere más trabajo. Desde luego, va a recibir dos parámetros por referencia, digamos:

```
procedure intercambiar(var a: int; var b: int)
```

Pero pensemos ahora en su cuerpo. Podríamos utilizar esto:

```
a := b;
b := a;
```

Ahora bien, digamos que **`a`** vale 3 y **`b`** vale 4. Si ejecutamos

```
a := b;
```

dejamos en **`a`** el valor que tiene **`b`**. Luego ambas variables pasan a valer 4. Si ahora ejecutamos

```
b := a;
```

entonces dejamos en **`b`** el valor que *ahora* tenemos en **`a`**. Este valor era el valor que teníamos en **`b`**, 4. Luego hemos hecho que ambas variables tengan lo que hay en **`b`** y hemos perdido el valor que había en **`a`**.

La forma de arreglar este problema y conseguir intercambiar los valores es utilizar una **variable auxiliar**. Esta variable la vamos a utilizar sólo para mantener

temporalmente el valor de **a**, para no perderlo y actualizar **b** después. El procedimiento quedaría como sigue si usamos otros nombres para los parámetros:

```
procedure intercambiar(var p1, p2: integer);
var
    aux: integer;
begin
    aux := p1;
    p1 := p2;
    p2 := aux;
end;
```

Como vemos, la interfaz (la cabecera del procedimiento) sigue siendo la misma: recibe dos puntos y los ordena, dejándolos ordenados a la salida. Pero declaramos una variable local **aux** para mantener uno de los valores de forma temporal y en lugar de actualizar **p2** a partir de **p1** lo actualizamos a partir del valor que teníamos inicialmente en **p1**, esto es, a partir de **aux**.

Nos falta implementar **escrpunto**. Este subprograma debe ser un procedimiento dado que corresponde a algo que tenemos que hacer y no a un valor que tenemos que calcular. Por no mencionar que tiene efectos laterales. Dado que no precisa modificar su parámetro este ha de pasarse por valor y no por referencia. Como el procedimiento necesitará calcular la distancia para el punto podemos utilizar la función **distancia** que ya teníamos una vez más. El programa completo nos ha resultado como sigue.

```
 1
 2 {
 3     leer dos puntos y escribirlos ordenados por distancia a Origen
 4 }
 5 program ordpt;
 6
 7 const
 8     Orig: integer = 0;
 9
10 procedure leerpunto(var p: integer);
11 begin
12     write('punto? ');
13     readln(p);
14 end;
15
16 procedure escrpunto(p: integer);
17 begin
18     writeln('punto = ', p);
19 end;
20
21 function distancia(p1, p2: integer): integer;
22 begin
```

```
23      if p1 > p2 then begin
24          result := p1 - p2;
25      end
26      else begin
27          result := p2 - p1;
28      end;
29  end;
30
31  procedure intercambiar(var p1, p2: integer);
32  var
33      aux: integer;
34  begin
35      aux := p1;
36      p1 := p2;
37      p2 := aux;
38  end;
39
40  procedure ordenarpordist(var p1, p2: integer);
41  begin
42      if distancia(p1, Orig) > distancia(p2, Orig) then begin
43          intercambiar(p1, p2);
44      end;
45  end;
46
47  var
48      p1, p2: integer;
49  begin
50      leerpunto(p1);
51      leerpunto(p2);
52      ordenarpordist(p1, p2);
53      escrpunto(p1);
54      escrpunto(p2);
55  end.
```

5.10. Resolver una ecuación de segundo grado

Queremos resolver cualquier ecuación de segundo grado. Recordamos que una ecuación de segundo grado adopta la forma

$$ax^2 + bx + c = 0$$

y que, de tenerlas, la ecuación tiene en principio dos soluciones con fórmulas

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad y \quad \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

De nuevo, procedemos como de costumbre. Lo primero es ver cómo representar la ecuación y después nos inventamos cuantos subprogramas necesitemos. La primera tarea es fácil: para nosotros la ecuación son tres coeficientes:*a*, *b* y *c*; todos ellos números reales.

Luego nuestro programa debe leer la ecuación, ver si tiene solución, calcular la solución e imprimir la solución. Es justo esto lo que programamos por ahora.

```
 1 {
 2     Resolver ecuacion de 2 grado
 3 }
 4 program ec2;
 5
 6     ...
 7
 8 var
 9     a, b, c: real;    { ax2+xb+c=0 }
10     x1, x2: real;
11 begin
12     leerec(a, b, c);
13     if tienesol(a, b, c) then begin
14         solucionarec(a, b, c, x1, x2);
15         escrsol(a, b, c, x1, x2);
16     end
17     else begin
18         writeln('sin solucion');
19     end;
20 end.
```

Para manipular la ecuación estamos utilizando tres variables de tipo **`real`**, para almacenar los coeficientes. Dado que a estos coeficientes se los conoce como*a*, *b* y *c*, esos han sido los nombres para las variables. Por otro lado sabemos que tendremos (generalmente) dos soluciones. Para ello declaramos otras dos variables más, llamadas **`x1`** y **`x2`**.

Siguiendo nuestra costumbre, nos hemos inventado sobre la marcha los subprogramas **`leerec`**, para leer la ecuación, **`tienesol`**, que nos dirá cuándo existe solución (real, no compleja) a la ecuación, **`solucionarec`**, que calcula dicha solución (recordemos que estará compuesta de dos posibles valores para *x* en realidad) e **`escrsol`**, para que informe de la solución.

No importa si inicialmente hubiésemos programado algo como

```
leerec(a, b, c);
solucionarec(a,b, c, x);
escrsol(sol);
```

En cuanto programemos unas versiones, inicialmente vacías, de estos

procedimientos y empecemos a utilizar el programa nos daremos cuenta de dos cosas: (1) no siempre tenemos solución y (2) la solución son dos valores diferentes en la mayoría de los casos. En este punto cambiamos el programa para tener esto en cuenta y a otra cosa.

Leer la ecuación requiere tres parámetros, por referencia los tres.

Para ver si existe una solución necesitamos que a sea distinto de cero , puesto que de otro modo tendremos problemas al hacer la división por $2a$. Además, si queremos sólo soluciones reales necesitamos que b^2 sea mayor o igual que $4ac$.

El cálculo de la solución requiere tres parámetros cuyo valor necesitamos (los pasamos por valor) para definir la ecuación y dos parámetros que vamos a devolver (los pasamos por referencia).

Por último, imprimir la solución no requiere modificar los parámetros. Podríamos habernos molestado menos en cómo escribimos la solución, pero parecía adecuado escribir un único valor si ambas soluciones son iguales. Nótese que el código común en ambas ramas del *if* en esta función se ha extraído de dicha sentencia, para no repetirlo.

El programa queda como sigue:

```
 1
 2 {
 3     Resolver ecuacion de 2 grado
 4 }
 5 program ec2;
 6
 7 uses math;
 8
 9 const
10     Eps: float = 0.00001;
11
12 procedure leerec(var a, b, c: real);
13 begin
14     read(a);
15     read(b);
16     read(c);
17 end;
18
19 function discriminante(a, b, c: real): real;
20 begin
21     result := b ** 2 - 4.0 * a * c;
22 end;
23
24 function tienesol(a, b, c: real): boolean;
25 begin
26     result := (discriminante(a, b, c) >= 0.0) and (a <> 0.0);
```

```
27 end;
28
29 procedure solucionarec(a, b, c: real; var x1, x2: real);
30 begin
31     x1 := (-b + sqrt(discriminante(a, b, c))) / (2.0 * a);
32     x2 := (-b - sqrt(discriminante(a, b, c))) / (2.0 * a);
33 end;
34
35 procedure escrsol(a, b, c: real; x1, x2: real);
36 begin
37     writeln(a:0:2, '*x**2 + ', b:0:2, '*x +', c:0:2, ' = 0');
38     if abs(x1-x2) < Eps then begin
39         writeln('    x = ', x1:0:2);
40     end
41     else begin
42         writeln('    x1 = ', x1:0:2);
43         writeln('    x2 = ', x2:0:2);
44     end;
45 end;
46
47 var
48     a, b, c: real;    { ax2+xb+c=0 }
49     x1, x2: real;
50 begin
51     leerec(a, b, c);
52     if tienesol(a, b, c) then begin
53         solucionarec(a, b, c, x1, x2);
54         escrsol(a, b, c, x1, x2);
55     end
56     else begin
57         writeln('sin solucion');
58     end;
59 end.
```

Los programas realmente no se terminan de hacerse nunca y siempre pueden mejorarse. Por ejemplo, podríamos hacer que el programa supiese manipular soluciones complejas. No obstante hay que saber cuando decir basta.

5.11. Problemas

Cuando el enunciado corresponda a un problema ya hecho te sugerimos que lo vuelvas a hacer sin mirar la solución.

1. Leer un número de la entrada estándar y escribir su tabla de multiplicar.
2. Hacer esto mismo sin emplear ningún literal numérico salvo el "**1**'' en el programa. Sugerencia: utilizar una variable para mantener el valor del factor que varía en la tabla de multiplicar y ajustarla a lo largo del programa.

3. Leer dos números de la entrada estándar y escribir el mayor de ellos.
4. Intercambiar dos variables de tipo entero.
5. Leer tres números de la entrada estándar y escribirlos ordenados.
6. Suponiendo que para jugar a los barcos queremos leer una casilla del tablero (esto es, un carácter de la **A** a la **K** y un número del 1 al 10). Haz un programa que lea una casilla del tablero y que imprima las casillas que la rodean. Supongamos que la casilla que leemos no está en el borde del tablero.
7. Haz un programa que lea una casilla del juego de los barcos y diga si la casilla está en el borde del tablero o no.
8. Haz que el programa hecho en el ejercicio 6 funcione correctamente si la casilla está en el borde del tablero.
9. Haz un programa que lea un dígito hexadecimal y lo convierta a decimal.
10. Haz el mismo programa pero para números de cuatro dígitos.
11. Haz un programa que lea la coordenada del centro de un círculo y valor para el radio del círculo y luego la coordenada de un punto y determine si el punto está dentro del círculo.
12. Haz un programa que escriba **DOD** en letras grandes empleando "#'' como carácter para dibujar las letras. Cada letra ha de escribirse bajo la letra anterior, como en un cartel de anuncio vertical.
13. Haz un programa que escriba **DODDODDOD** con las mismas directrices del problema anterior.
14. Cambia el programa anterior para que el usuario pueda indicar por la entrada qué carácter hay que utilizar para dibujar la letra **D** y qué carácter hay que utilizar para dibujar la letra **O**.
15. Si no lo has hecho, utiliza procedimientos y funciones de forma intensiva para resolver todos los ejercicios anteriores de este curso.
16. Haz que los ejercicios anteriores de este curso que realizan cálculos para valores dados sean capaces de leer dichos valores de la entrada.

Índice