

# Programación en Pascal. Procedimientos

Miguel Ortuño

Escuela Técnica Superior de Ingeniería de Telecomunicación  
Universidad Rey Juan Carlos

Septiembre de 2022



© 2022 Miguel Angel Ortuño Pérez.  
Algunos derechos reservados. Este documento se distribuye bajo la  
licencia *Atribución-CompartirIgual 4.0 Internacional* de Creative  
Commons, disponible en  
<https://creativecommons.org/licenses/by-sa/4.0/deed.es>

## 1 Procedimientos

- Introducción a los procedimientos
- Variables
- Procedimientos
- Paso de parámetro por referencia
- Procedimiento val
- Concatenación de cadenas
- Ámbito de las variables

Como hemos visto en el tema 3, una función bien escrita no puede tener efectos laterales, esto es, tiene que tener *transparencia referencial*

- Tiene que devolver su resultado a partir de los parámetros de entrada, y nada más
- No puede tener ningún efecto en ningún sitio, no puede modificar nada, no puede escribir nada en pantalla

## Función mal escrita:

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }  
program sin_transparencia_referencial;  
  
function media(a,b: integer): real;  
begin  
    result := (a + b ) / 2;  
    writeln('La media de ',a , ' y ' ,b, ' es ', result); // ¡MAL!  
end;  
  
const  
    A = 2;  
    B = 3;  
  
begin  
    writeln( media(A,B));  
end.
```

## Función bien escrita:

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }  
program transparencia_referencial;  
  
function media(a,b: integer): real;  
begin  
    result := (a + b ) / 2  
end;  
  
const  
    A = 2;  
    B = 3;  
  
begin  
    write('La media de ',A );  
    write(' y ', B);  
    writeln(' es ',media(A,B):0:3);  
end.
```

En este curso, hasta ahora solo hemos usado constantes y funciones sin efectos laterales. Por tanto, todas las ejecuciones del programa producían exactamente el mismo resultado. A esto se le denomina *programación funcional*

Esto es lo más adecuado en muchas ocasiones, pero si los programas solo tuvieran este comportamiento, estarían muy limitados. Cuando resulte conveniente, usaremos dos elementos más:

- Variables
- Procedimientos

# Variables

Una constante, como su nombre indica, es un valor que permanece inalterado durante toda la ejecución de un programa. Una variable es un nombre para un valor que podrá cambiar durante la ejecución

- Al igual que las constantes, las variables se declaran (indicamos su nombre y tipo) y se definen (les damos valor)
- La declaración y la definición de una variable es prácticamente igual que la declaración y definición de una constante, pero con la palabra reservada `var` en vez de `const`

```
var  
  marca, modelo: string;  
  cilindrada : integer;
```

- Las constantes se pueden declarar y definir o solamente definir, sin declarar. Pero las variables se declaran y definen siempre



Las variables se declaran en la parte declarativa de los bloques, junto a las constantes, inmediatamente antes de la parte de las sentencias (el begin-end)

En otras palabras, podemos declarar variables:

- Antes del begin de una función o un procedimiento
- Antes del begin del cuerpo del programa principal

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }  
program variable;  
var  
    a: integer;  
  
begin  
    a := 12;  
    writeln(a);  
end.
```

Al igual que en la declaración de constantes

- Después de `var` no hay *punto y coma*
- Hay un *punto y coma* al final de cada declaración

En estos casos usamos identificadores como *a*, *b*, *c*, etc porque son ejemplos *que no hace nada*. En un ejercicio no tan trivial, usaríamos nombres descriptivos: P.e. *importe*, *velocidad*, *hora*, *dni*, etc

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }
program variables01;

function suma(a,b: integer):integer;
var
    c: integer;    // Variable local de la función suma
begin
    c := a + b;
    result := c;
end;

const    // Constantes locales al programa principal
    X = 12;
    Y = 3;

var    // Variables locales al programa principal
    z : integer;

begin
    z := suma(X, Y);
    writeln(z);    // 15
end.
```

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }
program variables02;

function suma(a,b: integer):integer;
var
    c: integer;    // Variable local de la función suma
begin
    c := a + b;
    result := c
end;

var    // Variables locales al programa principal
    x,y,z : integer;

begin
    x := 4;
    y := 9;
    z := suma(x,y);
    writeln(z);    // 13

    x := 2;    // Podemos cambiar el valor de las variables.
    y := 6;    // No es necesario definir nuevas constantes.
    z := suma(x,y);
    writeln(z);    // 8
end.
```

El programador es responsable de inicializar todas las variables (darles un valor inicial)

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }  
program variable_no_inicializada;  
  
var  
    a: real;  
  
begin  
    writeln(a);    // ¡¡MUY MAL!!    No hemos inicializado la variable  
end.
```

- Si lo olvidamos, el compilador no dará un error. Como mucho un aviso

Warning: Variable "a" does not seem to be initialized

Una variable se puede inicializar (puede recibir su primer valor) o bien en la declaración, o bien en el cuerpo del programa / funcion / procedimiento

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }  
program inicializacion_variable;  
  
var  
    a: real = 3;  
    b : real ;  
  
begin  
    b := 4.2 ;  
    writeln(a);  
    writeln(b);  
end.
```

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }  
program uso_de_variables;  
  
var  
    x, y : integer;  
  
begin  
    x := 3;  
    y := x + 1 ;  
    x := y div 2 ;    // 4 div 2  
    writeln(x);        // 2  
    writeln(y);        // 4  
end
```

# Asignación vs Comparación

Es importante no confundir el operador de asignación `:=` con el operador de comparación `=`

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }  
  
program asignacion_vs_comparacion;  
  
var  
    x : integer;  
  
begin  
    x := 0 ;  
    x := x + 1 ;    // Ahora x vale 1  
    writeln(x = x + 1); // Escribe FALSE  
end.
```



```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }
program asignacion_vs_comparacion_02;

var
    x : integer;
    a : boolean ;

begin
    x := 3 ;
    a := x = x + 1;
    a := (x = x + 1);    // Lo mismo, más claro
    writeln(a) ;        // Escribe FALSE

    a := x = 4 ;        // x vale 3
    writeln(a) ;        // Escribe FALSE

    x := x + 1 ;        // ahora x vale 4
    a := x = 4 ;
    writeln(a);         // Escribe TRUE
end.
```

# Ahormados

El ahormado (*casting*) de variables funciona igual que el ahormado de constantes

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }  
program casting_variables;  
  
var  
    a: integer;  
    x: real;  
    c: char;  
begin  
    a := 3;  
    x := a; // La conversión de integer en real es automática  
  
    c := 'Z';  
    x := integer(c);  
    writeln(x:0:0); // escribe 90  
end.
```

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }
program casting_variables_02;

var
  x: char;
  y: integer;
  z: real;
begin
  x := 'a';
  writeln( integer(x)); // Escribe 97

  y:= 98;
  writeln( char(y)); // Escribe 'b'
  writeln( real(y)); // Escribe 9.800000000000000E+001

  z:= 65.7;
  { writeln( integer(z)); // ¡Esto es ilegal! }

  writeln( trunc(z)); // Escribe 65

  writeln( char( trunc(z) ) ); // Escribe 'A'

  // Esto es redundante, trunc(z) ya devuelve un integer
  writeln( char( integer( trunc(z) ) ) ); // Escribe 'A'
end.
```

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }
program casting_variables_03

var
    c: char;
begin
    c := 'A';

    writeln ( ord(c) );           // Escribe 65

    c := chr( ord(c) + 1 );
    writeln(c);                  // Escribe 'B'

    writeln( succ(c) );          // Escribe 'C'
end.
```

# Procedimientos

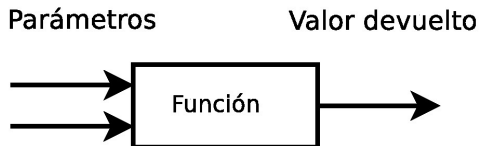
Recordemos que una función es un subprograma que

- No deberían tener efectos laterales
- Devuelve un valor
- No puede modificar sus argumentos

Un procedimiento

- Se espera que tengan efectos laterales. Por ejemplo, escribir en pantalla
- Puede modificar sus argumentos (aunque no es obligatorio)
- No devuelve ningún valor (aunque pueden modificar sus argumentos)

Un procedimiento es una acción con nombre

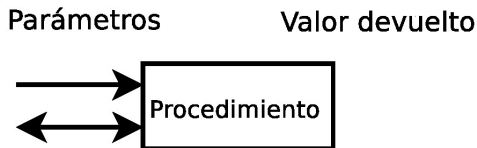


## Una función

- Recibe 1 o más parámetros de entrada<sup>1</sup>
- Devuelve exactamente 1 valor
- Si está bien escrita, no tiene efectos laterales

---

<sup>1</sup>En ocasiones excepcionales puede no tener ninguno



### Un procedimiento

- Puede no recibir ningún parámetro, puede recibir uno, puede recibir más de uno
- Alguno de los parámetros puede ser de salida (o no)
- No devuelve ningún valor

Supongamos que solo conocemos las funciones y tenemos este código

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }  
program sin_procedimiento;  
var  
    c: char;  
    i: integer;  
begin  
    c := 'a';  
    i := ord(c);  
    writeln('El código ASCII de ', c, ' es ', i);  
  
    c := 'b';  
    i := ord(c);  
    writeln('El código ASCII de ', c, ' es ', i);  
end.
```

El código ASCII de a es 97

El código ASCII de b es 98

Claramente es deseable factorizar esto (evitar el *copia y pega*), pero no es aceptable que una función tenga efectos laterales



```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }  
program procedimiento;  
  
procedure escribe_ascii(x: char);  
var  
    i: integer;  
  
begin  
    i := ord(x);  
    writeln('El código ASCII de ', x, ' es ', i);  
end;  
  
var  
    c: char;  
begin  
    c := 'a';  
    escribe_ascii(c);  
  
    c := 'b';  
    escribe_ascii(c);  
end.
```

El código ASCII de a es 97

El código ASCII de b es 98

Como las funciones, los procedimientos pueden no tener argumentos

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }  
program procedure_holamundo;  
  
procedure escribe_holamundo;  
begin  
    writeln('Hola, mundo');  
end;  
  
begin  
    escribe_holamundo;  
end.
```

- También podríamos declararlo y usarlo con paréntesis, pero sin argumentos

```
...  
procedure escribe_holamundo();  
...  
escribe_holamundo();
```

Un principiante puede pensar

- *Las funciones calculan expresiones, pueden seleccionar y hacer bucles. Pero no pueden escribir en pantalla*
- *Los procedimientos hacen lo mismo que las funciones, pero además sí pueden escribir en pantalla*

*... entonces los procedimientos son mejores. Usemos procedimientos para todo*

*¡No! Los funciones *calculan cosas* y los procedimientos *hacen cosas con las cosas calculadas**

- En este curso, prácticamente lo único que *hacemos* es escribir en pantalla
- En este curso puede parecer caprichosa esa limitación de las funciones. En nuestros ejemplos, las funciones podrían escribir en pantalla y aparentemente *no pasaría nada*

Pero el objetivo de este curso es adquirir hábitos de programación para entornos *reales*. En un entorno *real*, la necesidad de la separación función / procedimiento resulta muy evidente, porque podríamos tener

- Una función que *calcula algo*
- Un procedimiento que escribe ese algo en pantalla
- Otro procedimiento para insertarlo en una base de datos
- Otro procedimiento para escribir un informe para impresora
- Otro procedimiento para enviar el dato a Hacienda
- Otro procedimiento para preparar el dato para ver en un móvil
- Otro procedimiento para ver el dato en un tablet
- etc etc etc

En muchos lenguajes de programación no hay procedimientos, todo son funciones. Pero este principio es aplicable igualmente: si nuestro diseño es bueno tendremos

- *funciones función*, con transparencia referencial, que *calculan cosas*. Importante: devuelven valores.
- *funciones procedimiento*, con efectos laterales, que hacen cosas con las cosas calculadas por las funciones. Importante: no devuelven valores.

La idea fundamental, en cualquier lenguaje, es:

- 1 Un subprograma calcula las cosas
- 2 Otro subprograma distinto, hace cosas con lo calculado por el subprograma anterior.

En Pascal, el 1) suele ser una función, pero puede ser un procedimiento si necesitamos devolver más de una cosa (y no queremos usar registros)

En Pascal, el 2) es siempre un procedimiento.

# Readln

Con el procedimiento *readln* podemos leer un valor introducido por teclado y almacenarlo en una variable

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }  
program ejemplo_readln;  
  
var  
    c: char;  
begin  
    readln(c) ;  
    writeln(c);    // Escribe el caracter que hayamos introducido  
end.
```

Recuerda que un nombre de variable como *c* es típico en esta clase de ejemplos sencillos. En un programa real, tendríamos que elegir un nombre descriptivo

Observa que `readln` no devuelve nada. Ningún procedimiento devuelve nada. Si intentamos hacer esto

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }  
program ejemplo_readln_mal;  
  
var  
    c: char;  
begin  
    c:=readln(); //¡MAL! Los procedimientos no devuelven nada  
    writeln(c);  
end.
```

El compilador da un error

```
Compiling ejemplo_readln_mal.pas  
ejemplo_readln_mal.pas(8,17) Error: Incompatible types: got "untyped" expected "Char"  
ejemplo_readln_mal.pas(11) Fatal: There were 1 errors compiling module, stopping  
Fatal: Compilation aborted  
Error: /usr/bin/ppcx64 returned an error exitcode
```



```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }  
  
program ejemplo_readln_02;  
  
var  
    c: integer;  
begin  
    readln(c) ;  
    writeln(c);    // Escribe el entero que hayamos introducido  
                  // Da error de ejecución si no es un entero  
end.
```

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }  
program ejemplo_readln_03;  
  
var  
    i: integer;  
begin  
    writeln('Escribe un número entero: ');  
    readln(i) ;  
    write('La raíz cuadrada de ',i,' es ');  
    writeln( sqrt(i) );    // Escribe la raíz cuadrada del número  
                           // Da error de ejecución si no es un entero  
end.
```

# Halt

Con el procedimiento `halt` podemos concluir la ejecución del programa

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }  
program ejemplo_halt;  
  
var  
    i: string;  
begin  
    writeln('Escribe "q" si quieres concluir');  
    readln(i) ;  
    if i='q' then  
        halt           // Recuerda que aquí no se escribe ';' ;  
    else  
        writeln('El programa sigue...');  
    end.
```

# Paso de parámetro por referencia

- Paso por valor

La forma más habitual de pasar parámetros a funciones o procedimientos es el *paso por valor*<sup>2</sup>. Es la que hemos visto hasta ahora. El subprograma recibe una copia del parámetro, si el subprograma modifica el parámetro, los cambios se pierden al finalizar el subprograma

- Paso por referencia

Anteponiendo la palabra reservada `var` a un parámetro, pasa por referencia, no por valor. Esto significa que su valor puede modificarse dentro del procedimiento, y el cambio se verá después de la llamada al procedimiento.

- Observa que el procedimiento predefinido `readln()` está definido de forma que el parámetro se pasa por referencia

---

<sup>2</sup>En ocasiones denominado paso por copia

# Ejemplo de paso por valor

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }  
program paso_por_valor;  
  
procedure p(x:integer);  
begin  
    x := 0;  
    writeln(x);    // Escribe 0  
end;  
  
var  
    i: integer;  
begin  
    i := 3;  
    p(i);  
    writeln(i);    // Escribe 3, la variable no ha cambiado  
end.
```

# Ejemplo de paso por referencia

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }  
program paso_por_referencia;  
  
procedure p(var x:integer);  
begin  
    x := 0;  
    writeln(x);    // Escribe 0  
end;  
  
var  
    i: integer;  
begin  
    i := 3;  
    p(i);  
    writeln(i);    // Escribe 0, el procedimiento ha cambiado  
                  // la variable  
end.
```

La única diferencia en el fuente es añadir var, pero el comportamiento cambia por completo

Así, cuando pasemos parámetros a un procedimiento, normalmente

- Si queremos que cambien, pasamos por referencia, anteponiendo `var`
- Si no queremos que cambien, pasamos por valor, sin anteponer `var`

Con una excepción: objetos que ocupen mucha memoria

## Excepción: objetos grandes

Como hemos visto, al hacer un paso por valor (sin *val*), el procedimiento recibirá una copia del parámetro. Esta copia

- Duplica la memoria consumida
- Tarda cierto tiempo

El paso por referencia ahorra memoria y tiempo. Con tipos de datos que ocupan poco espacio, como los que hemos visto, esto no es importante. Pero sí lo será para objetos que ocupen mucha memoria

- Este ahorro también será necesario si una función intenta devolver un objeto muy grande: no podrá hacerlo. Será necesario escribir un procedimiento, donde el valor a devolver será un parámetro pasado por referencia

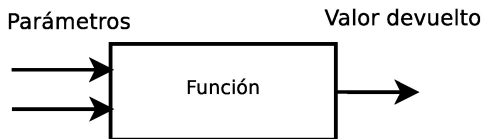


## En otras palabras

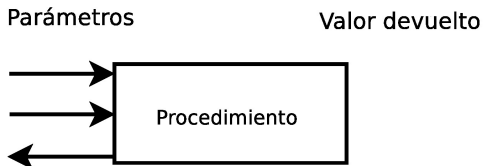
- A partir de una función, siempre es posible escribir un procedimiento equivalente. Basta eliminar el valor devuelto y añadir un parámetro de salida, con ese valor
- Siempre es *posible* pero ¿cuándo es *necesario*?
  - Cuando el valor a devolver es *grande*. En otros casos no tiene sentido

## Ejemplo 1 (gráfico)

Si tenemos una función con dos parámetros (de entrada)



Podemos convertirla en un procedimiento con dos parámetros de entrada y uno de salida



Ejemplo 2 (código) : Dado este programa que usa una función con un parámetro de entrada

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }  
  
program pts_funcion;  
  
const  
    EURPTS = 166.386; // Constante global, universal  
  
function pts_a_eur(pts : real) : real;  
begin  
    result := pts / EURPTS;  
end;  
  
procedure escribe_resultado(pts, eur: real);  
begin  
    write(pts:0:0 , ' PTS son ');  
    writeln(eur:0:2 , ' EUR');  
end;
```

```
var
    importe_pts, importe_eur : real;
begin // Cuerpo del programa principal
    importe_pts := 500;
    importe_eur := pts_a_eur(importe_pts);
    escribe_resultado(importe_pts, importe_eur);

    importe_pts := 1000000;
    importe_eur := pts_a_eur(importe_pts);
    escribe_resultado(importe_pts, importe_eur);
end.
```

Podemos escribir un programa equivalente, basado en un procedimiento equivalente con un parámetro de entrada y otro de salida

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }  
  
program pts_proc;  
  
const  
    EURPTS = 166.386; // Constante global, universal  
  
procedure pts_a_eur(pts : real; var eur: real);  
begin  
    eur := pts / EURPTS;  
end;  
  
procedure escribe_resultado(pts, eur: real);  
begin  
    write(pts:0:0 , ' PTS son ');  
    writeln(eur:0:2 , ' EUR');  
end;
```

```
var
    importe_pts, importe_eur : real;
begin
    importe_pts := 500;
    pts_a_eur(importe_pts, importe_eur);
    escribe_resultado(importe_pts, importe_eur);

    importe_pts := 1000000;
    pts_a_eur(importe_pts, importe_eur);
    escribe_resultado(importe_pts, importe_eur);
end.
```

Este ejemplo ha servido para ilustrar el concepto, no tiene ninguna otra utilidad ni tampoco sentido, porque *importe\_eur* es un único número, ocupa muy poca memoria

- Pero es necesario que sepamos usar el paso por valor. Si necesitamos devolver un objeto grande como por ejemplo un *array* con 1000 precios (lo veremos en el tema 8), entonces es imprescindible. También para devolver más de un objeto

# Paso por valor en funciones

También se pueden pasar parámetros por referencia a una función (no solo a un procedimiento)

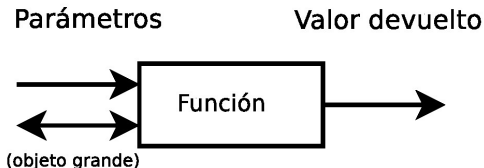
- Exactamente de la misma forma, anteponiendo `var`
- Normalmente no haremos esto. Pero como ahorra tiempo y memoria, para objetos grandes puede ser imprescindible. Un paso por valor de un objeto *grande* provocará un *desbordamiento de pila*
  - 1 Observa que no hablamos de *devolver* algo grande, como en el caso anterior, sino de *recibir* algo grande

- ¿Cuándo es *grande*? No es fácil saberlo a priori. Depende de la versión del compilador y del sistema operativo <sup>3</sup>
- Si es necesario pasar un objeto grande por referencia a una función, el programador será responsable de no modificarlo. El compilador lo permitirá, pero será un efecto lateral, que es un mal diseño (y en este curso, un ejercicio suspenso)

---

<sup>3</sup>En las versiones que manejamos ahora, el compilador de Windows acepta objetos mayores que el de Linux, pero esto podría cambiar en cualquier momento





Si alguno de los parámetros es un objeto *grande*, tendremos que pasarlo por referencia

- Desde el punto de vista del compilador, es un parámetro de entrada salida
- Aunque el compilador nos permita modificarlo, nunca debemos hacerlo. Nuestro único objetivo es ahorrar memoria
- En caso de que el objeto muy grande sea el valor a devolver, no podemos usar una función, sino un procedimiento (donde el valor a devolver será un parámetro de entrada/salida)

# Paso por referencia. Resumen

- Para pasar un parámetro por referencia, se antepone la palabra reservada *var* en la declaración
- Hace dos cosas
  - 1 Que el parámetro sea de entrada-salida, esto es, que la modificación del parámetro dentro del subprograma afecte también a la variable pasada como argumento. Lo usaremos en procedimientos
  - 2 Ahorra memoria. Lo usaremos cuando el objeto sea grande. Esto es imprescindible tanto en funciones como en procedimientos
- El compilador nos permitirá declarar un parámetro con paso por referencia en una función (aunque no sea un parámetro grande), por tanto nos permitirá modificarlo. Pero esto es un mal diseño
- En nuestro dialecto de Pascal (Object Pascal), las funciones no pueden devolver objetos grandes. Por tanto es necesario usar un procedimiento, con un parámetro de salida

## En otras palabras

- Cuando usamos paso por referencia (*anteponer var*) suceden dos cosas
  - 1 El compilador permite que modifiquemos el parámetro
  - 2 Se ahorra tiempo y memoria
- Pero cuando solo necesitemos una (ahorrar), no debemos usar la otra (modificar). Sería un mal diseño

# Otro ejemplo de paso por referencia

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

Si olvidamos añadir `var`, estamos pasando por valor y los cambios en el parámetro desaparecerán cuando la función termine su ejecución (En este caso, no era lo deseado)

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }  
  
program incrementa_mal;  
  
procedure incrementa(a:integer); // MAL: olvido var  
begin  
    a := a + 1;  
end;  
  
var  
    i: integer;  
begin  
    i := 3;  
    incrementa(i);  
    writeln(i); // Escribe 3  
end.
```

Este ejercicio tan sencillo (incrementar un número, usando un subprograma), también podemos hacerlo usando una función, no solo un procedimiento

Basta usar la variable tanto a la izquierda como a la derecha de la asignación

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }  
program incrementa_con_funcion;  
  
function incrementa(a: integer): integer;  
begin  
    result := a + 1;  
end;  
  
var  
    i: integer;  
begin  
    i := 3;  
    i := incrementa(i);  
    writeln(i);      // Escribe 4  
end.
```

Recuerda:

- Las funciones devuelven valores, las usamos para formar expresiones. Por ejemplo, si  $f$  y  $g$  son funciones, escribimos cosas como

```
writeln(f(x));  
y := f(x);  
z := f(g(x));  
y2 := f( alfa, beta, gamma);  
y3 := f( g(x1), g(x2), g(x3) );
```

$f$ ,  $g$ ,  $h$ , son nombres típicos de función en ejemplos como este, donde la función en sí misma no existe o es lo de menos. En un ejercicio completo los nombres serían descriptivos. P.e. *maximo*, *factorial*, *calcula\_rozamiento*, *tipo\_aplicable*, etc

- Los procedimientos no devuelven nada. Tal vez modifiquen un parámetro pero no podemos usarlos a la derecha del operador de asignación o como parámetro de una función o procedimiento

Por ejemplo, si  $p$  y  $q$  son procedimientos, escribiremos cosas como

```
p;  
p(x);  
q(x, y, f(z));
```

Pero jamás

```
writeln(p(x));    // MAL  
x := p(x);        // MAL  
p( q(x) );        // MAL
```

Si lo intentamos, el compilador dará un error



```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }
program incrementa_mal_02;

procedure incrementa(var a:integer);
begin
    a := a + 1;
end;

var
    i: integer;
begin
    i := 3;
    i := incrementa(i); // ¡¡MAL!! No es una función
    writeln( i );
end.
```

incrementa\_mal\_02.pas(13,7) Error: Invalid assignment, procedures return no value

Recuerda que una función devuelve un valor, y solo uno  
Si necesitamos un subprograma que devuelva no solo un valor, sino dos o más, hay dos formas de hacerlo

- Usar un procedimiento y pasar argumentos por referencia
- Usar registros (como veremos en el siguiente tema)

```
program devolver_pareja;

procedure division_entera(dividendo, divisor: integer;
    var cociente, resto: integer );
begin
    cociente := dividendo div divisor;
    resto := dividendo mod divisor;
end;

procedure escribir_valores(dividendo, divisor,
    cociente, resto:integer);
begin
    write('La división entera entre ', dividendo);
    write(' y ', divisor);
    write(' es ', cociente);
    writeln(' con un resto de ', resto);
end;
```

```
const
    Dividendo_ejemplo = 9;
    Divisor_ejemplo = 2;
var
    dividendo, divisor, cociente, resto : integer;
begin
    dividendo := Dividendo_ejemplo;
    divisor := Divisor_ejemplo;

    division_entera(dividendo, divisor, cociente, resto);
    escribir_valores(dividendo, divisor, cociente, resto);
end.
```

## Resultado:

La división entera entre 9 y 2 es 4 con un resto de 1

Observa que en el ejemplo anterior, en la declaración, hemos tenido que separar los argumentos pasados por valor de los argumentos pasados por referencia

Esto es, hemos escrito

```
procedure division_entera(dividendo, divisor: integer;  
    var cociente, resto: integer );
```

y no

```
procedure division_entera(dividendo, divisor,  
    var cociente, var resto : integer); // ¡¡MAL!!
```

# Procedimiento val

val es un procedimiento que hace uso de esta característica (modificar dos parámetros para devolver dos valores)

Sirve para convertir una cadena de texto en un número, si es posible

Ejemplos:

- La cadena '12' se puede convertir en el entero 12
- La cadena '4r2' no se puede convertir en un entero
- La cadena '4x2', tampoco (val convierte números, no expresiones)
- La cadena ' 8.2' se puede convertir en real (los espacios a la izquierda no importan)
- La cadena '24.8 ' no se puede convertir en real (los espacios a la derecha sí dan error)

Sus parámetros son:

- Una cadena
- Un parámetro principal de salida, que será un número del tipo necesario: entero o real <sup>4</sup>, según corresponda. Contendrá el valor numérico de la cadena, si tiene sentido
- Otro parámetro de salida, un código. Si vale 0 indica que la conversión ha sido posible. En otro caso, indica la posición de la cadena donde se produjo el error

---

<sup>4</sup>o algún tipo enumerado

end.



```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }  
program ejemplo_val_02;  
  
var  
    s: string;  
    i, codigo: integer;  
  
begin  
    write('Introduce un número entero ');  
    readln(s) ; // Lo que leemos es una cadena, 'por si acaso'  
    val(s, i, codigo);  
    if (codigo) = 0 then  
        writeln(i)  
    else  
        writeln('Error, ', s, ' no es un entero');  
end.
```

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }  
program ejemplo_val_03;  
  
var  
    s: string;  
    codigo: integer;  
    r: real;  
  
begin  
    write('Introduce un número real ');  
    readln(s) ;  
    val(s, r, codigo);  
    if (codigo) = 0 then  
        writeln(r)  
    else  
        writeln('Error, ',s,' no es un real');  
end.
```

# Concatenación de cadenas

Se pueden concatenar (unir) cadenas con el operador +

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }  
program concatenacion;  
  
var  
    s1, s2, s3: string;  
begin  
    s1 := 'hola, '  
    s2 := 'mundo';  
    s3 := s1 + s2;  
    writeln(s3);    // hola, mundo  
  
    s3 := 'concatenando ' + 'cadenas';  
    writeln(s3);    // concatenado cadenas  
end.
```

# Procedimiento str

`str` hace el paso inverso de `val`: toma un número o un tipo enumerado y lo convierte en cadena

Recibe dos parámetros

- El número o enumerado a convertir
- Una cadena, donde escribirá el resultado

```
program ejemplo_str;

var
  i: integer;
  r: real;

  s1, s2: string;
begin
  i := 4;
  r := 2.2311;

  str(i, s1); // Convierte i a cadena y lo guarda en s2
  str(r:0:2, s2); // Convierte r:0:2 a cadena y lo guarda en s3
  write(s1 );
  writeln(' ', ' + s2 ');
                // Escribe 4, 2.23
end.
```

# Ámbito de las variables

El ámbito de una variable es la zona del programa donde esa variable existe, donde se puede leer y escribir

Según su ámbito, las variables pueden ser

- Locales

- Locales del cuerpo principal, declaradas después de todas las funciones y procedimientos, justo antes del begin del cuerpo principal
- Locales de una función o procedimiento, declaradas justo antes del begin de esa función o procedimiento

- Globales

Son extremadamente peligrosas

- Algunas metodologías permiten que se usen en contadísimas excepciones y con mucho cuidado
- Para otras metodologías, no deben usarse nunca, en ningún caso. En este curso:  
Variable global  $\implies$  Suspense seguro

# Variable global

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }  
program variable_global;  
    // Péximo ejemplo. Suspenso seguro  
var  
    x: integer;  
  
    // Esta variable es global. Es visible en todo el código que viene  
    // a continuación, esto es, a todo el programa. Defecto muy severo.  
  
procedure incrementa();  
begin  
    x := x + 1;  
end;  
  
begin  
    x := 3;  
    incrementa();  
    writeln(x);      // Escribe 4  
end.
```

# Variable local del cuerpo principal

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

function incrementa(x:integer):integer;
begin
    x := x + 1;
    result := x;
end;

var      // Este es el lugar correcto para declarar las variables
         // que usa el cuerpo principal del programa. Así, serán
         // variables locales del cuerpo principal
    x: integer;
begin
    x := 3;
    x := incrementa(x);
    writeln(x);      // Escribe 4
end.
```



# Variable local de un subprograma

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }
program variable_local_02;

function incrementa(x:integer):integer;
var
    y:string;    // Este es un lugar correcto para declarar una
                 // variable. Es una variable local de la función

begin
    y := 'No uso esta variable pero es un ejemplo';
    x := x + 1;
    result := x;
end;

var
    x: integer;
begin
    x := 3;
    x := incrementa(x);
    writeln(x);    // Escribe 4
end.
```

# Problemas de las variables globales

Entre los problemas de las variables globales, destacamos dos

- Nunca podemos estar seguros de que una variable global sea verdaderamente global, porque una variable local puede taparla
- Supongamos que
  - 1 Un subprograma trabaja con una variable global y almacena cierto valor
  - 2 El programa sigue su curso, llamando a otros subprogramas
  - 3 El subprograma vuelve a trabajar con la variable global. Pero en el intervalo 2, cualquier otros subprograma puede haber hecho cualquier otra cosa con ese valor

El primer problema es serio, el segundo, mucho peor. Los veremos en los siguientes ejemplos

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }  
program ambitos_01;  
var  
    // Nunca se deben declarar variables aqui. Ahora lo hacemos para  
    // mostrar los problemas  
    nombre: string ;    // Variable global, suspenso seguro.  
  
procedure p( nombre: string);  
    // El argumento n oculta la variable global  
begin  
    writeln( nombre );    // Escribe María, el valor local, no el global  
end;  
  
begin  
    nombre := 'Juan' ;    // Definimos la variable global  
    p( 'María' );  
end.
```

```
{ $mode objfpc } { $H- } { $R+ } { $T+ } { $Q+ } { $V+ } { $D+ } { $X- } { $warnings on }
program ambitos_02;
var
    // Nunca se deben declarar variables aqui. Ahora lo hacemos para
    // ver los problemas

    nombre: string ;    // Variable global, suspenso seguro

function averigua_aeropuerto(codigo_iata: string):string;
begin
    if codigo_iata = 'MAD' then
        nombre := 'Madrid-Barajas Adolfo Suárez'
    else
        nombre := 'Aeropuerto Desconocido';
    result := nombre;
end;
// Esta función está machacando la variable global
```

```
var
  codigo_iata:string;           // Variables correctas, locales
  nombre_aeropuerto:string;

begin
  nombre := 'Juan García' ;    // Definimos la variable global
  codigo_iata := 'MAD' ;
  nombre_aeropuerto := averigua_aeropuerto(codigo_iata);
  writeln('Pasajero:', nombre);
  writeln('Aeropuerto destino:', nombre_aeropuerto);
end.
```

## Resultado:

Pasajero:Madrid-Barajas Adolfo Suárez

Aeropuerto destino:Madrid-Barajas Adolfo Suárez

# Resolución de ecuaciones de 2º grado

[https://gsyc.urjc.es/~mortuno/fpi/ecuacion\\_grado2.pas](https://gsyc.urjc.es/~mortuno/fpi/ecuacion_grado2.pas)