Programación en Pascal. Funciones

Miguel Ortuño
Escuela de Ingeniería de Fuenlabrada
Universidad Rey Juan Carlos

Marzo de 2024

Contenidos

Definición de problemas

La definición de un problema consiste en tres cosas:

- Saber qué resolvemos.
- Saber qué nos hace falta para resolverlo.
- Saber qué tendremos cuando lo hayamos resuelto.

Esto coincide con la declaración de una función.

- Cómo se llama la función.
- Qué necesita.
- Qué devuelve.

Funciones en pascal

Una función básica en Pascal tendrá la siguiente estructura:

```
function NOMBRE_FUNCION( LISTA_DE_PARAMETROS): TIPO_DEVUELTO;
begin
    result := EXPRESION_CON_LA_SOLUCION;
end;
```

- La lista de parámetros normalmente estará formado por 1 o más parámetros. Pero también puede ser una lista vacía, sin parámetros.
- Observa que para asignar valor a *result* empleamos el operador de asignación := [1], mientras que para dar valor a las constante usamos = [2].

[1] El mismo que usaremos para las variables, como veremos en el tema 5.

[2] Esta es una rareza de Pascal, en otros lenguajes no hay esta diferencia.

Ejemplo:

```
function long_circunf(r: real): real;
begin
    result := 2.0 * Pi * r;
end;
```

`result` es una palabra reservada que en nuestro dialecto de Pascal (Object Pascal) emplemos para darle valor a la función.

- En Pascal estándar usaríamos el mismo nombre de la función, pero esto tiene algunos inconvenientes.

```
function long_circunf(r: real): real;
begin
    long_circunf := 2.0 * Pi * r;
end;
```

Luego llamaremos a la función con un valor concreto p.e.

```
long_circunf(4.2);
```

En este caso

- 4.2 es el argumento, el valor concreto que suministramos a la función.
- r es el parámetro, el identificador que usamos para nombrar al argumento.

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

program longitud_circunferencia;

const
    Pi = 3.14159265;

function long_circunf(r: real): real;
begin
    result := 2.0 * Pi * r;
end;

begin
    writeln( long_circunf(4.2):0:3 );  // Escribe 26.389
end.
```

Observa que hemos declarado Pi al principio del programa, por ser una constante *universal*

Una forma alternativa de escribir el programa anterior es pasar la constante como un parámetro más de la función.

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

program longitud_circunferencia_v2;

const
    Pi = 3.14159265;

function long_circunf(r: real; pi: real): real;
begin
    result := 2.0 * pi * r;
end;

begin
    writeln( long_circunf(4.2, Pi):0:3 );   // Escribe 26.389
end.
```

- En este ejemplo, como Pi es una constante universalmente conocida, ambas soluciones serian aceptables. Un purista de la *programación funcional* posiblemente preferirá esta segunda solución, pero es discutible.
- Si la constante no es universalmente conocida y obvia como esta, es preferible este segundo enfoque: pasarla como parámetro.
- En caso de duda, también es preferible pasar la constante como parámetro de la función.

Importante: observa que la función *long_circunf* no utiliza la constante `Pi` (con mayúscula) sino el parámetro `pi` (con minúscula).

Una función en Pascal bien escrita es muy similar a una función matemática .

- Acepta uno o más parámetros.
- Devuelve un valor, calculado a partir de los parámetros (y de ninguna otra cosa).

Y nada más.

- NO lee ningún otro valor del exterior, solo sus parámetros.
- NO depende del estado del programa.
- NO depende del orden de evaluación de los parámetros.
- NO provoca ningún cambio en ninguna parte, solo devuelve su valor.
  - Esto incluye NO escribir nada en pantalla.

A esto se llama *transparencia referencial*.

Así, no hay ninguna diferencia entre

- Llamar a una función, con sus argumentos.
- Escribir el valor devuelto por la función.

Ejemplo. Tenemos un programa con la función

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

Luego llamamos a esta función

```
incrementa(3);
```

Si cumplimos la propiedad de transparencia referencial (que en este curso es obligatorio), nuestro programa se comportará exactamente igual en todo si, en vez de escribir esta llamada, escribimos

```
4;
```

Se denomina *efecto lateral* a cualquier acción producida por una función. Por tanto,
Estas dos cosas son equivalentes:

- Una función tiene transparencia referencial.
- Una función no tiene efectos laterales.

Podemos decir que una función con efectos laterales *hace lo que no debe*.

- El compilador no lo va a impedir, no se va a generar ningún error. Pero eso es código de muy mala calidad, propenso a errores (aunque lo podemos ver incluso en algunos libros de programación).
- En este curso no permitiremos funciones con efectos laterales.

Ejemplo de función con varios parámetros:

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

program media_v1;

function media(x: real; y:real): real;
begin
    result :=  (x + y) / 2;
end;

begin  // Aquí empieza el cuerpo principal
    writeln( media(4, 5) );
end.
```

Observa que writeln está en el cuerpo principal, nunca debe estar dentro de la función (porque eso sería un efecto lateral).

Otras consideraciones sobre las funciones

- En Pascal es necesario que la definición de una función aparezca en el código fuente antes que la llamada a esa función.
- Si una función tiene varios parámetros del mismo tipo, también podemos declararlos así:

```
function media(x, y: real): real;
begin
    result :=  (x + y) / 2;
end;
```

- En nuestro dialecto de Pascal, una función no puede devolver objetos *demasiado grandes* [3].

[3] La solución la veremos en el tema 5: será necesario usar un procedimiento y un parámetro de entrada-salida (parámetro por referencia).

Una función devuelve una expresión. En cualquier lugar donde se espera una expresión, se puede escribir una función.

```
        writeln( media(4, 5) + 1 );    // Escribe 5.5
```

En la llamada a una función, el argumento puede ser cualquier expresión: un literal, una constante, otra función...

```
        writeln( media(2 + 2, 3 + 2) + 1 );
```

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

program media_v3;

function media(x, y: real): real;
begin
    result :=  (x + y) / 2.0;
end;

begin
    writeln( media(4, 5)+1) ;       // Escribe  5.5000000000000000E+000

    writeln( (media(4, 5)+1):0:3 );   // Escribe  5.500

    writeln( media(2+2, 3+2) + 1); // Escribe  5.5000000000000000E+000
end.
```

De nuevo, todos los writeln están fuera de la función. Lo contrario sería un defecto muy severo.

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

program llamadas_funcion;

function suma(x, y: integer): integer;
begin
    result :=  x + y;
end;

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

const
    A = 4;    // Estas constantes no son universales,
    B = 1;    // son ejemplos particulares. Por eso
              // van aquí y no al principio.

begin
    writeln( suma(9, 2) );              // Escribe 11
    writeln( suma(A, B) );              // Escribe 5
    writeln( suma(A, incrementa(B))) ; // Escribe 6
end.
```

Independencia argumento - parámetro

El nombre de la constante que usamos como argumento no tiene relación que el nombre del parámetro. Pueden coincidir, pero en general no tienen por qué.

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

program nombres_parametros;

function media(x, y: real): real;
begin
    result := (x+y) / 2;
end;

const
    A = 2.0;
    B = 1.0;

begin
    writeln( media(A,B) );
end.
```

En el ejemplo anterior, *nombres_parametros*

- En el cuerpo del programa principal, para poder llamar a *writeln*, el compilador evalúa la expresión *media(A,B)*. Para ello, llama a la función *media* pasando como argumentos 2.0 y 1.0.
- La función *media* recibe los argumentos 2.0 y 1.0. Ignora por completo si estos valores eran constantes numéricas literales (números *tal cual*), o las constantes A y B, o son valores que se han calculado desde otra expresión, o son el resultado de una llamada a una función...
  La función solo *sabe* que el parámetro *x* vale 2.0 y que el parámetro *y* vale 1.0. Cualquier otra información no es relevante.

El siguiente ejemplo provocará un error de compilación: intenta usar las constantes A y B en la función *media*. Pero estas constantes solo se pueden usar en el cuerpo del programa principal.

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

function media(x, y: real): real;
begin
        result := (A+B) / 2 ;  // ¡MAL!
end;

const
        A = 2.0;
        B = 1.0;

begin
        writeln( media(A,B) );
end.
```

```
Free Pascal Compiler version 3.2.2+dfsg-9ubuntu1 [2022/04/11] for x86_64
Copyright (c) 1993-2021 by Florian Klaempfl and others
Target OS: Linux for x86-64
Compiling nombres_parametros_mal.pas
nombres_parametros_mal.pas(7,13) Error: Identifier not found "A"
nombres_parametros_mal.pas(7,15) Error: Identifier not found "B"
nombres_parametros_mal.pas(17) Fatal: There were 2 errors compiling module,
   stopping
Fatal: Compilation aborted
Error: /usr/bin/ppcx64 returned an error exitcode
```

Reutilización de nombres de parámetro

A continuación veremos un programa donde cierta función utiliza los nombres de parámetro *x,y* y otra función vuelve a emplear los mismos nombres.

- Esto es perfectamente correcto.
- Los parámetros solo se consideran dentro de la función donde se declaran. En programación decimos que *el ámbito de los parámetros es local*.
- Si otra función usa los mismos nombres, o no, es irrelevante.

Metáfora: una casa donde se usa la palabra *mamá* y esta identifica a una y solo una persona. En otro ámbito diferente, p.e. la casa de los vecinos, *mamá* será una persona distinta, pero esto no genera ningún problema.
El identificador debe ser único en su ámbito, pero en un programa normalmente habrá muchos ámbitos diferentes.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program reutilizacion_parametros;
function media(x, y: real): real;
begin
        result := (x+y) / 2;
end;

function suma(x, y: real): real;
begin
        result := x + y;
end;

const
        A = 2.0;
        B = 1.0;
        C = 4.0;
        D = 3.0;

begin
    writeln( media(A,B):0:1 );
    writeln( suma(C,D):0:1 );
end.
```

Importante:

- En ejemplos de funciones sencillas como estas, es tradicional usar nombres de una sola letra como *x,y* o tal vez *a,b* porque reslta obvio que se refieren al primer sumando y al segundo sumando. O al primer y segundo número de una media aritmética de dos números.
- Pero cuando las funciones no sean triviales, es necesario usar nombres de parámetro más descriptivos, como p.e *dni_alumno*, *masa_vehiculo*, *nota_corte*, etc etc

Diseño de programas

Los problemas ser resuelven dividiéndolos en subproblemas, que serán subprogramas.
Lo habitual es:

1. Primero, diseñar la solución *top-down* (arriba-abajo). Empezamos por el problema global, suponiendo que tenemos resueltos los subproblemas. Aplicamos esto las veces necesarias, hasta que tengamos un problema con solución directa.
2. Después, programar esa solución, *bottom-up* (abajo-arriba) . Programamos los subproblemas, empezando por los más sencillos. Probamos cada subprograma, con valores de ejemplo. Cuando estemos razonablemente seguros de que funciona, programamos un subprograma más complejo.

En Pascal, los subproblemas serán las funciones y los procedimientos (que veremos más adelante).

- Cualquier concepto de cierta entidad de nuestro programa deberá ser una función o procedimiento, que tendrá sentido por si mismo y que deberá ser verificado.
- El cuerpo del programa principal no debería calcular ni procesar nada, solo llamar a los subprogramas de nivel más alto.
- Los subprogramas deben ser *pequeños* ¿cuánto es pequeño? Depende.
  - Una función o procedimiento de 1 línea es perfectamente normal (si tiene entidad y sentido).
  - 5, 10, 15 líneas son valores habituales.
  - Una función o procedimiento que no cabe en una pantalla (de resolución media) debe levantarnos sospechas.

  Por supuesto, no basta con que el subprograma sea corto para poder decir que está bien escrito.

Problema: código repetido

Es fundamental que el código necesario para resolver un subproblema se escriba 1 vez y solo 1.

- Al mal programador *le da pereza* escribir la función adecuada y *copia y pega* las lineas que necesita. Este es uno de los defectos principales en los malos programas.
- Modificar unas líneas repetidas n veces obliga a hacer n modificaciones. Una función bien escrita basta con modificarla 1 vez.
  - Cualquier programa real tendrá muchas modificaciones.
- Una función bien escrita se prueba n veces. Si el código está repetido m veces, habría que probarlo $n \times m$ veces.
- En un programa nunca debe haber código igual o similar repetido: se debe escribir un único subprograma, particularizado en cada caso con los parámetros necesarios.

Pruebas

Es prácticamente imprescindible probar cada uno de nuestros subprogramas por separado.

- Esta es una de las principales diferencias entre un buen programador y uno malo.
- Los principiantes suelen escribir funciones grandes y complejas, un amasijo de líneas que:
  - El autor, cuando las escribe, cree entender. Más o menos.
  - Una persona distinta, incluyendo el autor cierto tiempo después, no entenderá.

  Un código que no se entiende es imposible de probar ni corregir.

Probar un programa es una disciplina en sí misma. En entornos rigurosos:

- Se usan herramientas automáticas que comprueban cada subprograma de forma sistemática, con grandes conjuntos de parámetros de entrada.
- Las pruebas las hacen personas distintas, incluso en lenguajes distintos. Esto no exime al programador de hacer sus propias pruebas, son pruebas adicionales.
- Cada vez que se añade o modifica algo, se repiten todas las pruebas, completas, desde cero.

En entornos menos exigentes deberemos, al menos:

- Hacer varias llamadas a cada una de nuestras funciones de forma independendiente, con parámetros controlado y *escritos a mano*, observando la salida.
- Probar algunos parámetros con valores *bajos*, algunos *medios* y algunos *altos*, prestando atención a los extremos que suelen ser los conflictivos.

Volumen de un cilindro hueco

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program cilindro_hueco;
const       // Constantes globales, universales
    Pi = 3.1415926;  // Observa que usamos =, no :=

function area_circulo(r: real): real;
begin
    result := Pi*sqr(r);
end;

function area_corona(r_interior: real; r_exterior: real): real;
begin
    result := area_circulo(r_exterior) - area_circulo(r_interior);
end;

function vol_cil_hueco(r_interior, r_exterior, alt: real): real;
begin
    result := area_corona(r_interior, r_exterior) * alt;
end;
```

```
const    // Constantes del cuerpo del programa
    Rmin = 3.2;
    Rmax = 4.3;
    Alt = 2.8;

begin
    //writeln( area_circulo(1));
    //writeln( area_circulo(0.1));
    //writeln( area_circulo(0));
    //writeln( area_circulo(100000));

    //writeln( area_corona( 10, 10));
    //writeln( area_corona( 0, 0));
    //writeln( area_corona( 0.1, 0.1));

    writeln('v = ', vol_cil_hueco(Rmin, Rmax, Alt) :0:3);
end.
```

Letra o número ASCII

Escribamos un programa que :

- Reciba un código ASCII en binario, como 7 números enteros con valor 0 o 1.
- Devuelva TRUE si se trata de un número o una letra, devuelva FALSE en otro caso.

Subproblemas que tendremos que resolver:

- Convertir un número binario en decimal.
- Saber si un código ascii se corresponde con un dígito.
- Saber si un código ascii se corresponde con una mayúscula.
- Saber si un código ascii se corresponde con una minúscula.

ASCII significa *American Standard Code for Information Interchange*, por tanto se trata siempre de letras inglesas.

¿Cómo convertir un número binario en decimal?
Supongamos 4 dígitos binarios: d1, d2, d3, d4.

- valor_decimal $= d1 * 2^3 + d2 * 2^2 + d3 * 2^1 + d4 * 2^0$
  Ejemplo: 1100 = 1*8 + 1*4 + 0*2 + 0*1 = 12

Observa que el valor de un número decimal se calcula de forma análoga:

- $6320 = 6 * 1000 + 3 * 100 + 2 * 10 + 0 * 1 =$
  $d1 * 10^3 + d2 * 10^2 + d3 * 10^1 + d4 * 10^0$

```
function binario_a_decimal(d1,d2,d3,d4,d5,d6,d7 : integer): integer;
// Recibe 7 dígitos correspondientes a un número binario, lo convierte
// en decimal
begin
    result := d1 * 2 ** 6 +
              d2 * 2 ** 5 +
              d3 * 2 ** 4 +
              d4 * 2 ** 3 +
              d5 * 2 ** 2 +
              d6 * 2 ** 1 +
              d7 * 2 ** 0 ;
end;
```

Para saber si es dígito, mayúscula o minúscula:

```
function es_digito_ascii( x: integer): boolean;
// Recibe un código ascii como número entero, indica si se corresponde
// con un  dígito
begin
    result :=  ( x >= ord('0'))   and   (x <= ord('9'));
end;

function es_minuscula( x: integer): boolean;
// Recibe un código ascii como número entero, indica si se corresponde
// con letra minúscula (inglesa)
begin
    result :=  ( x >= ord('a'))   and   (x <= ord('z'));
end;

function es_mayuscula( x: integer): boolean;
// Recibe un código ascii como número entero, indica si se corresponde
// con letra mayúscula (inglesa)
begin
    result :=  ( x >= ord('A'))   and   (x <= ord('Z'));
end;
```

Para saber si es número o letra:

```
function es_numero_o_letra( x: integer): boolean;
// Recibe un código ascii como número entero, indica si se corresponde
// con un digito o con una letra (inglesa)
begin
    result :=
        es_digito_ascii(x) or es_minuscula(x) or es_mayuscula(x);
end;
```

Ejemplo completo:
https://gsyc.urjc.es/~mortuno/fpi/letra_o_numero.pas