7. PROCEDIMIENTOS Y FUNCIONES

Conceptos: *Rutina, Subrutina, Subprograma, Procedimiento, Función, Parámetro, Parámetros reales y formales, Parámetros por valor y por variable, Variables globales y locales, Niveles, Ámbito, Recursividad.*

Resumen: En este tema se estudian las rutinas como elementos fundamentales para el desarrollo de algoritmos y programas. Una estrategia para la resolución de problemas complejos es su descomposición en otros más simples. Cada pequeño problema se resuelve mediante subprogramas o rutinas de forma que los programas pueden componerse de un programa principal y un conjunto de rutinas. En TurboPascal, existen dos tipos de rutinas: los procedimientos y las funciones. Se busca la familiarización del alumno con el diseño de rutinas y sus aspectos técnicos de implementación (tipo de rutina, número de parámetros o argumentos y su tipo, tipo de valor devuelto en el caso de las funciones). Se presta especial atención a los parámetros por valor y por variable y a los conceptos de identificadores globales y locales y su ámbito de utilización. A continuación se estudia el tratamiento de la recursividad y se finaliza repasando las rutinas predefinidas más importantes en TurboPascal.

Objetivos específicos. Al finalizar el tema, el alumno deberá ser capaz de:

a) Describir el mecanismo de funcionamiento de una función y de un procedimiento y sus diferencias, así como su utilidad (*Conocimiento*)
b) Definir los conceptos: cabecera de procedimiento o función, parámetros formales, variables locales, resultado de la función, llamada al procedimiento o función, parámetros reales y paso de parámetros (*Conocimiento*)
c) Describir la diferencia entre paso de parámetros por valor o por variable (*Conocimiento*)
d) Definir el concepto de recursión, recurrencia o recursividad (*Conocimiento*)
e) Interpretar el resultado de la declaración y llamada de una rutina dentro del código fuente de un programa (*Comprensión*)
f) Codificar una tarea sencilla convenientemente especificada utilizando una rutina (*Aplicación*)

7.1. INTRODUCCIÓN

En programación a los procedimientos y funciones también se les conoce por el nombre de *rutinas*, *subrutinas* o *subprogramas*. Son bloques de instrucciones que realizan tareas específicas. Las rutinas se declaran una sóla vez pero pueden ser utilizadas, mediante *llamadas*, todas las veces que se quiera en un programa. Una rutina es independiente del resto del programa por lo que, en principio, facilita el diseño, el seguimiento y la corrección de un programa. Pueden además almacenarse independientemente en colecciones llamadas *librerías* o *unidades*, lo cual permite que sean utilizadas en cualquier programa. De hecho, existen funciones y procedimientos que vienen ya construidos para el lenguaje de programación TurboPascal (`Cos`, `Sin`, `Exp`, `ReadLn`, `WriteLn`, `ClrScr`...), que están almacenados en distintas unidades (`System`, `Crt`, `Graph`...) y que el programador puede emplear en sus programas. Además, el programador puede construir sus propias unidades con las constantes, tipos de datos, variables, funciones y procedimientos,... que desee incluir.

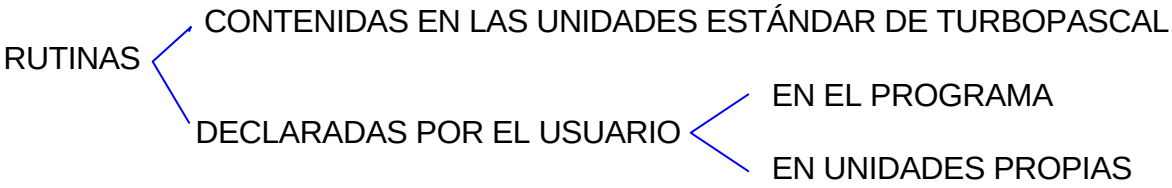

Figura 22. Clasificación esquemática de las subrutinas que se pueden emplear en un programa

7.2. CONSTRUCCIÓN Y USO DE PROCEDIMIENTOS

Un procedimiento es una parte del programa que realiza una acción específica basada a menudo en una serie de *parámetros* o *argumentos*. El procedimiento se ejecuta simplemente con una sentencia que es igual a su identificador seguido de los parámetros correspondientes, si los tiene, entre paréntesis. A diferencia de las funciones, no tiene sentido situarlos en una expresión porque no devuelven un valor una vez terminada su ejecución.

7.2.1. Declaración de un procedimiento

La forma general de declaración de un procedimiento es similar a la de un programa. La cabecera es, en este caso, obligatoria y comienza por la palabra reservada `PROCEDURE` y no `PROGRAM`. A continuación, viene la zona de declaraciones **locales** (en las subrutinas no pueden declararse unidades) y el cuerpo de sentencias del procedimiento.

El esquema de declaración de un procedimiento es el siguiente:

```
PROCEDURE Ident(parametros);   Cabecera del procedimiento
      CONST  {. . .}           Declaración de constantes locales
      TYPE {. . .}       Declaración de tipos locales
      VAR {. . .}              Declaración de variables locales
      FUNCTION  {. . .} Declaración de funciones locales
      PROCEDURE  {. . .}Declaración de funciones locales
      BEGIN
{. . .}                        Cuerpo del procedimiento
      END;
```

Como ya se explicará detenidamente más adelante, en general, lo que se declare localmente en (dentro de) un procedimiento no se podrá utilizar en otra parte del programa fuera del procedimiento, mientras que lo que se declare previamente a nivel global en el programa sí se podrá utilizar en el procedimiento.

7.2.2. Parámetros o argumentos

Los *parámetros* o *argumentos* son un mecanismo para pasar datos del programa principal a un procedimiento y viceversa. Los parámetros de la llamada para ejecutar la rutina se llaman parámetros **reales** mientras que los parámetros en la declaración de la rutina se llaman parámetros **formales**. A continuación se muestra un ejemplo de programa con un procedimiento que tiene un parámetro:

```
      program ejemplo1;
      var a : integer;
      procedure asteriscos(n:word);  { n, parametro formal }
  var i:integer;                      { i, variable local }
        begin
        for i:=1 to n do write('*');
        writeln
        end;
      begin
      a:=7;
      asteriscos(a);                     { a es el parametro real }
      asteriscos(4);                      { 4 es el parametro real }
      asteriscos(9-a)                   { 9-a es el parametro real }
      end.
```

La salida por pantalla tras la ejecución del programa anterior es la siguiente:

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

Si hay varios parámetros formales del mismo tipo, los identificadores se separarán por comas y si son de distintos tipos, con los correspondientes caracteres de punto y coma.

Ej. 1: `Procedure Ejemplo1(a,b,c:Integer);`
Ej. 2: `Procedure Ejemplo2(d,e:Integer;f,g:Char;t:real);`
Ej. 3: `Procedure Ejemplo3;  { procedimiento sin parametros }`

Los parámetros pueden ser de cualquier tipo predefinido o ya declarado en la definición de tipos, incluso pueden ser funciones o procedimientos. Asimismo pueden existir parámetros formales de distintas clases.

7.2.3. Parámetros formales POR VALOR (o de entrada)

Los parámetros formales por valor permiten la introducción de información en el procedimiento.

Sintaxis: *`identificador : tipo_dato;`*

o si hay **n** parámetros por valor del mismo tipo:

`identificador_1,...,identificador_n : tipo_dato;`

Los valores son introducidos en la llamada a la subrutina mediante el parámetro real que puede ser una constante, una variable o una expresión con un valor determinado. Al empezar a ejecutarse la subrutina se reserva espacio en memoria para el parámetro formal al que, en un principio, se le asigna el mismo valor que el parámetro real. Posteriormente los valores que vayan tomando durante la ejecución de la subrutina los parámetros real y formal serán independientes. Una vez terminada de ejecutar la subrutina se deja libre de uso el espacio de memoria correspondiente al parámetro formal. Ejemplo de un programa que incluye un procedimiento con un parámetro formal por valor:

```
program ejemplo2;
var r:integer;
procedure modi(f:integer); {f parametro formal por valor}
  begin
  f:=8
```

```
  end;
begin
r:=5;     { Antes de la llamada se le asigna el valor 5 }
modi(r);          { llamada a modi, r es el parámetro real }
writeln(r)           { tras la ejecucion sigue r vale 5 }
end.
```

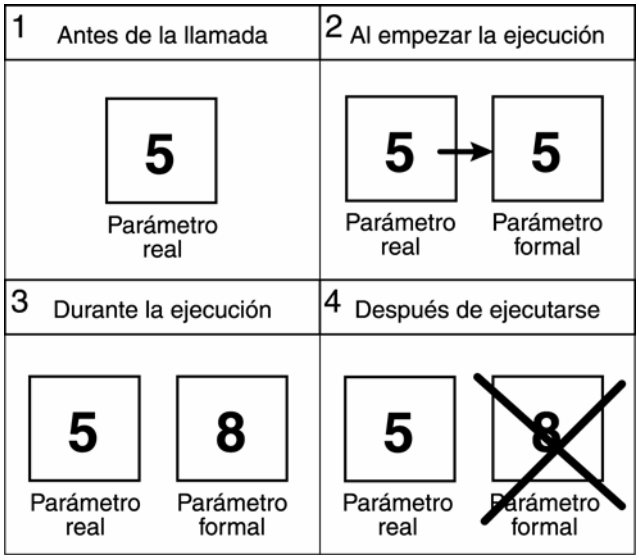

Figura 23. Utilización de la memoria al ejecutar una rutina con un parámetro formal por valor

7.2.4. Parámetros formales VARIABLES (o de entrada/salida)

Los parámetros formales variables permiten la introducción y obtención o salida de información en el procedimiento.

Sintaxis: **var** *identificador_1,...,identificador_n* : *tipo*;

En este caso, la información que pasamos no es un valor del parámetro real sino la dirección de memoria en la que se almacena el valor del parámetro real. Este espacio de la memoria es compartido con el parámetro formal durante la ejecución del procedimiento. Es decir, cualquier cambio en el parámetro formal afectará al real y viceversa, de forma que el valor final del parámetro formal en la subrutina lo conserva el parámetro real al terminar de ejecutarse la subrutina.

Los parámetros **reales** que corresponden a parámetros variables deben ser **variables** (¡nunca constantes, ni expresiones!) que pueden tener o no un valor al hacer la llamada a la subrutina. Por ejemplo, los parámetros variables se pueden utilizar para asignar un valor por primera vez a variables globales del programa (*inicialización* de variables). Ejemplo de un programa que incluye un procedimiento con un parámetro formal por valor:

```
program ejemplo3;
      var r:integer;
      procedure modi(var f:integer); {f param. formal variable}
        begin
        f:=8
        end;
      begin
      r:=5;    { Antes de la llamada se le asigna el valor 5 }
      modi(r);                          { r es el parámetro real }
      writeln(r)       { Se visualiza el nuevo valor de r: 8 }
      end.
```

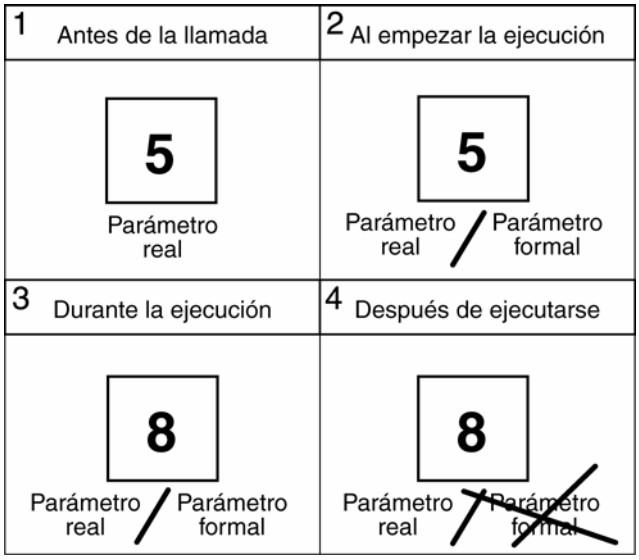

Figura 24. Utilización de la memoria al ejecutar una rutina con un parámetro formal variable

7.2.5. **Parámetros formales** VARIABLES SIN TIPO

Los parámetros formales variables sin tipo también permiten la introducción y obtención o salida de información en el procedimiento.

Sintaxis: VAR *identificador_1,...,identificador_n*;

Los procedimientos que declaran este tipo de parámetros formales se llaman de forma similar a los anteriores, pero pueden emplear parámetros reales de cualquier tipo, de forma que los parámetros formales asumen los tipos correspondientes de aquellos. Un ejemplo de procedimiento con parámetros formales de diversos tipos es el siguiente:

```
procedure ec(a,b:integer;var c,d:integer;var e:char;var f,g);
```

donde a y b son parámetros por valor; c, d y e son parámetros variables y f y g son parámetros variables sin tipo.

7.2.6. Normas generales de uso de parámetros

Tanto los parámetros de un procedimiento como lo declarado localmente sólo pueden utilizarse mientras se ejecuta dicha subrutina. Así, por ejemplo, una variable local de una subrutina sólo tiene reservada espacio de almacenamiento en memoria mientras ésta se ejecuta. Por su parte, los parámetros formales sólo están definidos durante la ejecución de las subrutinas: no tiene sentido hacer referencia a ellos fuera de éstas.

Es importante tener en cuenta siempre que el número de parámetros reales tiene que ser el mismo que el de parámetros formales.

Finalmente, los identificadores de los parámetros reales no tiene por qué coincidir con los de los parámetros formales, pero los tipos correspondientes han de ser idénticos (parámetros formales variables) o compatibles (parámetros formales por valor).

7.2.7. Ejemplos de declaración y llamada de procedimientos

A continuación se muestran varios ejemplos de declaraciones de procedimientos en un programa:

```
procedure cuadro;       { Procedimiento sin parametros }
```

```
  begin
  writeln('*****************************');
  writeln('*                           *');
  writeln('*****************************')
  end;
procedure linea(c:char; n:integer);
  var i :integer;     { variable local del procedimiento }
  begin
  for i:=1 to n do write(c);
  writeln
  end;
procedure inicializa(var i:integer);
  begin
  writeln('introduce un valor :');
  readln(i)
  end;
procedure esfera(radio:real; var superficie, volumen:real);
  begin
  superficie:=4*pi*radio*radio;
  volumen:=(4/3)*pi*radio*radio*radio
  end;
```

Dada la siguiente declaración de variables en un programa, se van a mostrar ejemplos de llamadas **incorrectas** al procedimiento `esfera`:

```
var  a,b,c : real;
     d     : integer;
     e     : char;
begin
a:=12; e:='6';
```

`esfera(a,b);` el número de parámetros reales y formales es distinto.

`esfera(`**`e`**`,b,c);` primer parámetro real no compatible con su correspondiente formal.

`esfera(3,b,`**`12`**`);` tercer parámetro real constante corresponde a un p. formal variable.

`esfera(4,b,`**`d`**`);` tercer parámetro real no compatible con su correspondiente formal.

`a:=``esfera(a,b,c);` procedimiento dentro de una expresión.

...

En el siguiente programa se incluye un ejemplo de llamada correcta al procedimiento esfera:

```
program calculos;
var a,b,c : real;
procedure esfera(radio:real; var superficie, volumen:real);
  begin
  superficie:=4*pi*radio*radio;
  volumen:=(4/3)*pi*radio*radio*radio
  end;
begin
writeln('Introduce el radio de la esfera : ');
readln(a);
esfera(a,b,c);
writeln('La superficie es: ',b,' metros cuadrados');
writeln('El volumen es: ',c,' metros cubicos')
end.
```

7.2.8. Procedimientos estándar de entrada y salida de datos

Para la entrada o salida de datos en los programas de TurboPascal se utilizan varios procedimientos estándar incluidos en la unidad `System`.

7.2.8.1 Procedimientos estándar de salida de datos

Los procedimientos estándar de salida o escritura de datos durante la ejecución de un programa son **`Write`** y **`WriteLn`**. La salida se realiza hacia la pantalla. La sintaxis de la llamada a cualquiera de estos procedimientos es:

`Write`(**`Ln`**)(*`expresion_1, expres_2, expres_n`*);

expresion_i son constantes, variables o expresiones para evaluar. expresion_*i* puede ser cualquiera de los elementos: *`entero`*, *`entero:n`*, *`real`*, *`real:n`*, *`real:n:m`*, *`caracter`*, *`caracter:n`*, *`cadena`*, *`cadena:n`*, *`logico`* o *`logico:n`*. Donde *entero*, *real*, *caracter*, *cadena* y *logico* indican datos (constantes, variables o expresiones) de dicho tipo; **`n`** es un número entero que indica la longitud del campo de salida (ocupará n espacios incluyendo un espacio que se reserva para el signo con los números reales) y **`m`** es un número entero que indica el número de dígitos que siguen al punto decimal. Cualquier otra expresión producirá un error de compilación.

La única diferencia entre los procedimientos `WriteLn` y `Write` se explica a continuación. Tras ejecutarse `WriteLn` el cursor salta al principio (primera columna) de la siguiente línea. Esta características hace que `WriteLn` sólo puede emplearse para la salida de datos por pantalla o con ficheros estructurados por líneas, es decir de tipo `TEXT`, tal y como se explica en el capítulo de *Archivos*. Tras ejecutarse `Write` el cursor no salta a la siguiente línea, quedándose en la columna a continuación del dato de salida.

Con estos procedimientos pueden escribirse valores de constantes ya sean literales o con nombre. En el primer caso si son caracteres o cadenas de caracteres el valor debe ir entre comillas simples. A continuación se muestran algunos ejemplos de utilización de estos procedimientos en un programa[12] teniendo la pantalla como salida de datos:

```
var i,j :integer; r : real;
begin
i:=12; j:=345;
write(i);
```

```
1 2 _
```

```
write(j);
```

```
1 2 3 4 5 _
```

```
writeln(i);
```

```
1 2 3 4 5 1 2
_
```

```
writeln(i,j);
write(i,j)
```

```
1 2 3 4 5
1 2 3 4 5 _
```

```
i:=12345;
r:=125.1435;
writeln(i);
writeln(i:8);
writeln(r)
writeln(r:10);
writeln(r:10:2);
writeln(r:10:5);
write('hola':5);
```

```
1 2 3 4 5
        1 2 3 4 5
  1 . 2 5 1 4 3 5 0 0 0 0 E + 0 2
  1 . 2 5 1 E + 0 2
          1 2 5 . 1 4
  1 2 5 . 1 4 3 5 0
  h o l a _
```

[12] El carácter _ indica dónde se sitúa el cursor tras la ejecución de la sentencia

Como puede observarse en los ejemplos anteriores, al tratar de visualizar datos de un tipo numérico, estos procedimientos los transforman en cadenas de caracteres antes de mostrarlos en pantalla. Ocurre lo mismo al manipular datos de tipo lógico. Para la escritura de valores lógicos puede utilizarse la siguiente sentencia:

```
WriteLn (Respuesta);
```

Dependiendo del valor almacenado en la variable Respuesta se visualizaría por pantalla las cadenas False o True. También puede emplearse una opción del siguiente tipo:

```
If Respuesta Then WriteLn('Cierto') Else WriteLn('Falso');
```

Con estos procedimientos, también pueden sacarse datos por impresora. Para esto se emplea una variable archivo declarada en una unidad estándar de TurboPascal.

Ej.:
```
Uses Printer; {Printer es una unidad de TurboPascal}
Begin
WriteLn(Lst,'Esto es una prueba')
End.
```

En resumen, con estos procedimientos pueden escribirse datos de tipos entero, real, `Char`, `String` o `Boolean` durante la ejecución de un programa.

7.2.8.2 Procedimientos estándar de entrada de datos

Los procedimientos estándar de entrada o lectura de datos durante la ejecución de un programa son **`Read`** / **`ReadLn`**. La entrada se realiza vía teclado. La sintaxis de la llamada es:

`Read`(**`Ln`**)(*`var_1, var_2,.., var_n`*);

Los datos que se van a leer se pueden ir visualizando a la vez por la pantalla. Los demás elementos *`var_i`* son los identificadores de las variables (no valen constantes, ni expresiones) en las que se quieren almacenar los datos.

El procedimiento `ReadLn` sólo puede utilizarse con el teclado o con ficheros de tipo `Text` ya que después de leer el/los dato/s salta al principio de la siguiente línea. Esto se verá más adelante en el capítulo de *Archivos*.

Pueden leerse varios datos con una sola sentencia: los identificadores de las variables irán separados por comas. En este caso, para que los valores sean leídos correctamente y almacenados en las variables respectivas, deben ir separados por un carácter que puede ser un espacio en blanco o un retorno de carro.

Para evitar problemas durante la ejecución, en cualquier caso, no es aconsejable utilizar más de un dato por instrucción. La entrada de datos y la asignación en las variables correspondientes sólo es válida después de pulsar la tecla de INTRO.

Ejemplo de utilización de un procedimiento de entrada de datos vía teclado:

```
var a,b,c  : integer;
 begin
 read(a,b,c);
```

```
1  2     8     4  1  7   intro
```

La introducción vía teclado de la secuencia `12` (*espacio*) `8` (*espacio*) `417` (*intro*) tiene como resultado la asignación de los valores `12`, `8` y `417` para las variables enteras `a`, `b` y `c`, respectivamente.

Como resumen, pueden leerse datos de tipos entero, real, `Char` y `String` durante la ejecución de un programa. Para otros tipos de dato, por ejemplo, para la lectura de valores lógicos no pueden emplearse estos procedimientos pero puede utilizarse el siguiente método o alguno similar:

```
var caracter : char; respuesta : boolean;
begin
writeln('teclea s o S para true');
```

```
readln(caracter);
respuesta:=(caracter='s') or (caracter='S');
```

7.3. CONSTRUCCIÓN Y USO DE FUNCIONES

Una función es una parte del programa que puede manipular datos y devolver un valor de un cierto tipo. Se ejecuta al hacer una llamada a dicha función dentro de una expresión. La llamada a una función puede incluirse en cualquier expresión en la que un dato del tipo que devuelve la función tenga sentido.

7.3.1. Declaración de una función

La forma de declaración de las funciones es análoga a la de los procedimientos. Sólo se diferencian en dos detalles.

En primer lugar, la cabecera, en la que hay que indicar el tipo del valor que devuelve la función. Por este motivo la llamada a una función debe ir incluida adecuadamente en una expresión. El tipo de dato que devuelve una función puede ser ordinal, real, cadena o puntero (no valen los demás tipos de dato estructurados: `Array`, `Record`, `Set` o `File`).

En segundo lugar, en el cuerpo de la función hay que indicar mediante una sentencia de asignación el valor que debe devolver la función. El primer término de esta sentencia de asignación es el identificador de la función y el segundo término, una constante, variable o expresión compatible con el tipo de dato que devuelva la función. Aparte de esto, todo lo dicho anteriormente para los procedimientos sirve para las funciones.

El esquema de declaración de una función es el siguiente:

```
FUNCTION Ident(parámetros): Tipo;    Cabecera de la función
      CONST {. . .}                  Declaración de constantes locales
      TYPE {. . .}                   Definición de tipos locales
      VAR {. . .}                    Declaración de variables locales
      FUNCTION {. . .}               Declaración de funciones locales
      PROCEDURE {. . .}              Declaración de procedimientos locales
      BEGIN
      {. . .}      Cuerpo de la función  (asignación de un valor a la función)
      END;
```

7.3.2. Ejemplo de función

Un ejemplo de declaración de una función que no tiene declaraciones locales sería:

```
function volumen(radio:real):real;
  begin
  volumen:=(4/3)*pi*radio*radio*radio
  end;
```

Y un ejemplo de llamada a la función anterior en el cuerpo principal de un programa:

```
program esfera;
var x:real;
function volumen(radio:real):real;
  begin
  volumen:=(4/3)*pi*radio*radio*radio
  end;
begin
write('Introduce un valor para el radio: ');
readln(x);
writeln('Volumen de la esfera de r = ',x,' es : ',volumen(x))
end.
```

7.4. IDENTIFICADORES LOCALES Y GLOBALES

Dentro de un programa pueden estar declarados diferentes identificadores de constantes, tipos de dato, variables, así como de procedimientos y funciones. Asimismo, en estas rutinas pueden incluirse declaraciones de otros elementos. Se dice que un identificador es **global** si está declarado en la sección de declaraciones del programa. Se dice que un identificador es **local** si está declarado dentro de una rutina, bien como parámetro o bien en la sección de declaraciones, y sólo está definido y puede utilizarse dentro de la propia rutina.

El identificador asociado al programa en la cabecera es un caso especial de identificador, ya que no es global: no está declarado en la sección de declaraciones del programa, no puede referenciarse posteriormente en el código fuente pero no puede volverse a declarar en la sección de declaraciones del programa. También puede definirse un identificador global como aquel que no está definido dentro de una función o procedimiento.

Se verán varios ejemplos de todos ellos en el siguiente programa:

```
program principal;
const nmax=100;
var x,y,z:real
function f1(a:real):real;
  var k:integer;
  begin
  { Cuerpo de la funcion . . . }
  end;
begin
{ Cuerpo del programa . . . }
end.
```

`nmax`, `x`, `y`, `z` y `f1` son identificadores globales, mientras que `a` y `k` son identificadores locales de la función `f1`. Si existen varias rutinas, en cada una de ellas puede existir un grupo de identificadores locales. Por ejemplo, en el programa de la figura 7.4:

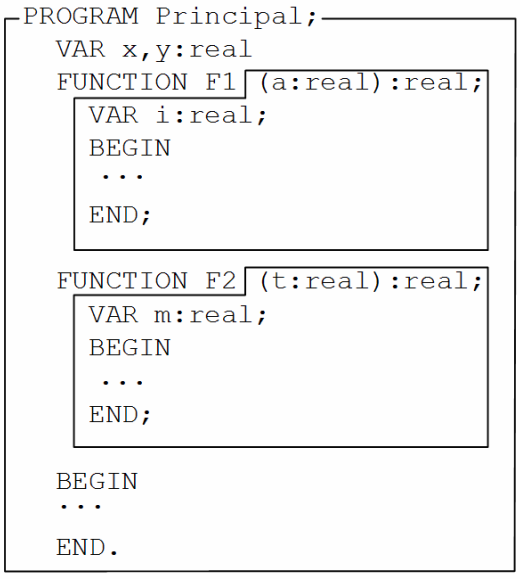

Figura 25. Declaración y estructura de un programa con dos funciones

Los identificadores del programa de la Figura 25 se clasifican según la Tabla 20.

Tabla 20. Identificadores del programa

Identificadores globales del programa	`x`, `y`, `f1` y `f2`
Identificadores locales de `f1`	`a` y `i`
Identificadores locales de `f2`	`t` y `m`

En el ejemplo de programa de la Figura 25, dentro de las funciones `F1` y `F2` pueden emplearse las variables globales del programa `x` e `y`, pero en el cuerpo del programa no pueden emplearse las variables locales `i` y `m` o los parámetros `a` y `t` de las funciones. Éstos sólo pueden emplearse dentro de sus respectivas funciones, `F1` y `F2`, exclusivamente. Estas rutinas delimitan unos niveles o bloques fuera de los cuales no puede utilizarse lo declarado dentro de ellas.

El *ámbito* de uso de cualquier identificador que se declare en una rutina, está restringido al nivel o bloque donde se ha declarado y, a partir de su declaración en la secuencia de código, nunca antes. En la figura 7.4 cada ámbito se puede corresponder con los póligonos dibujados: uno para el del programa principal y dos más para cada una de las rutinas.

7.5. REGLA DE USO DE IDENTIFICADORES LOCALES Y GLOBALES

Las normas de utilización de identificadores dentro de un programa pueden resumirse en la siguiente regla:

> *Dentro de cada bloque o rutina (o en el cuerpo del programa principal) puede emplearse:*
> *- lo declarado **previamente** (**localmente** en el caso de una rutina o **globalmente** en el caso del programa principal) y*
> *- lo que esté declarado **previamente** en los niveles superiores en los que esté anidada dicha rutina.*
> *El programa principal se sitúa en el nivel superior, las subrutinas definidas en el programa principal en el nivel inmediatamente inferior y así sucesivamente.*

Se define el *ámbito* de un identificador como el conjunto de los bloques (cuerpo del programa y/o rutinas) en los que puede ser utilizado.

7.5.1. Caso de llamadas a rutinas

Las llamadas a las rutinas pueden realizarse, no sólo desde el cuerpo del programa principal, sino también desde el cuerpo de los propios procedimientos y funciones, siempre y cuando la rutina a la que se realice la llamada esté previamente declarada. Por ejemplo, dadas la estructura y declaraciones de un programa mostradas en la Figura 25.

Desde el cuerpo del programa principal pueden hacerse llamadas a las funciones `F1` y `F2`. Desde el cuerpo de la función `F2` pueden realizarse llamadas a `F1` pero no al revés, ya que la función `F2` se declara después que `F1`. Es decir, sólo puede emplearse lo que esté previamente declarado en el código del programa.

7.5.2. Anidamiento de rutinas

Por otro lado, en un programa pueden existir subrutinas definidas dentro de otras. Esto es lo que se conoce como *anidamiento* y da lugar a la existencia de una jerarquía de niveles y de una serie de reglas que permiten determinar qué elementos se pueden utilizar en un determinado

punto del programa. Por ejemplo, en el programa de la Figura 26 se han declarado varias rutinas anidadas.

```
PROGRAM Principal;
VAR a,b :Integer;
PROCEDURE Pro1 (c:Integer);
  VAR d,e :Integer;
  PROCEDURE Pro2 (f:Integer);
    VAR g :Integer;
    BEGIN … END;
  PROCEDURE Pro3 (h:Integer);
    VAR i :Integer;
    BEGIN … END;
  BEGIN … END;
PROCEDURE Pro4 (j:Integer);
  VAR a :Integer;
  PROCEDURE Pro5 (l:Integer);
    VAR m,j :Integer;
    BEGIN … END;
  PROCEDURE Pro6 (n:Integer);
    VAR b :Integer;
    BEGIN … END;
  BEGIN...END;
BEGIN
  ...
END.
```

Figura 26. Declaración y estructura de un programa con rutinas anidadas

En el programa principal pueden emplearse las variables globales `a` y `b` y los procedimientos, también denominados globales, `pro1` y `pro4`.

En el cuerpo del procedimiento pro1 pueden utilizarse el parámetro formal `c`, las variables locales `d` y `e` y los procedimientos locales `pro2` y `pro3` y el propio procedimiento `pro1` (lo que, como se verá más adelante, se denomina *recursividad*). También pueden emplearse las variables globales `a` y `b`.

En el cuerpo del procedimiento `pro2` pueden emplearse el parámetro formal `f`, la variable local `g`, el propio procedimiento `pro2`, así como todo lo ya previamente declarado en los niveles superiores que lo contienen, en este caso, en el procedimiento `pro1` (parámetro formal `c` y variables locales `d` y `e`) y en el cuerpo del programa principal (variables globales `a` y `b` y el procedimiento `pro1`).

Dentro del procedimiento `pro3` pueden utilizarse el parámetro formal `h`, la variable local `i`, las variables locales `d` y `e`, el parámetro formal `c` y el procedimiento local `pro2` declarados en el procedimiento `pro1`, el propio procedimiento `pro3`, así como las variables globales `a` y `b` y el procedimiento `pro1`.

En el procedimiento `pro4` pueden emplearse el parámetro formal `j`, la variable local `a` y los procedimientos `pro5` y `pro6` y el propio procedimiento `pro4`. Además pueden emplearse la variable global `b` y el procedimiento `pro1`.

Se deja como ejercicio al lector, determinar qué elementos pueden emplearse en el cuerpo de los procedimientos `pro5` y `pro6`.

7.6. VENTAJAS DE UTILIZAR PROCEDIMIENTOS Y FUNCIONES

Las ventajas de utilizar subrutinas pueden resumirse en que:

a) Facilitan el diseño modular de los programas.
b) Se declaran una sóla vez pero pueden llamarse y ejecutarse más de una vez en un programa.
c) Pueden utilizarse en cualquier programa si están incluidos en unidades independientes como ya se explicará más adelante.

7.7. RECURSIVIDAD o RECURRENCIA

La recursividad es la propiedad mediante la cual un subprograma o rutina puede llamarse a sí mismo para realizar una tarea, es decir, se define en función de sí mismo. Aunque puede emplearse en TurboPascal, no todos los lenguajes de programación admiten el uso de la recursividad. Utilizando la recursividad, la resolución de un problema se reduce a uno esencialmente igual pero *algo menos* complejo. Es una herramienta muy potente y útil en la resolución de problemas. Por ejemplo, puede usarse en la definición matemática del factorial de un número:

Sin recursividad:	n!=1	si n=0
	n!=n*(n-1)*(n-2)*...*1	si n>0
Recursivamente:	n!=1	si n=0
	n!=n*(n-1)!	si n>0

A continuación, se muestra un ejemplo de dos formas de construir una función que devuelva el factorial de un número en TurboPascal:

Función no recursiva o iterativa

```
function factorial(n:word):word;
     var i,aux:word;
     begin
     aux:=1;
     for i:=1 to n do aux:=i*aux;
     factorial:=aux
     end;
```

Función recursiva

```
function factorial(n:word):word;
   begin
   if n>0
      then factorial:=n*factorial(n-1)
      else factorial:=1
   end;
```

Figura 27. Funciones que calculan el factorial de forma no recursiva y recursiva.

El seguimiento de la llamada a la función recursiva para calcular 4! = `factorial(4)` sería:

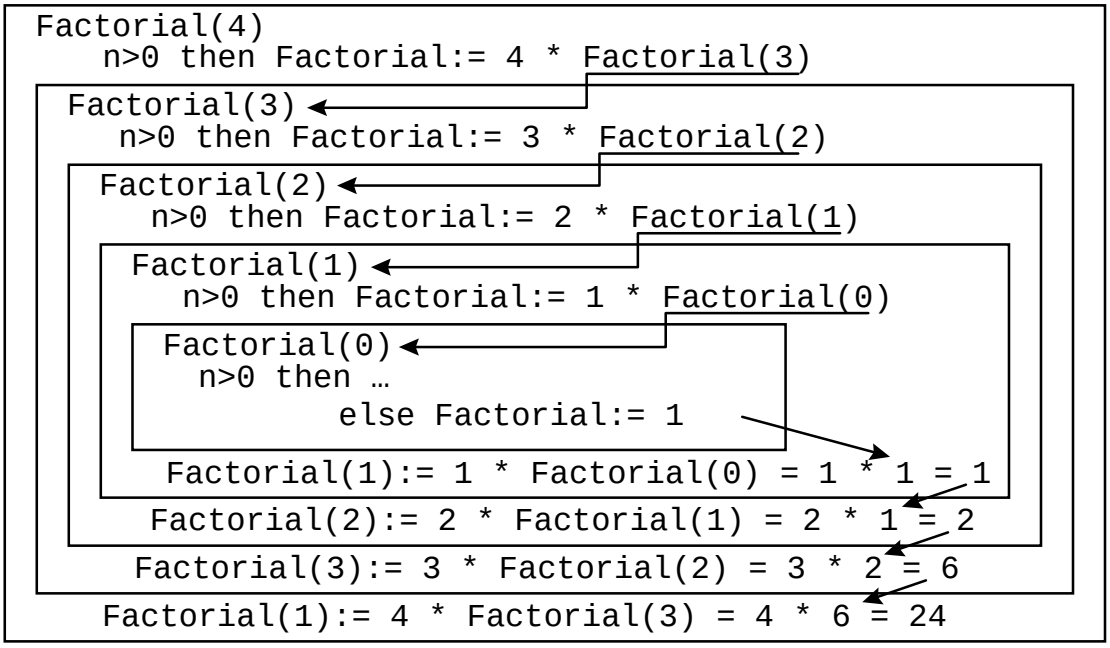

Figura 28. Modo de operación de la función factorial recursiva.

Las rutinas recursivas necesitan siempre una condición de salida: es necesario que se detenga después de un cierto número de llamadas para evitar un bucle sin fin.

NOTA: Como el intervalo de representación del tipo de dato `Word` es relativamente pequeño (0 ... 65535), la función `factorial` sólo puede calcular y devolver un resultado hasta el factorial de 8 (8! = 40320). Para soslayar este inconveniente puede emplearse el tipo `Longint` o `Real` de la siguiente manera:

Función no recursiva

```
function factorial(n:word):real;
     var i:word; aux:real;
     begin
     aux:=1;
     for i:=1 to n do aux:=i*aux;
     factorial:=aux
     end;
```

Función recursiva

```
function factorial(n:word):real;
   begin
   if n>0
      then factorial:=n*factorial(n-1)
      else factorial:=1
   end;
```

Figura 29. Funciones que calculan el factorial de forma no recursiva y recursiva de tipo real.

Otro ejemplo de aplicación de la recursividad puede encontrarse en el algoritmo de Euclides: el **m**áximo **c**omún **d**ivisor de dos números enteros es el entero mayor que divide a ambos. El algoritmo de Euclides dice que el m.c.d. de dos números enteros es el último resto no nulo de la serie consecutiva de divisiones enteras entre divisores y restos. Para calcular el m.c.d. de los enteros n y m:

1	2	3		k
n : m r1 q1	m : r1 r2 q2	r1 : r2 r3 q3	. . . . . .	r_{k-2} : r_{k-1} (= mcd) r_k=0 q_k

Podrían construirse funciones que calcularan el m.c.d. siguiendo diferentes algoritmos. Se presentan a continuación dos ejemplos, una iteración y una recursión:

1. Mediante una función no recursiva:

```
function mcd(n,m:word):word;
     var r:word;
     begin
     r:=1;
     while r<>0 do
          begin
          r:=n mod m;
          n:=m;
          m:=r
          end;
     mcd:=n
     end;
```

2. Mediante una función recursiva:

```
function mcd(n,m:word):word;
var r:word;
     begin
     r:=n mod m;
     if r=0 then mcd:=m
       else mcd:=mcd(m,r)
     end;
```

7.7.1. Tipos o formas de recursividad

Existen dos tipos o formas de recursividad en TurboPascal:

1. Directa o simple: Un subprograma se llama a sí mismo directamente. Es el tipo de recursividad empleado en los ejemplos vistos hasta ahora.
2. Indirecta o mutua: un subprograma A llama a otro subprograma B y éste a su vez llama al subprograma A. El problema es que una subrutina no puede ser utilizada antes de ser declarada. ¿Cómo se resuelve este problema? Los problemas de definición se resuelven con la palabra reservada **`forward`**. Este identificador indica al compilador que la rutina a la que se hace mención se declarará posteriormente a su llamada en la declaración de la segunda rutina.

7.7.2. Recursividad indirecta o mútua

La sintaxis de la declaración de rutinas que emplean la recursividad indirecta o mútua es la siguiente:

```
cabecera del primer subprograma; forward;
cabecera del segundo subprograma;
bloque del segundo subprograma;
identificador del primer subprograma;
bloque del primer subprograma;
```

A continuación se muestra un ejemplo de programa en el que se emplea la recursividad indirecta:

```
program rmutua;
var n:word;
function impar(i:word):boolean;forward;
function par(i:word):boolean;
  begin
  if i>0 then par:=impar(i-1)
         else par:=true
  end;
function impar;
  begin
  if i>0 then impar:=par(i-1)
         else impar:=false
  end;
begin
write('Introduce un numero mayor que 0: ');
readln(n);
writeln('Es ',par(n),' que ',n,' sea par')
end.
```

7.7.3. Consejos de empleo de la recursividad

La recursividad puede ser una herramienta muy potente en programación, pero es conveniente tener en cuenta lo siguiente:

a) No siempre la recursividad es la mejor manera de resolver un problema.
b) Si se construye una rutina recursiva hay que asegurarse de que existe una salida o punto en el que finalizan las llamadas recursivas a la rutina. Deben evitarse las rutinas con recursión infinita.
c) La recursión no sólo debe ser finita sino también relativamente pequeña para evitar problemas de saturación de memoria.

7.8. RUTINAS DE LA UNIDAD SYSTEM

Los procedimientos y funciones más importantes incluidas en la unidad `System` son:

abs(x):tipopropiodex;	Función que devuelve el valor absoluto de una expresión numérica real o entera
arctan(numero:real):real;	Función que devuelve el arco tangente de un dato
chr(numero:byte):char;	Función que devuelve el carácter ASCII correspondiente al número
cos(numero:real):real;	Función que devuelve el coseno de un ángulo expresado en radianes
dec(x [;decremento:longint]);	Procedimiento que decrementa el valor de la variable X en una unidad o, si se especifica, en la cantidad indicada por Decremento
exp(numero:real):real;	Función que devuelve el valor de elevar el número e al parámetro Numero
frac(numero: real):real;	Función que devuelve la parte fraccionaria contenida en el dato de la variable
inc(x [;decremento:longint]);	Procedimiento que incrementa el valor de la variable X en una unidad o, si se especifica, en la cantidad indicada por Incremento
int(numero: real):real;	Función que devuelve la parte entera del número especificado como parámetro
length(caden:string):integer;	Función que devuelve la longitud de la cadena de caracteres almacenada
ln(numero:real):real;	Función que devuelve el logaritmo neperiano del argumento
odd(n:longint):boolean;	Función que devuelve el valor cierto si el argumento es impar
ord (x): longint;	Función que devuelve la posición que ocupa el dato X en su tipo ordinal
pi:real;	Función que devuelve el valor numérico real de π
pred(x):tipopropiodex;	Función que devuelve el valor anterior de una variable
random:real; ó **random**(numero: word):word;	Función que devuelve un número real aleatorio entre cero y uno o entero aleatorio entre cero y el especificado en el parámetro en la llamada (sin incluir)
randomize:real;	Procedimiento que inicializa el generador de números aleatorios con un valor aleatorio obtenido del reloj del sistema
read(f:<*fichero*>;v); ó **read**(v);	Procedimiento que lee un dato de un fichero o del teclado, y lo almacena en una variable v
readln(f:<*fichero*>;v); ó **readln**(v);	Procedimientos similares a los anteriores, salvo que en el primer formato salta al principio de la siguiente línea del archivo de texto, después de leer el registro. En el segundo formato salta al principio de la siguiente línea de pantalla después de leer el dato
round(numero:real):longint;	Función que redondea un valor real y devuelve el valor entero más próximo
sin(numero:real):real;	Función que devuelve el valor del seno de Número
sqr(x):tipopropiodex;	Función que devuelve el valor del cuadrado de x
sqrt(x:real):real;	Función que devuelve el valor de la raíz cuadrada de x
succ(x):tipopropiodex;	Función que devuelve el elemento posterior a x en su tipo ordinal
trunc(numero:real):longint;	Función que convierte un valor real en entero, truncando los decimales
upcase(carácter:char):char;	Función que convierte en mayúsculas el carácter especificado como parámetro
write(f:<*fichero*>;dato); ó **write**(dato);	Procedimiento que escribe el contenido de una expresión en un fichero o en la pantalla
writeln(f:<*fichero*>;dato); ó **writeln**(dato);	Similar a **Write**, pero provocando un salto de línea después de escribir

Para utilizar lo declarado dentro de la unidad `system` en un programa, **NO** es necesario incluir en éste, la correspondiente sentencia de declaración de uso de la unidad `system`.

7.9. RUTINAS DE LA UNIDAD CRT

Los procedimientos y funciones incluidas en la unidad `Crt` son los siguientes:

assigncrt(var f:text);	Procedimiento que asocia la pantalla con el identificador de la variable archivo
clreol;	Procedimiento que borra una línea a partir de la posición actual del cursor
clrscr;	Procedimiento que borra la pantalla y sitúa el cursor en la esquina superior izquierda
delay(x:word);	Procedimiento que detiene la ejecución del programa el número indicado de milisegundos
delline;	Procedimiento que borra la línea donde se encuentra al cursor
gotoxy(x,y:byte);	Procedimiento que sitúa el cursor en la pantalla en las coordenadas columna X y fila Y
highvideo;	Procedimiento que hace que el texto en pantalla se escribe en alta intensidad
insline;	Procedimiento que inserta una línea en blanco en la posición donde se esté el cursor
keypressed:boolean;	Función que devuelve el valor true si se ha pulsado una tecla, y false en caso contrario
lowvideo;	Procedimiento que hace que el texto de la pantalla se escribe sin resaltar
normvideo;	Procedimiento que hace que, a partir de su llamada, el texto de la pantalla se escriba en el modo (resaltado o no) inicial
nosound;	Procedimiento que desactiva el altavoz cesando el sonido que pudiere estar emitiendo
readkey:char;	Función que detiene la ejecución del programa hasta que se pulsa una tecla, devolviendo su valor
sound(hertzios:word);	Procedimiento que genera un sonido a través del altavoz de la frecuencia indicada
textbackground(c:byte);	Procedimiento que establece el color de fondo indicado
textcolor(color:byte);	Procedimiento que establece el color de los caracteres
textmode(modo:integer);	Procedimiento que establece un modo de texto específico
wherex:byte;	Función que devuelve el nº de columna en que se encuentra el cursor
wherey:byte;	Función que devuelve el número de fila en que se encuentra el cursor
window(x1,y1,x2,y2:byte);	Procedimiento que crea una ventana de texto en la pantalla cuya esquina superior izquierda con coordenadas (X1,Y1) y la inferior derecha (X2,Y2)

Bibliografía básica

- **García-Beltrán, A., Martínez, R.** y **Jaén, J.A.** *Métodos Informáticos en TurboPascal*, Ed. Bellisco, 2ª edición, Madrid, 2002
- Borland Pascal with Objects - Language Guide, Editorial Borland, 1992
- Borland Pascal with Objects - Programmer's Reference, Editorial Borland, 1992
- **Duntemann, J.** *La Biblia de TurboPascal*, Anaya Multimedia, Madrid, 1991