# Capítulo 6

# Subprogramas (subalgoritmos): Funciones



## INTRODUCCIÓN

La resolución de problemas complejos se facilita considerablemente si se dividen en problemas más pequeños (subproblemas). *La solución de estos subproblemas se realiza con subalgoritmos*. El uso de subalgoritmos permite al programador desarrollar programas de problemas complejos utilizando el método descendente introducido en los capítulos anteriores. *Los subalgoritmos (subprogramas) pueden ser de dos tipos: funciones y procedimientos o subrutinas*. Los subalgoritmos son unidades de programa o módulos que están diseñados para ejecutar alguna tarea específica. Estas funciones y procedimientos se escriben solamente una vez, pero pueden ser referenciados en diferentes puntos de un programa, de modo que se puede evitar la duplicación innecesaria del código.

Las unidades de programas en el estilo de programación modular son independientes; el programador puede escribir cada módulo y verificarlo sin preocuparse de los detalles de otros módulos. Esto facilita considerablemente la localización de un error cuando se produce. Los programas desarrollados de este modo son normalmente también más fáciles de comprender, ya que la estructura de cada unidad de programa puede ser estudiada independientemente de las otras unidades de programa. En este capítulo se describen las *funciones y procedimientos*, junto con los conceptos de *variables locales y globales,* así como *parámetros*. Se introduce también el concepto de *recursividad* como una nueva herramienta de resolución de problemas.

## 6.1. INTRODUCCIÓN A LOS SUBALGORITMOS O SUBPROGRAMAS

Un método ya citado para solucionar un problema complejo es dividirlo en subproblemas —problemas más sencillos— y a continuación dividir estos subproblemas en otros más simples, hasta que los problemas más pequeños sean fáciles de resolver. Esta técnica de dividir el problema principal en subproblemas se suele denominar "*divide y vencerás" (divide and conquer)*. Este método de diseñar la solución de un problema principal obteniendo las soluciones de sus subproblemas se conoce como *diseño descendente (top-down design)*. Se denomina descendente, ya que se inicia en la parte superior con un problema general y el diseño específico de las soluciones de los subproblemas. Normalmente las partes en que se divide un programa deben poder desarrollarse independientemente entre sí.

Las soluciones de un diseño descendente pueden implementarse fácilmente en lenguajes de programación de alto nivel, como C/C++, Pascal o FORTRAN. Estas partes independientes se denominan *subprogramas* o *subalgoritmos* si se emplean desde el concepto algorítmico.

La correspondencia entre el diseño descendente y la solución por computadora en términos de programa principal y sus subprogramas se analizará a lo largo de este capítulo.

Consideremos el problema del cálculo de la superficie (área) de un rectángulo. Este problema se puede dividir en tres subproblemas:

```
subproblema 1: entrada de datos de altura y base.
subproblema 2: cálculo de la superficie.
subproblema 3: salida de resultados.
```

El algoritmo correspondiente que resuelve los tres *subproblemas* es:

```
leer (altura, base)                    //entrada de datos
area ← base * altura                   //cálculo de la superficie
escribir(base, altura, area)           //salida de resultados
```

El método descendente se muestra en la Figura 6.1.

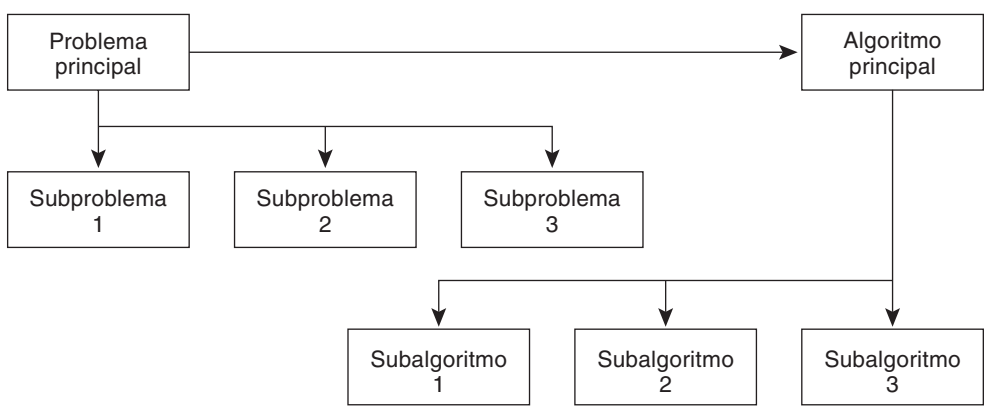



**Figura 6.1.** Diseño descendente.

El problema principal se soluciona por el correspondiente **programa** o **algoritmo principal** —también denominado *controlador o conductor (driver)*— y la solución de los subproblemas mediante **subprogramas**, conocidos como **procedimientos (subrutinas)** o **funciones**. Los subprogramas, cuando se tratan en lenguaje algorítmico, se denominan también *subalgoritmos*.

Un subprograma puede realizar las mismas acciones que un programa: 1) aceptar datos, 2) realizar algunos cálculos y 3) devolver resultados. Un subprograma, sin embargo, se utiliza por el programa para un propósito específico. El subprograma recibe datos desde el programa y le devuelve resultados. Haciendo un símil con una oficina, el problema es como el jefe que da instrucciones a sus subordinados —subprogramas—; cuando la tarea se termina, el subordinado devuelve sus resultados al jefe. Se dice que el programa principal *llama* o *invoca* al subprograma. El subprograma ejecuta una tarea, a continuación *devuelve* el control al programa. Esto puede suceder en diferentes

lugares del programa. Cada vez que el subprograma es llamado, el control retorna al lugar desde donde fue hecha la llamada (Figura 6.2). Un subprograma puede llamar a su vez a sus propios subprogramas (Figura 6.3). Existen —como ya se ha comentado— dos tipos importantes de subprogramas: *funciones* y *procedimientos o subrutinas*.

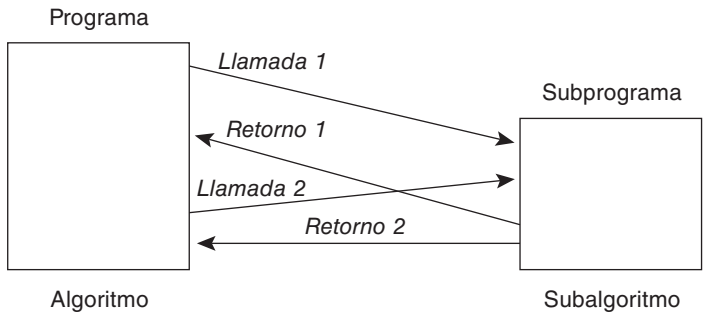



**Figura 6.2.** Un programa con un subprograma: función y procedimiento o subrutina, según la terminología específica del lenguaje: subrutina en BASIC y FORTRAN, *función* en C, C++, método en Java o C#, *procedimiento o función* en Pascal.

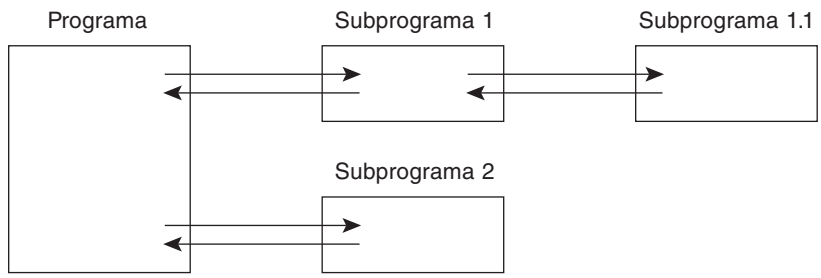



**Figura 6.3.** Un programa con diferentes niveles de subprogramas.

## 6.2. FUNCIONES

Matemáticamente una función es una operación que toma uno o más valores llamados *argumentos* y produce un valor denominado *resultado* —valor de la función para los argumentos dados—. Todos los lenguajes de programación tienen *funciones incorporadas*, *intrínsecas* o *internas* —en el Capítulo 3 se vieron algunos ejemplos—, y *funciones definidas por el usuario*. Así, por ejemplo

$$f(x) = \frac{x}{1 + x^2}$$

donde $f$ es el nombre de la función y $x$ es el argumento. Obsérvese que ningún valor específico se asocia con $x$; es un *parámetro formal* utilizado en la definición de la función. Para evaluar $f$ debemos darle un *valor real o actual* a $x$; con este valor se puede calcular el resultado. Con $x = 3$ se obtiene el valor 0.3 que se expresa escribiendo

$$f(3) = 0.3$$

$$f(3) = \frac{3}{1 + 9} = \frac{3}{10} = 0.3$$

Una función puede tener varios argumentos. Por consiguiente,

$$f(x, y) = \frac{x - y}{\sqrt{x} + \sqrt{y}}$$

es una función con dos argumentos. Sin embargo, solamente un único valor se asocia con la función para cualquier par de valores dados a los argumentos.

Cada lenguaje de programación tiene sus propias funciones incorporadas, que se utilizan escribiendo sus nombres con los argumentos adecuados en expresiones tales como

```
raiz2(A+cos(x))
```

Cuando la expresión se evalúa, el valor de *x* se da primero al subprograma (función) coseno y se calcula *cos(x)*. El valor de *A+cos(x)* se utiliza entonces como argumento de la función *raiz2* (raíz cuadrada), que evalúa el resultado final.

Cada función se evoca utilizando su nombre en una expresión con los argumentos actuales o reales encerrados entre paréntesis.

Las funciones incorporadas al sistema se denominan *funciones internas o intrínsecas* y las funciones definidas por el usuario, *funciones externas*. Cuando las funciones estándares o internas no permiten realizar el tipo de cálculo deseado es necesario recurrir a las funciones externas que pueden ser definidas por el usuario mediante una *declaración de función*.

A una función no se le llama explícitamente, sino que se le invoca o referencia mediante un nombre y una lista de parámetros actuales. El algoritmo o programa llama o invoca a la función con el nombre de esta última en una expresión seguida de una lista de argumentos que deben coincidir en cantidad, tipo y orden con los de la función que fue definida. La función devuelve un único valor.

Las funciones son diseñadas para realizar tareas específicas: toman una lista de valores —llamados *argumentos*— y devolver un único valor.

## 6.2.1. Declaración de funciones

La declaración de una función requiere una serie de pasos que la definen. Una función como tal subalgoritmo o subprograma tiene una constitución similar a los algoritmos, por consiguiente, constará de una cabecera que comenzará con el tipo del valor devuelto por la función, seguido de la palabra **función** y del nombre y argumentos de dicha función. A continuación irá el *cuerpo* de la función, que será una serie de acciones o instrucciones cuya ejecución hará que se asigne un valor al nombre de la función. Esto determina el valor particular del resultado que ha de devolverse al programa llamador.

La declaración de la función será;

```
<tipo_de_resultado> funcion <nombre_fun> (lista de parametros)
[declaraciones locales]
inicio
  <acciones>      //cuerpo de la funcion
  devolver (<expresion>)
fin_función
```

| | |
|---|---|
| `lista de parámetros` | lista de *parámetros formales* o *argumentos*, con uno o más argumentos de la siguiente forma:<br>`({E\|S\|E/S} tipo_de_datoA: parámetro 1[, parámetro 2]...;`<br>`{E\|S\|E/S} tipo_de_datoB: parámetro x[, parámetro y]...)` |
| *`nombre_func`* | *nombre asociado con la función, que será un nombre de identificador válido* |
| *`<acciones>`* | instrucciones que constituyen la definición de la funcion y que debe contener una única instrucción: **devolver** `(<expresion>)`; *expresión* sólo existe si la función se ha declarado con valor de retorno y *expresión* es el valor devuelto por la función |
| *`tipo_de_resultado`* | tipo del resultado que devuelve la función |

***Sentencia* `devolver (return)`**

La sentencia **`devolver`** (**`return`**, volver) se utiliza para regresar de una función (un *método* en programación orientada a objetos); **`devolver`** hace que el control del programa se transfiera al llamador de la función (método). Esta sentencia se puede utilizar para hacer que la ejecución regrese de nuevo al llamador de la función.

### *Regla*

La sentencia `devolver` termina inmediatamente la función en la cual se ejecuta.

Por ejemplo, la función:

$$f(x) = \frac{x}{1 + x^2}$$

se definirá como:

```
real función F(E real:x)
inicio
  devolver (x/(1+x*x))
fin_función
```

Otro ejemplo puede ser la definición de la función trigonométrica, cuyo valor es

$$tan(x) = \frac{sen(x)}{cos(x)}$$

donde *sen*(*x*) y *cos*(*x*) son las funciones seno y coseno —normalmente funciones internas—. La declaración de la función es

```
real función tan (E real:x)
//funcion tan igual a sen(x)/cos(x), angulo x en radianes
inicio
  devolver (sen(x)/cos(x))
fin_función
```

Observe que se incluye un comentario para describir la función. Es buena práctica incluir documentación que describa brevemente lo que hace la función, lo que representan sus parámetros o cualquier otra información que explique la definición de la función. *En aquellos lenguajes de programación —como Pascal— que exigen sección de declaraciones, éstas se situarán al principio de la función.*

Para que las acciones descritas en un subprograma función sean ejecutadas, se necesita que éste sea invocado desde un programa principal o desde otros subprogramas a fin de proporcionarle los argumentos de entrada necesarios para realizar esas acciones.

Los argumentos de la declaración de la función se denominan *parámetros formales, ficticios o mudos* (“dummy”); son nombres de variables, de otras funciones o procedimientos y que sólo se utilizan dentro del cuerpo de la función. Los argumentos utilizados en llamada a la función se denominan *parámetros actuales*, que a su vez pueden ser constantes, variables, expresiones, valores de funciones o nombres de funciones o procedimientos.

## 6.2.2. Invocación a las funciones

Una función puede ser llamada de la forma siguiente:

```
nombre_función     (lista de parametros actuales)
```

| | |
|---|---|
| `nombre_función` | función que llama |
| *`lista de parametros actuales`* | constantes, variables, expresiones, valores de funciones. nombres de funciones o procedimientos |

Cada vez que se llama a una función desde el algoritmo principal se establece automáticamente una correspondencia entre los parámetros formales y los parámetros actuales. Debe haber exactamente el mismo número de parámetros actuales que de parámetros formales en la declaración de la función y se presupone una correspondencia uno a uno de izquierda a derecha entre los parámetros formales y los actuales.

Una llamada a la función implica los siguientes pasos:

1. A cada parámetro formal se le asigna el valor real de su correspondiente parámetro actual.
2. Se ejecuta el cuerpo de acciones de la función.
3. Se devuelve el valor de la función y se retorna al punto de llamada.

---

**EJEMPLO 6.1**

*Definición de la función: $y = x^n$ (potencia n de x)*

```
real : función potencia(E real:x;E entero:n)
var
  entero: i, y
inicio
  y ← 1
  desde i ← 1 hasta abs(n) hacer
    y ← y*x
  fin_desde
  si n < 0 entonces
    y ← 1/y
  fin_si
  devolver (y)
fin_función
```

`abs(n)` es la función valor absoluto de *`n`* a fin de considerar exponentes positivos o negativos.

***Invocación de la función***

```
z ← potencia (2.5, -3)
```

*parámetros actuales*

***Transferencia de información***

```
x = 2.5   n = -3
z = 0.064
```

---

**EJEMPLO 6.2**

*Función potencia para el cálculo de `N` elevado a `A`. El número `N` deberá ser positivo, aunque podrá tener parte fraccionaria, `A` es un real.*

```
algoritmo Elevar_a_potencia
var
  real : a, n
```

```
inicio
   escribir('Deme numero positivo ')
   leer(n)
   escribir('Deme exponente ')
   leer(a)
   escribir('N elevado a =', potencia(n, a))
fin

real función potencia (E real: n, a)
inicio
  devolver(EXP(a * LN(n)))
fin_función
```

### EJEMPLO 6.3

*Diseñar un algoritmo que contenga un subprograma de cálculo del factorial de un número y una llamada al mismo.*

Como ya es conocido por el lector el algoritmo factorial, lo indicaremos expresamente.

```
entero función factorial(E entero:n)
var
  entero: i,f
  //advertencia, segun el resultado, f puede ser real
inicio
  f ← 1
  desde i ← 1 hasta n hacer
    f ← f * i
  fin_desde
  devolver (f)
fin_función
```

y el algoritmo que contiene un subprograma de cálculo del factorial de un número y una llamada al mismo:

```
algoritmo función_factorial
var entero: x, y, numero

inicio
  escribir ('Deme un numero entero y positivo')
  leer(numero)
  x ← factorial(numero)
  y ← factorial(5)
  escribir(x, y)
fin
```

En este caso los parámetros actuales son: una variable (*número*) y una constante (*5*).

### EJEMPLO 6.4

*Realizar el diseño de la función $y = x^3$ (cálculo del cubo de un número).*

```
algoritmo prueba
var
  entero: N
```

```
inicio    //Programa principal
   N ← cubo(2)
   escribir ('2 al cubo es', N)
   escribir ('3 al cubo es', cubo(3))
fin

entero función cubo(E entero: x)
inicio
  devolver(x*x*x)
fin_función
```

La salida del algoritmo sería:

```
2 al cubo es 8
3 al cubo es 27
```

Las funciones pueden tener muchos argumentos, pero solamente un resultado: *el valor de la función*. Esto limita su uso, aunque se encuentran con frecuencia en cálculos científicos. Un concepto más potente es el proporcionado por el subprograma procedimiento que se examina en el siguiente apartado.

### EJEMPLO 6.5

*Algoritmo que contiene y utiliza unas funciones (seno y coseno) a las que les podemos pasar el ángulo en grados.*

```
algoritmo Sen_cos_en_grados
var real : g

inicio
  escribir('Deme ángulo en grados')
  leer(g)
  escribir(seno(g))
  escribir(coseno(g))
fin

real función coseno (E real : g)
inicio
  devolver(COS(g*2*3.141592/360))
fin_función

real: función seno (E real g)
inicio
  devolver( SEN(g*2*3.141592/360))
fin_función
```

### EJEMPLO 6.6

*Algoritmo que simplifique un quebrado, dividiendo numerador y denominador por su máximo común divisor.*

```
algoritmo Simplificar_quebrado
var
entero : n, d

inicio
  escribir('Deme numerador')
  leer(n)
```

```
  escribir('Deme denominador')
  leer(d)
  escribir(n, '/', d, '=', n div mcd(n, d),'/', d div mcd(n, d))
fin
  entero función mcd (E entero: n, d)
  var
    entero : r
  inicio
     r ← n MOD d
     mientras r <> 0 hacer
       n ← d
       d ← r
       r ← n MOD d
    fin_mientras
    devolver(d)
  fin_función
```

### EJEMPLO 6.7

*Supuesto que nuestro compilador no tiene la función seno. Podríamos calcular el seno de x mediante la siguiente serie:*

$$sen(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots \text{ (hasta 17 términos)}$$

*x* (ángulo en radianes).

El programa nos tiene que permitir el cálculo del seno de ángulos en grados mediante el diseño de una función seno(x), que utilizará, a su vez, las funciones `potencia(x,n)` y `factorial(n)`, que también deberán ser implementadas en el algoritmo.

Se terminará cuando respondamos `N` (**no**) a la petición de otro ángulo.

```
algoritmo Calcular_seno
var real : gr
    carácter : resp

inicio
  repetir
    escribir('Deme ángulo en grados')
    leer(gr)
    escribir('Seno(', gr, ')=', seno(gr))
    escribir('¿Otro ángulo?')
    leer(resp)
  hasta_que resp = 'N'
fin

real función factorial (E entero:n)
var
  real : f
  entero : i
inicio
  f ← 1
  desde i ← 1 hasta n hacer
  f ← f * i
  fin_desde
  devolver(f)
fin_función
```

```
real función potencia (E real:x; E entero:n)
var real : pot
  entero : i
inicio
  pot ← 1
  desde i ← 1 hasta n hacer
  pot ← pot * x
  fin_desde
  devolver(pot)
fin_función

real función seno (E real:gr)
var real : x, s
  entero : i, n
inicio
  x ← gr * 3.141592 / 180
  s ← x
  desde i ← 2 hasta 17 hacer
    n ← 2 * i - 1
    si i MOD 2 <> 0 entonces
      s ← s - potencia(x, n) / factorial(n)
    si_no
      s ← s + potencia(x, n) / factorial(n)
    fin_si
  fin_desde
  devolver(s)
fin_función
```

## 6.3. PROCEDIMIENTOS (SUBRUTINAS)

Aunque las funciones son herramientas de programación muy útiles para la resolución de problemas, su alcance está muy limitado. Con frecuencia se requieren subprogramas que calculen varios resultados en vez de uno solo, o que realicen la ordenación de una serie de números, etc. En estas situaciones la *función* no es apropiada y se necesita disponer del otro tipo de subprograma: el *procedimiento o subrutina*.

Un *procedimiento o subrutina*[1] es un subprograma que ejecuta un proceso específico. Ningún valor está asociado con el nombre del procedimiento; por consiguiente, no puede ocurrir en una expresión. Un procedimiento se llama escribiendo su nombre, por ejemplo, `SORT`, para indicar que un procedimiento denominado `SORT (ORDENAR)` se va a usar. Cuando se invoca el procedimiento, los pasos que lo definen se ejecutan y a continuación se devuelve el control al programa que le llamó.

**Procedimiento *versus* función**

Los procedimientos y funciones son subprogramas cuyo diseño y misión son similares; sin embargo, existen unas diferencias esenciales entre ellos.

1. Un procedimiento es llamado desde el algoritmo o programa principal mediante su nombre y una lista de parámetros actuales, o bien con la instrucción **`llamar_a (call)`**. Al llamar al procedimiento se detiene momentáneamente el programa que se estuviera realizando y el control pasa al procedimiento llamado. Después que las acciones del procedimiento se ejecutan, se regresa a la acción inmediatamente siguiente a la que se llamó.
2. Las funciones devuelven un valor, los procedimientos pueden devolver 0,1 o *n* valores y en forma de lista de parámetros.
3. El procedimiento se declara igual que la función, pero su nombre no está asociado a ninguno de los resultados que obtiene.

[1] En FORTRAN, la subrutina representa el mismo concepto que procedimiento. No obstante, en la mayor parte de los lenguajes el término general para definir un subprograma es procedimiento o simplemente subprograma.

La *declaración de un procedimiento* es similar a la de funciones.

```
procedimiento nombre [(lista de parámetros formales)]
  <acciones>
fin_procedimiento
```

Los parámetros formales tienen el mismo significado que en las funciones; los parámetros variables —en aquellos lenguajes que los soportan, por ejemplo, Pascal— están precedidos cada uno de ellos por la palabra **var** para designar que ellos obtendrán resultados del procedimiento en lugar de los valores actuales asociados a ellos.

El procedimiento se llama mediante la instrucción

```
[llamar_a] nombre [(lista de parámetros actuales)]
```

La palabra **llamar_a (call)** es opcional y su existencia depende del lenguaje de programación.

El ejemplo siguiente ilustra la definición y uso de un procedimiento para realizar la división de dos números y obtener el cociente y el resto.

***Variables enteras:***

```
Dividendo
Divisor
Cociente
Resto
```

***Procedimiento***

```
procedimiento division (E entero:Dividendo,Divisor; S entero: Cociente, Resto)

inicio
  Cociente ← Dividendo DIV Divisor
  Resto ← Dividendo - Cociente * Divisor
fin_procedimiento
```

***Algoritmo principal***

```
algoritmo aritmética
var
  entero: M, N, P, Q, S, T
inicio
  leer(M, N)
  llamar_a division (M, N, P, Q)
  escribir(P, Q)
  llamar_a division (M * N – 4, N + 1, S, T)
  escribir(S, T)
fin
```

## 6.3.1. Sustitución de argumentos/parámetros

La lista de parámetros, bien *formales* en el procedimiento o *actuales* (reales) en la llamada se conoce como *lista de parámetros*.

```
procedimiento demo
.
.
.
fin_procedimiento
```

o bien

```
procedimiento demo (lista de parametros formales)
```

y la instrucción llamadora

```
llamar_a demo (lista de parametros actuales)
```

Cuando se llama al procedimiento, cada parámetro formal toma como valor inicial el valor del correspondiente parámetro actual. En el ejemplo siguiente se indican la sustitución de parámetros y el orden correcto.

```
algoritmo demo
  //definición del procedimiento
    entero: años
    real: numeros, tasa
inicio
  ...
  llamar_a calculo(numero, años, tasa)
  ...
fin

procedimiento calculo(S real: p1; E entero: p2; E real: p3)
inicio
  p3 ... p1 ... p2
fin_procedimiento
```

Las acciones sucesivas a realizar son las siguientes:

1. Los parámetros reales sustituyen a los parámetros formales.
2. El cuerpo de la declaración del procedimiento se sustituye por la llamada del procedimiento.
3. Por último, se ejecutan las acciones escritas por el código resultante.

---

### EJEMPLO 6.8 (DE PROCEDIMIENTO)

*Algoritmo que transforma un número introducido por teclado en notación decimal a romana. El número será entero y positivo y no excederá de 3.000.*

#### *Sin utilizar programación modular*

```
algoritmo romanos
var entero : n,digito,r,j

inicio
    repetir
      escribir('Deme número')
      leer(n)
    hasta_que (n >= 0) Y (n <= 3000)
    r ← n
    digito ← r DIV 1000
```

```
  r ← r MOD 1000
  desde j ← 1 hasta digito hacer
    escribir('M')
  fin_desde
  digito ← r DIV 100
  r ← r MOD 100
  si digito = 9 entonces
    escribir('C', 'M')
  si_no
    si digito > 4 entonces
       escribir('D')
       desde j ← 1 hasta digito - 5 hacer
         escribir('C')
       fin_desde
     si_no
       si digito = 4 entonces
         escribir('C','D')
       si_no
         desde j ← 1 hasta digito hacer
           escribir('C')
         fin_desde
       fin_si
    fin_si
fin_si
digito ← r DIV 10
r ← r MOD 10
si digito = 9 entonces
    escribir('X', 'C')
      si_no
    si digito > 4 entonces
      escribir('L')
      desde j ← 1 hasta digito - 5 hacer
      escribir('X')
      fin_desde
    si_no
      si digito = 4 entonces
        escribir('X','L')
      si_no
         desde j ← 1 hasta digito hacer
            escribir('X')
         fin_desde
       fin_si
     fin_si
fin_si
digito ← r
si digito = 9 entonces
    escribir('I', 'X')
si_no
    si digito > 4 entonces
      escribir('V')
      desde j ← 1 hasta digito - 5 hacer
        escribir('I')
      fin_desde
    si_no
      si digito = 4 entonces
        escribir('I','V')
```

```
      si_no
        desde j ← 1 hasta digito hacer
          escribir('I')
        fin_desde
      fin_si
    fin_si
  fin_si
fin
```

**Mediante programación modular**

```
algoritmo Romanos
var entero : n, r, digito

inicio
  repetir
    escribir('Deme número')
    leer(n)
  hasta_que (n >= 0) Y (n <= 3000)
  r ← n
  digito ← r Div 1000
  r ← r MOD 1000
  calccifrarom(digito, 'M', ' ',' ')
  digito ← r Div 100
  r ← r MOD 100
  calccifrarom(digito, 'C', 'D', 'M')
  digito ← r Div 10
  r ← r MOD 10
  calccifrarom(digito, 'X', 'L', 'C')
  digito ← r
  calccifrarom(digito, 'I', 'V', 'X')
fin

procedimiento calccifrarom(E entero: digito; E caracter: v1, v2, v3)
var entero: j
inicio
  si digito = 9 entonces
    escribir( v1, v3)
  si_no
    si digito > 4 entonces
      escribir(v2)
      desde j ← 1 hasta digito - 5 hacer
        escribir(v1)
      fin_desde
    si_no
       si digito = 4 entonces
         escribir(v1, v2)
       si_no
         desde j ← 1 hasta digito hacer
           escribir(v1)
         fin_desde
       fin_si
    fin_si
  fin_si
fin_procedimiento
```

## 6.4. ÁMBITO: VARIABLES LOCALES Y GLOBALES

Las variables utilizadas en los programas principales y subprogramas se clasifican en dos tipos:

- *variables locales;*
- *variables globales*.

Una *variable local* es aquella que está declarada y definida dentro de un subprograma, en el sentido de que está dentro de ese subprograma y es distinta de las variables con el mismo nombre declaradas en cualquier parte del programa principal. *El significado de una variable se confina al procedimiento en el que está declarada*. Cuando otro subprograma utiliza el mismo nombre se refiere a una posición diferente en memoria. Se dice que tales variables son *locales* al subprograma en el que están declaradas.

Una *variable global* es aquella que está declarada para el programa o algoritmo principal, del que dependen todos los subprogramas.

La parte del programa/algoritmo en que una variable se define se conoce como *ámbito* o *alcance* (*scope*, en inglés).

El uso de variables locales tiene muchas ventajas. En particular, hace a los subprogramas independientes, con la comunicación entre el programa principal y los subprogramas manipulados estructuralmente a través de la lista de parámetros. Para utilizar un procedimiento sólo necesitamos conocer lo que hace y no tenemos que estar preocupados por su diseño, es decir, cómo están programados.

Esta característica hace posible dividir grandes proyectos en piezas más pequeñas independientes. Cuando diferentes programadores están implicados, ellos pueden trabajar independientemente.

A pesar del hecho importante de los subprogramas independientes y las variables locales, la mayoría de los lenguajes proporcionan algún método para tratar ambos tipos de variables. (Véase Figura 6.4).

Una variable local a un subprograma no tiene ningún significado en otros subprogramas. Si un subprograma asigna un valor a una de sus variables locales, este valor no es accesible a otros programas, es decir, no pueden utilizar este valor. A veces, también es necesario que una variable tenga el mismo nombre en diferentes subprogramas.

Por el contrario, las variables globales tienen la ventaja de compartir información de diferentes subprogramas sin una correspondiente entrada en la lista de parámetros.

En un programa sencillo con un subprograma, cada variable u otro identificador es o bien local al procedimiento o global al programa completo. Sin embargo, si el programa incluye procedimientos que engloban a otros procedimientos —*procedimientos anidados*—, entonces la noción de global/local es algo más complicado de entender.

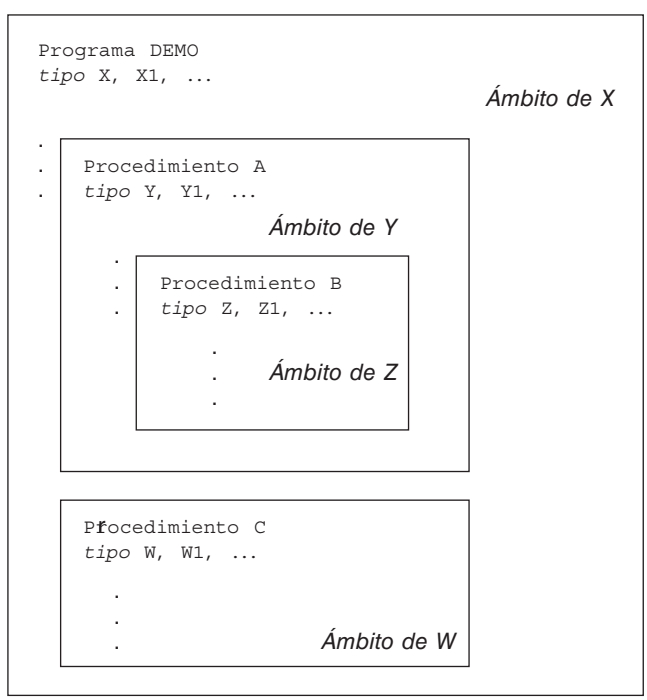
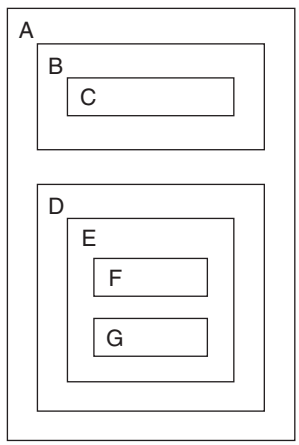



**Figura 6.4.** Ambito de identificadores.

El *ámbito* de un identificador (variables, constantes, procedimientos) es la parte del programa donde se conoce el identificador. Si un procedimiento está definido localmente a otro procedimiento, tendrá significado sólo dentro del ámbito de ese procedimiento. A las variables les sucede lo mismo; si están definidas localmente dentro de un procedimiento, su significado o uso se confina a cualquier función o procedimiento que pertenezca a esa definición.

La Figura 6.5 muestra un esquema de un programa con diferentes procedimientos, algunas variables son locales y otras globales. En la citada figura se muestra el ámbito de cada definición.




| Variables definidas en | Accesibles desde |
|---|---|
| A | A, B, C, D, E, F, G |
| B | B, C |
| C | C |
| D | D, E, F, G |
| E | E, F, G |
| F | F |
| G | G |

**Figura 6.5.** Ámbito de definición de variables.

Los lenguajes que admiten variables locales y globales suelen tener la posibilidad explícita de definir dichas variables como tales en el cuerpo del programa, o, lo que es lo mismo, definir su ámbito de actuación, para ello se utilizan las cabeceras de programas y subprogramas, con lo que se definen los ámbitos.

Las variables definidas en un ámbito son accesibles en el mismo, es decir, en todos los procedimientos interiores.

---

### EJEMPLO 6.9

*La función (signo) realiza la siguiente tarea: dado un número real x, si x es 0, entonces se devuelve un 0; si x es positivo, se devuelve 1, y si x es negativo, se devuelve un valor –1.*

La declaración de la función es

```
entero función signo(E real: x)
var entero:s
inicio
  //valores de signo: +1,0,-1
  si x = 0 entonces s ← 0
  si x > 0 entonces s ← 1
  si x < 0 entonces s ← -1
  devolver (s)
fin_función
```

Antes de llamar a la función, la variable `(S)`, como se declara dentro del subprograma, es local al subprograma y sólo se conoce dentro del mismo. Veamos ahora un pequeño algoritmo donde se invoque la función.

```
algoritmo SIGNOS
var
  entero: a, b, c
  real: x, y, z
```

```
inicio
  x ← 5.4
  a ← signo(x)
  y ← 0
  b ← signo(y)
  z ← 7.8975
  c ← signo(z - 9)
  escribir('Las respuestas son', a, ' ', b, ' ', c)
fin
```

Si se ejecuta este algoritmo, se obtienen los siguientes valores:

```
x = 5.4                  x es el parámetro actual de la primera llamada a signo(x)
a = signo(5.4)           a toma el valor 1
y = 0
b = signo(0)             b toma el valor de 0
z = 7.8975
c = signo(7.8975-9)      c toma el valor –1
```

La línea escrita al final será:

```
Las respuestas son 1 0 -1
```

### EJEMPLO 6.10

```
algoritmo DEMOX
var entero: A, X, Y
inicio
  x ← 5
  A ← 10
  y ← F(x)
  escribir (x, A, y)
fin

entero función F(E entero: N)
var
  entero: X
inicio
  A ← 5
  X ← 12
  devolver(N + A)
fin_función
```

A la variable global `A` se puede acceder desde el algoritmo y desde la función. Sin embargo, `X` identifica a dos variables distintas: una local al algoritmo y sólo se puede acceder desde él y otra local a la función.

Al ejecutar el algoritmo se obtendrían los siguientes resultados:

```
X = 5
A = 10
Y = F(5)                 invocación a la función F(N) se realiza un paso del parámetro actual X al parámetro
                         formal N
A = 5                    se modifica el valor de A en el algoritmo principal por ser A global
X = 12                   no se modifica el valor de X en el algoritmo principal porque X es local
F = 5+5 = 10             se pasa el valor del argumento X(5) a través del parámetro N
Y = 10
```

se escribirá la línea

```
5 5 10
```

ya que X es el valor de la variable local X en el algoritmo; A, el valor de A en la función, ya que se pasa este valor al algoritmo; Y es el valor de la función F(X).

## 6.5. COMUNICACIÓN CON SUBPROGRAMAS: PASO DE PARÁMETROS

Cuando un programa llama a un subprograma, la información se comunica a través de la lista de parámetros y se establece una correspondencia automática entre los parámetros formales y actuales. *Los parámetros actuales son "sustituidos"* o *"utilizados" en lugar de los parámetros formales*.

La declaración del subprograma se hace con

```
procedimiento nombre (clase tipo_de_dato: F1;
                      clase tipo_de_dato: F2;
                      ......................
                      clase tipo_de_dato :Fn)
  .
  .
  .
fin_procedimiento
```

y la llamada al subprograma con

```
llamar_a nombre (A1, A2, ..., An)
```

donde F1, F2, ..., Fn son los parámetros formales y A1, A2, ..., An los parámetros actuales o reales.

Las clases de parámetros podrían ser:

(**E**) Entrada
(**S**) Salida
(**E/S**) Entrada/Salida

Existen dos métodos para establecer la correspondencia de parámetros:

1. *Correspondencia posicional*. La correspondencia se establece aparejando los parámetros reales y formales según su posición en las listas: así, Fi se corresponde con $A_i$, donde i = 1, 2, ..., n. Este método tiene algunas desventajas de legibilidad cuando el número de parámetros es grande.
2. *Correspondencia por el nombre explícito*, también llamado *método de paso de parámetros por nombre*. En este método, en las llamadas se indica explícitamente la correspondencia entre los parámetros reales y formales. Este método se utiliza en **Ada**. Un ejemplo sería:

   ```
   SUB(Y => B, X => 30);
   ```

   que hace corresponder el parámetro actual B con el formal Y, y el parámetro actual 30 con el formal X durante la llamada de SUB.

Por lo general, la mayoría de los lenguajes usan exclusivamente la correspondencia posicional y ese será el método empleado en este libro.

Las cantidades de información que pueden pasarse como parámetros son *datos de tipos simples, estructurados* —en los lenguajes que admiten su declaración— y *subprogramas*.

### 6.5.1. Paso de parámetros

Existen diferentes métodos para la *transmisión o el paso de parámetros* a subprogramas. Es preciso conocer el método adoptado por cada lenguaje, ya que la elección puede afectar a la semántica del lenguaje. Dicho de otro modo, un mismo programa puede producir diferentes resultados bajo diferentes sistemas de paso de parámetros.

Los parámetros pueden ser clasificados como:

| | |
|---|---|
| *entradas:* | las entradas proporcionan valores desde el programa que llama y que se utilizan dentro de un procedimiento. En los subprogramas función, las entradas son los argumentos en el sentido tradicional; |
| *salidas:* | las salidas producen los resultados del subprograma; de nuevo si se utiliza el caso de una función, éste devuelve un valor calculado por dicha función, mientras que con procedimientos pueden calcularse cero, una o varias salidas; |
| *entradas/salidas:* | un solo parámetro se utiliza para mandar argumentos a un programa y para devolver resultados. |

Desgraciadamente, el conocimiento del tipo de parámetros no es suficiente para caracterizar su funcionamiento; por ello, examinaremos los diferentes métodos que se utilizan para pasar o transmitir parámetros.

Los métodos más empleados para realizar el paso de parámetros son:

- *paso por valor* (también conocido por *parámetro valor*),
- *paso por referencia o dirección* (también conocido por *parámetro variable*),
- *paso por nombre,*
- *paso por resultado*.

### 6.5.2. Paso por valor

El paso por valor se utiliza en muchos lenguajes de programación; por ejemplo, C, Modula-2, Pascal, Algol y Snobol. La razón de su popularidad es la analogía con los argumentos de una función, donde los valores se proporcionan en el orden de cálculo de resultados. Los parámetros se tratan como variables locales y los valores iniciales se proporcionan copiando los valores de los correspondientes argumentos.

Los parámetros formales —locales a la función— reciben como valores iniciales los valores de los parámetros actuales y con ello se ejecutan las acciones descritas en el subprograma.

No se hace diferencia entre un argumento que es variable, constante o expresión, ya que sólo importa el valor del argumento. La Figura 6.6 muestra el mecanismo de paso por valor de un procedimiento con tres parámetros.

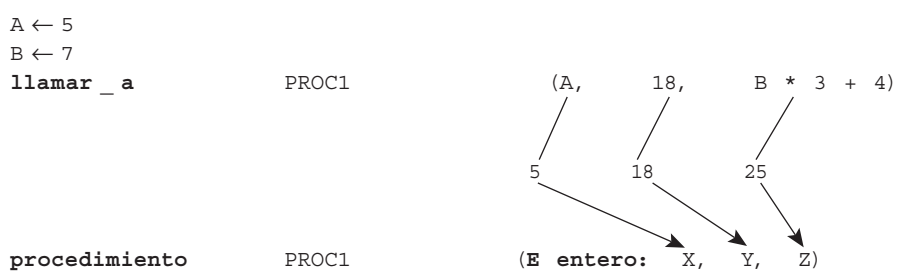



**Figura 6.6.** Paso por valor.

El mecanismo de paso se resume así:

Valor primer parámetro: `A = 5`.
Valor segundo parámetro: `constante` = `18`.
Valor tercer parámetro: expresión `B * 3 + 4 = 25`.

Los valores `5`, `18` y `25` se transforman en los parámetros `X`, `Y`, `Z` respectivamente cuando se ejecuta el procedimiento.

Aunque el *paso por valor* es sencillo, tiene una limitación acusada: *no existe ninguna otra conexión con los parámetros actuales*, y entonces los cambios que se produzcan por efecto del subprograma no producen cambios en los argumentos originales y, por consiguiente, no se pueden pasar valores de retorno al punto de llamada: es decir, todos los *parámetros* son sólo de *entrada*. El parámetro actual no puede modificarse por el subprograma. Cualquier cambio realizado en los valores de los parámetros formales durante la ejecución del subprograma se destruye cuando se termina el subprograma.

*La llamada por valor no devuelve información al programa que llama*.

Existe una variante de la llamada por valor y es la llamada por *valor resultado*. Las variables indicadas por los parámetros formales se inicializan en la llamada al subprograma por valor tras la ejecución del subprograma; los resultados (valores de los parámetros formales) se transfieren a los actuales. Este método se utiliza en algunas versiones de FORTRAN.

### 6.5.3. Paso por referencia

En numerosas ocasiones se requiere que ciertos parámetros sirvan como parámetros de salida, es decir, se devuelvan los resultados a la unidad o programas que llama. Este método se denomina *paso por referencia* o también de *llamada por dirección o variable*. La unidad que llama pasa a la unidad llamada la dirección del parámetro actual (que está en el ámbito de la unidad llamante). Una referencia al correspondiente parámetro formal se trata como una referencia a la posición de memoria, cuya dirección se ha pasado. Entonces una variable pasada como parámetro real es compartida, es decir, se puede modificar directamente por el subprograma.

Este método existe en FORTRAN, COBOL, Modula-2, Pascal, PL/1 y Algol 68. La característica de este método se debe a su simplicidad y su analogía directa con la idea de que las variables tienen una posición de memoria asignada desde la cual se pueden obtener o actualizar sus valores.

El área de almacenamiento (direcciones de memoria) se utiliza para pasar información de entrada y/o salida; en ambas direcciones.

En este método los parámetros son de entrada/salida y los parámetros se denominan *parámetros variables*.

Los parámetros valor y parámetros variable se suelen definir en la cabecera del subprograma. En el caso de lenguajes como Pascal, los parámetros variables deben ir precedidos por la palabra clave **var**;

```
program muestra;
//parametros actuales a, c, b y d paso por referencia
  procedure prueba(var x,y:integer);
  begin //procedimiento
    //proceso de los valores de x e y
  end;

begin
   .
   .
   .
1. prueba(a, c);
   .
   .
   .
2. prueba(b, d);
   .
   .
   .
end.
```

La primera llamada en (1) produce que los parámetros *a* y *c* sean sustituidos por *x* e *y* si los valores de *x* e *y* se modifican dentro de *a* o *c* en el algoritmo principal. De igual modo, *b* y *d* son sustituidos por *x* e *y*, y cualquier modificación de *x* o *y* en el procedimiento afectará también al programa principal.

*La llamada por referencia* es muy útil para programas donde se necesita la comunicación del valor en ambas direcciones.

### Notas

Ambos métodos de paso de parámetros se aplican tanto a la llamada de funciones como a las de procedimientos:

- Una función tiene la posibilidad de devolver los valores al programa principal de dos formas: *a*) como valor de la función, *b*) por medio de argumentos gobernados por la llamada de referencia en la correspondencia parámetro actual-parámetro formal.
- Un procedimiento sólo puede devolver valores por el método de devolución de resultados.

El lenguaje Pascal permite que el programador especifique el tipo de paso de parámetros y, en un mismo subprograma, unos parámetros se pueden especificar por valor y otros por referencia.

```
procedure Q(i:integer; var j:integer);
begin
  i := i+10;
  j := j+10;
  write(i, j)
end;
```

Los parámetros formales son `i`, `j`, donde `i` se pasa por valor y `j` por referencia.

## 6.5.4. Comparaciones de los métodos de paso de parámetros

Para examinar de modo práctico los diferentes métodos, consideremos un ejemplo único y veamos los diferentes valores que toman los parámetros. El algoritmo correspondiente con un procedimiento `SUBR`:

```
algoritmo DEMO
var
  entero: A,B,C
inicio   //DEMO
  A ← 3
  B ← 5
  C ← 17
  llamar_a SUBR(A, A, A + B, C)
  escribir(C)
fin      //DEMO

procedimiento SUBR (<Modo> entero: x, y;
                    E entero:z; <Modo> entero: v)
inicio
  x ← x+1
  v ← y+z
fin_procedimiento
```

#### Modo por valor

*a*) sólo por valor

*no se transmite ningún resultado, por consiguiente*
`C no varía C = 17`

*b*) valor_resultado

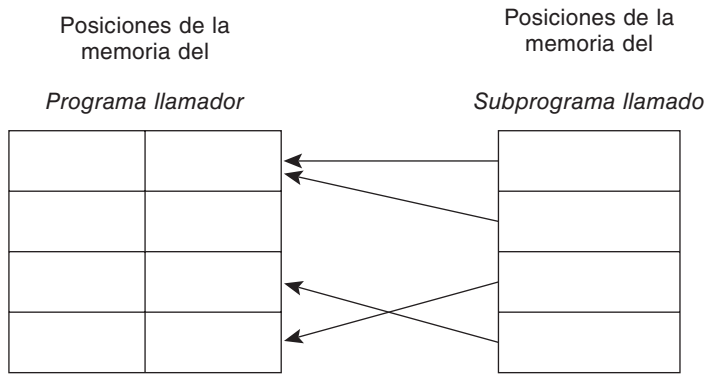
```
                                    x = A = 3
A = 3                               y = A = 3
B = 5        pasa al procedimiento  z = A + B = 8
C = 17                              v = C = 17
```

al ejecutar el procedimiento quedará

```
x = x + 1 = 3 + 1 = 4
v = y + z = 3 + 8 = 11
```

el parámetro llamado *v* pasa el valor del resultado *v* a su parámetro actual correspondiente, `C`.
Por tanto, `C = 11`.

#### *Modo por referencia*




`C` recibirá el valor `12`.

#### *Utilizando variables globales*

```
algoritmo DEMO
var entero: A,B,C
inicio
  A ← 3
  B ← 5
  c ← 17
  llamar_a SUBR
  escribir (c)
fin

procedimiento SUBR
inicio
  a ← a + 1
  c ← a + a + b
fin_procedimiento
```

Es decir, el valor de `C` será `13`.

La llamada por referencia es el sistema estándar utilizado por FORTRAN para pasar parámetros. La llamada `por nombre` es estándar en Algol 60. Simula 67 proporciona llamadas por valor, referencia y `nombre`.

Pascal permite pasar bien por valor bien por referencia

```
procedure demo(y:integer; var z:real);
```

especifica que *y* se pasa por valor mientras que *z* se pasa por referencia —indicado por la palabra reservada **`var`**—. La elección entre un sistema u otro puede venir determinado por diversas consideraciones, como evitar efectos laterales no deseados provocados por modificaciones inadvertidas de parámetros formales (*véase* Apartado 6.7).

### 6.5.5. Síntesis de la transmisión de parámetros

Los métodos de transmisión de parámetros más utilizados son *por valor* y *por referencia.*

El paso de un parámetro por valor significa que el valor del argumento —*parámetro actual o real*— se asigna al parámetro formal. En otras palabras, antes de que el subprograma comience a ejecutarse, el argumento se evalúa a un valor específico (por ejemplo, 8 o 12). Este valor se copia entonces en el correspondiente parámetro formal dentro del subprograma.

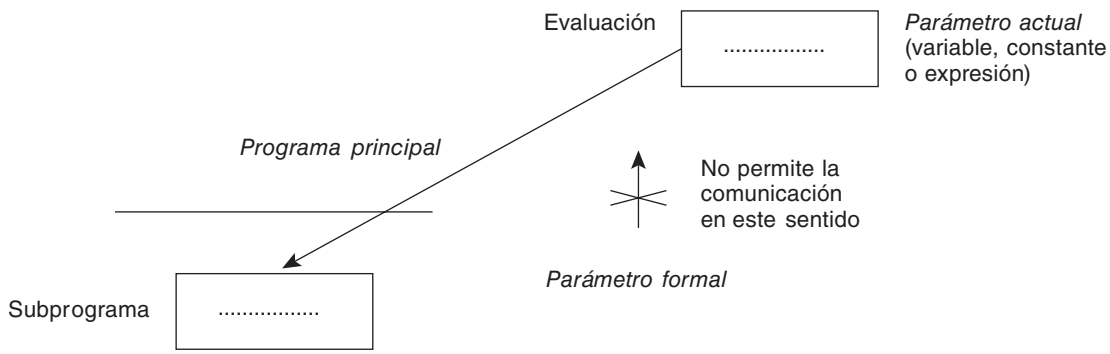



**Figura 6.7.** Paso de un parámetro por valor.

Una vez que el procedimiento arranca, cualquier cambio del valor de tal parámetro formal no se refleja en un cambio en el correspondiente argumento. Esto es, cuando el subprograma se termine, el argumento actual tendrá exactamente el mismo valor que cuando el subprograma comenzó, con independencia de lo que haya sucedido al parámetro formal. Este método es el método por defecto en Pascal si no se indica explícitamente otro. Estos parámetros de entrada se denominan *parámetros valor*. En los algoritmos indicaremos como < modo > E (entrada). El paso de un *parámetro por referencia o dirección* se llama *parámetro variable*, en oposición al parámetro por valor. En este caso, la posición o dirección (no el valor) del argumento o parámetro actual se envía al subprograma. Si a un parámetro formal se le da el atributo de parámetro variable —en Pascal con la palabra reservada **`var`**— y si el parámetro actual es una variable, entonces un cambio en el parámetro formal se refleja en un cambio en el correspondiente parámetro actual, ya que ambos tienen la misma posición de memoria.

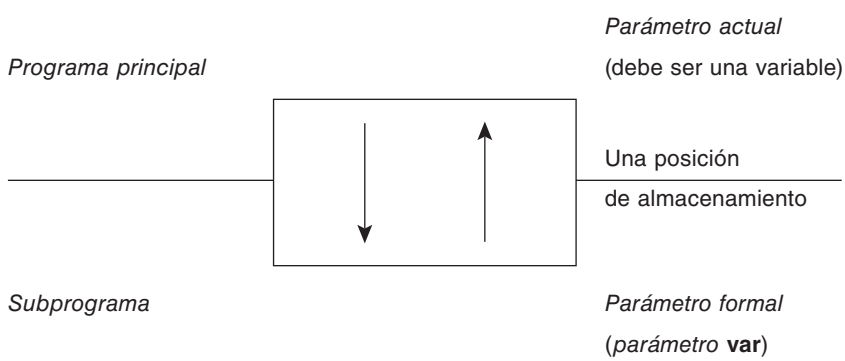



**Figura 6.8.** Paso de un parámetro por referencia.

Para indicar que deseamos transmitir un parámetro por dirección, lo indicaremos con la palabra *parámetro variable* —en Pascal se indica con la palabra reservada **`var`**— y especificaremos como <*modo*> **`E/S`** *(entrada/salida) o* **`S`** *(salida).*

---

**EJEMPLO 6.11**

*Se trata de realizar el cálculo del área de un círculo y la longitud de la circunferencia en función del valor del radio leído desde el teclado.*

Recordemos las fórmulas del área del círculo y de la longitud de la circunferencia:

$A = pi.r^2 = pi.r.r$
$C = 2.pi.r = 2.pi.r$ *donde* pi = 3.141592

Los parámetros de entrada: radio
Los parámetros de salida: área, longitud
El procedimiento *círculo* calcula los valores pedidos.

```
procedimiento circulo(E real: radio; S real: area, longitud)
  //parametros valor: radio
  //parametros variable: area, longitud
var
  real: pi
inicio
  pi ← 3.141592
  area ← pi * radio * radio
  longitud ← 2 * pi * radio
fin_procedimiento
```

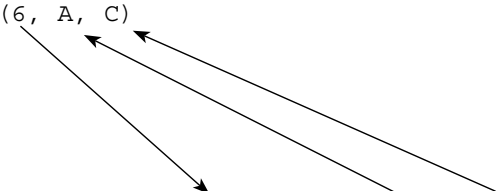
Los parámetros formales son: radio, area, longitud, de los cuales son de tipo valor (radio) y de tipo variable (area, longitud).
Invoquemos el procedimiento círculo utilizando la instrucción

```
llamar_a circulo(6, A, C)

  //{programa principal
inicio
  //llamada al procedimiento
  llamar_a circulo(6, A, C)
    .
    .
    .
fin

procedimiento circulo(E real: radio; S real: area, longitud)
//parametros valor:radio
//parametros variable: area, longitud

inicio
  pi ← 3.141592
  area ← pi * radio * radio
  longitud ← 2 * pi * radio
fin_procedimiento
```



---

### EJEMPLO 6.12

*Consideremos un subprograma* M *con dos parámetros formales:* i*, transmitido por valor, y* j*, por variable.*

```
algoritmo M
//variables A, B enteras
var
  entero: A, B
```

```
inicio
  A ← 2
  B ← 3
  llamar_a N(A,B)
  escribir(A, B)
fin    //algoritmo M

procedimiento N(E entero: i; E/S entero: j)
  //parametros valor i
  //parametros variable j
inicio
  i ← i + 10
  j ← j + 10
  escribir(i, j)
fin_procedimiento
```

Si se ejecuta el procedimiento `N`, veamos qué resultados se escribirán:

`A` y `B` son parámetros actuales.
`i` y `j` son parámetros formales.

Como *i* es por valor, se transmite el valor de `A` a *i*, es decir, *i* = `A = 2`. Cuando *i* se modifica por efecto de *i* ← *i*+`10` a `12`, `A` no cambia y, por consiguiente, a la terminación de `N`,`A` sigue valiendo `2`.

El parámetro `B` se transmite por referencia, es decir, *j* es un parámetro variable. Al comenzar la ejecución de `N`, `B` se almacena como el valor *j* y cuando se suma `10` al valor de *j*, *i* en sí mismo no cambia. El valor del parámetro `B` se cambia a `13`. Cuando los valores *i*, *j* se escriben en `N`, los resultados son:

```
12 y 13
```

pero cuando retornan a `M` y al imprimir los valores de `A` y `B`, sólo ha cambiado el valor `B`. El valor de *i* = `12` se pierde en `N` cuando éste ya termina. El valor de *j* también se pierde, pero éste es la dirección, no el valor `13`.

Se escribirá como resultado final de la instrucción ***escribir***`(A, B)`:

```
2 13
```

## 6.6. FUNCIONES Y PROCEDIMIENTOS COMO PARÁMETROS

Hasta ahora los subprogramas que hemos considerado implicaban dos tipos de parámetros formales: *parámetros valor* y *parámetros variable*. Sin embargo, en ocasiones se requiere que un procedimiento o función dado invoque a otro procedimiento o función que ha sido definido fuera del ámbito de ese procedimiento o función. Por ejemplo, se puede necesitar que un procedimiento `P` invoque la función `F` que puede estar o no definida en el procedimiento `P`; esto puede conseguirse transfiriendo como parámetro el procedimiento o función externa `(F)` o procedimiento o función dado (por ejemplo, el `P`). En resumen, algunos lenguajes de programación —entre ellos Pascal— admiten *parámetros procedimiento* y *parámetros función*.

### EJEMPLOS

```
procedimiento P(E func: F1; E real: x, y)
real función F(E func: F1, F2; E entero:x, y)
```

Los parámetros formales del procedimiento `P` son la función `F1` y las variables `x` e `y`, y los parámetros formales de la función `F` son las funciones `F1` y `F2`, y las variables `x` e `y`.

### Procedimientos función

Para ilustrar el uso de los parámetros función, consideremos la función *integral* para calcular el área bajo una curva *f*(*x*) para un intervalo *a* <= *x* <= *b*.

La técnica conocida para el cálculo del área es subdividir la región en rectángulos, como se muestra en la Figura 6.9, y sumar las áreas de los rectángulos. Estos rectángulos se construyen subdividiendo el intervalo [a, b] en *m* subintervalos iguales y formando rectángulos con estos subintervalos como bases y alturas dadas por los valores de *f* en los puntos medios de los subintervalos.

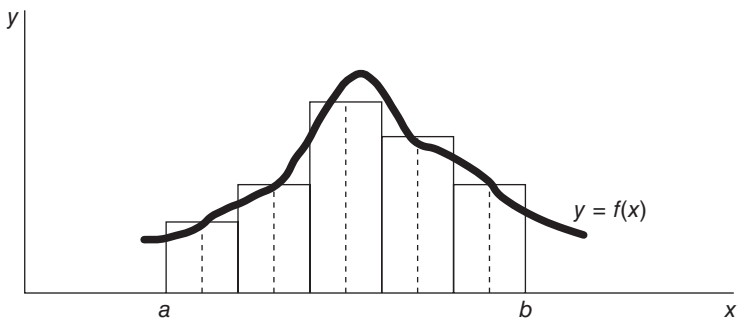



**Figura 6.9.** Cálculo del área bajo la curva *f*(*x*).

La función *integral* debe tener los parámetros formales *a, b* y *n*, que son parámetros valor ordinarios actuales de tipo real; se asocian con los parámetros formales *a* y *b*; un parámetro actual de tipo entero —las subdivisiones— se asocia con el parámetro formal *n* y una función actual se asocia con el parámetro formal *f*.

Los parámetros función se designan como tales con una cabecera de función dentro de la lista de parámetros formales. La función integral podrá definirse por

```
real función integral(E func: f; E real: a,b; E entero: n)
el tipo func-tipo real func función (E real: x)
```

aquí la función `f(x: real): real` especifica que *f* es una función parámetro que denota una función cuyo parámetro formal y valor son de tipo real. El correspondiente parámetro función actual debe ser una función que tiene un parámetro formal real y un valor real. Por ejemplo, si `Integrando` es una función de valor real con un parámetro y tipo real `función (E real:x):func`

```
Area ← Integral(Integrando, 0, 1.5, 20)
```

es una referencia válida a función.

*Diseñar un algoritmo que utilice la función Integral para calcular el área bajo el gráfico de las funciones f1 (x) =* $x^3 - 6x^3 + 10x$ *y f2(x) =* $x^2+3x+2$ *para 0 <= x <= 4.*

```
algoritmo Area_bajo_curvas

tipo
  real función(E real : x) : func
var
  real   : a,b
  entero : n

inicio
   escribir('¿Entre qué límites?')
   leer(a, b)
   escribir('¿Subintervalos?')
   leer(n)
```

```
   escribir(integral(f1, a, b, n))
   escribir(integral(f2, a, b, n))
fin

real FUNCIÓN f1 (E real : x)
inicio
  devolver( x * x * x - 6 * x * x + 10 * x)
fin_funcion

real FUNCIÓN f2 (E real : x)
inicio
  devolver( x * x + 3 * x + 2 )
fin_función

real FUNCIÓN integral (E func : f; E real : a, b; E entero : n)
var
  real : baserectangulo,altura,x,s
  entero : i

inicio
  baserectangulo ← (b - a) / n
  x ← a + baserectangulo/2
  s ← 0
  desde i ← 1 hasta n hacer
    altura ← f(x)
    s ← s + baserectangulo * altura
    x ← x + baserectangulo
  fin_desde
  devolver(s)
fin_función
```

## 6.7. LOS EFECTOS LATERALES

Las modificaciones que se produzcan mediante una función o procedimiento en los elementos situados fuera del subprograma (función o procedimiento) se denominan *efectos laterales*. Aunque en algunos casos los efectos laterales pueden ser beneficiosos en la programación, es conveniente no recurrir a ellos de modo general. Consideramos a continuación los efectos laterales en funciones y en procedimientos.

### 6.7.1. En procedimientos

La *comunicación* del procedimiento con el resto del programa se debe realizar normalmente a través de parámetros. Cualquier otra *comunicación* entre el procedimiento y el resto del programa se conoce como *efectos laterales*. Como ya se ha comentado, los efectos laterales son perjudiciales en la mayoría de los casos, como se indica en la Figura 6.10.

Si un procedimiento modifica una variable global (distinta de un parámetro actual), éste es un *efecto lateral*. Por ello, excepto en contadas ocasiones, no debe aparecer en la declaración del procedimiento. Si se necesita una variable temporal en un procedimiento, utilice una variable local, no una variable global. Si se desea que el programa modifique el valor de una variable global, utilice un parámetro formal variable en la declaración del procedimiento y a continuación utilice la variable global como el parámetro actual en una llamada al procedimiento.

En general, se debe seguir la regla de "*ninguna variable global en procedimientos*", aunque esta prohibición no significa que los procedimientos no puedan manipular variables globales. De hecho, el cambio de variables globales se deben pasar al procedimiento como parámetros actuales. Las variables globales no se deben utilizar directamente en las instrucciones en el cuerpo de un procedimiento; en su lugar, utilice un parámetro formal o variable local.

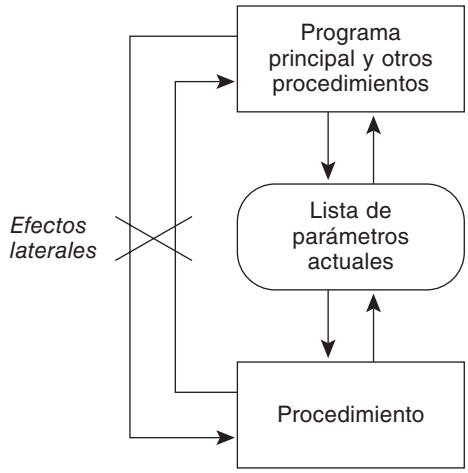



**Figura 6.10.** Efectos laterales en procedimientos.

En aquellos lenguajes en que es posible declarar constantes —como Pascal— se pueden utilizar constantes globales en una declaración de procedimiento; la razón reside en el hecho de que las constantes no pueden ser modificadas por el procedimiento y, por consiguiente, no existe peligro de que se puedan modificar inadvertidamente.

### 6.7.2. En funciones

Una función toma los valores de los argumentos y devuelve *un único valor*. Sin embargo, al igual que los procedimientos, una función —en algunos lenguajes de programación— puede hacer cosas similares a un procedimiento o subrutina. Una función puede tener parámetros variables además de parámetros valor en la lista de parámetros formales. Una función puede cambiar el contenido de una variable global y ejecutar instrucciones de entrada/salida (escribir un mensaje en la pantalla, leer un valor del teclado, etc.). Estas operaciones se conocen como *parámetros laterales* y se deben evitar.

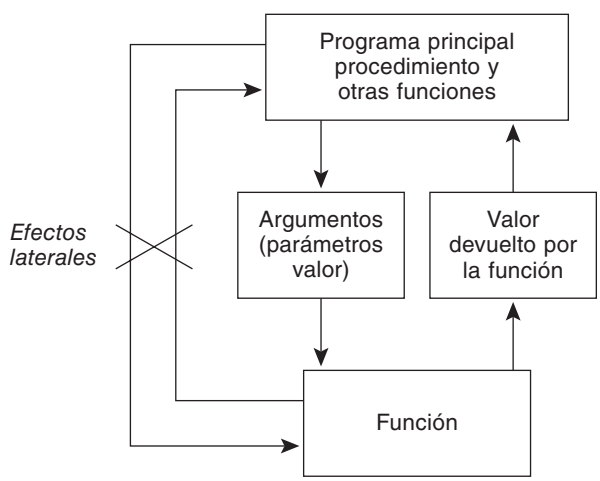



**Figura 6.11.** Efectos laterales en una función.

Los efectos laterales están considerados —normalmente— como una mala técnica de programación, pues hacen más difícil de entender los programas.

Toda la información que se transfiere entre procedimientos y funciones debe realizarse a través de la lista de parámetros y no a través de variables globales. Esto convertirá al procedimiento o función en módulos independientes que pueden ser comprobados y depurados por sí solos, lo que evitará preocuparnos por el resto de las partes del programa.

## 6.8. RECURSIÓN (RECURSIVIDAD)

Como ya se conoce, un subprograma puede llamar a cualquier otro subprograma y éste a otro, y así sucesivamente; dicho de otro modo, los subprogramas se pueden anidar. Se puede tener

```
A llamar_a B, B llamar_a C, C llamar_a D
```

Cuando se produce el retorno de los subprogramas a la terminación de cada uno de ellos el proceso resultante será

```
D retornar_a C, C retornar_a B, B retornar_a A
```

¿Qué sucedería si dos subprogramas de una secuencia son los mismos?

```
A llamar_a A
```

o bien

```
A llamar_a B, B llamar_a A
```

En primera instancia, parece incorrecta. Sin embargo, existen lenguajes de programación —Pascal, C, entre otros— en que un subprograma puede llamarse a sí mismo.

Una función o procedimiento que se puede llamar a sí mismo se llama *recursivo*. La *recursión* (**recursividad**) es una herramienta muy potente en algunas aplicaciones, sobre todo de cálculo. La recursión puede ser utilizada como una alternativa a la repetición o estructura repetitiva. El uso de la recursión es particularmente idóneo para la solución de aquellos problemas que pueden definirse de modo natural en términos recursivos.

La escritura de un procedimiento o función recursiva es similar a sus homónimos no recursivos; sin embargo, para evitar que la recursión continúe indefinidamente es preciso incluir una condición de terminación.

La razón de que existan lenguajes que admiten la recursividad se debe a la existencia de estructuras específicas tipo *pilas* (*stack*, en inglés) para este tipo de procesos y memorias dinámicas. Las direcciones de retorno y el estado de cada subprograma se guardan en estructuras tipo pilas (véase Capítulo 11). En el Capítulo 11 se profundizará en el tema de las pilas; ahora nos centraremos sólo en el concepto de recursividad y en su comprensión con ejemplos básicos.

---

### EJEMPLO 6.13

*Muchas funciones matemáticas se definen recursivamente. Un ejemplo de ello es el factorial de un número entero n.*

La función factorial se define como

```
      ⎧ 1 si   n = 0    0! = 1
n! =  ⎨
      ⎩ nx(n-1)x(n-2)x ... x3x2x1  si n > 0 n. (n-1) . (n-2)....3.2.1
```

Si se observa la fórmula anterior cuando `n > 0`, es fácil definir `n!` en función de `(n-1)!` Por ejemplo, `5!`

```
5! = 5x4x3x2x1          = 120
4! = 4x3x2x1            = 24
3! = 3x2x1              = 6
2! = 2x1                = 2
1! = 1x1                = 1
0! = 1                  = 1
```

Se pueden transformar las expresiones anteriores en

```
5! = 5x4!
4! = 4x3!
3! = 3x2!
2! = 2x1!
1! = 1x0!
```

En términos generales sería:

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n(n-1)! & \text{si } n > 0 \end{cases}$$

La función `FACTORIAL` de `N` expresada en términos recursivos sería:

```
FACTORIAL ← N * FACTORIAL(N - 1)
```

La definición de la función sería:

```
entero: función factorial(E entero: n)
//calculo recursivo del factorial
inicio
  si n = 0 entonces
    devolver (1)
  si_no devolver (n * factorial(n - 1))
  fin_si
fin_función
```

Para demostrar cómo esta versión recursiva de `FACTORIAL` calcula el valor de *n!,* consideremos el caso de *n = 3*. Un proceso gráfico se representa en la Figura 6.12.

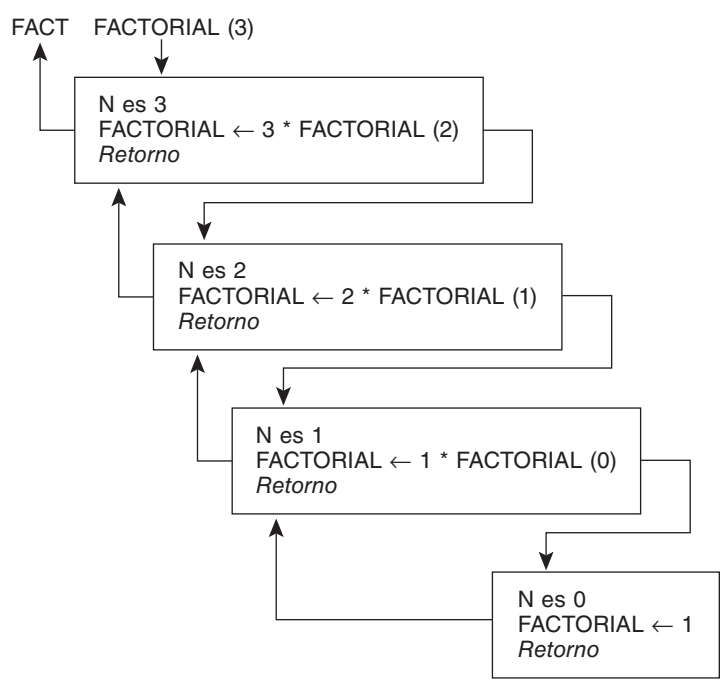



**Figura 6.12.** Cálculo recursivo de `FACTORIAL` de 3.

**EJEMPLO 6.14**

*Otro ejemplo típico de una función recursiva es la serie Fibonacci. Esta serie fue concebida originalmente como modelo para el crecimiento de una granja de conejos (multiplicación de conejos) por el matemático italiano del siglo XVI, Fibonacci.*

La serie es la siguiente:

1, 1, 2, 3, 5, 8, 13, 21, 34 ...

Esta serie crece muy rápidamente; como ejemplo, el término 15 es 610.
La serie de Fibonacci (*fib*) se expresa así

*fib(1) = 1*
*fib(2) = 1*
*fib(n) = fib(n – 1) + fib(n – 2)* para *n > 2*

Una función recursiva que calcula el elemento enésimo de la serie de Fibonacci es

```
entero: función fibonacci(E entero: n)
//calculo del elemento n-ésimo
inicio
  si (n = 1) o (n = 2) entonces
    devolver (1)
  si_no
    devolver (fibonacci(n - 2) + fibonacci(n - 1))
  fin_si
fin_función
```

Aunque es fácil de escribir la función de Fibonacci, no es muy eficaz definida de esta forma, ya que cada paso recursivo genera otras dos llamadas a la misma función.

## 6.9. FUNCIONES EN C/C++ , JAVA Y C#

La sintaxis básica y estructura de una función en C, C++, Java y C# son realmente idénticas

```
valor_retorno nombre_funcion (tipo1 arg1, tipo2 arg2 …)
{  // equivalente a la palabra reservada en pseudocódigo inicio
   // cuerpo de la función
}  // equivalente a la palabra reservada en pseudocódigo fin
```

En Java, todas las funciones deben estar asociadas con alguna clase y se denominan **métodos**. En C++, las funciones asociadas con una clase se llaman **funciones miembro**. Las funciones C tradicionales y las funciones no asociadas con ninguna clase en C++, se denominan simplemente **funciones no miembro**.

El nombre de la función y su lista de argumentos constituyen la ***signatura***. La lista de parámetros formales es la *interfaz de la función* con el mundo exterior, dado que es el punto de entrada para parámetros entrantes.

La descripción de una función se realiza en dos partes: *declaración de la función y definición de la función*. La **declaración de una función**, denominada también *prototipo de la función,* describe cómo se llama (invoca) a la función. Existen dos métodos para declarar una función:

1. Escribir la función completa antes de ser utilizada.
2. Definir el *prototipo de la función* que proporciona al compilador información suficiente para llamar a la función. El prototipo de una función es similar a la primera línea de la función, pero el prototipo no tiene cuerpo.

*Prototipo de función*
*Definición función*
*cabecera*
*cuerpo*

```
double precio_total (int numero, double precio);
double precio_total (int numero, double precio)
{
  const double IVA = 0.06; // impuesto 6%
  double subtotal;
    subtotal = numero * precio;
  return (subtotal + IVA * subtotal):
}
```

### *Paso de parámetros*

El paso de parámetros varía ligeramente entre Java y C++. Desde el enfoque de Java:

- No existen punteros como en C y C++;
- Los tipos integrados o incorporados (*built-in*, denominados también tipos primitivos de datos) se pasan siempre por valor;
- Tipos objeto (similares a las clases que son parte de los paquetes estándar de java) se pasan siempre por referencia

Las variables de Java que representan tipos objeto se llaman *variables de referencia* y aquellas que representan tipos integrados se llaman *variables de no referencia.*

---

**EJEMPLO 6.15. FUNCIÓN EN C++**

*La función triángulo calcula el área de un triángulo en C++*

```
// Función en C++
// Triángulo, cálculo del área o superficie
// Parámetros
// anchura – anchura del triángulo
// altura - altura del triángulo
// retorno (devuelve)
// Área del triángulo
float triangulo(float anchura, float altura)
{
    float area
    assert (anchura >= 0.0);
    assert (altura >= 0.0);
    return (area)
}
...
// Llamada por valor a la función area
Superficie = triangulo (2.5, 4.6); // paso de parámetros por valor
```

La llamada por valor ejecuta la función `triangulo`, calcula la fórmula del área del triángulo y su valor 11.50 se asigna a la variable `superficie`.

---

**EJEMPLO. 6.16. FUNCIÓN EN JAVA**

```
import java.awt.Point;     // se importa la clase
                           // Point parte del paquete
                           // estándar Java AWT
// paso por valor
public class DemoPasoParametros {
```

```
    public static void intercambio_por_valor (int x, int y){
        int aux;
        aux = x;
        x = y;
        Y = aunx;
  }
}

public static void intercambio_por_referencia (Punto p){ '
    int aux = p.x;
    p.x     = p.y;
    p.y     = aux;
    // …
}
// llamadas a la función
int m = 50;
int n = 75;
// …
intercambio_porvalor (m, n);
// …
punto unPunto= new Punto (-10, 50);
intercambio_porreferencia (unPunto)
// ...
```

En las líneas de código anteriores, `m` y `n` son variables integradas de tipo **`int`**. Para intercambiar los valores de las dos variables se pasan en el método `intercambio_porvalor`. Una copia de los valores `m` y `n` se pasan a la función `intercambio_porvalor` cuando se llama a la función.

En el caso de la llamada por referencia se ha instanciado e inicializado un objeto, `unPunto` (de la clase `Point` definido en el paquete `Java.awt.Point`) a los valores 50, 75. Cuando `unPunto` se pasa en un método llamado `intecambio_porreferencia`, el método intercambia el contenido de las coordenadas *x* e *y* del argumento. Es preciso observar que una variable referencia es realmente una dirección al objeto y no el objeto en sí mismo. Al pasar `unPunto`, en realidad se pasa la dirección del objeto y no una copia —como en el paso por valor— del objeto.

Esta característica de Java es equivalente semánticamente al modo en que funcionan las variables referencia y variables puntero en C++. En resumen, las variables referencia en Java son muy similares a las referencias C++.

## 6.10. ÁMBITO (ALCANCE) Y ALMACENAMIENTO EN C/C++ Y JAVA

Cada identificador (nombre de una entidad) debe referirse a una única identidad (tal como una variable, función, tipo, etc.). A pesar de este requisito, los nombres se pueden utilizar más de una vez en un programa. Un nombre se puede reutilizar mientras se utilice en diferentes contextos, a partir de los cuales los diferentes significados del nombre pueden ser empleados. El contexto utilizado para distinguir los significados de los nombres es su *alcance* o *ámbito (scope).* Un **ámbito** o **alcance** es una región del código de programa, donde se permite hacer referencia (uso) a un identificador.

Un nombre (identificador) se puede referir a diferentes entidades en diferentes ámbitos.

Los alcances están separados por los separadores `inicio-fin` (o llaves en los lenguajes C, C++, Java, etc.). Los nombres son visibles desde su punto de declaración hasta el final del alcance en el que aparece la declaración.

Los identificadores definidos fuera de cualquier función tienen *ámbito global*; son accesibles desde cualquier parte del programa. Los identificadores definidos en el cuerpo de una función se dicen que tienen *ámbito local.*

La *clase de almacenamiento* de una variable puede ser o bien *permanente* o *temporal.* Las variables globales son siempre permanentes; se crean e inicializan antes de que el programa arranque y permanecen hasta que se termina. Las variables temporales se asignan desde una sección de memoria llamada la pila (stack) en el principio del bloque.

Si se intentan asignar muchas variables temporales se puede obtener un error de desbordamiento de la pila. El espacio utilizado por las variables temporales se devuelve (se libera) a la pila al final del bloque. Cada vez que se entra al bloque, se inicializan las variables temporales.

Las variables locales son temporales a menos que sean declaradas estáticas `static` en C++.

### Definición y declaración de variables

El ámbito (alcance) de una variable es el área (bloque) del programa donde es válida la variable. En general, las definiciones o declaraciones de variables se pueden situar en cualquier parte de un programa donde esté permitida una sentencia. Una variable debe ser declarada o definida antes de que sea utilizada.

#### Regla

Es una buena idea definir un objeto cerca del punto en el cual se va a utilizar la primera vez.

Las variables pueden ser *globales* o *locales.* Una variable global es de alcance global y es válida desde el punto en que se declara hasta el final del programa. Su duración es la del programa, hasta que se acaba su ejecución. Una variable local es aquella que está definida en el interior del cuerpo de una función y es accesible sólo dentro de dicha función. El ámbito de una variable local se limita al bloque donde está declarada y no puede ser accedida (leída o asignada un valor) fuera de ese bloque.

En el cuerpo o bloque de una función se pueden definir variables locales que son "locales" a dicha función y sus nombres sólo son visibles en el ámbito de la función. Las variables locales sólo existen mientras la función se está ejecutando.

Un bloque es una sección de código encerrada entre inicio y fin (en el caso de C/C++ o Java/C#, encerrado entre llaves, { }).

Los nombres de las variables locales a una función son visibles sólo en el ámbito de la función y existen sólo mientras la función se está ejecutando. La ejecución se termina cuando se encuentra una sentencia `devolver (return)` y produce como resultado el valor especificado en dicha sentencia. Es posible declarar una variable local con el mismo nombre que una variable global, pero en el bloque donde está definida la variable local tiene prioridad sobre la variable global y se dice que esta variable se encuentra *oculta.* ***Si*** *una variable local oculta a una global* para que tenga el mismo nombre ***entonces***, se dice, que la variable global no es posible.

Una variable global es aquella que se define fuera del cuerpo de las funciones y están disponibles en todas las partes del programa, incluso en otros archivos como en lenguajes C++ donde un programa puede estar en dos archivos. En el caso de que un programa esté compuesto por dos archivos, en el primero se define la variable global y se declara en el segundo archivo donde se puede utilizar.

**EJEMPLO 6.17. VARIABLES LOCALES Y GLOBALES EN C++**

*alcance*

```
int cuenta;          // variable global
int main ( )         // función principal
{
  int local:         // variable local
    cuenta = 100:
  local = 500;
```

*global local*
*alcance*
*local_uno*

```
  {
    int local_uno;
    local_uno = cuenta + local;
  }
  // no se puede utilizar local_uno
}
```

Si en el segmento de código siguiente se declara una nueva variable local `cuenta`, ésta se oculta a la variable global `cuenta` inicializada a 100 en el cuerpo de la función.

```
int total;
int cuenta;

int main( )
{
  total = 0;
  cuenta = 100;

  int cuenta;
  cuenta = 0;
  while (true) {
    if (cuenta > 10)
        break;
    total += cuenta;
    ++cuenta;
    }
  }
  ++cuenta;
  return (0);
}
```

## 6.11. SOBRECARGA DE FUNCIONES EN C++ Y JAVA

Algunos lenguajes de programación como C++ o Java permiten la sobrecarga de funciones (funciones miembro en C++, métodos en Java). La **sobrecarga de funciones**, que aparecen en el mismo ámbito, significa que se pueden definir múltiples funciones con el mismo nombre pero con listas de parámetros diferentes.

*Sobrecarga de una función* es usar el mismo nombre para diferentes funciones, distintas unas de otras por sus listas de parámetros.

En realidad, la sobrecarga de funciones es una propiedad que facilita la tarea al programador cuando se desean diseñar funciones que realizan la misma tarea general pero que se aplican a tipos de parámetros diferentes. Estas funciones se pueden llamar sin preocuparse sobre cuál función se invoca ya que el compilador detecta el tipo de dato de los parámetros y ejecuta la función asociada a ellos.

Por ejemplo, se trata de ejecutar una función que imprima los valores de determinadas variables de diferentes tipos de datos: `car, entero, real, lógico`, etc. Así algunas funciones que realizan estas tareas serían:

```
nada ImprimirEnteros (entero n)
inicio
  escribir ('Visualizar')
  escribirn('El valor es', n)
fin
```

```
nada ImprimirCar(car c)
inicio
  escribir ('Visualizar')
  escribirn('El valor es', c)
fin

nada ImprimirReal(real r)
inicio
  escribir ('Visualizar')
  escribirn('El valor es', r)
fin

nada ImprimirLogico (lógico l)
inicio
  escribir ('Visualizar')
  escribirn('El valor es', l)
fin
```

Se necesitan cuatro funciones diferentes con cuatro nombres diferentes; si se utilizan funciones sobrecargadas, en lugar de utilizar un nombre para cada tipo de impresión de datos, se puede utilizar una función sobrecargada con el mismo nombre `Imprimir` y con distintos parámetros.

```
nada Imprimir (entero n)
inicio
  escribir ('Visualizar')
  escribirn('El valor es', n)
fin

nada Imprimir (car c)
inicio
  escribir ('Visualizar')
  escribirn('El valor es', c)
fin

nada Imprimir (real r)
inicio
  escribir ('Visualizar')
  escribirn('El valor es', r)
fin

nada Imprimir (logico l)
inicio
  escribir ('Visualizar')
  escribirn('El valor es', l)
fin
```

El código que llama a estas funciones podría ser:

```
Imprimir (entero1)
Imprimir (car1)
Imprimir (real1)
Imprimir (logico1)
```

De este modo, el nombre de la función tiene cuatro definiciones diferentes ya que el nombre `Imprimir` está sobrecargado.

La sobrecarga es una característica muy notable ya que hace a un programa más fácil de leer. Cuando se invoca a una función sobrecargada el compilador comprueba el número y tipo de argumentos en dicha llamada.

**EJERCICIO**

*Sobrecarga de la función media (media aritmética de dos o tres números reales).*

```
real media (real n1, real n2)
inicio
     devolver ((n1 + n2)/ 2.0)
fin

real media (real n1, real n2, real n3)
inicio
    devolver ((n1 + n2 + n3)/ 3.0)
fin
```

Algunas llamadas a la función son:

```
media (4.5, 7.5)
media (3.5, 5.5, 10.5)
```

C , Pascal y FORTRAN no soportan sobrecarga de funciones. *C++ y Java soportan sobrecarga de funciones miembro y métodos.*

**EJEMPLO**

*Función `cuadrado` que eleva al cuadrado el valor del argumento.*

```
entero cuadrado (entero n)
inicio
    devolver (n*n)
fin
```

```
entero cuadrado (real n)
inicio
    devolver (n * n)
fin
```

### *Sobrecarga en C++*

Las funciones anteriores escritas en C++

```
int cuadrado (int n)
{
    return (n * n);
}
```

```
float cuadrado (float n)
{
    return (n * n);
}
```

### *Sobrecarga en Java*

En Java se pueden diseñar en una clase métodos con el mismo nombre, e incluso en la biblioteca de clases de Java, la misma situación. Dos características diferencian los métodos con igual nombre:

- El número de argumentos que aceptan.
- El tipo de dato u objetos de cada argumento.

Estas dos características constituyen la *signatura* de un método. El uso de varios métodos con el mismo nombre y signaturas diferentes se denomina *sobrecarga.* La sobrecarga de métodos puede eliminar la necesidad de escribir métodos diferentes que realizan la misma acción. La sobrecarga facilita que existan métodos que se comportan de modo diferente basado en los argumentos que reciben.

Cuando se llama a un método de un objeto, en Java, hace corresponder el nombre del método y los argumentos para seleccionar cuál es la definición a ejecutar.

Para crear un método sobrecargado, se crean diferentes definiciones de métodos en una clase, cada uno con el mismo nombre pero diferente lista de argumentos. La diferencia puede ser el número, el tipo de argumentos o ambos. Java permite la sobrecarga de métodos pero cada lista de argumentos es única para el mismo nombre del método.

## ACTIVIDADES DE PROGRAMACIÓN RESUELTAS

**6.1.** *Realización del factorial de un número entero.*

```
entero: función factorial(E entero: n)
var entero: f, i
inicio
  si n = 0 entonces
    devolver (1)
  si_no
    desde i ← 1 hasta n hacer
      f ← f*i
    fin_desde
    devolver (f)
  fin_si
fin_función
```

**6.2.** *Diseñar un algoritmo que calcule el máximo común divisor de dos números mediante el algoritmo de Euclides.*

Sean los dos números A y B. El método para hallar el máximo común divisor (mcd) de dos números A y B por el método de Euclides es:

1. Dividir el número mayor (A) por el menor (B). Si el resto de la división es cero, el número B es el máximo común divisor.
2. Si la división no es exacta, se divide el número menor (B) por el resto de la división anterior.
3. Se siguen los pasos anteriores hasta obtener un resto cero. El último divisor es el mcd buscado.

```
Algoritmo

entero función mcd(E entero: a, b)
inicio
  mientras a <> b hacer
    si a > b entonces
      a ← a - b
    si_no
      b ← b - a
    fin_si
  fin_mientras
  devolver(a)
fin_funcion
```

**6.3.** *Para calcular el máximo común divisor (mcd) de dos números se recurre a una función específica definida con un subprograma. Se desea calcular la salida del programa principal con dos números A y B, cuyos valores son 10 y 25, es decir, el mcd (A, B) y comprobar el método de paso de parámetros por valor.*

```
algoritmo maxcomdiv
var
  entero: N, X, Y
inicio //programa principal
  x ← 10
  y ← 25
  n ← mcd(x, y)
  escribir(x, y, n)
fin
entero función mcd(E entero: a,b)
inicio
  mientras a <> b hacer
    si a > b entonces
      a ← a - b
```

```
      si_no
        b ← b - a
      fin_si
    fin_mientras
    devolver (a)
  fin_función
```

Los parámetros formales son `a` y `b` y recibirán los valores de `x` e `y`.

```
a = 10
b = 25
```

Las variables locales a la función son `A` y `B` y no modificarán los valores de las variables `x` e `y` del algoritmo principal.

| *Variables del programa principal* | *Variables de la función* |
|---|---|
| `x  y  N` | `a  b  mcd(a, b)` |
| `10  25` | `10  25` |

Las operaciones del algoritmo son:

```
a = 10   b = 25
```

1. `b > a` realizará la operación `b ← b - a`
   y por consiguiente *b* tomará el valor `25-10 = 15`
   y `a` sigue valiendo `10`
2. `a = 10   b = 15`
   se realiza la misma operación anterior
   `b ← b - a`, es decir, `b = 5`
   `a` permanece inalterable
3. `a = 10   b = 5`
   como `a > b` entonces se realiza `a ← a - b`, es decir, `a = 5`

Por consiguiente, los valores finales serían:

```
a = 5   b = 5   mcd(a, b) = 5
```

Como los valores `a` y `b` no se pasan al algoritmo principal, el resultado de su ejecución será:

```
10   25   5
```

**6.4.** *Realizar un algoritmo que permita ordenar tres números mediante un procedimiento de intercambio en dos variables (paso de parámetros por referencia).*

El algoritmo que permite realizar el intercambio de los valores de variables numéricas es el siguiente:

```
AUXI ← A
A    ← B
B    ← AUXI
```

y la definición del procedimiento será:

```
PROCEDIMIENTO intercambio (E/S real: a, b)
var real : auxi

inicio
  auxi ← a
  a ← b
  b ← auxi
fin_procedimiento
```

El algoritmo de ordenación se realizará mediante llamadas al procedimiento *intercambio.*

```
algoritmo Ordenar_3_numeros
var real : x,y,z

inicio
  escribir('Deme 3 números reales')
  leer(x, y, z)
  si x > y entonces
    intercambio (x, y)
  fin_si
  si y > z entonces
    intercambio (y, z)
  fin_si
  si x > y entonces
    intercambio (x, y)
  fin_si
  escribir( x, y, z)
fin
```

***Paso de parámetros por referencia***

Los tres números `X`, `Y`, `Z` que se van a ordenar son

```
132   45   15
```

Los pasos sucesivos al ejecutarse el algoritmo o programa principal son:

1. Lectura `X`, `Y`, `Z` parámetros actuales

   ```
   X = 132
   Y = 45
   Z = 15
   ```

2. Primera llamada al procedimiento `intercambio(a, b) x > y`.
   La correspondencia entre parámetros será la siguiente:

   | *parámetros actuales* | *parámetros formales* |
   |---|---|
   | `X` | `A` |
   | `Y` | `B` |

   Al ejecutarse el procedimiento se intercambiarán los valores de `A` y `B` que se devolverán a las variables `X` e `Y`; luego valdrán

   ```
   X = 45
   Y = 132
   ```

3. Segunda llamada al procedimiento *intercambio* con `Y > Z` (ya que `Y = 132` y `Z = 15`)

   | *parámetros actuales* | | *parámetros formales* |
   |---|---|---|
   | `Y` | ⟶ | `A` |
   | `Z` | ⟶ | `B` |

   Antes llamada al procedimiento `Y = 132`, `Z = 15`.
   Después terminación del procedimiento `Z = 132`, `Y = 15`, ya que `A` y `B` han intercambiado los valores recibidos, `132` y `15`.

4. Los valores actuales de X, Y, Z son 45, 15, 132; por consiguiente, `X > Y` y habrá que hacer otra nueva llamada al procedimiento *intercambio*.

| *parámetros actuales* | | *parámetros formales* |
|---|---|---|
| `X(45)` | ⟶ | `A(45)` |
| `Y(15)` | ⟶ | `B(15)` |

Después de la ejecución del procedimiento `A` y `B` intercambiarán sus valores y valdrán `A = 15`, `B = 45`, por lo que se pasan al algoritmo principal `X = 15`, `Y = 45`. Por consiguiente, el valor final de las tres variables será:

```
X = 15 Y = 45 Z = 132
```

ya ordenados de modo creciente.

**6.5.** *Diseñar un algoritmo que llame a la función* **`signo(X)`** *y calcule: a) el signo de un número, b) el signo de la función coseno.*

Variables de entrada: `P` (real)
Variables de salida: `Y`-signo del valor `P`-(entero) `Z`-signo del coseno de `P`-(entero);

*Pseudocódigo*

```
algoritmo signos
var entero: y, z
  real: P
inicio
  leer(P)
  Y ← signo(p)
  Z ← signo(cos (p))
  escribir(Y, Z)
fin

entero función signo(E real: x)
  inicio
    si x > 0 entonces
      devolver (1)
    si_no
      si x < 0 entonces
        devolver (-1)
      si_no
        devolver (0)
      fin_si
    fin_si
fin_función
```

*Notas de ejecución*

| *Parámetro actual* | *Parámetro formal* |
|---|---|
| `P` | `x` |

El parámetro formal `x` se sustituye por el parámetro actual. Así, por ejemplo, si el parámetro `P` vale `-1.45`. Los valores devueltos por la función `Signo` que se asignará a las variables `Y`, `Z` son:

```
Y ← Signo(-1.45)
Z ← Signo(Cos (-1.45))
```

resultando

```
Y = -1
Z = 1
```

## CONCEPTOS CLAVE

- alcance.
- ámbito.
- ámbito global.
- ámbito local.
- argumento.
- argumento actual.
- argumentos formales.
- argumentos reales.
- biblioteca estándar.
- cabecera de función.
- clase de almacenamiento.
- cuerpo de la función.
- función.
- función invocada.
- función llamada.
- función llamadora.
- función recursiva.
- módulo.
- parámetro.
- parámetro actual.
- parámetros formales.
- parámetros reales.
- paso por referencia.
- paso por valor.
- procedimiento.
- prototipo de función.
- sentencia **`devolver (return)`**.
- subprograma.
- rango.
- variable global.
- variable local.

## RESUMEN

Aunque los conceptos son similares, las unidades de programas definidas por el usuario se conocen generalmente por el término de *subprograma*s para representar los módulos correspondientes; sin embargo, se denominan con nombres diferentes en los distintos lenguajes de programación. Así en los lenguajes **C** y **C++** los subprogramas se denominan *funciones*; en los lenguajes de programación orientada a objetos (**C++**, **Java** y **C#**) y siempre que se definen dentro de las clases, se les suele también denominar *métodos* o funciones miembro; en **Pascal**, son *procedimientos* y *funciones;* en **Módula-2** los nombres son PROCEDIMIENTOS (procedures, incluso aunque algunos de ellos son realmente funciones); en **COBOL** se conocen como *párrafos* y en los "viejos" **FORTRAN** y **BASIC** se les conoce como *subrutinas* y *funciones.* Los conceptos más importantes sobre funciones y procedimientos son los siguientes:

1. Las funciones y procedimientos se pueden utilizar para romper un programa en módulos de menor complejidad. De esta forma un trabajo complejo se puede descomponer en otras unidades más pequeñas que interactúan unas con otras de un modo controlado. Estos módulos tienen las siguientes propiedades:
   *a*) El propósito de cada función o procedimiento debe estar claro y ser simple.
   *b*) Una función o procedimiento debe ser lo bastante corta como para ser comprendida en toda su entidad.
   *c*) Todas sus acciones deben estar interconectadas y trabajar al mismo nivel de detalle.
   *d*) El tamaño y la complejidad de un subprograma se pueden reducir llamando a otros subprogramas para que hagan subtareas.
2. Las funciones definidas por el usuario son subrutinas que realizan una operación y devuelven un valor al entorno o módulo que le llamó. Los argumentos pasados a las funciones se manipulan por la rutina para producir un valor de retorno. Algunas funciones calculan y devuelven valores, otras funciones no. Una función que no devuelve ningún valor, se denomina función void en el caso del lenguaje C.
3. Los procedimientos no devuelven ningún valor al módulo que le invocó. En realidad, los procedimientos ya se conservan sólo en algunos lenguajes procedimentales como Pascal. En el resto de los lenguajes sólo se implementan funciones y los procedimientos son equivalentes a funciones que no devuelven valor.
4. Una llamada a una función que devuelve un valor, se encuentra normalmente en una sentencia de asignación, una expresión o una sentencia de salida.
5. Los componentes básicos de una función son la cabecera de la función y el cuerpo de la función.
6. Los argumentos son el medio por el cual un programa llamador comunica o envía los datos a una función. Los parámetros son el medio por el cual una función recibe los datos enviados o comunicados. Cuando una función se llama, los argumentos reales en la llamada a la función se pasan a dicha función y sus valores se sustituyen en los parámetros formales de la misma.
7. Después de pasar los valores de los parámetros, el control se pasa a la función. El cálculo comienza en la parte superior de la función y prosigue hasta que se termina, en cuyo momento el resultado se devuelve al programa llamador.
8. Cada variable utilizada en un programa tiene un ámbito (rango o alcance) que determina en qué parte del programa se puede utilizar. El ámbito de una variable es local o global y se determina por la posición donde se sitúa la variable. Una variable local se define dentro de una función y sólo se puede utilizar dentro de la definición de dicha función o bloque. Una variable global está definida fuera de una función y se puede utilizar en cualquier fun-

ción a continuación de la definición de la variable. Todas las variables globales que no son inicializadas por el usuario, normalmente se inicializan a cero por la computadora.

9. Una solución recursiva es una en que la solución se puede expresar en términos de una versión más simple de sí misma. Es decir, una función recursiva se puede llamar a sí misma.
10. Si una solución de un problema se puede expresar repetitivamente o recursivamente con igual facilidad, la solución repetitiva es preferible, ya que se ejecuta más rápidamente y utiliza menos memoria. Sin embargo, en muchas aplicaciones avanzadas la recursión es más simple de visualizar y el único medio práctico de implementar una solución.

## EJERCICIOS

**6.1.** Diseñar una función que calcule la media de tres números leídos del teclado y poner un ejemplo de su aplicación.

**6.2.** Diseñar la función `FACTORIAL` que calcule el factorial de un número entero en el rango 100 a 1.000.000.

**6.3.** Diseñar un algoritmo para calcular el máximo común divisor de cuatro números basado en un subalgoritmo función mcd (máximo común divisor de dos números).

**6.4.** Diseñar una función que encuentre el mayor de dos números enteros.

**6.5.** Diseñar una función que calcule $x^n$ para `x`, variable real y `n` variable entera.

**6.6.** Diseñar un procedimiento que acepte un número de mes, un número de día y un número de año y los visualice en el formato

```
dd/mm/aa
```

Por ejemplo, los valores 19,09,1987 se visualizarían como

```
19/9/87
```

y para los valores 3, 9 y 1905

```
3/9/05
```

**6.7.** Realizar un procedimiento que realice la conversión de coordenadas polares ($r$, $\theta$) a coordenadas cartesianas ($x$, $y$)

$x = r.\cos(\theta)$
$y = r.\text{sen}(\theta)$

**6.8.** Escribir una función `Salario` que calcule los salarios de un trabajador para un número dado de horas trabajadas y un salario hora. Las horas que superen las 40 horas semanales se pagarán como extras con un salario hora 1,5 veces el salario ordinario.

**6.9.** Escribir una función booleana `Digito` que determine si un carácter es uno de los dígitos 0 al 9.

**6.10.** Diseñar una función que permita devolver el valor absoluto de un número.

**6.11.** Realizar un procedimiento que obtenga la división entera y el resto de la misma utilizando únicamente los operadores suma y resta.

**6.12.** Escribir una función que permita deducir si una fecha leída del teclado es válida.

**6.13.** Diseñar un algoritmo que transforme un número introducido por teclado en notación decimal a notación romana. El número será entero positivo y no excederá de 3.000.

**6.14.** Escribir el algoritmo de una función recursiva que: *a*) calcule el factorial de un número entero positivo, *b*) la potencia de un número entero positivo.