

(Versión de fecha 30 de octubre de 2024)

## 1 CASTING

$3/2 \rightarrow 1$

$3.0/2 \rightarrow 1.5$

(double) base/altura

sqrt( )

`int suma, num_el;`

`double media = (double) suma / num_el;`

`int x = 3.14; → x = 3;`

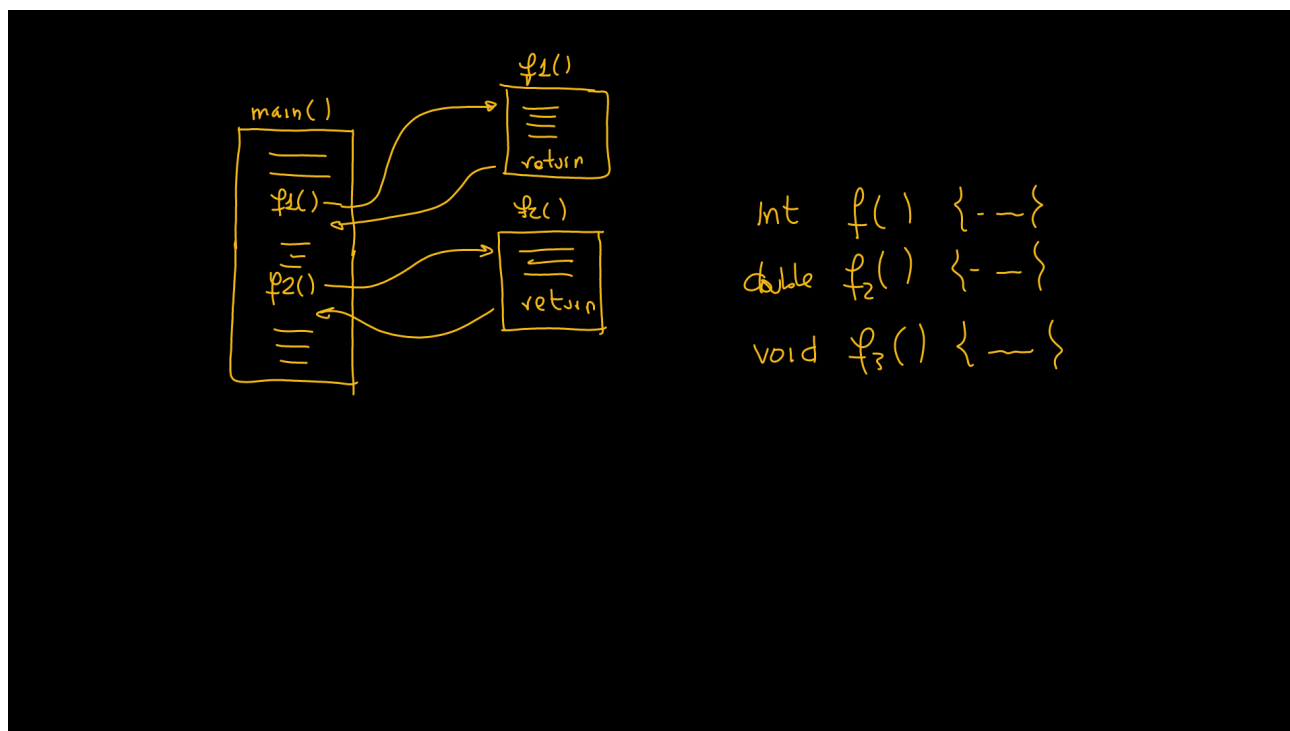
`int p = 3.99; → p = 3;`

`int p = sqrt(2); → p = 1`

`double x = 3; x = 3.00`

## 2 CONCEPTO DE FUNCIÓN

(Ver las diapositivas 3 a 9 del tema 2 del curso)



En programación, una *función* es un bloque de código que se puede ejecutar de manera independiente. Se suelen llamar también *procedimientos* o *subrutinas*.

Una función es como un miniprograma dentro de un programa. En general, las funciones reciben unos valores como entrada, realizan determinados cálculos y devuelven un valor como resultado. La Figura 1 esquematiza este proceso:

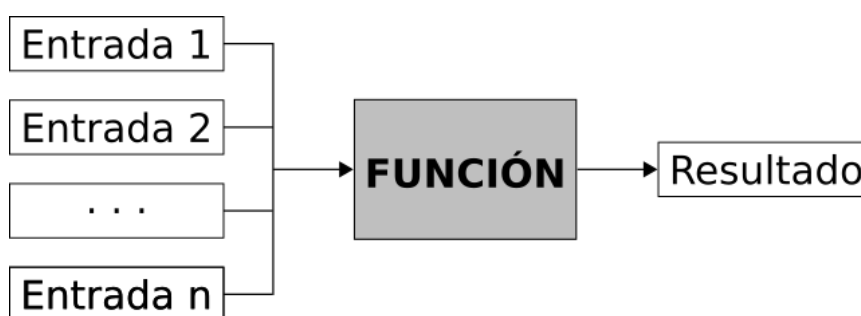


Figura 1: Esquema de funcionamiento de una *función*: hay unos valores de entrada, la función realiza determinados cálculos utilizando dichos valores y devuelve un resultado.

El código de las funciones consta de dos partes:

- **Signatura:** también llamada a veces la *cabecera* de la función, es la primera línea de código, en la cual se definen el nombre de la función, los valores de entrada que requiere la función y el tipo de datos del resultado que proporciona.
- **Cuerpo:** son las instrucciones de programa con las cuales la función realiza sus cálculos y devuelve el resultado.

A su vez, la *signatura* de la función consta de los siguientes elementos:

- Tipo de datos del valor devuelto: las funciones devuelven un valor de algún tipo de datos. Por ejemplo, si la función devuelve un valor del tipo *double*, se pondrá la palabra *double* como tipo de datos del valor devuelto. También puede suceder que la función no devuelva ningún valor, en cuyo caso se pone la palabra *void*.
- Nombre de la función: cada función tiene un nombre que debe ser único en su ámbito. Para los nombres de las funciones se siguen las mismas reglas que se vieron para los nombres de las variables.
- Lista de valores de entrada: a continuación del nombre de la función se pone, entre paréntesis y separados por comas, la lista de los valores de entrada a la función. Para cada valor, se escribirá su tipo de datos y un nombre de variable. Cada una de estas variables de entrada a la función recibe el nombre de *parámetro* de la función. La función puede tener cero o más parámetros de entrada.

El formato del código de una función se resume en la Figura 2:

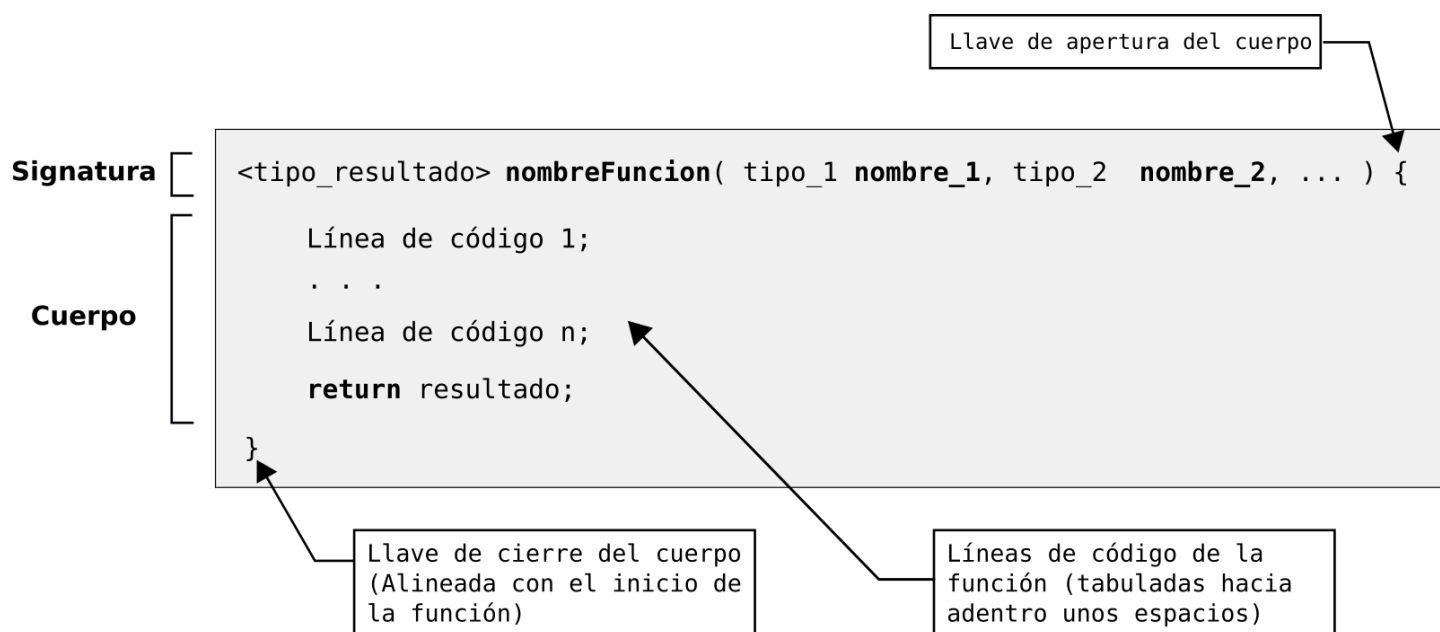


Figura 2: Componentes y formatos del código de una función.

El *cuerpo* son las instrucciones que sirven a la función para realizar sus cálculos. Se encierran entre dos llaves: la llave de apertura se pone al final de la línea de signatura y la de cierre tras la última línea de código y alineada con la línea de signatura.

Para favorecer la legibilidad del código, las líneas del cuerpo de la función se escriben alineadas 3 o 4 espacios más a la derecha que la línea de signatura.

Si la función devuelve un valor, dentro del cuerpo de la función habrá alguna instrucción *return*:

```
return resultado;
```

En la expresión anterior, *resultado* es el nombre de una variable cuyo valor será el que devuelva la función. Para este ejemplo se ha elegido como nombre de variable *resultado*, pero se puede utilizar cualquier otro.

El siguiente ejemplo de código corresponde a la definición una función, de nombre *cuadrado*, que recibe como entrada un número *double* y devuelve como resultado el cuadrado de dicho número:

```
double cuadrado(double x) {
    double resultado = x*x;
    return resultado;
}
```

Observe que la función define un parámetro de entrada del tipo *double* y de nombre *x*. Dentro del código de la función, se utiliza dicho nombre *x* para programar las operaciones que se harán con el valor que se reciba cuando se utilice la función con valores concretos para dicho parámetro. En este caso, el código de la función *cuadrado()* define una variable llamada *resultado* a la que se asigna el valor de  $x * x$ . Finalmente, la función devuelve el valor de la variable *resultado*.

### 3 LLAMADA A LAS FUNCIONES

Cuando se utiliza una función en un programa, se dice que se *llama* a la función, o que se hace *una llamada* a la función. También es habitual decir que se *invoca* la función.

Cuando se llama a una función desde una parte del programa, la ejecución pasa al código de la función. Cuando la función devuelve el resultado, la ejecución vuelve al punto del programa desde el que se hizo la llamada. La Figura 3 esquematiza este proceso. El orden de ejecución en dicho esquema sería:

1 → 2 → 3 → 4

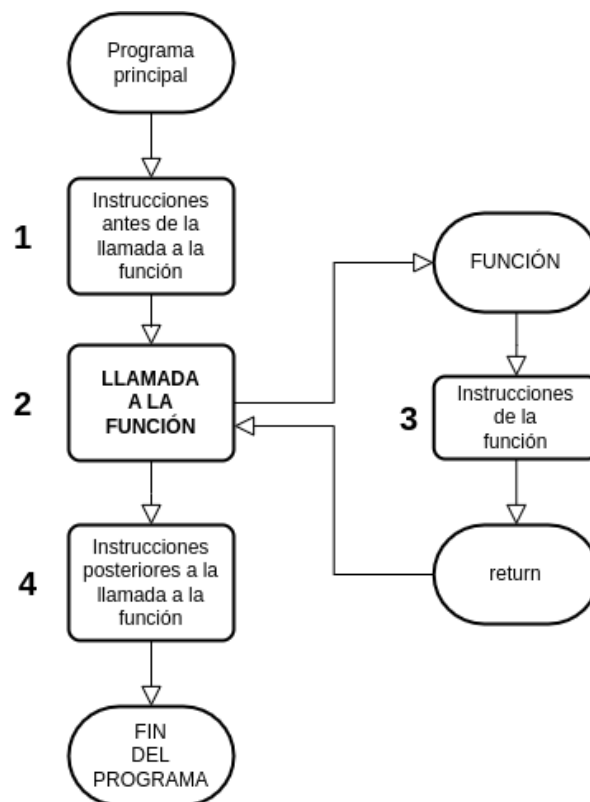


Figura 3: Esquema de ejecución en la llamada a una función.

Una vez definido el código de una función, para llamarla desde alguna parte del programa, se utiliza el nombre de la función, poniendo entre paréntesis los valores con los que se quiere hacer la llamada. El resultado se suele recoger en alguna variable del tipo adecuado a dicho resultado, como se hace en la siguiente línea de código:

```
double y = cuadrado(3.0);
```

Tras ejecutar esta línea de código, la variable *y* tendrá el valor *9.0*.

En la Figura 4 se puede ver el código de un programa que consta de dos funciones: la función *main()* y la función *cuadrado()*. En la función *main()*, se utiliza la función *cuadrado()* para calcular el cuadrado de 3.0 e imprimirlo en pantalla.

The image shows a code editor window titled 'ej1.c'. The code is as follows:

```
C ej1.c > ...
1  #include <stdio.h>
2
3  double cuadrado(double x) {
4      double resultado = x*x;
5      return resultado;
6  }
7
8  int main() {
9      double y = cuadrado(3.0);
10     printf("%.1f \n", y); // Imprime 9.0
11 }
```

Figura 4: Programa principal que utiliza una función llamada *cuadrado()*

Observa varias cosas del código anterior:

- Todos los programas tienen una función *main()* que hace las veces de programa principal. La ejecución del programa siempre empieza en la función *main()*.
- Para poder utilizar una función, es necesario que se haya definido antes de hacer la llamada. Por eso, el código de la función *cuadrado()* aparece antes que el código de la función *main()*, que es donde se hace la llamada a *cuadrado()*. Aunque el código de *cuadrado()* aparece antes, no se ejecuta hasta que se haga alguna llamada a la función dentro de *main()*.
- En la llamada a la función, se pone un valor del tipo de datos que se corresponda con el parámetro que se haya definido en la función. A dicho valor concreto que se pasa en la llamada a una función se le denomina *argumento*.

### Diferencia entre *parámetro* y *argumento*

No hay que confundir *parámetro* y *argumento*. El parámetro es la variable que se define en la declaración de la función, mientras que el argumento es el valor concreto con el que se hace la llamada a la función. En muchas ocasiones, se utilizan de manera indistinta los términos parámetro y argumento, pero hay ocasiones en las que distinguir entre ambos conceptos puede ser importante. Es la misma diferencia que existe entre el concepto de *variable* y el de *valor* asignado a dicha variable.

Al llamar a una función, en vez de utilizar un valor literal como argumento, se puede utilizar una expresión que de lugar a un valor del tipo adecuado. Observa el código de la Figura 5.

El argumento que recibe la función es el resultado de la expresión  $2.0 * w$ . En esta expresión,  $w$  se sustituye por el valor que tiene dicha variable en ese momento, que es 1.5, con lo que el argumento que finalmente le llega la función a través de su parámetro  $x$  es:

$$2.0 * w \Rightarrow 2.0 * 1.5 \Rightarrow 3.0$$

En el código de la función, en los lugares donde aparece el parámetro  $x$ , se sustituirá por el valor del argumento recibido, en este caso 3.0.

```

C ej1.c x
C ej1.c > main()
1  #include <stdio.h>
2
3  double cuadrado(double x) {
4      double resultado = x*x;
5      return resultado;
6  }
7
8  int main() {
9      double w = 1.5;
10     double y = cuadrado(2.0*w);
11     printf("%.1f \n", y); // Imprime 9.0
12 }

```

Figura 5: Llamada a una función utilizando una expresión como argumento

También es posible utilizar expresiones en las que alguno de los operandos sea una llamada a otra función.

## 4 EJEMPLO DE FUNCIONES

El código siguiente muestra un programa que solicita al usuario dos números enteros, calcula cuál de los dos es mayor, y muestra el resultado en pantalla:

### Ejemplo 1 Programa principal sin funciones

```

#include <stdio.h>

// Calcula el mayor de dos números
int main() {

    int n1, n2;
    printf("Teclee número entero:");
    scanf("%d", &n1);
    printf("Teclee número entero:");
    scanf("%d", &n2);

    int valor_mayor;
    if(n1 > n2) {
        valor_mayor = n1;
    } else {
        valor_mayor = n2;
    }

    printf("El valor mayor es: %d\n", valor_mayor);
}

```

```

shiguera@shigu...
$> ./ej2
Teclee número entero:35
Teclee número entero:47
El valor mayor es: 47
$> 

```

Vamos a *refactorizar* el programa anterior, añadiendo un par de funciones.

### Refactorización

El término *refactorización* se usa para describir la modificación del código fuente sin cambiar su comportamiento, lo que se conoce informalmente por *limpiar el código*.

Te recomiendo que eches un vistazo al siguiente artículo de la Wikipedia:

<https://es.wikipedia.org/wiki/Refactorización>

La primera modificación va a ser extraer una función que realice el cálculo del valor mayor de los dos teclea- dos por el usuario. Vamos a crear una función llamada *calcula\_valor\_mayor()* que reciba como argumentos dos valores enteros y devuelva un valor entero con el mayor de ellos. El código de la función podría ser el siguiente:

```
int calcula_valor_mayor(int p, int q) {
    int mayor;
    if(p>q) {
        mayor = p;
    } else {
        mayor=q;
    }
    return mayor;
}
```

El programa quedaría de la siguiente forma tras la modificación:

### Ejemplo 2 Programa refactorizado: extracción de una función

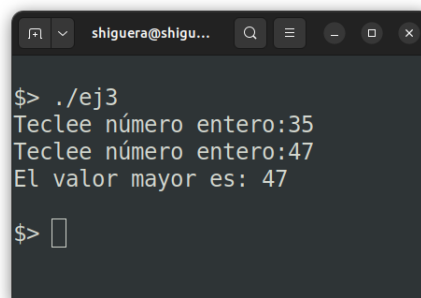
```
#include <stdio.h>

int calcula_valor_mayor(int p, int q) {
    int mayor;
    if(p>q) {
        mayor = p;
    } else {
        mayor=q;
    }
    return mayor;
}

int main() {
    // Entrada
    int n1, n2;
    printf("Teclee número entero:");
    scanf("%d", &n1);
    printf("Teclee número entero:");
    scanf("%d", &n2);

    // Procesamiento
    int valor_mayor = calcula_valor_mayor(n1, n2);

    // Salida
    printf("El valor mayor es: %d\n", valor_mayor);
}
```



```
$> ./ej3
Teclee número entero:35
Teclee número entero:47
El valor mayor es: 47
$> 
```

Observe que, tras extraer la función, el programa sigue funcionando exactamente igual que antes: el usuario no percibe la modificación.

La segunda refactorización que vamos a hacer es extraer una función que solicite un número entero al usuario, para eliminar la repetición de código que se produce en la parte correspondiente a la entrada del programa principal. En este caso, el código de la función podría ser el siguiente:

```
int pide_numero_entero() {
    int n;
    printf("Teclee número entero:");
    scanf("%d", &n);
    return n;
}
```

El programa con la modificación:

### Ejemplo 3 Programa refactorizado con dos funciones

```
#include <stdio.h>

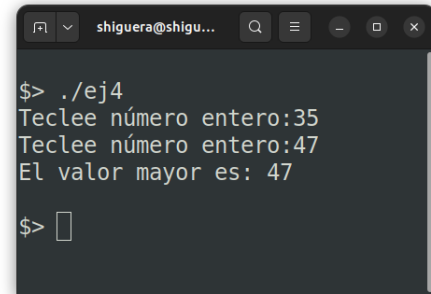
int pide_numero_entero() {
    int n;
    printf("Teclee número entero:");
    scanf("%d", &n);
    return n;
}

int calcula_valor_mayor(int p, int q) {
    int mayor;
    if(p>q) {
        mayor = p;
    } else {
        mayor=q;
    }
    return mayor;
}

int main() {
    // Entrada
    int n1 = pide_numero_entero();
    int n2 = pide_numero_entero();

    // Procesamiento
    int valor_mayor = calcula_valor_mayor(n1, n2);

    // Salida
    printf("El valor mayor es: %d\n", valor_mayor);
}
```



```
$> ./ej4
Teclee número entero:35
Teclee número entero:47
El valor mayor es: 47
$> 
```

El programa sigue funcionando igual, de cara al usuario. En cambio, si observas el código de la función *main()*, verás que es mucho más sencillo y más fácil de comprender. La sustitución de bloques de código por llamadas a funciones hace que el programa sea mucho más fácil de leer.

El programa resultante, tras las dos refactorizaciones que hemos hecho, tiene más líneas de código. El programa tenía inicialmente 23 líneas y ahora 30. Pero es más fácil de comprender, de probar, de modificar y de extender, si fuera necesario.

## 5 DEFINIR LAS FUNCIONES MEDIANTE PROTOTIPOS

Hemos dicho que, en lenguaje C, el compilador necesita que las funciones estén definidas antes de utilizarlas. Por eso, en el código de los ejemplos anteriores, poníamos el código de las funciones *pide\_numero\_entero()* y *calcula\_valor\_mayor()* antes del código de la función *main()*, que es donde se hacen las llamadas a las funciones.

Se puede utilizar otra técnica que consiste en poner antes de *main()* solo el *prototipo* de la función y poner el código completo de la misma tras la función *main()*. El prototipo es la línea de signatura, acabada en punto y coma.

La técnica de poner el prototipo de las funciones al principio del programa es la forma habitual de organizar el código. Se podría comparar con la declaración y asignación de valor de las variables: primero se *declara* la función poniendo el prototipo y luego se le *asigna valor* detallando el código completo de la misma.

El ejemplo anterior quedaría de la siguiente forma:



#### Ejemplo 4 Funciones con prototipo

```
#include <stdio.h>

// Prototipos
int calcula_valor_mayor(int p, int q);
int pide_numero_entero();

// Programa principal
int main() {
    int n1 = pide_numero_entero();
    int n2 = pide_numero_entero();

    int valor_mayor = calcula_valor_mayor(n1, n2);

    printf("El valor mayor es: %d\n", valor_mayor);
}

// Código de las funciones auxiliares
int calcula_valor_mayor(int p, int q) {
    int mayor;
    if(p>q) {
        mayor = p;
    } else {
        mayor=q;
    }
    return mayor;
}

int pide_numero_entero() {
    int n;
    printf("Teclee número entero:");
    scanf("%d", &n);
    return n;
}
```

#### Debugging

Es importante que comprendas perfectamente cuál es el flujo del programa cuando hay llamadas a funciones. Tienes que comprender bien en qué orden se realizan las distintas instrucciones.

Te recomiendo que pongas un *break point* al principio del programa de los ejemplos y lo ejecutes paso a paso en el depurador, hasta que comprendas perfectamente su funcionamiento.

## 6 BUENAS PRÁCTICAS EN LA CODIFICACIÓN DE FUNCIONES

### Nombre de las funciones:

Al igual que sucede en el caso de las variables, los nombres de funciones tienen que ser descriptivos e indicar claramente lo que hace la función.

Es habitual que los nombres de las variables sean sustantivos, mientras que los nombres de las funciones suelen ser verbos. Se puede utilizar el verbo en infinitivo o en imperativo.

Para el nombre de las funciones, igual que sucede con el nombre de las variables, se pueden seguir dos criterios:

- *camelCase*: primera palabra en minúsculas y siguientes palabras en mayúsculas, sin separación entre ellas.
- *snake\_case*: palabras en minúsculas separadas por el guión bajo.

Se puede utilizar uno u otro criterio, depende del estilo de programación de cada uno. Pero conviene ser consistente y utilizar el mismo criterio en todo el programa.

Los siguientes podrían ser nombres adecuados para una función:

`calculaPrecioNeto()`      `calcula_precio_netto()`

Un nombre largo, pero descriptivo, es mejor que un nombre corto y críptico. Dedicar tiempo para encontrar el nombre adecuado de las funciones, es un tiempo bien empleado. Un nombre adecuado ayudará a comprender mejor el objetivo de las funciones y ayudará a mejorar todo el programa. No importa probar varios nombres. Los IDEs como VSCode u otros hacen sencillo el proceso de cambiar el nombre de una función dentro de un proyecto, por lo que si se encuentra un nombre mejor que el que había, no hay que dudar en cambiarlo.

Hay funciones que devuelven un valor del tipo *boolean*. A estas funciones se les conoce como *predicados*. Es habitual en funciones que determinan si un objeto existe o cumple determinada condición. Son frecuentes nombres como los siguientes:

`isOpen()`      `exists_file()`

### **Single Responsibility Principle (SRP):**

Las funciones tienen que hacer una sola cosa y hacerla bien. Es mala práctica utilizar una función para hacer más de una cosa. En esos casos, es mejor dividir la función que hace dos cosas en dos funciones, cada una de las cuales hace una sola cosa.

### **Número de parámetros de las funciones:**

Las funciones tienen que tener cero, uno o dos parámetros. Tres parámetros o más parámetros suele ser señal de que la función está haciendo más de una cosa y que sería conveniente su descomposición en varias funciones más pequeñas.

En general, cuantos más parámetros tenga una función, más difícil será hacer los test que permitan comprobar que la función opera de forma correcta. Con un solo parámetro, la función será fácil de probar. Con dos parámetros, habrá que probar las combinaciones de los posibles valores de los dos parámetros. Con más de dos parámetros, las combinaciones posibles pueden ser inabordables.

Hay casos de funciones que no reciben ningún parámetro; suelen corresponder a lo que se denomina *acciones*: funciones que se limitan a realizar alguna tarea de entrada-salida, como escribir un mensaje en la pantalla, por ejemplo.

También puede suceder que una función que no recibe ningún parámetro se utilice para devolver el valor de una constante. En el ejemplo siguiente, se define una función de nombre *pi()*, que devuelve una aproximación del valor de  $\pi$ :

```
double pi() {  
    return 3.141592;  
}
```

En cuanto al valor devuelto, también puede suceder que una función no devuelva ningún valor. Puede tratarse de acciones, como las que se han comentado en párrafos anteriores.

### Las dos reglas de oro

La primera regla al codificar funciones es que tienen que ser pequeñas, en el sentido de tener pocas líneas de código. La segunda regla es que tienen que ser todavía más pequeñas.

En la metodología de programación conocida como *Extreme Programming, XP*<sup>1</sup>, se suele decir que una función no debe tener más líneas de las que caben en una pantalla.

Hay que tener en cuenta que dicha *regla* es de una época en la que las pantallas tenían unas 25 líneas de altura, de las cuales 2 o 3 quedaban ocupadas por la barra de menús o la línea de estatus, con lo cual quedaban unas 20 líneas efectivas para el código de una función.

Hoy en día, una pantalla de alta resolución con un tamaño de fuente pequeño puede permitir muchas más líneas de código, con lo que la regla queda desfasada. Esto no quiere decir que hoy día se puedan hacer funciones más grandes, muy al contrario, las funciones se deben mantener tan pequeñas como sea posible. El espíritu de esta regla es que el número de líneas de cada función debe mantenerse reducido. No debe asustar hacer funciones con solo una o dos líneas de código. En general, cuanto más pequeña sea una función, mejor.

---

<sup>1</sup>*Extreme Programming (XP)* es una metodología de desarrollo de software, desarrollada inicialmente por Kent Beck a finales del siglo pasado, cuyo objetivo es la producción de software de calidad en un entorno en el que los requerimientos del cliente varían frecuentemente. Está enmarcada en las técnicas de control de proyectos conocidas como *Agile Development*.

## 7 TERMINOLOGÍA UTILIZADA EN FUNCIONES

La Tabla 7 resume los términos habituales cuando se trabaja con funciones.

| Terminología utilizada en funciones |                                                                                                                                                                                                                                                                                                                                         |
|-------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Término                             | Explicación                                                                                                                                                                                                                                                                                                                             |
| <i>Función</i>                      | Bloque de código que se puede ejecutar de manera independiente dentro de un programa.                                                                                                                                                                                                                                                   |
| <i>Signatura</i>                    | Primera línea del código de una función, donde se definen el tipo de datos del valor devuelto, el nombre de la función y los parámetros de entrada a la misma.                                                                                                                                                                          |
| <i>Valor devuelto</i>               | Las funciones pueden devolver un valor.                                                                                                                                                                                                                                                                                                 |
| <i>void</i>                         | Palabra clave para indicar que la función no devuelve ningún valor.                                                                                                                                                                                                                                                                     |
| <i>Nombre</i>                       | Los nombres de funciones siguen el mismo criterio que los nombres de las variables. En C se puede usar el criterio <i>camelCase</i> : primera palabra en minúsculas y resto de palabras en mayúsculas y sin separaciones. También es frecuente usar el criterio <i>snake_case</i> : palabras en minúsculas separadas por el guion bajo. |
| <i>Parámetro</i>                    | Variable que define un valor de entrada a una función.                                                                                                                                                                                                                                                                                  |
| <i>Argumento</i>                    | Valor concreto con el que se hace la llamada a una función y que se sustituirá dentro del código de la misma por el parámetro correspondiente.                                                                                                                                                                                          |
| <i>Cuerpo</i>                       | Instrucciones del código de la función.                                                                                                                                                                                                                                                                                                 |
| <i>return</i>                       | Instrucción que se utiliza dentro del código de la función para devolver un valor.                                                                                                                                                                                                                                                      |
| <i>Lllamar, llamada</i>             | Ejecutar el código de la función.                                                                                                                                                                                                                                                                                                       |
| <i>Bloque de código</i>             | Instrucciones contenidas entre dos llaves.                                                                                                                                                                                                                                                                                              |
| <i>Ámbito</i>                       | Zona del programa donde una variable es accesible                                                                                                                                                                                                                                                                                       |
| <i>Visibilidad</i>                  | Se dice que una variable es <i>visible</i> cuando el programa está dentro de su ámbito y es posible acceder a su valor.                                                                                                                                                                                                                 |
| <i>Vida útil</i>                    | Periodo de tiempo durante la ejecución de un programa en el cuál una variable es accesible.                                                                                                                                                                                                                                             |
| <i>Variable local</i>               | Variable que solo existe dentro de un bloque de código.                                                                                                                                                                                                                                                                                 |