Contenidos

1. Introducción al concepto de subrutina o subprograma.
2. Definición de función C++.
3. Estructura de una función.
   a) Nombre de una función
   b) Tipo de retorno (tipo del valor devuelto)
   c) Valor retorno
   d) Lista de parámetros
      - Paso por valor
      - Paso por referencia
4. Los arrays y las estructuras como parámetros.
5. Declaración de funciones: Prototipos.

Introducción

En la mayoría de los casos, un determinado problema complejo lo podemos (y debemos) dividir en problemas más sencillos. Estos subproblemas se conocen en el contexto de la programación como "Módulos" o **subprogramas**.

Desde el punto de vista del diseño:

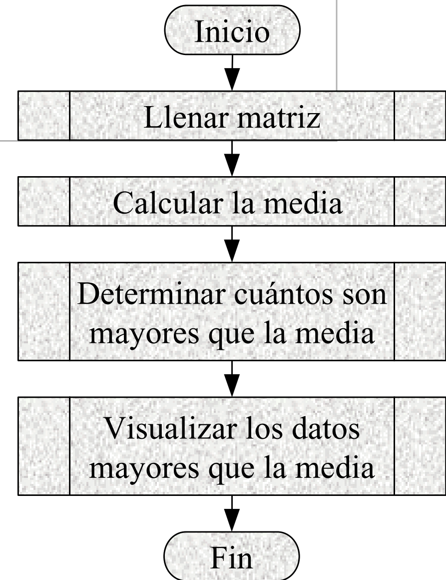
> Técnica de diseño conocida como **TOP DOWN**

- Se tratará de descomponer el problema original en partes.
- Se pueden codificar de forma independiente e incluso por diferentes personas.
- El problema final queda resuelto y estructurado en forma de módulos, lo que hace más sencilla su lectura y mantenimiento.

Diseño de algoritmos (diagramas de flujo)

Hoja 4 (Ejercicio 7)

7) Diseñar un algoritmo que llene una matriz de tamaño 3×4. Calcular la media de 12 valores almacenados en dicha matriz. Determinar cuántos son mayores que la media. Visualizar por pantalla los siguientes datos y en éste orden:
- Media,
- número de datos mayores que la media y
- lista de valores mayores que la media.



La solución a éste problema se podía descomponer en cuatro partes diferentes:

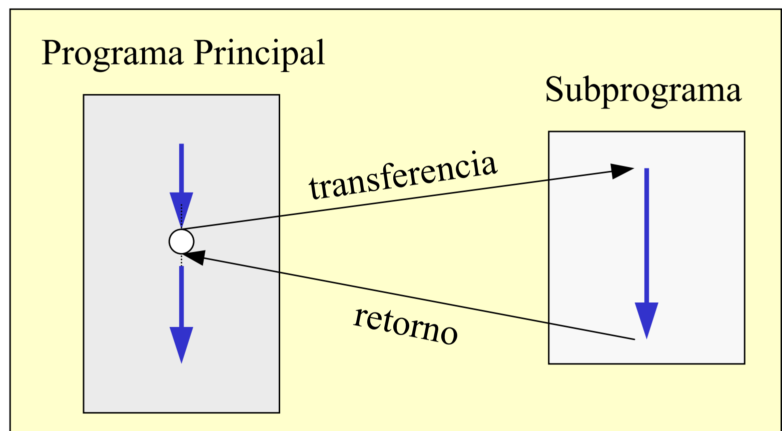
Ventajas:
- ahorro de espacio.
- más fácil entender lo que hace el algoritmo.
- más fácil de codificar.

Subprogramas

Un **subprograma** es una serie de instrucciones escritas independientemente del programa principal. Este subprograma está ligado al programa principal mediante un proceso de *transferencia/retorno*.

Transferencia
El control de ejecución se pasa al subprograma en el momento en que se requieren sus servicios.

Transferencia/retorno de **control** y **datos**



Retorno — El control de ejecución se devuelve al programa principal cuando el subprograma termina

Definición de FUNCIÓN

- C++ es un lenguaje modular, y por ésta razón, se puede dividir en varios módulos, cada uno de los cuales realiza una tarea determinada. Cada módulo es un subprograma llamado función.
- Una función es un miniprograma que se utiliza en un programa. Es un conjunto de sentencias que se pueden llamar desde cualquier parte del programa (incluso varias veces).
- Las funciones sirven para:
  - realizar tareas concretas y simplificar el programa
  - sirven para evitar escribir el mismo código varias veces.

Ventajas de utilizar funciones:

1.- Aislar mejor los problemas
2.- Escribir programas más rápido
3.- Programas más fáciles de mantener (más legibles y más cortos)

Ejemplo de uso de funciones:

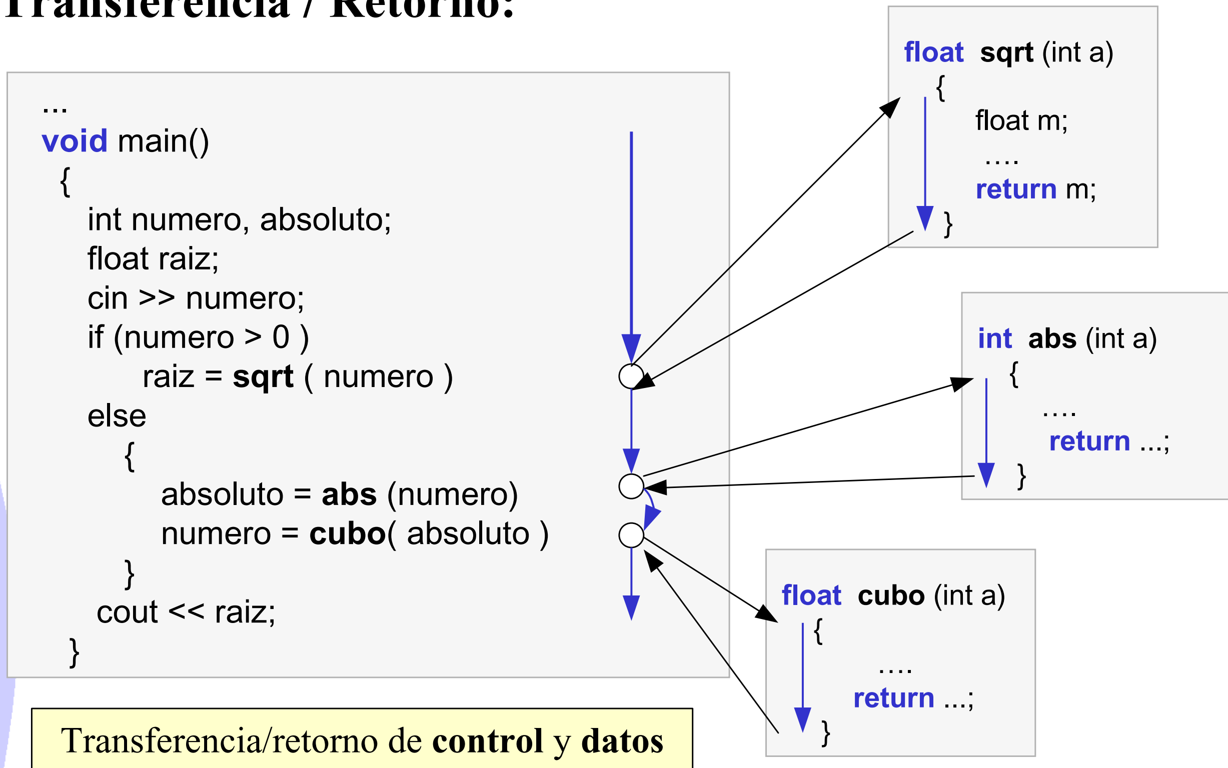
El ejemplo anterior se podría codificar de la siguiente manera:

```
...
void main()
 {
          ….
          llenar_matriz(m);
          calcular_media(m);
          mayores_media(m);
          imprimir_mayores(m);
 }
```

Frente a como lo estamos haciendo ahora

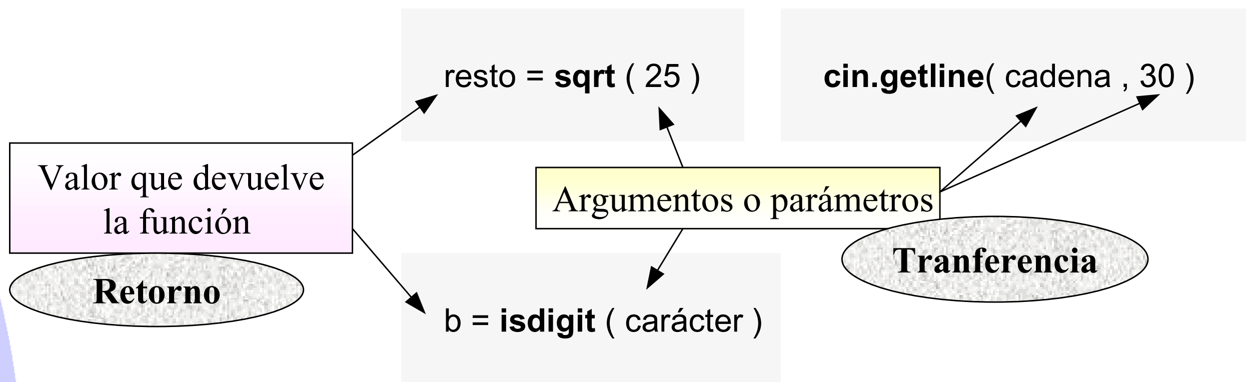
```
...
void main()
{
 ...
 ...
 for (int i=0; i<3; i++)
    for (int j=0; j<4; j++)
        cin >> m[i][j];
 suma = 0;
 for (int i=0; i<3; i++)
     for (int j=0; j<4; j++)
         suma = suma + m[i][j];
 media = suma/12;
 for (int i=0; i<3; i++)
       for (int j=0; j<4; j++)
           ...
  ...
 }
```

Transferencia / Retorno:



Estructura de una función

Hasta ahora, hemos visto y utilizado funciones estándar, es decir definidas en una biblioteca.



C++ nos permite definir nuestras propias funciones. Pocas veces veremos un programa que no use funciones. Una función "especial" y que se usa siempre es la función **main**.

Estructura de una función

La estructura de una función es la siguiente:

Sintaxis

```
<tipo_resultado> <nombre_de_la_función> ( lista_de_parámetros )
{
        cuerpo_de_la_función ;
        return  <expresión> ; //expresión es de tipo <tipo_resultado>
 }
```

Palabra reservada

Tipo_resultado: Es el tipo de dato que devuelve la función.
Expresión: valor que devuelve la función.
Lista de parámetros: aparecen con su tipo. La función utiliza éstos valores en el cuerpo.

```
int  maximo(int a, int b )
  {
      int m;
      if (a<b)
          m=b;
      else
          m=a;
     return m;
  }
```

Variable local

Estructura de una función

Una vez que se ha diseñado y codificado una función, se puede usar. Para usar una función, debemos ***llamarla o invocarla***. Una llamada, produce la ejecución de las instrucciones que se encuentran en el cuerpo.

Programa principal

```
void  main()
  {
      int x, y, mayor ;
      cin >> x >> y ;
      mayor = maximo( x, y);
      cout << mayor;
  }
```

```
int  maximo ( int a, int b )
  {
      int m;
      if (a<b)
          m=b;
      else
          m=a;
     return m;
  }
```

$maximo : int \times int \rightarrow int$

Estructura de una función: Nombre de la función

El nombre que se les da a las funciones, debe ser un identificador válido, es decir,

- Debe comenzar con una letra o subrayado (_).
- Después de la primera letra pueden aparecer otras letras, dígitos y caracteres.
- No debe contener espacios en blanco.
- C++ distingue entre mayúsculas o minúsculas.

Nombres de funciones : **_leer** , **visualizar_tabla_1** , **leer_matriz** , etc ...

Es muy importante en la fase de diseño de un algoritmo, utilizar nombres que nos permitan intuir la tarea que realizan las funciones, sobre todo a la hora de mantener y modificar programas.

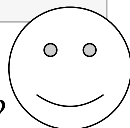
Estructura de una función: Nombre de la función

Por ejemplo, qué hacen los siguientes programas:

```
...
void main()
 {
          ....
        llenar_matriz(m);
        calcular_media(m);
        mayores_que_la_mediam(m);
        imprimir_mayores(m);
 }
```

```
...
void main()
 {
          ....
          primera_funcion(m);
          segunda_funcion(m);
          funcion_3(m);
          mi_fincion(m);
 }
```

Parece más intuitivo, ¿no?



Si se nos pide un cambio en algún punto del programa, por la forma de imprimir la matriz,

¿qué función hemos de modificar?

Estructura de una función: Tipo de dato de retorno

Las funciones en C++ las podemos dividir en varios tipos:

- Funciones que realizan una tarea específica pero que **no devuelven valores** al programa principal o a la función que la llamó.

El tipo de dato de retorno ha de ser → **void**

Se llaman *Procedimientos*

- Funciones que realizan operaciones con los argumentos o manipulan datos y **devuelven un valor**. Dicho valor, puede ser el resultado de esas operaciones ó un indicador de si la manipulación de los datos ha sido exitosa o no.

Si la función devuelve un valor, ha de ser uno de los siguientes:

Tipos simples: **int**, **char**, **float**, **...**

Un tipo **struct**

Cualquier objeto o puntero de C++

Lo veremos más adelante

Estructura de una función: Tipo de dato de retorno

Ejemplos:

```
float  media (float x, float  y )
   {
     ….
   }
```

```
int  maximo (int a, int b )
   {
     ….
   }
```

```
char  siguiente_car (char c )
   {
     ….
   }
```

```
disco  buscar_cd ( int num )
   {
     ….
   }
```

```
bool  encontrado (  )
   {
     ….
   }
```

```
void  visualizar_vector( int v[ ] )
   {
     ….
   }
```

Estructura de una función: Tipo de dato de retorno

Ejemplo:

```
struct complejo
  {
     int real;
     int imaginaria;
  };
```

Función

```
complejo crea_complejo( int a, int b )
  {
   complejo c;
   c.real = a;
   c.imaginaria = b;
   return c;
  }
```

```
void main()
{

   complejo m;
   m = crea_complejo(2,4);
   …
   …

}
```


Estructura de una función: Valor de retorno

Una función solo puede devolver un valor. El valor se devuelve mediante la sentencia **return**

```
return  <expresión> ;
```

1. C++ comprueba la compatibilidad de tipos, (no se puede devolver un valor de tipo int, si el tipo de retorno es struct).

2. Una vez que se ejecuta ésta sentencia, termina la ejecución de la función.

3. Una función puede tener cualquier número de sentencias return, pero **al menos debe haber una. Cuando se ejecuta return se sale de la función.**

4. El valor devuelto puede ser: una constante, variable ó una expresión.

Funciones

Estructura de una función: Valor de retorno

Ejemplos:

```
complejo crea_complejo( int a, int b )
  {
   complejo c;
   c.real = a;
   c.imaginaria = b;
   return a;
  }
```

```
bool funcion( int a )
{
   bool negativo;
   if (a <0)
     {
       negativo = true;
       return negativo;
     }
   while (a < 100)
   {
      cout << a;
      a++;
   }
   return false;
}
```

```
void main()
{
  bool resultado;
  resultado = funcion (-5);
  resultado = funcion (5);
}
```


Funciones

Estructura de una función: Valor de retorno

Ejemplos:

```
int  suma_tres ( int a , int b, int c )
 {
      return (a+b+c);
  }
```

```
bool  dividir ( int a , int b, float& cociente )
 {
     if   ( b = 0 )
        return false;
     else
           cociente = a/b;
     return true;
 }
```

```
void main()
{
   int resultado;
   bool ok;
   resultado = suma_tres (2,  x, y );
   ok = dividir (0, 3, resultado);
   if (ok ==true)
         cout << resultado;
   else
        cout << "error-división   por cero";
   cout << resultado;
}
```

Estructura de una función: Valor de retorno

```
resultado = suma (6 , 8 );
```

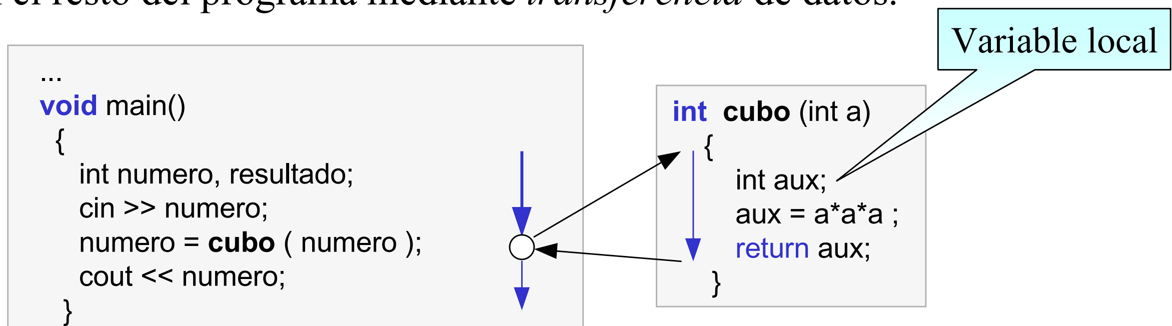
Cuando se llama a una función, puede haber una variable que guarde el valor que devolverá la función, es decir, llamaremos a la función mediante una sentencia de asignación.

Estructura de una función: Lista de parámetros

Las funciones trabajan con dos tipos de datos:

1. Variables locales: declaradas en el cuerpo de la función. Estas variables solo son conocidas dentro de la función y se crean y se destruyen con la función.

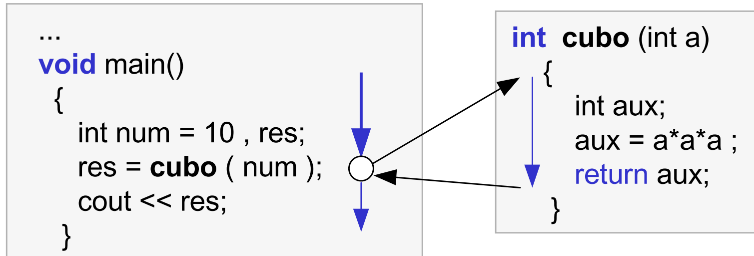
2. Parámetros: Los parámetros permiten la comunicación de la función con el resto del programa mediante *transferencia* de datos.

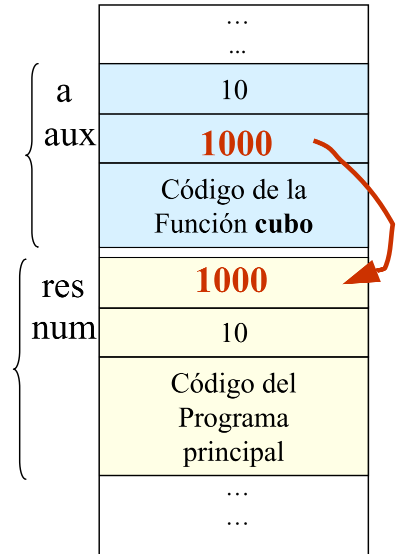


Estructura de una función: Lista de parámetros

C++ proporciona dos métodos para realizar ésta transferencia de datos a la función. Hablaremos a partir de ahora de *paso de parámetros a la función.*

1. Paso de parámetros por valor





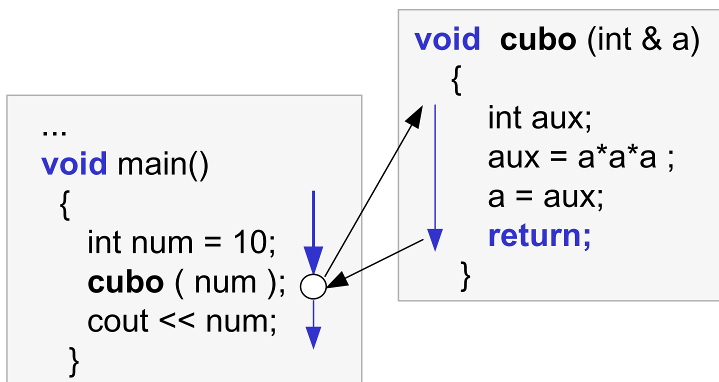
* El programa principal se interrumpe para comenzar la ejecución de la función

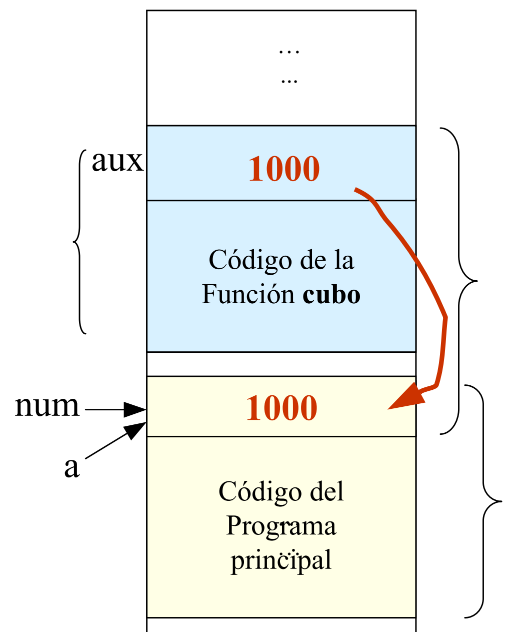
* Se reserva memoria para el código de la función, para las variables locales y para los parámetros.

MEMORIA

Estructura de una función: Lista de parámetros

2. Paso de parámetros por referencia





Con & el compilador no reserva memoria para los parámetros, sino que utiliza la misma porción de memoria. Por tanto los cambios afectan a la variable

MEMORIA

Estructura de una función: Lista de parámetros

Paso de parámetros por valor

- Cuando se llama a la función, se pasa solo el valor de la variable.
- Este método también se llama *paso por copia.*
- El compilador hace una copia *de los parámetros.* Esto implica que cualquier modificación en el valor de los parámetros no se mantiene cuando termina la función.
- Utilizaremos éste método cuando no necesitemos que se modifiquen los parámetros con los que se llama.

La mayoría de los ejemplos que hemos visto hasta ahora, utilizan parámetros por valor.

Estructura de una función: Lista de parámetros

Paso de parámetros por referencia

- También se llama *paso por dirección.*

- Cuando se llama a la función, se pasa la dirección de memoria donde se encuentra almacenada la variable parámetro.

- El compilador no hace copia, no reserva memoria para los parámetros.

- Usaremos éste método cuando necesitamos que la función modifique el valor de los parámetros y que devuelva el valor modificado.

Para pasar un parámetro por referencia, hay que poner el operador de dirección & detrás del tipo del parámetro.

```
void  cubo (int & a)
   {        ....     }
```

Ejemplo de uso de paso de parámetros

```
void main()
{
  int m;
  m = area_rectangulo( 2 , 3 );
  cout << m ;

  int lado1 = 2, lado2 = 6 ;
  m = area_rectangulo( lado1 , lado2 );
  cout << m;

  int b = 10, e = 4, r= 0;
  potencia (b, e, r);
  cout << r;

  }
```

```
int area_rectangulo (int a, int b)
{
  int aux;
  aux = a*b;
  a=0;
  b=0;
  return aux;
}
```

```
void potencia( int x, int y, int& z)
{
   z = 1;
   for ( int i=1;  i<= y ; i++ )
       z = z * x ;
}
```

Parámetros por valor: a, b, x, y   Parámetros por referencia: z

Paso de parámetros

Los arrays como parámetros:

No hace falta añadir el operador de dirección & cuando el parámetro que se pasa en un array.

Los arrays se pasan siempre por referencia aunque no lleven &.

No es necesario indicar el tamaño del array en la declaración.

```
void  visualizar_matriz( int m[3][4] )
  { … }
```

```
void  visualizar_vector( int v[ ] )
  {  ….    }
```

Las estructuras como parámetros:

Las estructuras, si son muy grandes, se pasan por referencia haciendo uso del operador & como es habitual para el resto de parámetros.

Funciones

Declaración de las funciones : Prototipos

A excepción de la función main(), en el módulo del programa debe aparecer la declaración de las funciones que se utilicen en dicho módulo. Esta declaración recibe el nombre de PROTOTIPO.

<tipo_resultado> <nombre_de_la_función> (*lista_de_parámetros*) **;**

Sintaxis del prototipo

```
#include <iostream.h>

void  potencia (int x, int y, int& z );

void main()
{
...
}

void potencia( int x, int y, int& z)
{
 ….
}
```

Prototipo

Codificación

El prototipo, informa de la existencia de la función, el tipo de datos que devuelve y los parámetros que tiene definidos.

Funciones

Características importantes relativas a funciones

1. La instrucción **return**
   a) fuerza la salida inmediata de la función.
   b) sirve para devolver un valor. Dicho valor puede ser constante, variable ó una expresión.

return (4+i);	**return 7;**	**return x;**

2. No se pueden declarar (DECLARAR != USAR) unas funciones dentro de otras.

3. Las constantes, variables y tipos de datos declarados en el cuerpo de la función son locales a la misma y no se pueden utilizar fuera de ella.

4. El cuerpo de la función encerrado entre llaves, no acaba en ';'.