Índice general

TEMA 3. MODULARIZACIÓN

3.1. FUNCIONES

3.1.1. INTRODUCCIÓN A LAS FUNCIONES

3.1.1.1. LAS FUNCIONES REALIZAN UNA TAREA

***Objetivo*: Descomponer la resolución de un problema en tareas menos complejas.**

Cada tarea que identifiquemos será resuelta por un algoritmo , y éste vendrá *encapsulado* en una función .

Las funciones como `sqrt`, `tolower`, etc., no son sino ejemplos de funciones incluidas en `math` y `ctype` respectivamente que resuelven tareas concretas, devolviendo un valor. Suelen ser de notación prefija, con argumentos (en su caso) separados por comas y encerrados entre paréntesis.

```
int main(){
   float lado1, lado2, hip, aux;

   <Asignación de valores a los lados>

   aux = lado1*lado1+lado2*lado2;
   hip = sqrt(aux);
   ..............................
}
```

Si queremos realizar la misma operación para dos triángulos:

```
.............................
int main(){
   float lado1_A, lado2_A, lado1_B, lado2_B,
          hip_A, hip_B,
          aux_A, aux_B;

   <Asignación de valores a los lados>

   aux_A = lado1_A*lado1_A+lado2_A*lado2_A;
   hip_A = sqrt(aux_A);

   aux_B = lado1_B*lado1_B+lado2_B*lado2_B;
   hip_B = sqrt(aux_B);
   .............................
}
```

¿No sería más claro, menos propenso a errores y más reutilizable si existiese en alguna biblioteca la función `Hipotenusa`**, de forma que el código fuese el siguiente?**

```
.............................
int main(){
   float lado1_A, lado2_A, lado1_B, lado2_B,
          hip_A, hip_B,

   <Asignación de valores a los lados>

   hip_A = Hipotenusa(lado1_A, lado2_A);
   hip_B = Hipotenusa(lado1_B, lado2_B);
   .............................
```

En este ejemplo, `Hipotenusa` **es una función que resuelve la tarea de calcular la hipotenusa de un triángulo, dado el valor de los lados.**

Podría dar la sensación de que primero debemos escribir el programa con todas las sentencias y luego construiríamos la función `Hipotenusa`**. Esto no es así: el programador debe identificar las funciones antes de escribir una sola línea de código**

3.1.1.2. DEFINICIÓN

<*tipo*> <*nombre-función*> ([<*parám. formales*>]) {
[<*sentencias*>]

return <*expresión*>;
}

- **Por ahora, la definición se pondrá después de la inclusión de bibliotecas y antes del `main`. En general, antes de usar una función en cualquier sitio, hay que poner su definición.**
- **Diremos que**
  <*tipo*> <*nombre-función*> (<*parám. formales*>)
  es la cabecera de la función.
- **El cuerpo de la función debe contener:**

  `return` **<*expresión*>;**

  donde <*expresión*> ha de ser del mismo tipo que el especificado en la cabecera de la función (también puede ser un tipo *compatible*). El valor que contenga dicha expresión es el valor que devuelve la función.

```
float Cuadrado(float entrada){
   return entrada*entrada;
}
```

3.1.1.3. PARÁMETROS FORMALES Y ACTUALES

- **Los *parámetros formales* son aquellos especificados en la cabecera de la función.**
  Al declarar un parámetro formal hay que especificar su tipo de dato.
  Los parámetros formales sólo se conocen dentro de la función.
- **Los *parámetros actuales* son las expresiones pasadas como argumentos en la llamada a una función.**
  ***Llamada*:**

  <nombre-función> (*<lista parámetros actuales>*);

```
float Cuadrado(float entrada){
   return entrada*entrada;
}
int main(){
   float resultado, valor;

   valor = 4;
   resultado = Cuadrado(valor);     /* resultado = 16 */
   printf("El cuadrado de %f es %f", valor, resultado);
}
```

***Nota*. En inglés se usan los términos *formal and actual parameters*. Tradicionalmente se mantuvo la traducción *literal*. Hubiese sido mejor llamarlos *parámetros genéricos* y *parámetros de llamada* o algo así.**

▶ *Flujo de control*

Cuando se ejecuta la llamada `resultado = Cuadrado(valor);` **el flujo de control salta a la definición de la función.**

- ▷ **Se realiza la correspondencia entre los parámetros.**
  El correspondiente parámetro formal recibe una copia del parámetro actual, es decir, se realiza la siguiente *asignación en tiempo de ejecución*:

  parámetro formal* = *parámetro actual

  En el ejemplo, `entrada = 4`

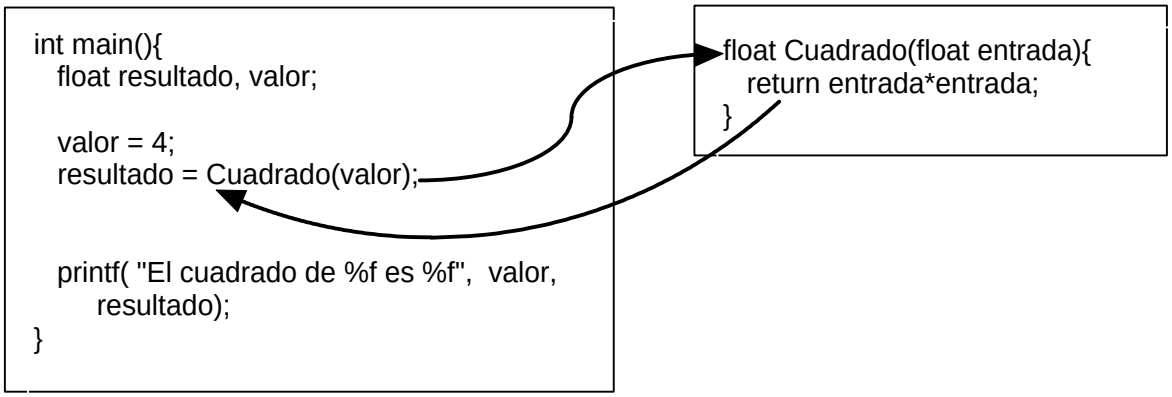
  Esta forma de pasar parámetros se conoce como *paso por valor*.

- ▷ **Empiezan a ejecutarse las sentencias de la función y cuando se llega a alguna sentencia** `return` $<$***expresión***$>$**, la función termina y devuelve** $<$***expresión***$>$ **al sitio en el que fue llamada.**

  La llamada a una función es una expresión

  `Cuadrado(valor)` **es una expresión.**

- ▷ **A continuación, el flujo de control prosigue por la línea siguiente a la llamada.**



Ejercicio. Definid la función `Hipotenusa`

▶ *Correspondencia entre parámetros actuales y formales*

▷ **Debe haber exactamente el *mismo número* de parámetros actuales que de parámetros formales.**

```
float Cuadrado(float entrada){
   return entrada*entrada;
}
int main(){
   float resultado;
   resultado = Cuadrado(5, 8);  /* Error en compilación */
}
```

▷ **Debemos garantizar que el parámetro actual tenga un valor correcto antes de llamar a la función.**

```
float Cuadrado(float entrada){
   return entrada*entrada;
}
int main(){
   float resultado, valor;
   resultado = Cuadrado(valor);  /* Error lógico */
}
```

▷ **La correspondencia se establece por *orden de aparición*, uno a uno y de izquierda a derecha.**

```
#include <stdio.h>

float Resta(float valor_1, float valor_2){
   return valor_1 - valor2;
}
int main(){
   float un_valor = 5.0, otro_valor = 4.0;
```

```
    printf("\nResta = %f", Resta(un_valor, otro_valor));
    printf("\nResta = %f", Resta(otro_valor, un_valor));
}
```

▷ **El parámetro actual puede ser una expresión. Primero se evalúa la expresión y luego se realiza la llamada a la función.**

```
hip = Hipotenusa(ladoA+3,ladoB*5);
```

▷ **Cada parámetro formal y su correspondiente parámetro actual han de ser del *mismo tipo* (o *compatible*)**

```
float Cuadrado(float entrada){
    return entrada*entrada;
}
int main(){
    float resultado;
    int valor = 7;
    resultado = Cuadrado(valor);
}
```

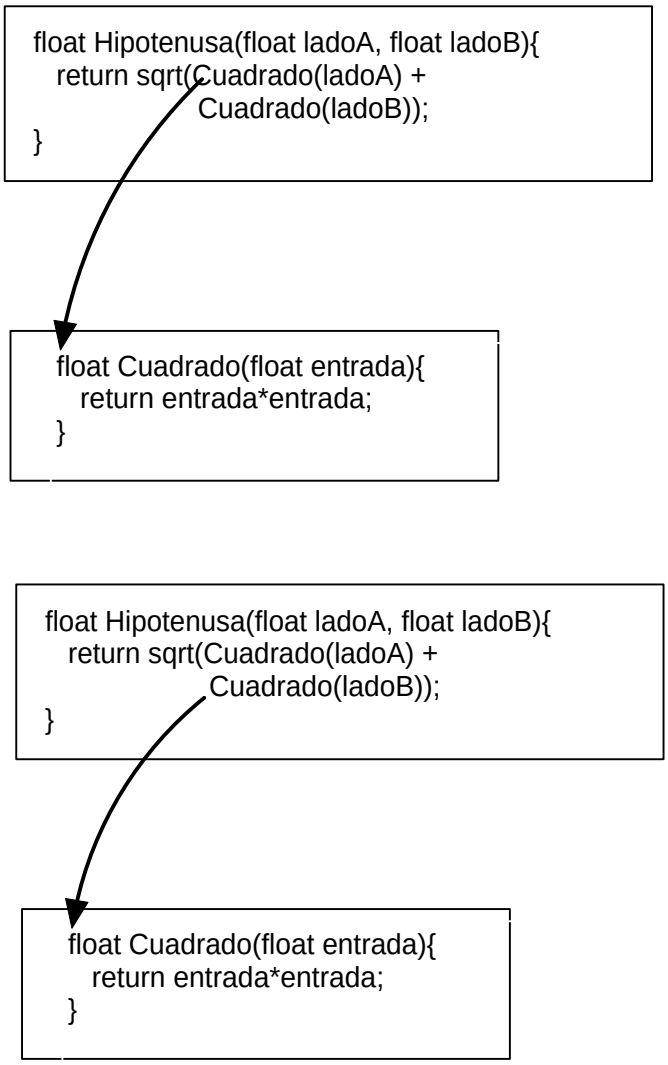
***Problema*: que el tipo del parámetro formal sea más *pequeño* que el actual. El formal se puede quedar con basura.**

```
int Cuadrado(int entrada){
    return entrada*entrada;
}
int main(){
    int resultado;
    resultado = Cuadrado(400000000000);
                /* devuelve basura */
```

Ejercicio. **Definid la función `Media` para que devuelva la media aritmética de dos reales.**
Incluid también la función `Cuadrado` anterior.
En el `main`, calculad en una única sentencia el cuadrado de la media aritmética entre dos reales e imprimid el resultado.

▷ **Dentro de una función se puede llamar a cualquier otra función que esté definida con anterioridad. El paso de parámetros entre funciones sigue los mismos criterios.**

```
#include <stdio.h>
#include <math.h>

float Cuadrado(float entrada){
   return entrada*entrada;
}

float Hipotenusa(float ladoA, float ladoB){
   return sqrt(Cuadrado(ladoA) + Cuadrado(ladoB));
}
```


float Hipotenusa(float ladoA, float ladoB){
return sqrt(Cuadrado(ladoA) +
Cuadrado(ladoB));
}
float Cuadrado(float entrada){
return entrada*entrada;
}
float Hipotenusa(float ladoA, float ladoB){
return sqrt(Cuadrado(ladoA) +
Cuadrado(ladoB));
}
float Cuadrado(float entrada){
return entrada*entrada;
}

3.1.1.4. ENTRADAS Y SALIDA DE UNA FUNCIÓN

Las funciones encapsulan un algoritmo:

- ▷ **Nombre de la función → Nos indica qué hace el algoritmo**
- ▷ **Parámetros → Datos de entrada del algoritmo**
- ▷ **Valor devuelto por la función en la sentencia `return` → Dato de salida del algoritmo**

> ***Viendo la cabecera de una función,***
> ***sabemos qué hace***
> ***y qué necesita para hacerlo.***
> ***No nos importa cómo lo hace***

Ejemplo. **Hipotenusa:**

- ▷ ***Cabecera de la función*:**
  `float Hipotenusa (float lado1, float lado2)`
- ▷ ***Datos de entrada*: lado1, lado2 (reales)**
- ▷ ***Dato de salida*: hipotenusa (real)**

Ejemplo. **Cuadrado:**

- ▷ ***Cabecera de la función*:**
  `float Cuadrado (float valor)`
- ▷ ***Datos de entrada*: valor (real)**
- ▷ ***Dato de salida*: cuadrado (real)**

Ventajas en el uso de funciones:

- **Reutilización . Definimos una única vez la función y la llamamos donde sea necesario.**
- **Código menos propenso a errores . Al estar el código de la función escrito una única vez, es menos propenso a errores (en un `copy-paste` posiblemente se nos olvidará incluir una línea)**
- **Actualización . Ante posibles cambios futuros, sólo debemos cambiar el código que hay dentro de la definición de la función.**
- **Abstracción . En la llamada a la función,**

  ```
  hip = Hipotenusa(lado1, lado2);
  ```

  sólo nos preocupamos de saber su nombre y cómo se utiliza (los parámetros y el valor devuelto), pero no de saber cómo lo hace.

3.1.2. ÁMBITO DE UN DATO

- **Los parámetros formales son como variables que sólo existen dentro de la función.**
- **Además, dentro de una función podemos declarar otras constantes y variables. Sólo se conocen dentro de la función y se les llama *datos locales*.**

```
<tipo> <nombre-función> (<lista parámetros formales>) {
  [<Constantes Locales>]
  [<Variables Locales>]
  [<Sentencias>]

  return <expresión>;
}
```

Al igual que ocurre con la declaración de variables del `main`, las variables locales a una función no inicializadas a un valor concreto tendrán un valor indeterminado (*basura*) al inicio de la ejecución de la función.

Ejemplo. **Calculad el factorial de un valor.**

Recordemos la forma de calcular el factorial de un entero `n`:

```
scanf("%d", &n);
fact = 1;
for (i=2; i<=n ; i++)
   fact = fact*i;
```

- ▷ ***Datos de entrada*****: `n` (entero)**
- ▷ ***Dato de salida*****: factorial de `fact` (entero)**
- ▷ ***Cabecera de la función*****: `int Factorial (int n)`**

```
.............
int Factorial (int n){
   int i;
   int fact = 1;

   for (i=2; i<=n; i++)
      fact = fact * i;

   return fact;
}
int main(){
   int valor, resultado;
   printf("\nIntroduzca valor");
   scanf("%d", &valor);

   resultado = Factorial(valor);
   printf("Factorial de %d = %d", valor, resultado);
}
```

```
#include <stdio.h>

int FactorialMAL (){
   int i, fact = 1;
   int n;

   scanf("%d", &n);          /*  E/S  SUSPENSO */

   for (i=2; i<=n; i++)
      fact = fact * i;

   return fact;
}

int main(){
   int resultado;

   resultado = FactorialMAL();
   printf("%d", resultado);
}
```

Dentro de una función no podemos acceder a datos definidos en otras funciones ni a los de `main`.

```
#include <stdio.h>

int Factorial (int n){
   int i;
   int fact = 1;

   valor = 5;   /*  Error de compilación */

   for (i=2; i<=n; i++)
      fact = fact * i;

   return fact;
}

int main(){
   int valor, resultado;

   i = 5;    /*  Error de compilación */

   printf("\nIntroduzca valor");
   scanf("%d", &valor);

   resultado = Factorial(valor);
   printf("Factorial de %d = %d", valor, resultado);

}
```

Ámbito **(*scope*) de un dato (variable o constante)** `v` **es el conjunto de todos aquellas funciones que pueden referenciar a** `v`**.**

- ▷ **Sólo puede usarse en la propia función en el que está definido (ya sea un parámetro formal o un dato local)**
- ▷ **No puede usarse en otras funciones ni en el programa principal.**
- ▷ **En C++ se permiten definir variables locales a un bloque: Y NO EN C**

```
while (condicion){
   int i;
   ............
}

for (i=0; i<10; i++){
   ............
}
```

En este caso, el ámbito de la variable termina con la llave `}` **que cierra la estructura condicional, repetitiva, etc.**

Consejo:	**No abusad en el uso de las variables definidas en un bloque. Restringidlo por ahora a variables auxiliares y contadores**

Al estar perfectamente delimitado el ámbito de un dato, los nombres dados a los parámetros formales pueden ser iguales a los actuales.

```
#include <stdio.h>

int Factorial (int n){      /* <- n de Factorial */
   int i;
   int fact = 1;

   for (i=2; i<=n; i++)
      fact = fact * i;

   return fact;
}

int main(){
   int n = 3 , resultado;   /* <- n de main */

   resultado = Factorial(n);
   printf("Factorial de %d = %d", n, resultado);

         /* Imprime en pantalla lo siguiente:
            Factorial de 3 = 6 */
}
```

```
#include <stdio.h>
#include <math.h>

float Hipotenusa(float ladoA, float ladoB){
   return sqrt(ladoA*ladoA+ladoB*ladoB);
}

int main(){
   float ladoA, ladoB, hip;

   printf("\nIntroduzca primer lado");
   scanf("%f", &ladoA);
   printf("\nIntroduzca segundo lado");
   scanf("%f", &ladoB);

   hip = Hipotenusa(ladoA,ladoB);
   printf("\nLa hipotenusa vale %f", hip);
}
```

Incluso podemos cambiar el valor del parámetro formal, que el actual no se modifica.

```
#include <stdio.h>

int Suma_desde_0_hasta(int tope){
    int suma;

    suma = 0;
    while (tope>0){
        suma = suma + tope;
        tope--;
    }

    return suma;
}

int main(){
  int tope=5, resultado;

  resultado = Suma_desde_0_hasta(tope);
  printf("Suma hasta %d = %d", tope, resultado);

        /* Imprime en pantalla lo siguiente:
           Suma hasta 5 = 13 */
}
```

En la medida de lo posible, procurad darles nombres distintos a los parámetros formales y actuales.

3.1.3. La Pila

Cada vez que se llama a una función, se crea un entorno de trabajo asociado a él, en una zona de memoria específica: la *pila* (*stack*).

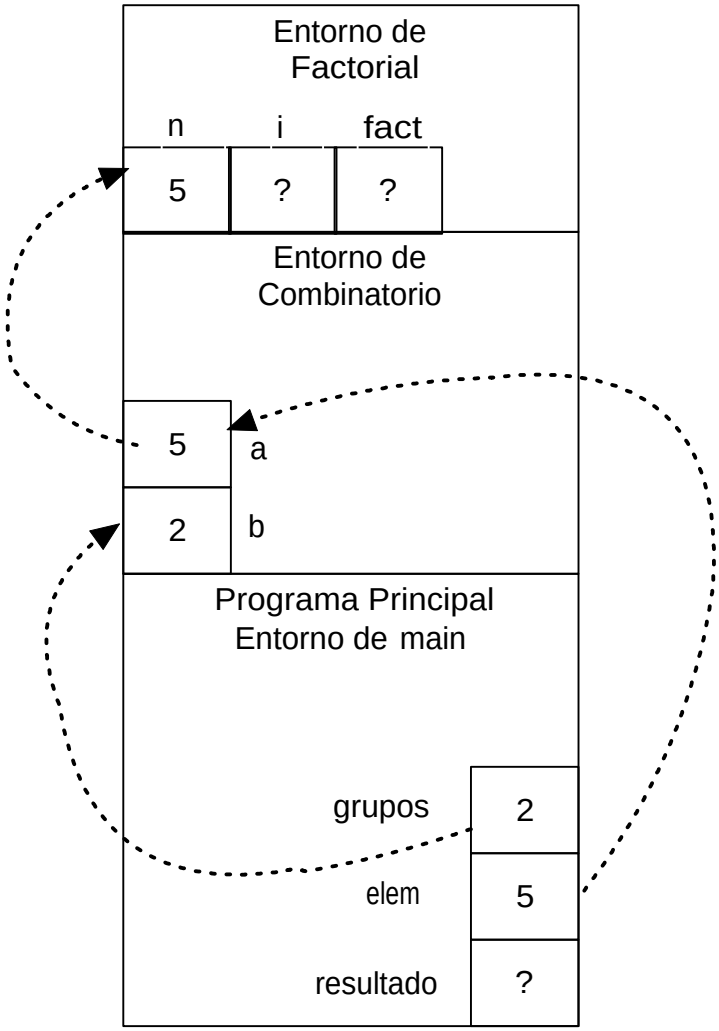
- **Cada entorno, sólo puede acceder a sus propios datos. Es decir, durante la ejecución de la función, no se tiene acceso al parámetro actual puesto que están en entornos de memoria distintos. Por tanto, las modificaciones en el parámetro formal no afectan al actual.**

  ***Nota*. Existen mecanismos para acceder a otras zonas (paso por referencia y punteros). Veremos el paso por referencia en otra sección.**

- **En el entorno se almacenan, entre otras cosas:**
  - **Los parámetros formales**
  - **Los datos locales (constantes y variables).**
  - **La *dirección de retorno* de la función.**

- **Cuando una función llama a otra, sus respectivos entornos se almacenan apilados uno encima del otro. Hasta que no termine de ejecutarse la última función llamada, el control no pasará a la anterior.**

```
#include <stdio.h>

int Factorial (int n){
   int i;
   int fact = 1;

   for (i=2; i<=n; i++)
      fact = fact * i;

   return fact;
}

int Combinatorio(int a, int b){
   return Factorial(a)/(Factorial(b) * Factorial(a-b));
}

int main(){
   int resultado, elem=5, grupos=2;

   resultado = Combinatorio(elem,grupos);
   printf("%d sobre %d = %d", elem, grupos, resultado);
}
```


Entorno de
Factorial
n
i
fact
5
?
?
Entorno de
Combinatorio
5
a
2
b
Programa Principal
Entorno de main
grupos
2
elem
5
resultado
?

Pila

`main` es de hecho una función como otra cualquiera, por lo que también se almacena en la pila. Es la primera función llamada al ejecutarse el programa.

Devuelve un entero al Sistema Operativo y puede tener dos parámetros más, int argc char* argv[]. La declaración completa sería:

```
int main(int argc, char* argv[]){
   ...........
}
```

- **Si el programa termina sin errores, se debe devolver 0**
  Puede indicarse incluyendo `return 0;` al final de `main` (antes de }}
  En C, si no se devuelve nada, se devuelve 0 por defecto, por lo que podríamos suprimir `return 0;`
- **Si el programa termina con un error, debe devolver un entero distinto de 0**

3.1.4. EJEMPLOS DE FUNCIONES

Ejercicio. **Comprobad si un número es par.**

Ejercicio. **Calculad el MCD de dos enteros.**

Ejercicio. **Leed un valor desde el teclado, obligando a que sea un positivo. Devolved el valor leído.**

3.1.5. EJEMPLOS DE FUNCIONES MAL CODIFICADAS

```
int f (int n) {    /* MAL */
     return n+2;
     printf("Nunca se ejecuta \n"); /* MAL */
}

int Factorial (int n)  {
   int i, aux;

   aux = 1;
   for (i=2; i<=n; i++)  {
      aux = aux * i;
      return aux;     /* MAL */
   }
}

int EsPar (int n)  {
    if (n%2==0)
        return 0;   /* MAL */
}

int EsPrimo (int n){
   int i;

   for (i=2 ; i<=sqrt(n) ; i++)
      if (n%i == 0)
         return 0;   /* MAL */
   return 1;
}
```

> ***Consejo*: No introducid sentencias `return` en varios sitios distintos del cuerpo de la función. Dejadlo al final de la función**

Ejercicio. **Re-escribid la función `EsPrimo`.**

3.1.6. DOCUMENTACIÓN DE UNA FUNCIÓN

Hay dos tipos de comentarios:

- ***Descripción del algoritmo que implementa la función***
  - **Describe cómo se resuelve la tarea encomendada a la función.**
  - **Se incluye dentro del código de la función o delante**
  - **Sólo describimos la esencia del algoritmo.**

Ejemplo. **El mayor de tres números**

```
int Max3 (int a, int b, int c){
   int max;

   /* Calcular el máximo entre a y b          :-)
      Calcular el máximo entre max y c
   */

   if (a>=b)
      max = a;
   else
      max = b;
   if (c>max)
      max = c;

   return max;
}
```

Nunca parafrasearemos el código

```
int Max3 (int a, int b, int c){
   int max;

   /* Si a es >= b, asignamos max=a
      En otro caso, le asignamos b
      Una vez hecho lo anterior, vemos si
      c es mayor que max, en cuyo caso
      le asignamos c                          :-(
   */

   if (a>=b)
      max = a;
   else
      max = b;
   if (c>max)
      max = c;

   return max;
}
```

Usar descripciones de algoritmos que sean concisas, pero claras y completas, con una presentación visual agradable y esquemática.

▶ *Descripción de la cabecera de la función*

- ▷ **Describe qué tarea resuelve la función.**
- ▷ **También describe los parámetros (cuando no sea obvio).**
- ▷ **Se incluye fuera , justo antes de la cabecera de la función**

```
/*
   Identificador: Max3
   Tipo devuelto: int
   Cometido:      Calcula el máximo de tres valores
   Entradas:      Tres enteros a, b, c
   Salidas:       El máximo entre ellos.
*/
int Max3 (int a, int b, int c){
   int max;

   /* Calcular el máximo entre a y b
      Calcular el máximo entre max y c
   */

   if (a>=b)
      max = a;
   else
      max = b;
   if (c>max)
      max = c;

   return max;
}
```

Incluiremos:

- ▷ **Descripción de los datos de entrada, incluyendo las restricciones (o precondiciones) que deben satisfacer para obtener unos datos de salida correctos.**
- ▷ **Descripción breve del cometido de la función. Si no podemos resumirlo en un par de líneas, entonces la función es demasiado compleja (poca cohesión) y posiblemente debería dividirse en varias funciones.**
- ▷ **Descripción del valor devuelto, incluyendo las restricciones (o postcondiciones) que resultan de su aplicación.**
- ▷ **Indicación de las bibliotecas que necesite. Opcionalmente, se puede incluir un gráfico que lo ilustre.**
- ▷ **En una documentación externa (en papel o en un fichero), también se puede incluir un *diagrama de dependencias* (bibliotecas usadas)**

De esta forma, construimos *fichas* que identifican funciones, lo más independientemente posible del lenguaje de programación (C en nuestro caso).

Ampliación:

Consultad el capítulo 19 del libro *Code Complete* de Steve McConnell sobre normas para escribir comentarios claros y fáciles de mantener.

`http://www.geocities.com/aplicacionesjava/`

En el examen es imperativo incluir la descripción del algoritmo. La de la cabecera es algo adicional.

3.2. FUNCIONES VOID

3.2.1. MOTIVACIÓN Y DEFINICIÓN

Ejemplo. **Supongamos que en el `main` nos encontramos este trozo de código:**

```
int i, tope_lineas;

.....................

for (i=1; i<=tope_lineas ; i++)
    printf("\n************");
printf("Programa básico de Trigonometría");
for (i=1; i<=tope_lineas ; i++)
    printf("\n************");
.....................
```

¿No sería más fácil de entender si el código del programa principal hubiese sido el siguiente?

```
.....................
Presentacion(tope_lineas);
.....................
```

En este ejemplo, `Presentacion` resuelve la tarea de realizar la presentación del programa por pantalla, pero no calcula (devuelve) ningún valor, como por ejemplo las funciones `sqrt` o `Hipotenusa`. Por eso, su llamada constituye una sentencia y no aparece dentro de una expresión.

Este tipo particular de funciones que no devuelven ningún valor, se definen como sigue:

```
void <nombre-función> (<lista parámetros formales>) {
  [<Constantes Locales>]
  [<Variables Locales>]
  [<Sentencias>]
}
```

El paso de parámetros y la definición de datos locales sigue las mismas normas que el resto de funciones.

Observad que no hay sentencia `return`. La función `void` termina cuando se ejecuta la última sentencia de la función.

```
#include <stdio.h>
#include <math.h>

float Hipotenusa(float ladoA, float ladoB){
   return sqrt(ladoA*ladoA+ladoB*ladoB);
}

float LeePositivo(){
   float lee_positivo;

   do{
      scanf("%f", &lee_positivo);
   }while (lee_positivo<=0);

   return lee_positivo;
}

void Presentacion(int tope_lineas){
   int i;
   for (i=1; i<=tope_lineas ; i++)
      printf("\n**********");
   printf("Programa básico de Trigonometría");
   for (i=1; i<=tope_lineas ; i++)
      printf("\n**********");
}

void MostrarHipotenusa (float valor){
   printf("\nLa hipotenusa vale %f", valor);
}
```

```
int main(){
   float lado1, lado2, hip;

   Presentacion(3);

   lado1 = LeePositivo();
   lado2 = LeePositivo();

   hip = Hipotenusa(lado1,lado2);
   MostrarHipotenusa(hip);
}
```

Recordemos que el programador debe identificar primero las tareas y las funciones que las resuelven (incluidas las `void`***) antes de escribir una línea de código.***

Llamadas entre funciones `void`.

```
void ImprimirLineas (int num_lineas){
   int i;
   for (i=1; i<=num_lineas ; i++)
      printf("\n**********");
}
void Presentacion(int tope_lineas){
   ImprimirLineas (tope_lineas);
   printf("Programa básico de Trigonometría");
   ImprimirLineas (tope_lineas);
}
```

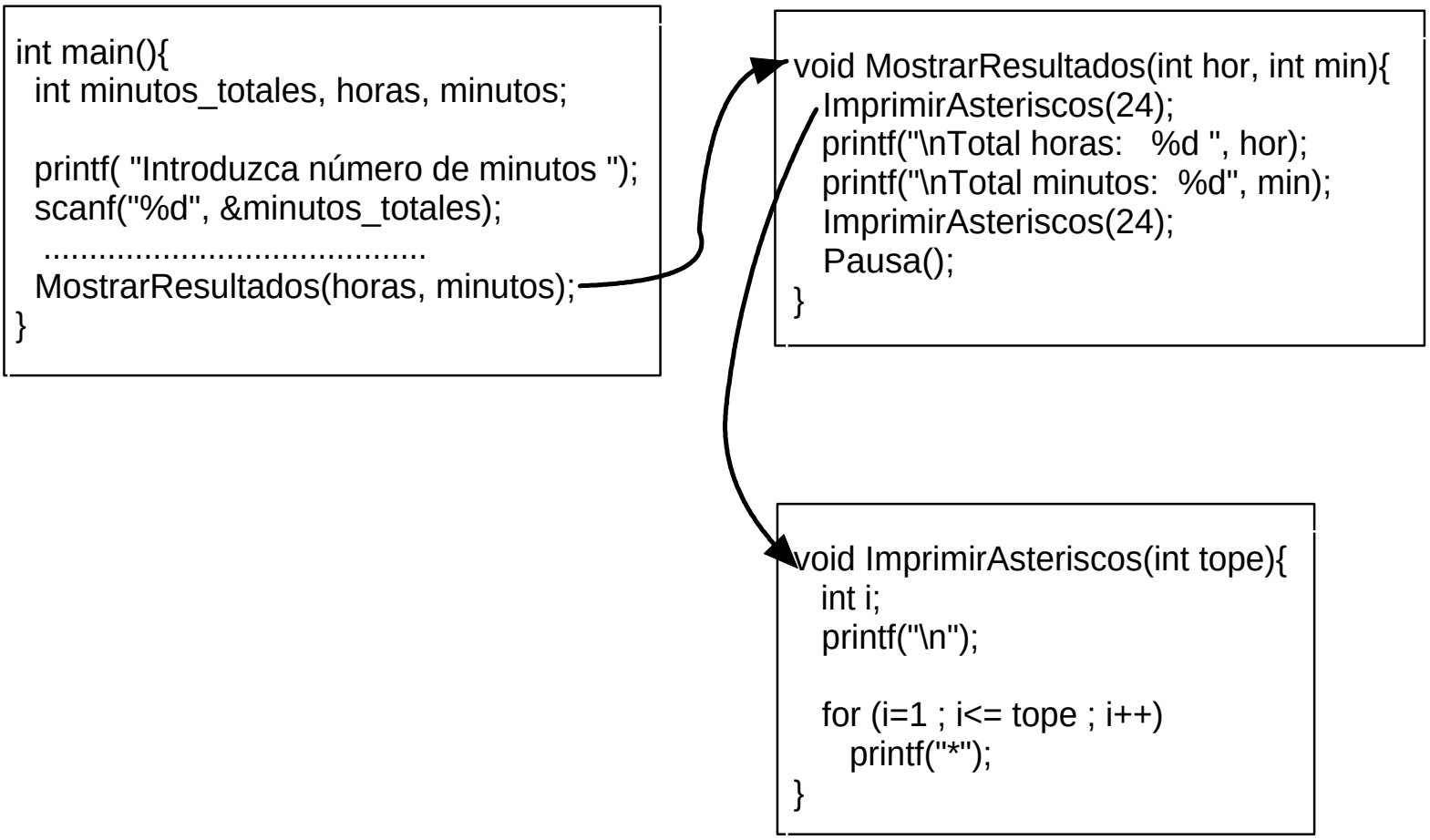
O incluso:

```
void ImprimirAsteriscos (int num_asteriscos){
   int i;
   for (i=1 ; i<=num_asteriscos ; i++)
      printf("*");
}
void ImprimirLineas (int num_lineas){
   int i;
   for (i=1; i<=num_lineas ; i++){
      printf("\n");
      ImprimirAsteriscos(10);
   }
}
void Presentacion(int tope_lineas){
   ImprimirLineas (tope_lineas);
   printf("Programa básico de Trigonometría");
   ImprimirLineas (tope_lineas);
}
```

Llamadas entre funciones `void`. Queremos calcular el cociente y el resto de la división entera entre dos valores.

```
#include <stdio.h>
#include <stdlib.h>

void Pausa(){
   printf("\n");
   system("pause");
}
void ImprimirAsteriscos (int num_asteriscos){
   int i;
   printf("\n");
   for (i=1 ; i<=num_asteriscos ; i++)
      printf("*");
}
void MostrarResultados (int hor, int min){
   ImprimirAsteriscos(24);
   printf("\nTotal horas:    %d", hor);
   printf("\nTotal minutos:  %d", min);
   ImprimirAsteriscos(24);
   Pausa();
}
int main(){
   int minutos_totales, horas, minutos;

   printf("Introduzca número de minutos ");
   scanf("%d", &minutos_totales);

   horas = minutos_totales / 60;
   minutos = minutos_totales % 60;
```

```
        MostrarResultados(horas, minutos);
    }
```



3.2.2. PASO DE PARÁMETROS POR REFERENCIA

3.2.2.1. MOTIVACIÓN

Supongamos que queremos encapsular el cálculo del cociente y del resto. Debemos usar sendas funciones:

```
#include <stdio.h>

int Cociente(int divdo, int dvsor){
   return divdo / dvsor;
}
int Resto(int divdo, int dvsor){
   return divdo % dvsor;
}
int main(){
   int minutos_totales, horas, minutos;

   printf("Introduzca número de minutos ");
   scanf("%d", &minutos_totales);

   horas = Cociente(minutos_totales, 60);
   minutos =  Resto(minutos_totales, 60);
   MostrarResultados(horas, minutos);
}
```

El problema es que parece lógico suponer que siempre que voy a calcular el cociente, también voy a necesitar calcular el resto. Por lo tanto, habrá siempre cierto *código repetido*:

```
horas = Cociente(minutos_totales, 60);
minutos =  Resto(minutos_totales, 60);
```

¿Cómo devolvemos dos valores simultáneamente?

- ▷ **Una función usual sólo puede devolver un único valor**
- ▷ **Nos preguntamos si lo podemos hacer usando un** `void`

```
#include <stdio.h>
.............................
/* MAL */
void CocienteRestoMAL(int divdo, int dvsor, int coc, int res){
   coc = divdo / dvsor;
   res = divdo % dvsor;
}

int main(){
   int minutos_totales, horas, minutos;

   printf("Introduzca número de minutos ");
   scanf("%d", &minutos_totales);

   CocienteRestoMAL(minutos_totales, 60, horas, minutos);
   MostrarResultados(horas, minutos);
}
```

Las sentencias:

```
coc = divdo / dvsor;
res = divdo % dvsor;
```

modifican las variables locales `coc` **y** `res` **de** `CocienteRestoMAL`**. Las variables** `horas` **y** `minutos` **no se modifican (se quedan con** `?`**).**

En definitiva:

- ▷ **La llamada a una función *normal* devuelve un único valor.**
- ▷ **La llamada a una función** `void` **no devuelve ningún valor.**

Sin embargo, los lenguajes de programación permiten otra forma de pasar parámetros, para que puedan modificarse varias variables, a la vez, desde dentro de una misma función.

3.2.2.2. DECLARACIÓN DE PARÁMETROS PASADOS POR REFERENCIA

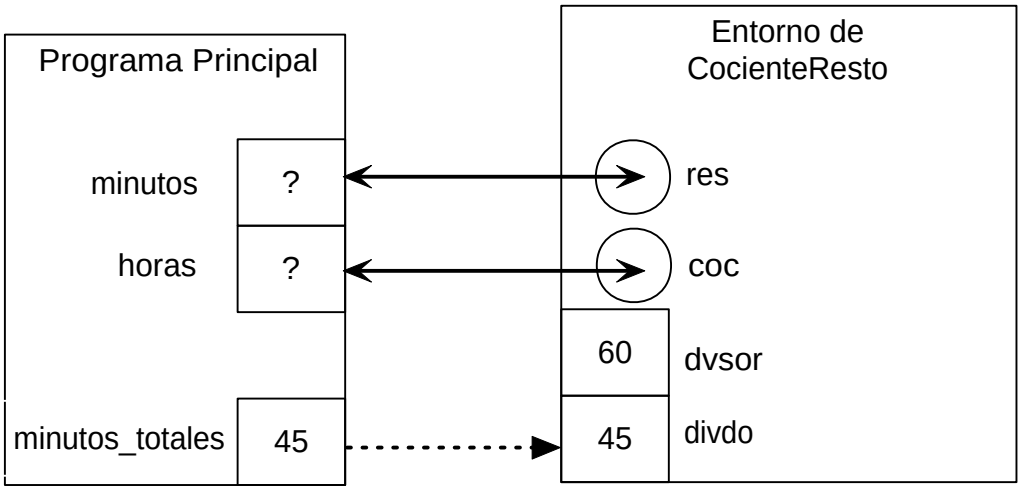
***Paso por referencia*. Es una forma de pasar parámetros distinta al paso por valor. Se indica poniendo un `*` en la declaración del parámetro formal.**

- **El parámetro formal estará ligado al parámetro actual.**

  Dentro de la función a través del parámetro formal, modificaremos el parámetro actual

- ***Dentro* del cuerpo de la función, el parámetro se trata precedido de `*` (es decir, `(*identificador)`).**
- **En la *llamada*, el parámetro se precede de `&` (es decir, `&identificador`). Esto es lo que venimos haciendo en la función scanf cada vez que la usamos.**
- **Obviamente, en una misma función puede haber parámetros pasados por valor y otros por referencia.**
- **Por ahora, usaremos los pasos por referencia sólo en las funciones `void` pero se puede usar en cualquier función.**

```
#include <stdio.h>
.............................
/* Bien */
void CocienteResto(int divdo, int dvsor, int *coc, int *res){
   *coc = divdo / dvsor;
   *res = divdo % dvsor;
}

int main(){
   int minutos_totales, horas, minutos;

   printf("Introduzca número de minutos ");
   scanf("%d", &minutos_totales);

   /*
      Antes de la llamada a CocienteResto
      las variables horas y minutos contienen ?
   */

   CocienteResto(minutos_totales, 60, &horas, &minutos);
   MostrarResultados(horas, minutos);
}
```


Programa Principal
minutos
?
horas
?
minutos_totales
45
Entorno de
CocienteResto
res
coc
60
dvsor
45
divdo

Pila

Algunas cuestiones:

- **Viendo la llamada a una función podemos saber cuales son los parámetros pasados por referencia. Las precedidas por** `&`

```
CocienteResto(dividendo,divisor,&cociente,&resto);
                    /* ¿Por valor?  ¿Por referencia? */
```

- **Las constantes, literales y expresiones no pueden ser parámetros actuales pasados por referencia. Deben pasarse por valor.**

```
CocienteResto(dividendo,divisor,cociente,resto+1);
                      /* Llamada Incorrecta */
```

- **Procurad agrupar primero los parámetros formales por valor. Es lógico: primero indicamos lo que la función necesita, y luego lo que modifica.**

```
void CocienteResto(int divdo, int dvsor,
                   int *coc, int *res)   /* :-) */

void CocienteResto(int *coc, int *res,
                   int divdo, int dvsor) /* :-( */
```

Ejercicio.

Cread una función `void` que encapsule el algoritmo del cálculo de las raíces de una parábola

$$ax^2 + bx + c = 0$$

En el caso de que no existan dos raíces reales, en un parámetro llamado `hay_dos_raices_reales` se pondrá el valor `false`.

Hacedlo primero con tres funciones y luego con un único `void`.

Ejercicio. **Cread una función `void` para leer dos datos desde teclado, obligando a que ambos sean positivos. Usad la función `LeePositivo`:**

```
float LeePositivo(){
   float lee_positivo;

   do{
      scanf("%f", &lee_positivo);
   }while (lee_positivo<=0);

   return lee_positivo;
}
```

3.2.2.3. FUNCIONES VERSUS FUNCIONES VOID CON UN PARÁMETRO POR REFERENCIA

```
int Factorial (int n){
    int i;
    int aux = 1;

    for (i=2; i<=n; i++)
        aux = aux * i;
    return aux;
}
```

Llamada:

```
variable = Factorial(4);
printf("Factorial de 4 = %d", variable);
```

Codifiquemos la función `Factorial` como una función `void`. Debemos guardar el resultado en un paso por referencia.

```
void Factorial (int n, int *resultado){
    int i;
    int fact = 1;

    for (i=2; i<=n; i++)
        fact = fact* i;
    *resultado = fact;
}
```

Incluso podemos suprimir la variable `fact`:

```
void Factorial (int n, int *resultado){
    int i;

    *resultado = 1;
    for (i=2; i<=n; i++)
        *resultado = (*resultado) * i;
}
```

Llamada:

```
Factorial (4, &variable);
printf("Factorial de 4 = %d", variable);
```

Si se quiere calcular un único valor, es mejor usar una función que lo devuelva que una función `void` con un paso por referencia:

- **Supongamos que queremos modificar una variable con su factorial:**
  - **Con una función:**

    ```
    variable = Factorial (variable);
    ```

  - **Con una función `void`:**

    ```
    Factorial (variable, &variable); /* Algo raro! */
    ```

- **Otra desventaja: no podríamos usar `Factorial` dentro de una expresión.**

Ejemplo. **Construid una función para escribir en pantalla un menú, y elegir una opción del usuario.**

▶ ***Con una función que devuelve un*** `char`

```
#include <stdio.h>

char Menu(){
   char tecla;

   printf("\nElija una opción\n");
   printf("\nS. Sumar");
   printf("\nR. Restar");
   printf("\nM. Media Aritmética\n");
   scanf("%c", &tecla);

   return tecla;
}

int main(){
   char opcion;

   opcion = Menu();
   switch (opcion)
      ......................
}
```

▶ *Con una función* `void`

```
#include <stdio.h>

void Menu(char *tecla){
   printf("\nElija una opción\n");
   printf("\nS. Sumar");
   printf("\nR. Restar");
   printf("\nM. Media Aritmética\n");
   scanf("%c", tecla);   /* No usamos &  pues tecla */
}                        /* ya es un puntero */

int main(){
   char opcion;

   Menu(&opcion);
   switch (opcion)
      ......................
}
```

3.2.2.4. OTRAS CUESTIONES SOBRE EL PASO POR REFERENCIA

Pasamos por referencia los datos que vamos a modificar. Pero obviamente, si tienen algún valor previamente establecido, podemos acceder a él.

Ejemplo. **Escribir una función `void` para incrementar en 1 una variable entera.**

```
void Incrementa (int *valor){
   *valor = *valor+1;
}

int main(){
   int dato;

   dato = 7;
   Incrementa(&dato);
   printf("\ndato = %d", dato);  /* <- Imprime 8 */
}
```

Ejercicio. **Construid una función para intercambiar el valor de dos variables de tipo `float`.**

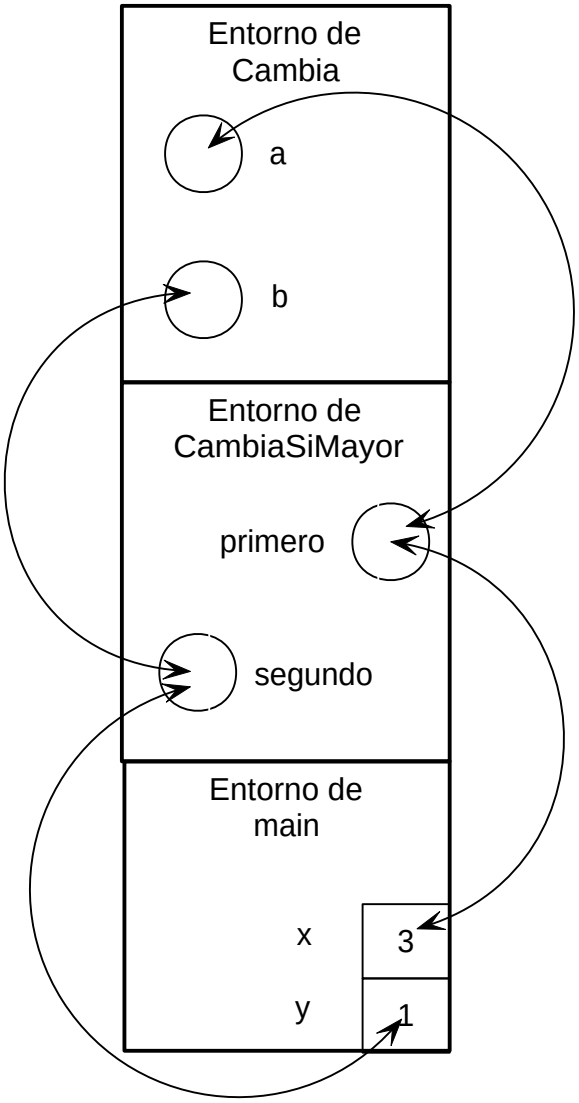
De hecho, los pasos por valor no serían *necesarios*.
Pero son imprescindibles si no queremos modificar por accidente el parámetro actual.
¿Qué pasaría en el siguiente ejemplo?

```
/* Calcula la suma de los enteros entre 1 y tope */
void CalcularSumaMAL(int *tope, int *suma){
   *suma = 0;
   while ((*tope)>0){
      *suma = *suma + *tope;
      (*tope)--;
   }
}
```

Una función con un parámetro `par` pasado por referencia, puede llamar a una segunda función y pasar por referencia a `par` (obviamente, también por valor)

Ejemplo. **Construir una función `void` que acepte por referencia dos enteros. Si el primero es mayor que el segundo debe intercambiarlos. En caso contrario, debe dejarlos igual.**

```
#include <stdio.h>

void Cambia(float *a, float *b){
   float aux;

   aux = *a;
   *a = *b;
   *b = aux;
}

void CambiaSiMayor (float *primero, float *segundo){
   if (*primero > *segundo)
      Cambia (primero,segundo);
}

int main() {
   float x=3, y=1;

   CambiaSiMayor(&x,&y);
   printf("%f %f", x, y);
}
```


Entorno de
Cambia
a
b
Entorno de
CambiaSiMayor
primero
segundo
Entorno de
main
x
3
y
1

3.3. DISEÑO DE FUNCIONES

3.3.1. CONCEPTOS BÁSICOS

3.3.1.1. DATOS DE ENTRADA Y DE SALIDA

Resumen de lo visto hasta ahora:

- **Los parámetros por valor son datos que necesita conocer una función (son *entradas*).**
- **El valor devuelto por una función, es un dato calculado por ella: es una *salida*.**
- **Los pasados por referencia (usados normalmente en los `void`) corresponden a datos modificados por la función (son *salidas*).**
  Recordemos que en algunos casos, también se usaba dentro de la función el valor que tuviese parámetro por referencia (`void Incrementa(float *valor)`)
  En estos casos, el parámetro por referencia sería de entrada y salida.

Parámetros pasados por valor	$\rightarrow$	**Datos de entrada**
Parámetros pasados por referencia	$\rightarrow$	**Datos de salida**
		Datos de e/s

Datos de Entrada $\longrightarrow$ **Función** $\longrightarrow$ **1 dato de Salida**

Datos de Entrada $\longrightarrow$ **Función** `void` $\longrightarrow$ **0, 2 o más salidas**

***Nota*. No confundir datos de entrada (resp. salida) con entrada de datos -`scanf`- (resp. salida de resultados -`printf`-).**

- ***Función para calcular la media aritmética:***
  - **Datos de entrada: Un conjunto de N valores numéricos.**
  - **Datos de salida: Un real con la media aritmética.**
- ***Función para dibujar un cuadrado:***
  - **Datos de entrada: longitud del lado (L), coordenadas iniciales (X,Y)**
  - **Datos de salida: Ninguno**
- ***Función para calcular la desviación típica:***
  - **Datos de entrada: Un conjunto de N valores numéricos.**
  - **Datos de salida: Un real con la desviación típica.**

Los datos de salida de unas funciones son datos de entrada para otras:

```
int dividendo, divisor, cociente, resto;

LecturaValores(&dividendo,&divisor);
CocienteResto(dividendo,divisor,&cociente,&resto);
MostrarResultados(cociente,resto);
```

3.3.1.2. COHESIÓN Y ACOPLAMIENTO

***Cohesión*: Es una medida del grado de identificación de una función con una tarea concreta.**

Cuanto más específica sea la tarea de una función, mayor será su cohesión. Por ejemplo, una función que haga cinco tareas distintas tendrá menor cohesión que si sólo realiza una. De hecho, una función sólo debería realizar una tarea.

Es una medida de la relación de los elementos dentro de una función.

***Acoplamiento*: Es una medida del grado de interacción o dependencia entre las funciones de un programa.**

Por ejemplo, si una función llama a otras 10 funciones, tendrá mayor acoplamiento que si llama a 2. Esto hará que su mantenimiento y posible actualización futura sea más compleja.

Es una medida de la relación que hay entre las funciones.

Objetivo: Conseguir una cohesión alta y un bajo acoplamiento.
Ambos objetivos son incompatibles, por lo que siempre habrá que buscar un equilibrio satisfactorio.

Nota. En general, se habla del grado de cohesión y acoplamiento de módulos software (usualmente funciones y clases)

Veamos patrones de diseño de funciones para conseguir una alta cohesión y un bajo acoplamiento.

Nota:

Se utilizan los términos:

- ***Cohesión fuerte*** **= Cohesión alta** `:-)`
- ***Cohesión débil*** **= Cohesión baja** `:-(`
- ***Acoplamiento fuerte*** **= Acoplamiento alto** `:-(`
- ***Acoplamiento débil*** **= Acoplamiento bajo** `:-)`

3.3.2. PATRONES DE DISEÑO DE FUNCIONES

3.3.2.1. LAS FUNCIONES COMO TRABAJADORES DE UNA EMPRESA

Las funciones resuelven tareas específicas.
Como en la vida real, el trabajador que resuelve una tarea en una empresa, debe hacerlo de la forma más autónoma posible.

- **No le dirá al director los detalles de cómo ha resuelto la tarea**
- **No pedirá al director más datos ni acciones iniciales de los estrictamente necesarios (el trabajador ya es *mayorcito*)**
- **Lo hará de forma que se pueda reutilizar su esfuerzo desde otras secciones o compañías (puede que nos interese cambiar de empresa y llevarnos nuestra experiencia)**
- **Debe comprobar que ha realizado correctamente la tarea (el director está ocupado en otras cosas)**
  Si ha pasado algo, lo comunicará por los cauces establecidos (y no con una pancarta)
- **No podrá interferir en el desarrollo de otras tareas (el trabajador es una pieza fundamental el engranaje, pero no puede poner zancadillas a los compañeros)**

Con ello, obtenemos funciones autónomas (*cajas negras*)

Veamos cómo conseguirlo.

3.3.2.2. EL DISEÑO DEL PROGRAMA PRINCIPAL

El programa principal tendrá bastantes líneas de código. Las variables *importantes* del programa deben estar declaradas como variables de `main` (al menos).

```
float HazloTodo(){
   float final;
   int dato1, dato2;
   char opcion;

   LeerDatos(&dato1, &dato2);
   opcion = Menu();
   final = Computos(opcion, dato1, dato2);

   return final;
}
int main(){
   float  final;

   final = HazloTodo();    /* :-( */
   SalidaPantalla(final);
}
```

La función `HazloTodo` tiene una cohesión muy baja, ya que hace muchas cosas distintas `:-((`. La función `HazloTodo` no se podrá reutilizar en ningún otro programa.

En general, cuantas más tareas haga una función, más difícil será reutilizarla en otros programas.

Otro ejemplo: Cálculo de las raíces de una ecuación de segundo grado.

```
void HazloTodo (float primer_coef, float segundo_coef,
                float tercer_coef){
   int NumeroRaices;
   float raiz1, raiz2;

   Ec_2_grado(primer_coef,segundo_coef,tercer_coef,
              &raiz1,&raiz2,&NumeroRaices);
   Escribe_Dist(raiz1,raiz2,NumeroRaices);
}

int main(){
   float coef1, coef2, coef3;

   Leer_Parabola(&coef1, &coef2, &coef3);
   HazloTodo(coef1, coef2, coef3);        /* :-( */
}
```

Podemos pensar en la función `main` como el director de una empresa. El director no es un vago. Se encarga de dirigir a sus empleados, que no es poco.

Nunca han de construirse funciones *monolíticas* que realizan muchas tareas y por tanto con baja cohesión

3.3.2.3. EVITAR PARÁMETROS INNECESARIOS

Hay que diseñar una función pensando en lo que estrictamente necesita, evitando pasar parámetros innecesarios.

```
int Factorial (int n){  /* :-) */
   int fact=1;
   int i;

   for (i=2 ; i<=n ; i++)
      fact = fact*i;

   return fact;
}
int main(){
   printf("\nFactorial de 3 = %d", Factorial(3));  /* :-) */
}
```

```
int FactorialMAL (int i, int n){  /* :-( */
   int fact=1;

   while (i<=n){
      fact = fact*i;
      i++;
   }
   return fact;
}
int main(){
  printf("\nFactorial de 3 = %d", FactorialMAL(2,3)); /* :-( */
}
```

```
char Mi_Tolower (char caracter, int amplitud){  /* :-( */
   char minuscula;

   minuscula = caracter + amplitud;
   return minuscula;
}
```

Esta primera versión de `Mi_Tolower` **necesita un parámetro adicional innecesariamente. Al depender de más elementos, aumenta su acoplamiento** `:-(`

```
char Mi_Tolower (char caracter){  /* :-) */
   char minuscula;
   const int amplitud = 'a'-'A';

   minuscula = caracter + amplitud;
   return minuscula;
}
```

Evitar pasar parámetros innecesarios, disminuyendo así el acoplamiento

3.3.2.4. SEPARAR E/C/S

Nunca mezclar E/C/S en una misma función. Es decir, diseñaremos las funciones de tal forma que si una función realiza ciertos cómputos, no realizará otras tareas como E/S (`scanf`/`printf`)

Cuantas menos tareas haga una función, mayor cohesión `:-)`

```
void ImprimirFactorialMAL(int n){   /* :-( */
   int fact=1;
   int i;

   for (i=2 ; i<=n ; i++)
      fact = fact*i;

   printf("\nEl factorial de %d es %d", n, fact);
}

int main(){
   ImprimirFactorialMAL(3);
}
```

```
int Factorial (int n){     /* :-) */
   int fact=1;
   int i;

   for (i=2 ; i<=n ; i++)
      fact = fact*i;

   return fact;
}

void ImprimirFactorial(int n){
   printf("\nEl factorial de %d es %d", n, Factorial(n));
}

int main(){
   ImprimirFactorial(3);
}
```

Obviamente, `ImprimirFactorial` **calcula el factorial y lo imprime en pantalla. Pero lo bueno es que hemos conseguido aislar los cómputos del factorial en una función, para que pueda ser usada en otros programas que no quieran escribir el resultado con** `printf`**.**

Una función de cálculo NUNCA debe imprimir mensajes. Hacedlo, en su caso, en la función que la llama. El incumplimiento de esta norma garantiza el Suspenso automático.

***Nota*. Lo mismo ocurre con las entradas de datos** `scanf`

3.3.2.5. QUE LA LLAMADA A LA FUNCIÓN NO NECESITE HACER SIEMPRE LAS MISMAS ACCIONES PREVIAS

Ejemplo. **Construid una función para calcular la potencia de dos reales. Si se pasan ambos a cero, la función debe devolver** `1` **en una variable llamada** `Indet`**.**

```
.............

void PotenciaMAL(float base, float exponente,
                 float *resultado, int *Indet){
   if ((base==0) && (exponente == 0))
      *Indet = 1;
   else
      *resultado = pow(base,exponente);
}
int main(){
   float dato1, dato2, pot;
   int Indeterminacion;

   <Lectura de dato1 y dato2>

   Indeterminacion = 0;   /* <- Inicialización necesaria
                             antes de la llamada. :-( */
   PotenciaMAL(dato1, dato2, &pot, &Indeterminacion);
   if (Indeterminacion)
      printf("\n0^0 = Indeterminación");
   else
      printf("\n%f^%f = %f", dato1, dato2, pot);
}
```

En esta primera versión, la llamada a `PotenciaMAL` requiere que siempre ejecutemos previamente `Indeterminacion = 0;`
Problema: Alguna vez se nos olvidará! $\Rightarrow$ Propenso a errores.

```
void Potencia (float base, float exponente,
               float *resultado, int *Indet){
   if ((base==0) && (exponente == 0))
      *Indet = 1;
   else{
      *resultado = pow(base,exponente);
      *Indet = 0;
   }
}
```

O si se prefiere:

```
void Potencia (float base, float exponente,
               float *resultado, int *Indet){

   *Indet = (base==0) && (exponente == 0);

   if (!(*Indet))
      *resultado = pow(base,exponente);
}
```

```
int main(){
  float dato1, dato2, pot;
  int Indeterminacion;

  <Lectura de dato1 y dato2>

  Potencia(dato1, dato2, &pot, &Indeterminacion);  /* :-) */

  if (Indeterminacion)
      printf("\n0^0 = Indeterminación");
  else
      printf("\n%f^%f = %f", dato1, dato2, pot);
}
```

Las inicializaciones que requiera una función, de ben hacerse dentro de ella.
Esto disminuye el acoplamiento

3.3.2.6. VALIDACIÓN DE DATOS DEVUELTOS

Ejemplo. **Menú de operaciones. Hay que comprobar que la opción leída sea correcta.**

▶ ***Primera versión***

```
#include <stdio.h>
#include <ctype.h>

void SalidaPantalla (float final){
   printf("\n\nResultado = %f", final);
}

char MenuMAL(){
   char tecla;

   printf("\nElija una opción\n");
   printf("\nS. Sumar");
   printf("\nR. Restar");
   printf("\nM. Media Aritmética\n");
   scanf("%c", &tecla);

   return tecla;
}

int main(){
   int    dato1=3, dato2=5;
   float resultado;
   char   opcion;
```

```
   do{                                  /* :-( */
      opcion = MenuMAL();
      opcion = toupper(opcion);
   }while ((opcion!='S') && (opcion!='R') && (opcion!='M'));

   switch (opcion){
      case 'S': resultado = dato1+dato2;
                break;
      case 'R': resultado = dato1-dato2;
                break;
      case 'M': resultado = (dato1+dato2)/2;
                break;
   }

   SalidaPantalla(resultado);
}
```

▶ *Segunda versión*

```
#include <stdio.h>
#include <ctype.h>

void SalidaPantalla (float final){
   printf("\n\nResultado = %f", final);
}

char Menu (){
   char tecla;

   printf("\nElija una opción\n");
   printf("\nS. Sumar");
   printf("\nR. Restar");
   printf("\nM. Media Aritmética\n");

   do{                                  /* :-) */
      scanf("%c", &tecla);
      tecla = toupper(tecla);
   }while ((tecla!='S') && (tecla!='R') && (tecla!='M'));

   return tecla;
}

int main(){
   int    dato1=3, dato2=5;
   float resultado;
   char   opcion;

   opcion = Menu();
```

```
   switch (opcion){
      case 'S': resultado = dato1+dato2;
                break;
      case 'R': resultado = dato1-dato2;
                break;
      case 'M': resultado = dato1*1.0/dato2;
                break;
   }

   SalidaPantalla(resultado);
}
```

Este es un ejemplo de compromiso cohesión / acoplamiento

- ▷ **Dentro de la función, realizamos dos tareas: leer un valor y comprobar que es correcto. Disminuye la cohesión** `:-(`
- ▷ **En el** `main` **simplemente llamamos a la función, sin necesidad de comprobar sistemáticamente la validez del dato. Disminuye el acoplamiento** `:-)`

En este caso optaremos por pagar el precio de disminuir la cohesión.

Procurad que las funciones comprueben
la validez de los datos devueltos
Esto disminuye el acoplamiento

3.3.2.7. EVITAR EFECTOS COLATERALES

La mayoría de los lenguajes de programación (incluido C) permiten definir variables fuera de las funciones. Automáticamente, su ámbito incluye todos las funciones definidos después. Por eso, se conocen con el nombre de *variables globales*.

El uso de variables globales permite que las funciones se puedan comunicar a través de ellas y no de los parámetros. Esto es pernicioso para la programación estructurada, fomentando la aparición de *efectos laterales*.

Ejemplo. **Supongamos un programa para la gestión de un aeropuerto. Tendrá dos funciones:** `GestionMaletas` **y** `ControlVuelos`**. El primero controla las cintas transportadoras y como máximo puede gestionar 50 maletas. El segundo controla los vuelos en el área del aeropuerto y como máximo puede gestionar 30. El problema es que ambos van a usar el mismo dato global** `Max` **para representar dichos máximos.**

▶ *Primera Versión*

```
#include <stdio.h>

int Max;                /* Variables globales */
int saturacion;

void GestionMaletas(){
   Max = 50;            /* !Efecto lateral! */
   if (NumMaletasActual <= Max)
      [acciones maletas]
   else
      ActivarEmergenciaMaletas();

void ControlVuelos(){
   if (NumVuelosActual <= Max)
      [acciones vuelos]
   else{
      ActivarEmergenciaVuelos();
      saturacion = 1;
   }
}
int main(){
   Max = 30;    /* Máximo número de vuelos */
   saturacion = 0;

   while (!saturacion){
      GestionMaletas();   /* Efecto lateral: Max = 50 */
      ControlVuelos();
   }
```

▶ *Segunda Versión*

```
#include <stdio.h>

void GestionMaletas(int Max){
   if (NumMaletasActual <= Max)
      [acciones maletas]
   else
      ActivarEmergenciaMaletas();
}

int ControlVuelos(int Max){
   int saturacion=0;

   if (NumVuelosActual <= Max)
      [acciones vuelos]
   else{
      ActivarEmergenciaVuelos();
      saturacion = 1;
   }
   return saturacion;
}

int main(){
   int saturacion = 0;

   while (!saturacion){
      GestionMaletas(50);              /* :-) */
      saturacion = ControlVuelos(30);
   }
```

El uso de variables globales, hace que cualquier función las pueda usar y/o modificar, lo que provoca efectos laterales. Este sería uno de los ejemplos más graves de alto acoplamiento (“*acoplamiento externo*”)

***Nota*. Usualmente, se permite el uso de constantes globales , ya que si no pueden modificarse, no se producen efectos laterales**

***Ejemplo.* Supongamos un programa de trigonometría, con muchas funciones que necesitan usar el valor de π. Soluciones:**

1. **Se declara la constante `Pi` local a cada función.**

```
int FuncionTrigonometria(int parametro){
  const float Pi = 3.1416;
  .......
}
```

   Problema: Se repite la declaración de la constante en muchas funciones.

2. **Se pasa como parámetro:**

```
int FuncionTrigonometria(float Pi, int parametro){
   .........
}
int main(){
   const float Pi_4_dec = 3.1416;
   .....
   valor = FuncionTrigonometria(Pi_4_dec, dato)
}
```

   Problema: Hay que añadir el parámetro `Pi` a todas las funciones trigonométricas.

3. Se declara `Pi` como constante global y se usa en todas las funciones

```
#include <stdio.h>

const float Pi = 3.1416;   /* Cte "universal" */

int FuncionTrigonometria(int parametro){
   .....  /* <-- Podemos usar la cte global Pi */
}

int main(){
   valor = FuncionTrigonometria(dato)
}
```

Nos *ahorramos* un parámetro.

En este ejemplo, al trabajar con una constante *universal* como es π, podría justificarse el uso de constantes globales. Pero en otros casos, no está tan claro:

```
int SalarioNeto (int Retencion, int SalarioBruto){
   .........
}

int main(){
   const float IRPF = 0.18;  /* Cte "no universal" */
   int Sueldo, SalBruto;

   Sueldo = SalarioNeto(IRPF, SalBruto);
   ......................
}
```

```
const float IRPF = 0.18     /* Cte "universal" ? */

int SalarioNeto (int SalarioBruto){
   .........
}

int main(){
   int Sueldo, SalBruto;

   Sueldo = SalarioNeto(SalBruto);
   ......................
}
```

- ▷ ***Desventajas*** **en el uso de constantes globales:**
  - **– La cabecera de la función no contiene todas las entradas necesarias.**
  - **– No es posible usar la función en otro programa, a no ser que también se defina la constante en el nuevo programa (por eso tiene un alto acoplamiento)**
- ▷ ***Ventajas*****:**
  - **– Las cabeceras de las funciones que usan constantes se simplifican.**

Encontrar la solución apropiada a cada problema no es sencillo.

3.3.3. MEJORANDO LA CONSTRUCCIÓN DE FUNCIONES

3.3.3.1. GESTIÓN DE ERRORES

A veces no es posible que una función consiga realizar su tarea correctamente. Por ejemplo, una función que lea datos de un fichero e intenta acceder a un directorio no existente.

```
float LeerDatoFicheroMAL(){
   ................
   if (error)
      printf("\nSe produjo un error");  /* C/S */
   ................
}

int main(){
   float valor;
   ................
   valor = LeerDatoFicheroMAL();

   /* ¿Que hago ahora? */
}
```

Se incluirá un paso por referencia que indique lo sucedido:

- ▷ `bool` **si sólo hay que distinguir dos posibles situaciones**
- ▷ `char, int` **u otro tipo cuando hay más errores posibles.**

```
void LeerDatoFichero(float *dato, int *error){...}

int main(){
   int ErrorAcceso;
   float valor;
   ................
   LeerDatoFichero(&valor, &ErrorAcceso);

   if (ErrorAcceso)
      printf("\nError de acceso al fichero");
   else
      [Operaciones]
}
```

```
void LeerDatoFichero(float *dato, char *error){....}

int main(){
   char ErrorAcceso;
   float valor;
   ................
   LeerDatoFichero(&valor, &ErrorAcceso);

   switch (ErrorAcceso){
      case '-':  [Operaciones]
                 break;
      case 'E':  printf("\nDisco protegido contra escritura");
                 break;
      case 'F':  printf("\nFichero no existente");
                 break;
      ................
   }
}
```

▷ **Para identificar más de dos errores, es mucho mejor usar un tipo *enumerado***

```
enum TipoError {ninguno, proteccion_escritura,
                fichero_no_existe};
void LeerDatoFichero(float *dato, enum TipoError *error){....}

int main(){
   enum TipoError ErrorAcceso;
   float valor;
   ................
   LeerDatoFichero(&valor, &ErrorAcceso);

   switch (ErrorAcceso){
      case ninguno:
         [Operaciones]
         break;
      case proteccion_escritura:
          printf("\nDisco protegido contra escritura");
         break;
      case fichero_no_existe:
         printf("\nFichero no existente");
         break;
```

Dentro de una función de cómputos, el control de errores NUNCA se hará a través de un mensaje en pantalla. Se usará una variable de control (error) La violación de esta norma implica el Suspenso automático

3.3.3.2. FOMENTAR LA REUTILIZACIÓN EN OTROS PROBLEMAS

Supongamos que queremos calcular la distancia entre las raíces reales de una ecuación de segundo grado.

- ***Primera modularización***

```
Ec_2_grado(float coef1, float coef2, float coef3,
           float *distancia, int *NumRaices)
```

dónde `NumRaices` **representa el número de raíces reales. Esta función será de poca utilidad (*reusabilidad*) en otras aplicaciones trigonométricas, ya que no calcula las raíces sino la distancia.**

- ***Segunda modularización***

```
Ec_2_grado(float coef1, float coef2, float coef3,
           float *raiz1, float *raiz2, int *NumRaices)
```

Y en el programa principal calculamos la distancia entre las raíces.

> **Procurad diseñar una función pensando en su posterior reutilización en otros problemas**

Resumen

Normas para diseñar una función:

- ▷ **La cabecera debe contener exclusivamente los datos de E/S que sean imprescindibles para la función. El resto deben definirse como datos locales**
  → *ocultamiento de información*.
- ▷ **Debe ser lo más reutilizable posible:**
  - **No deben mezclarse E/C/S en una misma función.**
  - **La cabecera debe diseñarse pensando en la posible aplicación a otros problemas.**
- ▷ **Debe minimizarse la cantidad de información o acciones a realizar antes de su llamada, para su correcto funcionamiento.**
- ▷ **Debe validar los datos devueltos.**
- ▷ **Si pueden producirse errores, debe informarle a la función que la llame a través de un parámetro por referencia llamado, por ejemplo,** `error`
- ▷ **No debe realizar efectos laterales.**
  La comunicación entre funciones se realiza obligatoriamente a través de los parámetros.

3.3.4. CUESTIONES ESPECÍFICAS DE C

3.3.4.1. FUNCIONES: SENTENCIAS Y EXPRESIONES

Hemos dicho que la llamada a una función no constituye una sentencia de un programa. Sin embargo esto no es cierto. En C, si una función se llama sin usar el valor devuelto, éste se pierde y no pasa nada. Esto es una consecuencia del hecho de que cualquier expresión puede constituir una sentencia en C.

```
#include <stdio.h>

float Media (float x1, float x2) {
   return (x1+x2)/2.0;
}

int main()  {
   float dato1=3.0, dato2=4.0, resultado;

   Media(dato1,dato2);  /* Sintácticamente correcto en C
                           pero se pierde el valor devuelto */

   printf("\nMedia entre %f y %f = %f",
                  dato1, dato2, resultado);
                        /* Imprime basura */
}
```

¿En qué situaciones es útil? A veces podemos estar interesados en construir una función para realizar cierta tarea y sólo en determinadas ocasiones que haga *algo más*. Por ejemplo, en la biblioteca `<stdio.h>`, tenemos la función `printf` que veni-

mos utilizando

```
#include <stdio.h>

int main(){
   printf("Hola");
}
```

Pero también devuelve un valor, que es el número de caracteres impresos:

```
#include <stdio.h>

int main(){
   int total;

   total = printf("Hola ");
   printf("Total caracteres impresos: %d", total);
}
```

3.3.4.2. OTRAS CUESTIONES SOBRE FUNCIONES

- **Es posible llamar a una función en la inicialización de una variable:**

  ```
  int resultado = Factorial(3);
  ```

- **Es posible definir la función `main` con parámetros int argc y char *argv[].**
- **Una función `void` puede incluir una sentencia `return;` (sin expresión), para salir de la ejecución de la función en ese momento.**

- **En algunos lenguajes de programación, a las funciones `void` se les denomina *procedimientos*.**
- **En ocasiones, usaremos el término *módulo* como sinónimo de función (ya sea un `void` o devuelva un valor)**

3.4. MODULARIZACIÓN

3.4.1. PROTOTIPOS

Podemos decirle al compilador cual es la cabecera o *prototipo* de una función y definirla posteriormente.

```
#include<stdio.h>

/* Prototipos: */
void CambiaSiMayor(float *primero, float *segundo);
void Cambia(float *a, float *b);

int main() {
   float x=3, y=1;

   CambiaSiMayor(&x,&y);
   printf("%f %f", x, y);
}
void CambiaSiMayor (float *primero, float *segundo){
   if (*primero > *segundo)
      Cambia (primero,segundo);
}
void Cambia(float *a, float *b){
   float aux;
   aux = *a;
   *a = *b;
   *b = aux;
}
```

Nota:

De hecho, lo único que un `#include` hace es incluir ficheros de texto que contienen (entre otras cosas) los prototipos (o *cabeceras*) de las funciones. La definición de dichas funciones se incluye en otros ficheros, normalmente compilados (ficheros binarios) que se enlazan con el principal. Es por eso que a los ficheros que se incluyen con `#include`, se les llama ficheros de *cabeceras*.

► *Ventajas e inconvenientes al usar prototipos*

- ▷ **Permite centrarnos en el uso de la función y no en cómo está implementada `:-)`**
- ▷ **Tenemos dos cabeceras idénticas: la del prototipo y la de la definición. Aunque es código repetido, no es propenso a errores ya que el compilador genera un fallo si ambas cabeceras no son iguales.**

  Pero es muy tedioso de mantener `:-(`
- ▷ **Informa al compilador el tipo de la función y ya no la compilará como el tipo por defecto int. Cuando se usa antes de la definición.**

En casi todos los compiladores podremos crear el programa

```
#include <stdio.h>

void CambiaSiMayor(float *primero, float *segundo);
void Cambia(float *a, float *b);

int main() {
   float x=3, y=1;

   CambiaSiMayor(&x,&y);
   printf("%f %f", x, y);
}
```

y lo compilaremos para comprobar que las llamadas son correctas. Obviamente no se puede ejecutar porque falta la definición de las funciones.

?Dónde se incluye la definición de las funciones?

- **En el mismo fichero, después o antes del `main`.**
- **En otro fichero de texto, que habrá que agregarlo al proyecto.**
- **En otro fichero *compilado*, que habrá que agregarlo al proyecto.**

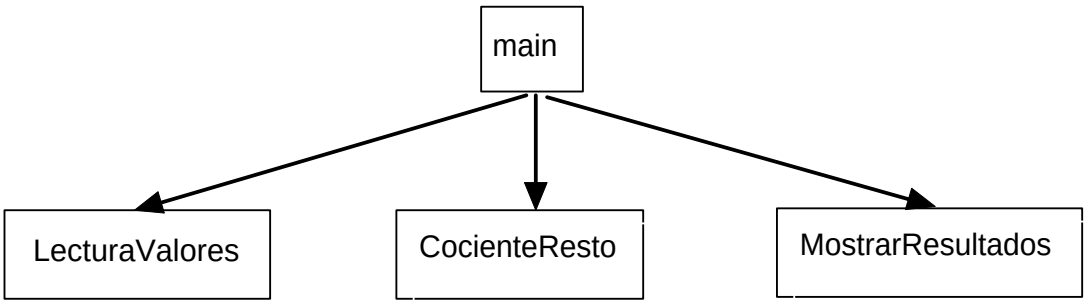
El proceso por el cual se *junta* la definición de las funciones con el `main` se denomina *enlace* (*link*)

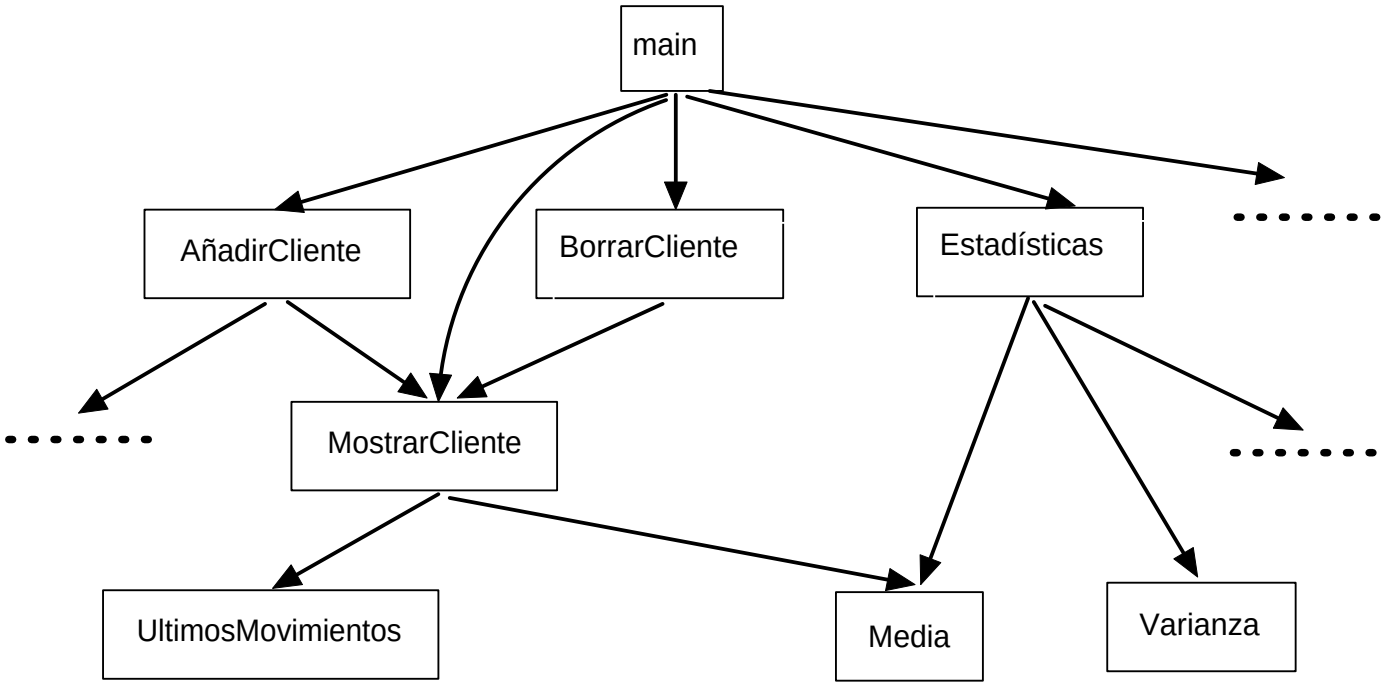
3.4.2. DISEÑO MODULAR PARA LA INTEGRACIÓN DE LAS FUNCIONES

Las funciones resuelven tareas específicas, pero

¿Cómo se integran las funciones en un programa complejo?

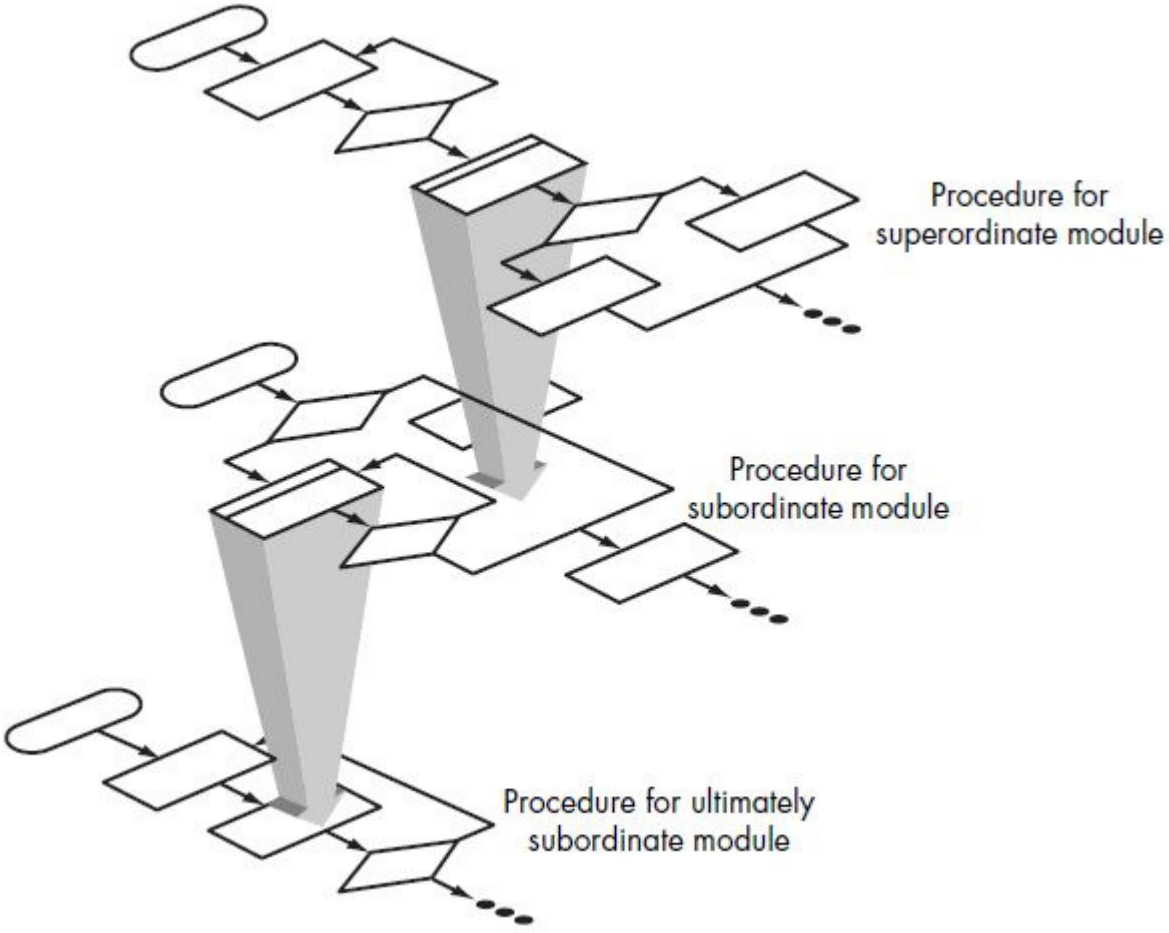
La descomposición modular puede ser sencilla (las flechas indican llamadas de una función a otra), pero en problemas reales será compleja, con varios niveles





Las funciones de alto nivel son las encargadas de definir la lógica de la aplicación y coordinar las llamadas a las funciones de niveles inferiores.

¿En qué orden construimos las funciones?


Procedure for
superordinate module
Procedure for
subordinate module
Procedure for ultimately
subordinate module

- ▷ ***Diseño descendente*** **(*Top-Down*).**
  Se diseñan las funciones de primer nivel
  Se vuelve a hacer lo mismo para cada nivel →
  → *refinamiento sucesivo* (*stepwise refinement*)
- ▷ ***Integración ascendente*** **(*Bottom-Up*).**
  Se diseñan y construyen las funciones de bajo nivel
  Se van diseñando y construyendo las funciones de los niveles superiores

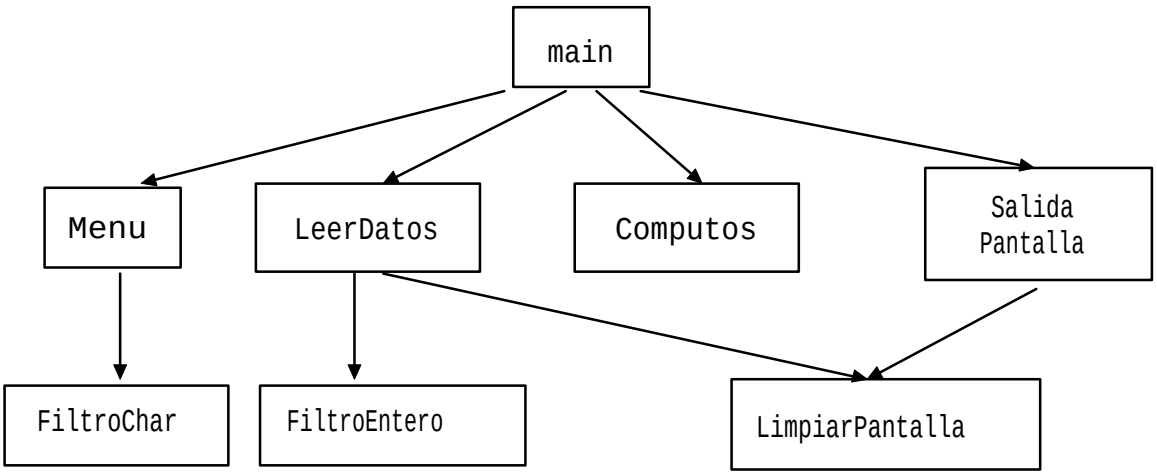
Usualmente se llega a un compromiso entre ambos:

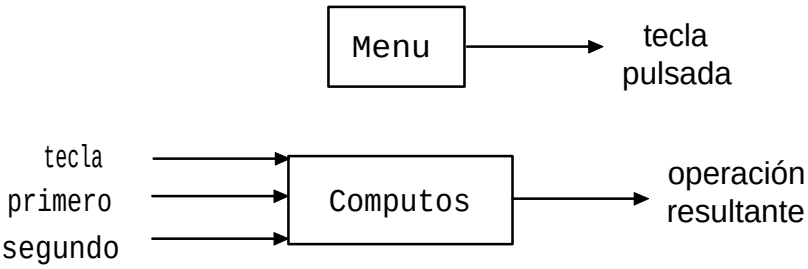
- ▷ **Se realiza el diseño de la descomposición modular de las funciones de primer nivel. Obtenemos sus prototipos.**
- ▷ **Se escribe el esqueleto de la función principal (`main`), con las llamadas a los prototipos anteriores.**
- ▷ **Esto mismo se realiza (descendentemente) en cada nivel (ahora, la función principal es la del nivel superior)**
- ▷ **El proceso sigue hasta que lleguemos a un nivel suficientemente bajo (funciones que resuelven tareas muy específicas). Vamos implementando las llamadas a las funciones, aunque no estén terminadas (*stubs*).**
- ▷ **En paralelo, o posteriormente, nos centramos en una *rama* y empezamos a implementar y validar las funciones de los últimos niveles. Así se van integrando en las funciones de niveles superiores.**
- ▷ **Durante este proceso, es posible que haya que revisar la descomposición modular.**

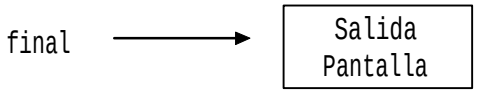
Ejemplo.

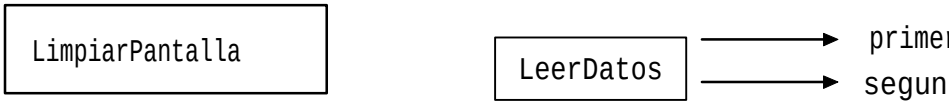
1. ***Análisis de requisitos*.**
   Ofrecer un menú al usuario para sumar, restar o hallar la media aritmética de dos enteros.

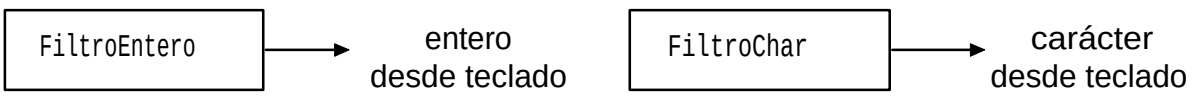
2. ***Descomposición modular*.**











Los prototipos serían:

```
void LimpiarPantalla();
char FiltroChar();
int FiltroEntero();
char Menu();
void LeerDatos (int *primero, int *segundo);
float Computos (char tecla, int primero, int segundo);
void SalidaPantalla (float final);
```

3. *Diseño descendente. Nivel 1*. Construimos el *esqueleto* de `main` (todavía no se han definido las funciones)

```
#include <stdio.h>

void LeerDatos (int *primero, int *segundo);
char Menu ();
float Computos (char tecla, int primero, int segundo);
void SalidaPantalla (float final);

int main(){
   int     dato1, dato2;
   float  final;
   char    opcion;

   LeerDatos(&dato1, &dato2);
   opcion = Menu();
   final = Computos(opcion, dato1, dato2);
   SalidaPantalla(final);
}
```

4. ***Diseño descendente. Nivel 2*. Construimos el *esqueleto* de cada una de las funciones llamadas en el `main`. Por ejemplo:**

```
void LimpiarPantalla();
int FiltroEntero();

void LeerDatos(int *primero, int *segundo){
   LimpiarPantalla();
   *primero = FiltroEntero();
   *segundo = FiltroEntero();
}
```

Hacemos lo mismo con el resto de funciones: `Menu`, `Computos`, `SalidaPantalla`. **Por ejemplo:**

```
char FiltroChar(){
}

char Menu(){
   char tecla;

   printf("\nElija una opción\n");
   printf("\nS. Sumar");
   printf("\nR. Restar");
   printf("\nM. Media Aritmética\n");
   tecla = FiltroChar();

   return tecla;
}
```

5. *Integración ascendente*. Procederíamos a definir las funciones, aunque sea una primera versión (stub)

```
void LimpiarPantalla(){
   printf("\n\n");
}
int FiltroEntero(){
   int aux;

   printf("Introduzca un entero");
   scanf("%d", &aux);
   return aux;
}
```

Cuando tengamos tiempo, implementaremos mejor estas funciones. Pero con esta primera versión, ya pueden empezar las pruebas.

Hacemos lo mismo con el resto de ramas. Por ejemplo:

```
char FiltroChar(){
   char caracter;

   printf("Introduzca un carácter ");
   scanf("%c", &caracter);

   return caracter;
}

char Menu(){
   char tecla;

   printf("\nElija una opción\n");
   printf("\nS. Sumar");
   printf("\nR. Restar");
   printf("\nM. Media Aritmética\n");
   tecla = FiltroChar();

   return tecla;
}
```

El programa completo, siguiendo estos pasos, podría quedar como sigue:

```
#include <stdio.h>

void LimpiarPantalla();
char FiltroChar();
int FiltroEntero();
char Menu();
void LeerDatos (int *primero, int *segundo);
float Computos (char tecla, int primero, int segundo);
void SalidaPantalla (float final);

int main(){
   int     dato1, dato2;
   float  final;
   char    opcion;

   LeerDatos(&dato1, &dato2);
   opcion = Menu();
   final = Computos(opcion, dato1, dato2);
   SalidaPantalla(final);
}

void LimpiarPantalla(){
      printf("\n\n");
}

char FiltroChar(){
   char caracter;
```

```
    printf("Introduzca un carácter ");
    scanf("%c", &caracter);
    return caracter;
}

int FiltroEntero(){
    int aux;

    printf("Introduzca un entero");
    scanf("%d", &aux);
    return aux;
}

char Menu(){
    char tecla;

    printf("\nElija una opción\n");
    printf("\nS. Sumar");
    printf("\nR. Restar");
    printf("\nM. Media Aritmética\n");
    tecla = FiltroChar();
    return tecla;
}

void LeerDatos (int *primero, int *segundo){
    LimpiarPantalla();
    *primero = FiltroEntero();
    *segundo = FiltroEntero();
}

float Computos (char tecla, int primero, int segundo){
```

```
    float resultado;

    switch (tecla){
       case 'S': resultado = primero+segundo;
                 break;
       case 'R': resultado = primero-segundo;
                 break;
       case 'M': resultado = (primero+segundo)/2.0;
                 break;
    }
    return resultado;
}

void SalidaPantalla (float final){
    LimpiarPantalla();
    printf("\n\nResultado = %f", final);
}
```

Ampliación:

Consultad *Diseño estructurado* en Wikipedia

3.4.3. OTRAS METODOLOGÍAS

El *diseño modular* de una solución se basa en que sólo tenemos funciones que se comunican a través de parámetros (los datos del programa).

No hemos entrado en detalles y sólo hemos visto un ejemplo simple de diseño top-down con integración ascendente.

Cuando el programa es medianamente complejo, este diseño no es fácil de realizar ni mantener. Hoy en día, el más aceptado es el *diseño orientado a objetos*.

La idea es encapsular en un mismo sitio (en un *objeto*) los datos y las funciones. El enfoque del programa se basa en ver cómo se comunican entre sí los objetos.

3.4.4. EL CICLO DE VIDA DE UN PROGRAMA

Programar no es únicamente ejecutar el compilador, escribir el programa, y venderlo.

$$\textit{Problema} \rightarrow \textit{Análisis de requisitos} \leftrightarrow \textit{Diseño} \leftrightarrow \textit{Implementación} \leftrightarrow \textit{Validación y Verificación}$$

- ▷ ***Análisis de requisitos*.**
  Especificación de las necesidades de la empresa.
- ▷ ***Diseño*.**
  - **Elección de la arquitectura (sistema operativo, servidor web, etc)**
  - **Elección de la metodología (estructurada, funcional, orientada a objetos, etc)**
  - **Diseño de la solución, según la metodología usada.**
    Si es estructurada, identificación de las tareas, descomposición modular del problema, y diseño de las funciones (que resuelven dichas tareas).
- ▷ ***Implementación*.**
  - **Elección de las herramientas de programación (Visual C++.NET, Java, ASP, etc)**
  - **Codificación de los componentes (funciones, objetos, etc)**
- ▷ ***Proceso de validación y verificación*.**
  Será fundamental la realización de baterías de pruebas.

3.5. FUNCIONES RECURSIVAS

3.5.1. CONCEPTO

Hacer que una función en el cuerpo de la función llame a la misma función

```
<tipo> funcion (argumentos)
{
    ........
    funcion(argumentos);
}
```

- ▷ **Se trata de, en potencia, un bucle infinito. Cuidado con la condición de fin. Hay que pensar en ella y estar seguro que la recursión para.**
- ▷ **En la codificación siempre se codifica primero la condición de fin y luego el caso general.**
- ▷ **Lo fácil es hacer funciones que devuelven un valor y que no reciben parametros por referencia.**
- ▷ **Cuantos menos parámetros mejor, pues todo se aloja en la pila. Corremos riesgo de agotar la pila si tenemos muchos parámetros y la recursión da muchas vueltas.**

3.5.2. EJEMPLOS

3.5.2.1. FACTORIAL

```
int factorial(int n)
```

```
{
   int salida;

   if (n==0)
      return 1;

   salida = n * factorial(n-1);

   return(salida);
}
```

3.5.2.2. FIBONACCI

```
int fibonacci(int n)
{
   int salida;

   if (n<=1)
      return 1;

   salida = fibonacci(n-1)+fibonacci(n-2);

   return(salida);
}
```

3.5.2.3. LIBERAR LISTA

```
struct Nodo{
   char info[100];
```

```
   struct Nodo *sig;
};

void liberarLista(struct Nodo **lista)
{
  if (*lista != NULL)
    {
      if((*lista)->sig != NULL)
        liberarLista(&((*lista) -> sig));

      free (*lista);
      *lista = NULL;
    }
}
```