

# Índice general

<b>3.1. Conceptos Básicos . . . . .</b>	<b>3</b>
<b>3.1.1. Tareas y funciones . . . . .</b>	<b>3</b>
<b>3.2. Funciones . . . . .</b>	<b>6</b>
<b>3.2.1. Introducción a las funciones . . . . .</b>	<b>6</b>
<b>3.2.1.1. Definición . . . . .</b>	<b>6</b>
<b>3.2.1.2. Parámetros formales y actuales .</b>	<b>7</b>
<b>3.2.2. Ámbito de un dato . . . . .</b>	<b>15</b>
<b>3.2.3. La Pila . . . . .</b>	<b>23</b>
<b>3.2.4. Ejemplos de funciones . . . . .</b>	<b>27</b>
<b>3.2.5. Ejemplos de funciones mal codificadas .</b>	<b>28</b>
<b>3.3. Funciones void . . . . .</b>	<b>30</b>
<b>3.3.1. Motivación y Definición . . . . .</b>	<b>30</b>
<b>3.3.2. Paso de parámetros por referencia . . . .</b>	<b>37</b>
<b>3.3.3. Funciones versus funciones void con un         parámetro por referencia . . . . .</b>	<b>50</b>
<b>3.4. Modularización . . . . .</b>	<b>55</b>
<b>3.4.1. El ciclo de vida de un programa . . . . .</b>	<b>55</b>
<b>3.4.2. Diseño modular de la solución . . . . .</b>	<b>57</b>
<b>3.4.2.1. Datos de entrada y de salida . . .</b>	<b>57</b>

3.4.2.2.	Metodología e integración de las funciones . . . . .	59
3.4.2.3.	Abanico de entrada y de salida .	71
3.4.2.4.	El diseño del programa principal	72
3.4.3.	Diseño de una función . . . . .	74
3.4.3.1.	Documentación de una función .	74
3.4.3.2.	Las funciones como trabajadores de una empresa . . . . .	79
3.4.3.3.	Ocultamiento de información . .	80
3.4.3.4.	Separar E/C/S . . . . .	82
3.4.3.5.	Información de errores . . . . .	84
3.4.3.6.	Fomentar la reutilización en otros problemas . . . . .	87
3.4.3.7.	Que la llamada a la función no necesite siempre hacer las mismas acciones previas . . . . .	88
3.4.3.8.	Validación de datos devueltos . .	90
3.4.3.9.	Evitar efectos colaterales . . . .	94
3.4.3.10.	Resumen . . . . .	99
3.5.	Cuestiones específicas de C++ . . . . .	100
3.5.1.	Funciones: sentencias y expresiones . . .	100
3.5.2.	Otras cuestiones sobre funciones . . . .	102

## TEMA 3. MODULARIZACIÓN

### 3.1. CONCEPTOS BÁSICOS

#### 3.1.1. TAREAS Y FUNCIONES

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

Cada tarea que identifiquemos será resuelta por una función. Suelen ser de notación prefija, con argumentos (en su caso) separados por comas y encerrados entre paréntesis.

Las funciones como `sqrt`, `tolower`, etc., no son sino ejemplos de funciones incluidas en `cmath` y `cctype` respectivamente que resuelven tareas concretas, devolviendo un valor.

**Ejemplo.**

```
.....  
int main(){  
    double 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(){
    double 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(){
    double 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, sabiendo el valor de los lados.

Importante. 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.

## 3.2. FUNCIONES

### 3.2.1. INTRODUCCIÓN A LAS FUNCIONES

#### 3.2.1.1. 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.

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

### 3.2.1.2. PARÁMETROS FORMALES Y ACTUALES

- Los parámetros formales son aquellos especificados en la cabecera de la función (*entrada*).  
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>);`

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

int main(){
    double resultado, valor;

    valor = 4;
    resultado = Cuadrado(valor);    // resultado = 16
    cout << "El cuadrado de " << valor << " es "
         << resultado;
}
```

### ***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, en tiempo de ejecución se realiza la asignación

***parámetro formal = parámetro actual***

En el ejemplo, `entrada = 4`

- 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. Dicho lugar es, en general, una expresión del mismo tipo que la expresión devuelta por `return`
- A continuación, el flujo de control prosigue por la línea siguiente a la llamada.



```
int main(){  
    double resultado, valor;
```

```
    valor = 4;  
    resultado = Cuadrado(valor);
```

```
    cout << "El cuadrado de " << valor << " es "  
          << resultado;
```

```
}
```

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

The diagram illustrates the modularization of code. A box on the left contains the `main` function, and a box on the right contains the `Cuadrado` function. A curved arrow originates from the `Cuadrado(valor)` call in `main` and points to the `Cuadrado` function definition. Another curved arrow originates from the closing brace of the `Cuadrado` function and points back to the `resultado` variable in `main`, indicating the return of the function's result.

## Correspondencia entre parámetros actuales y formales:

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

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

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

```
#include <iostream>
#include <cmath>
using namespace std;
double Resta(double valor_1, double valor_2){
    return valor_1 - valor_2;
}
int main(){
    double un_valor = 5.0, otro_valor = 4.0;

    cout << "\nResta = " << Resta(un_valor, otro_valor);
    cout << "\nResta = " << Resta(otro_valor, un_valor);
}
```

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

**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(4000000000000);
        // devuelve basura
    .....
}
```

- **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);
```

**Ejercicio.** Definid la función `Media` 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.

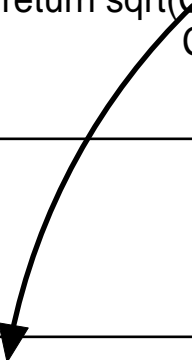
- **Dentro de una función se puede llamar a cualquier otra función que esté definida con anterioridad. El paso de parámetros entre funciones sigue los mismos criterios.**

```
#include <iostream>
#include <cmath>
using namespace std;

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

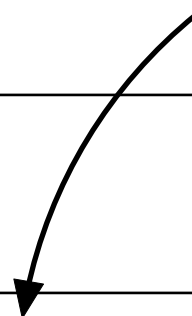
double Hipotenusa(double ladoA, double ladoB){
    return sqrt(Cuadrado(ladoA) + Cuadrado(ladoB));
}
```

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



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

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



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

### 3.2.2. ÁMBITO DE UN DATO

Dentro de una función podemos declarar constantes y variables. Sólo se conocen dentro de la función. Se les llama *datos locales*.

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

Las variables locales 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.**

```
#include <iostream>
using namespace std;

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

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

    return aux;
}

int main(){
    int valor, resultado;

    cout << "\nIntroduzca valor";
    cin >> valor;

    resultado = Factorial(valor);
    cout << "Factorial de " << valor << " = "
        << resultado;
}
```



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

```
#include <iostream>
using namespace std;

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

    valor = 5;    // Error de compilación

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

    return aux;
}

int main(){
    int valor, resultado;

    i = 5;    // Error de compilación

    cout << "\nIntroduzca valor";
    cin >> valor;

    resultado = Factorial(valor);
    cout << "Factorial de " << valor << " = "
        << resultado;

}
```

**Ámbito** 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:

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

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

En este caso, el ámbito de la variable termina con la llave } que cierra la estructura condicional, repetitiva, etc. El uso de variables con un ámbito restringido a un bloque, es una buena práctica, ya que oculta información.

Pero ¡no abusar! restringirlo 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 <iostream>
using namespace std;

int Factorial (int valor){
    int i;
    int aux = 1;

    for (i=2; i<=valor; i++)
        aux = aux * i;

    return aux;
}

int main(){
    int valor = 3 , resultado;

    resultado = Factorial(valor);
    cout << "Factorial de " << valor << " = "
        << resultado;

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

```
#include <iostream>
#include <cmath>
using namespace std;

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

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

    cout << "\nIntroduzca primer lado";
    cin >> ladoA;
    cout << "\nIntroduzca segundo lado";
    cin >> ladoB;

    hip = Hipotenusa(ladoA,ladoB);
    cout << "\nLa hipotenusa vale " << valor;
}
```

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

```
#include <iostream>
using namespace std;

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);
    cout << "Suma hasta " << 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.**

**Debemos tener cuidado con la declaración de variables con igual nombre que otra definida en un ámbito *superior*.**

```
#include <iostream>
using namespace std;

int Suma_desde_0_hasta(int tope){
    int suma;

    suma = 0;
    while (tope>0){
        int suma = 0;

        suma = suma + tope;
        tope--;
    }

    return suma;
}

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

    resultado = Suma_desde_0_hasta(tope);
    cout << "Suma hasta " << tope << " = "
        << resultado;

    // Imprime en pantalla lo siguiente:
    // Suma hasta 5 = 0                                :-(
}
```

### 3.2.3. 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.

- En principio, cada entorno, sólo puede acceder a sus propios datos.

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 (ya sean por valor o por referencia)
  - 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.
- `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 más parámetros, pero en MP1 sólo veremos el caso sin parámetros. Por eso, la hemos declarado siempre como:

```
int main(){  
    .....  
}
```

- 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



```
#include <iostream>
using namespace std;

int Factorial (int valor){
    int i;
    int aux = 1;

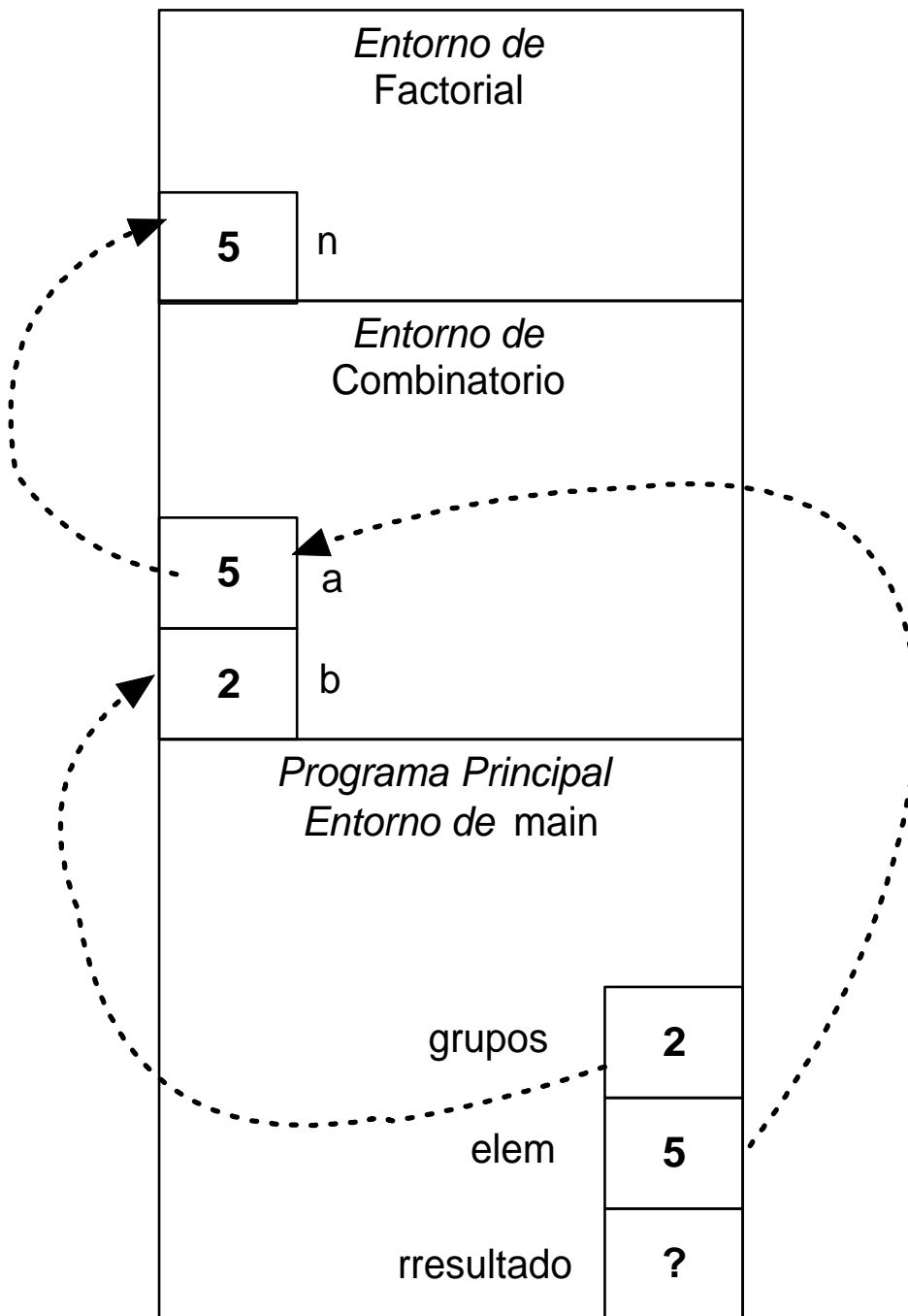
    for (i=2; i<=valor; i++)
        aux = aux * i;

    return aux;
}

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);
    cout << elem << " sobre " << grupos << " = "
        << resultado;
}
```



**Pila**

## 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)
- **Modificación.** 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.

### 3.2.4. EJEMPLOS DE FUNCIONES

*Ejercicio.* Comprobad si un número es par.

*Ejercicio.* Comprobad si un número es primo.

*Ejercicio.* Calculad el MCD de dos enteros.

*Ejercicio.* Calculad el MCM de dos enteros.

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

### 3.2.5. EJEMPLOS DE FUNCIONES MAL CODIFICADAS

```
int f (int n) {    //  :-(
    return n+2;
    cout << "Nunca se ejecuta \n";
}
```

```
int Factorial (int n)  {    //  :-(
    int i, aux;

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

```
bool EsPar (int n)  {    //  :-(
    if (n%2==0)
        return true;
}
```

```
bool EsPrimo (int n){    //  :-(
    int i;

    for (i=2 ; i<=sqrt(n) ; i++)
        if (n%i == 0)
            return false;
    return true;
}
```

**En la medida de lo posible, no introduciremos sentencias `return` en varios sitios distintos del cuerpo de la función. Dejadlo al final de la función.**

## 3.3. FUNCIONES VOID

### 3.3.1. MOTIVACIÓN Y DEFINICIÓN

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

```
int i;

.....

for (i=1; i<=3 ; i++)
    cout << "\n*****";
cout << "Programa básico de Trigonometría";
for (i=1; i<=3 ; i++)
    cout << "\n*****";

.....
```

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

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

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.

**Caso conocido:** `system("pause")`

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

```
void <nombre-procedim> (<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 <iostream>
#include <cmath>
using namespace std;

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

double LeePositivo(){
    int leeme;

    do{
        cin >> leeme;
    }while (leeme<=0);
}

void Presentacion(){
    int i;
    for (i=1; i<=3 ; i++)
        cout << "\n*****";
    cout << "Programa básico de Trigonometría";
    for (i=1; i<=3 ; i++)
        cout << "\n*****";
}

void MostrarHipotenusa (double valor){
    cout << "\nLa hipotenusa vale " << valor;
}
```



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

    Presentacion();

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

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

**Importante.** 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.** Queremos calcular el cociente y el resto de la división entera entre dos valores.

```
#include <iostream>
using namespace std;
void Pausa(){
    cout << endl;
    system("pause");
}
void ImprimirAsteriscos (int tope){
    cout << endl;
    for (int i=1 ; i<=tope ; i++)
        cout << "*";
}
void MostrarResultados (int coc, int res){
    ImprimirAsteriscos(24);
    cout << "\nEl cociente es:      " << coc;
    cout << "\nEl resto entero es: " << res;
    ImprimirAsteriscos(24);
    Pausa();
}
int main(){
    int dividendo, divisor, cociente, resto;
    cout << "Introduzca un entero ";
    cin >> dividendo;
    cout << "Introduzca otro entero ";
    cin >> divisor;
    cociente = dividendo / divisor;
    resto = dividendo % divisor;
    MostrarResultados (cociente, resto);
}
```

```
int main(){
    int dividendo, divisor,
        cociente, resto

    cout << "Introduzca un entero";
    .....
    MostrarResultados(cociente, resto);
}
```

```
void MostrarResultados(int coc, int res)
{
    ImprimirAsteriscos(24);
    cout << "\nEl cociente .....
    cout << "\nEl resto .....
    ImprimirAsteriscos(24);
    Pausa();
}
```

```
void ImprimirAsteriscos(int tope){
    int i;
    cout << endl;

    for (i=1 ; i<= tope ; i++)
        cout << "*";
}
```

**Ejercicio.** Cread una función para que imprima en pantalla todos los divisores de un número entero.

### 3.3.2. PASO DE PARÁMETROS POR REFERENCIA

Supongamos que queremos calcular el cociente y el resto, dentro de una función.

```
#include <iostream>
using namespace std;
.....
//  :-(
void CocienteResto(int divdo, int dvsor, int coc, int res){
    coc = divdo / dvsor;
    res = divdo % dvsor;
}

int main(){
    int dividendo, divisor, cociente, resto;

    cout << "Introduzca un entero ";
    cin >> dividendo;
    cout << "Introduzca otro entero ";
    cin >> divisor;

    CocienteResto(dividendo, divisor, cociente, resto);
    MostrarResultados(cociente, resto);
}
```

## Las sentencias:

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

**modifican las variables locales `coc` y `res` de `CocienteRestoMAL`. Las variables `cociente` y `resto` no se modifican (se quedan con ?). Solución: Paso por referencia.**

---

## Formas de pasar un parámetro:

- ***Paso por valor.*** El parámetro formal recibirá una copia del valor del parámetro actual. 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 formal no afectan al actual.

Por ahora, sólo hemos usado el paso por valor.

- ***Paso por referencia.*** El parámetro formal estará ligado al parámetro actual. Las modificaciones en el formal afectan al actual. Se indicará poniendo un `&` en la declaración del parámetro formal.

Tanto *dentro* del cuerpo de la función como en la *llamada*, el parámetro se trata como si fuera un dato más (es decir, el identificador sin `&`).

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`.

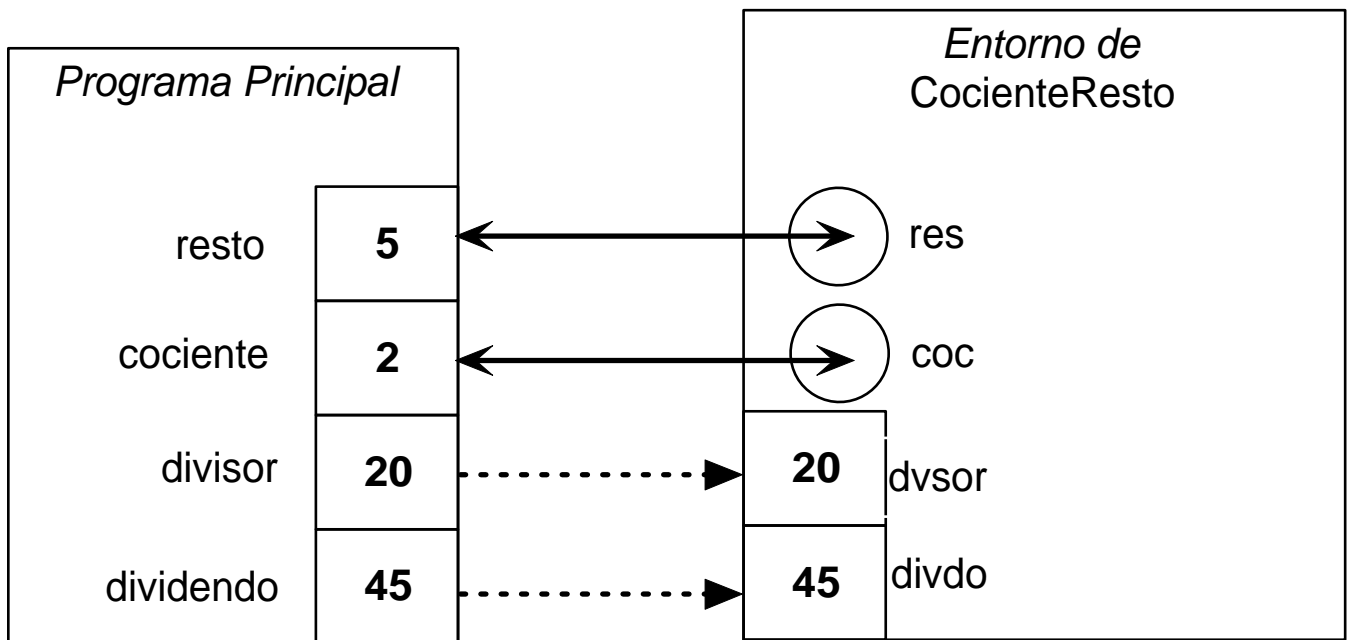
**Nota.** En C no existe el paso por referencia.

```
#include <iostream>
using namespace std;
.....
//  :-)
void CocienteResto(int divdo, int dvsor, int &coc, int &res){
    coc = divdo / dvsor;
    res = divdo % dvsor;
}

int main(){
    int dividendo, divisor, cociente, resto;

    cout << "Introduzca un entero ";
    cin >> dividendo;
    cout << "Introduzca otro entero ";
    cin >> divisor;

    CocienteResto(dividendo, divisor, cociente, resto);
    MostrarResultados(cociente, resto);
}
```



Pila

La lectura del dividendo y divisor también se puede encapsular en una función. En este caso, vamos a construir una función `void` para modificar dos variables: por tanto serán pasadas por referencia.



```
#include <iostream>
#include <cstdlib>
using namespace std;

void Pausa(){
    cout << endl;
    system("pause");
}

void ImprimirAsteriscos (int tope){
    int i;

    cout << endl;
    for (i=1 ; i<=tope ; i++)
        cout << "*";
}

void MostrarResultados (int cociente, int resto){
    ImprimirAsteriscos(24);
    cout << "\nEl cociente es:      " << cociente;
    cout << "\nEl resto entero es: " << resto;
    ImprimirAsteriscos(24);
    Pausa();
}

void LecturaValores(int &divdo, int &dvsor){
    cout << "Introduzca un entero ";
    cin >> divdo;
    cout << "Introduzca otro entero ";
    cin >> dvsor;
}
```

```
void CocienteResto(int divdo, int dvsor,
                   int &coc, int &res){
    coc = divdo / dvsor;
    res = divdo % dvsor;
}

int main(){
    int dividendo, divisor, cociente, resto;

    // Antes de la llamada a LecturaValores,
    // dividendo y divisor contienen ?

    LecturaValores(dividendo,divisor);

    // Después de la llamada a LecturaValores,
    // dividendo y divisor contienen los valores
    // introducidos por el usuario
    // Por contra, antes de la llamada a CocienteResto
    // cociente y resto contienen ?

    CocienteResto(dividendo, divisor, cociente, resto);
    MostrarResultados (cociente, resto);
}
```

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 = 1;
    Incrementa(dato);
    cout << "\ndato = " << dato;    // <- Imprime 2
}
```

---

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

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 <iostream>
using namespace std;

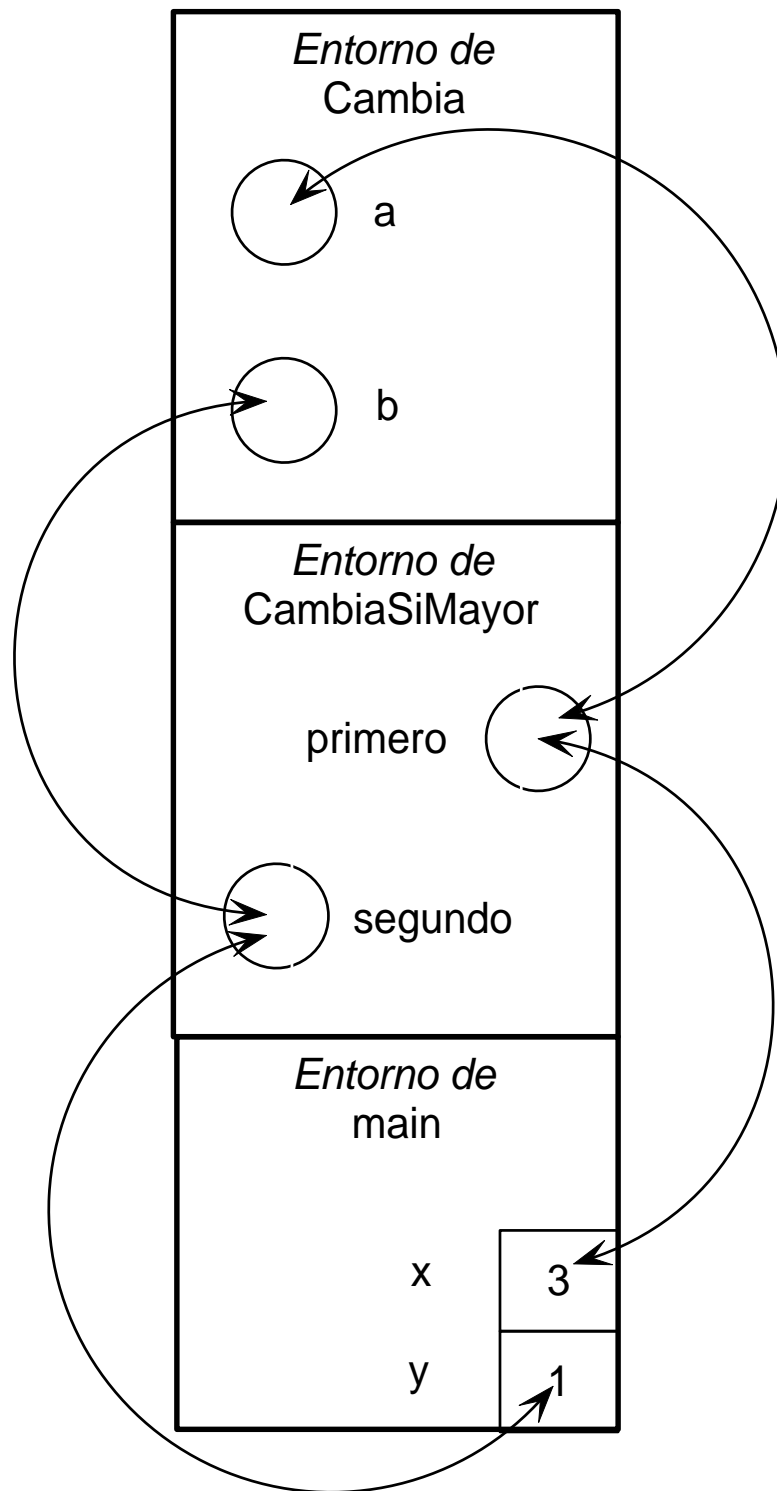
void Cambia(double &a, double &b){
    double aux;

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

void CambiaSiMayor (double &primero, double &segundo){
    if (primero > segundo)
        Cambia (primero,segundo);
}

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

    CambiaSiMayor(x,y);
    cout << x << " " << y;
}
```



## Algunas cuestiones:

- Viendo la llamada a una función no podemos saber cuáles son los parámetros pasados por referencia. Debemos mirar la cabecera de la función :- (

```
CocienteResto(dividendo,divisor,cociente,resto);  
// ¿Por valor/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) :- (
```

- En C++ no se pueden definir funciones dentro de otras. Todas están al mismo nivel.

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

```
#include <iostream>
using namespace std;

// Prototipos:
void CambiaSiMayor(double &primero, double &segundo);
void Cambia(double &a, double &b);

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

    CambiaSiMayor(x,y);
    cout << x << " " << y;
}

void CambiaSiMayor (double &primero, double &segundo){
    if (primero > segundo)
        Cambia (primero,segundo);
}

void Cambia(double &a, double &b){
    double aux;

    aux = a;
    a = b;
    b = aux;
}
```



**Incluso no es necesario poner los nombres de los parámetros en el prototipo:**

```
void Cambia(double &, double &); // Prototipo
```

**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*.

### 3.3.3. 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);  
cout << "Factorial de 4 = " << 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 aux = 1;  
  
    for (i=2; i<=n; i++)  
        aux = aux * i;  
    resultado = aux;  
}
```

**Incluso podemos suprimir la variable aux:**

```
void Factorial (int n, int &resultado){  
    int i;  
  
    resultado = 1;  
    for (i=2; i<=n; i++)  
        resultado = resultado * i;  
}
```

**Llamada:**

```
Factorial (4, variable);  
cout << "Factorial de 4 = " << variable;
```

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

**Con una función que devuelve un** char.

```
#include <iostream>
using namespace std;

char Menu(){
    char tecla;

    cout << "\nElija una opción\n";
    cout << "\nS. Sumar";
    cout << "\nR. Restar";
    cout << "\nM. Media Aritmética\n";
    cin >> tecla;

    return tecla;
}

int main(){
    char opcion;

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

**Con una función void.**

```
#include <iostream>
using namespace std;

void Menu(char &tecla){
    cout << "\nElija una opción\n";
    cout << "\nS. Sumar";
    cout << "\nR. Restar";
    cout << "\nM. Media Aritmética\n";
    cin >> tecla;
}

int main(){
    char opcion;

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

En cualquier caso, 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.

## 3.4. MODULARIZACIÓN

### 3.4.1. EL CICLO DE VIDA DE UN PROGRAMA

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

*Problema*  $\longrightarrow$  *Análisis de requisitos*  $\longleftrightarrow$   
 $\longleftrightarrow$  *Diseño*  $\longleftrightarrow$  *Implementación*  $\longleftrightarrow$   
 $\longleftrightarrow$  *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, atendiendo a 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.**

**Todo este proceso se detallará en la asignatura *Ingeniería del software*.**



## 3.4.2. DISEÑO MODULAR DE LA SOLUCIÓN

### 3.4.2.1. DATOS DE ENTRADA Y DE SALIDA

Datos de Entrada  $\longrightarrow$  **Algoritmo**  $\longrightarrow$  Datos de Salida

**Nota.** No confundir datos de entrada (resp. salida) con entrada de datos -cin- (resp. salida de resultados -cout-).

**Algoritmo para calcular la media aritmética:**

- Datos de entrada: Un conjunto de N valores numéricos.
- Datos de salida: Un real con la media aritmética.

**Algoritmo para dibujar un cuadrado:**

- Datos de entrada: longitud del lado (L), coordenadas iniciales (X,Y)
- Datos de salida: Ninguno

**Algoritmo para calcular la desviación típica:**

- Datos de entrada: Un conjunto de N valores numéricos.
- Datos de salida: Un real con la desviación típica.

Las salidas de un algoritmo son entradas a otros algoritmos.

---

Algoritmos  $\Longleftrightarrow$  Funciones

Datos de entrada (salida)  $\Longleftrightarrow$  parámetros de las funciones

Parámetros pasados por valor	→	Datos de entrada
Parámetros pasados por referencia	→	Datos de salida
		Datos de e/s

Datos de Entrada	→	<b>Función</b>	→	1 dato de Salida
------------------	---	----------------	---	------------------

Datos de Entrada	→	<b>Función void</b>	→	0, 2 o más salidas
------------------	---	---------------------	---	--------------------

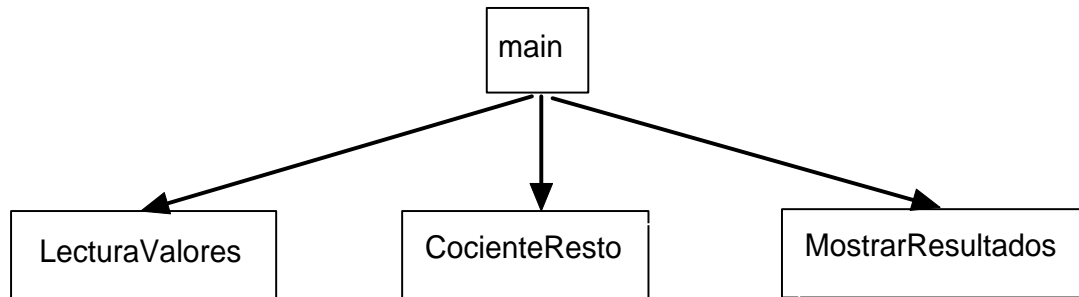
Los datos de salida de unas funciones son datos de entrada para otras:

```
int main(){
    int dividendo, divisor, cociente, resto;

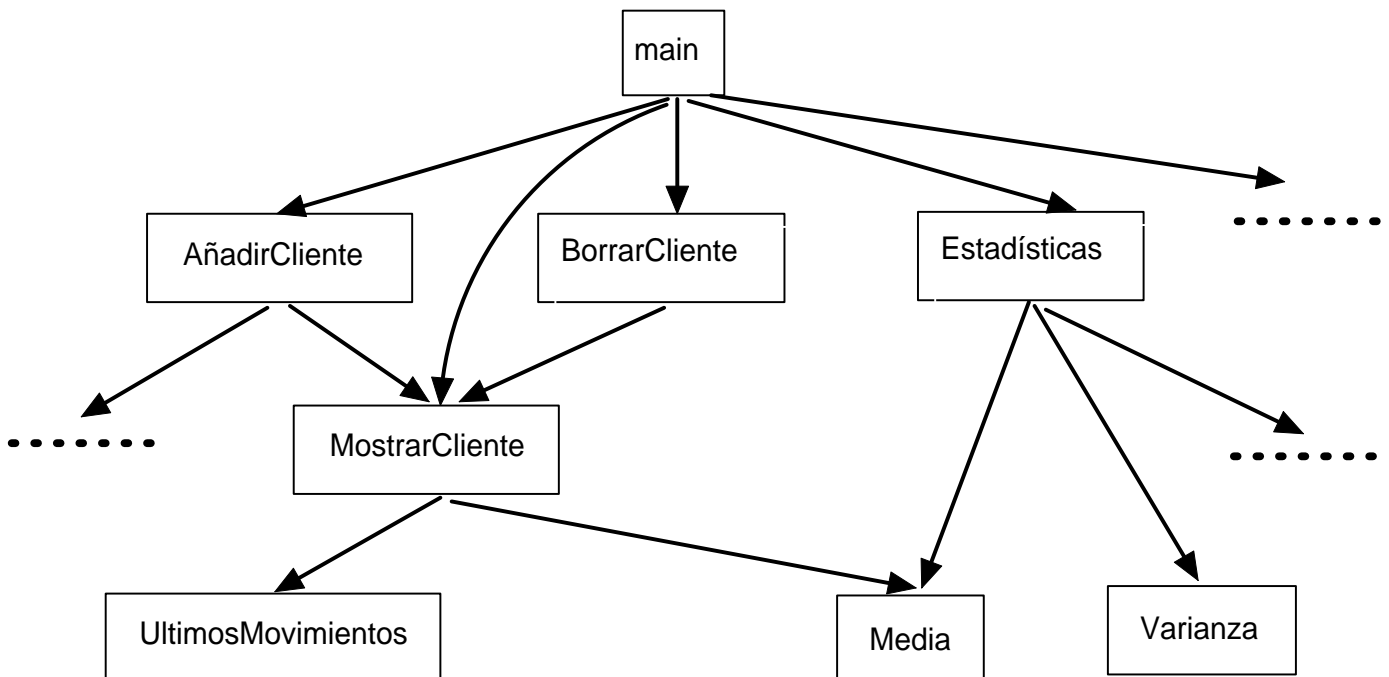
    LecturaValores(dividendo,divisor);
    CocienteResto(dividendo,divisor,cociente,resto);
    MostrarResultados (cociente,resto);
}
```

### 3.4.2.2. METODOLOGÍA E INTEGRACIÓN DE LAS FUNCIONES

La descomposición modular puede ser sencilla como la anterior (las flechas indican llamadas de una función a otra):



pero en problemas reales será compleja, con varios niveles:



## Pasos a seguir:

- Se realiza el diseño de la descomposición modular de las funciones de alto nivel.
- Se escribe el esqueleto del programa, con las llamadas a las funciones dentro del `main` (primer nivel). En esta fase, es muy útil el concepto de prototipo en C++, ya que permite definir las cabeceras de las funciones y realizar las llamadas, comprobando que éstas son correctas, sin necesidad de definir completamente dichas funciones.

Esto mismo se realiza (descendentemente) en cada nivel, 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*).

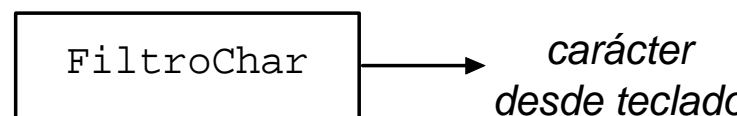
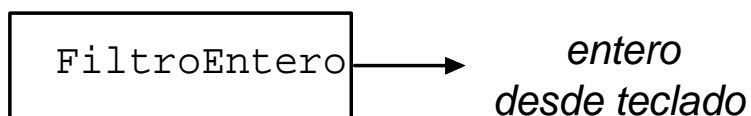
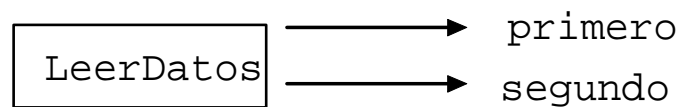
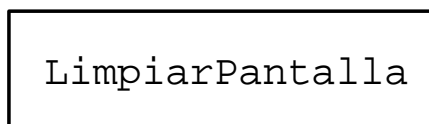
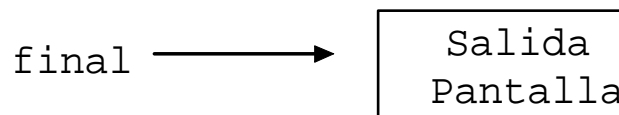
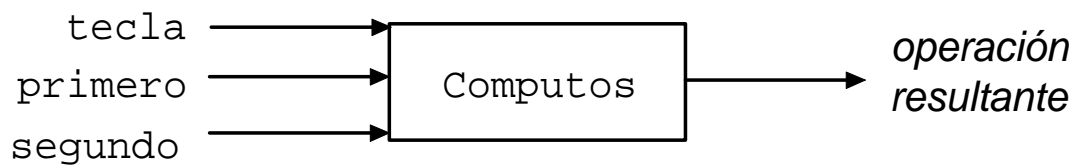
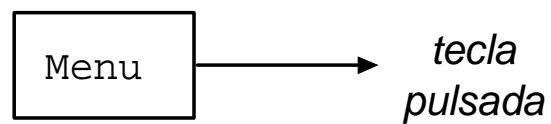
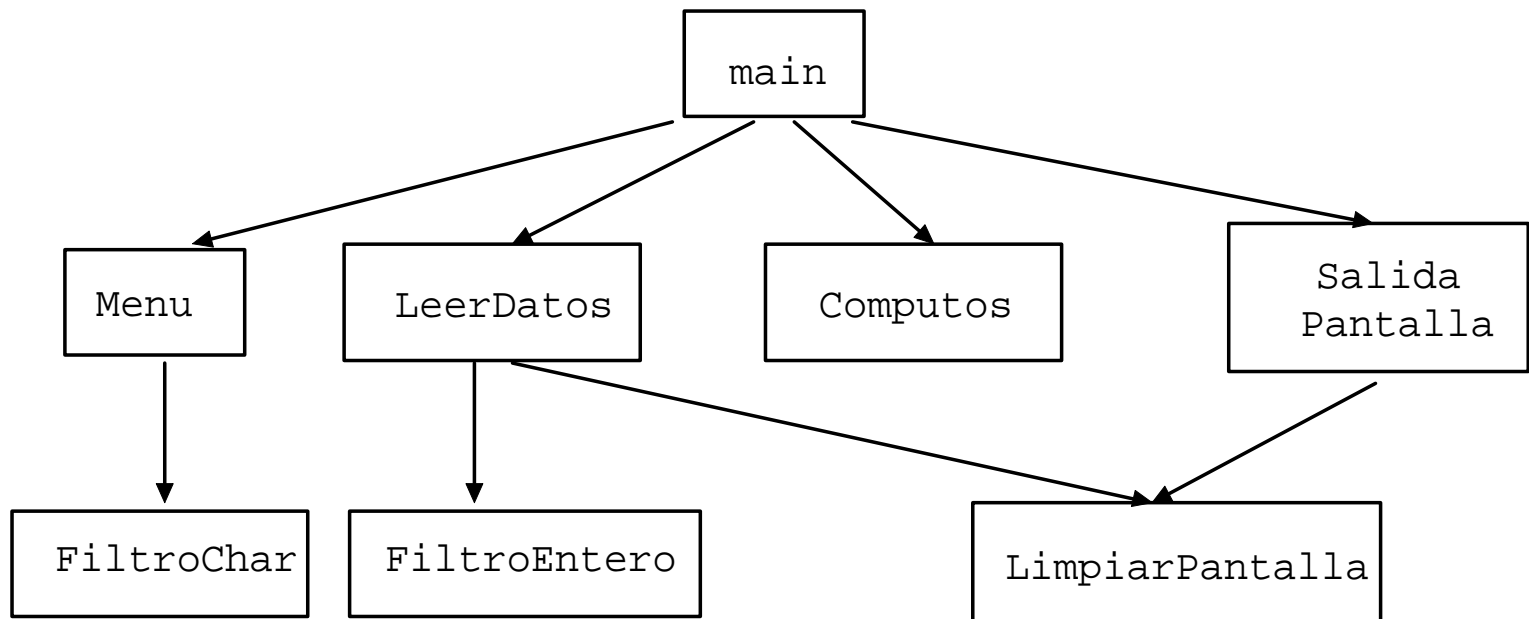
- En paralelo, o posteriormente, nos centramos en una *rama* y empezamos a implementar y validar las funciones de los últimos niveles. Así se van integrando en las funciones de niveles superiores.
- Durante este proceso, es posible que haya que revisar la descomposición modular.

***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);  
double Computos (char tecla, int primero, int segundo);  
void SalidaPantalla (double final);
```

### 3. Integración descendente. Nivel 1. Construimos el esqueleto de main (todavía no se han definido las funciones)

```
#include <iostream>

using namespace std;

void LeerDatos (int &primero, int &segundo);
char Menu ();
double Computos (char tecla, int primero, int segundo);
void SalidaPantalla (double final);

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

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



#### 4. Integración 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;

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

    return tecla;
}
```

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

```
void LimpiarPantalla(){
    cout << '\n' << '\n';
}

int FiltroEntero(){
    int aux;

    cout << "Introduzca un entero";
    cin >> 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 character;

    cout << "Introduzca un carácter ";
    cin >> character;

    return character;
}
```

```
char Menu(){
    char tecla;

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

    return tecla;
}
```

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

```
#include <iostream>
using namespace std;

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

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

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

void LimpiarPantalla(){
    cout << '\n' << '\n';
}

char FiltroChar(){
    char character;
```

```
    cout << "Introduzca un carácter ";
    cin >> character;
    return character;
}

int FiltroEntero(){
    int aux;

    cout << "Introduzca un entero";
    cin >> aux;
    return aux;
}

char Menu(){
    char tecla;

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

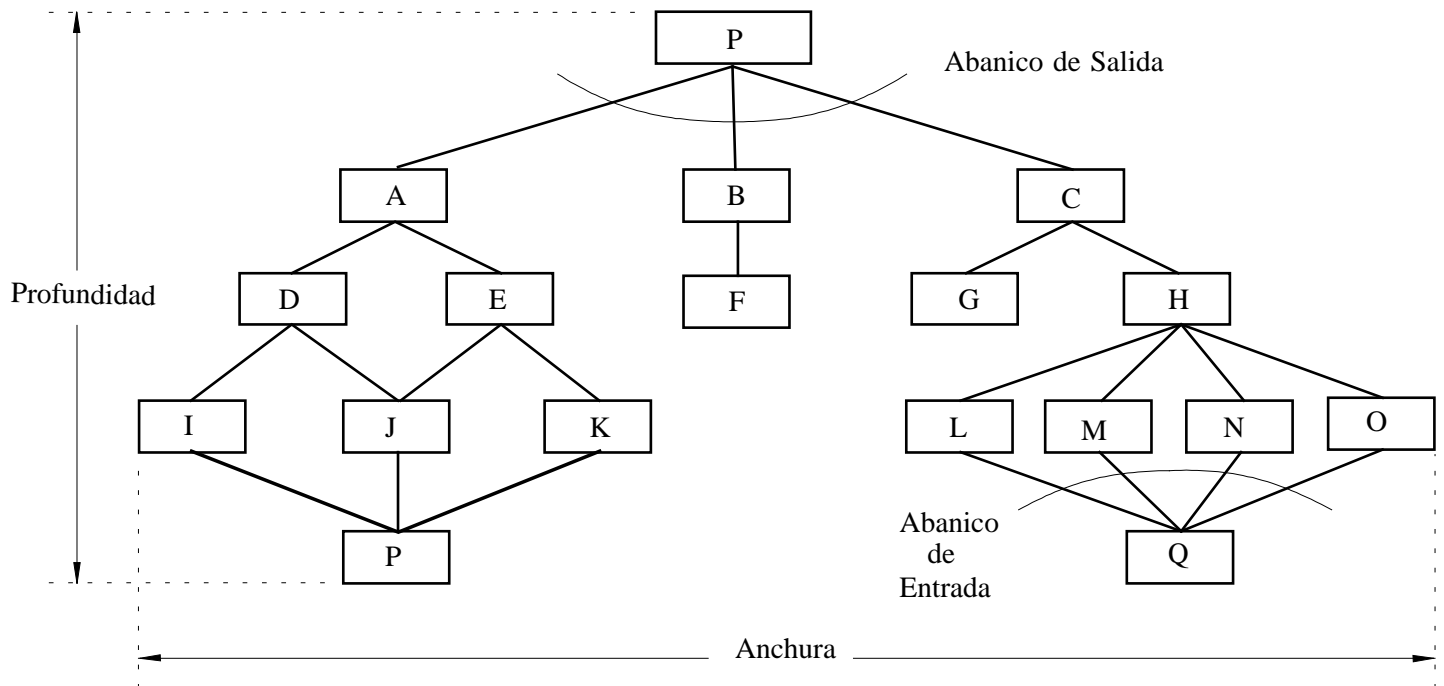
void LeerDatos (int &primero, int &segundo){
    LimpiarPantalla();
    primero = FiltroEntero();
    segundo = FiltroEntero();
}
```

```
double Computos (char tecla, int primero, int segundo){
    double resultado;

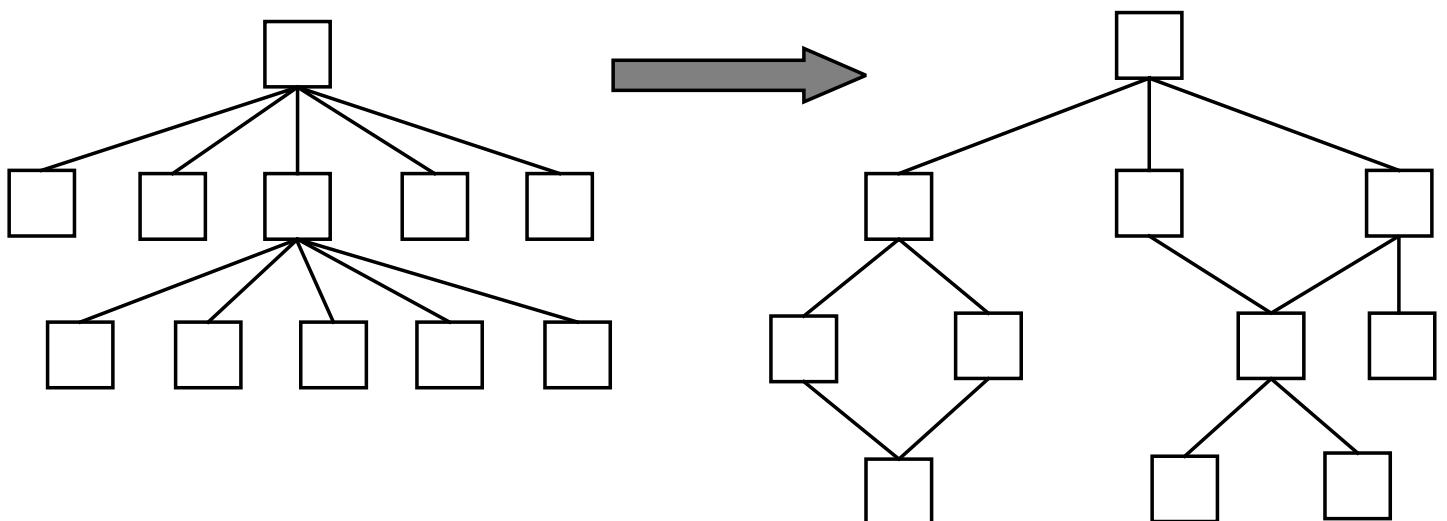
    switch (tecla){
        case 'S': resultado = primero+segundo;
                    break;
        case 'R': resultado = primero-segundo;
                    break;
        case 'M': resultado = (primero+segundo)/2.0;
                    break;
    }
    return resultado;
}

void SalidaPantalla (double final){
    LimpiarPantalla();
    cout << "\n\nResultado = " << final;
}
```

### 3.4.2.3. ABANICO DE ENTRADA Y DE SALIDA



**Consejo: Aumentar el abanico de entrada conforme se desciende**



### 3.4.2.4. EL DISEÑO DEL PROGRAMA PRINCIPAL

Evitar el uso de funciones monolíticas dentro de `main`.

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

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

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

    return final;
}

int main(){
    double final;

    final = HazloTodo();
    SalidaPantalla(final);
}
```



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

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

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

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

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

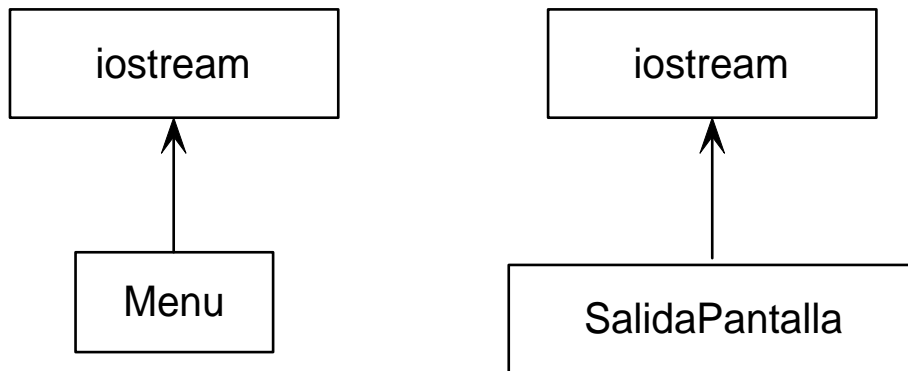
**Podemos pensar en la función `main` como el director de una empresa. El director no es un vago. Se encarga de dirigir a sus empleados, que no es poco.**

### 3.4.3. DISEÑO DE UNA FUNCIÓN

#### 3.4.3.1. DOCUMENTACIÓN DE UNA FUNCIÓN

En la fase de definición de prototipos (integración descendente) debemos documentar apropiadamente las funciones:

- **Descripción de la tarea que realiza la función.**  
Se describe lo que hace, **PERO NO COMO LO HACE**. Esto se hará dentro de la función.  
Importante: Usar comentarios concisos, pero claros y completos, con una presentación visual agradable y esquemática.
- **Descripción de los datos de entrada, incluyendo las restricciones (o precondiciones) que deben satisfacer para obtener unos datos de salida correctos.**
- **Descripción de los datos de salida, 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).

**Documentemos algunos de las funciones del ejemplo del menú.**

```
//-----  
/*  
Identificador: Menu  
Tipo devuelto: char  
Cometido:    Presentar un menú al usuario  
              para sumar, restar o dividir dos enteros.  
              Leer la opción del usuario.  
Entradas:    Ninguna  
Salidas:     Opción del usuario  
              'S' para sumar  
              'R' para restar  
              'M' para la media  
Bibliotecas: iostream  
*/  
char Menu();  
//-----
```

```
//-----  
/*  
Identificador: Computos  
Tipo devuelto: double  
                Resultado de la operación realizada  
Cometido:      Dependiendo del valor de tecla:  
                sumar, restar o  
                hallar la media de dos enteros.  
Entradas:  
    tecla:      Opción del usuario  
                'S' para sumar  
                'R' para restar  
                'M' para la media  
    primero:    Primer entero para operar  
    segundo:    Segundo entero para operar  
Bibliotecas:   Ninguna  
*/  
double Computos(char tecla, int primero, int segundo);  
//-----
```

```
//-----  
/*  
Identificador: LeerDatos  
Tipo devuelto: void  
Cometido: Leer dos enteros desde teclado.  
Entradas: Ninguna  
Salidas:  
    primero: primer entero  
    segundo: segundo entero  
Bibliotecas: iostream  
*/  
void LeerDatos (int &primero, int &segundo);  
//-----  
  
//-----  
/*  
Identificador: SalidaPantalla  
Tipo devuelto: void  
Cometido: Imprimir en pantalla el resultado  
           de la operación escogida por el usuario  
Entradas:  
    final: Valor a mostrar.  
Salidas: Ninguna  
Bibliotecas: iostream  
*/  
void SalidaPantalla (double final);  
//-----
```

### 3.4.3.2. 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.

### 3.4.3.3. OCULTAMIENTO DE INFORMACIÓN

**Hay que diseñar una función pensando en lo que estrictamente necesita.**

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

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

    return fact;
}
int main(){
    cout << "\nFactorial de 3 = " << Factorial(3); // :-)
}
```

---

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

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



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

    minuscula = character - amplitud;
    return minuscula;
}
```

---

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

    minuscula = character - amplitud;
    return minuscula;
}
```

### 3.4.3.4. SEPARAR E/C/S

**En la medida de lo posible, no mezclar E/C/S en una misma función.**

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

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

    cout << "\nEl factorial de " << n << " es " << fact;
}

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

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

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

    return fact;
}

void ImprimirFactorial(int n){
    cout << "\nEl factorial de " << n << " es "
        << 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 cálculos del factorial en una función, para que pueda ser usada en otros programas que no quieran escribir el resultado con `cout`.

***Una función de cálculo NUNCA debe imprimir mensajes.  
Hacedlo, en su caso, en la función que la llama***

### 3.4.3.5. INFORMACIÓN 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.

```
double LeerDatoFicheroMAL(){
    .....
    if (error)
        cout << "\nSe produjo un error"; // C/S :-(
    .....
}

int main(){
    double 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(double &dato, bool &error){...}

int main(){
    bool ErrorAcceso;
    double valor;
    .....
    LeerDatoFichero(valor,ErrorAcceso);

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

```
void LeerDatoFichero(double &dato, char &error){....}

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

    switch (ErrorAcceso){
        case '0':  [Operaciones]
                    break;
        case '1':  cout << "\nDisco protegido contra escritura";
                    break;
        case '2':  cout << "\nFichero no existente";
                    break;
        .....
    }
}
```

***Si una función tiene una variable de control (error), debe asignarle siempre un valor antes de terminar***

### 3.4.3.6. 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(double coef1, double coef2, double coef3,  
           double &distancia, double &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(double coef1, double coef2, double coef3,  
           double &raiz1, double &raiz2, double &NumRaices)
```

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

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

### 3.4.3.7. QUE LA LLAMADA A LA FUNCIÓN NO NECESITE SIEMPRE HACER 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 true en una variable llamada Indet.

.....

```
void PotenciaMAL(double base, double exponente,
                 double &resultado, bool &Indet){
    if ((base==0) && (exponente == 0))
        Indet = true;
    else
        resultado = pow(base,exponente);
}

int main(){
    double dato1, dato2, pot;
    bool Indeterminacion;

    <Lectura de dato1 y dato2>

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



**En esta primera versión, la llamada a PotenciaMAL requiere que siempre ejecutemos previamente `Indeterminacion = false`;**  
**Problema: Alguna vez se nos olvidará!**

```
void Potencia (double base, double exponente,
               double &resultado, bool &Indet){
    if ((base==0) && (exponente == 0))
        Indet = true;
    else{
        resultado = pow(base,exponente);
        Indet = false;
    }
}

int main(){
    double dato1, dato2, pot;
    bool Indeterminacion;

    <Lectura de dato1 y dato2>

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

    if (Indeterminacion)
        cout << "\n0^0 = Indeterminación";
    else
        cout << "\n" << dato1 << "^" << dato2 << " = " << pot;
}
```

**Conseguimos una función mucho más robusta.**

***Las inicializaciones que requiera una función, deben hacerse dentro de ella.***

### 3.4.3.8. VALIDACIÓN DE DATOS DEVUELTOS

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

**Primera versión:**

```
#include <iostream>
#include <cctype>
using namespace std;

void SalidaPantalla (double final){
    cout << "\n\nResultado = " << final;
}

char MenuMAL(){
    char tecla;

    cout << "\nElija una opción\n";
    cout << "\nS. Sumar";
    cout << "\nR. Restar";
    cout << "\nM. Media Aritmética\n";
    cin >> tecla;
    return tecla;
}

int main(){
    int    dato1=3, dato2=5;
    double 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*1.0/dato2;
                break;
}

SalidaPantalla(resultado);
}
```

## Segunda versión:

```
#include <iostream>
#include <cctype>
using namespace std;

void SalidaPantalla (double final){
    cout << "\n\nResultado = " << final;
}

char Menu (){
    char tecla;

    cout << "\nElija una opción\n";
    cout << "\nS. Sumar";
    cout << "\nR. Restar";
    cout << "\nM. Media Aritmética\n";

    do{                                     // :-)
        cin >> tecla;
        tecla = toupper(tecla);
    }while ((tecla!='S') && (tecla!='R') && (tecla!='M'));

    return tecla;
}

int main(){
    int    dato1=3, dato2=5;
    double 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);
}
```

**Conseguimos una función mucho más robusta.**

***Procurad que las funciones comprueben  
la validez de los datos devueltos***

### 3.4.3.9. EVITAR EFECTOS COLATERALES

La mayoría de los lenguajes de programación (incluido C++) permiten definir variables fuera de las funciones. Automáticamente, su ámbito incluye todas las funciones definidas 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:**

```
int Max;
bool saturacion;

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

void ControlVuelos(){
    if (NumVuelosActual <= Max)
        [acciones vuelos]
    else{
        ActivarEmergenciaVuelos();
        saturacion = true;
    }
}

int main(){
    Max = 30;  // <- Máximo número de vuelos

    saturacion = false;

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

**Segunda Versión:**

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

bool ControlVuelos(int Max){
    bool saturacion=false;

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

int main(){
    bool saturacion;

    do{
        GestionMaletas(50);
        saturacion = ControlVuelos(30);
    }while (!saturacion)
}
```



**Resumiendo:** El uso de variables globales, hace que cualquier función las pueda usar y/o modificar, lo que provoca efectos laterales. Esto va en contra de un principio básico en programación: *ocultación de información*.

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

```
const double Pi = 3.1415927;    // Cte "universal"

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

**Nos ahorramos un parámetro con respecto a:**

```
int FuncionTrigonometria(int Pi, int parametro){
    .....
}
```

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

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

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

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

---

```
cons int IRPF = 0.18      // Cte "no 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.

### ***Ventajas:***

- Las cabeceras de las funciones que usan constantes se simplifican.

**Encontrar la solución apropiada a cada problema no es sencillo.**

### 3.4.3.10. 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.

## 3.5. CUESTIONES ESPECÍFICAS DE C++

### 3.5.1. 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++ y otros lenguajes. 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 <iostream>
using namespace std;

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

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

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

    cout << "\nMedia entre "
         << dato1 << " y " << 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 `<cstdio>`, se define la función `printf` análoga a `cout`

```
#include <cstdio>
using namespace std;

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

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

```
#include <cstdio>
#include <iostream>
using namespace std;

int main(){
    int total;

    total = printf("Hola ");
    cout << "Total caracteres impresos: " << total;
}
```

Realmente, una función `void` es una función más que devuelve un *tipo nulo* (`void`) a la función que la llame.

### 3.5.2. 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 otros parámetros. Esto se verá en MP11.
- 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* para como sinónimo de función (ya sea un `void` o devuelva un valor)