

TEMA 5. FUNCIONES

Grado en Ingeniería en Tecnologías Industriales
Programación



CONTENIDOS

5.1. PROGRAMACIÓN MODULAR

5.2. IMPLEMENTACIÓN DE FUNCIONES

5.3. LLAMADA A FUNCIONES

5.4. PASO DE PARÁMETROS A UNA FUNCIÓN: POR VALOR Y POR REFERENCIA

5.5. ÁMBITO DE DECLARACIÓN DE VARIABLES. VISIBILIDAD

5.6. BIBLIOTECAS DE FUNCIONES

5.7. ANEXOS

5.7.1. BIBLIOTECAS ESTÁNDAR DE C

5.7.2. BIBLIOTECAS DE FUNCIONES EN DEV-C++

- 5.1. PROGRAMACIÓN MODULAR
- 5.2. IMPLEMENTACIÓN DE FUNCIONES
- 5.3. LLAMADA A FUNCIONES
- 5.4. PASO DE PARÁMETROS A UNA FUNCIÓN: POR VALOR Y POR REFERENCIA
- 5.5. ÁMBITO DE DECLARACIÓN DE VARIABLES. VISIBILIDAD
- 5.6. BIBLIOTECAS DE FUNCIONES
- 5.7. ANEXOS
 - 3.7.1. BIBLIOTECAS ESTÁNDAR DE C
 - 3.7.2. BIBLIOTECAS DE FUNCIONES EN DEV-C++

5.1. PROGRAMACIÓN MODULAR

Programación modular

- Técnicas de programación para crear buenos programas
 - Programación modular
 - Programación estructurada
- ¿Qué es un buen programa?
 - **Funcionalmente correcto:** Produce los resultados requeridos
 - **Legible:** Fácilmente comprensible por cualquier programador
 - **Modificable:** Diseñado de forma que la incorporación de modificaciones sea sencilla
 - **Fácil de depurar:** Diseñado de forma que la localización y corrección de errores sea sencilla
 - **Bien documentado:** Incluye comentarios y documentación suplementaria que permite a otro programador comprender su funcionamiento

Programación estructurada

- Programación convencional:
 - Elabora programas sin seguir ningún método de programación.
 - Resultado: Programas muy largos y muy difíciles de mantener.
- Programación estructurada
 - Todo programa tiene un único punto de inicio y un único punto de fin
 - Uso de un número limitado de estructuras de control: secuenciales, alternativas y repetitivas
 - Prohibidos los saltos de una instrucción a otra

Programación modular

- Programación modular:
 - Se basa en la descomposición del problema en problemas más simples (**módulos**) que se pueden analizar, programar y depurar independientemente
- Un módulo es:
 - Un conjunto de instrucciones que realizan una tarea concreta y/o proporcionan unos determinados resultados, y que puede ser llamada (invocada) desde el programa principal o desde otros módulos
 - **Módulo, subprograma o función son sinónimos**
 - En C, los llamamos **funciones**
 - Ejemplos: `funcionOrdenarLista`, `funcionCalcularMedia`

Ventajas de la programación modular

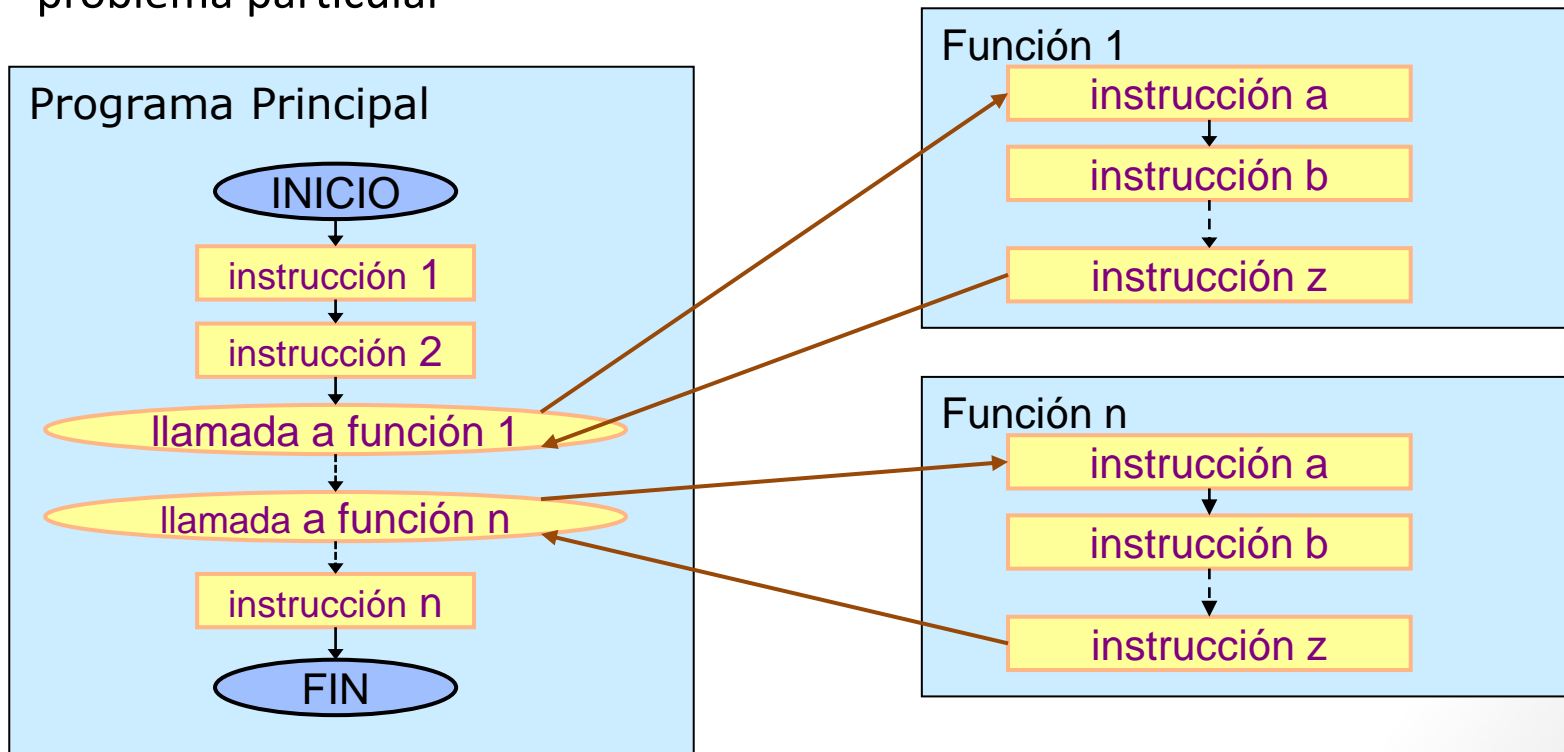
- Ventajas de la Programación Modular:
 - Programas más **estructurados y legibles**.
 - Programas más cortos y simples, debido a la división del problema complejo en partes
 - Los **subprogramas** son **independientes**
 - Se pueden **escribir, compilar y probar** de forma independiente, por lo que en un programa de gran tamaño pueden trabajar distintos programadores.
 - Se puede **modificar** un subprograma sin afectar al resto de del programa, por lo que se pueden realizar cambios en un módulo sin que sea necesario modificar (ni volver a probar) el resto
 - Los **subprogramas** son **reutilizables**.
 - Los módulos pueden ser utilizados en distintos programas que hagan uso de la misma funcionalidad.

Programación modular

- **La adecuada división de un programa en subprogramas constituye un aspecto fundamental en el desarrollo de cualquier programa.**

Programación modular

- Un programa consta de:
 - **Programa principal**: contiene operaciones fundamentales y las llamadas a los subprogramas.
 - **Subprogramas** (funciones) : programas independientes que resuelven un problema particular



Programación modular en C: Funciones

- C es un lenguaje que orientado a la programación modular
- Todo programa en C se compone de varios módulos denominados **funciones**
- El programa principal en C es la función **main**
 - Se puede ver como una función a su vez, llamada desde el sistema operativo
- Las funciones son llamadas desde la función main o desde otras funciones

- 5.1. PROGRAMACIÓN MODULAR
- 5.2. IMPLEMENTACIÓN DE FUNCIONES
- 5.3. LLAMADA A FUNCIONES
- 5.4. PASO DE PARÁMETROS A UNA FUNCIÓN: POR VALOR Y POR REFERENCIA
- 5.5. ÁMBITO DE DECLARACIÓN DE VARIABLES. VISIBILIDAD
- 5.6. BIBLIOTECAS DE FUNCIONES
- 5.7. ANEXOS
 - 3.7.1. BIBLIOTECAS ESTÁNDAR DE C
 - 3.7.2. BIBLIOTECAS DE FUNCIONES EN DEV-C++

5.2. IMPLEMENTACIÓN DE FUNCIONES

Elementos fundamentales

- Los elementos fundamentales de una función son
 - Su **nombre**, que se usa para llamarla (**invocarla**)
 - Ejemplo: **CalcularMedia**
 - El **resultado** que devuelve
 - Ejemplo: media
 - Los datos que usa para realizar su tarea
 - Ejemplo: num1, num2, num3
 - Llamados **parámetros o argumentos**
 - Las instrucciones que realizan la tarea

Declaración de una función: Prototipo

- Prototipo
 - Para poder usar una función, primero hay que declararla, igual que se hace con las variables
 - Para ello se usa una forma predefinida denominada prototipo
 - El prototipo informa de la existencia de la función, que está implementada más adelante
 - Tiene que estar antes de que se utilice por primera vez esa función
 - al comienzo del programa (después de los `#define` e `#include`)
 - El prototipo de una función siempre termina con el carácter `“;”`.

Prototipo

- En el prototipo se especifica
 - **tipo**: Tipo de dato que devuelve la función.
 - **nombre**: Nombre asignado a la función.
 - Autoexplicativo.
 - **Debería empezar por una letra minúscula.**
 - Si contiene varias palabras se marcarán poniendo la inicial de la segunda y subsiguientes palabras en mayúsculas.
 - **lista de parámetros**: Datos de entrada con los que trabaja la función

- Sintaxis del prototipo en C int a, int b

tipo nombre (lista de parámetros);

void , int, char, float, double, uint8_t, uint16.....

Prototipo: Ejemplos

- Ejemplos de prototipos de funciones

```
int potencia (int base, int exponente);
```

```
float suma (float n1, float n2);
```

```
void mostrarDatos (int a, int b);
```

```
int leerDato(void);
```

Valor devuelto por una función

- Resultado devuelto por la función
 - Una función realiza una serie de tareas y devuelve un **resultado**
 - También llamado valor de retorno
 - Al declarar la función hay que definir el **tipo de datos** de ese resultado
 - Puede ser int, float, double, char
 - No puede devolver un vector ni una matriz
 - Sí puede devolver un puntero a un vector o a una matriz
 - También puede no devolver ningún valor
 - En ese caso se declara que el tipo de dato devuelto es void

Parámetros o argumentos

- Parámetros o argumentos
 - Son los datos que la función recibe desde el programa que la llama
 - Parámetro y argumento son sinónimos
 - Para cada parámetro hay que indicar su nombre y tipo de datos
 - Se puede omitir el nombre (muy desaconsejable)
 - Puede haber uno, ninguno o más de uno
 - Si hay más de uno, se escriben **separados por comas**
 - Si no hay ninguno se escribe **void**
 - El valor de los parámetros puede modificarse o no dentro de la función

Estructura de un programa en C

```
#include          Directivas para el preprocesador  
#define
```

```
Declaraciones globales  
    Prototipos de las funciones
```

```
Función principal main  
int main (void)  
{  
    Declaración de variables y constantes locales  
    instrucciones  
}
```

```
Implementación de las funciones  
tipo1 funci(..)  
{  
    . . .  
}
```

Definición de una función

- Además del prototipo de la función hay que escribir la definición de la función
- Es el código de la función propiamente dicho, las instrucciones con las que la función realiza las tareas para las que ha sido diseñada.
- puede ubicarse en cualquier lugar del programa, con dos restricciones:
 - debe hallarse después de su prototipo
 - no puede estar dentro de la definición de otra función (incluida main)
- La definición de una función tiene dos partes
 - cabecera (header) .h
 - cuerpo (body) .c

Definición de una función: Cabecera

- Cabecera
 - primera línea
 - Es idéntica al prototipo declarado para la función
 - con dos diferencias:
 - no termina en “;”
 - en la lista de parámetros es obligatorio incluir el tipo y el nombre de los parámetros
 - en el prototipo el nombre se puede omitir (muy desaconsejado)

- Ejemplo

```
int suma (int a, int b)
```

Definición de una función: Cuerpo

- Cuerpo
 - conjunto de instrucciones que se ejecutan cada vez que se realiza una llamada a la función
 - entre llaves
- Ejemplo

```
{  
    int r;  
    r=a+b;  
    return r;  
}
```

return;

Definición de una función: Setencia return

- Instrucción **return**

- El valor devuelto por la función (valor de retorno) se especifica con la instrucción return
 - el tipo de dato de este valor debe coincidir con el definido en la cabecera y el prototipo de la función.
 - si la función no devuelve ningún valor, el tipo del valor de retorno debe ser void `return;`
- La instrucción return devuelve el control del programa a la función desde la que se realizó la llamada
 - Si hay otras instrucciones detrás no se ejecutan, finaliza la ejecución de la función

se recomienda incluir siempre una única sentencia *return* y que ésta esté situada al final de la función

(22)

Definición de una función: Variables locales

- Una función puede tener sus propias variables
 - Se las denomina **variables locales**
 - Se declaran al comienzo del cuerpo de la función
 - **Son sólo visibles dentro del bloque en el que han sido definidas, ocultas para el resto del programa**
 - Se crean cada vez que se ejecuta la función

Definición de una función

- Sintaxis de la definición de la función

```
tipo nombre(lista de parámetros)
{
    declaración de variables locales
    código ejecutable
    return (expresión);
}
```


Definición de una función: Ejemplo

Cabecera de la
función

```
int suma (int a, int b)
/*****
*** Función que devuelve la suma de dos números
*****/
```

Variable local de
la función

```
{
int r;
r=a+b;
return r;
}
```

Cuerpo de la
función

- 5.1. PROGRAMACIÓN MODULAR
- 5.2. IMPLEMENTACIÓN DE FUNCIONES
- 5.3. LLAMADA A FUNCIONES
- 5.4. PASO DE PARÁMETROS A UNA FUNCIÓN: POR VALOR Y POR REFERENCIA
- 5.5. ÁMBITO DE DECLARACIÓN DE VARIABLES. VISIBILIDAD
- 5.6. BIBLIOTECAS DE FUNCIONES
- 5.7. ANEXOS
 - 3.7.1. BIBLIOTECAS ESTÁNDAR DE C
 - 3.7.2. BIBLIOTECAS DE FUNCIONES EN DEV-C++

5.3. LLAMADA A FUNCIONES

Invocación de funciones

- La llamada (**invocación**) a una función se hace incluyendo su nombre en una expresión o instrucción
 - Ya sea en el programa principal o de otra función.
- El nombre de la función debe ir seguido de la lista de parámetros separados por comas y encerrados entre **paréntesis**.
 - Si la función no acepta parámetros se utilizan los paréntesis sin nada entre ellos.

```
c=suma (a, b) ;  
printf ("%d", suma (3, 7) ;
```

Invocación de funciones

- A los parámetros que aparecen en la **llamada** se les denomina **parámetros reales**
 - pueden ser variables, constantes y expresiones.
- A los parámetros que aparecen en la **definición** se les llama **parámetros formales**
 - Solo pueden ser variables
- El **número** de parámetros reales (en la llamada a una función) debe coincidir con el número de parámetros formales (en la definición y en la declaración)
- El **tipo de dato** de cada parámetro real debe coincidir con el tipo de dato del parámetro formal correspondiente
- Cuando el programa encuentra el nombre de la función, evalúa los parámetros reales, **pasa copia de dichos** valores a la función y entrega el control de la ejecución a la función.

Ejemplo de función

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

```
int suma(int a, int b);
```

```
int main(void)  
{
```

```
    //Declaración de variables del programa principal  
    int n1, n2, resu;
```

```
    //Leemos dos números
```

```
    printf("Dame dos numeros\n");
```

```
    scanf("%d%d",&n1, &n2);
```

```
    //Calculamos su suma llamando a una función:
```

```
    resu=suma(n1,n2);
```

```
    printf("La suma de %d y %d es %d", n1, n2, resu);
```

```
    return (0);
```

```
}
```

```
int suma (int a, int b)  
{
```

```
    int r;
```

```
    r=a+b;
```

```
    return r;
```

```
}
```

Prototipo de la
función suma

Parámetros formales

Llamada a la
función suma

Parámetros reales

Declaración de
la función suma

- 5.1. PROGRAMACIÓN MODULAR
- 5.2. IMPLEMENTACIÓN DE FUNCIONES
- 5.3. LLAMADA A FUNCIONES
- 5.4. PASO DE PARÁMETROS A UNA FUNCIÓN: POR VALOR Y POR REFERENCIA
- 5.5. ÁMBITO DE DECLARACIÓN DE VARIABLES. VISIBILIDAD
- 5.6. BIBLIOTECAS DE FUNCIONES
- 5.7. ANEXOS
 - 3.7.1. BIBLIOTECAS ESTÁNDAR DE C
 - 3.7.2. BIBLIOTECAS DE FUNCIONES EN DEV-C++

5.4. PASO DE PARÁMETROS A UNA FUNCIÓN: POR VALOR Y POR REFERENCIA

Paso de parámetros

- En la llamada a una función se pasan datos del programa principal a esa función ¿Cómo?
 - Se establece automáticamente una **correspondencia** entre los parámetros de la llamada (reales) y los del subprograma (formales).
 - Esta correspondencia está definida por la **posición**:
 - El primer parámetro real se corresponde con el primer parámetro formal; el segundo parámetro real con el segundo formal y así sucesivamente
 - Deben coincidir en número y tipo de datos
- Dos formas de pasar parámetros
 - **Por valor** – se pasa una copia
 - **Por variable** – se pasa un puntero

Paso de parámetros: Ejemplo

```
#include <stdio.h>

int suma(int a, int b);

int main(void)
{
    int n1, n2, resu; //Declaración de variables del prog. principal
    //Leemos dos números
    printf ("Dame dos numeros\n");
    scanf ("%d", &n1);
    scanf ("%d", &n2);

    //Calculamos su suma llamando a una función:
    resu=suma(n1,n2);

    printf("La suma de %d y %d es %d", n1, n2, resu);
    return 0;
}

int suma (int a, int b)
{
    int r;
    r=a+b;
    return r;
}
```


Paso de parámetros por valor

- La función recibe una **copia** de los valores de los parámetros reales
 - Esta copia queda almacenada en el parámetro formal (parámetro de la función receptora)
- La función trabaja sobre el parámetro formal.
 - Si se **cambia** el valor de un parámetro formal, el cambio sólo es visible dentro de la función y **no tiene efecto fuera de ella**
- Se debe usar el paso de parámetros por valor siempre que los argumentos no se van a modificar dentro de la función
 - ¿Y si se quiere modificar el valor de los parámetros pasados a una función y devolver este valor modificado?
 - **Usar entonces paso por referencia**

Paso de parámetros por valor: Ejemplo

```
/* Ejemplo: Paso de parámetros por valor*/
#include <stdio.h>
void demol(int valor);
void main(void)
{
    int n=10;
    printf("Antes de llamar a Demol, n=%d\n",n);
    demol(n);
    printf("Despues de llamar a Demol, n=%d\n",n);
    return ( );
}
void demol(int valor)
{
    printf("Dentro de Demol, valor=%d\n",valor);
    valor=999;
    printf("Dentro de Demol, valor=%d\n",valor);
    return( );
}
```

Resultado de la ejecución:

```
Antes de llamar a Demol, n=10
Dentro de Demol, valor=10
Dentro de Demol, valor=999
Despues de llamar a Demol, n=10;
```

Paso de parámetros por referencia

- Paso de parámetros por **referencia**:
 - Se pasa a la función una **referencia a la dirección** de memoria en la que está almacenado el dato que se quiere modificar (**puntero a la variable**)
 - Y no una nueva variable con una copia del parámetro real como se hace en el paso por valor
- Después de la llamada a la función, los valores se habrán modificado en el programa principal
 - Decimos que son parámetros de salida, la función devuelve resultados al programa principal a través de esos valores
 - El paso por referencia se usa al escribir funciones que devuelven más de un valor al programa principal
 - Con return se puede devolver un único valor de retorno

Paso de parámetros por referencia: Sintaxis

- En el programa principal (llamada):
 - El parámetro real va precedido por el operador dirección, indicando que pasamos un puntero a ese parámetro:

&var1

- En el prototipo de la función, en el encabezado y en el cuerpo de la función:
 - El parámetro formal va precedido por el operador indirección, indicando que accedemos al contenido de ese parámetro

<tipo *param1>

(*param1)

Paso de parámetros por referencia: Ejemplo

```
void incrementar (int *a);
```

```
int main (void){  
    int var1=1;  
    incrementar(&var1);  
    return 0;  
}
```

```
void incrementar (int *a){  
    *a=*a + 1;  
    return;  
}
```

Parámetro real: Referencia a la dirección de memoria en la que está almacenado el dato que se quiere modificar (el símbolo & precede al nombre de la variable)

Parámetro formal: Recibe la dirección de memoria en la que está almacenado el parámetro real. El parámetro formal está declarado como puntero al tipo de la variable original: int*)

Para acceder a la variable original se utiliza el operador de indirección (*) sobre el parámetro formal.

Para recordarlo, puede verse como si:

- En la llamada, el nombre del parámetro real comenzara por "&".
- En la función, el nombre del parámetro formal comenzará por "*".

Paso de parámetros por referencia: Ejemplo 2

```
#include <stdio.h>
void intercambio(int *x, int *y);
```

```
int main(void)
{
    int i=3;
    int j=50;
    printf("i=%d y j= %d\n", i, j);

    intercambio(&i, &j);
    printf("i=%d y j= %d\n", i, j);
    return 0;
}
```

```
void intercambio(int *x, int *y)
{
    int aux;
    aux=*x; //Paso1. aux toma el valor "apuntado" por x (i)
    *x=*y;  //Paso2. *x (i) toma el valor de *y (j)
    *y=aux; //Paso3. *y (i) toma el valor de aux
    return;
}
```

Parámetros reales: Referencia a la dirección de memoria en la que están almacenados los datos que se desean modificar (se usa el operador de dirección: "&")

Parámetros formales: Reciben la dirección de memoria en la que están almacenados los parámetros reales que se desean modificar. (Deben declararse como punteros)

Para acceder a la variable original se usa el **operador indirección: "*"**

Resumen:

- Paso de parámetros por **valor**:
 - El valor del parámetro real se **copia** en el parámetro formal
 - Los **cambios** efectuados sobre el parámetro formal (dentro de la función) **no** quedan **reflejados en el parámetro real** (fuera de la función)
- Paso de parámetros por **referencia**:
 - Los parámetros formales se declaran como punteros y reciben la **dirección de memoria** en la que se almacena el correspondiente parámetro real
 - para ello se usa el operador de dirección **&**
 - Cualquier **modificación** sobre el parámetro formal que se realice en la función **se mantendrá** una vez que termine la función
- El paso de parámetros por referencia **permite que una función pueda modificar más de un valor**

Parámetros *const* de una función

- Para garantizar que no se modifica por error el valor de una variable se puede forzar al compilador a que impida cualquier modificación, añadiendo el especificador **const** a la descripción de un parámetro formal:

```
int sumaDatos(const int x, const int y);
```

- El especificador **const** indica al compilador que, dentro de la función, el parámetro es de sólo lectura (no se puede modificar).
 - Si se intenta modificar el valor de este parámetro se producirá un mensaje de error en tiempo de compilación.

Parámetros *const* de una función: Ejemplo

```
#include <stdio.h>

int suma(int a, int b);

int main(void)
{
    int n1, n2, resu; //Declaración de variables del prog. principal

    //Leemos dos números
    printf("Dame dos numeros\n");
    scanf("%d",&n1);

    scanf("%d",&n2);

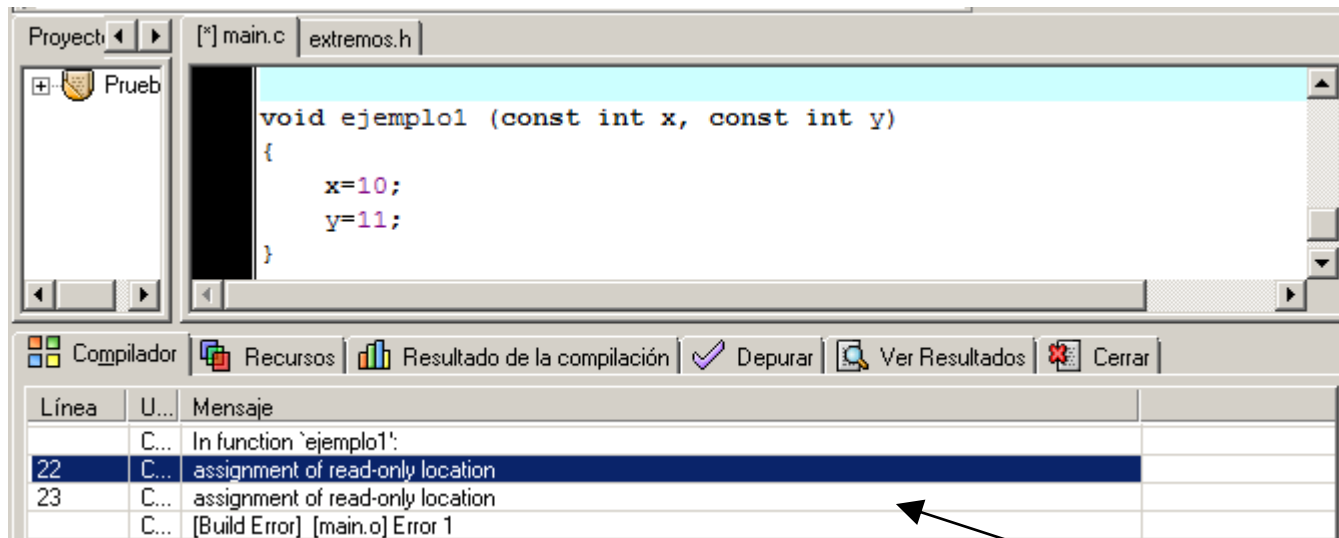
    //Calculamos su suma llamando a una función:
    resu=suma(n1,n2);

    printf("La suma de %d y %d es %d", n1, n2, resu);
    return 0;
}

int suma (const int a, const int b)
{
    int r;
    r=a+b;
    return r;
}
```

Indica al compilador que los parámetros son de lectura (no se pueden modificar)

Parámetros *const* de una función



Mensajes de error

- 5.1. PROGRAMACIÓN MODULAR
- 5.2. IMPLEMENTACIÓN DE FUNCIONES
- 5.3. LLAMADA A FUNCIONES
- 5.4. PASO DE PARÁMETROS A UNA FUNCIÓN: POR VALOR Y POR REFERENCIA
- 5.5. ÁMBITO DE DECLARACIÓN DE VARIABLES. VISIBILIDAD
- 5.6. BIBLIOTECAS DE FUNCIONES
- 5.7. ANEXOS
 - 3.7.1. BIBLIOTECAS ESTÁNDAR DE C
 - 3.7.2. BIBLIOTECAS DE FUNCIONES EN DEV-C++

5.5. ÁMBITO DE DECLARACIÓN DE VARIABLES

Ámbito de declaración de una variable

- El ámbito de una variable define desde dónde se puede acceder al valor dicha variable:
 - Variables **locales**: dentro de una función
 - Variables **globales**: desde todo el programa
- Variables Locales:
 - Se definen dentro de una función.
 - Sólo son visibles desde la función en la que están definidas.
 - Pueden definirse como:
 - Automáticas: Se crean cuando se llama a la función y se destruyen cuando la función acaba
 - Estáticas: El valor de la variable perdura de una ejecución de la función a otra - **No está en el temario de esta asignatura** –

Ámbito de declaración de una variable

- Variables Globales:
 - Se definen fuera de las funciones.
 - Son accesibles desde cualquier función
- Hay que **evitar el uso de variables globales dentro de las funciones** que componen el programa
 - El uso de variables globales en las funciones:
 - Reduce la legibilidad del programa y dificulta su modificación y depuración
 - Hace más difícil seguir los cambios que se producen en los valores de las variables, se cometen más errores
 - Impide que las funciones sean reutilizables en otro programa
 - Siempre que se tenga que compartir información entre funciones las variables se pasarán como parámetros

- 5.1. PROGRAMACIÓN MODULAR
- 5.2. IMPLEMENTACIÓN DE FUNCIONES
- 5.3. LLAMADA A FUNCIONES
- 5.4. PASO DE PARÁMETROS A UNA FUNCIÓN: POR VALOR Y POR REFERENCIA
- 5.5. ÁMBITO DE DECLARACIÓN DE VARIABLES. VISIBILIDAD
- 5.6. BIBLIOTECAS DE FUNCIONES
- 5.7. ANEXOS
 - 3.7.1. BIBLIOTECAS ESTÁNDAR DE C
 - 3.7.2. BIBLIOTECAS DE FUNCIONES EN DEV-C++

5.6. BIBLIOTECAS DE FUNCIONES

Bibliotecas de funciones

- El programa principal y los subprogramas pueden estar en un mismo fichero de código C o en diferentes
- Agrupar funciones en un fichero independiente facilita su reutilización: Bibliotecas de funciones
- Para ello hay que crear dos ficheros
 - cabecera (*.h)
 - fuente (*.c)
- Y luego incluir el fichero que contiene las funciones en el programa

```
#include "funciones1.h"
```

```
int main (void) {  
    . . .  
    return (0);  
}
```

Fichero cabecera que
contiene la declaración de
ciertas funciones

Bibliotecas de funciones:

- El fichero cabecera (.h) incluye:
 - La definición de los tipos de datos asociados a las funciones (estructuras, tema siguiente)
 - Los prototipos de las funciones incluidas en el módulo
- El fichero fuente (.c) incluye:
 - Definición las funciones declaradas en el fichero cabecera.

RESUMEN Y PUNTOS A RECORDAR



Resumen:

- Una función es un fragmento de código independiente que se encarga de resolver una determinada tarea.
- Las funciones siempre devuelven un único valor
 - puede ser del tipo void
- parámetros **formales** – definición de la función
- parámetros **reales** - llamada a la función
- Si una función no acepta parámetros indica con la palabra reservada void en la definición
 - Cuando se llama a la función se utilizan los paréntesis sin nada entre ellos.

Resumen:

- Una función finaliza su ejecución cuando llega al final o cuando se ejecuta dentro de ella la sentencia **return**.
- es recomendable incluir siempre una sentencia return
 - No es obligatorio
- Mediante return, una función puede devolver un único valor a la función que la llamó.

Puntos a recordar:

- El prototipo de una función incluye el nombre de la función, su tipo y los parámetros que acepta finalizando con un punto y coma.
- Un parámetro que se pasa **por valor** a una función no resultará modificado una vez terminada la ejecución de la función
- Cuando una función debe modificar el valor del parámetro pasado y **devolver este valor modificado** se debe utilizar el paso de parámetros **por referencia**:
 - El parámetro real irá precedido del símbolo **&**.
 - El parámetro formal será un **puntero**.
- Una variable local es una variable que sólo puede ser accedida dentro de la función en la que se define.
 - Se deben usar variables locales para los datos que el programa principal no necesita conocer

- 5.1. PROGRAMACIÓN MODULAR
- 5.2. IMPLEMENTACIÓN DE FUNCIONES
- 5.3. LLAMADA A FUNCIONES
- 5.4. PASO DE PARÁMETROS A UNA FUNCIÓN: POR VALOR Y POR REFERENCIA
- 5.5. ÁMBITO DE DECLARACIÓN DE VARIABLES. VISIBILIDAD
- 5.6. BIBLIOTECAS DE FUNCIONES
- 5.7. ANEXOS
 - 3.7.1. BIBLIOTECAS ESTÁNDAR DE C
 - 3.7.2. BIBLIOTECAS DE FUNCIONES EN DEV-C++

5.7.1 BIBLIOTECAS ESTÁNDAR DE C

Bibliotecas estándar en C

- Todas las versiones de C ofrecen una biblioteca estándar de funciones que proporciona soporte para las operaciones que se realizan con más frecuencia.
- Estas funciones permiten realizar una operación con sólo una llamada a la función (sin necesidad de escribir su código fuente).
- Las funciones estándar o predefinidas se dividen en grupos:
 - Todas las funciones que pertenecen al mismo grupo se declaran en el mismo archivo cabecera.
 - Se pueden incluir tantos archivos de cabecera como sean necesarios.
- Ya hemos visto algunos ejemplos
 - `stdio.h`, `math.h`, `string.h`

Bibliotecas estándar en C: Ejemplos

- Las más utilizadas

<complex.h>	Funciones relacionadas con la aritmética de complejos
<ctype.h>	Manipulación de caracteres
<errno.h>	Permite controlar errores
<float.h>	Añade funcionalidades a los tipos de coma flotante
<math.h>	Funciones numéricas
<stdio.h>	Operaciones de Entrada/Salida (standard input output - io)
<string.h>	Manipulación de cadenas de caracteres (string)
<time.h>	Funciones de fecha y hora
<stdlib.h>	Funciones de valor absoluto, generación de números aleatorios, búsqueda y ordenación, conversión de cadenas, gestión de memoria y comunicación con el entorno de ejecución

<stdio.h>

- *stdio* define varios tipos, macros y funciones necesarias para leer e imprimir valores.
- Las funciones y macros más utilizadas son:
 - Macro **getchar**
 - Prototipo: `int getchar(void);`
 - Descripción: Devuelve el carácter introducido por teclado.
 - Función **gets**
 - Prototipo: `int *gets (char *cadena);`
 - Descripción: Devuelve la cadena de caracteres introducida por teclado.
 - Macro **putchar**
 - Prototipo: `int putchar(int c);`
 - Descripción: Muestra por pantalla el carácter pasado como parámetro.
 - Función **puts**
 - Prototipo: `int puts(const char *cadena);`
 - Descripción: Muestra por pantalla una cadena de caracteres
 - Función **printf**
 - Prototipo: `int printf(const char *formato, ...);`
 - Descripción: Imprime por **pantalla** según el formato pasado como parámetro.
 - Función **scanf**
 - Prototipo: `int scanf(const char *formato, ...);`
 - Descripción: Lee de **teclado** los elementos indicados en el formato y los almacena en los siguientes parámetros (que deberán ser pasados por referencia)

<stdio.h>

- Funciones relacionadas con la E/S de ficheros:
 - Función **fprintf**
 - Prototipo: `int fprintf(FILE *stream, const char *formato, ...);`
 - Descripción: Imprime en *fichero* según el formato pasado como parámetro.
 - Función **fscanf**
 - Prototipo: `int fscanf(FILE *stream, const char *formato, ...);`
 - Descripción: Lee de *fichero* los elementos indicados en el formato y los almacena en los siguientes parámetros.
 - Función **fopen**
 - Prototipo: `FILE *fopen(const char *nombre, const char *modo);`
 - Descripción: Abre un nuevo archivo y devuelve un *stream* asociado. El primer parámetro representa el nombre del archivo y el segundo el modo de apertura.
 - Función **fclose**
 - Prototipo: `int *fclose(FILE *stream);`
 - Descripción: Realiza todas las escrituras pendientes y cierra el archivo asociado al *stream*.

<stdlib.h>

- *stdlib* define varios tipos, macros y funciones relacionadas con:
 - Conversión de cadenas de caracteres
 - Generación de números aleatorios
 - Gestión de memoria
 - Comunicación con el entorno de ejecución
 - Búsqueda y ordenación
- Algunas de las funciones más utilizadas son:
 - Función **atof**
 - Prototipo: `double atof (const char *nprt);`
 - Descripción: Transforma la cadena de caracteres pasada como parámetro a su valor *double* correspondiente
 - Función **atoi**
 - Prototipo: `int atoi (const char *nprt);`
 - Descripción: Transforma la cadena de caracteres pasada como parámetro a su valor entero correspondiente

<stdlib.h>

- Función **rand**
 - Prototipo: `int rand (void);`
 - Descripción: Devuelve un número aleatorio entre 0 y RAND_MAX
- Función **srand**
 - Prototipo: `void srand (unsigned int seed);`
 - Descripción: Indica la semilla inicial para la secuencia de números aleatorios generados al llamar a la función rand.
- Función **malloc**
 - Prototipo: `void *malloc (size_t size);`
 - Descripción: Reserva una zona de memoria de un tamaño de bytes indicado como parámetro y devuelve la dirección de comienzo de la misma.
- Función **realloc**
 - Prototipo: `void *realloc (void *prt, size_t size);`
 - Descripción: Cambia el tamaño de la zona apuntada por el puntero pasado como parámetro para pasar a ser del tamaño indicado en el segundo parámetro.
- Función **free**
 - Prototipo: `void free (void *prt);`
 - Descripción: Libera la memoria dinámica apuntada por *prt*

<string.h>

- `string.h` define las funciones utilizadas en el manejo de cadenas de caracteres (`string`).
- Algunas de las funciones más utilizadas son:
 - Función **strlen**
 - Prototipo: `unsigned strlen (const char *s);`
 - Utilidad: Contar el número de caracteres de una cadena.
 - Función **strcat**
 - Prototipo: `char *strcat (char *s1, const char *s2);`
 - Utilidad: Unir dos cadenas de caracteres poniendo `s2` a continuación de `s1`. La cadena resultante se almacena en `s1`.
 - Función **strcmp**
 - Prototipo: `int strcmp (const char *s1, const char *s2)`
 - Utilidad: Compara dos cadenas de caracteres. Devuelve 0 si las cadenas son iguales, un valor `<0` si `s1` es menor (en orden alfabético) que `s2`, y un valor `>0` si `s1` es mayor que `s2`.
 - Función **strcpy**
 - Prototipo: `char *strcpy (char *s1, const char *s2)`
 - Utilidad: Copia en `s1`, la cadena almacenada en `s2`

<math.h>

- *math* define diferentes macros y funciones matemáticas.
- Las funciones más utilizadas son:
 - Funciones **ceil** y **floor**
 - Prototipo: `double ceil (double x);`
 - Descripción: Redondea por exceso (por defecto) al entero más próximo.
 - Función **fabs**
 - Prototipo: `double fabs (double x);`
 - Descripción: Calcula el valor absoluto de un número.
 - Función **fmod**
 - Prototipo: `double fmod (double x, double y);`
 - Descripción: Devuelve el resto de la división de x entre y.
 - Función **sqrt**
 - Prototipo: `double sqrt (double x);`
 - Descripción: Calcula la raíz cuadrada de un número.
 - Función **pow**
 - Prototipo: `double pow (double x, double y);`
 - Descripción: Devuelve el resultado de elevar x a y.

<ctype.h>

- *ctype* incluye funciones que permiten la clasificación y conversión de caracteres:
- Las funciones más utilizadas son:
 - Función **isalnum**
 - Prototipo: `int isalnum (int c);`
 - Descripción: Devuelve verdadero (valor numérico distinto de cero) si el parámetro es una letra o un dígito.
 - Función **isctrl**
 - Prototipo: `int isctrl (int c);`
 - Descripción: Devuelve *verdadero* si *c* es un carácter de control.
 - Función **isdigit**
 - Prototipo: `int isdigit (int c);`
 - Descripción: Devuelve *verdadero* si *c* es un dígito.
 - Función **tolower**
 - Prototipo: `int tolower (int c);`
 - Descripción: Devuelve el carácter en minúscula correspondiente al carácter pasado por parámetro.
 - Función **toupper**
 - Prototipo: `int toupper (int c);`
 - Descripción: Devuelve el carácter en mayúscula correspondiente al carácter pasado por parámetro.

<complex.h>

- *complex* define las macros y funciones necesarias para implementar la aritmética de números complejos.
- Las funciones más utilizadas son:
 - Función **cabs**
 - Prototipo: `double cabs(double complex z);`
 - Utilidad: Calcula el valor absoluto de un número complejo.
 - Función **cimag**
 - Prototipo: `double cimag(double complex z);`
 - Utilidad: Devuelve la parte imaginaria de un número complejo.
 - Función **creal**
 - Prototipo: `double creal(double complex z);`
 - Utilidad: Devuelve la parte real de un número complejo
 - Función **csqrt**
 - Prototipo: `double complex csqrt(double complex z);`
 - Utilidad: Calcula la raíz cuadrada de un complejo.
 - Funciones **ccos** y **csin**
 - Prototipo: `double complex ccos(double complex z);`
 - Utilidad: Calcula el coseno (seno) complejo de z

Funciones de biblioteca: Ejemplo de uso

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

Archivo de cabecera
para operaciones E/S

```
#define TAM_CADENA 80
```

```
int main (void)
{
```

```
    //Declaración de Variables:
```

```
    char nombre[TAM_CADENA];
    char apellidos[TAM_CADENA];
    char nombreCompleto[TAM_CADENA*2];
```

Archivo de
cabecera para
trabajar con
cadenas

```
printf ("Introduzca su nombre: \n");
gets(nombre);
```

```
printf ("Introduzca sus apellidos: \n");
gets(apellidos);
```

```
/* Se almacena en nombreCompleto el nombre y los apellidos*/
```


Funciones de biblioteca: Ejemplo de uso

```
/* inicia nombreCompleto a la cadena vacía */
strcpy (nombreCompleto, "");

/*concatena el nombre*/
strcat(nombreCompleto, nombre);

/*concatena un espacio en blanco*/
strcat(nombreCompleto, " ");

/*concatena los apellidos*/
strcat(nombreCompleto, apellidos);

/*Se imprime el nombre completo*/
printf("Su nombre es: %s\n", nombreCompleto);

return 0;
}
```

Funciones de biblioteca: Resumen

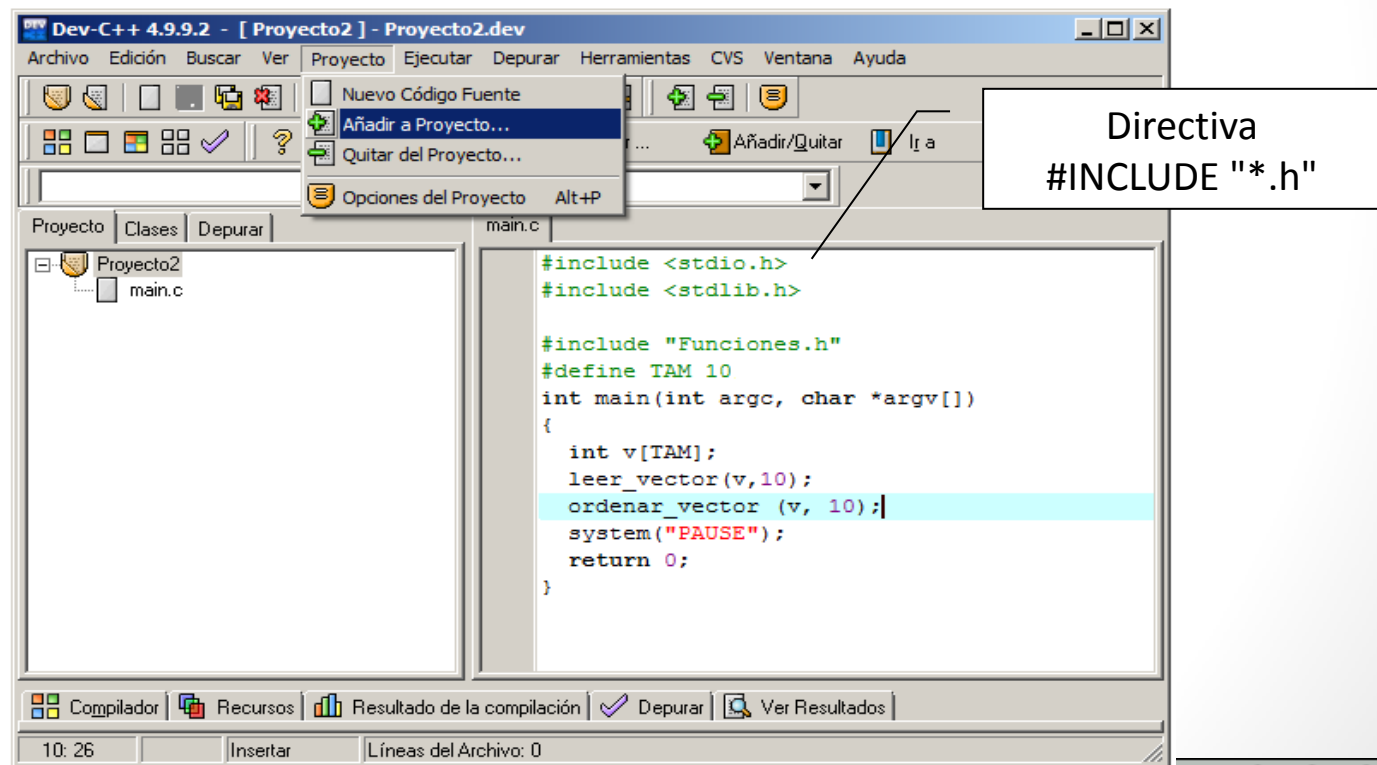
Función	Tipo	Propósito	lib
abs(i)	int	Devuelve el valor absoluto de i	stdlib.h
fmod(d1, d2)	double	Devuelve el resto de la división d1/d2 (con el signo de d1)	math.h
sqrt(d)	double	Devuelve la raíz cuadrada de d	math.h
atoi(s)	long	Convierte la cadena s en un entero	stdlib.h
atof(s)	double	Convierte la cadena s en un número de doble precisión	stdlib.h
floor(d)	double	Devuelve el valor entero equivalente al redondeo por defecto de d	math.h
ceil(d)	double	Devuelve el valor entero equivalente al redondeo por exceso de d	math.h
exp(d)	double	Función exponencial: e elevado a d	math.h
log(d)	double	Devuelve el logaritmo natural de d	math.h
rand(void)	int	Devuelve un valor aleatorio positivo	math.h
sin(d)	double	Seno de d (en radianes)	math.h
cos(d)	double	Coseno de d (en radianes)	math.h
tan(d)	double	Tangente de d (en radianes)	math.h
asin(x)	double	Arco seno de x	math.h
acos(x)	double	Arco coseno de x	math.h
printf(..)	int	Escribe datos en pantalla	stdio.h
scanf(..)	int	Lee datos de teclado	stdio.h
strcpy(s1,s2)	char*	Copia la cadena s2 en la cadena s1	string.h
strlen(s1)	int	Devuelve el número de caracteres de la cadena s1	string.h

- 5.1. PROGRAMACIÓN MODULAR
- 5.2. IMPLEMENTACIÓN DE FUNCIONES
- 5.3. LLAMADA A FUNCIONES
- 5.4. PASO DE PARÁMETROS A UNA FUNCIÓN: POR VALOR Y POR REFERENCIA
- 5.5. ÁMBITO DE DECLARACIÓN DE VARIABLES. VISIBILIDAD
- 5.6. BIBLIOTECAS DE FUNCIONES
- 5.7. ANEXOS
 - 3.7.1. BIBLIOTECAS ESTÁNDAR DE C
 - 3.7.2. BIBLIOTECAS DE FUNCIONES EN DEV-C++

5.7.2 BIBLIOTECAS DE FUNCIONES EN DEV-C++

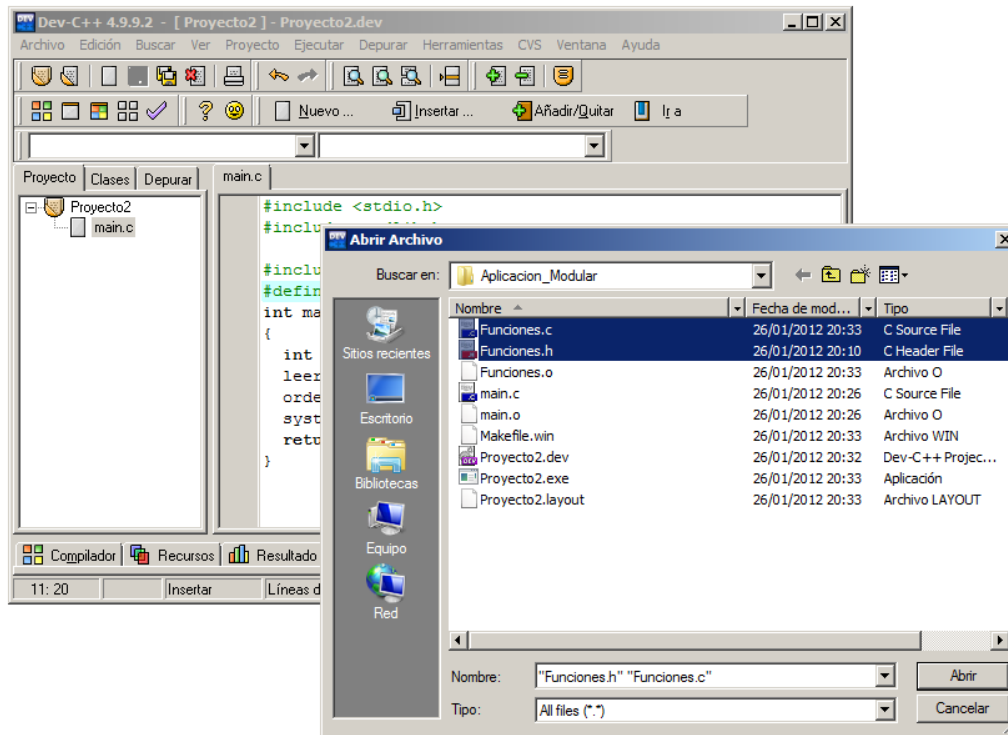
Aplicaciones modulares en el entorno Dev-C++

1. Crear un nuevo Proyecto
2. Añadir ficheros fuente a un proyecto:
 - Proyecto -> Nuevo Código Fuente (si el fichero fuente no existe)
 - Proyecto -> Añadir a Proyecto (si el fichero existe)



Aplicaciones modulares en el entorno Dev-C++

3. Seleccionar los ficheros fuente que componen el módulo



Después de añadir los ficheros, aparecerá una pestaña por cada uno de ellos

Aplicaciones modulares en el entorno Dev-C++

4. Escribir el código del módulo principal (sólo extensión .c)
 - Incluir la directiva **#INCLUDE "fichero.h"**
5. Compilar:
 - ✓ **Ejecutar -> Compilar:** Compila sólo aquellos ficheros que han sido modificados tras la última compilación.
 - ✓ **Ejecutar -> Compila el archivo actual:** Compila sólo el fichero actual
 - ✓ **Ejecutar -> Reconstruir todo:** Realiza una compilación completa del proyecto:

Si no se dispone de los ficheros fuente, el paso 2 (con la opción "Nuevo Código Fuente") se repetirá dos veces: una para el fichero .h y otra para el fichero .c

- Guardamos los ficheros en el directorio de trabajo
- Compilamos el proyecto

TEMA 5. FUNCIONES

Grado en Ingeniería en Tecnologías Industriales
Programación

