Departamento de Lenguajes y Ciencias de la Computación Universidad de Málaga

TEMA 5: Funciones

Fundamentos de Informática (Ingeniería Técnica Industrial)

E.U. Politécnica

Autores: M.C. Aranda, A. Fernández, J. Galindo, M. Trella

Índice de contenidos

Utilidad de los subprogramas

- Los programas suelen ser suficientemente **largos** y **complejos**, como para que en su solución tenga que aplicarse la técnica de la **Programación Estructurada:**
  - **Dividir el programa en subprogramas** (orutinas) máselementales de forma que su solución sea más simple.
  - Un **subprograma** es un segmento de código que resuelve una parte del problema global.
    - Un **subprograma** resuelve un problema más o menos concreto, y puede ser tan simple o complejo como sea necesario.
    - Un **subprograma** puede, a su vez, dividirse en otros subprogramas más simples.
    - Un **subprograma** puede utilizarse siempre que sea necesario, de forma que si tenemos que ejecutarlo varias veces sólo es necesario escribirlo una vez.
    - Un **subprograma**, en ocasiones, puede reutilizarse en otro programa distinto, por lo que reduce el trabajo.
  - Un **programa sin subprogramas** es más laborioso de escribir, más largo, más difícil de modificar, más difícil de detectar errores...

Subprogramas en C: Funciones

- Algunos lenguajes de programación tienen dos **tipos de Subprogramas** (procedimientos y funciones).
- **Los Subprogramas en lenguaje C son sólo del tipo funciones.**
- Un **programa en C** está estructurado en **funciones**.
  - De hecho, **`main()` es una función**, es la función principal.
- Las **Funciones en Programación** son similares a las funciones matemáticas, pudiendo tener argumentos o parámetros. **Ejemplo:**
  - **Definición** de una función en matemáticas: **f(x) = x + 5;**
    - **f** es el **nombre de la función** que tiene un **argumento formal**: **x**.
    - La función **f**, una vez definida, puede usarse o **llamarse** en algunas expresiones: **y = f(4) + 1;**
      - Para descubrir el valor de **f(4)** hay que sustituir el **argumento formal x** por el valor **4** (**argumento actual**), **f(4) = 4+5 = 9**, de donde obtenemos que **y = 9 + 1 = 10**.
      - Se dice que la función **f devuelve** el valor **9**.
- Las **Funciones en Programación** son más generales que las funciones matemáticas: Pueden tener argumentos de cualquier tipo y pueden incorporar estructuras de control.

Subprogramas en C: Funciones

- Las **Funciones en Programación** también tienen todos los elementos vistos en las funciones matemáticas:
  - **Definición:** Donde se define su **nombre**, los argumentos que va a tener (nombre, número y tipo) y lo que la función debe hacer o cómo calcular el valor que **devuelve** la función. También se define el **tipo** del valor que se devuelve (**tipo de la función**).
  - **Argumentos formales:** Son los argumentos que se ponen en la definición de una función.
  - **Llamada a una función:** Es la instrucción donde se invoca o se utiliza la función.
  - **Argumentos actuales:** Son los argumentos utilizados en la llamada.
- **Ejemplo:** Implementar en **C** la función **f** anterior:

Nombre de la función — **Argumento formal y su tipo**

Tipo del valor que devuelve la función (tipo de la función): Debe coincidir con el tipo de la expresión que haya después de la palabra **return**.

```
int f (int x){
   return x + 5;
}
```

Valor que devuelve la función

Definición de una Función: Formato

```
TipoDevuelto NombreFunción ([ArgumentosFormales]){
  Bloque_de_instrucciones;
}
```

- **Observaciones:**
  - **ArgumentosFormales:** Es una lista con la declaración de todos los argumentos formales, separados por comas. Puede estar vacía.
  - **TipoDevuelto:** Es el tipo del valor que la función va a devolver.
  - **Bloque_de_instrucciones:**
    - Es la **implementación** de la función.
    - Pueden declararse **variables LOCALES** a la función.
    - Puede contener la sentencia **return** para devolver el valor que corresponda.
      - Formato: **return <expresión>;**
      - Esta sentencia puede aparecer varias veces.
      - En cuanto se ejecute una sentencia **return** la función termina, devolviendo el valor que haya en esa sentencia **return**.
      - El valor devuelto debe ser del tipo especificado (**TipoDevuelto**).

Ejemplo: Una función *(definición y llamada)*

- Las funciones se pueden definir **antes** de la función **main()**.
- En un programa podemos **definir** una función una vez y **utilizarla** (o llamarla) siempre que sea necesario.
  - **La llamada a una función puede utilizarse en cualquier lugar en el que pueda utilizarse una expresión del tipo de la función.**
- **Ejemplo:** ¿Qué **salida** produce el siguiente programa para un determinado valor de **entrada**?

```
#include<stdio.h>

int f (int x){
  return x + 5;
}

void main(){
  int a,b;
  printf("\n- Introduzca un número: ");
  scanf("%i",&a);
  b = f(a) + a - f(5);
  a = f(b+1);
  printf("\n- Valores: %i, %i y %i.", a, b, f(a-b));
}
```

Ejemplo de Ejecución:

```
- Introduzca un número: 3

- Valores: 7, 1 y 11.
```


Ejemplos de Funciones

```
/* Factorial de n (n!). Devuelve -1 si hay error */
long int Factorial (int n){
  unsigned long int factorial=1;
  if (n<0)
     return -1; /* Error */
  else { for ( ; n>1; n--)
           factorial = factorial * n;
         return factorial;
  }
}
/* Números Combinatorios. Devuelve -1 si n<m */
long int Combinatorios(int m, int n){
  if (n<m)
     return -1;
  return Factorial(n) / (Factorial(m) * Factorial(n-m));
}
```

- Número de partes de *m* elementos que se pueden tomar de un conjunto de ***n*** elementos:

$$C_n^m = \frac{n!}{m!(n-m)!}$$

- **Variables LOCALES:** Son aquellas que se declaran **dentro** de una función. Las variables de sus **argumentos formales** son **locales**.
  - Sólo tienen sentido y sólo pueden usarse **dentro** de esa función.
  - Su **nombre** es totalmente **independiente** de las variables locales de otras funciones, incluyendo la función **main()**.
    - Observe las variables **n** de las funciones anteriores: Se llaman igual pero son **variables distintas: Tienen distinto *ámbito* (*visibilidad*), distinto *significado*, distinta *utilidad*...**

Prototipo de una Función: Formato

```
TipoDevuelto  NombreF  ([Tipo_Argumentos_Formales]);
```

- **Observaciones:**
  - **`Tipo_Argumentos_Formales`:** Es una lista con la declaración de todos los argumentos formales, separados por comas. Pueden ponerse sólo los tipos (sin el nombre de las variables).
  - **Prototipo**, declaración de una función o **cabecera** (*header*)**:** Sirve para especificar o declarar que existe una función con cierto número y tipo de argumentos y que devuelve cierto tipo de datos.
    - Esos datos son los únicos imprescindibles para poder utilizar la función (para poder llamarla).
    - El **prototipo** no define el proceso que se realiza, el cual se especifica en la **definición** de la función.
    - Por supuesto, el prototipo de una función debe ser coherente con la cabecera de su definición, donde quiera que ésta sea definida.
- **Ejemplo:** Prototipo de la función **`Combinatorios`:**

```
long int Combinatorios ( int m, int n);
```


¿Dónde definir las funciones en C?

- **En un programa en C las funciones pueden incluirse de dos formas distintas:**
  - **1. Antes de la función `main()`:**
    - En este caso se definirá la función **`main()`** al final.
    - Las funciones se ordenarán teniendo en cuenta que para utilizar una función ésta debe estar definida previamente.
  - **2. Después de la función `main()`:**
    - En este caso se definirá la función **`main()`** al principio.
    - Antes de la función **`main()`** se incluirán los prototipos de todas las funciones.
      - Esto es imprescindible, ya que, al estar definidas las funciones al final, se desconocerían sus características. Los prototipos declaran estas características y hacen que las funciones puedan utilizarse.
    - El orden de las funciones ya no es importante, ya que todas podrán utilizar a todas las demás, puesto que están todos los prototipos declarados previamente.
    - Esta segunda forma es más laboriosa pero evita errores en el orden de las funciones. Además, el tener todos los prototipos juntos simplifica localizar las características de cada función.

Funciones de tipo `void`

- Puede definirse una **Función que no devuelva ningún valor:** El tipo de esta función debe ser **`void`**.
  - Estas funciones pueden contener la sentencia **`return`**, pero sin ir seguida de una expresión.
    - En cuanto se ejecuta la sentencia **`return`**, la función termina su ejecución (sin devolver nada, por supuesto).
    - Este uso de la sentencia **`return`** es desaconsejado, especialmente en programadores principiantes, debido a que complica la comprensión de la función al poder existir varias formas de que ésta finalice.
- **Ejemplo:**

```
/* Escribe una línea de n caracteres ch */
void Linea (unsigned n, char ch){
  unsigned i;

  for (i=1; i<=n; i++)
    putchar(ch);
  putchar('\n');
}
```

  - **Llamadas** en un fragmento de programa:

```
...
scanf("%i", &longitud);
Linea (longitud, '*');
...
Linea (10, '#');
...
```

- La palabra **`void`** también se usa cuando una función no tiene argumentos. Ejemplo: **`int func1(void){...`**

Paso de Argumentos POR VALOR

- Cuando se produce la **Llamada a una Función**, se **transfiere la ejecución** del programa a la definición de la función. **Pasos:**
  - **1. Se declaran las variables de los argumentos formales.**
  - **2. Se COPIA el VALOR de los argumentos actuales en las variables de los argumentos formales.**
    - **Esta COPIA se hace por orden:** El primer argumento actual en el primer argumento formal, el segundo en el segundo...
    - **Esta COPIA NO SE HACE POR EL NOMBRE** de los respectivos argumentos formales y actuales.
    - Observe que se produce una **COPIA del valor:** Si el argumento actual es una variable, se copia su valor en el correspondiente argumento formal, pero ambos **argumentos actuales y formales son variables DISTINTAS.**
  - **3. Se declaran las variables locales a la función.**
  - **4. Se ejecuta el código de la función.**
  - **5. Al terminar la función las variables LOCALES son destruidas.**
- **Si los ARGUMENTOS FORMALES se MODIFICAN dentro de la FUNCIÓN, NO se MODIFICARÁN los ARGUMENTOS ACTUALES: ¡ SON VARIABLES DISTINTAS !**

Paso de args. POR VALOR: Ejemplo

- **Ejemplo:** ¿Qué **salida** produce el siguiente programa para distintas **entradas**?

```
#include<stdio.h>
float func (float x, float y){
  x = x + 1;
  y = y + 2;
  return x - y;
}
void main(){
  float x,y,z;
  printf("\n- Introduzca un número: ");
  scanf("%f",&x);
  y = x + x;
  z = func(y,x);
  x = func(y,z);
  func(x,y);
  printf("\n- Valores: %.1f, %.1f y %.1f.", x, y, z);
}
```

EjemplodeEjecución:

```
- Introduzca un número: 3

- Valores: 3.0, 6.0 y 2.0.
```

- Observe que la llamada **`func(x,y)`** no efectúa ninguna operación con el valor que devuelve la función **`func`**, por lo que ese valor se pierde (pero el programa no genera ningún error). En este ejemplo, como los argumentos pasan por valor y la función no hace nada especial, esta llamada es absurda y no modifica en nada la ejecución del programa.

Paso de Argumentos Por REFERENCIA

- **RECUERDA:** En un **paso de argumentos por VALOR**, si los arg. formales se modifican, los arg. actuales NO cambian.
  - De hecho, los arg. actuales pueden ser constantes (que NO pueden cambiar).
- Sin embargo, a veces resulta muy útil **modificar**, en una función, variables de los **argumentos actuales**. Esto se consigue usando el **Paso de Argumentos POR REFERENCIA:**
  - Esto es otra forma de conseguir que una función devuelva valores (aparte de usando **`return`**).
  - Además, de esta forma una función puede devolver tantos valores como se deseen.
  - **En lenguaje C, TODOS los pasos de argumentos son por VALOR.**
  - Se llama **Paso de Argumentos POR REFERENCIA** a una técnica que permite a una función modificar variables utilizadas como argumentos actuales.
  - Primero explicaremos cómo aplicar esta técnica y posteriormente veremos por qué funciona y por qué esta técnica se basa en un paso de argumentos **por VALOR** (el **único** posible en **C**).

Paso de Argumentos Por REFERENCIA

- **TÉCNICA** para usar un **paso de argumentos por REFERENCIA:**
  - **1. En la Llamada a la Función:**
    - Usar el **OperadordeDirección** **&** (*ampersand*) delante de la variable para la que se desea el paso de arg. por referencia.
      - **Ejemplo: `Func1(&a);`** (la variable **a** pasa por referencia)
    - Recuerde que el **operador &** indica, por tanto, que dicha variable puede verse **modificada** por la función.
      - Esta es la razón por la que se usa el **operador &** en las variables leídas por la **función `scanf()`:** Como sabe, esta función lee un valor y modifica la variable en cuestión asignándole el valor leído.
  - **2. En la Definición de la Función:**
    - Usar el **Operador de Indirección** ***** (asterisco) delante del argumento formal correspondiente.
    - Se usará el **operador *** cada vez que la variable del argumento formal sea utilizada, tanto en la declaración de la variable (entre los paréntesis de la función) como en el cuerpo de la función.
    - No debe confundirse con el operador de multiplicación.
      - **Ejemplo: `*a = *a+7;`** (la variable del arg. formal se incrementa)

Paso de args. POR REFERENCIA: Ejemplo

- **Ejemplo:** ¿Qué **salida** produce el siguiente programa para distintas **entradas**?

```
#include<stdio.h>
void f2 (int *x, char *ch){
  *x = *x + 5;
  if (*ch == 'A')
     *ch = 'P';
  else
     *ch = 'K';
}
void main(){
  int x, y=1;
  char letra='A';
  printf("\n- Introduzca un número: ");
  scanf("%i",&x);
  y = y + x;
  f2(&y,&letra);
  printf("\n- Valores: %i, %i y %c.", x, y, letra);
  f2(&x,&letra);
  printf("\n- Valores: %i, %i y %c.", x, y, letra);
}
```

EjemplodeEjecución:

```
- Introduzca un número: 3

- Valores: 3, 9 y P.
- Valores: 8, 9 y K.
```

Paso de args. POR REFERENCIA: Otro Ejemplo

- **Ejemplo:** ¿Qué **salida** produce el siguiente programa?

```
#include<stdio.h>

/* Intercambio de valores entre dos variables */
void swap (int *x, int *y){
  int aux=*x;
  *x = *y;
  *y = aux;
}

void main(){
  int x=1, y=2, z=3;
  swap(&x,&y);
  swap(&y,&z);
  printf("\n- Valores: %i, %i y %i.", x, y, z);
}
```

– **Resultado:**

```
- Valores: 2, 3 y 1.
```


Introducción a los Punteros: Declaración

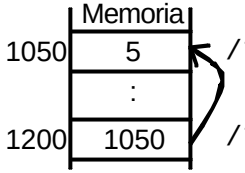
- **PUNTERO:** Variable que almacena una **dirección de memoria**.
- Una **variable** contiene siempre un valor (de un tipo). Un **puntero** contiene la **dirección** de una variable que contiene un valor.
  – **El valor de un puntero es una dirección de memoria.**
- Son importantes en **C** porque facilitan la programación y permiten obtener un código más compacto y eficiente.
- **DECLARACIÓN:** `tipo *nombre_variable;`
  – **La variable declarada es un "puntero al `tipo` de dato especificado": `nombre_variable` almacenará la dirección de memoria en la cual se almacenará un dato de ese `tipo`.**
  – **Con esa declaración se reserva memoria SÓLO para el puntero, NUNCA para la variable a la que apunta.**
- **EJEMPLOS:**

```
int *contPtr;       /* contPtr es un puntero a int */
float *res;           /* res es un puntero a float */
unsigned int *nosigno;   /* puntero a unsigned int */
char *mensaje;     /* mensaje es un puntero a char */
```


Introducción a los Punteros: Operadores

- **Operador de Dirección (de una variable): &** (*ampersand*)
  – **Operador unario que devuelve la dirección de su operando.**
  – **EJEMPLO:**

```
void main() {
    int y, *yPtr; /* yPtr es puntero a entero */
    y=5;
    yPtr=&y;   /* yPtr toma la dirección de y */
}
```




```
/* y tiene el valor 5 y está en la dirección 1050 */

/* yPtr tiene el valor 1050 y está en la dirección 1200 */
```

  – Se dice que una variable se refiere ***directamente*** a un valor y un puntero se refiere ***indirectamente*** a un valor.
  – El **puntero**, al tener una **dirección de memoria** es como si **apuntara** a dicha **dirección**.
  – **Recuerda:** Las direcciones de memoria de cada variable las asigna el S.O. (Sistema Operativo) y el programa no puede ni cambiarlas ni usar otras posiciones distintas a las asignadas por el S.O.

Introducción a los Punteros: Operadores

- **Operador de Indirección: *** (*asterisco*)
  – **Operador unario cuyo operando debe ser un puntero.**
  – **Devuelve el valor de la variable hacia la cual su operando apunta.**
  – **Es el operador opuesto a &: `*(&x)` es lo mismo que `x`.**
  – **EJEMPLO:**

Contenido de la dirección de **x**

```
void main() {
   int x, /* Entero */
       y, /* Entero */
       *py; /* Puntero */

   printf ("\n- Introduzca un número: ");
   scanf("%i",&y);
   py = &y;
   x = *py + 10; /* Suma 10 con el contenido de la dirección py */
   printf("\n- Enteros: %d, %d y %d.", y, *py, x);
   printf("\n- Direcciones: %p y %p.", &y, py);
}
```

EjemplodeEjecución:

```
- Introduzca un número: 3

- Enteros: 3, 3 y 13.
- Direcciones: FFF2 y FFF2.
```

  – **NOTA:** El modificador **`%p`** de **`printf()`** se utiliza para escribir un puntero.

Paso de args. POR REFERENCIA: Más Ejemplos

```
void PerimetroCirculo (double *r){
  *r = 2 * 3.141592 * *r; /* 2 PI R */
}
...
  double X;
...
  PerimetroCirculo(&X);
  printf("\n- El perímetro es: %f.", X);
...
```

Un paso de arg. por Referencia es un paso de arg. por Valor de la dirección de una variable.

- **Cuando se produce la llamada a `PerimetroCirculo()`, su argumento es la dirección de la variable `X` (esto es, `&X`).**
- **Lo que se pasa como argumento es dicha dirección, que se copia en el argumento formal `r` (**que es de un tipo compatible, puesto que es un puntero a **`double`**, como **`&X`).**
- **Ahora `r` vale la dirección de `X`, y `*r` es el contenido de dicha dirección, o sea `*r` es lo mismo que `X` (**ya que el contenido de la dirección de memoria de `X` es `X`).**
- **Dentro de la función se modifica `*r`, que es lo mismo que modificar `X`.**
- **RESUMEN: Al pasar a la función la dirección de `X` hacemos que la función pueda modificar su contenido usando, simplemente, el operador de indirección sobre su argumento formal (`*r`).**

Paso de args. POR REFERENCIA: Más Ejemplos

- **Función:**

```
void Potencia2y3 (int N, int *cuadrado, int *cubo){
  *cuadrado = N * N;
  *cubo = *cuadrado * N;
}
```

- **Llamada:**

```
...
  int a, a2, a3;

...
  printf ("\n- Introduzca un número: ");
  scanf("%i",&a);

  Potencia2y3(a,&a2,&a3);
  printf("\n- Valores: %i, %i y %i.", a, a2, a3);

  Potencia2y3(a2,&a,&a3);
  printf("\n- Valores: %i, %i y %i.", a, a2, a3);
...
```

Ejemplo de Ejecución:

```
- Introduzca un número: 3

- Valores: 3, 9 y 21.
- Valores: 81, 9 y 729.
```


Variables GLOBALES: Mejor NO usarlas

- Son variables que se declaran **fuera** de todas las funciones**:** Pueden ser utilizadas por todas las funciones que haya después de su declaración.
  - Normalmente, estas variables se declaran antes que las funciones, por lo que su **ámbito o visibilidad** es global**:** Pueden usarse en cualquier función.
  - **Su uso está desaconsejado, especialmente en programadores noveles**, porque complica la comprensión de los programas y pueden dar lugar a *efectos laterales* erróneos que suelen ser muy difícil de localizar.
  - Si hay **ambigüedad** entre **variables locales y globales** se usan las **locales**.

```
#include <stdio.h>
int a; /* Esta es una variable global */

int func1 (int a){
  return a*a;
}

int func2 (int x){
  int a=1;
  return a+x;
}

int func3 (int x){
  return a+x;
}
```

```
void main(){
  int x=3;
  a = x - 1;
  x=func1(a);
  a=func2(x);
  x=func3(a);
  printf("\n- Valores: %i y %i.",
         x, a);
}
```

Salida:

```
- Valores: 10 y 5.
```


Funciones: Ejemplos

- **Escribir una función para obtener cierto dígito de un número entero, que puede ser negativo.**

```
/***** ***** ***** ***** *****
Devuelve el dígito i-ésimo del número N.
Se cuentan de derecha a izquierda empezando en 0.
Devuelve -1 si N no tiene suficientes dígitos.
***** ***** ***** ***** *****/
int Digit (unsigned i, long N){
  if (N==0 && i==0)
     return 0;

  if (N<0)
     N = N * -1; /* Pasamos N a positivo */

  N = N / (long) pow(10,i);
  if (N==0)
     return -1;

  return N%10;
}
```

Función de `math.h`

División SIN decimales. Equivalente a (con `j` como **`unsigned`**):

```
for (j=1;j<=i;j++)
  N = N / 10;
```

Funciones: Ejemplos

- **Escribir una función que acepte dos caracteres como argumentos y que devuelva en el tercer argumento la suma de esos dos caracteres (como número entero positivo).**
  - **Ejemplo:** Para **'9'** y **'5'** (que son caracteres) calcularía el valor **14** (como número entero positivo).
  - La función devolverá (con **return**) cero si todo fue bien y uno si algún carácter no contiene dígitos numéricos.

```
/***** ***** ***** ***** *****
Suma caracteres, convirtiéndolos a enteros.
Devuelve:  0 si todo fue bien y
           1 si algún carácter no es numérico.
***** ***** ***** ***** *****/
int SumaChars (char ch1, char ch2, unsigned *suma){
  if (ch1<'0' || ch1>'9' || ch2<'0' || ch2>'9')
     return 1; /* ch1 o ch2 no son números */

  *suma = (ch1 - '0') + (ch2 - '0');
  return 0;
}
```


Estructura de un Programa en C

- **1. Comentarios:** En primer lugar se deben incluir unos comentarios aclarando qué hace el programa, requisitos, autor, fecha... y cuantas observaciones deseen hacerse.
- **2. Inclusión de las Bibliotecas:**
  - **Las del sistema** (como **stdio.h**, **math.h**, etc.) van entre ángulos: **<...>**.
  - **Las creadas por el programador** van entre comillas dobles: **"..."**.
- **3. Declaraciones Globales:** Variables y constantes globales (con **const**), constantes simbólicas del preprocesador (con **#define**) y definición de tipos de datos (con **typedef**).
  - Los **comentarios** explicando cada declaración son muy importantes.
  - **Como norma general, NO DEBEN USARSE VARIABLES GLOBALES**.
- **4. Prototipos de las Funciones.**
- **5. Implementación de las Funciones, incluida `main()`**, que puede ponerse la primera o la última. Las demás funciones deberían ponerse en el mismo orden que sus prototipos para que sea fácil localizarlas.
  - Antes de cada función debe incluirse un **comentario** indicando qué hace la función, significado de sus argumentos, requisitos que deben cumplir éstos y cualesquiera otras observaciones que deban tenerse en cuenta para usar dicha función. Es buena medida separar cada función con una línea de asteriscos como comentario.
  - Si se escriben antes las funciones que son llamadas por otras, poniendo **`main()`** al final, los prototipos no son estrictamente necesarios.