Programación de dispositivos electrónicos

Unidad 3
Funciones-Vectores-Strings

Héctor Albano Diego Codevilla

Funciones

Vamos a empezar a intentar descomponer los problemas en problemas más sencillos, que sean más simples de resolver. Esto nos puede suponer varias ventajas:

- Cada “programa sencillo" independiente será más fácil de programar, al realizar una función breve y concreta.
- El "programa principal" será más fácil de leer, porque no necesitará contener todos los detalles de cómo se hace cada cosa.
- Podremos repartir el trabajo, para que cada persona se encargue de realizar un “programa sencillo", y finalmente se integrará el trabajo individual de cada persona.

Estos programas sencillos son lo que suele llamar "subrutinas", "procedimientos" o "funciones". En el lenguaje C, el nombre que más se usa es el de funciones.

```
#include <stdio.h>
#include <stdlib.h>
void saludar(void); /* saludar()*/
int main(void)
{
saludar();
return 0;
}
void saludar(void) /* saludar()*/
{
printf("Bienvenido al primer programa\n");
printf("de ejemplo\n");
printf("de funciones\n");
}
```

Prototipo

Llamado

Definición

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

int cuadrado ( int n )
{
return n*n;
}
int main()
{
int numero;
int resultado;
numero= 5;
resultado = cuadrado(numero);

printf("El cuadrado del numero %d es %d",numero, resultado);

printf(" y el de 3 es %d", cuadrado(3));
return 0;
}
```

Definición

Valor devuelto por la función

Llamado

Llamado

Nota: como la función se definió antes no hace falta el prototipo

Variables locales y globales

- Las variables se pueden declarar dentro de un bloque o dentro de una función, en ese caso, las variables solo tendrán validez dentro del bloque o función. Fuera de los mismos no tendrán validez. Es lo que llamaremos "variables locales".

- Por el contrario, si declaramos una variable al comienzo del programa, fuera de todos los bloques de programa, será una "variable global", a la que se podrá acceder desde cualquier parte del archivo.

- En general, deberemos intentar que la mayor cantidad de variables posible sean locales (lo ideal sería que todas lo fueran). Así hacemos que cada parte del programa trabaje con sus propios datos, y ayudamos a evitar que un error en una parte del programa pueda afectar al resto.

```
#include <stdio.h>
#include <stdlib.h>
int sumar(int a, int b);

int suma=0;                    /*suma es una variable global*/
                               /*es valida para todo el archivo */
int main()
{
int op1, op2;          /* op1 y op2 son locales de la función main*/
                       /*en otra función del mismo archivo no son válidas*/

printf("Introduzca el operador 1\n");
scanf("%d", &op1);
printf("Introduzca el operador 2\n");
scanf("%d", &op2);
printf("%d mas %d vale %d\n", op1, op2, sumar(op1,op2));
return 0;
}
int sumar(int a, int b)
{
suma = a+b;          /*suma existe dentro de la función sumar*/
return suma;
}
```

```
#include <stdio.h>
#include <stdlib.h>
int potencia(int base, int exponente)
{
int temporal = 1;                    /* Valor que voy hallando */
for(int i=1; i<=exponente; i++)
{                                    /* Multiplico “exponente” veces */
temporal *= base;                    /* Y calculo el valor temporal */
}
return temporal;                     /* Devuelvo el valor de las multiplicaciones, */
}

int main()
{
int num1, num2;
printf("Introduzca la base: ");
scanf("%d", &num1);
printf("Introduzca el exponente: ");
scanf("%d", &num2);
printf("%d elevado a %d vale %d", num1, num2, potencia(num1,num2));
return 0;
}
```

En este caso, las variable "temporal" es local a la función "potencia‟ para "main" no existe, si la intentáramos usar obtendríamos un mensaje de error. La variable “i” solo existe dentro del bloque for, fuera del for no existe, si la intentáramos usar también obtendríamos un mensaje de error.
De igual modo, "num1" y "num2" son locales para "main": desde la función "potencia" no podemos acceder a su valor (ni para leerlo ni para modificarlo), sólo es posible desde "main".

- Para poder acceder a una variable global desde otro archivo se debe agregar el prefijo **extern** al declarar la variable en el otro archivo (la variable tiene el mismo nombre).

- Cualquier declaración de una variable puede tener el prefijo **static**. Las variables estáticas en C tienen las siguientes propiedades:

1) No pueden ser accedidas desde otro fichero. Por tanto, los prefijos extern y static no pueden ser utilizados en la misma declaración.
2) Mantienen su valor a lo largo de toda la ejecución del archivo independientemente del ámbito en el que estén definidas.
3) Al ser declarada se le asigna el valor deseado, si no se le asigna valor se la inicializa con 0.
4) Si una variable estática está declarada fuera de las funciones, será accesible únicamente por el código que le siga en el mismo fichero de su declaración.
5) Si una variable estática está declarada en una función, sólo será accesible desde esa función y mantendrá su valor entre ejecuciones de la función.

```
#include <stdio.h>
#include <stdlib.h>
int funcion(void)
{
  static int numero = 10; /* Variable estática */
  numero++;             /* Mantiene el valor de la ejecución anterior */
  return numero;
}

int main(void)
{
   /* Imprime el resultado de dos invocaciones a la función */
   printf("%d\n", funcion());  /* Imprime el número 11 */
   printf("%d\n", funcion());  /* ¡Imprime el número 12! */
   return 0;
}
```

Algunas funciones útiles:

Números aleatorios

En un programa de gestión o una utilidad que nos ayuda a administrar un sistema, no es habitual que podamos permitir que las cosas ocurran al azar. Para un juego sí suele ser conveniente que haya algo de azar, para que una partida no sea exactamente igual a la anterior.

Generar números al azar ("números aleatorios") usando C no es difícil. Si nos ceñimos al estándar ANSI C, tenemos una función llamada "rand()", que nos devuelve un número entero entre 0 y el valor más alto que pueda tener un número entero en nuestro sistema.
Generalmente, nos interesarán números mucho más pequeños (por ejemplo, del 1 al 100), por lo que "recortaremos" usando la operación módulo ("%", el resto de la división).

Vamos a verlo con algún ejemplo:
Para obtener un número del 0 al 9 haríamos x = rand() % 10;
Para obtener un número del 10 al 29 haríamos x = rand() % 20 + 10;
Para obtener un número del 1 al 100 haríamos x = rand() % 100 + 1;
Para obtener un número del 50 al 60 haríamos x = rand() % 11 + 50;

Los números que genera un ordenador no son realmente al azar, sino "pseudo-aleatorios", cada uno calculado a partir del siguiente.

Podemos elegir cual queremos que sea el primer número de esa serie (la "semilla"), pero si usamos uno prefijado, los números que se generarán serán siempre los mismos. Por eso, será conveniente que el primer número se base en el reloj interno del ordenador: como es casi imposible que el programa se ponga en marcha dos días exactamente a la misma hora (incluyendo milésimas de segundo), la serie de números al azar que obtengamos será distinta cada vez.
La "semilla" la indicamos con "srand", y si queremos basarnos en el reloj interno del ordenador, lo que haremos será srand(time(0)); antes de hacer ninguna llamada a "rand()".
Para usar "rand()" y "srand()", incluir la librería “stdlib“. Si queremos que la semilla se tome a partir del reloj interno se debe incluir "time"

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main()
{
int n;
srand(time(0));
n = rand() % 10 + 1;
printf("Un número entre 1 y 10: %d\n", n);
return 0;
}
```

Funciones matemáticas

Dentro del fichero de cabecera "math.h" tenemos acceso a muchas funciones matemáticas predefinidas en C, como:

acos(x): Arco coseno
asin(x): Arco seno
atan(x): Arco tangente
atan2(y,x): Arco tangente de y/x (por si x o y son 0)
ceil(x): El valor entero superior a x y más cercano a él
cos(x): Coseno
cosh(x): Coseno hiperbólico
exp(x): Exponencial de x (e elevado a x)
fabs(x): Valor absoluto
floor(x): El mayor valor entero que es menor que x
fmod(x,y): Resto de la división x/y
log(x): Logaritmo natural (o neperiano, en base "e")
log10(x): Logaritmo en base 10
pow(x,y): x elevado a y
sin(x): Seno
sinh(x): Seno hiperbólico
sqrt(x): Raíz cuadrada
tan(x): Tangente
tanh(x): Tangente hiperbólica

(todos ellos usan parámetros X e Y de tipo "double") y una serie de constantes como

M_E, el número "e", con un valor de 2.71828...
M_PI, el número "Pi", 3.14159...

La mayoría de ellas son específicas para ciertos problemas matemáticos, especialmente si interviene la trigonometría o si hay que usar logaritmos o exponenciales. Pero vamos a destacar las que sí pueden resultar útiles en situaciones más variadas:

Ej
El valor absoluto: si queremos trabajar sólo con números positivos usaríamos n = fabs(x);

Vectores

Definición de un vector y acceso a los datos

Una tabla, vector, matriz o array (o "arreglo") es un conjunto de elementos, todos los cuales son del mismo tipo. Estos elementos tendrán todos el mismo nombre, y ocuparán un espacio contiguo en la memoria.
Por ejemplo, si queremos definir un grupo de 4 números enteros, usaríamos

```
int ejemplo[4];
```

Podemos acceder a cada uno de los valores individuales indicando su nombre (ejemplo) y el número de elemento que nos interesa, pero con una precaución: se empieza a numerar desde 0, así que en el caso anterior tendríamos 4 elementos, que serían ejemplo[0], ejemplo[1],ejemplo[2], ejemplo[3].
Al igual que ocurría con las variables, podemos dar valor a los elementos de un vector al principio del programa. Los indicaremos todos entre llaves, separados por comas:

```
int numero[5] ={200, 150, 100, -50, 300};
int punto[] = {10, 0, -10};
```

Para acceder a varios elementos de un array, sin tener siempre que repetirlos todos como por ej:

```
suma = numero[0] + numero[1] + numero[2] + numero[3] + numero[4];
```

El "truco" consistirá en emplear cualquiera de las estructuras repetitivas que ya hemos visto

(while, do-while, for), por ejemplo así:

```
suma = 0;                    /* Valor inicial */
for (i=0; i<=4; i++)
suma += numero[i];
```

```
#include<stdio.h>
int main()
{
int datos[100];
int i=0;
int k;
        printf("Ingrese distintos de cero \n");
        scanf("%d",&datos[0]);
        while (datos[i]!=0 && i<=99)
        {
                i = i+1;
                scanf("%d",&datos[i]);
        }
        for (k=0;k<=i-1;k+=1)
        {
                printf("orden %d valor %d\n",k+1,datos[k]);
        }
        return 0;
}
```

Paso de Vectores como parámetros a una función:

Ejemplo:

```
float promedio(int a, float x[])          // Definición de la función
                                          // Note que se incluyen los corchetes
{
                    // calculo del promedio
                    // Note que el tamaño del vector ´x´ es pasado en ´a´
}

void main ()
{
int n;
float prom;
float lista[100];
…
prom = promedio(n, lista);  // Esta llamada pasa como parámetros
                            // actuales la longitud del vector y
                            // el vector. Note que no se incluyen los corchetes
…
}
```

Strings

Definición. Lectura desde teclado

Las cadenas de texto se crean como "arrays" de caracteres. Están formadas por una sucesión de caracteres terminada con un carácter nulo (\0), de modo que tendremos que reservar una letra más de las que necesitamos. Por ejemplo, para guardar el texto "Hola" usaríamos "char saludo[5]".

Este carácter nulo lo utilizarán todas las órdenes estándar que tienen que ver con manejo de cadenas: las que las muestran en pantalla, las que comparan cadenas, las que dan a una cadena un cierto valor, etc. Si no queremos usar esas funciones y sólo vamos a acceder letra a letra nos bastaría con "charsaludo[4]". Esto no es lo habitual.

```
#include <stdio.h>
int main()
{
char texto[40];          /* Para guardar hasta 39 letras */
printf("Introduce tu nombre: ");
scanf("%s", &texto);
printf("Hola, %s\n", texto);
return 0;
}
```

Para dar un valor inicial a una cadena de texto se puede hacer usando dos formatos distintos:

```
char nombre[50]= {'J','u','a','n'};
char nombre[50]="Juan";
```

Este último formato sólo se puede usar cuando se declara la variable, al principio del programa. Si ya estamos dentro del programa, deberemos usar necesariamente la orden "strcpy" para dar un valor a una cadena de texto.

• Si la cadena contiene espacios, se lee sólo hasta el primer espacio. Esto se puede considerar una ventaja o un inconveniente, según el uso que se le quiera dar. En cualquier caso, dentro de muy poco veremos cómo evitarlo si queremos.

• Podemos leer (o modificar) una de las letras de una cadena de igual forma que leemos o modificamos los elementos de cualquier tabla: el primer elemento será texto[0], el segundo será texto[1].

• Siendo estrictos, no hace falta el "&" en "scanf" cuando estamos leyendo cadenas de texto (sí para los demás tipos de datos).

Longitud de la cadena:

En una cadena lo habitual es que realmente no ocupemos todas las letras que podríamos llegar a usar. Si guardamos una cierta cantidad de letras menor al tamaño reservado (mas el carácter nulo que marca el final), tendremos posiciones que generalmente contendrán "basura" (lo que hubiera previamente en esas posiciones de memoria, porque el compilador las reserva para nosotros pero no las "limpia"). Si queremos saber cual es la longitud real de nuestra cadena tenemos dos opciones:

•Podemos leer la cadena carácter por carácter desde el principio hasta que encontremos el carácter nulo (\0) que marca el final.
•Hay una orden predefinida que lo hace por nosotros, y que nos dice cuantas letras hemos usado realmente en nuestra cadena. Es "strlen".

```
#include <stdio.h>
#include <string.h>
int main()
{
char texto[40];
printf("Introduce una palabra: ");
scanf("%s", texto);
printf("Has tecleado %d letras", strlen(texto));
return 0;
}
```

Como es de esperar, si escribimos "Hola", esta orden nos dirá que hemos tecleado 4 letras (no cuenta el \0 que se añade automáticamente al final).

Si empleamos "strlen", o alguna de las otras órdenes relacionadas con cadenas de texto que veremos en este tema, debemos incluir <string.h>, que es donde se definen todas ellas.

Entrada/salida para cadenas: gets, puts

Hemos visto que si leemos una cadena de texto con "scanf", se paraba en el primer espacio en blanco y no seguía leyendo a partir de ese punto. Existen otras órdenes que están diseñadas específicamente para manejar cadenas de texto, y que nos podrán servir en casos como éste. Para leer una cadena de texto (completa, sin parar en el primer espacio), usaríamos la orden "gets"

```
gets(texto);
```

De igual modo, para escribir un texto en pantalla podemos usar "puts", que muestra la cadena de texto y avanza a la línea siguiente:

```
puts(texto);
```

Sería equivalente a esta otra orden: printf("%s\n", texto);

Existe un posible problema cuando se mezcla el uso de "gets" y el de "scanf": si primero leemos un número, al usar "scanf("%d", ...", la variable numérica guardará el número... pero el “Enter” que pulsamos en el teclado después de introducir ese número queda esperando en el buffer (la memoria intermedia del teclado). Si a continuación leemos un segundo número, no hay problema, porque se omite ese Enter, pero si leemos una cadena de texto, ese Enter es aceptable, porque representaría una cadena vacía. Por eso, cuando primero leemos un número y luego una cadena usando "gets", tendremos que "absorber" el Enter, o de lo contrario el texto no se leería correctamente. Una forma de hacerlo sería usando “getchar” o limpiar el buffer correspondiente usando “fflush”:

```
scanf("%d", &numero);
getchar();
gets(texto);

scanf("%d", &numero);
fflush(stdin);
gets(texto);
```

Además, existe un problema adicional: muchos compiladores que sigan el estándar C99 pueden dar el aviso de que "gets es una orden no segura y no debería utilizarse". Es posible que esta orden ni siquiera esté disponible en compiladores nuevos. Se debe a que "gets" no comprueba que haya espacio suficiente para los datos que introduzca el usuario, lo que puede dar lugar a un error de desbordamiento y a que el programa se comporte de forma imprevisible. En un fuente de práctica, creado por un principiante, no es peligroso, pero sí en caso de un programa en C "profesional" que esté dando servicio a una página Web (por ejemplo), donde el uso de "gets" sí podría suponer una vulnerabilidad.
Para evitar este inconveniente podemos usar la función:

fgets (nombre_string, int max_car_leer, nombre_buffer);

Primero se establece donde se quiere copiar la línea leída. A continuación el máximo número de caracteres incluyendo el '\0' que se pueden leer. En el ejemplo, fgets lee del *buffer* hasta que encuentra un '\n' o hasta que haya copiado en s un máximo de 29 caracteres. La propia fgets se encarga de incluir el '\0' para finalizar la cadena. El último parámetro hace referencia de donde se obtiene los datos. En el caso de leer de consola se indica stdin (standard input). Otra diferencia importante con gets es que el retorno de carro se copia también en la cadena.

Ej
```
char s[30];
fgets(s, 30, stdin);
```

Asignando a una cadena el valor de otra: strcpy, strncpy; strcat

Cuando queremos asignar una cadena de caracteres a otra NO se puede escribir saludo="hola" ni algo como texto1=texto2. La forma correcta de guardar en una cadena de texto un cierto valor es:

strcpy (destino, origen); o strcpy (“textoDefinitivo”, “textoProvisional”);

Ej:strcpy (saludo, "hola");

Es nuestra responsabilidad que en la cadena de destino haya suficiente espacio reservado para copiar lo que queremos. Si no es así, estaremos sobreescribiendo direcciones de memoria en las que no sabemos qué hay. Para evitar este problema, tenemos una forma de indicar que queremos copiar sólo los primeros n bytes de origen, usando la función "strncpy", así:

strncpy (destino, origen, n);

Finalmente, podemos añadir una cadena al final de otra (concatenarla), con

strcat (destino, origen);

```
#include <stdio.h>
#include <string.h>
int main()
{
char texto1[40], texto2[40], texto3[10];
printf("Introduce un frase: ");
gets(texto1);
strcpy(texto2, texto1);
printf("Una copia de tu texto es %s\n", texto2);
strncpy(texto3, texto1, 4);
printf("Y sus 4 primeras letras son %s\n", texto3);
return 0;
}
```

```
 #include <stdio.h>
 #include <string.h>
 int main()
 {
 char texto1[40], texto2[40], texto3[40];
 printf("Introduce tu nombre: ");
 gets(texto1);
 printf("Introduce tu apellido: ");
 gets(texto2);
 strcat(texto1, " "); /* Añado un espacio  */
 strcat(texto1, texto2); /* Y luego el apellido */
 printf("Te llamas %s\n", texto1);
 return 0;
 }
```

Comparando cadenas: strcmp

Para comparar dos cadenas alfabéticamente (para ver si son iguales o para poder ordenarlas, por ejemplo), usamos strcmp (cadena1, cadena2); Esta función devuelve un número entero, que será:

- 0 si ambas cadenas son iguales.
- Un número negativo, si cadena1 < cadena2.
- Un número positivo, si cadena1 > cadena2.

Hay que tener cuidado, porque las cadenas se comparan como en un diccionario, pero hay que tener en cuenta ciertas cosas:

- Al igual que en un diccionario, todas las palabras que empiecen por B se consideran "mayores" que las que empiezan por A (la B está a continuación de la A).
- Si dos cadenas empiezan por la misma letra (o las mismas letras), se ordenan basándose en la primera letra diferente, también al igual que en el diccionario (AMA es "mayor" que ALA).
- La primera diferencia está que en que se distingue entre mayúsculas y minúsculas. Para más detalles, en el código ASCII las mayúsculas aparecen antes que las minúsculas, así que las palabras escritas en mayúsculas se consideran "menores" que las palabras escritas en minúsculas. Por ejemplo, "ala" es menor que "hola", porque una empieza por "a" y la otra empieza por "h", pero "Hola" es menor que "ala" porque la primera empieza con una letra en mayúsculas y la segunda con una letra en minúsculas.
- La segunda diferencia es que el código ASCII estándar no incluye eñe, vocales acentuadas ni caracteres internacionales, así que estos caracteres "extraños" aparecen después de los caracteres "normales", de modo que "adiós" se considera "mayor" que "adiposo", porque la "o" acentuada está después de todas las letras del alfabeto inglés.

Funciones de cadenas: sprintf, sscanf, strstr, atoi...

La funcion **"sprintf"** crea una cadena de texto a partir de una especificación de formato y unos ciertos parámetros, al igual que hace "printf", pero la diferencia está en que "printf" manda su salida a la pantalla, mientras que "sprintf" la deja guardada en una cadena de texto.

```
char cadena[100];
sprintf (cadena, "El número %d multiplicado por 2 vale %d\n", 50, 50*2);
```

Es útil para formatear texto que no vaya a aparecer directamente en pantalla de texto, sino que lo vayamos a enviar a un fichero, o que queramos mostrar en pantalla gráfica, o enviar a través de una red mediante "sockets".

La función **"sscanf"** es similar a "scanf", con la diferencia de que los valores para las variables no se leen desde el teclado, sino desde una cadena de texto.

```
strcpy(cadena, "20 30");
sscanf(cadena, "%d %d", &primerNum, &segundoNum);
```

Será muy útil para analizar el contenido de ficheros de texto. Además sscanf devuelve el número de valores que realmente se han detectado, de modo que podemos comprobar si ha tomado todos los que esperábamos o alguno menos (porque el usuario haya tecleado menos de los que esperábamos o porque alguno esté tecleado incorrectamente).

La función "strstr" permite comprobar si una cadena contiene un cierto texto. Devuelve NULL (un valor especial, que nos encontraremos cada vez más a partir de ahora) si no la contiene, y otro valor (no daremos más detalles por ahora sobre qué tipo de valor ni por qué) en casi de que sí la contenga:

```
if (strstr (frase, "Hola ") == NULL);
printf("No has dicho la palabra Hola ");
```

Si queremos extraer el valor numérico de una cadena, no sólo tenemos "sscanf". Otras alternativas más sencillas (aunque menos fiables, porque devuelven 0 en caso de que no exista valor numérico), son "atoi" si queremos convertir a "int" y "atof" si queremos extraer un valor con decimales (un "float"):

```
valorEntero = atoi(texto);
valorReal = atof(texto);
```

Nota: éstas no son todas las posibilidades que tenemos para manipular cadenas, pero posiblemente sí son las más habituales. Hay otras que nos permiten buscar una letra dentro de una cadena (strchr), "dar la vuelta" a una cadena (strrev), etc. Según el compilador que usemos, podemos tener incluso funciones ya preparadas para convertir una cadena a mayúsculas (strupr) o a minúsculas (strlwr).