Arquitectura de sistemas

Abelardo Pardo

University of Sydney
School of Electrical and Information Engineering
NSW, 2006, Australia
Autor principal del curso de 2009 a 2012

Iria Estévez Ayres

Damaris Fuentes Lorenzo

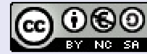
Pablo Basanta Val

Pedro J. Muñoz Merino

Hugo A. Parada

Derick Leony

Universidad Carlos III de Madrid
Departamento de Ingeniería Telemática
Avenida Universidad 30, E28911 Leganés (Madrid), España

© Universidad Carlos III de Madrid | Licencia Creative Commons 

Capítulo 8. Operaciones de entrada y salida

Tabla de contenidos

8.1. Introducción
8.2. Funciones E/S carácter a carácter

- 8.2.1. La función `getc`
- 8.2.2. La función `getchar`
- 8.2.3. La función `putc`
- 8.2.4. La función `putchar`

8.3. Funciones E/S para tipos de datos

- 8.3.1. La función `gets`
- 8.3.2. La función `puts`
- 8.3.3. La función `scanf`
- 8.3.4. La función `printf`

8.4. Funciones de entrada para leer strings de manera segura

- 8.4.1. La función `fgets`
- 8.4.2. La función `getline`

8.5. Bibliografía de apoyo
8.6. Preguntas de autoevaluación

8.1. Introducción

C ofrece un conjunto de funciones para realizar operaciones de entrada y salida (E/S) con las cuales puedes leer y escribir cualquier tipo de fichero.

En C, un *fichero* se puede referir a un fichero en disco, a un terminal, a una impresora, etc. Dicho de otro modo, un fichero representa un dispositivo concreto con el que puedes intercambiar información. C trata a cualquier tipo de fichero como una serie de bytes (o caracteres). Este conjunto de bytes, que es lo que realmente se transfiere entre un fichero y un programa, se le conoce como *flujo* (*stream* en inglés).

Antes de poder trabajar con un fichero, tienes que abrir ese fichero. En C cuentas con 3 flujos de ficheros que ya están abiertos, disponibles para que se usen en cualquier programa:

- `stdin`: La entrada estándar de lectura. Generalmente va asociado al teclado.
- `stdout`: La salida estándar de escritura. Generalmente va asociado a la pantalla del terminal.
- `stderr`: La salida estándar de escritura para mensajes de error. Generalmente también va asociado a la pantalla del terminal.

En las siguientes secciones veremos diferentes formas de usar `stdin` y `stdout`. Todas las operaciones que vamos a ver necesitan la librería `stdio.h` para funcionar.

8.2. Funciones E/S carácter a carácter

Vamos a centrarnos en cómo hacer que nuestro programa lea un carácter de un usuario (generalmente introducido por teclado) y lo escriba en la salida (que será la pantalla del terminal).

8.2.1. La función `getc`

La función `getc` lee el siguiente carácter de un flujo de fichero y devuelve el valor numérico de ese carácter (lo devuelve como entero).

```
#include <stdio.h>
int getc(FILE *stream);
```

Aquí `FILE *stream` es una variable con el flujo de fichero. Si llega al final del fichero o hay algún error, la función devuelve `EOF`.

> **Nota**
>
> No te fijes ahora en el tipo de datos `FILE`, pues por ahora vamos a usar como flujos de ficheros `stdin` y `stdout`, que están ya predefinidos. Por otro lado, `EOF` es una constante declarada en el fichero de cabecera `stdio.h`. Generalmente vale `-1`, pero usa mejor `EOF`, por si luego usas el programa con otro compilador u otro sistema operativo que use otro valor distinto.

8.2.2. La función getchar

La función `getchar` es equivalente a la función `getc(stdin)`.

```
#include <stdio.h>
int getchar(void);
```

Aquí `void` indica que no se necesita ningún argumento para llamar a la función, pues entiende que leeremos el caracter desde la entrada estándar.

El siguiente programa lee dos caracteres introducidos por el usuario desde el teclado, de las dos formas posibles vistas hasta ahora, y luego los imprime en la pantalla:

```
1   #include <stdio.h>
2
3   int main(void)
4   {
5     int caracter1;
6     char caracter2;
7
8     printf("Por favor, teclea dos caracteres seguidos:\n");
9     caracter1 = getc(stdin);
10    caracter2 = getchar();
11    printf("El primer caracter que has introducido es: %c\n",caracter1);
12    printf("El segundo caracter que has introducido es: %c\n",caracter2);
13    return 0;
14  }
```

Como puedes ver, pese a que las funciones están esperando un carácter almacenado como entero, la variable `caracter2` se ha declarado de tipo `char` y, aún así, el programa no da error y funciona. Esto es así porque, internamente, las variables de tipo `char` son almacenadas con su correspondencia numérica, de ahí que puedan pasarse a estas funciones que esperan un tipo `int`.

8.2.3. La función putc

La función `putc` escribe un carácter al flujo del fichero especificado.

```
#include <stdio.h>
int putc(int c, FILE *stream);
```

Aquí `c` es el carácter, guardado como entero, y el segundo argumento es el fichero. Si la operación tiene algún error, devuelve `EOF`; si no, devuelve el carácter escrito.

8.2.4. La función putchar

La función `putchar` también se usa para escribir un carácter en la pantalla. La única diferencia es que no necesita el segundo argumento, pues usa siempre la salida estándar que esté predefinida.

```
#include <stdio.h>
int putchar(int c);
```

El siguiente programa saca por pantalla el carácter 'A' con las dos funciones que acabamos de ver.

```
1   #include <stdio.h>
2
3   int main(void)
4   {
5     int caracter1 = 65; /*Suele ser el valor numérico de A*/
6     char caracter2 = 'A';
7
8     printf("La letra con valor numérico de 65 es:\n");
9     putc(caracter1,stdout);
10    printf("Y la variable caracter2 contiene la letra:\n");
11    putchar(caracter2);
12    return 0;
13  }
```

De la misma manera que ocurría con `getc` y `getchar`, aunque `putc` y `putchar` esperan variables de tipo `int`, podemos pasarles variables declaradas como `char`, al ser este tipo de variables internamente almacenadas con su correspondencia numérica.

8.3. Funciones E/S para tipos de datos

En vez de leer carácter a carácter, podemos leer tipos de datos de una vez como enteros, cadenas de caracteres, etc. Para ello contamos con las funciones `gets`, que lee cadenas de caracteres, y `scanf`, que además maneja otros tipos de datos, y que veremos a continuación. De la misma manera, podemos escribir una cadena de caracteres con `puts` y otros tipos de datos de una vez con la función `printf` que ya habéis visto. Aquí hablaremos un poco más sobre ella.

8.3.1. La función `gets`

Para leer de entrada una línea carácter a carácter ya hemos visto las funciones `getc` y `getchar`. Para leer una línea completa contamos con diferentes funciones; una de ellas es `gets`:

```
#include <stdio.h>
char *gets(char *s);
```

Los caracteres leídos de entrada se guardan en el array `s`. La función deja de leer y añade el carácter de terminación '\0' cuando encuentra el carácter de nueva línea o el de fin de fichero `EOF`. Si todo va bien devuelve `s`, y si hay algún error devuelve un puntero a `NULL`.

8.3.2. La función `puts`

La función `puts` se usa para escribir una secuencia de caracteres al flujo de salida estándar:

```
#include <stdio.h>
int *puts(const char *s);
```

`s` se refiere al array que contiene la cadena de caracteres. Si la función se realiza correctamente, devuelve 0. Si no, devuelve algo distinto de cero.

El siguiente programa muestra un ejemplo del funcionamiento de `gets` y `puts`.

```
1   #include <stdio.h>
2   #define TAM_MAXIMO 80
3
4   int main(void)
5   {
```

```
6   char cadena[TAM_MAXIMO];
7
8   printf("Por favor, escribe una línea de no más de 80 caracteres:\n");
9   gets(cadena);
10  printf("La línea que has introducido es:\n");
11  puts(cadena);
12  return 0;
13 }
```

Nota

Como podéis ver, el uso de `gets` es algo peligroso. La función no conoce el tamaño del array que le vas a pasar, simplemente va leyendo datos de la entrada hasta que llega al final. Esto da problemas si el usuario teclea más caracteres de lo que puede almacenar el array declarado, pudiendo provocar que otras variables se vean afectadas al ser sobreescritas o, como poco, que la aplicación se comporte de manera inesperada. Más tarde veremos funciones seguras para leer cadenas.

8.3.3. La función `scanf`

La función `scanf` permite leer varios tipos de datos de una sola vez, tales como enteros, números decimales o cadenas de caracteres.

```
#include <stdio.h>
int scanf(const char *format,...);
```

Aquí se pueden indicar varios especificadores de formato en la variable de tipo puntero `format`, dependiendo del tipo que se quiere leer, como con `printf`. Si todo va bien, devuelve el número de datos leídos. Si hay algún error, devuelve `EOF`.

Si usamos el especificador `%s` para leer una cadena, la función lee caracteres hasta encontrar un espacio, un intro, un tabulador, un tabulador vertical o un retorno de carro. Los caracteres que lee se guardan en un array que debe ser lo suficientemente grande como para almacenarlos. Añade el carácter nulo al final automáticamente.

Nota

Al igual que pasaba con `gets`, es muy peligroso usar `scanf` para leer cadenas, pues `scanf` no tiene en cuenta la longitud de lo leído y admitirá que el usuario escriba más caracteres que lo que el array definido en el programa pueda admitir. Como resultado, `scanf` escribirá los caracteres que ya no quepan en el array definido en otras porciones de memoria que pueden contener otros datos que está usando nuestro programa. Esto desembocará en comportamientos anómalos, en fugas de memoria, etc. Más tarde veremos cómo poder leer cadenas de caracteres de manera más segura.

Con `scanf`, a diferencia de `printf`, hay que ***pasar punteros a los argumentos*** para que `scanf` pueda modificar sus valores.

El siguiente programa muestra el uso de `scanf` para distintos especificadores de formato.

```
1  #include <stdio.h>
2  #define TAM_MAXIMO 80
3
4  int main(void)
5  {
6    char cadena[TAM_MAXIMO];
7    int entero1, entero2;
8    float decimal;
9
10   printf("Introduce dos enteros separados por un espacio: \n");
11   scanf("%d %d", &entero1, &entero2);
12   printf("Introduce un número decimal:\n");
```

```
12     printf("Introduce un número decimal:\n");
13     scanf("%f", &decimal);
14     printf("Introduce una cadena:\n");
15     scanf("%s",cadena);
16     printf("Esto es todo lo que has escrito:\n");
17     printf("%d %d %f %s\n",entero1,entero2,decimal,cadena);
18     return 0;
19 }
```

Nota que en la línea 15 usamos `scanf` para leer una serie de caracteres, y esos caracteres (más el caracter de terminación de cadena) se guardan en el array apuntado por `cadena`. Sin embargo, aquí no se utiliza el operador de dirección `&`, porque el propio nombre de un array (`cadena` en este caso), apunta (es equivalente) a la dirección de comienzo del propio array.

8.3.4. La función `printf`

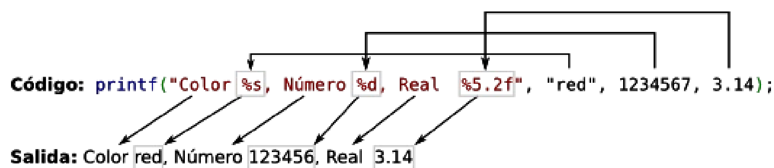
La función **printf** (que deriva su nombre de "*print formatted*") imprime un mensaje por pantalla utilizando una "cadena de formato" que incluye las instrucciones para mezclar múltiples cadenas en la cadena final a mostrar por pantalla. Lenguajes como Java también incluyen funciones similares a esta (ver Método `printf` de la clase `PrintStream`).

`printf` es una función especial porque recibe un número variable de parámetros. El primer parámetro es fijo y es la cadena de formato. En ella se incluye texto a imprimir literalmente y ***marcas*** a reemplazar por texto que se obtiene de los parámetros adicionales. Por tanto, `printf` se llama con tantos parámetros como marcas haya en la cadena de formato más uno (la propia cadena de formato). El siguiente ejemplo muestra cómo se imprime el valor de la variable `contador`.

```
printf("El valor es %d.\n", contador);
```

El símbolo "`%`" denota el comienzo de la marca de formato. La marca "`%d`" se reemplaza por el valor de la variable `contador` y se imprime la cadena resultante. El símbolo "`\n`" representa un salto de línea. La salida, por defecto, se justifica a la derecha del ancho total que le hallamos dado al campo, que por defecto tiene como longitud la longitud de la cadena.

Si en la cadena de formato aparecen varias marcas, los valores a incluir se toman en el mismo orden en el que aparecen. La siguiente figura muestra un ejemplo en el que la cadena de formato tiene tres marcas, `%s`, `%d` y `%5.2f`, que se procesan utilizando respectivamente la cadena "red", el entero `1234567` y el número real `3.14`.



No se comprueba que el número de marcas en la cadena de formato y el número de parámetros restantes sea consistente. En caso de error, el comportamiento de `printf` es indeterminado.

Las marcas en la cadena de formato deben tener la siguiente estructura (los campos entre corchetes son optativos):

```
%[parameter][flags][width][.precision][length]type
```

Toda marca, por tanto, comienza por el símbolo "`%`" y termina con su tipo. Cada uno de los nombres (*parameter, flags, width, precision, length* y *type*) representa un conjunto de valores posibles que se explican a continuación.

Parameter	Descripción
`n$`	Se reemplaza "n" por un número para cambiar el orden en el que se procesan los argumentos. Por ejemplo `%3$d` se refiere al tercer argumento independientemente del lugar que ocupa en la cadena de formato.
Flags	Descripción

Flags	Descripción
número	Rellena con espacios (o con ceros, ver siguiente flag) a la izquierda hasta el valor del número.
0	Se rellena con ceros a la izquierda hasta el valor dado por el flag anterior. Por ejemplo “`%03d`” imprime un número justificado con ceros hasta tres dígitos.
+	Imprimir el signo de un número
-	Justifica el campo a la izquierda (por defecto ya hemos dicho que se justifica a la derecha)
#	Formato alternativo. Para reales se dejan ceros al final y se imprime siempre la coma. Para números que no están en base 10, se añade un prefijo denotando la base.
Width	**Descripción**
número	Tamaño del ancho del campo donde se imprimirá el valor.
*	Igual que el caso anterior, pero el número a utilizar se pasa como parámetro justo antes del valor. Por ejemplo `printf("%*d", 5, 10)` imprime el número 10, pero con un ancho de cinco dígitos (es decir, rellenará con 3 espacios en blanco a la izquierda).
Precision	**Descripción**
número	Tamaño de la parte decimal para números reales. Número de caracteres a imprimir para cadenas de texto
*	Igual que el caso anterior, pero el número a utilizar se pasa como parámetro justo antes del valor. Por ejemplo `printf("%.*s", 3, "abcdef")` imprime “abc”.
Length	**Descripción**
hh	Convertir variable de tipo `char` a entero e imprimir
h	Convertir variable de tipo `short` a entero e imprimir
l	Para enteros, se espera una variable de tipo `long`.
ll	Para enteros, se espera una variable de tipo `long long`.
L	Para reales, se espera una variable de tipo `long double`.
z	Para enteros, se espera un argumento de tipo `size_t`.
Type	**Descripción**
`%c`	Imprime el carácter ASCII correspondiente
`%d`, `%i`	Conversión decimal con signo de un entero
`%x`, `%X`	Conversión hexadecimal sin signo
`%p`	Dirección de memoria (puntero)
`%e`, `%E`	Conversión a coma flotante con signo en notación científica
`%f`, `%F`	Conversión a coma flotante con signo, usando punto decimal
`%g`, `%G`	Conversión a coma flotante, usando la notación que requiera menor espacio
`%o`	Conversión octal sin signo de un entero
`%u`	Conversión decimal sin signo de un entero
`%s`	Cadena de caracteres (terminada en '\0')
`%%`	Imprime el símbolo %

Ejemplos de marcas de formato

Las marcas de formato que se incluyen como parte de la cadena que se pasa como primer parámetro a `printf` ofrece muchas posibilidades. A continuación se muestran algunos ejemplos:

- Especificar el ancho mínimo: Podemos poner un entero entre el símbolo de porcentaje (%) y el especificador de formato, para indicar que la salida alcance un ancho mínimo. Por ejemplo, `%10f`

especificador de formato, para indicar que la salida alcance un ancho mínimo. Por ejemplo, %10d asegura que la salida va a tener al menos 10 espacios de ancho. Esto es útil cuando se van a imprimir datos en forma de columnas. Por ejemplo, si tenemos:

```
int num = 12;
int num2 = 12345;
printf("%d\n",num2);
printf("%5d\n",num);
```

se imprime:

```
12345
   12
```

- Alinear la salida: Por defecto, la salida cuando se especifica ancho mínimo está justificada a la derecha. Para justificarla a la izquierda, hay que preceder el dígito de la anchura con el signo menos (-). Por ejemplo, `%-12d` especifica un ancho mínimo de 12, saca la salida justificada a la izquierda.
- Especificador de precisión: Puedes poner un punto (.) y un entero después de especificar un ancho de campo mínimo para especificar la precisión. En un dato de tipo float, esto permite especificar el número de decimales a sacar. En un dato de tipo entero o en cadenas de caracteres, especifica el ancho o longitud máxima.

Comprueba con estas preguntas si has entendido este documento.

1. Queremos imprimir la cadena guardada en `char string[30];` con la función `printf`:
    - Tenemos que usar `printf("%c\n",string[0]);`.
    - Tenemos que usar `printf("%s\n",string[0]);`.
    - Tenemos que usar `printf("%s\n",string);`

8.4. Funciones de entrada para leer strings de manera segura

Como hemos visto en el apartado anterior, las funciones gets y scanf usadas para leer cadenas de caracteres son propensas a errores, pues no controlan que el usuario haya introducido más caracteres que el que puede albergar el array definido para almacenarlos. Para ello, contamos con dos funciones que hacen segura la lectura de cadenas introducidas por el usuario, que son `fgets` y `getline`.

8.4.1. La función `fgets`

La sintaxis de la función `fgets` es:

```
#include <stdio.h>
char *fgets(char *s, int n, FILE *stream);
```

`s` referencia a un array de caracteres, que almacena lo que lee de un fichero apuntado por `stream`. `n` especifica el número máximo de elementos del array (caracteres). La función lee hasta `n-1` caracteres, y añade el caracter nulo para que la cadena quede bien formada. Si no ha llegado hasta n-1 caracteres pero llega a un caracter de nueva línea, '\n', también para de leer y devuelve la cadena (con ese carácter de nueva línea incluido). Si no hay errores, la función devuelve el puntero `s`. Si encuentra el final de fichero (EOF) o existe algún error, devuelve `null`.

`fgets` es así más segura que `gets`, pero tienes que lidiar con un par de cosas. Una es que si has introducido más caracteres de los que ha podido leer, el resto de caracteres se quedan en el buffer de entrada (en stdin), así que tendrás que eliminarlos para que no sean leídos en el próximo `fgets` si no quieres. Lo segundo es que `fgets` incluye el carácter '\n' al final, así que tendrás que quitarlo.

8.4.2. La función `getline`

La librería GNU ofrece la función no estándar getline que hace sencilla la lectura de líneas:

```
#include <stdio.h>
ssize_t getline(char **lineptr, size_t *n, FILE *stream);
```

Esta función lee una línea entera de stream, almacenando el texto (incluyendo el carácter de nueva línea y el de terminación) en un buffer y almacenando la dirección del buffer en *lineptr. Antes de llamar a getline, tienes que colocar en *lineptr la dirección de un buffer de *n bytes de largo usando malloc. Si este buffer no es lo suficientemente grande como para albergar la frase leída, getline hace el buffer más grande usando realloc, actualizando la nueva dirección en *lineptr y el tamaño en *n. Si inicializas *lineptr a null y *n a cero, getline hace ese malloc por tí.

Si todo ha ido bien, *lineptr es un char * que apunta al texto que se ha leido. La función devuelve el número de caracteres leídos aunque sin contar el carácter '\0' de terminación (sí con el de nueva línea); si ha habido algún error o se alcanza final de fichero, devuelve -1.

Nota

Siempre que tengas que leer una línea de texto del teclado, hazlo pues con getline; te evitarás cualquier problema posterior.

El siguiente programa muestra cómo usar getline para leer una línea de texto desde teclado de manera segura. Intenta teclear más de 10 caracteres, verás que getline lo gestiona correctamente, independientemente del número de caracteres que teclees.

```
1  #include <stdio.h>
2  #define TAM_MAXIMO 10
3  
4  int main(void)
5  {
6    ssize_t bytes_leidos;
7    size_t numero_bytes;
8    //ssize_t y size_t son sinónimos de unsigned int
9    char *cadena;
10 
11   puts("Por favor, introduce una línea de texto:\n");
12   /* Puedes pasar a getline los argumentos inicializados: */
13   //numero_bytes = TAM_MAXIMO;
14   //cadena = (char *) malloc (numero_bytes + 1);
15   //bytes_leidos = getline(&cadena, &numero_bytes, stdin);
16 
17   /*O bien, más sencillo, poner el número a 0 y la cadena a NULL, para que él mismo te haga
18   la reserva necesaria*/
19   numero_bytes = 0;
20   cadena = NULL;
21   bytes_leidos = getline(&cadena, &numero_bytes, stdin);
22 
23   if (bytes_leidos == -1)
24   {
25     puts("Error.");
26   }
27   else
28   {
29     puts("La línea es:");
30     puts(cadena);
31   }
32   free(cadena);
33 
34   return 0;
35 }
```

Comprueba con estas preguntas si has entendido este documento.

1. Indica la afirmación INCORRECTA respecto a la función `getline`:

    - Devuelve el número de caracteres leídos, incluyendo el caracter '\n' de fin de línea, pero no cuenta el '\0' de final de cadena.
    - Si el buffer de memoria que recibe como parámetro de entrada no tiene espacio suficiente, devuelve un error.
    - Si el puntero al buffer de memoria que se le pasa para guardar el resultado es `NULL`, reserva memoria internamente con un `malloc`.

8.5. Bibliografía de apoyo

- ***Funciones de entrada y salida***:
    - (Teoría y ejemplos) "Programación en C: Metodología, algoritmos y estructura de datos", páginas 129-134, 470-476, 506-508.
    - (Teoría y ejemplos) "The GNU C Programming Tutorial ", páginas 123-130, 132-134, 136-147.
    - (Teoría y ejemplos) "Practical C Programming", páginas 40-46.
- ***Problemas resueltos***: "Problemas resueltos de Programación en Lenguaje C", ejemplos 2.6-2.8, 2.11-2.15, 2.17 (páginas 57-65).

8.6. Preguntas de autoevaluación

Comprueba con estas preguntas si has entendido este documento.

1. Si queremos leer una línea de teclado e imprimirla por pantalla:

    - Podemos usar para leerla tanto `scanf` como `getline`, pues ambas funciones, si no tienen espacio para guardar lo leído, piden automáticamente memoria.
    - Usaremos `getline` para leer y `printf` para imprimir. La memoria pedida por `getline` la gestiona de forma automática el sistema operativo.
    - Usaremos `getline` para leer, `printf` para imprimir y `free` para liberar la memoria pedida por `getline`.

2. Si queremos imprimir la dirección de memoria donde está guardada la cadena `char cadena[100];`, usaremos:

    - No se pueden imprimir direcciones de memoria de cadenas.
    - `printf("%s\n",cadena);`
    - `printf("%p\n",cadena);`

3. Queremos imprimir la cadena guardada en `char string[30];` con la función `printf`:

    - No tenemos que inicializar `string`.
    - La cadena DEBE tener un caracter cero '\0' final, si no, puede producirse un fallo de segmento (segmentation fault).
    - Podemos usar `string.printf("%s");`