



UNLaM

Dto. Ingeniería e Investigaciones Tecnológicas

---

**ATENCIÓN:**

Este material es continuación del archivo: "05-UnaFuncionDeMenu.pdf", [Semana 3] / [Una Función de Menú].

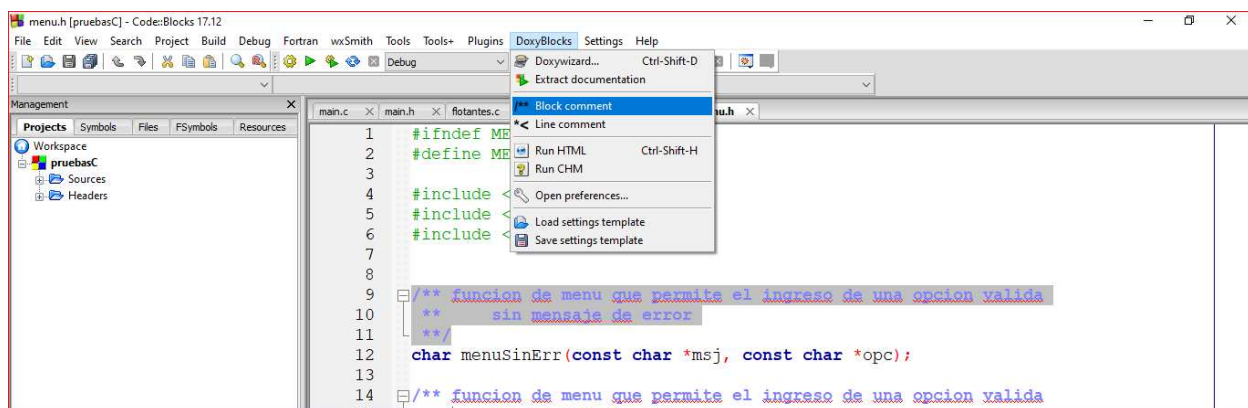
**Objetivos:**

- comprender el uso de las cadenas de control de conversión de formato utilizadas, por la familia de funciones "...printf", para mostrar por pantalla variables del tipo 'char' (además de arrays de 'char'), de tipo entero y de punto flotante, .
- desarrollar funciones cuyos prototipos no se "publican" en el ".h" correspondiente, dado que son funciones "de servicio" a ser invocadas desde una función de menú.
- incentivar a que se pongan en práctica y se prueben estas funciones en un proyecto compuesto por varios archivos fuente, a partir de la práctica es cómo se fijan los conocimientos.
- brindar una introducción a la documentación de funciones, que aunque muy breve y acotada, es un paso inicial e introductorio a las buenas prácticas de un profesional.

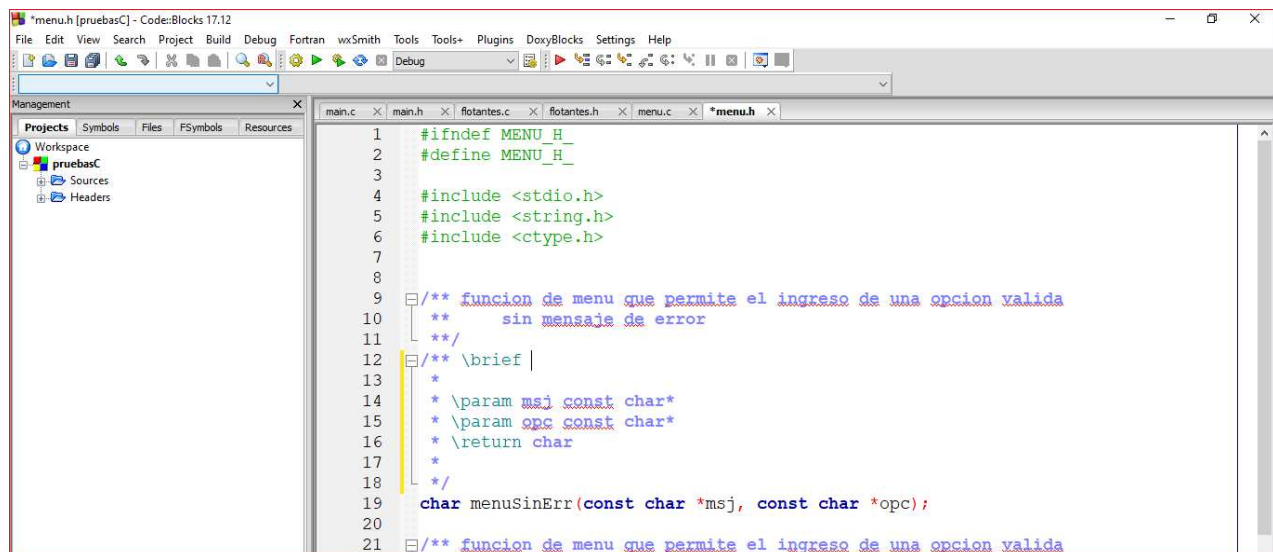
### Comencemos con una digresión – Documentar funciones.

NOTA: No forma parte de contenidos a evaluar en la materia.

Abra el proyecto de Uso de Punto Flotante – Función de Menú (que yo denominé "pruebaC"), con que estuvimos trabajando. En "menu.h" seleccione con el cursor el precario comentario de documentación que habíamos escrito y del menú desplegable [DoxyBlocks] seleccione [Block comment].



Contra lo esperado, no eliminó el comentario anterior, ...



... insertó un comentario de documentación en el lugar en que quedó el cursor y sorpresivamente (como pudo), completó los parámetros, así que completemos y corrijamos ...



UNLaM

Dto. Ingeniería e Investigaciones Tecnológicas

```

1  #ifndef MENU_H
2  #define MENU_H
3
4  #include <stdio.h>
5  #include <string.h>
6  #include <ctype.h>
7
8  /** Al comenzar a ejecutarse, muestra el menú de opciones.<br>
9   * y permite el ingreso por teclado de una opción.<br>
10  * Si es una opción válida.<br>
11  * - la devuelve.<br>
12  * de lo contrario.<br>
13  * - repite desde el comienzo.</code>
14  */
15
16  /** \brief función de menú que permite el ingreso de una
17   * opción válida sin mensaje de error.
18   * \param msj (E) : mensaje o menú de opciones a desplegar
19   * \param opc (E) : conjunto de opciones válidas
20   * \return la opción (válida), elegida
21   */
22  char menuSinErr(const char *msj, const char *opc);
23
24  /** función de menú que permite el ingreso de una opción válida

```

- la primera llave ({} ) marca una descripción detallada, y será mostrada al final
- la segunda llave ({} ) abarca:
  - la descripción breve (\brief), se mostrará al comienzo
  - los argumentos recibidos (\param), se mostrarán a continuación
  - qué devuelve la función (\return), se mostrará a continuación del anterior
- se han utilizado unos pocos "tags" de HTML (rectángulos en rojo), que serán utilizados al mostrar la documentación emergente:
  - (<br>), que se vaya al siguiente renglón (aunque a continuación haya texto)
  - (<code>) y (</code>), marcas de inicio y fin de que sea mostrado como código
  - ("– – ") como el exceso de espacios en blanco se elimina (queda sólo uno), se han usado dos guiones seguidos por espacio en blanco para que se muestre con sangría el trozo de pseudocódigo
  - haga la prueba de utilizar <b> por ejemplo después de <code> y </b> antes de </code> para que se muestre en "bold"
  - haga la prueba de encerrar entre <u> y </u> lo que quiera subrayar (underline), entre <i> y </i> para letra inclinada (italics)

Tenga en cuenta que como regla general si pone más de un "tag" que abre (sin "/"), al poner los que cierran (con "/"), van en orden inverso. <br> no tiene cierre.



UNLaM

Dto. Ingeniería e Investigaciones Tecnológicas

Y con esto ya tenemos bastante variedad para la documentación que el IDE de Code::Blocks puede utilizar (sólo utiliza un subconjunto reducido, y no todas las secciones habituales de documentación). Tampoco utiliza todas las *secciones* de otros documentadores (por ejemplo: `\details` o `\author`).

En la función "main", si comienza a modificar la función invocada se despliega la documentación.

```

1  #include "main.h"
2
3
4  int main()
5  {
6      char opcion;
7
8      do
9      {
10         opcion = menu(MENU_MSJ1, MENU_OPC1);
11         switch(
12             {
13             case 'f': menu2(): char
14             case 'd': menu3(): char
15             case 't': menu5(): char
16             case 'e': menuConErr(): char
17             case 's': menuSinErr(): char
18             }
19         ) while(opc
20         return 0;
21     }
22

```

char menuSinErr(const char\* msj, const char\* opc)  
(function)  
función de menú que permite el ingreso de una opción válida sin mensaje de error.  
Parameters:  
msj (E): mensaje o menú de opciones a desplegar  
opc (E): conjunto de opciones válidas  
Returns:  
la opción (válida), elegida  
Description:  
Al comenzar a ejecutarse, muestra el menú de opciones y permite el ingreso por teclado de una opción. Si es una opción válida  
- - la devuelve de lo contrario  
- - repite desde el comienzo

[Open declaration](#)  
[Open implementation](#)

En cambio, para la otra función que aún no tiene los comentarios, resultará:

```

1  #include "main.h"
2
3
4  int main()
5  {
6      char opcion;
7
8      do
9      {
10         opcion = menu(MENU_MSJ1, MENU_OPC1);
11         switch(
12             {
13             case 'f': menu2(): char
14             case 'd': menu3(): char
15             case 't': menu5(): char
16             case 'e': menuConErr(): char
17             case 's': menuSinErr(): char
18             }
19         ) while(opc
20         return 0;
21     }
22

```

char menuConErr(const char\* msj, const char\* opc)  
(function)  
Description:  
funcion de menu que permite el ingreso de una opcion valida con mensaje de error

[Open declaration](#)  
[Open implementation](#)  
[Close Top](#)

Debería eliminarlas de "menu.c", no se usan.

**UNLaM**Dto. Ingeniería e Investigaciones Tecnológicas

---

Recuerde que dispone de una función de menú de uso general, que seguramente resultará de utilidad en otros programas.

Es de esperar que lo que le hemos mostrado, le resultará más que suficiente, para exceder lo esperable en una evaluación técnica para un puesto de trabajo en la que usted se presente y podrá *marcar* la diferencia.

Lo visto hasta acá, en este documento, no hace a nuestra materia. Tal como comenzamos: "No forma parte de contenidos a evaluar en la materia.", pero no olvide que es conveniente, para usted, ir haciendo una mínima documentación de lo que hace, porque un tiempo después, uno se terminará preguntando "¿qué hacía esta función?".

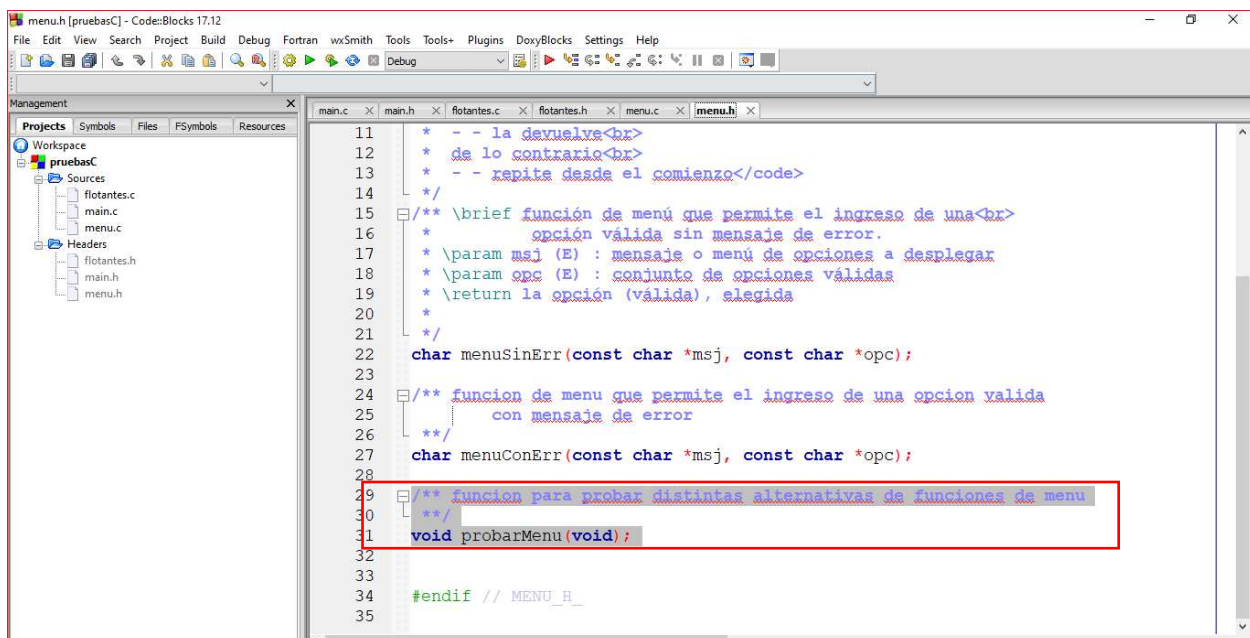
**Fin de la digresión: – Documentar funciones.**

Continuamos con lo que nos atañe:



### Continuando con nuestro proyecto.

Continuando con nuestro proyecto, llamado: "**pruebasC**", en "**menu.h**" proceda a eliminar la declaración de "**probarMenu**" que ya no utilizaremos:



```
11  * -- la devuelve<br>
12  * de lo contrario<br>
13  * -- repite desde el comienzo</code>
14  */
15  /** \brief función de menú que permite el ingreso de una<br>
16  * opción válida sin mensaje de error.
17  * \param msj (E) : mensaje o menú de opciones a desplegar
18  * \param opc (E) : conjunto de opciones válidas
19  * \return la opción (válida), elegida
20  */
21  /**
22  char menuSinErr(const char *msj, const char *opc);
23  */
24  /** función de menú que permite el ingreso de una opción válida
25  * con mensaje de error
26  */
27  char menuConErr(const char *msj, const char *opc);
28  */
29  /** función para probar distintas alternativas de funciones de menú
30  */
31  void probarMenu(void);
32  */
33
34 #endif // MENU_H_
35
```

Y luego en "**menu.c**", ... para eliminar las definiciones de las funciones excedentes "**menu2**", "**menu3**" y "**menu5**" (o "**menu4**" de acuerdo a la adoptada como versión definitiva), puede proceder de la siguiente manera.

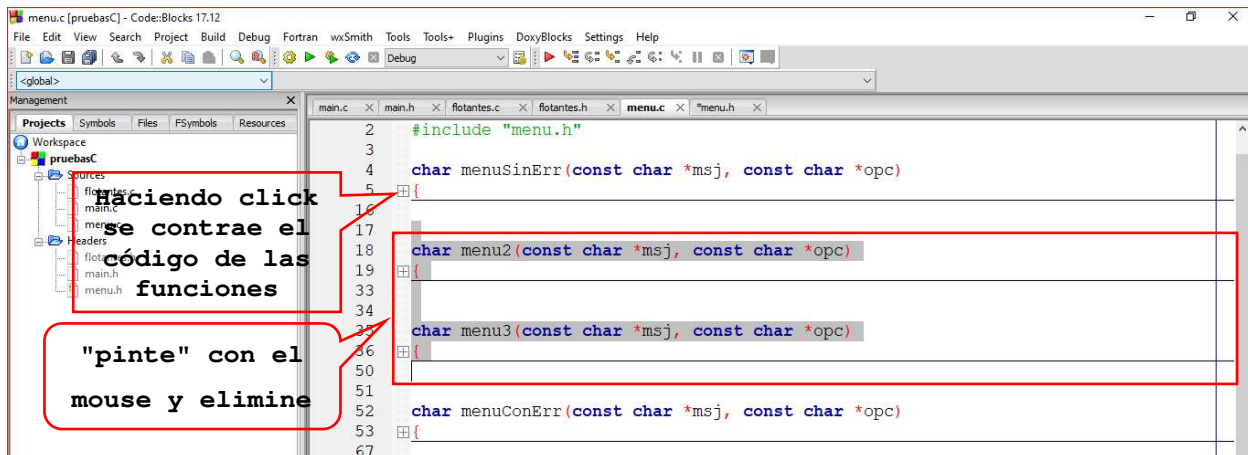
Haciendo "**click**" en el pequeño cuadrado con un "-" que el IDE muestra a la izquierda del margen junto al "*prototipo*" de cada función se contraerá el código de la misma. De este modo, podrán eliminar las definiciones de las funciones que no se van a conservar sin tener que "*pintar*" muchas líneas o cometer errores.



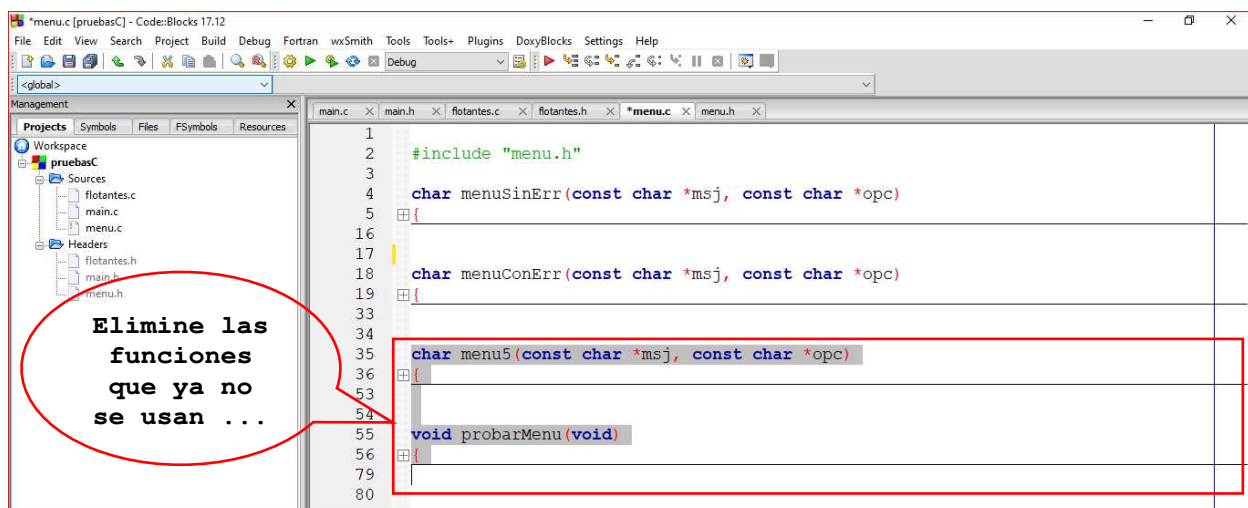


UNLaM

Dto. Ingeniería e Investigaciones Tecnológicas



Haciendo lo mismo con las otras funciones a eliminar:



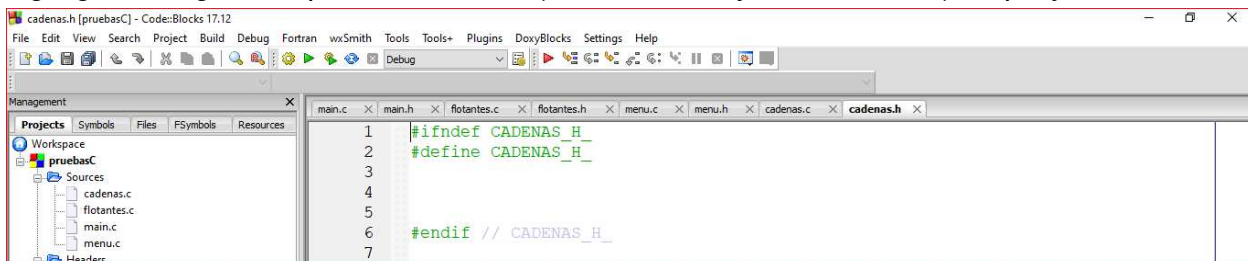
Una vez eliminadas las definiciones de las funciones que no se utilizarán más, "menu.c" queda:

```

main.c x main.h x flotantes.c x flotantes.h x *menu.c x menu.h x
1
2 #include "menu.h"
3
4 char menuSinErr(const char *msj, const char *opc)
5 {
6     char opta;
7
8     do
9     {
10         printf("%s", msj);
11         fflush(stdin);
12         scanf("%c", &opta);
13     } while(strchr(opc, opta) == NULL);
14     return opta;
15 }
16
17
18 char menuConErr(const char *msj, const char *opc)
19 {
20     char opta;
21     int priVez = 1;
22
23     do
24     {
25         printf("%s",
26             priVez ? priVez = 0, "" : "ERROR - Opcion NO Valida.\n",
27             msj);
28         fflush(stdin);
29         scanf("%c", &opta);
30     } while(strchr(opc, opta) == NULL);
31     return opta;
32 }
33
34

```

Agregue el siguiente par de archivos ("**cadenas.c**" y "**cadenas.h**"), al proyecto

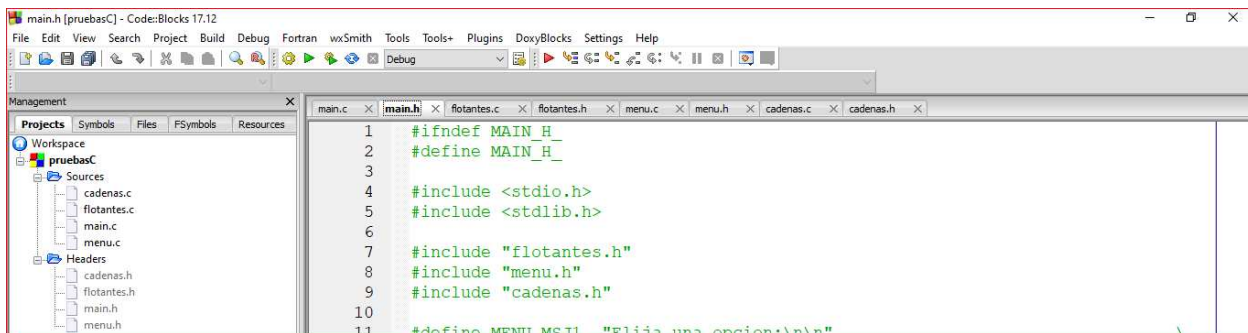


```

cadenas.h [pruebasC] - Code::Blocks 17.12
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help
1 #ifndef CADENAS_H_
2 #define CADENAS_H_
3
4
5
6 #endif // CADENAS_H_
7

```

Escriba los correspondientes **#include** "**cadenas.h**" en "**main.h**" y en "**cadenas.c**"

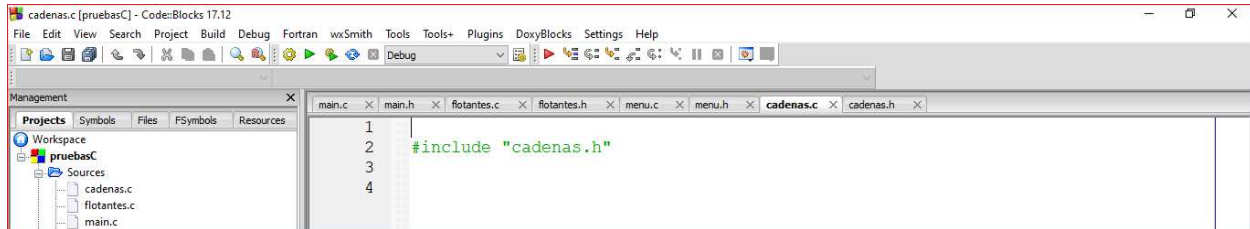


```

main.h [pruebasC] - Code::Blocks 17.12
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help
1 #ifndef MAIN_H_
2 #define MAIN_H_
3
4 #include <stdio.h>
5 #include <stdlib.h>
6
7 #include "flotantes.h"
8 #include "menu.h"
9 #include "cadenas.h"
10
11 #define MENU_MSJL "Elija una opcion:\n\n"

```





### Uso de Cadenas de Conversión de Formato de Salida.

Veremos cómo se muestran con las funciones de la familia "...**printf**" los distintos tipos de datos. Por funciones de la familia "...**printf**" nos referimos (y sólo utilizaremos en nuestra materia), a "**printf**"<sup>(1)</sup>, "**fprintf**"<sup>(2)</sup> y "**sprintf**"<sup>(3)</sup>. Las dos primeras hacen su salida por una "**stream**" de archivo (**stdout**), la primera<sup>(1)</sup> y un archivo (abierto en modo conveniente), la segunda<sup>(2)</sup> en su primer argumento. En el caso de la tercera<sup>(3)</sup> y su salida es en un **array** de **char**, que debe tener un tamaño suficiente para contener la cadena que se genera.

La función "**fprintf**" puede reemplazar (y exceder), el uso la función "**printf**". Su prototipo es

```
int fprintf(FILE *stream, const char *format [, argument]);
```

Como el Lenguaje C en buena medida se maneja con **streams** y dado que tanto **stdout** (salida normal, habitualmente, pantalla), como **stderr** (salida de error, habitualmente, pantalla); es que decimos "**fprintf**" excede las *prestaciones* de "**printf**".

Modo de uso, por ejemplo, para:

```
printf("Hola, mundo\n");
```

Será:

```
fprintf(stdout, "Hola, mundo\n"); // equivalente al printf
fprintf(stderr, "Hola, mundo\n"); // "casi" equivalente al printf
```



UNLaM

Dto. Ingeniería e Investigaciones Tecnológicas

En cuanto a la función "sprintf":

```
char    cad[30]; // debe tener suficiente espacio
// ...
sprintf(cad, "Hola, mundo\n"); // almacena a partir de la
                                // dirección de memoria implícita
                                // en el identificador cad, la
                                // cadena agregando a continuación
                                // la marca de fin de cadena "\0"
```

Y en cuanto a la función "fprintf":

```
FILE *fp; // puntero a FILE con el que se abrirá un archivo

if( (fp = fopen(nomArch, "wt") != NULL )
{
    fprintf(fp, "Hola, mundo\n"); // almacena en el mismo la
                                // cadena de caracteres y en la
                                // plataforma DOS / Windows el
                                // carácter "\n" lo reemplaza por
                                // la secuencia "\r\n" que es la
                                // marca de fin de registro de los
                                // archivos de texto
```

Todo lo que se especifica a continuación vale para las tres funciones:

Los especificadores de formato están compuestos, en general, por la secuencia:

**%[flags][width][.precision][size]type**

### Especificadores de formato para los char

#### Uso del tipo (type) de conversión para los char --> %c

Entre todos los indicadores opcionales anteriores, sólo valen para el tipo (type), "c" (un solo char mostrado como letra):

- ([flags]) única posibilidad el "-" (guión del medio), que sirve para alinear a la izquierda en el caso de utilizar un tamaño de campo mayor que 1 (uno). Caso de no querer alinear a la izquierda, nada.

- ([width]) que indica el tamaño de campo.

NOTA: el par de [ ] indica que son parámetros opcionales.



UNLaM

Dto. Ingeniería e Investigaciones Tecnológicas

En definitiva a un **char** se lo puede mostrar con "%c" o "%5c" o "%-5c" (si se quisiera un tamaño de campo 5).

```

C:\Users\USER\Desktop\pruebasC\bin\Debug\pruebasC.exe
Mostrando un char con [flags] y [width]:
-----
Mostrando con:
%3c : " K" (solo [width])
%-3c: "K " (con [flag] y [width])
Mostrando el caracter y su numero de ASCII en decimal, octal y hexadecimal:
%c "K" - %d "75" - %o "113" - %x "4b" - %X "4B"
  
```

Recuerde que los **char** son enteros y se los puede mostrar como tales con los "type" "d" / "x" / "X" / "o" (ver más adelante).

El siguiente *trozo* de código corresponde al anterior *trozo* de salida por pantalla.

```

cadenas.c [pruebasC] - Code::Blocks 17.12
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help
<global>
usoDeCadenasDeControlDeFormato(void) : void
Management
Workspace
pruebasC
Sources
cadenas.c
flotantes.c
main.c
menu.c
Headers
cadenas.h
flotantes.h
main.h
menu.h

1
2 #include "cadenas.h"
3
4
5 void _mostrarCharConFlagYWidth()
6 {
7     char c2 = 75,
8         cadContrImpr[100];
9
10    puts("Mostrando un char con [flags] y [width]:\n"
11         "-----");
12    printf("Mostrando con:\n"
13           "%3c : \"%3c\" (solo [width])\n"
14           "%-3c: \"%-3c\" (con [flag] y [width])\n", c2, c2);
15    printf("Mostrando el caracter y su numero de ASCII en decimal, octal"
16           " y hexadecimal:\n"
17           "%c \"%c\" - %d \"%d\" - %o \"%o\" - %x \"%x\" - %X \"%X\" \n",
18           c2, c2, c2, c2, c2);
19    puts("Se copia en un array de char una cadena de conversion ");
  
```

NOTE QUE EN EL CÓDIGO ANTERIOR:

- una cadena de caracteres (p. ej.: líneas 10 y 11), comienza y termina con el carácter ("), si se necesita continuar la cadena (en el mismo renglón o uno o varios renglones más abajo), basta con escribir la / s siguiente / s cadena / s de caracteres comenzando y terminando con el carácter (").
- la función **puts** muestra por pantalla la cadena de caracteres tal como la escribimos y después de hacerlo, se va al siguiente *renglón*.
- se utiliza la secuencia de escape "\n" (nueva línea), que produce un salto de *renglón*.



- se utiliza "\" (en la línea 13 y otras), porque las comillas ("), inician y terminan cadenas de caracteres, y se quiere mostrar por pantalla las (") encerrando lo que se muestra por pantalla.
- se utiliza "%" (en la línea 13 y otras), porque con eso la función `printf` muestra un solo carácter "%", de lo contrario interpreta que comienza un especificador de control de formato para el que necesitará un argumento y si no lo tiene, suele producir una violación de memoria.
- para los especificadores "**type**" (tipo de dato), para enteros, por ahora diremos lo que ya conoce: "**d**" muestra un entero en decimal, en tanto que "**x**" en hexadecimal con las letras en minúscula, lo mismo con "**X**" pero en mayúscula y finalmente "**o**" lo muestra en octal.
- el tamaño de campo (`[width]`), en lugar de estar expresamente indicado con una constante literal (*hardcoded*), se puede utilizar un carácter "\*" por lo que deberá ponerse un argumento más (entero), antes del argumento correspondiente al `char`.

Por ejemplo:

```
char    c2 = 'K';      // siguiendo el código anterior
int     ancho = 11;    // width en castellano

printf("Se muestra el carácter %*c en un tamaño de campo"
      " de %d posiciones alineado a la derecha\n",
      ancho, c2, ancho);
printf("Se muestra el carácter %-*c en un tamaño de campo"
      " de %d posiciones alineado a la izquierda\n",
      ancho, c2, ancho);
```



UNLaM

Dto. Ingeniería e Investigaciones Tecnológicas

Codifique el siguiente código, pruébelo y saque sus conclusiones la salida por pantalla debería ser suficiente:

```

42 void _mostrarCharControl(void)
43 {
44     puts("Caracteres especiales o caracteres de control:\n"
45         "-----");
46     printf("%d es el nro de ASCII de \"%c\" (caracter nulo)\n", '\0', '\0');
47     printf("%d es el nro de ASCII de \"%c\" (beep alarm)\n", '\a', '\a');
48     printf("%d es el nro de ASCII de \"%c\" (back space)\n", '\b', '\b');
49     printf("%d es el nro de ASCII de \"%c\" (tab)\n", '\t', '\t');
50     printf("%d es el nro de ASCII de \"%c\" (nueva linea)\n", '\n', '\n');
51     printf("%d es el nro de ASCII de \"%c\" (tab vertical)\n", '\v', '\v');
52     printf("%d es el nro de ASCII de \"%c\" (salto de pagina)\n", '\f', '\f');
53     printf("%d es el nro de ASCII de \"%c\" (retorno de carro)\n", '\r', '\r');
54     printf("%d es el nro de ASCII de \"%c\" "
55         " (Marca fisica de fin-de-archivo)\n", 26, 26);
56     printf("Tenga en cuenta que los numeros de ASCII del 0 al 31 inclusive "
57         "son los llamados\n"
58         "caracteres de control. Por ejemplo: a un dispositivo externo como "
59         "una impresor-\n"
60         "ra de matriz de puntos se le envian caracteres de control para "
61         "iniciar un cam-\n"
62         "bio de tamaño de letra, interlineado, etc. Esto dependiendo "
63         "del tipo de dispo-\n"
64         "sitivo y su protocolo de comunicacion.\n");
65     puts("\n");
66 }

```

```

C:\Users\USER\Desktop\pruebasC\bin\Debug\pruebasC.exe
Caracteres especiales o caracteres de control:
-----
0 es el nro de ASCII de " " (caracter nulo)
7 es el nro de ASCII de "" (beep alarm)
8 es el nro de ASCII de " " (back space)
9 es el nro de ASCII de " " (tab)
10 es el nro de ASCII de "
" (nueva linea)
11 es el nro de ASCII de "□" (tab vertical)
12 es el nro de ASCII de "□" (salto de pagina)
" (retorno de carro) de "
26 es el nro de ASCII de "□" (Marca fisica de fin-de-archivo)
Tenga en cuenta que los numeros de ASCII del 0 al 31 inclusive son los llamados
caracteres de control. Por ejemplo: a un dispositivo externo como una impresor-
ra de matriz de puntos se le envian caracteres de control para iniciar un cam-
bio de tamaño de letra, interlineado, etc. Esto dependiendo del tipo de dispo-
sitivo y su protocolo de comunicacion.

```



UNLaM

Dto. Ingeniería e Investigaciones Tecnológicas

---

Recordemos que:

Los especificadores de formato están compuestos, en general, por la secuencia:

**`%[flags][width][.precision][size]type`**

### **Especificadores de formato para los array de char**

#### **Uso del tipo (`type`) de conversión de cadenas --> `%s`**

Los array de `char` (a los que usualmente nos referimos como cadenas de caracteres), no son un tipo de dato, pero el Lenguaje C implementa toda una batería de funciones para tratar con las cadenas de caracteres que ellos representan. Y la familia de funciones `printf` no es la excepción.

- **[flags]**: el único válido es el (-), guion del medio, que permite alinear a izquierda dentro de un *campo* más grande que la cadena (o el recorte de la misma -ver **[.precision]**); en caso de no querer alinear a izquierda, no lleva nada.
- **[width]**: tamaño de campo en que se muestra la cadena. Si es mayor que el tamaño de la cadena quedará alineado a derecha, salvo que esté indicado el **[flags]** (-).
- **[.precision]**: máxima cantidad de caracteres a mostrar.

Los dos enteros para tamaño de campo y la cantidad máxima son opcionales e independientes entre sí. Se pueden reemplazar en la cadena de conversión de formato por (\*), con lo que deberá haber una constante literal entera o una variable entera, por cada asterisco que se indique, como argumentos para el `printf`.





UNLaM

Dto. Ingeniería e Investigaciones Tecnológicas

Vea la siguiente salida por pantalla, y a continuación pruebe el código que la generó, despierte su parte lúdica y *juegue* un poco. Practicar es la mejor forma de fijar los conocimientos.

```

C:\Users\USER\Desktop\pruebas\bin\Debug\pruebasC.exe
La cadena "Hola que tal" tiene 12 caracteres
/* con tamaño de campo fijo, variable y fijo sin alinear */
"   Hola que tal"
"   Hola que tal"
/* con tamaño de campo fijo, variable y fijo alineado a izq. */
"Hola que tal"
"Hola que tal"
/* campo fijo y variable, con precisión, sin alinear */
"   Hola qu"
"   Hola qu"
/* campo fijo y variable, con precisión, alineado */
"Hola qu"
"Hola qu"
/* solo con precisión muestra esa cantidad de caracteres */
"Hola qu"
"Hola qu"
/* con campo menor que la longitud muestra la cadena completa */
"Hola que tal"   con: campo = 7
"Hola que tal"   con: campo = 7

```

```

main.c  x  main.h  x  flotantes.c  x  flotantes.h  x  menu.c  x  menu.h  x  cadenas.c  x  cadenas.h  x
90  void _mostrarCadenas(void)
91  {
92      char    cad[] = { "Hola que tal" };
93      int     campo,
94      prec;
95
96      printf("La cadena \"%s\" tiene %d caracteres\n", cad, strlen(cad));
97      campo = 20;
98      puts("/* con tamaño de campo fijo, variable y fijo sin alinear */");
99      printf("\n%20s\n", cad);
100     printf("\n\"%s\"\n", campo, cad);
101     puts("/* con tamaño de campo fijo, variable y fijo alineado a izq. */");
102     printf("\n%-20s\n", cad);
103     printf("\n\"%-s\"\n", campo, cad);
104     puts("/* campo fijo y variable, con precisión, sin alinear */");
105     prec = 7;
106     printf("\n%20.7s\n", cad);
107     printf("\n\"%.7s\"\n", campo, prec, cad);
108     puts("/* campo fijo y variable, con precisión, alineado */");
109     prec = 7;
110     printf("\n%-20.7s\n", cad);
111     printf("\n\"%-7s\"\n", campo, prec, cad);
112     puts("/* solo con precisión muestra esa cantidad de caracteres */");
113     printf("\n\"%.7s\"\n", cad);
114     printf("\n\"%.7s\"\n", prec, cad);
115     puts("/* con campo menor que la longitud muestra la cadena completa */");
116     printf("\n\"%7s\"   con: campo = 7\n", cad);
117     printf("\n\"%7s\"   con: campo = %d\n", prec, cad, prec);
118
119
120
121

```

Recordemos nuevamente que:



UNLaM

Dto. Ingeniería e Investigaciones Tecnológicas

Repetimos que los especificadores de formato están compuestos, en general, por la secuencia:

**%[flags][width][.precision][size]type**

### Especificadores de formato para los enteros

#### Uso del tipo (type) de conversión de enteros --> %d

Para el tipo de dato en el caso de enteros, no sólo se utiliza el (type) 'd':

Enteros (tenga en cuenta que en bases distintas de la base 10, el signo no tiene sentido).

- '%d' y '%i' -base 10 o decimal para enteros con signo (además de '%u')
- '%o' – base 8 u octal
- '%x' – base 16 o hexadecimal (0 ... 9 y a ... f)
- '%X' – base 16 o hexadecimal (0 ... 9 y A ... F)
- '%u' – base 10 o decimal, enteros sin signo (unsigned)

Tenga en cuenta que en las bibliotecas está la declaración de un tipo de dato **size\_t** que habitualmente es un **unsigned**. O **unsigned long**. Ambos para este compilador son de 4 Bytes. En **stddef.h** se encuentra la declaración:

```

206  /* __size_t is a typedef on FreeBSD 5, must not trash it. */
207  #elif defined (__VMS)
208  /* __size_t is also a typedef on VMS. */
209  #else
210  #define __size_t
211  #endif
212  #ifndef __SIZE_TYPE__
213  #define __SIZE_TYPE__ long unsigned int
214  #endif
215  #if !(defined (__GNU__) && defined (size_t))
216  typedef __SIZE_TYPE__ size_t;
217  #ifdef __BEOS__
218  typedef long __size_t;
219  #endif /* __BEOS__ */

```



UNLaM

Dto. Ingeniería e Investigaciones Tecnológicas

Por lo que el siguiente código cumplirá con su cometido con algunas salvedades ...

```

205 void _mostrarEnterosDIUXX(void)
206 {
207     int     ent1  = 2147483647;
208           ent2  = -2147483648;
209     unsigned uEnt  = 4294967295u;
210     size_t   stEnt = 4294967295u;
211
212     puts("/* maximo valor entero (ent1) */");
213     printf("%d\n" - "%i\n" <-- con %d y %i\n"
214           "%o\n" - "%x\n" - "%X\n" - "%u\n" <-- con %o %x %X %u\n",
215           ent1, ent1, ent1, ent1, ent1, ent1);
216     puts("/* minimo valor entero (ent2) */");
217     printf("%d\n" - "%i\n" <-- con %d y %i\n"
218           "%o\n" - "%x\n" - "%X\n" - "%u\n" <-- con %o %x %X %u\n",
219           ent2, ent2, ent2, ent2, ent2, ent2);
220     puts("/* mostrando un unsigned (uEnt) */");
221     printf("%d\n" - "%i\n" <-- con %d y %i\n"
222           "%o\n" - "%x\n" - "%X\n" - "%u\n" <-- con %o %x %X %u\n",
223           uEnt, uEnt, uEnt, uEnt, uEnt, uEnt);
224     puts("/* mostrando un size_t (stEnt) */");
225     printf("%d\n" - "%i\n" <-- con %d y %i\n"
226           "%o\n" - "%x\n" - "%X\n" - "%u\n" <-- con %o %x %X %u\n",
227           stEnt, stEnt, stEnt, stEnt, stEnt, stEnt);

```

Y su salida por pantalla será ...

```

C:\Users\USER\Desktop\pruebasC\bin\Debug\pruebasC.exe
/* maximo valor entero (ent1) */
"2147483647" - "2147483647" <-- con %d y %i
"1777777777" - "7fffffff" - "7FFFFFFF" - "2147483647" <-- con %o %x %X %u
/* minimo valor entero (ent2) */
"-2147483648" - "-2147483648" <-- con %d y %i
"2000000000" - "80000000" - "800000000" - "2147483648" <-- con %o %x %X %u
/* mostrando un unsigned (uEnt) */
"-1" - "-1" <-- con %d y %i
"3777777777" - "ffffffff" - "FFFFFFFF" - "4294967295" <-- con %o %x %X %u
/* mostrando un size_t (stEnt) */
"-1" - "-1" <-- con %d y %i
"3777777777" - "ffffffff" - "FFFFFFFF" - "4294967295" <-- con %o %x %X %u

```

NOTE que al mostrar el máximo y el mínimo valor entero (dos primeros **printf**)

- **%d** o **%i**, se obtienen los resultados esperados.
- **%o**, **%x** y **%X** sin ninguna objeción, muestran con la configuración de bits de las variables (tal como si fueran **unsigned**).
- **%u** es un caso especial, hace lo mismo que los tres anteriores mostrando (los cuatro), la variable convertida a **unsigned**, (**printf** hace internamente esta conversión).

Por lo tanto: **%u** es un especificador (**type**) específico de los **unsigned**.



UNLaM

Dto. Ingeniería e Investigaciones Tecnológicas

NOTE que al mostrar el máximo unsigned y size\_t (dos últimos `printf`), con ...

- `%d` o `%i`, se obtienen resultados *inesperados* por quien no tiene en cuenta que muestra como entero con signo más allá del hecho de ser variables sin signo (ambas son de 4 Bytes), porque salvo en base 10 el signo no tiene sentido.

- `%o`, `%x`, `%X` y `%u` son especificadores de tipo **unsigned** y muestran como tales.

En definitiva, un **unsigned** o un **size\_t** **deben** mostrarse con `%u`.

Dado el siguiente trozo de código, dónde todas las variables a mostrar son enteras ...

```

231 void _mostrarEnteros(void)
232 {
233     long long entLL = 0x51235123C123A1B3;
234     long entL = entLL;
235     int entI = entL;
236     short entS = entI;
237     char entC = entS;
238
239     printf("%I64d\\\" <-- entLL\\n\"
240           \"%ld\\\" <-- entL\\n\"
241           \"%d\\\" <-- entI\\n\"
242           \"%hd\\\" <-- entS\\n\"
243           \"%d\\\" <-- entC\\n\",
244           entLL, entL, entI, entS, entC);
245

```

... saque conclusiones acerca de lo que hace y por qué (recuerde que los **long int** y los **int** se almacenan en 4 bytes y los **short int** en 2 ...

```

C:\Users\USER\Desktop\pruebasC\bin\Debug\pruebasC.exe
"5846605955263078835" <-- entLL
"-1054629453" <-- entL
"-1054629453" <-- entI
"-24141" <-- entS
"-77" <-- entC

```

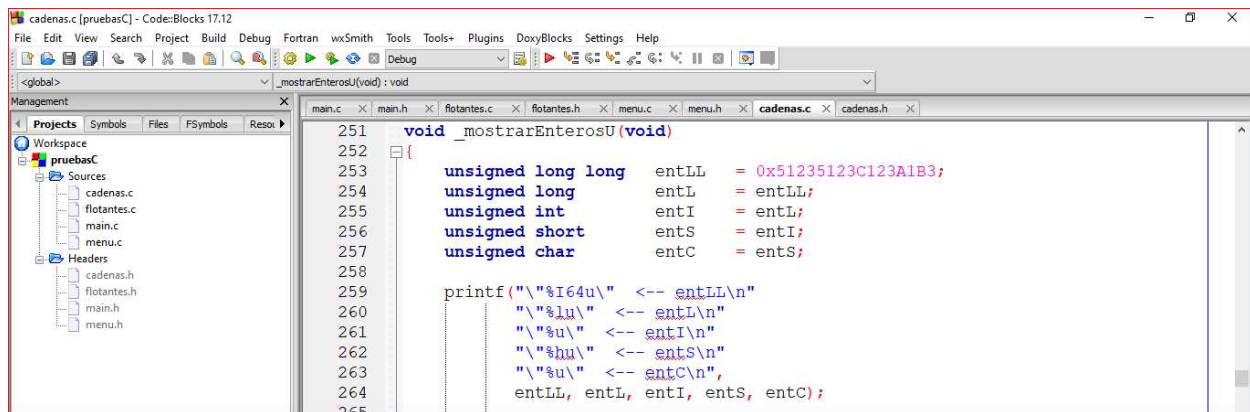
Notó el especificador de tamaño mencionado como parámetro opcional [**size**] antes del (o prefijo del) especificador de tipo (**type**). Este es el prefijo a utilizar para indicar cómo se muestran.



UNLaM

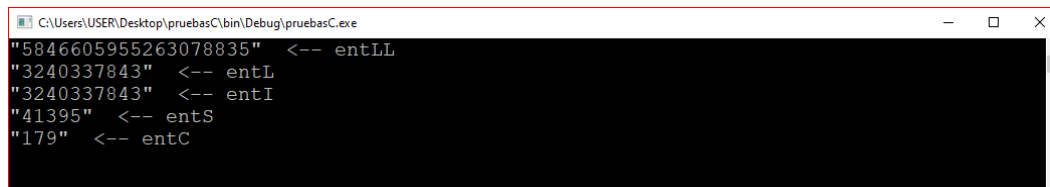
Dto. Ingeniería e Investigaciones Tecnológicas

Dado el siguiente trozo de código, dónde todas las variables a mostrar son enteras sin signo ...



```
251 void _mostrarEnterosU(void)
252 {
253     unsigned long long entLL = 0x51235123C123A1B3;
254     unsigned long entL = entLL;
255     unsigned int entI = entL;
256     unsigned short entS = entI;
257     unsigned char entC = entS;
258
259     printf("%I64u\\\" <-- entLL\\n\"
260           \"%lu\\\" <-- entL\\n\"
261           \"%u\\\" <-- entI\\n\"
262           \"%hu\\\" <-- entS\\n\"
263           \"%u\\\" <-- entC\\n\",
264           entLL, entL, entI, entS, entC);
265 }
```

... y su salida por pantalla resulta ...



```
C:\Users\USER\Desktop\pruebasC\bin\Debug\pruebasC.exe
"5846605955263078835" <-- entLL
"3240337843" <-- entL
"3240337843" <-- entI
"41395" <-- entS
"179" <-- entC
```

Se utiliza el especificador de tipo (**type**), precedido por el especificador de tamaño [**size**], que corresponde. Para los **char** y los **short** no es necesario, al menos con este compilador, para mostrar en decimal los **unsigned short int** ni los **unsigned char**. Pero hábituese a las buenas prácticas.

NOTA: en otros compiladores se utiliza '%11d' y '%11u' para los **long long**, pero este compilador no lo reconoce.





UNLaM

Dto. Ingeniería e Investigaciones Tecnológicas

En cuanto al uso de `[.precision]`, no es válido para los tipos de datos enteros.

Los `[flags]` válidos son: ' ' (blanco), '-', '+', y '0'.

Si se indica un `[width]` mayor que la representación del número,

- ' ' (blanco) deja el lugar en blanco del signo si es positivo
- '-' alinea a la izquierda, por defecto alinea a derecha
- '+' exhibe siempre el signo (positivo o negativo)
- '0' completa con ceros a la izquierda

```

271 void _mostrarEnterosF(void)
272 {
273     int      ent1   = 1550,
274             ent2   = -1379,
275             campo  = 15;
276
277     printf("\n\"%d\" \"%-d\" \"%+d\" \"%+d\"\\n\"
278           \"%0*d\" \"%+0*d\" \"%0*d\"\\n\",
279           campo, ent1, campo, ent1, campo, ent1, campo, ent1,
280           campo, ent1, campo, ent1, campo, ent1);
281     printf("\n\"%d\" \"%-d\" \"%+d\" \"%+d\"\\n\"
282           \"%0*d\" \"%+0*d\" \"%0*d\"\\n\",
283           campo, ent2, campo, ent2, campo, ent2, campo, ent2,
284           campo, ent2, campo, ent2, campo, ent2);
285

```

Output window (C:\Users\USER\Desktop\pruebasC\bin\Debug\pruebasC.exe):

```

1550" "1550" "+" "+1550" "+1550" "
" 00000000001550" "+00000000001550" "00000000001550"
-1379" "-1379" " " "-1379" "-1379" "
"-00000000001379" "-00000000001379" "-00000000001379"

```

Las cadenas de conversión de formato de uso más habitual con tamaño de campo son la primera y la última: `(%*d)` y `(%0*d)` o si no usa el argumento reemplazable, con el tamaño de campo en lugar del `(*)`, p. ej.: `(%8d)` o `(%08d)` para mostrar un DNI o grabarlo en un archivo.





UNLaM

Dto. Ingeniería e Investigaciones Tecnológicas

Repetimos nuevamente:

Los especificadores de formato están compuestos, en general, por la secuencia:

**%[flags][width][.precision][size]type**

### Especificadores de formato para punto flotante

**Uso del tipo (*type*), de conversión de punto flotante --> %f**

Para las variables de punto flotante, además del indicador de tipo (*type*), '*f*', se pueden utilizar (entre otros, dependiendo del compilador): '*e*', '*E*', '*g*', y '*G*'. Estos cuatro son con notación científica. Los dos últimos ('*g*', y '*G*'), si la representación (con seis dígitos), entra en el campo, no usa la notación científica. En nuestra materia no los utilizaremos.

```

338 void _mostrarPtoFlotante(void)
339 {
340     float    flo1 = 1234.567898765432f,
341             flo2 = 1234567898.765432f;
342     double   dob1 = 9876.5432198765432198,
343             dob2 = 9876543219876543.2198;
344
345     printf("\n%f\n \"%e\" \"%E\" \"%g\" \"%G\"\n",
346           flo1, flo1, flo1, flo1, flo1);
347     printf("\n%f\n \"%e\" \"%E\" \"%g\" \"%G\"\n",
348           flo2, flo2, flo2, flo2, flo2);
349     printf("\n%f\n \"%e\" \"%E\" \"%g\" \"%G\"\n",
350           dob1, dob1, dob1, dob1, dob1);
351     printf("\n%f\n \"%e\" \"%E\" \"%g\" \"%G\"\n",
352           dob2, dob2, dob2, dob2, dob2);
353 }
  
```

```

C:\Users\USER\Desktop\pruebasC\bin\Debug\pruebasC.exe
"1234.567871" "1.234568e+003" "1.234568E+003" "1234.57" "1234.57"
"1234567936.000000" "1.234568e+009" "1.234568E+009" "1.23457e+009" "1.23457E+009"
"9876.543220" "9.876543e+003" "9.876543E+003" "9876.54" "9876.54"
"9876543219876544.000000" "9.876543e+015" "9.876543E+015" "9.87654e+015" "9.87654E+015"
  
```

Recuerde la precisión de los '*float*' (siete dígitos), y de los '*double*' (15 dígitos).

Note que con este compilador el indicador de tamaño ('*l*'), lo utilizamos o no, muestra correctamente.



UNLaM

Dto. Ingeniería e Investigaciones Tecnológicas

Sólo nos concentraremos en los float y double dentro de rangos de parte entera con dos (a veces a lo sumo tres), decimales.

```

357 void _mostrarFlotanteFlagWidPrec(void)
358 {
359     float      flo1 = 987.5461111f,
360             flo2 = -flo1;
361
362     puts("Mostrando float:");
363     printf("%-15f" <-- a la izquierda\n
364           "%-15f" <-- a la izquierda\n
365           "%-15f" <-- a la izquierda, lugar para el signo\n
366           "%-15f" <-- a la izquierda, lugar para el signo\n
367           "%+-15f" <-- a la izquierda, signo siempre\n
368           "%+-15f" <-- a la izquierda, signo siempre\n
369           "%15f" <-- por defecto a la derecha\n
370           "%15f" <-- por defecto a la derecha\n",
371           flo1, flo2, flo1, flo2, flo1, flo2, flo1, flo2);
372     puts("En la materia utilizaremos estos formatos:");
373     printf("%9.2f" - "%9.2f" <-- habitualmente asi\n
374           "%09.2f" - "%09.2f" <-- ... o tambien asi\n",
375           flo1, flo2, flo1, flo2);
  
```

```

Mostrando float:
"987.546082" <-- a la izquierda
"-987.546082" <-- a la izquierda
" 987.546082" <-- a la izquierda, lugar para el signo
"-987.546082" <-- a la izquierda, lugar para el signo
"+987.546082" <-- a la izquierda, signo siempre
"-987.546082" <-- a la izquierda, signo siempre
"   987.546082" <-- por defecto a la derecha
"  -987.546082" <-- por defecto a la derecha

En la materia utilizaremos estos formatos:
"   987.55" - "  -987.55" <-- habitualmente asi
"000987.55" - "-00987.55" <-- ... o tambien asi
  
```

Note que, si el 'float' hubiera sido el resultado de un cálculo con más de dos decimales, al mostrarlo con dos posiciones para la parte decimal, sufrirá un redondeo. Haga sus pruebas con distintos valores.

Con un tamaño de campo de nueve posiciones de las cuales dos son para la parte decimal, una para el punto, una más para el signo (si en sus datos esto es posible), quedando cinco posiciones para la parte entera. Por lo que se puede mostrar perfectamente flotantes con decenas de miles y centésimos.

Si se requiere una mayor cantidad de dígitos exactos, se utilizarán variables del tipo 'double'.



UNLaM

Dto. Ingeniería e Investigaciones Tecnológicas

Con un ejemplo similar para variables del tipo 'double' ...

```

378 void _mostrarDoublesFlagWidPrec(void)
379 {
380     double    dob1 = 987654321123.456789,
381             dob2 = -dob1;
382
383     puts("Mostrando double:");
384     printf("%-22f" <-- a la izquierda\n
385            "%-22f" <-- a la izquierda\n
386            "%-22f" <-- a la izquierda, lugar para el signo\n
387            "%-22f" <-- a la izquierda, lugar para el signo\n
388            "%+22f" <-- a la izquierda, signo siempre\n
389            "%+22f" <-- a la izquierda, signo siempre\n
390            "%22f" <-- por defecto a la derecha\n
391            "%22f" <-- por defecto a la derecha\n
392            "%+22f" <-- por defecto a la derecha, signo siempre\n
393            "%+22f" <-- por defecto a la derecha, signo siempre\n",
394           dob1, dob2, dob1, dob2, dob1, dob2, dob1, dob2, dob1, dob2);
395
396     dob1 = 100222333.5546783169;
397     dob2 = -dob1;
398     printf("En la materia utilizaremos estos formatos para mostrar nueve
399           " digitos enteros con dos decimales de los double:\n
400           " dob1 = \"%12f\" - dob2 = \"%12f\"\n
401           " \"%13.2f\" - \"%13.2f\" <-- habitualmente asi\n
402           " \"%013.2f\" - \"%013.2f\" <-- ... o tambien asi\n",
403           dob1, dob2, dob1, dob2, dob1, dob2, dob1, dob2, dob1, dob2);

```

```

C:\Users\USER\Desktop\pruebasC\bin\Debug\pruebasC.exe
Mostrando double:
"987654321123.456790" <-- a la izquierda
"-987654321123.456790" <-- a la izquierda
" 987654321123.456790" <-- a la izquierda, lugar para el signo
"-987654321123.456790" <-- a la izquierda, lugar para el signo
"+987654321123.456790" <-- a la izquierda, signo siempre
"-987654321123.456790" <-- a la izquierda, signo siempre
"  987654321123.456790" <-- por defecto a la derecha
"-987654321123.456790" <-- por defecto a la derecha
"+987654321123.456790" <-- por defecto a la derecha, signo siempre
"-987654321123.456790" <-- por defecto a la derecha, signo siempre

En la materia utilizaremos estos formatos para mostrar nueve digitos enteros con dos
decimales de los double:
dob1 = "100222333.554678320000" - dob2 = "-100222333.554678320000"
" 100222333.55" - "-100222333.55" <-- habitualmente asi
"0100222333.55" - "-100222333.55" <-- ... o tambien asi

```

En definitiva, para los 'float' y los 'double', los mostraremos en la materia con:

- 'type' 'f'
- 'size' nada (aunque, con este compilador puede utilizar 'lf' con 'float' y 'double')
- '.precision' en nuestra materia utilizaremos 2, 3, o nada, al solo indicar '.', se omitirá la parte decimal (pruébelo)
- 'width' opcionalmente un tamaño de campo, según los casos
- 'flags' válidos '-' o nada para la alineación, '+' para que siempre muestre el signo, '0' (cero), para completar con cero a la izquierda

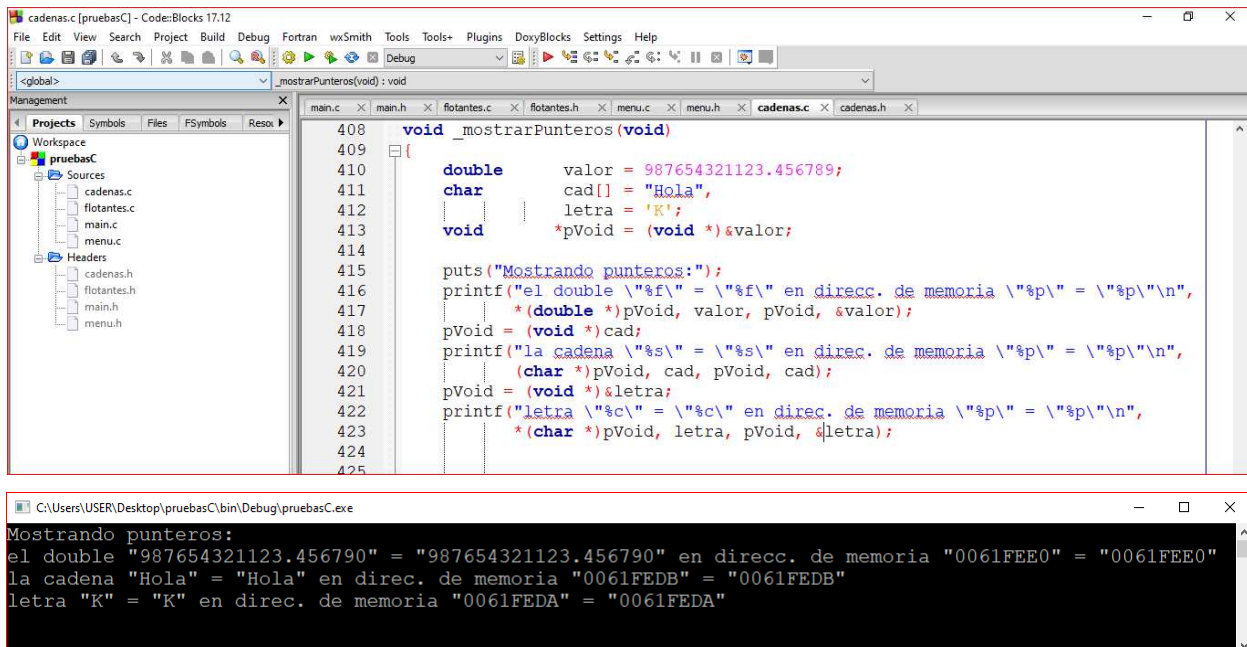
Finalmente:

Los especificadores de formato están compuestos, en general, por la secuencia:

**% [flags] [width] [.precision] [size] type**

### Especificadores de formato para punteros

Uso del tipo (*type*), de conversión de punto flotante --> **%p**



The screenshot shows a code editor with a project named 'pruebasC'. The code in 'main.c' defines a function `_mostrarPunteros(void)` that demonstrates pointer usage. It declares a `double` variable `valor`, a `char` array `cad`, a `char` variable `letra`, and a `void` pointer `pVoid`. The function prints the memory addresses of these variables using the `%p` format specifier. Below the code, a terminal window shows the output of the program, displaying the memory addresses for each variable.

```
408 void _mostrarPunteros(void)
409 {
410     double valor = 987654321123.456789;
411     char cad[] = "Hola",
412         letra = 'K';
413     void *pVoid = (void *)&valor;
414
415     puts("Mostrando punteros:");
416     printf("el double \"%f\" = \"%f\" en direcc. de memoria \"%p\" = \"%p\"\\n",
417           *(double *)pVoid, valor, pVoid, &valor);
418     pVoid = (void *)&cad;
419     printf("la cadena \"%s\" = \"%s\" en direc. de memoria \"%p\" = \"%p\"\\n",
420           (char *)pVoid, cad, pVoid, cad);
421     pVoid = (void *)&letra;
422     printf("letra \"%c\" = \"%c\" en direc. de memoria \"%p\" = \"%p\"\\n",
423           (char *)pVoid, letra, pVoid, &letra);
424 }
425
```

Mostrando punteros:  
el double "987654321123.456790" = "987654321123.456790" en direcc. de memoria "0061FEE0" = "0061FEE0"  
la cadena "Hola" = "Hola" en direc. de memoria "0061FEDB" = "0061FEDB"  
letra "K" = "K" en direc. de memoria "0061FEDA" = "0061FEDA"

En el caso de mostrar punteros, el uso de **[flags]**, **[width]**, **[.precision]** y **[size]** deja de tener sentido porque las direcciones de memoria se muestran en hexadecimal, de acuerdo con el sistema operativo y con el compilador empleados (y este último es de 32 bits).





UNLaM

Dto. Ingeniería e Investigaciones Tecnológicas

Agregue en el archivo "cadenas.c" ...

```

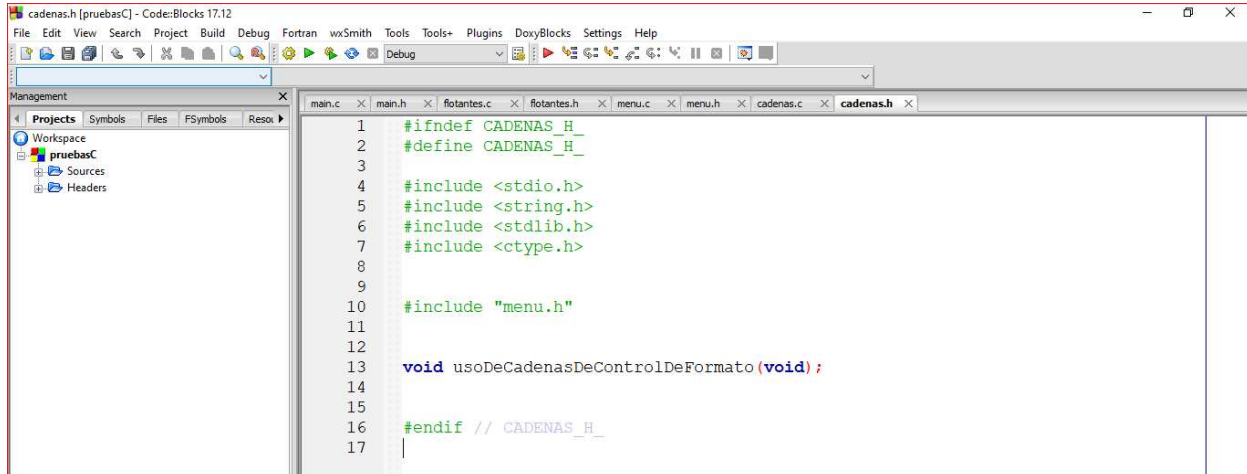
442 void usoDeCadenasDeControlDeFormato(void)
443 {
444     char opcion;
445
446     do
447     {
448         _mostrarMensGralCadContr();
449         opcion = menuConErr("- Sub Menu -\n"
450                             "Ver cadenas de control de impresion de:\n"
451                             "C / c - char\n"
452                             ". . . .\n"
453                             "T - Salir\n"
454                             "----> ",
455                             "cCtT");
456
457         switch(toupper(opcion))
458         {
459             case 'C': _mostrarCharConFlagYWidth(); break;
460             //
461         } while( opcion != 'T');
462     }
463 }
  
```

```

431 void _mostrarMensGralCadContr(void)
432 {
433     puts("Cadenas de Control de Impresion - Cadenas de Conversion de Formato.\n"
434         "===== \n"
435         "Son de la forma general:\n"
436         "%[flags][width][.precision][size]type\n"
437         "cuales se usan depende del tipo (type) de dato "
438         "de la variable a mostrar.");
439 }
440
  
```

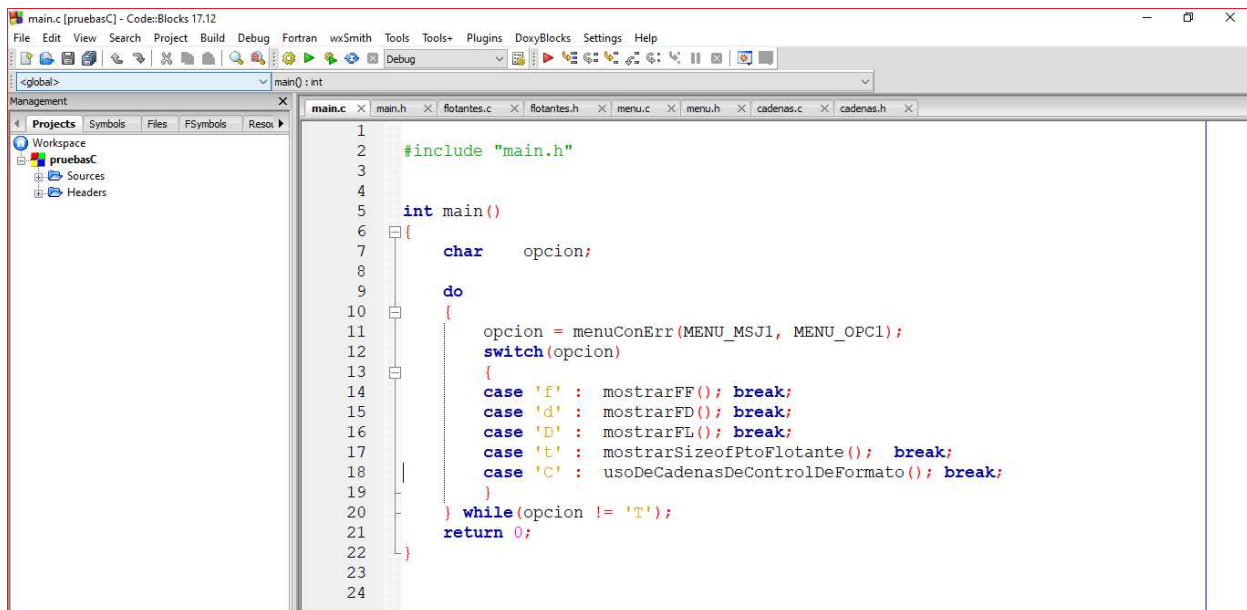
Desarrolle, antes de estas funciones, las funciones que prueban las cadenas de conversión de formato.

En "cadenas.h" tan sólo declare la función que despliega el submenú ...



```
1 #ifndef CADENAS_H
2 #define CADENAS_H
3
4 #include <stdio.h>
5 #include <string.h>
6 #include <stdlib.h>
7 #include <ctype.h>
8
9
10 #include "menu.h"
11
12
13 void usoDeCadenasDeControlDeFormato(void);
14
15
16 #endif // CADENAS_H
17
```

Y finalmente en "main.c" ...



```
1
2 #include "main.h"
3
4
5 int main()
6 {
7     char opcion;
8
9     do
10     {
11         opcion = menuConErr(MENU_MSJ1, MENU_OPC1);
12         switch(opcion)
13         {
14             case 'F' : mostrarFF(); break;
15             case 'd' : mostrarFD(); break;
16             case 'D' : mostrarFL(); break;
17             case 't' : mostrarSizeofPtoFlotante(); break;
18             case 'C' : usoDeCadenasDeControlDeFormato(); break;
19         }
20     } while(opcion != 'T');
21     return 0;
22 }
23
24
```

Es una pauta de estilo habitual, que las funciones que no se "publican" en su ".h", comience su "nombre" con '\_' (guión bajo). No es obligatorio.