SINTAXIS Y FUNCIONNES DE C

1.1. Gramática Léxica

1.1.1. Elementos Léxicos

```
<token> ->
     <palabra reservada> |
     <identificador> |
     <constante> |
     <literal de cadena> |
     <punctuator>
<token de preprocesamiento> ->
     <nombre de encabezado> |
     <identificador> |
     <número de preprocesador>|
     <constante carácter> |
     <literal de cadena> |
     <punctuator> |
     cada uno de los caracteres no-espacio-blanco que no sea uno de los anteriores
```

1.1.2. Palabras Reservadas

```
<palabra reservada> -> una de
     auto break case char const continue default do
     double else enum extern float for goto if
     int long register return short signed sizeof static
     struct switch typedef union unsigned void volatile while
```

1.1.3. Identificadores

```
<identificador> ->
     <no dígito> |
     <identificador> <no dígito> |
     <identificador> <dígito>
<no dígito> -> uno de
     _ a b c d e f g h i j k l m n o p q r s t u v w x y z
     A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
<dígito> -> uno de
     0 1 2 3 4 5 6 7 8 9
```

- Toda implementación debe distinguir, como mínimo, los primeros 31 caracteres de un identificador que actúa como nombre de variable, de función, de constante o de tipo.

- Dado que los identificadores constituyen un Lenguaje Regular, podemos describirlos mediante la definición regular que figura en el Ejemplo 6 del libro "Autómatas Finitos y Expresiones Regulares", página 113:

<letra> = [a-zA-Z] (cualquier letra minúscula o mayúscula del alfabeto reducido)
<dígito> = [0-9]

```
<subrayado> = _
<primer carácter> = <letra> | <subrayado>
<otro carácter> = <letra> | <dígito> | <subrayado>
<identificador> = <primer carácter> <otro carácter>*
```

1.1.4. Constantes

```
<constante> ->
     <constante entera> |
     <constante real> |
     <constante carácter> |
     <constante enumeración>
```

- En general, en computación las constantes enteras *no* son un subconjunto de las constantes reales.

Constante Entera

```
<constante entera> ->
     <constante decimal> <sufijo entero>? |
     <constante octal> <sufijo entero>? |
     <constante hexadecimal> <sufijo entero>?
<constante decimal> ->
     <dígito no cero> |
     <constante decimal> <dígito>
<dígito no cero> -> uno de
     1 2 3 4 5 6 7 8 9
<dígito> -> uno de
     0 1 2 3 4 5 6 7 8 9
<constante octal> ->
     0 |
     <constante octal> <dígito octal>
<dígito octal> -> uno de
     0 1 2 3 4 5 6 7
<constante hexadecimal> ->
     0x <dígito hexadecimal> |
     0X <dígito hexadecimal> |
     <constante hexadecimal> <dígito hexadecimal>
<dígito hexadecimal> -> uno de
     0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F
<sufijo entero> ->
     <sufijo "unsigned"> <sufijo "long">? |
     <sufijo "long"> <sufijo "unsigned">?
<sufijo "unsigned"> -> uno de
     u U
<sufijo "long"> -> uno de
     l L
```

- El tipo de una constante entera depende de su valor y será representada como primero corresponda, según la siguiente lista: **`int`**, **`unsigned int`**, **`long`**, **`unsigned long`**.

- El lenguaje de "Las constantes enteras en ANSI C" es regular; por lo tanto, podemos describirlo a través de una definición regular como la que figura en el Ejemplo 7 del libro "Autómatas Finitos y Expresiones Regulares", página 113:

```
<sufijo U> = u | U
<sufijo L> = l | L
```

```
<sufijo entero> =
     <sufijo U> |
     <sufijo L> |
     <sufijo U> <sufijo L> |
     <sufijo L> <sufijo U>
<dígito decimal> = [0-9]
<dígito decimal no nulo> = [1-9]
<dígito hexadecimal> = [0-9a-fA-F]
<dígito octal> = [0-7]
<prefijo hexadecimal> = 0x | 0X
<constante decimal> = <dígito decimal no nulo> <dígito decimal>*
<constante hexadecimal> = <prefijo hexadecimal> <dígito hexadecimal>+
<constante octal> = 0 <dígito octal>*
<constante incompleta> =
     <constante decimal> |
     <constante hexadecimal> |
     <constante octal>
<constante entera> = <constante incompleta> <sufijo entero>?
```

Constante Real

```
<constante real> ->
     <constante fracción> <parte exponente>? <sufijo real>? |
     <secuencia dígitos> <parte exponente> <sufijo real>?
<constante fracción> ->
     <secuencia dígitos>? . <secuencia dígitos> |
     <secuencia dígitos> .
<parte exponente> ->
     e <signo>? <secuencia dígitos> |
     E <signo>? <secuencia dígitos>
<signo> -> uno de + -
<secuencia dígitos> ->
     <dígito> |
     <secuencia dígitos> <dígito>
<dígito> -> uno de 0 1 2 3 4 5 6 7 8 9
<sufijo real> -> uno de f F l L
```

- Si no tiene sufijo, la constante real es **double**.

- En inglés esta constante es conocida como <floating-point-constant>, <constante de punto flotante>.

Constante Carácter

```
<constante carácter> ->
     '<carácter-c>' |
     '<secuencia de escape>'
<carácter-c> -> cualquiera excepto
     ' \
<secuencia de escape> ->
     <secuencia de escape simple> |
     <secuencia de escape octal> |
     <secuencia de escape hexadecimal>
<secuencia de escape simple> -> uno de
     \' \" \? \\ \a \b \f \n \r \t \v
<secuencia de escape octal> ->
     \<dígito octal> |
     \<dígito octal> <dígito octal> |
```

```
        \<dígito octal> <dígito octal> <dígito octal>
    <dígito octal> -> uno de
        0 1 2 3 4 5 6 7
    secuencia de escape hexadecimal> ->
        \x<dígito hexadecimal> |
        \x <dígito hexadecimal> <dígito hexadecimal>
    \x admite únicamente la x minúscula.
    <dígito hexadecimal> -> uno de
        0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F
```

Constante Enumeración

```
<constante enumeración> ->
    <identificador>
```

1.1.5. Constantes Cadena

```
<constante cadena> ->
    "<secuencia caracteres-s>"
<secuencia caracteres-s> ->
    <carácter-s> |
    <secuencia caracteres-s> <carácter-s>
<carácter-s> ->
    cualquiera excepto " \ |
    <secuencia de escape>
<secuencia de escape> ->
    <secuencia de escape simple> |
    <secuencia de escape octal> |
    <secuencia de escape hexadecimal>
<secuencia de escape simple> -> uno de
    \' \" \? \\ \a \b \f \n \r \t \v
<secuencia de escape octal> ->
    \ <dígito octal> |
    \ <dígito octal> <dígito octal> |
    \ <dígito octal> <dígito octal> <dígito octal>
<dígito octal> -> uno de
    0 1 2 3 4 5 6 7
<secuencia de escape hexadecimal> ->
    \x <dígito hexadecimal> |
    \x <dígito hexadecimal> <dígito hexadecimal
\x admite únicamente la x minúscula.
<dígito hexadecimal> -> uno de
    0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F
```

- En inglés, estas constantes son referidas como <string literals>, <literales de cadena>.

- Notar que no están agrupadas con el resto de las constantes.

1.1.6. Punctuators – Caracteres de Puntuación

```
punctuator -> uno de
    [ ] ( ) { } . ->
    ++ -- & * + - ~ !
    / % << >> < > <= >= == != ^ | && ||
    = *= /= %= += -= <<= >>= &= ^= |=
    ? : ; ... , # ##
```

- La mayoría cumple el papel de operador, ver sección "*Precedencia y Asociatividad de los 45 Operadores*".

1.1.7. Nombre de Encabezados

```
<nombre de encabezado> ->
     < <secuencia de caracteres h> > |
     " <secuencia de caracteres q> "
<secuencia de caracteres h> ->
     <carácter h> |
     <secuencia de caracteres h> <carácter h>
<carácter h> ->
     cualquier miembro del conjunto de caracteres fuente excepto el carácter nueva-
     línea y el carácter >
<secuencia de caracteres q> ->
     <carácter q> |
     <secuencia de caracteres q> <carácter q>
<carácter q> ->
     cualquier miembro del conjunto de caracteres fuente excepto el carácter nueva-
     línea y el carácter "
```

1.1.8. Números de Preprocesador

```
<número de preprocesador> ->
     <dígito> |
     . <dígito> |
     <número de preprocesador> <dígito> |
     <número de preprocesador>  <identificador no dígito> |
     <número de preprocesador> e <sign> |
     <número de preprocesador> E <sign>|
     <número de preprocesador> p <sign>|
     <número de preprocesador> P <sign>|
     <número de preprocesador> .
```

1.2. Gramática de Estructura de Frases

1.2.1. Expresiones

```
<expresión> ->
     <expresión de asignación> |
     <expresión> , <expresión de asignación>
<expresión de asignación> ->
     <expresión condicional> |
     <expresión unaria> <operador asignación> <expresión de asignación>
<expresión condicional> ->
     <expresión O lógico> |
     <expresión O lógico> ? <expresión> : <expresión condicional>
<operador asignación> -> uno de
     = *= /= %= += -= <<= >>= &= ^= |=
<expresión O lógico> ->
     <expresión Y lógico> |
     <expresión O lógico> || <expresión Y lógico>
```

```
<expresión Y lógico> ->
     <expresión O inclusivo> |
     <expresión Y lógico> && <expresión O inclusivo>
<expresión O inclusivo> ->
     <expresión O excluyente> |
     <expresión O inclusivo> | <expresión O excluyente>
<expresión O excluyente> ->
     <expresión Y> |
     <expresión O excluyente> ^ <expresión Y>
<expresión Y> ->
     <expresión de igualdad> |
     <expresión Y> & <expresión de igualdad>
<expresión de igualdad> ->
     <expresión relacional> |
     <expresión de igualdad> == <expresión relacional> |
     <expresión de igualdad> != <expresión relacional>
<expresión relacional> ->
     <expresión de corrimiento> |
     <expresión relacional> < <expresión de corrimiento> |
     <expresión relacional> > <expresión de corrimiento> |
     <expresión relacional> <= <expresión de corrimiento> |
     <expresión relacional> >= <expresión de corrimiento>
<expresión de corrimiento> ->
     <expresión aditiva> |
     <expresión de corrimiento> << <expresión aditiva> |
     <expresión de corrimiento> >> <expresión aditiva>
<expresión aditiva> ->
     <expresión multiplicativa> |
     <expresión aditiva> + <expresión multiplicativa> |
     <expresión aditiva> - <expresión multiplicativa>
<expresión multiplicativa> ->
     <expresión de conversión> |
     <expresión multiplicativa> * <expresión de conversión> |
     <expresión multiplicativa> / <expresión de conversión> |
     <expresión multiplicativa> % <expresión de conversión>
<expresión de conversión> ->
     <expresión unaria> |
     (<nombre de tipo>) <expresión de conversión>
<expresión unaria> ->
     <expresión sufijo> |
     ++ <expresión unaria> |
     -- <expresión unaria> |
     <operador unario> <expresión de conversión> |
     sizeof <expresión unaria> |
     sizeof (<nombre de tipo>)
<nombre de tipo> está descripto más adelante, en la sección Declaraciones.
<operador unario> -> uno de & * + - ~ !
<expresión sufijo> ->
     <expresión primaria> |
     <expresión sufijo> [<expresión>] | /* arreglo */
     <expresión sufijo> (<lista de argumentos>?) | /* invocación */
     <expresión sufijo> . <identificador> |
     <expresión sufijo> -> <identificador> |
     <expresión sufijo> ++ |
     <expresión sufijo> --
<lista de argumentos> ->
     <expresión de asignación> |
     <lista de argumentos> , <expresión de asignación>
```

```
<expresión primaria> ->
    <identificador> |
    <constante> |
    <constante cadena> |
    (<expresión>)
```

Expresiones Constantes

```
<expresión constante> -> <expresión condicional>
```

- Las expresiones constantes pueden ser evaluadas durante la traducción en lugar de durante la ejecución.

Precedencia y Asociatividad de los 45 Operadores

ID Asociatividad Izquierda-Derecha, DI Asociatividad Derecha-Izquierda

1	ID	operadores de acceso	`( )`		invocación a función
			`[ ]`		subíndice de arreglo
			`.`		acceso a struct y a union
			`->`		acceso a struct y a union
2	DI	operadores unarios (operan	`+`	`-`	signos positivo y negativo
		sobre un solo operando)	`~`		complemento por bit
			`!`		NOT lógico
			`&`		dirección de
			`*`		"indirección"
			`++`		pre-incremento
			`--`		pre-decremento
			`sizeof`		tamaño de
			(*tipo*)		conversión explícita
3	ID	operadores multiplicativos	`*`		multiplicación
			`/`		cociente
			`%`		módulo o resto
4	ID	operadores aditivos	`+`	`-`	suma y resta
5	ID	operadores de desplazamiento	`<<`		desplazamiento de bits a izquierda
			`>>`		desplazamiento de bits a derecha
6	ID	operadores relacionales	`<`	`>`	
			`<=`	`>=`	
7	ID	operadores de igualdad	`==`	`!=`	igual a y distinto de
8	ID	operadores binarios por bit	`&`		AND
9	ID		`^`		OR exclusivo
10	ID		`|`		OR
11	ID	operadores binarios lógicos	`&&`		AND
12	ID		`||`		OR
13	DI	operador condicional	`? :`		(único que opera sobre 3 operandos)
14	DI	operadores de asignación	`=`		
			`*=`	`/=`	`%=` `+=` `-=`
			`<<=`	`>>=`	`&=` `^=` `|=`
15	ID	operador concatenación expresiones	`,`		"coma"

- Los operadores **&&** , **||** y **,** "coma" son los únicos que garantizan que los operandos sean evaluados en un orden determinado (de izquierda a derecha).

- El operador condicional (**? :**) evalúa solo un operando, entre el 2do. y el 3ro., según corresponda.

1.2.2. Declaraciones

- Una declaración especifica la interpretación y los atributos de un conjunto de identificadores.

- Si una declaración provoca reserva de memoria, se la llama *definición*.

```
<declaración> ->
     <especificadores de declaración> <lista de declaradores>?
<especificadores de declaración> ->
     <especificador de clase de almacenamiento> <especificadores de declaración>? |
     <especificador de tipo> <especificadores de declaración>? |
     <calificador de tipo> <especificadores de declaración>?
<lista de declaradores> ->
     <declarador> |
     <lista de declaradores> , <declarador>
<declarador> ->
     <decla> |
     <decla> = <inicializador>
<inicializador> ->
     <expresión de asignación> | /* Inicialización de tipos escalares */
     {<lista de inicializadores>} | /* Inicialización de tipos estructurados */
     {<lista de inicializadores> , }
<lista de inicializadores> ->
     <inicializador> |
     <lista de inicializadores> , <inicializador>
<especificador de clase de almacenamiento> -> uno de
     typedef static auto register extern
```

- No más de un especificador de clase de almacenamiento> puede haber en una declaración

```
<especificador de tipo> -> uno de
     void char short int long float double signed unsigned
     <especificador de "struct" o "union">
     <especificador de "enum">
     <nombre de "typedef">
<calificador de tipo> -> const | volatile
<especificador de "struct" o "union"> ->
     <"struct" o "union"> <identificador>? {<lista de declaraciones "struct">} |
     <"struct" o "union"> <identificador>
<"struct" o "union"> -> struct | union
<lista de declaraciones "struct"> ->
     <declaración "struct"> |
     <lista de declaraciones "struct"> <declaración "struct">
<declaración "struct"> ->
     <lista de calificadores> <declaradores "struct"> ;
<lista de calificadores> ->
     <especificador de tipo> <lista de calificadores>? |
     <calificador de tipo> <lista de calificadores>?
<declaradores "struct"> ->
     <decla "struct"> |
     <declaradores "struct"> , <decla "struct">
<decla "struct"> ->
     <decla> |
     <decla>? : <expresión constante>
<decla> -> <puntero>? <declarador directo>
```

```
<puntero> ->
     * <lista calificadores tipos>? |
     * <lista calificadores tipos>? <puntero>
<lista calificadores tipos> ->
     <calificador de tipo> |
     <lista calificadores tipos> <calificador de tipo>
<declarador directo> ->
     <identificador> |
     ( <decla> ) |
     <declarador directo> [ <expresión constante>? ] |
     <declarador directo> ( <lista tipos parámetros> ) /* Declarador nuevo estilo */
     <declarador directo> ( <lista de identificadores>? ) /* Declarador estilo
     obsoleto */
<lista tipos parámetros> ->
     <lista de parámetros> |
     <lista de parámetros> ,  . . .
<lista de parámetros> ->
     <declaración de parámetro> |
     <lista de parámetros> , <declaración de parámetro>
<declaración de parámetro> ->
     <especificadores de declaración> <decla> | /* Parámetros "nombrados" */
     <especificadores de declaración> <declarador abstracto>? /* Parámetros
     "anónimos" */
<lista de identificadores> ->
     <identificador> |
     <lista de identificadores> , <identificador>
<especificador de "enum"> ->
     enum <identificador>? { <lista de enumeradores> } |
     enum <identificador>
<lista de enumeradores> ->
     <enumerador> | <lista de enumeradores> , <enumerador>
<enumerador> ->
     <constante de enumeración> |
     <constante de enumeración> = <expresión constante>
<constante de enumeración> -> <identificador>
<nombre de "typedef"> -> <identificador>
<nombre de tipo> -> <lista de calificadores> <declarador abstracto>?
<declarador abstracto> ->
     <puntero> |
     <puntero>? <declarador abstracto directo>
<declarador abstracto directo> ->
     ( <declarador abstracto> ) |
     <declarador abstracto directo>? [ <expresión constante>? ] |
     <declarador abstracto directo>? ( <lista tipos parámetros>? )
```

1.2.3. Sentencias

```
<sentencia> ->
     <sentencia expresión> |
     <sentencia compuesta> |
     <sentencia de selección> |
     <sentencia de iteración> |
     <sentencia etiquetada> |
     <sentencia de salto>
<sentencia expresión> ->
     <expresión>? ;
```

```
<sentencia compuesta> ->
     {<lista de declaraciones>? <lista de sentencias>?}
<lista de declaraciones> ->
     <declaración> |
     <lista de declaraciones> <declaración>
<lista de sentencias> ->
     <sentencia> |
     <lista de sentencias> <sentencia>
```

- La sentencia compuesta también se denomina *bloque*.

```
<sentencia de selección> ->
     if (<expresión>) <sentencia> |
     if (<expresión>) <sentencia> else <sentencia> |
     switch (<expresión>) <sentencia>
La expresión e controla un switch debe ser de tipo entero.
<sentencia de iteración> ->
     while (<expresión>) <sentencia> |
     do <sentencia> while (<expresión>) ;  |
     for (<expresión>? ; <expresión>? ; <expresión>?) <sentencia>
<sentencia etiquetada> ->
     case <expresión constante> : <sentencia> |
     default : <sentencia> |
     <identificador> : <sentencia>
Las sentencias case y default se utilizan solo dentro de una sentencia switch.
<sentencia de salto> ->
     continue ; |
     break ; |
     return <expresión>? ; |
     goto <identificador> ;
```

- La sentencia **continue** solo debe aparecer dentro del cuerpo de un ciclo. La sentencia **break** solo debe aparecer dentro de un **switch** o en el cuerpo de un ciclo. La sentencia **return** con una expresión no puede aparecer en una función **void**.

1.2.4. Definiciones Externas

```
<unidad de traducción> ->
     <declaración externa> |
     <unidad de traducción> <declaración externa>
<declaración externa> ->
     <definición de función> |
     <declaración>
```

- La unidad de texto de programa luego del preprocesamiento es una *unidad de traducción*, la cual consiste en una secuencia de declaraciones externas.

- Las *declaraciones externas* son llamadas así porque aparece fuera de cualquier función. Los términos *alcance de archivo* y *alcance externo* son sinónimos.

- Si la declaración de un identificador para un *objeto* tiene *alcance de archivo* y un *inicializador*, la declaración es una definición externa para el identificador.

```
<definición de función> ->
     <especificadores de declaración>? <decla> <lista de declaraciones>? <sentencia
     compuesta>
```

1.3. Gramática del Preprocesador

```
<archivo de preprocesamiento> ->
     <grupo>?
<grupo> ->
     <parte de grupo> |
     <grupo parte de grupo>
<parte de grupo> ->
     <sección if> | <línea de control> | <línea de texto> |
     # <no directiva>
<sección if> ->
     <grupo if> <grupos elif>? <grupo else>? <línea endif>
<grupo if> ->
     # if <expresión constante> <nueva línea> <grupo>? |
     # ifdef <identificador> <nueva línea> <grupo>? |
     # ifndef <identificador> <nueva línea>< grupo>?
<grupos elif> ->
     <grupo elif> |
     <grupos elif> <grupo elif>
<grupo elif> ->
     # elif <expresión constante> <nueva línea> <grupo>?
<grupo else> ->
     # else <nueva línea> grupo>?
<línea endif> ->
     # endif <nueva línea>
<línea de control> ->
     # include <tokens pp> <nueva línea> |
     # define <identificador> <lista de reemplazos> <nueva línea> |
     # define <identificador> <parizq> <lista de identificadores>? ) <lista de
     reemplazos> <nueva línea> |
     # define <identificador> <parizq> ... ) <lista de reemplazos> <nueva línea> |
     # define <identificador> <parizq> <lista de identificadores> , ... ) <lista de
     reemplazos> <nueva línea> |
     # undef <identificador> <nueva línea> |
     # line <tokens pp> <nueva línea> |
     # error <tokens pp>? <nueva línea> |
     # pragma <tokens pp> <nueva línea> |
     # <nueva línea>
<línea de texto> ->
     <tokens pp>? <nueva línea>
<no directiva> ->
     <tokens pp> <nueva línea>
<parizq> ->
     un carácter ( no inmediatamente precedido por un espacio blanco
<lista de reemplazos> ->
     <tokens pp>?
<tokens pp> ->
     <token de preprocesamiento> |
     <tokens pp> <token de preprocesamiento>
<nueva línea> -> el carácter nueva línea
Las expresiones constantes de if y elif  pueden estar formadas por los operadores
     comunes y/o por los siguientes operadores de preprocesamiento:

     defined ( <identificador> )
     defined <identificador>
     #
     ##
```

2. Biblioteca

2.1. Definiciones Comunes <stddef.h>

Define, entre otros elementos, el tipo `size_t` y la macro `NULL`. Ambas son definidas, también en otros encabezados, como en `<stdio.h>`.

`size_t`
Tipo entero sin signo que retorna el operador `sizeof`. Generalmente `unsigned int`.

`NULL`
Macro que se expande a un puntero nulo, definido por la implementación. Generalmente el entero cero `0` ó `0L`, ó la expresión constante `(void*)0` .

2.2. Manejo de Caracteres <ctype.h>

`int isalnum (int);`
Determina si el carácter dado `isalpha` o `isdigit` Retorna (ok ? ≠0 : 0).

`int isalpha (int);`
Determina si el carácter dado es una letra (entre las 26 minúsculas y mayúsculas del alfabeto inglés). Retorna (ok ? ≠0 : 0).

`int isdigit (int);`
Determina si el carácter dado es un dígito decimal (entre '0' y '9'). Retorna (ok ? ≠0 : 0).

`int islower (int);`
Determina si el carácter dado es una letra minúscula (entre las 26 del alfabeto inglés). Retorna (ok ? ≠0 : 0)

`int isprint (int);`
Determina si el carácter dado es imprimible, incluye al espacio. Retorna (ok ? ≠0 : 0)

`int isspace (int);`
Determina si el carácter dado es alguno de estos: espacio (`' '`), `'\n'`, `'\t'`, `'\r'`, `'\f'`, `'\v'`. Retorna (ok ? ≠0 : 0)

`int isupper (int);`
Determina si el carácter dado es una letra mayúscula (entre las 26 del alfabeto inglés). Retorna (ok ? ≠0 : 0)

`int isxdigit (int);`
Determina si el carácter dado es un dígito hexadecimal (`'0'`..`'9'`, `'a'`..`'f'` o `'A'`..`'F'`). Retorna (ok ? ≠0 : 0)

`int tolower (int c);`
Si `c` es una letra mayúscula (entre las 26 del alfabeto inglés), la convierte a minúscula. Retorna (mayúscula ? minúscula : `c`)

`int toupper (int c);`
Si `c` es una letra minúscula (entre las 26 del alfabeto inglés), la convierte a mayúscula. Retorna (minúscula ? mayúscula : `c`)

2.3. Manejo de Cadenas <string.h>

Define el tipo `size_t` y la macro `NULL`, ver *Definiciones Comunes*.

`unsigned strlen (const char*);`
Cuenta los caracteres que forman la cadena dada hasta el 1er carácter `'\0'`, excluido. Retorna (longitud de la cadena).

2.3.1. Concatenación

`char* strcat (char* s, const char* t);`
Concatena la cadena `t` a la cadena `s` sobre `s`. Retorna (`s`).

`char* strncat (char* s, const char* t, size_t n);`
Concatena hasta `n` caracteres de `t`, previos al carácter nulo, a la cadena `s`; agrega siempre un `'\0'`. Retorna (`s`).

2.3.2. Copia

`char* strncpy (char* s, const char* t, size_t n);`
Copia hasta `n` caracteres de `t` en `s`; si la longitud de la cadena `t` es < `n`, agrega caracteres nulos en `s` hasta completar `n` caracteres en total; atención: no agrega automáticamente el carácter nulo. Retorna (`s`).

`char* strcpy (char* s, const char* t);`
Copia la cadena `t` en `s` (es la asignación entre cadenas). Retorna (`s`).

2.3.3. Búsqueda y Comparación

`char* strchr (const char* s, int c);`
Ubica la 1ra. aparición de `c` (convertido a `char`) en la cadena `s`; el `'\0'` es considerado como parte de la cadena. Retorna (ok ? puntero al carácter localizado : `NULL`)

`char* strstr (const char* s, const char* t);`
Ubica la 1ra. ocurrencia de la cadena t (excluyendo al '\0') en la cadena s. Retorna (ok ? puntero a la subcadena localizada : `NULL`).

`int strcmp (const char*, const char*);`
Compara "lexicográficamente" ambas cadenas. Retorna (0 si las cadenas son iguales; < 0 si la 1ra. es "menor" que la 2da.; > 0 si la 1ra. es "mayor" que la 2da.)

`int strncmp (const char* s, const char* t, size_t n);`
Compara hasta `n` caracteres de `s` y de `t`. Retorna (como `strcmp`).

`char* strtok (char*, const char*);`
Separa en "tokens" a la cadena dada como 1er. argumento; altera la cadena original; el 1er. argumento es la cadena que contiene a los "tokens"; el 2do. argumento es una cadena con caracteres separadores de "tokens". Retorna (ok ? puntero al 1er. carácter del "token" detectado : `NULL`).

2.3.4. Manejo de Memoria

`void* memchr(const void* s, int c, size_t n);`

Localiza la primer ocurrencia de **c** (convertido a un **unsigned char**) en los **n** iniciales caracteres (cada uno interpretado como **unsigned char**) del objeto apuntado por **s**. Retorna (ok ? puntero al carácter localizado : **NULL**).

```
int memcmp (const void* p, const void* q, unsigned n);
```
Compara los primeros **n** bytes del objeto apuntado por **p** con los del objeto apuntado por **q**. Retorna (0 si son iguales; < 0 si el 1ero. es "menor" que el 2do.; > 0 si el 1ero. es "mayor" que el 2do.)

```
void* memcpy (void* p, const void* q, unsigned n);
```
Copia **n** bytes del objeto apuntado por **q** en el objeto apuntado por **p**; si la copia tiene lugar entre objetos que se superponen, el resultado es indefinido. Retorna (**p**).

```
void* memmove (void* p, const void* q, unsigned n);
```
Igual que **memcpy**, pero actúa correctamente si los objetos se superponen. Retorna (**p**).

```
void* memset (void* p, int c, unsigned n);
```
Inicializa los primeros **n** bytes del objeto apuntado por **p** con el valor de **c** (convertido a **unsigned char**). Retorna (**p**).

2.4. Utilidades Generales <stdlib.h>

2.4.1. Tips y Macros

size_t
NULL
Ver *Definiciones Comunes*.

EXIT_FAILURE
EXIT_SUCCESS
Macros que se expanden a expresiones constantes enteras que pueden ser utilizadas como argumentos de **exit** ó valores de retorno de **main** para retornar al entorno de ejecución un estado de terminación no exitosa o exitosa, respectivamente.

RAND_MAX
Macro que se expande a una expresión constante entera que es el máximo valor retornado por la función **rand**, como mínimo su valor debe ser 32767.

2.4.2. Conversión

```
double atof (const char*);
```
Convierte una cadena que representa un real **double** a número **double**. Retorna (número obtenido, no necesariamente correcto).

```
int atoi (const char*);
```
Convierte una cadena que representa un entero **int** a número **int**. Retorna (número obtenido, no necesariamente correcto) .

```
long atol (const char*);
```
Convierte una cadena que representa un entero **long** a número **long**. Retorna (número obtenido, no necesariamente correcto).

```
double strtod (const char* p, char** end);
```

Convierte como **atof** y, si el 2do. argumento no es **NULL**, un puntero al primer carácter no convertible es colocado en el objeto apuntado por **end**. Retorna como **atof**.

`long strtol (const char* p, char** end, int base);`
Similar a **atol** pero para cualquier base entre 2 y 36; si la base es 0, admite la representación decimal, hexadecimal u octal; ver **strtod** por el parámetro **end**. Retorna como **atol**.

`unsigned long strtoul (const char* p, char** end, int base);`
Igual que **strtol** pero convierte a **unsigned long**. Retorna como **atol**, pero **unsigned long**.

2.4.3. Administración de Memoria

`void* malloc (size_t t);`
Reserva espacio en memoria para almacenar un objeto de tamaño **t**. Retorna (ok ? puntero al espacio reservado : **NULL**)

`void* calloc (size_t n, size_t t);`
Reserva espacio en memoria para almacenar un objeto de **n** elementos, cada uno de tamaño **t**. El espacio es inicializado con todos sus bits en cero. Retorna (ok ? puntero al espacio reservado : **NULL**)

`void free (void* p);`
Libera el espacio de memoria apuntado por **p**. No retorna valor.

`void* realloc (void* p, size_t t);`
Reubica el objeto apuntado por **p** en un nuevo espacio de memoria de tamaño **t** bytes. Retorna (ok ? puntero al posible nuevo espacio : **NULL**).

2.4.4. Números Pseudo-Aleatorios

`int rand (void);`
Determina un entero pseudo-aleatorio entre 0 y **RAND_MAX**. Retorna (entero pseudo-aleatorio).

`void srand (unsigned x);`
Inicia una secuencia de números pseudo-aleatorios, utilizando a **x** como semilla. No retorna valor.

2.4.5. Comunicación con el Entorno

`void exit (int estado);`
Produce una terminación normal del programa. Todos los flujos con *buffers* con datos no escritos son escritos, y todos los flujos asociados a archivos son cerrados. Si el valor de **estado** es **EXIT_SUCCESS** se informa al ambiente de ejecución que el programa terminó exitosamente, si es **EXIT_FAILURE** se informa lo contrario. Equivalente a la sentencia **return estado**; desde la llamada inicial de **main**. Esta función *no retorna a su función llamante*.

`void abort (void);`
Produce una terminación anormal del programa. Se informa al ambiente de ejecución que se produjo una terminación no exitosa. Esta función *no retorna a su función llamante*.

`int system (const char* lineadecomando);`
Si **lineadecomando** es **NULL**, informa si el sistema posee un procesador de comandos. Si **lineadecomando** no es **NULL**, se lo pasa al procesador de comandos para que lo ejecute. Retorna (**lineacomando** ? valor definido por la implementación, generalmente el nivel de error del programa ejecutado : (sistema posee procesador de comandos ? ≠0 : 0)).

2.4.6. Búsqueda y Ordenamiento

```
void* bsearch (
     const void* k,
     const void* b,
     unsigned n,
     unsigned t,
     int (*fc) (const void*, const void*)
);
```

Realiza una búsqueda binaria del objeto `*k` en un arreglo apuntado por `b`, de `n` elementos, cada uno de tamaño `t` bytes, ordenado ascendentemente. La función de comparación `fc` debe retornar un entero < 0, 0 o > 0 según la ubicación de `*k` con respecto al elemento del arreglo con el cual se compara. Retorna (encontrado ? puntero al objeto : `NULL`).

```
void qsort (
     const void* b,
     unsigned n,
     unsigned t,
     int (*fc) (const void*, const void*)
);
```

Ordena ascendentemente un arreglo apuntado por `b`, de `n` elementos de tamaño `t` cada uno; la función de comparación `fc` debe retornar un entero < 0, 0 o > 0 según su 1er. argumento sea, respectivamente, menor, igual o mayor que el 2do. No retorna valor.

2.5. Entrada / Salida <stdio.h>

2.5.1. Tipos

`size_t`
Ver *Definiciones Comunes*.

`FILE`
Registra toda la información necesitada para controlar un *flujo*, incluyendo su *indicador de posición en el archivo*, puntero asociado a un *buffer* (si se utiliza), un *indicador de error* que registra sin un error de lectura/escritura ha ocurrido, y un *indicador de fin de archivo* que registra si el fin del archivo ha sido alcanzado.

`fpos_t`
Posibilita registrar la información que especifica unívocamente cada posición dentro de un archivo.

2.5.2. Macros

`NULL`
Ver *Definiciones Comunes*.

`EOF`
Expresión constante entera con tipo `int` y valor negativo que es retornada por varias funciones para indicar *fin de archivo*; es decir, no hay mas datos entrantes que puedan ser leídos desde un *flujo*, esta situación puede ser porque se llegó al fin del archivo o porque ocurrió algún error. Contrastar con `feof` y `ferror`.

`SEEK_CUR`
`SEEK_END`

`SEEK_SET`
Argumentos para la función `fseek`.

`stderr`
`stdin`
`stdout`
Expresiones del tipo `FILE*` que apuntan a objetos asociados con los flujos estándar de error, entrada y salida respectivamente.

2.5.3. Operaciones sobre Archivos

```
int remove(const char* nombrearchivo);
```
Elimina al archivo cuyo nombre es el apuntado por `nombrearchivo`. Retorna (ok ? 0 : ≠0)

```
int rename(const char* viejo, const char* nuevo);
```
Renombra al archivo cuyo nombre es la cadena apuntada por `viejo` con el nombre dado por la cadena apuntada por `nuevo`. Retorna (ok ? 0 : ≠0).

2.5.4. Acceso

```
FILE* fopen (
     const char* nombrearchivo,
     const char* modo
);
```
Abre el archivo cuyo nombre es la cadena apuntada por `nombrearchivo` asociando un flujo con este según el `modo` de apertura. Retorna (ok ? puntero al objeto que controla el flujo : `NULL`).

```
FILE* freopen(
     const char* nombrearchivo,
     const char* modo,
     FILE* flujo
);
```
Abre el archivo cuyo nombre es la cadena apuntada por `nombrearchivo` y lo asocia con el flujo apuntado por `flujo`. La cadena apuntada por `modo` cumple la misma función que en `fopen`. Uso más común es para el redireccionamiento de `stderr`, `stdin` y `stdout` ya que estos son del tipo `FILE*` pero no necesariamente *lvalues* utilizables junto con `fopen`. Retorna (ok ? flujo : `NULL`).

```
int fflush (FILE* flujo);
```
Escribe todos los datos que aún se encuentran en el buffer del flujo apuntado por `flujo`. Su uso es imprescindible si se mezcla `scanf` con `gets` o `scanf` con `getchar`, si se usan varios `fgets`, etc. Retorna (ok ? 0 : `EOF`).

```
int fclose (FILE* flujo);
```
Vacía el *buffer* del flujo apuntado por `flujo` y cierra el archivo asociado. Retorna (ok ? 0 : `EOF`)

2.5.5. Entrada / Salida Formateada

Flujos en General

```
int fprintf (FILE* f, const char* s, ...);
```
Escritura formateada en un archivo ASCII. Retorna (ok ? cantidad de caracteres escritos : < 0).

```
int fscanf (FILE* f, const char*, ...);
```

Lectura formateada desde un archivo ASCII. Retorna (cantidad de campos almacenados) o retorna (**EOF** si detecta fin de archivo).

Flujos stdin y stdout

`int scanf (const char*, ...);`
Lectura formateada desde **stdin**. Retorna (ok ? cantidad de ítems almacenados : **EOF**).

`int printf (const char*, ...);`
Escritura formateada sobre **stdout**. Retorna (ok ? cantidad de caracteres transmitidos : < 0).

Cadenas

`int sprintf (char* s, const char*, ...);`
Escritura formateada en memoria, construyendo la cadena **s**. Retorna (cantidad de caracteres escritos).

`int sscanf (const char* s, const char*, ...);`
Lectura formateada desde una cadena **s**. Retorna (ok ? cantidad de datos almacenados : **EOF**).

2.5.6. Entrada / Salida de a Caracteres

`int fgetc (FILE*); ó`
`int getc (FILE*);`
Lee un carácter (de un archivo ASCII) o un byte (de un archivo binario). Retorna (ok ? carácter/byte leído : **EOF**).

`int getchar (void);`
Lectura por carácter desde **stdin**. Retorna (ok ? próximo carácter del buffer : **EOF**).

`int fputc (int c, FILE* f); ó`
`int putc (int c, FILE* f);`
Escribe un carácter (en un archivo ASCII) o un byte (en un archivo binario). Retorna (ok ? **c** : **EOF**).

`int putchar (int);`
Eescritura por carácter sobre stdout. Retorna (ok ? carácter transmitido : **EOF**).

`int ungetc (int c, FILE* f);`
"Devuelve" el carácter o byte **c** para una próxima lectura. Retorna (ok ? **c** : **EOF**).

2.5.7. Entrada / Salida de a Cadenas

`char* fgets (char* s, int n, FILE* f);`
Lee, desde el flujo apuntado **f**, una secuencia de a lo sumo **n**-1 caracteres y la almacena en el objeto apuntado por **s**. No se leen más caracteres luego del carácter nueva línea o del fin del archivo. Un carácter nulo es escrito inmediatamente después del último carácter almacenado; de esta forma, s queda apuntando a una cadena. Importante su uso con **stdin**. Si leyó correctamente, **s** apunta a los caracteres leídos y retorna **s**. Sí leyó sólo el fin del archivo, el objeto apuntado por **s** no es modificado y retorna **NULL**. Si hubo un error, contenido del objeto es indeterminado y retorna **NULL**. Retorna (ok ? **s** : **NULL**).

`char* gets (char* s);`
Lectura por cadena desde **stdin**; es mejor usar **fgets**() con **stdin** . Retorna (ok ? **s** : **NULL**).

`int fputs (const char* s, FILE* f);`

Escribe la cadena apuntada por **s** en el flujo **f**. Retorna (ok ? último carácter escrito : **EOF**).

```
int puts (const char* s);
```
Escribe la cadena apuntada por **s** en **stdout**. Retorna (ok ? ≥ 0 : **EOF**).

2.5.8. Entrada / Salida de a Bloques

```
unsigned fread (void* p, unsigned t, unsigned n, FILE* f);
```
Lee hasta **n** bloques contiguos de **t** bytes cada uno desde el flujo **f** y los almacena en el objeto apuntado por **p**. Retorna (ok ? **n** : < **n**).

```
unsigned fwrite (void* p, unsigned t, unsigned n, FILE* f);
```
Escribe n bloques de t bytes cada uno, siendo el primero el apuntado por p y los siguientes, sus contiguos, en el flujo apuntado por f. Retorna (ok ? **n** : < **n**).

2.5.9. Posicionamiento

```
int fseek (
     FILE* flujo,
     long desplazamiento,
     int desde
);
```
Ubica el *indicador de posición de archivo* del flujo binario apuntado por **flujo**, **desplazamiento** caracteres a partir de **desde**. **desde** puede ser **SEEK_SET**, **SEEK_CUR** ó **SEEK_END**, comienzo, posición actual y final del archivo respectivamente. Para flujos de texto, **desplazamiento** deber ser cero o un valor retornado por **ftell** y **desde** debe ser **SEEK_SET**. En caso de éxito los efectos de **ungetc** son deshechos, el *indicador de fin de archivo* es desactivado y la próxima operación puede ser de lectura o escritura. Retorna (ok ? 0 : ≠ 0).

```
int fsetpos (FILE* flujo, const fpos_t* posicion);
```
Ubica el *indicador de posición de archivo* (y otros estados) del flujo apuntado por **flujo** según el valor del objeto apuntado por **posicion**, el cual debe ser un valor obtenido por una llamada exitosa a **fgetpos**. En caso de éxito los efectos de **ungetc** son deshechos, el *indicador de fin de archivo* es desactivado y la próxima operación puede ser de lectura o escritura. Retorna (ok ? 0 : ≠ 0).

```
int fgetpos (FILE* flujo, fpos_t* posicion);
```
Almacena el *indicador de posición de archivo* (y otros estados) del flujo apuntado por **flujo** en el objeto apuntado por **posicion**, cuyo valor tiene significado sólo para la función **fsetpos** para el restablecimiento del *indicador de posición de archivo* al momento de la llamada a **fgetpos**. Retorna (ok ? 0 : ≠ 0).

```
long ftell (FILE* flujo);
```
Obtiene el valor actual del *indicador de posición de archivo* para el flujo apuntado por **flujo**. Para flujos binarios es el número de caracteres (bytes ó posición) desde el comienzo del archivo. Para flujos de texto la valor retornado es sólo útil como argumento de **fseek** para reubicar el indicador al momento del llamado a **ftell**. Retorna (ok ? indicador de posición de archivo : –1L).

```
void rewind(FILE *stream);
```
Establece el indicador de posición de archivo del flujo apuntado por **flujo** al principio del archivo. Semánticamente equivalente a **(void)fseek(stream, 0L, SEEK_SET)**, salvo que el indicador de error del flujo es desactivado. No retorna valor.

2.5.10. Manejo de Errores

`int feof (FILE* flujo);`
Chequea el *indicador de fin de archivo* del flujo apuntado por **flujo**. Contrastar con la macro **EOF** (Contrastar con la macro **EOF** retornada por algunas funciones). Retorna (*indicador de fin de archivo* activado ? ≠ 0 : 0).

`int ferror (FILE* flujo);`
Chequea el *indicador de error* del flujo apuntado por **flujo** (Contrastar con la macro **EOF** retornada por algunas funciones). Retorna (*indicador de error* activado ? ≠ 0 : 0).

`void clearerr(FILE* flujo);`
Desactiva los indicadores de fin de archivo y error del flujo apuntado por flujo. No retorna valor.

`void perror(const char* s);`
Escribe en el flujo estándar de error (**stderr**) la cadena apuntada por **s**, seguida de dos puntos (**:**), un espacio, un mensaje de error apropiado y por último un carácter nueva línea (**\n**). El mensaje de error está en función a la expresión **errno**. No retorna valor.

2.6. Otros

2.6.1. Hora y Fecha <time.h>

NULL
`size_t`
Ver *Definiciones Comunes*.

`time_t`
`clock_t`
Tipos aritméticos capaces de representar el tiempo. Generalmente **long**.

CLOCKS_PER_SEC
Macro que expande a una expresión constante de tipo **clock_t** que es el número por segundos del valor retornado por la función **clock**.

`clock_t clock(void);`
Determina el tiempo de procesador utilizado desde un punto relacionado con la invocación del programa. Para conocer el valor en segundos, dividir por **CLOCKS_PER_SEC**. Retorna (ok ? el tiempo transcurrido: **(clock_t)(-1)**.

`char* ctime (time_t* t);`
Convierte el tiempo de ***t** a fecha y hora en una cadena con formato fijo. Ejemplo: Mon Sep 17 04:31:52 1973\n\0. Retorna (cadena con fecha y hora).

`time_t time (time_t* t);`
Determina el tiempo transcurrido en segundos desde la hora 0 de una fecha base; por ejemplo: desde el 1/1/70. Retorna (tiempo transcurrido). Si **t** no es **NULL**, también es asignado a ***t**.

2.6.2. Matemática

`int abs(int i);`
`long int labs(long int i);`
<stdlib.h> Calcula el valore del entero **i**. Retorna (valor absoluto de **i**).

`double ceil (double x);`

<math.h> Calcula el entero más próximo, no menor que **x**. Retorna (entero calculado, expresado como **double**).

double floor (double x);
<math.h> Calcula el entero más próximo, no mayor que **x**. Retorna (entero calculado, expresado como **double**).

double pow (double x, double z);
<math.h> Calcula x^z; hay error de dominio si $x < 0$ y **z** no es un valor entero, o si **x** es 0 y $z \neq 0$. Retorna (ok ? x^z : error de dominio o de rango).

double sqrt (double x);
<math.h> Calcula la raíz cuadrada no negativa de **x**. Retorna ($x \geq 0.0$? raíz cuadrada : error de dominio).

2.7. Los Formatos

2.7.1. Funciones printf, sprintf, fprintf

- Antes de interpretarse la cadena de formatos, todo argumento **float** es convertido a **double**, y todo argumento **char** o **short** es convertido a **int**.

- Si la cantidad de argumentos es menor que la cantidad de formatos, el comportamiento es indefinido. Si la cantidad de argumentos es mayor que la cantidad de formatos, los argumentos sobrantes son evaluados como siempre pero son ignorados. El retorno de la función se produce cuando llega al final de la cadena de formatos.

Cada *especificación de conversión* se realiza mediante la siguiente codificación:

```
% [banderas] [ancho] [precisión] especificador
```

Banderas

Opcional, modifican el significado de la especificación de conversión.

– (ejemplo: %–30s) justifica la conversión a izquierda y rellena con espacios a derecha (si es necesario)

0 (ejemplo: %04x) rellena con ceros a izquierda (después del signo o de un prefijo)

+ (ejemplo: %+5d) si el número convertido es positivo, genera un signo **+** como primer carácter.

<espacio>
(ejemplo: % 5d) un espacio genera un espacio si el número convertido es positivo

(ejemplo: %#x) altera el comportamiento de ciertas conversiones: El especificador **x** produce **0x** como prefijo y el especificador **X** produce el prefijo **0X** Las conversiones para números reales generan un **.** (punto decimal) aún si el número es entero.

Ancho

Opcional, especifica la cantidad mínima de caracteres a ser generados por la conversión.

Es un entero decimal sin signo (ejemplo: %10c).

- Si se escribe un * (ejemplo: %*d), entonces toma al próximo argumento int como valor del ancho; si este valor es negativo, contribuye con una bandera –.
- Si la conversión produce menos caracteres que la cantidad indicada por ancho, habrá relleno; en ausencia de las banderas – o 0, rellena con blancos a izquierda.

Precisión

Opcional, controla la cantidad de caracteres generados por ciertas conversiones.

Se escribe como . (punto) seguido de un entero decimal sin signo.

Un . sólo especifica una precisión cero.

Si se escribe `.*`, el valor del próximo argumento `int` es tomado como precisión; si este valor es negativo, se considera que la precisión es cero.

La precisión (ejemplos: %.10e, %.*s) especifica:

- si se convierte un entero => la cantidad mínima de dígitos a generar
- para los especificadores `e`, `E` o `f` => la cantidad de dígitos a derecha del punto decimal
- para los especificadores `g` o `G` => la cantidad máxima de dígitos significativos a generar
- para el especificador `s` => la cantidad máxima de caracteres a generar

Especificador

Obligatorio, determina cómo interpreta y convierte al correspondiente argumento.

Sea p el valor del campo precisión; entonces:

`c` convierte el argumento `int` a `unsigned char` para generar un carácter

`d` convierte el argumento `int` a una secuencia de al menos `p` dígitos decimales; por omisión, el valor de `p` es 1.

`hd` convierte el argumento `int` a `short` y luego actúa como `d`.

`ld` convierte el argumento `long` igual que `d`.

`i`, `hi`, `li` igual que `d`, `hd`, `ld` respectivamente.

`u` convierte el argumento `unsigned int` y genera una secuencia, sin signo, de un mínimo de `p` dígitos decimales; por omisión, el valor de `p` es 1.

`hu` convierte el argumento `int` a `unsigned short` y luego actúa igual que `u`.

`lu` convierte el argumento `long` a `unsigned long` y luego actúa igual que `u`.

`x`,`X` convierte el argumento `int` a `unsigned int` y luego genera una secuencia, sin signo, de un mínimo de `p` dígitos hexadecimales. Los dígitos hexadecimales con valores `10` a `15` se representan con las letras `a` a `f` o las letras `A` a `F`, respectivamente. Por omisión, el valor de `p` es 1.

hx,hX convierte el argumento **int** a **unsigned short** y luego actúa igual que **x** o **X**, según corresponda.

lx,lX convierte el argumento **long** a **unsigned long** y luego actúa igual que **x** o **X**, según corresponda.

O convierte el argumento **int** a **unsigned int** y luego genera una secuencia sin signo de al menos **p** dígitos octales. Por omisión, el valor de **p** es 1.

ho convierte el argumento **int** a **unsigned short** y luego actúa igual que **o**.

lo convierte el argumento **long** igual que **o**.

e,E convierte el argumento double a una secuencia de la forma **d**.**ddde±dd** o **d**.**dddE±dd** (notación punto flotante); ejemplo: **0.123456e+04** o **0.123456E+04**. Cada **d** significa dígito decimal; el exponente está formado por signo y dos dígitos decimales como mínimo; por omisión, la precisión es 6. Si **p** es **0** y no figura la bandera **#**, el punto decimal (.) es omitido.

Le, LE convierte el argumento **long double** igual que **e** o **E**, respectivamente.

F convierte el argumento **double** a una secuencia de la forma **d**.**dddddd** (notación punto fijo). Por omisión, la precisión es **6**. Si **p** es **0** y no figura la bandera **#**, el punto decimal (.) es omitido.

Lf convierte el argumento **long double** igual que **f**.

P convierte el argumento de tipo **void*** a una secuencia de caracteres definida por la implementación, como, por ejemplo, la representación hexadecimal de una dirección de memoria.

S genera los caracteres de la cadena apuntada por el argumento, que debe ser de tipo **char***. Si se especifica una precisión, entonces genera **p** caracteres como máximo.

% no hay conversión; genera el carácter **%**.

2.7.2. Funciones scanf, sscanf, fscanf

Cada *especificación de conversión* se realiza mediante la siguiente codificación:

```
% [*] [ancho] especificador
```

*** (Asterisco)** Opcional, indica "supresión de asignación" del campo "scaneado". Ejemplo: %*s indica "saltear" una secuencia de caracteres que no contiene blancos.

Ancho Opcional, especifica la cantidad máxima de caracteres a ser convertidos para su asignación.

Especificador Obligatorio, determina cómo interpreta y convierte al dato que deberá ser almacenado.

C almacena un carácter. No "saltea" espacios en blanco.

D convierte el entero (base 10) ingresado y lo almacena en una variable **int**.

Hd como **d** pero almacena en una variable **short**.

Ld como **d** pero almacena en una variable **long**.

u,**hu**,**lu**
como **d**, **hd**, **ld** respectivamente pero el entero ingresado es sin signo y es almacenado en una variable **unsigned**.

o,**ho**,**lo**
como **d**, **hd**, **ld** respectivamente pero el entero ingresado es interpretado en base 8 y lo almacena en una variable **unsigned**.

x(**X**),**hx**(**hX**),**lx**(**lX**)
: como **d, hd, ld** respectivamente pero el entero ingresado es interpretado en base 16 y almacenado en una variable **unsigned**.

i,**hi**,**li**
: igual que **d**, **hd**, **ld** respectivamente, pero interpretan también enteros *octales* (que comienzan con **0**) y enteros *hexadecimales* (que comienzan con **0x** o **0X**).

e,**E**,**f**,**g**,**G**
: interpreta el dato como "número real" y lo almacena en una variable **float**.

le,**lE**,**lf**,**lg**,**lG**
: como el especificador **e** pero lo almacena en una variable **double**.

Le,**LE**,**Lf**,**Lg**,**LG**
: como el especificador **e** pero lo almacena en una variable **long double**.

s
: almacena una secuencia de caracteres (sin blancos) en una zona de memoria que representa un arreglo de **char**. Siempre agrega el carácter **'\0'** como centinela.