

C

C: [Learning C with Pebble](#) | [Beej's Guide to C Programming](#) | [C++ reference](#)
man: [Linux man pages online](#) | [Linux man pages](#)
Bash: [Bash Reference Manual](#) | [Introduction to Bash](#) | [BashGuide](#)

Norminette: `norminette -R CheckForbiddenSourceHeader`

Compilar: `cc -Wall -Wextra -Werror`

Tipos

`int` Integer (4 bytes)
`char` Single character (1 byte)
`char*` String
`float` Floating point (4 bytes)

int

Una variable de tipo `int` puede tener un valor entre -2.147.483.648 y 2.147.483.647.
(Léase: *Dos mil ciento cuarenta y siete millones, etc.*).

```
int num = 12;

// equivale a:

int num;
num = 12;
```

char

Una variable de tipo `char` puede tener un valor entre -128 y 128.
Además, también puede tener el valor de un carácter.

```
char c = 97;

// es lo mismo que:

char c = 'a';
```

El símbolo de salto de línea se considera que es un carácter:

```
char new_line = '\n';
```

Tipos

En C la longitud mínima que usaremos es de 1 byte, esto es, 8 bits. Por ejemplo: `00101001`.

En una arquitectura de 64 bits la longitud máxima es de 8 bytes.

De esta manera tenemos tipos que ocuparán entre 1 y 8 bytes.

Para averiguar el tamaño en bytes podemos usar la función `sizeof()`.

<code>void</code>	tipo <code>void</code>	<code>sizeof(void)</code> = 1 byte (8 bits)
<code>char</code>	tipo <code>char</code> (carácter)	<code>sizeof(char)</code> = 1 byte (8 bits)
<code>short</code>	tipo <code>short</code>	<code>sizeof(short)</code> = 2 bytes (16 bits)
<code>int</code>	tipo <code>int</code> (entero)	<code>sizeof(int)</code> = 4 bytes (32 bits)
<code>float</code>	tipo <code>float</code>	<code>sizeof(float)</code> = 4 bytes (32 bits)
<code>long</code>	tipo <code>long</code> (entero)	<code>sizeof(long)</code> = 8 bytes
<code>long long</code>	tipo <code>long</code> (entero)	<code>sizeof(long)</code> = 8 bytes

Los punteros son tipos de 8 bytes independientemente del tipo al que apunten.

(Si tu dedo apunta a una hormiga y después a la luna, ¿cuánto mide tu dedo?)

<code>void *</code>	tipo puntero	<code>sizeof(void *)</code> = 8 bytes (64 bits)
<code>char *</code>	tipo puntero	<code>sizeof(char *)</code> = 8 bytes (64 bits)
<code>int *</code>	tipo puntero	<code>sizeof(int *)</code> = 8 bytes (64 bits)

`void *` es como un comodín, cualquier tipo de puntero.

Rangos de valores

En realidad todas las variables guardan números. El rango de números de cada tipo depende de si son `signed` o `unsigned`. Si se omite por defecto es `signed`.

Para calcular el número de valores distintos elevamos 2 al número de bits de ese tipo. La base es 2 porque estamos en un sistema binario, sólo hay dos valores: `0` y `1`.

char

Un dato de tipo `char` puede almacenar hasta 2^8 valores distintos: 256.

Si es `unsigned char` los valores serán entre `0` y `255`. Si es `signed char`, entre `-128` y `127`.

<code>unsigned char</code>	<code>0</code>	<code>1</code>	<code>2</code>	<code>3</code>	<code>...</code>	<code>253</code>	<code>254</code>	<code>255</code>					
<code>signed char</code>	<code>-128</code>	<code>-127</code>	<code>-126</code>	<code>...</code>	<code>-2</code>	<code>-1</code>	<code>0</code>	<code>1</code>	<code>2</code>	<code>...</code>	<code>125</code>	<code>126</code>	<code>127</code>

int

Un dato de tipo `int` puede almacenar hasta 2^{32} valores distintos: 4.294.967.296.

<code>unsigned int</code>	<code>0</code>	<code>→</code>	<code>4.294.967.295</code>
<code>signed int</code>	<code>-2.147.483.648</code>	<code>→</code>	<code>2.147.483.647</code>

Overflow

Si un `signed char` admite valores entre -128 y 127, ¿qué sucede si le pasamos un valor mayor, como 128? 128 *da la vuelta* y se traduce a -128. 129 se convierte en -127. 128 se convierte en -126, etc.

Tabla ASCII

0	NUL	32	(space)	64	@	96	`
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	\$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	'	71	G	103	g
8	BS	40	(72	H	104	h
9	TAB	41)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[123	{
28	FS	60	<	92	\	124	
29	GS	61	=	93]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	DEL

Atajos

<code>i = i + 1;</code>	<code>=</code>	<code>i += 1;</code>
<code>i = i * 10;</code>	<code>=</code>	<code>i *= 10;</code>

<code>while (n > 0)</code>	<code>=</code>	<code>while (n--)</code>
<code>{</code>		<code>{</code>
<code> //...</code>		<code> //...</code>
<code> n--;</code>		<code>}</code>
<code>}</code>		

<code>*a = *b;</code>	<code>=</code>	<code>*a-- = *b--</code>
<code>*a++;</code>		
<code>*b++;</code>		

<code>*a = *b;</code>	<code>=</code>	<code>*a-- = *b--;</code>
<code>*a--;</code>		
<code>*b--;</code>		

<code>arr[i]</code>	<code>=</code>	<code>*(arr + i)</code>
---------------------	----------------	-------------------------

Pointers

Un puntero es una variable que contiene la dirección de memoria de un dato.

No es el dato en sí, pero te dice dónde encontrarlo.

Por ejemplo, un pointer en notación hexadecimal se escribe `0x7ffd99493000`.

Para declarar una variable que sea un puntero hay que especificar el tipo al que se refiere el puntero.

Además, se incluye el modificador `*`, que significa "puntero a...".

Por ejemplo, aquí tenemos una variable `ptr` que apunta a un `int` (un número entero):

```
int *ptr;
```

Es decir, `*ptr` se debe leer como "contenido de `ptr`", donde `ptr` es una dirección de memoria almacenada en `ptr`.

Estas formas también son válidas y equivalentes:

```
int *ptr;
int* ptr;
int * ptr;
int*ptr;
```

Si una variable apunta a un entero, eso significa que se puede usar allí donde puede usarse un entero.

Con lo cual los dos códigos siguientes son equivalentes:

<pre>int main(void) { int a = 0; a = 10; a = a - 3; printf("%d\n", a); }</pre>	<pre>int main(void) { int a = 0; *pint = &a; *pint = 10; *pint = *pint - 3; printf("%d\n", *pint); }</pre>
---	--

Operadores & y *

`&` devuelve la dirección del operando.

`*` interpreta el operando como una dirección y devuelve su contenido.

Por ejemplo, `&a` devuelve la dirección de `a`, `*a` devuelve el contenido de `a`.

Operador &

`&` es el operador `address-of`. Cuando precede a una variable indica la dirección o referencia de esa variable en la memoria (un pointer). Algunas funciones no reciben la variable en sí, sino un pointer (¿write sería un ejemplo?).

Por ejemplo, en esta función estamos pasando la dirección de la variable `num`:

```
int num = 10;           // Esta es la variable num
my_function(&num);      // Y aquí pasamos la dirección de num, no su valor
```

Operador *

1) Cuando se usa en la declaración de una variable, el `*` indica que la variable es un puntero, es decir, que la variable guarda la dirección en la memoria de otra variable.

```
int *ptr_to_int;           // ptr_to_int es un pointer a un int
char *ptr_to_char;        // ptr_to_char es un pointer a un char
```

2) Cuando se usa antes de una variable de puntero, el asterisco `*` accede al valor en esa dirección (dereferencing).

```
int num = 10;
int *ptr = &num;           // ptr stores the address of num
printf("%d\n", *ptr);      // Dereferencing ptr, output: 10
```

3) Cuando se usa en el parámetro de una función, `*` indica que la función recibe un pointer y puede modificar el valor original.

referencing y dereferencing

```
int main(void)
{
    char c;
    char *ptr;

    c = 'J';

    ptr = &c;           // referencing
    *ptr = 'M';         // dereferencing

    write(1, &c, 1);
    return(1);
}
```

Strings

Los strings son del tipo `char*`, es decir, un puntero a una dirección de un valor de tipo `char`.

```
char *str = "Hello"

// equivale a

char *str;
str = "Hello";
```

Un string siempre acaba con el código `'\0'`. De esta manera, nos podemos referir simplemente a la dirección de la primera letra y el string se alargará hasta que se encuentre con un `'\0'`.

str, *str, &str

`str` es de tipo `char*` y se refiere a la **dirección** en la memoria donde comienza el string. (La dirección de `'H'`, un puntero a `'H'`). Por ejemplo, `0x1000`.

`*str` es de tipo `char` y se refiere al **valor** de `str`. (El carácter `'H'`). Si hacemos `str++`, entonces `*str` será `'e'`.

`&str` es de tipo `char **` y se refiere a la dirección donde se guarda el puntero `str`. Por ejemplo, `0x2000`.

Expresión	Tipo	Valor
<code>str</code>	<code>char *</code>	Dirección de <code>"Hello"</code> (en realidad, dirección de <code>'H'</code>)
<code>*str</code>	<code>char</code>	<code>'H'</code>
<code>&str</code>	<code>char **</code>	Dirección del puntero <code>str</code>

Pointer arithmetic

A los punteros se les puede sumar o restar números enteros.

Si a un puntero le sumamos `1` obtenemos el puntero adyacente. Así recorreremos un string:

```
char *str;

str = "Hola";

while(*str != '\0')
{
    str++;
}
```

Primero declaramos una variable `str` de tipo `char*`.

Si fuera un único carácter lo dejaríamos como `char`, pero como es un string ponemos `char*` para indicar que es un puntero al primer carácter.

Para recorrer el bucle comprobamos si el contenido del puntero no es `'\0'`, es decir, `*str != '\0'`. Como queremos comprobar el contenido del puntero añadimos `*`.

Para comprobar el siguiente carácter sumamos `1` al puntero: `str++`.

En este caso no ponemos `*` porque queremos sumar `1` al puntero en sí, no a su contenido.

Recorrer un string al revés

Tenemos un string `str`. Creamos un puntero `ptr` y guardamos el comienzo del string `str`. Recorremos el string entero hasta la posición final. Luego retrocedemos con `str--` para evitar `'\0'`. La condición para recorrer el bucle `while` es que `ptr` sea mayor o igual que `str`.

```
char *str = "Hello, world!";
char *ptr = str;

while (*ptr != '\0') {
    ptr++;
}

ptr--;

while (ptr >= str) {
    // ...
    ptr--;
}
```

Función para omitir espacios

En este bucle `while` vamos cambiando el puntero para evitar los espacios:

```
int ft_atoi_base(char *str, char*base)
{
    int integer;

    while ((*str >= 9 && *str <= 13) || *str == ' ')
        str++;

    // ...
}
```

¿Qué hay que hacer para extraer ese bucle a una función separada?

En ese caso la función recibe `*str` pero tiene que devolver `str`.

```
char *skip_whitespace(char *str)
{
    while ((*str >= 9 && *str <= 13) || *str == ' ')
        str++;
    return (str);
}
```

Y para usarla:

```
int ft_atoi_base(char *str, char*base)
{
    int integer;

    str = skip_whitespace(str);

    // ...
}
```


Strings y arrays

Los strings y arrays son parecidos. Un string debe estar delimitado por un carácter `'\0'`.

```
char *str = "Hola"
int *arr = { 12, -7, 33, 81, -24 }
```

Si inicializamos un string como un puntero o como array estamos usando memorias distintas.
Por ejemplo:

```
char str[] = "Hello!"
```

Este string con notación de array se asigna a la memoria **stack**. No hace falta liberarla, se hace automáticamente. Por es adecuada para hacer cosas temporales, pruebas.

```
char str* = "Hello!"
```

Pero este string se guarda en la memoria **heap**. Habrá que usar `malloc()` y `free()` para asignar espacio y liberarlo después.

Strings como arrays

Los strings se pueden escribir en la misma notación que los arrays.

Dentro de los corchetes se indica la longitud, incluyendo `'\0'` (terminator o null character).

Por ejemplo, un string como `Hello!` tendrá 7 caracteres (6 + 1 null character):

```
char str[7] = "Hello!";
```

También podemos inicializar un array sin tamaño y dejar que el compilador lo calcule:

```
char str[] = "Hello!";
```

El tamaño y la longitud se calculan de manera distinta: `strlen()` no incluye `'\0'`, pero `sizeof()` sí:

```
char str[] = "Hello!";           // {'H', 'e', 'l', 'l', 'o', '!', '\0'}
strlen(str);                     // 6
sizeof(str);                     // 7
```

Cuando recorremos un string en un bucle estas dos formas son equivalentes:

```
str[i]           // equivale a      *(str + i)
```

Diferencias entre notación de puntero y de array

¿Cuáles son las diferencias entre estas dos líneas?

```
char *str = "Hello";
char str[] = "Hello";
```

***str**

`str` es un **puntero** al primer carácter.

Se puede reasignar `str`. Por ejemplo, se puede hacer `str = "World"`. Ya no apunta a `"Hello"`, sino a otra dirección.

Pero no se puede hacer `*str = 'j'` porque está en memoria de sólo lectura.

str[]

`str` es un **array** de caracteres.

Se puede modificar el contenido: `str[0] = 'j'`.

Pero no se puede reasignar `str` porque no es un puntero. No se puede hacer `str = "World"`.

Cómo copiar un string

Si usamos el operador de asignación (`=`) para copiar un string estamos copiando un puntero al primer carácter. Para copiar un string hay que recorrerlo byte por byte.

Antes de hacer una copia hay que reservar espacio para el destino de esa copia. Ese espacio debería ser igual o más grande que el original.

```

#include <stdio.h>
#include <string.h>

int main(void)
{
    char s[] = "Hello, world!";
    char t[100];           // Each char is one byte, so plenty of room

    strcpy(t, s);          // This makes a copy of the string!

    t[0] = 'z';            // We modify t

    // And s remains unaffected because it's a different string
    printf("%s\n", s);     // "Hello, world!"

    // But t has been changed
    printf("%s\n", t);     // "zello, world!"
}

```

Notice with `strcpy()`, the destination pointer is the first argument, and the source pointer is the second. A mnemonic I use to remember this is that it's the order you would have put `t` and `s` if an assignment `=` worked for strings, with the source on the right and the destination on the left.

Arrays

Para crear un array de 10 elementos de tipo `int` :

```
int arr[10];  
  
arr[0] = 13;  
arr[1] = 29;  
arr[2] = 86;  
arr[3] = 24;  
// etc.
```

La longitud de un array `arr` se calcula de esta manera:

```
sizeof(arr) / sizeof(arr[0])
```

Es decir, si un `int` son 4 bytes, entonces un array de 10 `int` tendrá un tamaño de 40 bytes.

Argumentos

Si a un programa pasamos argumentos en la línea de comandos serán recibidos por `main()`.

La estructura de `main()` es la siguiente:

```
int main(int argc, char *argv[]);
```

`argc` (argument count) indica el número de argumentos pasados, incluyendo el nombre del programa.

`*argv[]` (argument vector) es un array a uno o varios strings. `argv[0]` apunta al nombre del programa; `argv[1]`, al primer argumento en la línea de comandos; `argv[2]` al segundo, etc.

Por ejemplo, ejecutamos un programa `a.out`:

```
$ ./a.out arg1 arg2 arg3
```

Y en `main()` recibimos los argumentos:

```
int main(int argc, char *argv[]) {  
    printf("Number of arguments: %d\n", argc);  
  
    for (int i = 0; i < argc; i++) {  
        printf("Argument %d: %s\n", i, argv[i]);  
    }  
  
    return 0;  
}
```

Resultado:

```
Number of arguments: 4  
Argument 0: ./program  
Argument 1: arg1  
Argument 2: arg2  
Argument 3: arg3
```

malloc()

La función `malloc()` sirve para reservar espacios para el contenido de un array.
Se importa desde `stdlib.h`.

Para crear un array de números inicializamos un puntero de tipo `int`: `int *arr`.
Y asignamos valores con la función `malloc()`.

`malloc()` recibe el número de bytes que se van a reservar en la memoria.
Como un entero son 4 bytes, para almacenar 10 enteros sería $4 \times 10 = 40$ bytes.
Para averiguar el tamaño de un tipo podemos usar `sizeof()`, por ejemplo: `sizeof(int)`.
Por ejemplo, un array de 10 enteros:

```
int *arr;  
  
arr = malloc(10 * sizeof(int));
```

`malloc()` devuelve un puntero al primer elemento, que estará vacío: `void *`.
Para rellenar el array lo hacemos con un bucle:

```
for (int i = 0; i < 10, i++)  
    arr[i] = i;
```

free

Siempre que se use `malloc()` al final hay que liberar el espacio que se ha reservado:

```
free(arr);
```

cast

`malloc()` devuelve un puntero vacío: `void *`.
Se puede convertir (*cast*, *parse*) a otro tipo de dato. Por ejemplo, para convertirlo a `int`:

```
arr = (int*)malloc(10 * sizeof(int));
```

Resumen ejercicios Piscina C01-C04

C01

OK: ft_ft, ft_ultimate_ft, ft_swap, ft_div_mod, ft_ultimate_div_mod, ft_putstr, ft_strlen

Sin entregar: ft_rev_int_tab, ft_sort_int_tab

C02

OK: ft_strcpy, ft_strncpy, ft_str_is_alpha, ft_str_is_numeric, ft_str_is_lowercase, ft_str_is_uppercase, ft_str_is_printable, ft_strupcase, ft_strlowercase

KO: ft_strcapitalize

Sin entregar: ft_strlcpy, ft_putstr_non_printable, ft_print_memory

C03

OK: ft_strcmp, ft_strncmp, ft_strcat, ft_strncat, ft_strstr

KO: ft_strlcat

C04

OK: ft_strlen, ft_putstr, ft_putnbr

KO: ft_atoi, ft_putnbr_base, ft_atoi_base

C05: Recursividad

C06: Argumentos main

C07: malloc

C08: ft.h

C09: Makefile

C10: Makefile

C11: Bucles

C12: t_list

C13: t_btree

Libft

Strings

`isalpha`, `isdigit`, `isalnum`, `isascii`, `isprint`
`strlen`, `strcpy`, `strcat`
`toupper`, `tolower`
`strchr`, `strrchr`, `strncmp`, `strnstr`

atoi

```
int atoi(char *nptr);
```

The `atoi()` function converts the initial portion of the string pointed to by `nptr` to `int`.

Returns the converted value or `0` on error.

The behavior is the same as `strtol(nptr, NULL, 10);` except that `atoi()` does not detect errors.

strtol

The string may begin with an arbitrary amount of white space (as determined by `isspace()`) followed by a single optional `'+'` or `'-'` sign.

The remainder of the string is converted to a `long` value in the obvious manner, stopping at the first character which is not a valid digit in the given base.

mem

Nota: En los prototipos de las funciones se ha sustituido `s[.n]` por `*s` y se han omitido los `const`.

memset

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

The `memset()` function fills the first `n` bytes of the memory area pointed to by `s` with the constant byte `c`. Returns a pointer to memory area `s`.

bzero

```
void bzero(void *s, size_t n);
```

The `bzero()` function erases the data in the `n` bytes of the memory starting at the location pointed to by `s`, by writing zeros (bytes containing `'\0'`) to that area. Return value: none.

memcpy

```
void *memcpy(void *dest, void *src, size_t n);
```

The `memcpy()` function copies `n` bytes from memory area `src` to memory area `dest`. The memory areas must not overlap. Use `memmove()` if the memory areas do overlap. Returns a pointer to `dest`.

memmove

```
void *memmove(void *dest, void *src, size_t n);
```

The `memmove()` function copies `n` bytes from memory area `src` to memory area `dest`. The memory areas may overlap: copying takes place as though the bytes in `src` are first copied into a temporary array that does not overlap `src` or `dest`, and the bytes are then copied from the temporary array to `dest`. Returns a pointer to `dest`.

memchr

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

The `memchr()` function scans the initial `n` bytes of the memory area pointed to by `s` for the first instance of `c`. Both `c` and the bytes of the memory area pointed to by `s` are interpreted as `unsigned char`. Returns a pointer to the matching byte or `NULL` if the character does not occur in the given memory area.

memcmp

```
int memcmp(void *s1, void *s2, size_t n);
```

The `memcmp()` function compares the first `n` bytes (each interpreted as `unsigned char`) of the memory areas `s1` and `s2`.

Returns an integer less than, equal to, or greater than zero if the first `n` bytes of `s1` is found, respectively, to be less than, to match, or be greater than the first `n` bytes of `s2`.

For a nonzero return value, the sign is determined by the sign of the difference between the first pair of bytes (interpreted as `unsigned char`) that differ in `s1` and `s2`.

If `n` is zero, the return value is zero.

malloc

calloc

```
void *calloc(size_t nelem, size_t elsize);
```

strdup

```
char *strdup(const char *s);
```