



LIBFT

Apuntes para aprender a desarrollar la librería

DESCRIPCIÓN BREVE

Este proyecto consiste en programar una librería mediante el lenguaje C. Contiene 34 funciones de propósito general y 9 funciones relacionadas con listas. Con estos apuntes se pretende acercar a l@s alumnos un conocimiento más completo sobre las tareas a realizar.

Pere Rovira Casas

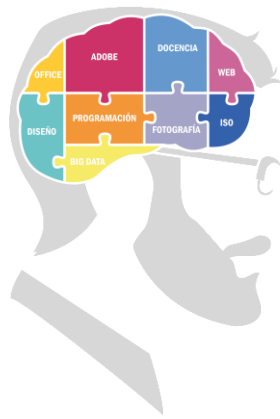
42 Cursus – v01

CURSUS42

LIBFT.H

Introducción.....	4
Comprobaciones booleanas	5
Arrays.....	6
Punteros	7
free(void *ptr)	10
Comparar punteros if (ptr2 > ptr1)	11
Bucles while.....	14
while (n < nbr)	14
while (n > nbr)	14
while(*str != '\0') / while(!str)	15
while(bool <operators> bool).....	17
while(n--)... ..	18
Condicionales if (puntero)	19
Type Casting	20
Función como parámetro	21
File Descriptor	23
stdout	23
STDOUT_FILENO	24
fflush (stdio.h)	25
ft_atoi().....	27
ft_bzero()	29
ft_calloc().....	30
ft_isalnum()	33
ft_isalpha().....	34
ft_isascii().....	35
ft_isdigit()	36
ft_isprint().....	37
ft_itoa().....	38
ft_memchr()	41
ft_memcmp().....	42
ft_memcpy()	43
ft_memmove().....	44
ft_memset().....	45
ft_putchar_fd()	46
ft_putendl_fd() / endl – C++	47
ft_putnbr_fd().....	49
ft_putstr_fd() / puts() – C++	51
ft_split() / strtok – C++.....	52
ft_strchr().....	55
ft_strdup()	56
ft_striteri()	57
ft_strjoin().....	58
ft_strlcat().....	59
ft_strlcpy()	60
ft_strlen().....	61
ft_strmapi().....	62
ft_strncmp().....	64
ft_strnstr()	66
ft_strrchr()	67

ft_strtrim()	68
ft_substr() / substr – C++	70
ft_tolower()	71
ft_toupper()	72
Header o archivo de cabecera	73
Directivas	74
#include	74
#define	74
#ifdef/#ifndef	74
libft.h	75
Compilación	76
Bibliotecas de funciones	76
gcc	76
ar	76
Test	77
make – Makefile	78
Reglas	79
Definiciones de variables	80
Ruta de archivos	82
Makefile 1.1.4	83
BONUS – Listas Enlazadas (básico)	84
Definición	84
Insertar un elemento en la lista	84
Aplicación de la función	85
ft_lstnew()	86
ft_lstadd_front()	88
ft_lstsize()	89
ft_lstlast()	90
ft_lstadd_back()	91
ft_lstdelone()	92
ft_lstclear()	93
ft_lstiter()	94
ft_lstmap()	95



[LinkedIn](#)

[GitHub](#)

[YouTube](#)

Introducción

A continuación, se presenta una serie de apuntes orientativos sobre el primer proyecto del [Cursus42](#), donde debemos crear nuestra biblioteca de funciones. En mi caso, he combinado el desarrollo por mí mismo, copiarlas o bien aplicar mejoras a partir de las propuestas de los compañeros. El objetivo es aprender a desarrollar un código limpio, claro y estructurados. No se ha inventado ninguna rueda, y muchos códigos pueden ser iguales a otros repositorios o manuales didácticos de programación.

Este proyecto consta de dos fases más bonus. El Subject utilizado es la versión 16.

FASE I	FASE II	BONUS
Makefile	ft_substr	ft_lstnew
ft_isalpha	ft_strjoin	ft_lstadd_front
ft_isdigit	ft_strtrim	ft_lstsize
ft_isalnum	ft_split	ft_lstlast
ft_isascii	ft_itoa	ft_lstadd_back
ft_isprint	ft_strmapi	ft_lstdelone
ft_strlen	ft_striteri	ft_lstclear
ft_memset	ft_putchar_fd	ft_lstiter
ft_bzero	ft_putstr_fd	ft_lstmap
ft_memcpy	ft_putendl_fd	
ft_memmove	ft_putnbr_fd	
ft_strlcpy		
ft_strlcat		
ft_toupper		
ft_tolower		
ft_strchr		
ft_strrchr		
ft_strncmp		
ft_memchr		
ft_strnstr		
ft_atoi		
ft_calloc		
ft_strlcat		

Cada capítulo contiene información detallada sobre la función y su código en C. También he redactado capítulos adicionales que ayudarán a entender el comportamiento de las funciones, como, por ejemplo: comprobaciones booleanas, ‘file descriptor’, la función como parámetro, notas sobre el preprocesador, etc.

Esta guía surge de la necesidad de reescribir a mano y explicar detalladamente mediante la búsqueda de información, todas las funciones solicitadas con el fin de retener, estudiar y prestar mejor atención a los ejercicios propuestos. Puede haber capítulos incompletos y/o con algún error de concepto. Mis disculpas.

Si se desea comprobar código mediante “copiar/pegar” se pueden obtener los errores de compilación `stray\200` y `stray\342` debido a que se han copiado caracteres *UNICODE* en vez de *ASCII* en operaciones elementales, como simples o dobles comillas, signos negativos, etc. Para ello, hay que reescribirlos.

- Correcto: `printf("Hello World\n");`
- Error: `printf("Hello World\n");`

Comprobaciones booleanas

Las comprobaciones booleanas son un método efectivo que permiten reducir líneas de código, aunque pueden causar cierta confusión de no entenderse su funcionamiento.

Un ejemplo muy interesante que muestra equivalencia, es el siguiente:

- `while (*pointer != '\0')`
- `while (*pointer)`

La forma más elemental de entender un valor booleano es mediante ceros y unos. Es sencillo captar la idea al afirmar que:

- 0: falso.
- 1: verdadero.

Por tanto, aprendemos la idea de que 0 es cerrado y 1 es abierto. No obstante, cuando programamos, obtenemos distintos rangos de valores y muchas veces ni siquiera sabemos cuáles van a ser. Así pues, algo como esto, podría bien ser equivalente:

- `if (number < 0 || number > 0)`
- `if (number != 0)`

Cuando operamos con verdadero/falso, hay que tener muy claro qué significa cada concepto:

- **Verdadero:** `valor != 0`
- **Falso:** `valor == 0`

Sabemos que los datos no dejan de ser números: una letra es un número, un número decimal es un número, un resultado distinto a 0 es un número. Estos valores hay que entenderlos como tal, así, cualquier valor que sea distinto a 0 será verdadero si así lo definimos a la hora de escribir nuestro código.

Volviendo al ejemplo inicial podemos entender lo siguiente:

- `while (*pointer != '\0')`: Mientras el valor que apunta sea distinto a nulo. (verdadera).
- `while (*pointer)`: Mientras el valor que apunta sea verdadero.
- `while (!*pointer)`: Mientras el valor que apunta sea nulo (falso).

Comprobamos que el segundo `while` puede estar analizando letras ('abcd...') con lo cual es distinto a nulo. También puede estar analizando valores distintos a nulo (-10, 7, 56...), con lo que también da resultados verdaderos, aunque haya valores negativos.

Y este es el punto clave: ¿Y si existen valores como -57, 7, 0, 56...?

Aquí es el punto donde determina cuándo necesitamos sintetizar o no el código. Por tanto, si no interesa en absoluto tener valores 0 de tipo `int`, (por ejemplo) el código resultará útil, en caso contrario, debemos reformularlo de otra manera.

En cambio, si estamos analizando una cadena de caracteres, `while (*pointer)` puede ser una buena opción, ya que lo examinaremos hasta el final del array. Así pues, en estos apuntes veremos condiciones parecidas a esta, donde comprueba que ambas variables sean distintas a 0, y eso implica que sus valores pueden ser negativos o positivos.

- `if (number && size).`

Arrays

La diferencia entre un array y un puntero suele ser complicada de comprender al principio. Ciertamente, un array y un puntero no son lo mismo. Un array consta de una serie de celdas (direcciones de memoria) que almacenan un valor.

- `char str[] = "Hello";` `//'\0'` included by default
- `char str[6] = {'H', 'e', 'l', 'l', 'o', '\0'};` `//must enter '\0'`

MEMORIA	0x7ffd41c6f458	0x7ffd41c6f459	0x7ffd41c6f460	0x7ffd41c6f461	0x7ffd41c6f462	0x7ffd41c6f463
POSICIÓN ARRAY	0	1	2	3	4	5
VALOR	H	e	l	l	o	\0
TAMAÑO	1 byte (8 bits)	1 byte (8 bits)	1 byte (8 bits)	1 byte (8 bits)	1 byte (8 bits)	1 byte (8 bits)
6 bytes / 48 bits						

Si queremos acceder a un valor en concreto del array y, por ejemplo, mostrarlo en pantalla, lo haremos de la siguiente manera:

```
write(1, str[1], 1);           //display the second character >>'e'
```

Output: e

Si queremos mostrarlos todos, podemos hacerlo mediante un bucle:

```
int i;
i = 0;
while(str[i] != '\0')
{
    write(1, str[i], 1);    //remember: str is char type
    i++;
}
```

Output: Hello

Viendo la tabla anterior, entendemos que los arrays son:

- **FILA 1:** Los datos que contienen están ordenados de forma consecutiva, es decir, se encuentran en un rango consecutivo de la memoria y terminan siempre con un valor nulo.
- **FILA 2:** Su posición dentro del array empieza desde 0 hasta donde llegue.
- **FILA 3:** Los datos que contiene son del mismo tipo (`int`, `char`, `long...`).
- **FILA 4:** Cada celda que contiene el valor tiene una capacidad de x bits según el tipo de dato que contenga. Por ejemplo, cada celda de un array de tipo `int` tendrá una capacidad de 32 bits (4 bytes), mientras que cada celda de un array de tipo `char` tendrá una capacidad de 8 bits (1 byte)¹.

Por tanto: Un array de 6 elementos de tipo `char` ocupa 6 bytes, que eso son 48 bits, ya que $48 = 6 * 8$.

Así que, por un momento imaginemos un mueble con distintos cajones donde cada cajón guarda un objeto de valor y quedémonos con esa idea.

Más información: C Programming Strings

¹ Char, Short, Int and Long Types, MQL4

Punteros

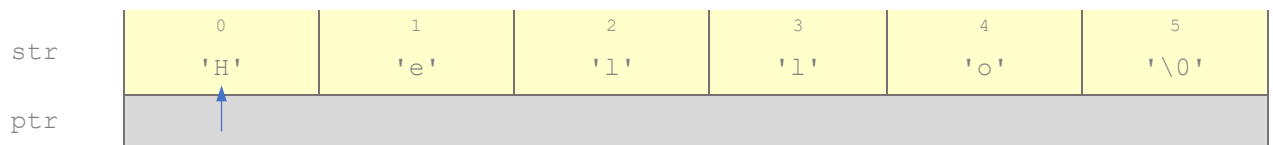
Un puntero es una variable que apunta a una posición de memoria de un array. La peculiaridad de los punteros es que pueden confundirse fácilmente con un array, ya que aparentemente funcionan igual, pero eso no es así. Gracias a los punteros, podemos reservar o liberar memoria dinámicamente en tiempo de ejecución.

Cuando creamos un puntero debemos tener claro su uso y/o finalidad, por eso es importante que al declararlo sea del mismo tipo que el array al que apuntará. Para diferenciarlo de una variable, lo antecedemos con un asterisco.

```
char str[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
char *ptr;
```

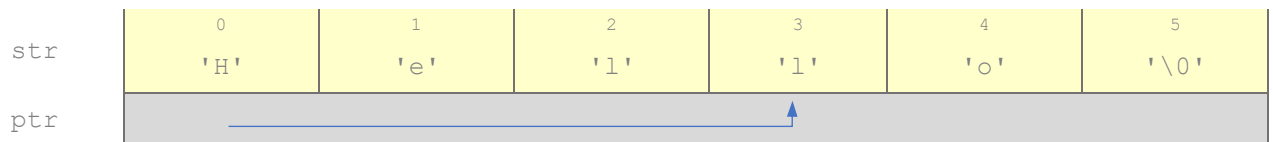
Con la siguiente línea, indicamos que `ptr` apunta a la primera posición del array `str`:

```
ptr = str;
```



Si queremos apuntar a otra posición de memoria, simplemente se lo indicamos mediante `'&'`:

```
ptr = &str[3];
```



Una vez controlamos la posición de memoria a la que accedemos, para operar con el valor del array mediante el puntero, lo haremos con el asterisco. Un par de ejemplos mediante `write`² y `printf`.

- `write(1, &*ptr, 1);`
- `printf("%c", *ptr);`

Output: `l`

Y si queremos iterar un puntero, podemos hacer un bucle como el que sigue:

```
int i;
i = 0;
while(ptr[i] != '\0')
{
    write(1, &ptr[i], 1);
    i++;
}
```

Output: `Hello`

² Se recomienda estudiar la función `write` para entender los motivos de introducir el carácter `'&'`.

Entonces, fijémonos en cómo es cada uno de los códigos propuestos que nos imprime una palabra en pantalla:

Array	Puntero
<pre>#include <unistd.h> int main(void) { int i; char str[] = "Hello"; i = 0; while (str[i] != '\0') { write(1, &str[i], 1); i++; } return (0); }</pre>	<pre>#include <unistd.h> int main(void) { int i; char str[] = "Hello"; char *ptr; ptr = str; i = 0; while (ptr[i] != '\0') { write(1, &ptr[i], 1); i++; } return (0); }</pre>

Simplifiquemos mucho más el código mediante `printf()`

Array	Puntero
<pre>#include <stdio.h> int main(void) { char str[] = "Hello"; printf("%s", str); return (0); }</pre>	<pre>#include <stdio.h> int main(void) { int i; char str[] = "Hello"; char *ptr; ptr = str; printf("%s", ptr); return (0); }</pre>

Aunque aparentemente muestran el mismo resultado, el modo en cómo operan es muy distinto. Si recordamos el mueble que representa el array, en el caso del puntero podemos decir que es un cajón externo vinculado a posiciones de la memoria del array apuntado. Por tanto, hay características interesantes a tener en cuenta siguiendo nuestro ejemplo:

- Cuando pasamos el array por parámetro, estamos pasando 6 bytes (en este caso `Hello\0`) por parámetro, es decir, el mueble entero.
- En caso de pasar un puntero por parámetro, estamos pasando una variable que apunta a una posición del array, es decir, le estamos pasando 1 byte, o lo que sería lo mismo: un cajón vinculado a uno de los cajones del mueble.
- El puntero apunta distintos valores (y puede alterarlos), por lo tanto, un puntero apuntará a posiciones individuales de memoria. Decir que un puntero apunta al `string Hello` no es correcto, en todo caso, apuntará a uno de los caracteres (o posiciones de memoria) del `string`.
- Podemos desvincular un puntero y asociarlo a otro array según necesidades.

Así que ahora es fácil comprobar que no es lo mismo pasar por parámetro `"Hello"` que `'H'`.

sizeof(type-name)

Cuando operamos con datos debemos saber cuánta memoria (en bytes) debemos reservar, y según la plataforma que utilicemos, estos tamaños pueden ser distintos. Entonces, una plataforma de 32 bits no necesita la misma memoria que una plataforma de 64 bits para albergar, por ejemplo, 10 enteros.

La función `sizeof(type-name)`³⁴ recibe un único parámetro para devolvernos un tamaño en bytes. Este tamaño es de tipo `size_t`, definido en la librería `stddef.h`. Esto nos permite desarrollar código capaz de adaptarse a distintas plataformas.

Pero cuando ejecutamos el prototipo `sizeof(type-name *)`, nos devolverá el número de bytes utilizados para almacenar un puntero. En caso de apuntar una ubicación de la memoria de tipo `int`, notaremos que obtenemos resultados de 4 bytes (si el procesador es de 32 bits) o de 8 bytes (si el procesador es de 64 bits)⁵:

```
#include <stdio.h>
int main(void)
{
    int    array[20];
    int    *ptr;

    ptr = array;
    printf("%lu", sizeof(ptr));          //size of a pointer
    return (0);
}
```

Output: 4 << 32 bits machine

Output: 8 << 64 bits machine

Así, cuando utilizamos el prototipo `sizeof(int)`, significa que nos devuelve un número de 4 bytes, es decir, 32 bits ($4 * 8 = 32$), puesto que cada byte está compuesto de 8 bits. En el siguiente ejemplo veremos que la cantidad de bytes para un array de 20 elementos `int` es 80, pues cada elemento de tipo `int` es de 4 bytes, por tanto $4 * 20 = 80$:

```
#include <stdio.h>
int main(void)
{
    int    array[20];
    printf("%lu", sizeof(array));        //size of an array[20]
    return (0);
}
```

Output: 80

Por tanto, cuando utilizamos `sizeof(type-name)` para un tipo de datos, siempre dará un resultado fijo, sea cual fuese el procesador utilizado. Por ejemplo, para el tipo `int` son 4 bytes y para el tipo `char` 1 byte.

```
printf("%ld", sizeof(int));
```

Output: 4

```
printf("%ld", sizeof(char));
```

Output: 1

³ IBM

⁴ CPP Reference

⁵ Diferencia entre `sizeof(int)` i `sizeof(int *)`

free(void *ptr)

Sabemos que podemos reservar memoria dinámicamente mediante los punteros⁶. Con `malloc` podemos hacerlo; simplemente debemos pedirle el tamaño (en bytes) y nos devolverá un puntero inespecífico (`void *`) de ese tamaño. Si no hubiera memoria suficiente nos devolvería un puntero nulo.

- `void *malloc(size_t size); //size >> number of bytes`

Así, para crear un puntero y asignarle memoria lo hacemos así:

- `char *ptr = (char *)malloc(1000);`

Conociendo `sizeof(type-name)`, resulta más sencillo reservar la memoria exacta que necesitamos:

- `int *ptr1 = (int *)malloc(15 * sizeof(int)); // 15 values type in`
- `char *ptr2 = (char *)malloc(5 * sizeof(char)); // 5 values type char`

Con `ft_calloc` podemos hacer lo mismo, pero asegurándonos que la memoria reservada la inicializábamos a cero.

- `int *ptr1 = (int *)ft_calloc(15, sizeof(int));`
- `char *ptr2 = (char *)ft_calloc(5, sizeof(char));`

Partamos de la base que existen dos tipos de memoria: *Stack (pila)* y *Heap (memoria dinámica)*⁷. Los punteros se crean en la memoria dinámica⁸. Esto es una gran ventaja, pues podemos asignar y reasignar memoria en tiempo de ejecución. Es sencillo deducir que a medida que la llenamos, esta se va terminando, y por eso, cuando ya no la necesitemos deberemos liberarla para seguir ejecutando los procesos del programa y mantener un óptimo rendimiento. Por tanto, si creamos un puntero que utilizamos para una serie de cálculos, hay que liberar esa memoria cuando ya no lo usemos más:

- `free(void *ptr);`

Ejemplo:

```
#include <stdlib.h>
int main(void)
{
    while(1)
    {
        int *ptr = (int *)malloc(100 * sizeof(int));
        //operaciones con ptr...
        //...

        free(ptr);
    }
    return (0);
}
```

⁶ `ft_calloc >> malloc`

⁷ Memoria dinámica: `malloc` y `free`

⁸ ¿Cómo funciona/se usa la función `free` en C?, [Stackoverflow](#)

Comparar punteros if (ptr2 > ptr1)

Un puntero apunta a una dirección de memoria y el puntero tiene el poder de alterar los valores que alberga esa dirección de memoria. Digamos que, desde la lejanía, un puntero puede modificar la posición de un ítem que contenga un array. Simplemente hay que vincular el puntero con el array, y analizarlo de las maneras oportunas para manipular el array.

Cuando ya lo tenemos algo controlado, puede que lleguemos a un punto parecido al siguiente, donde se tiende a pensar erróneamente que se compara la longitud de dos punteros:

- `if (str2 > str1)`

Fijémonos en el siguiente ejemplo donde la longitud en bytes entre dos punteros que apuntan a distintos arrays del mismo tipo, arroja un resultado igual:

```
#include <stdio.h>
int main(void)
{
    char    str1[] = "0123456789";
    char    str2[] = "123456";
    char    *ptr1;
    char    *ptr2;

    ptr1 = str1;
    ptr2 = str2;
    printf("str1: %ld\n", sizeof(str1));
    printf("str2: %ld\n", sizeof(str2));
    printf("ptr1: %ld\n", sizeof(ptr1));
    printf("ptr2: %ld\n", sizeof(ptr2));
    printf("char: %ld\n", sizeof(char));
    return (0);
}
```

Output:

```
str1: 11 bytes
str2: 7  bytes
ptr1: 8  bytes    //<<
ptr2: 8  bytes    //<<
char: 1  bytes
```

Entonces la pregunta es: ¿Cómo se comparan los punteros?

Para realizar una comparación necesitamos que los punteros sean del mismo tipo. Si no lo fueran, podrán compararse después de la conversión (`type cast`)⁹.

Comprobar que los punteros apuntan a la misma dirección de memoria.

- `ptr1 == ptr2`

Comprobar que los punteros no apuntan a la misma dirección de memoria.

- `ptr1 != ptr2`

Comprobar que los punteros apuntan al mismo valor.

- `*ptr1 == *ptr2`

Comprobar que los punteros no apuntan al mismo valor.

- `*ptr1 != *ptr2`

Comprobar que la memoria apuntada por `ptr1` es inferior a la apuntada por `ptr2`.

- `ptr1 < ptr2`

Comprobar que la memoria apuntada por `ptr1` es superior a la apuntada por `ptr2`.

- `ptr1 > ptr2`

Ejemplo 1:

```
#include <stdio.h>
int main(void)
{
    char str[] = "0123456789";
    char *ptr1;
    char *ptr2;

    //Output: Equal!
    ptr1 = str;
    ptr2 = str;

    //Output: Different!
    //ptr1 = &str[3];
    //ptr2 = &str[4];

    if (ptr1 == ptr2)
        printf("Equal!");
    else
        printf("Different!");
    return (0);
}
```

⁹ How to compare pointers, Stackoverflow

Ejemplo 2:

```
#include <stdio.h>
int main(void)
{
    char    str1[] = "9876543210";
    char    *ptr1;
    char    *ptr2;

    ptr1 = &str1[2]; //point to char '7'
    ptr2 = &str1[4]; //point to char '5'

    printf("ptr1 value:\t%c\tmemory location:\t%p\n", *ptr1, &ptr1);
    printf("ptr2 value:\t%c\tmemory location:\t%p\n", *ptr2, &ptr2);

    if (ptr1 > ptr2)
        printf("ptr1 > ptr2 = %s", ptr1 > ptr2 ? "true" : "false");
    else
        printf("ptr1 < ptr2 = %s", ptr1 < ptr2 ? "true" : "false");

    return (0);
}
```

Output: (los valores de la memoria pueden variar)

```
ptr1 value:      7          memory location:      0x7ffd41c6f458
ptr2 value:      5          memory location:      0x7ffd41c6f460
ptr1 < ptr2 = true
```

Bucles while

A lo largo de este documento encontraremos una forma muy común de ejecutar el bucle `while`, no obstante, es importante entender su funcionamiento y los tipos de iteraciones que puede haber para escoger la que más nos interese. En todo caso, antes de nada, sería interesante entender ciertas formas de asignarle un valor a la variable iterativa. Existen dos formas¹⁰:

- **`n-- / n++`**: Primero toma el valor de la variable y luego reduce/incrementa en una unidad.
- **`--n / ++n`**: Primero reduce/incrementa en una unidad y luego toma el valor de la variable.

```
int n = 3;
int p = n++; //now n is 4 and p is 3 (same of n += 1 or n + 1)
```

```
int n = 3;
int p = ++n; //now n is 4 and p is 4
```

while (n < nbr)

En este caso el bucle se ejecutará mientras `n` sea inferior a `nbr`:

```
int main(void)
{
    int n = 0;
    int nbr = 10;

    while (n < nbr)
    {
        printf("%d", n);
        n++;
    }
    return (0);
}
```

Output: 0 1 2 3 4 5 6 7 8 9

while (n > nbr)

En este caso el bucle se ejecutará mientras `n` sea superior a `nbr`:

```
int main(void)
{
    int n = 10;
    int nbr = 0;

    while (n > nbr)
    {
        printf("%d", n);
        n--;
    }
    return (0);
}
```

Output: 10 9 8 7 6 5 4 3 2 1

¹⁰ Difference between ++n and n++ - cplusplus.com

```
while(*str != '\0') / while(!str)
```

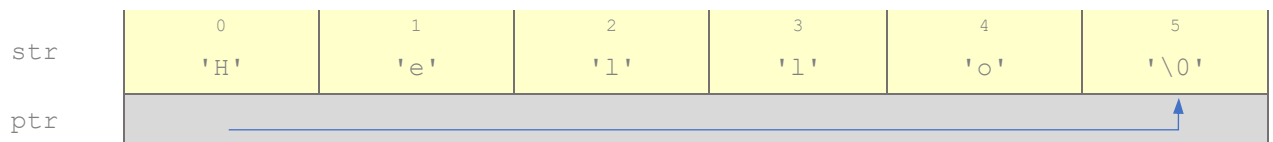
Aquí recorreremos un array mediante un puntero, examinando el contenido de la posición a la que apunta. Le preguntamos que mientras la posición de memoria sea distinta a nulo, haga algo. En estos casos es muy importante mover la posición del puntero, pues de no hacerse, el bucle no avanza:

Programa 1:

```
int main(void)
{
    char    str[] = "Hello";
    char    *ptr;

    ptr = str;
    while (*ptr != '\0')
        ptr++; //increase 1 position
    return (0);
}
```

Es muy importante entender que estamos iterando el array, es decir: estamos indicando al puntero que apunte a +1 dirección de memoria hasta que encuentre un valor `NULL`, eso significa que, en caso de recuperarlo a posteriori, seguiría apuntando a la última posición, y acciones como `printf("%s", ptr)` daría un resultado `NULL`.

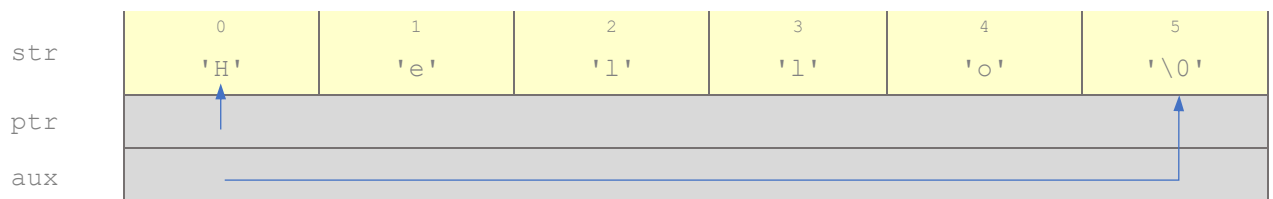


Una forma de mantener la posición es crear un puntero auxiliar, así el código quedaría de la siguiente manera:

Programa 2:

```
int main(void)
{
    char    str[] = "Hello";
    char    *ptr;
    char    *aux;

    ptr = str;
    aux = ptr;
    while (*aux != '\0')
        aux++; // move aux +1 position but keeps main position of ptr
    return (0);
}
```

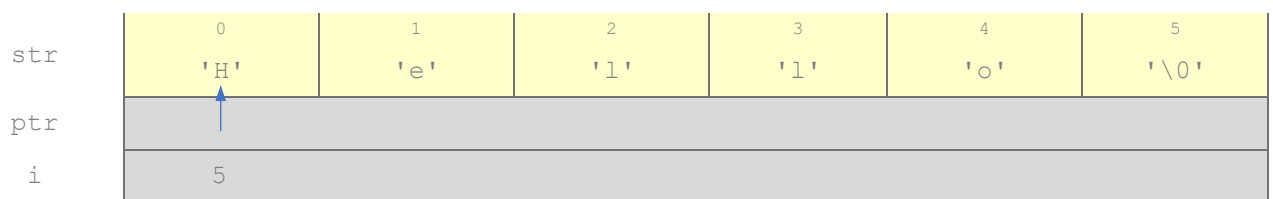


Programa 3:

Un equivalente al puntero auxiliar sería mediante una variable de iteración:

```
int main(void)
{
    char str[] = "Hello";
    char *ptr;
    int i;

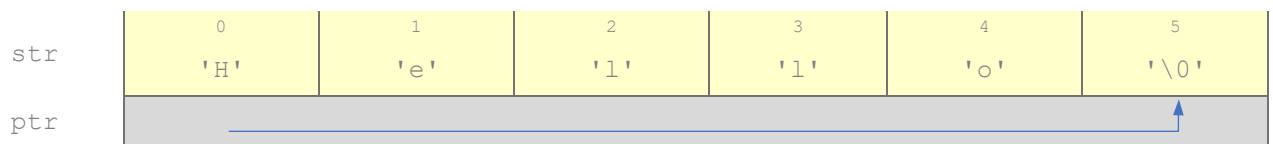
    i = 0;
    ptr = str;
    while(ptr[i] != '\0')
        ptr[i++];
    return (0);
}
```



Podemos ver que el código puede escribirse según nuestras necesidades, aunque también, según sea la mejor comprensión o planteamiento para entender el código, así como su finalidad. En el siguiente ejemplo se muestra un símil del **Programa 1**, con el siguiente planteamiento: mientras `*ptr == true`, esto es lo mismo que decir mientras `*ptr != '\0'`.

```
int main(void)
{
    char str[] = "Hello";
    char *ptr;

    ptr = str;
    while (*ptr)
        ptr++;
    return (0);
}
```



`while(bool <operators> bool)`

Ahora que sabemos que `true != 0` y `false == 0`, resultará sencillo entender bucles como los siguientes:

Mientras el contenido del valor que apunta `*str != nulo` y la función dé un resultado `!= nulo`.

- `while(*str && function(str))`

Mientras `len > -1` y `*str` apunte a una dirección de memoria `!= nulo`.

- `while (len-- && *s)`

Mientras `i` sea inferior al resultado de la función.

- `while (i < function(str))`

Paralelamente, disponemos de dos maneras de representar exactamente lo mismo con una variable indexadora¹¹. En estos casos, como pasó en el capítulo de comprobaciones booleanas, se utilizará el código que sea más comprensible.

Mientras el contenido de la posición `*str[i] != nulo`.

- `while (*(str + i))`
- `while (str[i] != '\0')`

A continuación, otro ejemplo donde las dos líneas hacen exactamente lo mismo¹².

- `while (*(str + i) && *(str + i) != c)`
- `while (str[i] && str[i] != c)`

Otra forma equivalente de incrementar +1 la posición del puntero, con conversión incluida:

- `buf1 = (unsigned char *)buf1++;`
- `buf1 = (unsigned char *)buf1 + 1;`

Ejemplo aplicado:

```
#include <stdio.h>
int main(void) {
    char str[] = "abcdefghijklmnopqrstuvwxyz";
    char *ptr;
    int i;

    ptr = str;
    i = 0;
    while(str[i] != '\0')    //while(*(str + i))
    {
        printf("%c ", *(str + i));
        printf(">>%c | ", str[i]);
        i++;
    }
    return (0);
}
```

¹¹ What's the difference between `ptr[i]` and `*(ptr + i)`?, stackoverflow

¹² 18.5.3 Pointer element access, C# Language Specification 5.0, , Microsoft

while(n--)

Este es uno de los bucles más usados en los ejercicios debido a su enorme sencillez y facilidad de uso¹³. Irá decrementando en una unidad hasta $n < 0$. El valor de n una vez termine el bucle será $n == -1$. Los códigos más habituales son parecidos a este, ya sea para contar la longitud de un array, mostrar en pantalla un número determinado de caracteres del array, iterar un array...:

```
int    main(void)
{
    int    n;
    int    sb;
    char    str[] = "Hello";
    char    *ptr;

    ptr = str;
    n = 0;
    sb = 4;
    while(n < sb)
    {
        write(1, &*ptr, 1);
        n++;
        ptr++;
    }
    return (0);
}
```

Fijémonos en el siguiente ejemplo, donde mediante el uso de dos sentidos de iteración, conseguimos el mismo resultado con menos líneas de código.

```
int    main(void)
{
    int    n;
    char    str[] = "Hello";
    char    *ptr;

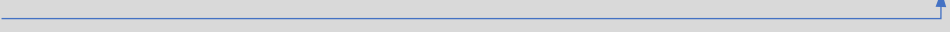
    ptr = str;
    n = 5;
    while(n--)
        write(1, &*ptr++, 1);
    return (0);
}
```

¹³ Difference between while(n--) and while (n = n - 1), Stackoverflow

Si sustituimos la línea del bucle por la siguiente, tendréis una idea de cómo funciona la iteración. Veréis que mientras `n` se reduce hasta `-1`, `ptr` apunta a la siguiente posición desde `0`, consiguiendo dos recorridos simultáneos:

```
printf("%d:%c | ", n, *ptr++);
```

Output: 3:H | 2:e | 1:l | 0:l | 0:o |

	0	1	2	3	4	5
str	'H'	'e'	'l'	'l'	'o'	'\0'
ptr						
n	-1					

Condicionales if (puntero)

Igual que con el bucle `while`, tenemos modos de saber el estado de un valor para seguir operando con punteros. Los más significativos son los siguientes:

Si el puntero es nulo, `str == falso`:

- `if (!str)`

Si el valor consecutivo apuntado por `*str == c`:

- `if (*str++ == c)`

Si el valor que apunta un puntero es distinto al que apunta otro puntero, `*str1 != *str2`:

- `if (*str1 != *str2)`

Type Casting

En algunas partes del código encontraremos conversiones de valores. Esto sirve para almacenar tipos de valores distintos en variables ya definidas. Por ejemplo¹⁴:

```
int      a = 17;
double   b;
b = (double)a / 5;
```

Comprobamos que debemos tener una variable del tipo que nos interesa (`double`) que albergará variables de otro tipo (en este caso `int`) convertidas a `double` para almacenar correctamente el resultado.

- Output (casting): 3.400000
- Output no cast: 3.000000

Para la conversión debemos insertar entre paréntesis el tipo de dato que utilizaremos seguido de la variable de distinto tipo:

```
(double) a
```

En caso de utilizar punteros, convertimos los valores que albergan, por tanto (nótese el asterisco):

- `*ptr`

Pasa a ser:

- `*(double)ptr`

Pudiendo encontrar líneas de código como la siguiente, donde ambos datos son del mismo tipo:

- `if (*(unsigned char *)ptr++ == (unsigned char)c)`
- `if (*str == (char)c) //str is declared as char str[]`

¹⁴ C – Type Casting, Tutorialspoint
42Barcelona

Función como parámetro

Existe la posibilidad de pasar funciones por parámetro¹⁵. La forma de definir el prototipo es el siguiente:

```
void myfunction(void (*f)(int));
```

- **void myfunction():** Es la función que va a contener los parámetros que le indiquemos. Dara un resultado de tipo `void`.
- **(*f):** es un puntero que apunta a la función entrada por parámetro y tiene un retorno `void`.
- **(int):** indica que la función por parámetro tiene un parámetro de tipo `int`.

En el siguiente ejemplo, comprobamos como la función `void printNumber(int nbr)` imprime por pantalla su parámetro.

```
#include <stdio.h>
void printNumber(int nbr)
{
    printf("%d\n", nbr);
}
```

A continuación, tenemos una nueva función que recibe la función `void` por parámetro. Para usarla correctamente debemos usar la siguiente sintaxis:

- `(*f)(i);`

Donde indicamos la identificación del parámetro y la variable que vamos a usar, quedando de la siguiente manera:¹⁶

```
void myFunction(void (*f)(int))
{
    for(int i = 0; i < 5; i++)
    {
        (*f)(i);
    }
}
```

Y para usar ambas funciones, creamos el programa que nos ejecuta la función `myFunction()` con el parámetro `printNumber()`.

```
int main(void)
{
    myFunction(printNumber);
    return (0);
}
```

¹⁵ C Programming Language: Passing a Function as a Parameter, jraleman - Medium

¹⁶ How do you pass a function as a parameter in C?, stackoverflow

También es posible almacenar la dirección de una función en un puntero. Partiendo de un ejemplo parecido al anterior, el siguiente programa obtiene la dirección de memoria de una función mediante un puntero, lo usamos como parámetro e imprimimos su dirección en pantalla:

```
#include <stdio.h>
void    printNumber(int n)
{
    printf("%d\n", n);
}

void    myFunction(int value, void (*f)(int))
{
    while(value > -1 && value < 10)
        f(value--);
}

int     main(void)
{
    void    (*f_ptr)(int);

    f_ptr = &printNumber;
    myFunction(9, *f_ptr);
    printf("%p", f_ptr);
    return (0);
}
```

File Descriptor

Un *descriptor de archivo* es un número que identifica de forma única un archivo abierto en el sistema operativo de una computadora. Describe un recurso de datos y cómo se puede acceder a ese recurso.

stdout

Existen unos punteros globales predefinidos de tipo `FILE` que apuntan a un flujo de entrada, salida o salida de errores. Estos se encuentran en la librería `stdio.h`:

Descripción	Identificador	Número de archivo (VALOR)
Entrada desde el teclado	<code>stdin</code>	0
Salida a la consola	<code>stdout</code>	1
Salida de error a la consola	<code>stderr</code>	2

Estos punteros son constantes, esto significa que no podemos alterar su valor en ningún caso, y pueden utilizarse como argumentos en las funciones, como por ejemplo en `fprintf()` o `fputs()`. Por ejemplo:

```
#include <stdio.h>
int main(void)
{
    FILE *fd;

    fd = stdout;
    fprintf(fd, "Hello World\n");
    return (0);
}
```

También podríamos prescindir de la creación del puntero `*fd` para insertar directamente `stdout` a `fprintf()`.

- `fprintf(stdout, "Hello World\n");`

Por ahora, no es necesaria más información al respecto.

Referencias: ¹⁷ ¹⁸ ¹⁹ ²⁰ ²¹ ²² ²³

¹⁷ Descriptor de archivos, IBM

¹⁸ Descriptor de archivo, Wikipedia

¹⁹ Tutorial de STDIN y STDOUT en Lenguaje C, DuarteCorporation Tutoriales, Youtube

²⁰ Descriptores de archivos, Ian Wienand, LibreTexts

²¹ Stdin(3) – Linux manual page

²² Macro stdout ANSI C, conclase.net

²³ ¿Qué son los descriptores de archivos abiertos de Linux?, blogground media

STDOUT_FILENO

Ahora bien, por otro lado, disponemos de las llamadas al sistema (`system call`), como podría ser la función `write()`, entre otras. Nuestro primer ejercicio en 42 fue un simple `write` y el uso de la librería `unistd.h`. Todos atamos cabos tanto en el segundo como en el tercer parámetro. ¿Pero y el primer parámetro?

- `write(1, 'z', 1);`

Sin saberlo, estábamos utilizando el valor asociado a `stdout` para que el sistema supiera que su finalidad es mostrar el valor 'z' en pantalla. Pero si incorporamos como parámetro `stdout` a la función, el compilador dará un problema grave:

- `write(stdout, 'z', 1);`
- `error: passing argument 1 of 'write' makes integer from pointer without a cast.`
- `stdout` debe ser `FILE *`

Para obtener el valor según la biblioteca, debemos utilizar las siguientes constantes:

Descripción	Identificador	Número de archivo (VALOR)
Entrada desde el teclado	STDIN_FILENO	0
Salida a la consola	STDOUT_FILENO	1
Salida de error a la consola	STDERR_FILENO	2

Eso se debe a que tanto `stdin`, `stdout` y `stderr` son macros, mientras que `STDIN_FILENO`, `STDOUT_FILENO` y `STDERR_FILENO` son constantes que contienen los valores definidos. Estos valores son de tipo `int`, y corresponden al tipo de parámetro que necesita la función `write()`. Por tanto, el siguiente programa funciona correctamente:

```
#include <unistd.h>
int main(void)
{
    int fd;

    fd = STDOUT_FILENO;
    write(fd, "z\n", 2);
    return (0);
}
```

Referencias: ²⁴ ²⁵ ²⁶ ²⁷ ²⁸

²⁴ The difference between stdout and STDOUT_FILENO, stackoverflow

²⁵ `fileno()` — Get the file descriptor from an open stream, IBM

²⁶ Macro stdout ANSI C

²⁷ `stdout(3)` - Linux man page

²⁸ File descriptors en Linux

fflush (stdio.h)

Antes de continuar con las funciones, es necesario conocer la función `fflush`, ya que está ligado con la funcionalidad de `ft_putendl_fd` que proviene de `cin/cout` de C++.

Para entender qué hace esta función debemos imaginar el siguiente programa:

```
#include <stdio.h>
int main(void)
{
    int a, b;

    printf("Introduzca el primer numero: ");
    scanf("%d", &a );
    printf("Introduzca el segundo numero: ");
    scanf("%d", &b);
    printf("Los valores son: %d, %d ", a, b);
    return (0);
}
```

Notemos que el programa se detiene a la espera de la introducción de un valor mediante `scanf`. Esto significa que cuando tecleemos el valor, este se almacena en una memoria intermedia (o buffer), y no será hasta que presionemos `intro` que la variable obtendrá el valor pertinente. Además, se producirá un salto de línea automático.

Si en vez de variables de tipo `int`, leemos variables de tipo `char`, nos encontraremos que también va a leernos el salto de línea, con lo que el resultado final no será correcto:

```
#include <stdio.h>
int main(void)
{
    char a, b, c;

    printf("Introduzca primer caracter: ");
    scanf("%c", &a);
    printf("Introduzca segundo caracter: ");
    scanf("%c", &b);
    printf("Introduzca tercer caracter: ");
    scanf("%c", &c);
    printf("Los valores son: %c, %c, %c ", a, b, c);
    return (0);
}
```

Pudiéndolo resolver modificando el tipo de resultado:

```
printf("Los valores son: %d, %d, %d ", a, b, c);
```

Pero tampoco nos resuelve el tema, puesto que mostrará los números que pertenecen a los caracteres en vez de los caracteres:

Los valores son: 102, 10, 104

Entonces, como la solución no ha sido buena, la respuesta correcta es vaciar la memoria intermedia del teclado mediante `fflush` antes de leer los caracteres:

```
#include <stdio.h>
int main(void)
{
    char a, b, c;

    printf("Introduzca primer caracter: ");
    scanf("%c", &a);
    printf("Introduzca segundo caracter: ");
    fflush(stdin);
    scanf("%c", &b);
    printf("Introduzca tercer caracter: ");
    fflush(stdin);
    scanf("%c", &c);
    printf("Los valores son: %c, %c, %c ", a, b, c);
    return (0);
}
```

Por tanto, podemos definir a `fflush` como una función que se utiliza solo para los flujos de salida. Su propósito es limpiar el buffer (memoria intermedia) de salida (`stdout`).

Por temas de compatibilidad, no es recomendable usarlo para flujos de entrada (`stdin`), pues su comportamiento puede ser indefinido.

Referencias: ²⁹ ³⁰ ³¹

²⁹ La Función `fflush` en Lenguaje C

³⁰ `fflush`

³¹ ¿para que sirve flush en c++?

ft_atoi()

Prototipo: `int ft_atoi(const char *str);`

Librería: `stdlib.h`

Parámetros:

- ***str:** Dirección en memoria de tipo `char`.

Definición: Convierte una serie de caracteres a un valor entero. Significa *ASCII To Integer*. La serie de entrada es una secuencia de caracteres que puede ser interpretada como un valor numérico con signo. En el momento en que la función no puede seguir leyendo parte del número, la función deja de leer la serie.

En este ejercicio se ha desarrollado una función `static` que permite detectar espacios y caracteres no imprimibles, comprendidos entre 9 y 13 (incluidos).

Limitaciones: No reconoce ni puntos decimales ni exponentes. Rango válido `-2147483648` al `2147483647` (incluidos). Para el valor negativo es necesario realizar un `cast` para representar el símbolo y tener espacio suficiente para representar el signo menos.

Valores devueltos:

- **Int:** En caso de convertir la serie de entrada correctamente.
- **0:** Si no puede convertir la serie de entrada en un número.
- **No definido:** En caso de desbordamiento.

Otras funciones: `atof`, `atol`, `strtod`, `strtol`

Notas sobre el cálculo:

```
n = (n * 10) + (sign * (*str - 48));
```

- $1 = (0 * 10) + (1 * (49 - 48))$
- $12 = (1 * 10) + (1 * (50 - 48))$
- $123 = (12 * 10) + (1 * (51 - 48))$
- $1234 = (123 * 10) + (1 * (52 - 48))$
- $12345 = (1234 * 10) + (1 * (53 - 48))$

Referencias: ³² ³³ ³⁴ ³⁵

³² Wikibooks.org

³³ IBM

³⁴ Aprende a programar

³⁵ Microsoft

Algoritmo: incluye `ft_isdigit()` y `INT_MIN` y `INT_MAX` de la biblioteca `limits.h`

```
static int  check_space(int c)
{
    if ((c > 8 && c < 14) || (c == 32))
        return (1);
    return (0);
}

int  ft_atoi(const char *str)
{
    long long int    n;
    int              sign;

    n = 0;
    sign = 1;
    while (*str && check_space(*str))
        str++;
    if (*str == '-' || *str == '+')
    {
        if (*str == '-')
            sign *= -1;
        str++;
    }
    while (*str && ft_isdigit(*str))
    {
        n = (n * 10) + (sign * (*str - '0'));
        if (n < INT_MIN || n > INT_MAX)
            return (0);
        str++;
    }
    return (n);
}
```

Programa:

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

int  main(void)
{
    char  str1[] = "    \t - + 123456789";
    char  str2[] = "    \t \n 123456789";
    printf("%d\n", ft_atoi(str1));
    printf("%d", atoi(str2));
    return (0);
}
```

ft_bzero()

Prototipo: `void ft_bzero(void *s, size_t n);`

Librería: `strings.h`

Parámetros:

- ***s:** Dirección en memoria de tipo `void`.
- **n:** Tamaño de los datos de tipo `size_t`.

Definición: Partiendo de la posición de `*s`, pone a cero los primeros `n` bytes del área de memoria.

Limitaciones: Esta función no es estándar de C³⁶, por lo que es preferible usar la función `memset()`.

Valores devueltos: Ninguno.

Otras funciones: `memset`, `swab`

Referencias: ³⁷ ³⁸ ³⁹

Algoritmo:

```
void ft_bzero(void *s, size_t n)
{
    while (n--)
        *(unsigned char *)s++ = 0;
}
```

³⁶ Why user bzero over memset?, Stackoverflow

³⁷ IBM

³⁸ MAN7

³⁹ Manpages.ubuntu

ft_calloc()

Prototipo: `void *ft_calloc(size_t number, size_t size);`

Librería: `stdlib.h` y `malloc.h`

Parámetros:

- **number:** Número de elementos que albergará, de tipo `size_t`.
- **size:** Tamaño en bytes de cada elemento que albergará, de tipo `size_t`.

Definición: Significa *'Clear Allocation'*, y se utiliza para asignar y gestionar memoria dinámica en tiempo de ejecución. Para su correcto funcionamiento necesita de la función `malloc` y así obtener el espacio de memoria solicitado.

Esta función reserva un bloque de memoria continua de bytes, permitiendo acceder a este directamente. El valor devuelto es un puntero apuntando el primer elemento de la zona asignada, con todas sus celdas inicializadas a 0. Esto asegura que la memoria no contenga elementos inicializados que puedan contener datos basura.

- `void *calloc(elements, sizeof(data_type))`

Limitaciones: Además de ser más lento que otros métodos de asignación de memoria (`malloc`), no es compatible con todos los tipos de datos, es decir, no funciona con los tipos de datos *no primitivos* como los punteros o los structs.

Valores devueltos:

- ***pointer:** si no puede reservar la memoria solicitada, devuelve un puntero `NULL`.
- **Comportamiento indefinido:** En caso de que `number` o `size` sea 0 devuelve un puntero a un bloque de memoria con tamaño distinto a cero. El resultado se obtiene cuando se intenta leer dicho puntero.

Otras funciones: `malloc`

Notas sobre el cálculo⁴⁰: `number > (UINT_MAX / size)`

Debemos asegurarnos de que existe suficiente espacio para seguir con el proceso. Una de las condiciones es saber si la cantidad de elementos es superior a la división entre el valor máximo de `unsigned int` entre `size`. Eso es debido a que no existen posiciones de memoria negativas, por tanto, no tiene sentido introducir posiciones inferiores 0.

Siempre se empezará a introducir datos o a iterar, a partir de la posición 0 del puntero. `UINT_MAX` es un valor de 32 bits, así que será el máximo permitido de memoria que puede reservar un puntero o array, donde el rango es de 0 a 4294967295⁴¹.

Por tanto, la expresión `number > (UINT_MAX / size)` indica que el número de elementos no puede ser mayor a `UINT_MAX / sizeof(type-name)`.

Ejemplo 1: El puntero tiene espacio para 5 elementos de 32 bits cada uno.

- `int *numbers = ft_calloc(5, sizeof(int));`

Ejemplo 2: El puntero no tiene espacio suficiente, pues sobrepasamos la capacidad permitida.

- `int *numbers = ft_calloc(UINT_MAX + 1, sizeof(int));`

⁴⁰ Vector of pointer to object, max size - Stackoverflow

⁴¹ Numeric Limits, cppreference.com

Notas sobre el cálculo: `memory = malloc(number * size);`

Asignación dinámica de memoria en tiempo de ejecución. `Malloc` pertenece a la biblioteca `stdlib.h`, y solo admite un parámetro, que es la cantidad de bytes asignados. En el siguiente ejemplo se muestra como asignamos memoria a un puntero mediante la conversión de datos. La variable `n` indica el número de elementos que necesitamos:

- `ptr = (char *)malloc(sizeof(char) * (n + 1)); // +1 is for last item '\0'`

Referencias: [42](#) [43](#) [44](#) [45](#) [46](#) [47](#) [48](#)

Algoritmo: (incluye `UINT_MAX` de la biblioteca `limits.h` y la función `ft_bzero()`)

```
void *ft_calloc(size_t number, size_t size)
{
    void *memory;

    if (number && size && number > (UINT_MAX / size))
        return (NULL);
    memory = malloc(number * size);
    if (!memory)
        return (NULL);
    ft_bzero(memory, number * size);
    return (memory);
}
```

⁴² IBM

⁴³ Microsoft

⁴⁴ Aprende Informaticas

⁴⁵ Función `calloc` en C - Youtube

⁴⁶ What is `calloc` - educative.io

⁴⁷ `malloc` VS `calloc` – Fernando Cortés

⁴⁸ `malloc` - Wikipedia

Programa:

```
#include <limits.h>
#include <malloc.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char str[] = {'H', '\0'};
    char *ptr;

    ptr = str;
    printf("*ptr:\t\t\t%s\n", ptr);
    printf("*ptr size of bytes:\t%ld\n", sizeof(ptr));
    printf("*ptr size of bits:\t%ld\n\n", sizeof(ptr) * sizeof(char *));

    // comment next two lines to check ft_calloc()
    ptr = ft_calloc(sizeof(str), sizeof(char *));
    printf("ptr = ft_calloc(sizeof(str), sizeof(char *));\n\n");

    //if success
    if (!*ptr)
        printf("*ptr:\t\t\t%p\n", NULL);
    else
        printf("*ptr:\t\t\t%s\n", ptr);
    printf("*ptr size of bytes:\t%ld\n", sizeof(ptr));
    printf("*ptr size of bits:\t%ld\n\n", sizeof(ptr) * sizeof(char *));
    return (0);
}
```

ft_isalnum()

Prototipo: `int ft_isalnum(int c);`

Librería: `ctype.h`

Parámetros:

- **c:** Valor de tipo `int` que se va a probar.

Definición: Comprueba si hay un carácter alfanumérico comprendido entre los rangos:

- '0' – '9'
- 'A' – 'Z'
- 'a' – 'z'

Limitaciones: N/A.

Valores devueltos:

- **0:** En caso de NO encontrar un valor alfanumérico.
- **1:** En caso de encontrar un valor alfanumérico.
- **Comportamiento indefinido:** En caso de que el parámetro `c` no sea `EOF` o en el intervalo de `0x00` – `0xFF` (ambos incluidos). En caso de usar bibliotecas `CRT`, la función genera una aserción, es decir, considera que es siempre cierta.

Otras funciones: `iswalnum`, `_isalnum_l`, `_iswalnum_l`

Referencias: ⁴⁹ ⁵⁰ ⁵¹ ⁵²

Algoritmo:

```
int ft_isalnum(int c)
{
    if ((c >= '0' && c <= '9')
        || (c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z'))
        return (1);
    return (0);
}
```

⁴⁹ IBM

⁵⁰ isalnum - Tutorialspoint

⁵¹ Aprende Informaticas

⁵² ctype – conclase.net

ft_isalpha()

Prototipo: `int ft_isalpha(int c);`

Librería: `ctype.h`

Parámetros:

- **c:** Valor de tipo `int` que se va a probar.

Definición: Comprueba si el carácter alfanumérico se encuentra entre los rangos 'A' – 'Z' y 'a' – 'z'.

Limitaciones: N/A.

Valores devueltos:

- **0:** En caso de NO encontrar el valor alfanumérico en el rango esperado.
- **1:** En caso de encontrar el valor alfanumérico en el rango esperado.
- **Comportamiento indefinido:** En caso de que el parámetro `c` no sea `EOF` o en el intervalo de `0x00` – `0xFF` (ambos incluidos). En caso de usar bibliotecas `CRT`, la función genera una aserción, es decir, considera que es siempre cierta.

Otras funciones: `iswalpha`, `_isalpha_l`, `_iswalpha_l`

Referencias: ⁵³ ⁵⁴ ⁵⁵ ⁵⁶

Algoritmo:

```
int ft_isalpha(int c)
{
    if ((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z'))
        return (1);
    return (0);
}
```

⁵³ IBM

⁵⁴ Microsoft

⁵⁵ `isalpha` – aprendeaprogramar.com

⁵⁶ `ctype` – conclase.net

ft_isascii()

Prototipo: `int ft_isascii(int c);`

Librería: `ctype.h`

Parámetros:

- **c:** Valor de tipo `int` que se va a probar.

Definición: Comprueba si el valor `int` pertenece a la tabla `ASCII` en el entorno local actual, comprendido entre `0x00` – `0x7F`. Son caracteres `US-ASCII` de 7 bits.

Limitaciones: Puede verse afectada por la categoría `LC_CTYPE` del entorno local actual.

Valores devueltos:

- **0:** En caso de NO encontrar el valor alfanumérico en el rango esperado.
- **1:** En caso de encontrar el valor alfanumérico en el rango esperado.

Otras funciones: `isascii`, `__isascii`, `iswascii`

Referencias: ⁵⁷ ⁵⁸ ⁵⁹ ⁶⁰

Algoritmo:

```
int ft_isascii(int c)
{
    if (c >= 0 && c <= 127)
        return (1);
    return (0);
}
```

⁵⁷ IBM

⁵⁸ Microsoft

⁵⁹ `isascii` – manpages

⁶⁰ `ctype` – conclase.net

ft_isdigit()

Prototipo: `int ft_isdigit(int c);`

Librería: `ctype.h`

Parámetros:

- **c:** Valor de tipo `int` que se va a probar.

Definición: Comprueba si el valor `int` es un dígito comprendido entre '0' – '9'.

Limitaciones: Puede verse afectada por la categoría `LC_CTYPE` del entorno local actual.

Valores devueltos:

- **0:** En caso de NO encontrar el valor en el rango esperado.
- **1:** En caso de encontrar el valor en el rango esperado.

Otras funciones: `ctype`, `isalpha`, `isupper`, `islower`, `isdigit`, `isxdigit`, `isalnum`, `isspace`, `ispunct`, `isprint`, `isgraph`, `iscntrl` o `isascii` Subroutines

Referencias: ⁶¹ ⁶² ⁶³ ⁶⁴

Algoritmo:

```
int ft_isdigit(int c)
{
    if (c >= '0' && c <= '9')
        return (1);
    return (0);
}
```

⁶¹ IBM

⁶² `isdigit` – aprendeaprogramar.com

⁶³ `isdigit` – conclase.net

⁶⁴ `ctype` – programiz.com

ft_isprint()

Prototipo: `int ft_isprint(int arg);`

Librería: `ctype.h`

Parámetros:

- **arg:** Valor de tipo `int` que se va a probar.

Definición: Comprueba si el valor `int` es un dígito comprendido entre 32 – 126. Este rango contiene todos aquellos caracteres imprimibles según la tabla `ASCII`.

Limitaciones: Puede verse afectada por la categoría `LC_CTYPE` del entorno local actual.

Valores devueltos:

- **0:** En caso de NO encontrar el valor en el rango esperado.
- **1:** En caso de encontrar el valor en el rango esperado.
- **Comportamiento indefinido:** En caso de que el parámetro `c` no sea `EOF` o en el intervalo de `0x00` – `0xFF` (ambos incluidos). En caso de usar bibliotecas `CRT`, la función genera una aserción, es decir, considera que es siempre cierta.

Otras funciones: `iswprint`, `_isprint_l`, `_iswprint_l`

Referencias: ⁶⁵ ⁶⁶ ⁶⁷ ⁶⁸

Algoritmo:

```
int ft_isprint(int arg)
{
    if (arg >= ' ' && arg <= '~')
        return (1);
    return (0);
}
```

⁶⁵ Macro `isprint` ANSI C

⁶⁶ `Isprint`, Microsoft

⁶⁷ `Isprint`, IBM

⁶⁸ `Isprint`, [geeksforgeeks.org](https://www.geeksforgeeks.org)

ft_itoa()

Prototipo: `char *ft_itoa(int n);`

Librería: `stdlib.h`

Parámetros:

- **n:** Valor de tipo `int` que será convertido a una cadena ASCII.

Definición: Esta función se utiliza para convertir un valor `int` en una cadena de texto. La cadena contendrá tantos caracteres como dígitos tenga el resultado. No es una función estándar, con lo que su uso en algunas aplicaciones puede dar comportamientos inesperados.

Limitaciones: El buffer debe tener la suficiente capacidad para albergar la cadena de caracteres.

Valores devueltos:

- ***pointer:** Puntero a un buffer. No se devuelve ningún error.

Otras funciones: `itoa`, `_itoa`, `ltoa`, `_ltoa`, `ultoa`, `_ultoa`, `_i64toa`, `_ui64toa`, `_itow`, `_ltow`, `_ultow`, `_i64tow`, `_ui64tow`

Existe una versión ampliada con los siguientes parámetros que se desarrollará en otra ocasión:

- `char *ft_itoa(int n, char *buffer, int radix);`

Referencias: ^{69 70 71 72}

Resultado:

```

Number:                -123456
Sign:                  1
Converting -123456:    1 *= -123456
Convert to absolute value: 123456
                        Adding +1 to length because negative sign.
Process 1:             -123456 /= 10
Process 2:             -12345  /= 10
Process 3:             -1234   /= 10
Process 4:             -123    /= 10
Process 5:             -12     /= 10
Process 6:             -1      /= 10
Sign:                  1 // 0 if Base Number is positive
Len:                   7
Sign:                  1
Converting -123456:    1 *= -123456
Nbr:                   123456
Char while (len--):    123456 | 12345 | 1234 | 123 | 12 | 1 | 0 |
Add sign:              -123456
  
```

⁶⁹ `itoa`, Microsoft

⁷⁰ `itoa`, IBM

⁷¹ `itoa`, `cdiv`

⁷² `itoa`, `cplusplus`

Algoritmo:

```
static int nbr_len(int nbr)
{
    int    len;

    len = 0;
    if (nbr < 1)
        len++;
    while (nbr)
    {
        nbr /= 10;
        len++;
    }
    return (len);
}

static long long abs_val(long long n)
{
    long long    nb;

    nb = 1;
    if (n < 0)
        nb *= -n;
    else
        nb *= n;
    return (nb);
}

static char *create_str(size_t n)
{
    char    *str;

    str = (char *)malloc(sizeof(char) * (n + 1));
    if (!str)
        return (NULL);
    return (str);
}
```



```
char    *ft_itoa(int n)
{
    unsigned int    nbr;
    int             sign;
    int             len;
    char            *str;

    sign = 0;
    if (n < 0)
        sign = 1;
    len = nbr_len(n);
    str = create_str(len);
    if (!str)
        return (NULL);
    str[len] = '\0';
    nbr = abs_val(n);
    while (len--)
    {
        str[len] = '0' + nbr % 10;
        nbr /= 10;
    }
    if (sign)
        str[++len] = '-';
    return (str);
}
```

Programa:

```
int main(void)
{
    size_t    number;
    char      *str;

    number = -123456;
    str = ft_itoa(number);
    return (0);
}
```

`ft_memchr()`

Prototipo: `void *ft_memchr(const void *buffer, int c, size_t count);`

Librería: `string.h`

Parámetros:

- ***buffer:** Puntero al búfer.
- **c:** Carácter que va a buscar.
- **count:** Número de caracteres que se comprobarán.

Definición: Localiza la primera aparición del carácter `int` (convertido a `unsigned char`) en los primeros `count` caracteres. Cada carácter se interpreta como `unsigned char` del objeto apuntado por `*buffer`. Se detiene cuando encuentra el primer `c` o cuando ha terminado de comprobar su existencia mediante `count`.

Limitaciones: N/A.

Valores devueltos:

- ***pointer:** Retorna un puntero con la posición de `c` en `*buffer`.
- **NULL:** En caso de no encontrar nada, retorna `NULL`.

Otras funciones: `wmemchr`

Referencias: ⁷³ ⁷⁴ ⁷⁵ ⁷⁶

Algoritmo:

```
void *ft_memchr(const void *buffer, int c, size_t count)
{
    while (count-- > 0)
    {
        if (*(unsigned char *)buffer == (unsigned char)c)
            return ((void *)buffer);
        buffer++;
    }
    return (NULL);
}
```

Programa:

```
int main(void)
{
    char str[] = "abcdefghgabcdefg";
    char *ptr;
    char *mycharacter;

    ptr = str;
    mycharacter = ft_memchr(ptr, 'e', sizeof(str));
    return (0);
}
```

⁷³ Función `memchr` ANSI C, Con Clase

⁷⁴ `memchr()` — Search Buffer, IBM

⁷⁵ `memchr`, Microsoft

⁷⁶ C library function - `memchr()`

ft_memcmp()

Prototipo: `int ft_memcmp(const void *buf1, const void *buf2, size_t count);`

Librería: `string.h`

Parámetros:

- ***buf1:** Primer buffer.
- ***buf2:** Segundo buffer.
- **count:** Número de caracteres que se compararán.

Definición: Compara los primeros caracteres `count` de ambos búferes devolviendo un valor que indica la relación entre ellos. Los valores deben interpretarse como `unsigned char`.

Limitaciones: N/A.

Valores devueltos:

- **< 0:** *buf1 es menor que *buf2.
- **0:** *buf1 es idéntica a *buf2.
- **> 0:** *buf1 es mayor que *buf2.

Otras funciones: `wmemcmp`

Referencias: ⁷⁷ ⁷⁸ ⁷⁹ ⁸⁰

Algoritmo:

```
int ft_memcmp(const void *buf1, const void *buf2, size_t count)
{
    while (count--)
    {
        if (*(unsigned char *)buf1 != *(unsigned char *)buf2)
            return (*(unsigned char *)buf1 - *(unsigned char *)buf2);
        buf1 = (unsigned char *)buf1 + 1;
        buf2 = (unsigned char *)buf2 + 1;
    }
    return (0);
}
```

Programa:

```
int main(void)
{
    char str1[] = "abcdf";
    char str2[] = "abcde";
    char *ptr1;
    char *ptr2;

    ptr1 = str1;
    ptr2 = str2;
    printf("Compare: %d", ft_memcmp(str1, str2, 5));
    return (0);
}
```

⁷⁷ Función `memcmp` ANSI C, Con Clase

⁷⁸ `memcmp()` — Compare Buffers, IBM

⁷⁹ `memcmp`, Microsoft

⁸⁰ C library function - `memcmp()`

ft_memcpy()

Prototipo: `void *ft_memcpy(void *dest, const void *src, size_t count);`

Librería: `string.h`

Parámetros:

- ***dest:** Nuevo búfer.
- ***src:** Búfer del que copiar.
- **count:** Número de caracteres que se van a copiar.

Definición: Copia un determinado número `count` de caracteres de `*src` a `*dest`.

Limitaciones: Es necesario que el `*dest` \geq `*src`.

Valores devueltos:

- ***pointer:** Retorna el puntero destino con los valores copiados.
- **Comportamiento indefinido:** Cuando las regiones origen y destino se superponen.

Otras funciones: `wmemcpy`

Referencias: ⁸¹ ⁸² ⁸³

Algoritmo:

```
void *ft_memcpy(void *dest, const void *src, size_t count)
{
    void *ptr_dest;

    ptr_dest = dest;
    if (!dest && !src)
        return (dest);
    while (count--)
        *(char *)dest++ = *(char *)src++;
    return (ptr_dest);
}
```

⁸¹ `memcpy()` — Coopy Buffer, IBM

⁸² `memcpy`, Microsoft

⁸³ C library function - `memcpy()`

ft_memmove()

Prototipo: void *ft_memmove(void *dest, const void *src, size_t count);

Librería: string.h

Parámetros:

- ***dest:** Objeto destino.
- ***src:** Objeto de origen.
- **count:** Número de bytes que se copiarán.

Definición: Copia un determinado número `count` de caracteres de `*src` a `*dest`. En caso de que los bytes de origen se superpongan con los de destino, se garantiza que serán copiados antes de sobrescribirse.

Limitaciones: Es necesario que la longitud de `*dest` \geq `*src`.

Valores devueltos:

- ***pointer:** Retorna el puntero destino con los valores copiados.

Otras funciones: wmemmove

Referencias: ⁸⁴ ⁸⁵ ⁸⁶ ⁸⁷

Algoritmo:

```
void *ft_memmove(void *dest, const void *src, size_t count)
{
    void *ptr_dest;

    ptr_dest = dest;
    if (!dest && !src)
        return (dest);
    if (dest == src)
        return (dest);
    if (dest > src)
    {
        while (count--)
            ((char *)dest)[count] = ((char *)src)[count];
    }
    else
    {
        while (count--)
            *(char *)dest++ = *(char *)src++;
    }
    return (ptr_dest);
}
```

⁸⁴ memmove() — Coopy Bytes, IBM

⁸⁵ memmove, Microsoft

⁸⁶ Función memmove ANSI C

⁸⁷ C library function - memmove()

ft_memset()

Prototipo: `void *ft_memset(void *dest, int c, size_t count);`

Librería: `string.h`

Parámetros:

- ***dest:** Puntero destino.
- **c:** Carácter que se establecerá.
- **n:** Número de caracteres.

Definición: Establece los primeros `n` caracteres de `*dest` con el valor `c`.

Limitaciones: Es necesario que la longitud de `*dest` $\geq n$.

Valores devueltos:

- ***pointer:** Retorna el puntero destino con los nuevos valores cambiados.

Otras funciones: `wmemset`

Referencias: ^{88 89 90}

Algoritmo:

```
void *ft_memset(void *dest, int c, size_t count)
{
    void *ptr_dest;

    ptr_dest = dest;
    while (count--)
        *(unsigned char *)dest++ = (unsigned char)c;
    return (ptr_dest);
}
```

⁸⁸ `memset()` — Establecer bytes en valor, IBM

⁸⁹ `memset`, Microsoft

⁹⁰ Función `memset` ANSI C

ft_putchar_fd()

Prototipo: `void ft_putchar_fd(char c, int fd);`

Librería: N/A

Parámetros:

- **c:** Valor `ASCII` que debe representar.
- **fd:** Descriptor de archivo. Su valor determinará una entrada, salida o error.

Definición: Escribe el carácter especificado del parámetro `c`. Esta función se identifica en base a las características de 42. La función original definida por Microsoft es `putchar` y se encuentra en la librería `stdio.h`.

Limitaciones: N/A.

Valores devueltos: Devuelve el carácter escrito.

Observaciones: Descriptor de archivos (file descriptor). Devuelve `EOF` al llegar al final del archivo o flujo de datos y no existir más datos (`putchar`).

Otras funciones: `putchar`, `putwchar`

Referencias: ⁹¹ ⁹² ⁹³

Algoritmo:

```
void ft_putchar_fd(char c, int fd)
{
    write (fd, &c, 1);
}
```

⁹¹ `Putchar()` – write a character

⁹² `putchar`, Microsoft

⁹³ `EOF`, Wikipedia

`ft_putendl_fd()` / `endl` – C++

Prototipo: `void ft_putendl_fd(char *str, int fd);`

Librería: `iostream`.

Namespace: `std`.

Parámetros:

- ***str:** Puntero a la cadena `char`.
- **fd:** Descriptor de archivo. Su valor determinará una entrada, salida o error.

Definición: Inserta un salto de línea al final del contenido de una cadena `char`. Esta función se utiliza en C++ como `endl` y determina su `file descriptor` según la función de la biblioteca `iostream` utilizada (`cin/cout`). Comparativa entre la función `ft_putendl_fd` y `endl`:

C	C++
<code>ft_putendl_fd(s, 0)</code>	<code>cin << s <<std::endl</code>
<code>ft_putendl_fd(s, 1)</code>	<code>cout << s <<std::endl</code>

Además, `endl` implica un `fflush`, así que deducimos que cada vez que se utilice `ft_putendl_fd`, con valor `file descriptor = 1`, deberemos tener en cuenta el uso de `fflush`.

Limitaciones: N/A.

Valores devueltos según el file descriptor indicado:

- Contenido de una cadena `char` seguido de un salto de línea (`STDOUT_FILENO`).
- Contenido de una cadena `char` seguido de un salto de línea (`STDIN_FILENO`).
- "Error: File not found", seguido de un salto de línea (`STDERR_FILENO`).

Otras funciones: N/A.

Referencias: ⁹⁴ ⁹⁵ ⁹⁶ ⁹⁷ ⁹⁸

Algoritmo:

```
void ft_putendl_fd(char *str, int fd)
{
    if (str)
    {
        write(fd, str, ft_strlen(str));
        write(fd, "\n", 1);
    }
}
```

⁹⁴ Part II: `ft_putendl_fd`, NSHAID

⁹⁵ funciones `<ostream>`, Microsoft

⁹⁶ `std::endl`, `cplusplus` `memset` ANSI C

⁹⁷ Programación Avanzada, Miguel A. Rodríguez Florido

⁹⁸ ¿para qué sirve flush en c++?, [stackoverflow](#)

Programa:

```
#include <stdio.h>
#include <unistd.h>
int  main(void)
{
    char  str[] = "Hello World!";
    ft_putendl_fd(str, STDOUT_FILENO);
    printf("%s", str);
    return (0);
}
```

ft_putnbr_fd()

Prototipo: void ft_putnbr_fd(int n, int fd)

Librería: N/A.

Parámetros:

- **n:** Número entero de tipo `int`.
- **fd:** Descriptor de archivo. Su valor determinará una entrada, salida o error.

Definición: Función exclusiva de 42 que muestra en pantalla un número de tipo `int` pasado por parámetro.

Limitaciones: N/A

Valores devueltos: Valor de tipo `int`. Cambiando la secuencia de `*base`, conseguimos números hexadecimales.

Otras funciones: N/A.

Referencias: ⁹⁹ ¹⁰⁰

Notas sobre el cálculo: `c = *(base + (size_t)nb % base_len);`

Output:

```
ASCII(1) - 49 = *(48 + 1 mod 10)
ASCII(5) - 53 = *(48 + 5 mod 10)
ASCII(6) - 54 = *(48 + 6 mod 10)
ASCII(7) - 55 = *(48 + 7 mod 10)
```

Algoritmo:

```
static void putnbr_base_fd(int nbr, const char *base, int fd)
{
    int          base_len;
    long long int nb;
    char         c;

    base_len = ft_strlen(base);
    if (nbr < 0)
    {
        ft_putchar_fd('-', fd);
        nb = -((long long int)nbr);
    }
    else
        nb = (long long int)nbr;
    if (nb < base_len)
    {
        c = base[(size_t)nb % base_len];
        ft_putchar_fd(c, fd);
    }
    else
    {
        putnbr_base_fd(nb / base_len, base, fd);
        putnbr_base_fd(nb % base_len, base, fd);
    }
}
```

⁹⁹ C: Create a function to display number just with "write" function, Stackoverflow

¹⁰⁰ C++ Why does LLONG_MIN == -LLONG_MIN, Stackoverflow

```
void ft_putnbr_fd(int n, int fd)
{
    const char    *base;
    long long int  l;

    base = "0123456789";
    l = (long long int)n;
    putnbr_base_fd(l, base, fd);
}
```

Programa:

```
#include <limits.h>
#include <stdio.h>
#include <unistd.h>

int main(void)
{
    int    n;
    int    fd;

    n = 1567;
    fd = STDOUT_FILENO;
    ft_putnbr_fd(n, fd);
    return (0);
}
```

ft_putstr_fd() / puts() – C++

Prototipo: void ft_putstr_fd(char *str, int fd);

Librería: stdio.h

Parámetros:

- ***str:** Puntero que apunta a la cadena de caracteres.
- **fd:** Descriptor de archivo. Su valor determinará una entrada, salida o error.

Definición: Función creada para 42. Muestra uno a uno, en pantalla, los caracteres de una cadena de caracteres. En contrapartida, existe la función `puts()` que devuelve resultado de tipo `int`, y escribe una secuencia de caracteres al flujo de salida estándar.

Limitaciones: La función no certifica que todo haya sido correcto.

Valores devueltos:

- **N/A:** No devuelve ningún valor.
- **0:** Si la función se realiza correctamente (`*puts()`).
- **<>0:** Valor distinto a cero en caso de algún error (`*puts()`).

Otras funciones: `gets`

Referencias: ¹⁰¹

Algoritmo:

```
void    ft_putstr_fd(char *str, int fd)
{
    if (str)
    {
        write(fd, str, ft_strlen(str));
    }
}
```

¹⁰¹ 8.3.2. La función `puts`, Arquitectura de Sistemas UC3M

ft_split() / strtok – C++**Prototipo:** `char **ft_split(const char *str, char c);`**Librería:** `stdio.h`**Parámetros:**

- ***str:** Puntero que indica la cadena de caracteres a dividir.
- **c:** Carácter que se utilizará para dividir la cadena apuntada por *s.

Definición: Función creada para 42. Divide una cadena de caracteres según el carácter indicado en el parámetro c. Esta función se basa en `*strtok` de C++.

Limitaciones: La línea marcada en la función `**Split` puede dar errores de compilación: `few arguments`. En estos casos utilizar la versión antigua:

- `get_word_len(&(s + j), c)`

Valores devueltos: Retorna un array donde cada uno de sus elementos contiene la dirección a cada nueva cadena de caracteres. El último elemento del array contiene el valor 0 que indica su final.

Otras funciones: `strtok`.**Referencias:** ¹⁰² ¹⁰³ ¹⁰⁴ ¹⁰⁵**Algoritmo:**

```
static size_t    count_words(char const *str, char c)
{
    size_t    count;
    size_t    i;

    count = 0;
    i = 0;
    while (str[i])
    {
        if (str[i] != c)
        {
            count++;
            while(str[i] && str[i] != c)
                i++;
        }
        else if (str[i] == c)
            i++;
    }
    return (count);
}
```

¹⁰² Split a string into tokens – strtok, codingame.com¹⁰³ Función strtok ANSI C, conclase.net¹⁰⁴ strtok ()- Tokenizar serie, IBM¹⁰⁵ Splitting a string using strtok() in C

```
static size_t  get_word_len(const char *str, char c)
{
    size_t  i;

    i = 0;
    while (str[i] && str[i] != c)
        i++;
    return (i);
}

static void free_array(size_t i, char **array)
{
    while (i > 0)
    {
        i--;
        free(array[i]);
    }
    free(array);
}

static char **split(const char *s, char c, char **array, size_t count_words)
{
    size_t  i;
    size_t  j;

    i = 0;
    j = 0;
    while (i < count_words)
    {
        while (s[j] && s[j] == c)
            j++;
        array[i] = ft_substr(s, j, get_word_len(&s[j]), c);
        if (!array[i])
        {
            free_array(i, array);
            return (NULL);
        }
        while (s[j] && s[j] != c)
            j++;
        i++;
    }
    array[i] = NULL;
    return (array);
}

char **ft_split(const char *str, char c)
{
    char  **array;
    size_t  words;

    if (!str)
        return (NULL);
    words = count_words(str, c);
    array = (char **)malloc(sizeof(char *) * (words + 1));
    if (!array)
        return (NULL);
    array = split(str, c, array, words);
    return (array);
}
```

Programa:

```
int    main(void)
{
    int    i;
    const char *str1 = "My,Name,is,Guybrush";
    char **split_array1 = ft_split(str1, ',');

    i = 0;
    printf("Phrase:\t%s\n", str1);
    while(split_array1[i] != NULL)
    {
        printf("\t%s\n", split_array1[i]);
        i++;
    }
    printf("Amount:\t%d", i);
    return (0);
}
```

Output:

```
Phrase: My,Name,is,Guybrush
My
Name
is
Guybrush
Amount: 4
```

ft_strchr()

Prototipo: `char *ft_strchr(const char *str, int c);`

Librería: `string.h`

Parámetros:

- ***str:** Cadena de origen terminada en `NULL`.
- **c:** Carácter que se buscará.

Definición: Busca la primera aparición del carácter `c` en el puntero `*str`. El carácter `c` también puede ser `'\0'`. El final nulo de una `string` también puede incluirse en la búsqueda. En caso de no encontrarlo, devuelve `NULL`.

Limitaciones: El valor de salida puede verse afectado por el valor de `LC_TYPE` de la configuración regional.

Valores devueltos:

- ***str:** Retorna el puntero en la primera posición de `c`.
- **NULL:** No encuentra `c`.

Otras funciones: `wcschr`, `_mbschr`, `_mbschr_l`

Referencias: ¹⁰⁶ ¹⁰⁷ ¹⁰⁸

Algoritmo:

```
char *ft_strchr(const char *str, int c)
{
    while (*str)
    {
        if (*str == (char)c)
            return ((char *)str);
        str++;
    }
    if (*str == (char)c)
        return ((char *)str);
    else
        return (NULL);
}
```

¹⁰⁶ `strchr`, Microsoft

¹⁰⁷ `strchr`, Aprende a programar

¹⁰⁸ `strchr()` — Search for Character, IBM

ft_strdup()

Prototipo: `char *ft_strdup(const char *str);`

Librería: `string.h`

Parámetros:

- ***str:** Cadena de origen terminada en `NULL`.

Definición: Reserva un espacio de memoria mediante `malloc` para realizar una copia de la cadena `*str`. La cadena origen debe tener necesariamente el carácter `'\0'` que la finalice. Debe utilizarse la función `free` para liberar la memoria.

Limitaciones: En desuso. Generan la advertencia del compilador (nivel 3) `C4996`.

Valores devueltos:

- ***pointer:** Retorna el puntero que contiene la cadena duplicada.
- **NULL:** Cuando hay insuficiente memoria.

Otras funciones: `wcsdup`

Referencias: ¹⁰⁹ ¹¹⁰ ¹¹¹ ¹¹²

Algoritmo:

```
static char *create_str(size_t n)
{
    char *str;

    str = (char *)malloc(sizeof(char) * (n + 1));
    if (!str)
        return (NULL);
    return (str);
}

char *ft_strdup(const char *str)
{
    char *dest;
    char *start;

    dest = create_str(ft_strlen(str));
    if (!dest)
        return (NULL);
    start = dest;
    while (*str)
        *dest++ = *str++;
    *dest = '\0';
    return (start);
};
```

¹⁰⁹ `strdup`, [geeksforgeeks](#)

¹¹⁰ `strdup`, [ArchLinux](#)

¹¹¹ `strdup()` — Duplicate String, IBM

¹¹² `strdup`, Microsoft

ft_striteri()

Prototipo: void ft_striteri(char *s, void (*f)(unsigned int, char *));

Librería: N/A.

Parámetros:

- ***s:** Puntero que apunta a la cadena que va a ser iterada.
- **void (*f)(unsigned int, char *):** Función compuesta de dos parámetros que se aplicará a cada carácter de la cadena.

Definición: Función creada para 42. Toma un puntero que apunta a una cadena y aplica una función de unas características específicas a cada carácter de la cadena.

Limitaciones: N/A.

Valores devueltos: Ninguno.

Otras funciones: N/A.

Algoritmo:

```
void    ft_striteri(char *s, void (*f)(unsigned int, char *))
{
    size_t  i;

    if (s && f)
    {
        i = 0;
        while (*s)
            f(i++, s++);
    }
}
```

Ejemplo:

```
void uppercase(unsigned int index, char *ch)
{
    *ch = ft_toupper(*ch);
}

char str[] = "hello";
ft_striteri(str, uppercase);
```

ft_strjoin()

Prototipo: `char *ft_strjoin(char const *s1, char const *s2);`

Librería: N/A.

Parámetros:

- ***s1:** Cadena de caracteres.
- ***s2:** Cadena de caracteres.

Definición: Une dos cadenas de caracteres.

Limitaciones: La cadena resultante debe tener un tamaño $\geq \text{strlen}(s1) + \text{strlen}(s2) + 1$.

Valores devueltos: Puntero que apunta a una nueva cadena de caracteres donde se han unido *s1 y *s2. Debe tener una terminación en `'\0'`.

Otras funciones: N/A.

Algoritmo:

```
static char      *create_str(size_t n)
{
    char      *str;

    str = (char *)malloc(sizeof(char) * (n + 1));
    if (!str)
        return (NULL);
    return (str);
}

char      *ft_strjoin(char const *s1, char const *s2)
{
    char      *str;
    char      *ptr_str;

    if (!s1 || !s2)
        return (NULL);
    str = create_str(ft_strlen(s1) + ft_strlen(s2));
    if (!str)
        return (NULL);
    ptr_str = str;
    while (*s1)
        *str++ = *s1++;
    while (*s2)
        *str++ = *s2++;
    *str = '\0';
    return (ptr_str);
}
```

ft_strlcat()

Prototipo: `size_t ft_strlcat(char *dest, const char *src, size_t n);`

Librería: `string.h`

Parámetros:

- ***dest:** Puntero que va a contener la concatenación terminada en `NULL`.
- ***src:** Cadena terminada en `NULL` que se va a concatenar a `*dest`.
- **n:** Valor máximo permitido de `*dest` que será `strlen(src) + strlen(dest) - 1`.

Definición: Copia y concatena cadenas. Agrega la cadena `*src` terminada en `NULL` al final de `*dest`, en otras palabras, toma el tamaño completo de `*dest` (no solo la longitud) y garantiza la terminación `NULL` del resultado, siempre que quede al menos un byte libre en `*dest`. Únicamente se copiará/concatenará el espacio disponible.

Limitaciones: La cadena `*src` debe ser una cadena “C” verdadera, es decir, debe terminar en `NULL`.

Valores devueltos:

- **t_size:** longitud total de la concatenación `strlen(src) + strlen(dest) - 1`.
- **Compartimento indefinido:** Cuando `*src` y `*dest` se superponen.

Otras funciones: `snprintf`, `strncat`, `strncpy`

Referencias: ¹¹³ ¹¹⁴ ¹¹⁵

Algoritmo:

```
size_t      ft_strlcat(char *dest, const char *src, size_t n)
{
    const char    *s;
    size_t        dest_len;
    size_t        total_len;

    if ((!dest || !src) && !n)
        return (0);
    s = src;
    dest_len = 0;
    while (*(dest + dest_len) && dest_len < n)
        dest_len++;
    if (dest_len < n)
        total_len = dest_len + ft_strlen(s);
    else
        return (n + ft_strlen(s));
    while (*s && (dest_len + 1) < n)
    {
        *(dest + dest_len) = *s++;
        dest_len++;
    }
    *(dest + dest_len) = '\0';
    return (total_len);
}
```

¹¹³ `strlcat`, mksoftware

¹¹⁴ `strlcat`, GLIB

¹¹⁵ `strlcat`, Linux man page

ft_strlcpy()

Prototipo: `size_t ft_strlcpy(char *dest, const char *src, size_t n);`

Librería: `string.h`

Parámetros:

- ***dest:** Puntero que va a contener la copia de `*src` terminada en `NULL`.
- ***src:** Cadena terminada en `NULL` que se va a copiar en `*dest`.
- **n:** Valor máximo permitido será `strlen(src) + strlen(dest) - 1`.

Definición: Agrega la cadena `*src` a `*dest`. Toma el tamaño completo de `*dest` (no solo la longitud) y garantiza la terminación `NULL` de `*dest`, siempre que quede al menos un byte libre en `*dest`.

Limitaciones: La cadena `*src` debe ser una cadena “C” verdadera, es decir, debe terminar en `NULL`.

Valores devueltos:

- **t_size:** longitud total de `*src`.
- **Compartimento indefinido:** Cuando `*src` y `*dest` se superponen.

Otras funciones: `snprintf`, `strncat`, `strncpy`

Algoritmo:

```
size_t      ft_strlcpy(char *dest, const char *src, size_t n)
{
    size_t    i;

    i = 0;
    while (*(src + i))
        i++;
    if (!n)
        return (i);
    while (--n && *src)
        *dest++ = *src++;
    *dest = '\0';
    return (i);
}
```

ft_strlen()

Prototipo: `size_t ft_strlen(const char *str);`

Librería: `string.h`

Parámetros:

- ***str:** Cadena de origen terminada en `NULL`.

Definición: Obtiene la longitud de una cadena usando la configuración regional actual o una configuración regional especificada. El valor devuelto será siempre igual al número de bytes, incluso si la cadena contiene caracteres multibyte.

Valores devueltos:

- **size_t:** Número de caracteres de `*str` sin incluir el carácter `'\0'` de terminación.

Limitaciones: Representa una posible amenaza por un problema de saturación del búfer. Estos problemas producen una elevación de privilegios no justificada. Consultar [Evitar saturaciones de búfer](#).

Otras funciones: `wcslen`, `_mbslen`, `_mbslen_l`, `_mbstrlen`, `_mbstrlen_l`

Algoritmo:

```
size_t      ft_strlen(const char *str)
{
    size_t    i;

    i = 0;
    while (str[i])
        i++;
    return (i);
}
```

Equivalente:

```
size_t      ft_strlen(const char *str)
{
    size_t    i;

    i = 0;
    while (*(str + i))
        i++;
    return (i);
}
```

ft_strmapi()

Prototipo: char *ft_strmapi(const char *s, char (*f)(unsigned int, char));

Librería: N/A.

Parámetros:

- ***s:** Puntero que apunta a la cadena que va a ser iterada.
- **char (*f)(unsigned int, char):** Función compuesta de dos parámetros que se aplicará a cada carácter de la cadena.

Definición: Función creada para 42. Toma un puntero que apunta a una cadena y aplica una función de unas características específicas a cada carácter de la cadena.

Limitaciones: N/A.

Valores devueltos: Nueva cadena de caracteres con la función (*f) aplicada.

Otras funciones: N/A.

Algoritmo:

```
static char      *create_str(size_t n)
{
    char      *str;

    str = (char *)malloc(sizeof(char) * (n + 1));
    if (!str)
        return (NULL);
    return (str);
}

char      *ft_strmapi(const char *s, char (*f)(unsigned int, char))
{
    size_t  i;
    char      *str;
    char      *result;

    if (!s)
        return (NULL);
    str = create_str(ft_strlen(s));
    if (!str)
        return (NULL);
    i = 0;
    result = str;
    while (*s)
        *str++ = f(i++, *s++);
    *str = '\0';
    return (result);
}
```

Programa:

Para que el siguiente código funcione, hay que añadir dos parámetros más a `*ft_strmapi` y a la función a la que apunta, quedando de la siguiente manera:

```
char *ft_strmapi(const char *s, char (*f)(unsigned int, char, int, int), int key, int keynum);
```

Aplicar también esta modificación en la correspondiente línea de la función:

- `*str++ = f(i++, *s++, key, keynum);`

```
static char CypherNum(char c, int keynum)
{
    if (keynum < 0 || keynum > 9)
        keynum = 0;
    c = (unsigned char)c + keynum;
    if (c < '0' || c > '9')
        c = ('0' - 1) + (c - '9');
    return (c);
}

static char Cypher(unsigned int i, char c, int key, int keyNum)
{
    if (key < 0 || key > 26)
        key = 0;

    if (ft_isdigit(c))
        return (CypherNum(c, keyNum));

    if (ft_isalpha(c))
        c = ft_tolower(c);

    if ((c < 'a' || c > 'z') || c == ' ')
        return (c);

    c = (unsigned char)c + key;
    if (c < 'a' || c > 'z')
        c = ('a' - 1) + (c - 'z');
    return (c);
}

int main(void)
{
    char str[] = "nceraqb pbfvpnf ra 97";
    char *ptr;
    int key;
    int keynum;

    key = 13;           //Clave cifrado letras minúsculas
    keynum = 5;         //Clave cifrado números
    ptr = str;
    printf("Original:\t%s\n", ptr);
    printf("Cypher:\t\t%s\n", ft_strmapi(ptr, Cypher, key, keynum));
    return (0);
}
```

Output:

```
Original:      nceraqb pbfvpnf ra 97
Cypher:       Aprendo cosicas en 42
```


ft_strncmp()

Prototipo: `int ft_strncmp(const char *s1, const char *s2, size_t n);`

Librería: `string.h`

Parámetros:

- ***s1:** Primera cadena.
- ***s2:** Segunda cadena.
- **n:** Número de caracteres que se compararán.

Definición: Compara los primeros caracteres `n` de ambas cadenas devolviendo un valor que indica la relación entre ellos. Los valores deben interpretarse como `unsigned char`.

Limitaciones: N/A.

Valores devueltos:

- **< 0:** `*s1` es menor que `*s2`.
- **0:** `*s1` es idéntica a `*s2`.
- **> 0:** `*s1` es mayor que `*s2`.

Otras funciones: `wcsncmp`, `_mbsncmp`, `_mbsncmp_l`

Referencias: ¹¹⁶ ¹¹⁷

Algoritmo:

```
int ft_strncmp(const char *s1, const char *s2, size_t n)
{
    if (n == 0)
        return (0);
    while ((*s1 && *s2) && (*s1 == *s2) && n > 1)
    {
        s1++;
        s2++;
        n--;
    }
    return (((unsigned char)(*s1) - (unsigned char)(*s2)));
}
```

¹¹⁶ `strncmp()` — Compare Strings, IBM

¹¹⁷ `strncmp()`, Microsoft

Notas sobre el cálculo: `while ((*s1 && *s2) && (*s1 == *s2) && n > 1)`

La variable `n > 1` es debido a que primero movemos las posiciones de los punteros y luego restamos la posición de `n`:

str1	0	1	2	3	4	5
	'a'	'b'	'c'	'd'	'e'	'\0'
str2	'a'	'b'	'c'	'd'	'e'	'\0'
s1						
s2						
N	2					

	0	1	2	3	4	5
str1	'a'	'b'	'c'	'd'	'e'	'\0'
str2	'a'	'b'	'c'	'd'	'e'	'\0'
s1	++					
s2	++					
n	2 <<!!					

Programa:

```
int main(void)
{
    char str1[] = "abcdefg";
    char str2[] = "abcdefg";
    printf("%d\n", ft_strncmp(str1, str2, 4));
    return (0);
}
```

ft_strnstr()

Prototipo: `char *ft_strnstr(const char *big, const char *little, size_t n);`

Librería: `string.h`

Parámetros:

- ***big:** Primera cadena.
- ***little:** Segunda cadena.
- **n:** Número de caracteres que se compararán.

Definición: Busca la secuencia de caracteres contenida en la subcadena en una cadena de texto. Determina si la cadena `*big` contiene `*little`, indicando su posición en caso de encontrarla. En caso de encontrar varias, retornará la posición de la primera coincidencia. Los valores deben interpretarse como `char`.

Limitaciones: N/A.

Valores devueltos:

- ***big:** En caso de que no se encuentre `*little`.
- **+n:** Posición de la cadena de texto encontrada.
- **NULL:** Si no se encuentra la `*little` dentro de `*big`.

Otras funciones: N/A.

Algoritmo:

```
char *ft_strnstr(const char *big, const char *little, size_t n)
{
    size_t    j;

    if (!big && !n)
        return (NULL);
    if (!*little)
        return ((char *)big);
    while (*big && n--)
    {
        j = 0;
        while ((big[j] == little[j])
                && little[j] && j <= n)
        {
            if (!little[j + 1])
                return ((char *)big);
            j++;
        }
        big++;
    }
    return (NULL);
}
```

ft_strrchr()

Prototipo: `char *ft_strrchr(const char *str, int c);`

Librería: `string.h`

Parámetros:

- ***str:** Cadena originaria donde buscar.
- **c:** Valor que se quiere encontrar.

Definición: Busca la última aparición de `c` en serie. Los valores deben interpretarse como `char`. El carácter nulo final se considera parte de la serie.

Limitaciones: El comportamiento de esta función puede verse afectado por la categoría `LC_CTYPE` del entorno actual.

Valores devueltos:

- ***puntero:** Puntero a la última posición de la aparición de `c`.
- **NULL:** En caso de no encontrar `c`.

Otras funciones: N/A.

Algoritmo:

```
char *ft_strrchr(const char *str, int c)
{
    const char *aux;

    aux = NULL;
    while (*str)
    {
        if (*str == (char)c)
            aux = str;
        str++;
    }
    if (*str == (char)c)
        return ((char *)str);
    else
        return ((char *)aux);
}
```

ft_strtrim()

Prototipo: `char *ft_strtrim(const char *s1, const char *set);`

Librería: `publib.h`

Parámetros:

- ***s1:** Puntero que apunta a la cadena de caracteres.
- **const char *set:** Carácter a eliminar.

Definición: Función creada para 42. Elimina el carácter especificado por parámetro `*set` que se encuentren en el principio y en el final de la cadena de caracteres apuntada por `*s1`. Proviene de la función original `strtrim()` que elimina los espacios en blanco. Incorpora `ft_strlen()`.

Limitaciones: N/A.

Valores devueltos: Nueva cadena de caracteres con la función `(*f)` aplicada.

Otras funciones: `strtrim`.

Referencias: ^{118 119}

Algoritmo:

```
static size_t check_char(const char *str, const char c)
{
    size_t i;

    if (!str)
        return (0);
    i = 0;
    while (str[i])
    {
        if (str[i] == c)
            return (1);
        i++;
    }
    return (0);
}

static char *create_str(size_t n)
{
    char *str;

    str = (char *)malloc(sizeof(char) * (n + 1));
    if (!str)
        return (NULL);
    return (str);
}
```

¹¹⁸ Implementation of the C Function `strtrim()`, jraleman, Medium

¹¹⁹ `strtrim(3pub)` [debian man page]

```

char    *ft_strtrim(const char *s1, const char *set)
{
    char    *trim;
    size_t  start;
    size_t  end;
    size_t  i;

    if (!s1 || !set)
        return (NULL);
    start = 0;
    while (s1[start] && check_char(set, s1[start]))
        start++;
    end = ft_strlen(s1);
    while (end > start && check_char(set, s1[end - 1]))
        end--;
    trim = create_str(end - start);
    if (!trim)
        return (NULL);
    i = 0;
    while ((start + i) < end)
    {
        trim[i] = s1[start + i];
        i++;
    }
    trim[i] = '\0';
    return (trim);
}

```

Programa:

```

#include <stdlib.h>
#include <stdio.h>
int  main(void)
{
    char  str[] = "--Hello World--";
    char  *ptr;
    char  *result;

    ptr = str;
    result = ft_strtrim(ptr, "--");
    return (0);
}

```

ft_substr() / substr – C++

Prototipo: `char *ft_substr(char const *s, unsigned int start, size_t len);`

Librería: `stdlib.h`

Parámetros:

- ***s:** Puntero que apunta a la cadena de caracteres.
- **start:** Inicio de la posición de `*s` para iniciar la subcadena.
- **len:** Longitud desde la posición `start` de la subcadena.

Definición: Retorna una nueva cadena de caracteres de longitud `len` dentro de la la subcadena `*s`. La primera posición empieza desde 0. Proviene de la función `substr` de C++.

Limitaciones: N/A.

Valores devueltos:

- **Subcadena:** Posición del primer carácter de la subcadena resultante.
- **Cadena vacía:** En caso de que la cadena inicial y la subcadena tengan la misma longitud.
- **start > len:** Out of range.

Otras funciones: `substr`.

Algoritmo:

```
static char    *create_str(size_t n)
{
    char    *str;

    str = (char *)malloc(sizeof(char) * (n + 1));
    if (!str)
        return (NULL);
    return (str);
}
char    *ft_substr(char const *s, unsigned int start, size_t len)
{
    char    *str;
    char    *ptr_str;

    if (!s)
        return (NULL);
    if (start > ft_strlen(s))
        len = 0;
    else if (len > (ft_strlen(s) - start))
        len = ft_strlen(s) - start;
    str = create_str(len);
    if (!str)
        return (NULL);
    s += start;
    ptr_str = str;
    str[len] = '\0';
    while (len-- && *s)
        *str++ = *s++;
    return (ptr_str);
}
```

`ft_tolower()`

Prototipo: `int ft_tolower(int c);`

Librería: `ctype.h`

Parámetros:

- **c:** Valor que se quiere convertir a minúscula.

Definición: La función `tolower()` convierte la letra mayúscula `c` en la letra minúscula correspondiente. Devuelve el carácter convertido. Si el carácter `c` no tiene un carácter en mayúsculas correspondientes, las funciones devuelven `c` sin modificar.

Limitaciones: El comportamiento de esta función puede verse afectado por la categoría `LC_CTYPE` del entorno actual.

Valores devueltos:

- **int:** Si `c` está entre 'A' y 'Z' retorna el valor convertido.
- **int:** En caso de no encontrar `c`, retorna el valor sin conversión.

Otras funciones: N/A.

Algoritmo:

```
int ft_tolower(int c)
{
    if (c >= 'A' && c <= 'Z')
        c += 32;
    return (c);
}
```


ft_toupper()

Prototipo: `int ft_toupper(int c);`

Librería: `ctype.h`

Parámetros:

- **c:** Valor que se quiere convertir a mayúscula.

Definición: La función `toupper()` convierte la letra minúscula `c` en la letra mayúscula correspondiente. Devuelve el carácter convertido. Si el carácter `c` no tiene un carácter en minúsculas correspondientes, las funciones devuelven `c` sin modificar.

Limitaciones: El comportamiento de esta función puede verse afectado por la categoría `LC_CTYPE` del entorno actual.

Valores devueltos:

- **int:** Si `c` está entre 'a' y 'z' retorna el valor convertido.
- **int:** En caso de no encontrar `c`, retorna el valor sin conversión.

Otras funciones: N/A.

Algoritmo:

```
int ft_toupper(int c)
{
    if (c >= 'a' && c <= 'z')
        c -= 32;
    return (c);
}
```

Header o archivo de cabecera

Se denomina header file, en español fichero/archivo (de) cabecera, o include file.

Un header file contiene, normalmente, una declaración directa de clases, subrutinas, variables u otros identificadores. Aquellos programadores que desean declarar identificadores estándares en más de un archivo fuente pueden colocarlos en un único header file, que se incluirá cuando el código que contiene sea requerido por otros archivos.

La biblioteca estándar de C y la biblioteca estándar de C++ tradicionalmente declaran sus funciones estándar en header files.

Para el proyecto libft y la creación de la librería, antes es necesario entender qué son las directivas de preprocesador y algunas de las principales directivas que usaremos:

- `#include`
- `#ifdef` / `#ifndef`
- `#define`

Directivas de preprocesador: El preprocesador es un programa que invoca el compilador para procesar el código antes de la compilación. Los mandatos para ese programa, conocidos como directivas, son líneas del archivo fuente que empiezan por el carácter `#`, que los distingue de las líneas del texto del programa fuente. El efecto de cada directiva de preprocesador es un cambio en el texto del código fuente, y el resultado es un nuevo archivo de código fuente, que no contiene las directivas. El código fuente preprocesado, un archivo intermedio, debe ser un programa C o C++ válido, porque se convierte en la entrada del compilador.

Por ejemplo: La directiva `#include` sirve para insertar ficheros externos dentro de nuestro fichero de código fuente. Estos ficheros son conocidos como ficheros incluidos, ficheros de cabecera o `headers`. El preprocesador elimina la línea `#include` y, conceptualmente, la sustituye por el fichero especificado.

- `#include <stdlib.h>`
- `#include "libft.h"`

Directivas

#include: Indica al preprocesador el contenido de un archivo especificado en el punto donde aparece la directiva. El emplazamiento del `#include` puede influir sobre el ámbito y la duración de cualquiera de los identificadores en el interior del fichero incluido.

La diferencia entre escribir el nombre del fichero entre "`<>`" o "`\"`", está en el algoritmo usado para encontrar los ficheros a incluir. En el primer caso el preprocesador buscará en los directorios `include` definidos en el compilador. En el segundo, se buscará primero en el directorio actual, es decir, en el que se encuentre el fichero fuente, si no existe en ese directorio, se trabajará como el primer caso.

Si se proporciona el camino como parte del nombre de fichero, sólo se buscará en el directorio especificado, por ejemplo:

- `#include "c:\includes\header_file.h"`

Pero, ¿dónde están los directorios definidos por el compilador? Para ello debemos dirigirnos al manual de nuestro compilador. En caso de usar GCC en Linux, utilizamos la siguiente orden en la consola para que nos muestre la información requerida:

- `cpp -v`

Veremos que las rutas en las que buscará los archivos son las siguientes:

```
#include "..." search starts here:
#include <...> search starts here:
 /usr/lib/gcc/x86_64-linux-gnu/11/include
 /usr/local/include
 /usr/include/x86_64-linux-gnu
 /usr/include
```

Y si nos dirigimos a las carpetas indicadas, encontraremos las bibliotecas del sistema, aunque es cierto que algunas carpetas puedan estar vacías.

- `\usr\include\stdio.h`

#define: Crea una macro que es la asociación de un identificador o identificador parametrizado con una cadena de token. Una vez definida la macro, el compilador puede sustituir la cadena de token para cada aparición del identificador del archivo de código fuente.

- `#define IDENTIFICADOR`

#ifdef/#ifndef: La directiva `#ifndef` comprueba si se ha definido o no una macro. Si el identificador especificado no está definido como una macro, las líneas de código inmediatamente después de la condición se pasan al compilador.

Estas directivas solo comprueban la presencia o ausencia de identificadores definidos con `#define`.

libft.h

El siguiente archivo contiene todas las funciones de la Parte I y II del proyecto LIBFT. En él se puede ver como aplicamos la directiva `#ifndef`, que en caso de que no esté definida, la definimos con el nombre `LIBFT_H`. A continuación, incluimos las librerías del sistema necesarias para usar `write`, `t_size` o `UINT_MAX` (por ejemplo) y finalmente indicamos el prototipo de todas las funciones que hemos creado en archivos diferentes.

```
#ifndef LIBFT_H
# define LIBFT_H

# include <unistd.h>
# include <limits.h>
# include <stdlib.h>

int      ft_atoi(const char *str);
void     ft_bzero(void *str, size_t n);
void     *ft_calloc(size_t number, size_t size);
int      ft_isalnum(int c);
int      ft_isalpha(int c);
int      ft_isascii(int c);
int      ft_isdigit(int c);
int      ft_isprint(int arg);
char     *ft_itoa(int n);
void     *ft_memchr(const void *buffer, int c, size_t n);
int      ft_memcmp(const void *buf1, const void *buf2, size_t n);
void     *ft_memcpy(void *dest, const void *src, size_t n);
void     *ft_memmove(void *dest, const void *src, size_t count);
void     *ft_memset(void *str, int c, size_t count);
void     ft_putchar_fd(char c, int fd);
void     ft_putendl_fd(char *str, int fd);
void     ft_putnbr_fd(int n, int fd);
void     ft_putstr_fd(char *str, int fd);
char     **ft_split(const char *str, char c);
char     *ft_strchr(const char *str, int c);
char     *ft_strdup(const char *str);
void     ft_striteri(char *s, void (*f)(unsigned int, char *));
char     *ft_strjoin(char const *s1, char const *s2);
size_t   ft_strlcat(char *dest, const char *src, size_t n);
size_t   ft_strlcpy(char *dest, const char *src, size_t n);
size_t   ft_strlen(const char *str);
char     *ft_strmapi(const char *s, char (*f)(unsigned int, char));
int      ft_strncmp(const char *s1, const char *s2, size_t n);
char     *ft_strnstr(const char *big, const char *little, size_t n);
char     *ft_strrchr(const char *str, int c);
char     *ft_strtrim(const char *s1, const char *set);
char     *ft_substr(char const *s, unsigned int start, size_t len);
int      ft_tolower(int c);
int      ft_toupper(int c);

#endif
```

Compilación

Antes de desarrollar el archivo Makefile, debemos entender ciertos conceptos para compilar nuestro proyecto y hacerlo funcional. A continuación, se muestra un preámbulo sobre como desarrollaremos ciertas órdenes que va a contener nuestro archivo Makefile.

Bibliotecas de funciones: Una biblioteca de funciones es un tipo de política especial que contiene únicamente funciones definidas por el usuario. Este archivo lo hemos creado anteriormente y es `libft.h`. Por el momento no tiene más complicación que la de escribir todos los prototipos de las funciones, asegurándonos de que sus nombres están bien escritos y tienen los parámetros exactamente igual que en los archivos `*.c`.

gcc: Es el compilador que se encarga de transformar los archivos `filename.c` a código máquina. Las opciones que nos interesan son `-c` y `-o`.

- **-c (sin enlace):** indica que compile los archivos como objetos, dando un nuevo archivo con extensión `filename.o`. Los objetos se usan para realizar librerías.
- **-o (salida del fichero):** el compilador genera por defecto la salida `a.out` de los ficheros. Cuando señalamos el flag `-o`, le indicamos el nuevo nombre.

El siguiente comando da como salida un archivo ejecutable llamado `newname`:

- `gcc file1.c -o newname`

Para ejecutarlo, deberíamos hacer lo siguiente:

- `./newname`

Así pues, conociendo los flags `-c` y `-o`. Con `-c` indicamos que los archivos deben ser objetos y con `-o` indicamos los nuevos nombres de esos objetos, resultando un archivo compilado de extensión `*.o`:

- `gcc -Wall -Wextra -Werror -c -o file1.c file2.c[...] file1.o file2.o[...]`

ar: Este comando se usa para crear bibliotecas de funciones cuando desarrollamos software. Crea archivos únicos que actúan como contenedor de otros archivos. Es parecido al comando `tar`, donde puedes agregarlos, eliminarlos o extraerlos.

Uno de los usos más utilizados de `ar` es para la creación de bibliotecas estáticas y para crear los paquetes de archivos `.deb`. Antes de utilizar el comando, debemos tener nuestros archivos compilados. Las etiquetas que utilizaremos serán:

- **-c (crear):** Con esta opción crea el archivo de la biblioteca, en nuestro caso `libft.a`.
- **-r (agregar o reemplazar):** Esta opción agrega los archivos seleccionados a la biblioteca.
- **-s (indexar):** Crea un índice de los archivos dentro de la biblioteca.

El comando queda de la siguiente manera:

- `ar -crs libft.a file1.o file2.o file3.o [...]`

Notaremos que se crea el archivo `libft.a`. Para que liste los módulos que contiene nuestra nueva biblioteca; simplemente volvemos a ejecutar `ar` con el flag `-t`, aunque este comando no es necesario.

- `ar -t libft.a`

Test: Para testear la librería y su correcta vinculación, seguimos estas instrucciones (se recomienda leer primero el capítulo [Makefile](#) para compilar la librería):

Creamos un directorio nuevo llamado `\test\`.

- `mkdir test`

Copiamos los archivos `libft.h` y `libft.a` al directorio `\test\` (explicado en el capítulo [Makefile](#)).

Dentro de este, creamos un nuevo fichero llamado `test.c` y desarrollamos la siguiente función:

- `cd test`
- `vim test.c`

Este es el contenido de `test.c`. Fijémonos que escribimos un `#include libft.h` y utilizaremos una de las funciones que hemos creado. Tanto la función como la biblioteca que usa `write` (`unistd.h`) se encuentran en `libft.h`.

```
#include "libft.h"
int      main(void)
{
    char    c;

    c = (char)ft_toupper('s');
    write(1, &c, 1);
    write(1, "\n", 1);
    return (0);
}
```

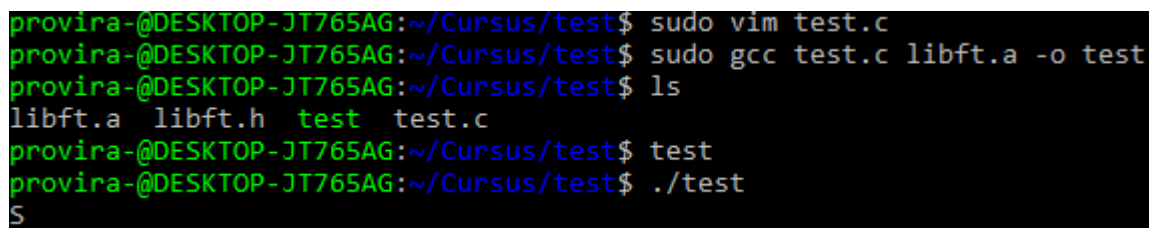
Guardamos el archivo y compilamos:

- `sudo gcc test.c libft.a -o test`

Si todo ha ido bien, ejecutamos:

- `./test`

Y el resultado es justo lo que nos hace el programa: la S mayúscula y un salto de línea.



```
provira-@DESKTOP-JT765AG:~/Cursus/test$ sudo vim test.c
provira-@DESKTOP-JT765AG:~/Cursus/test$ sudo gcc test.c libft.a -o test
provira-@DESKTOP-JT765AG:~/Cursus/test$ ls
libft.a  libft.h  test  test.c
provira-@DESKTOP-JT765AG:~/Cursus/test$ test
provira-@DESKTOP-JT765AG:~/Cursus/test$ ./test
S
```

**En mi caso, por permisos de usuario, utilizo 'sudo', y es una opción desaconsejada.*

make – Makefile

El comando de linux `make` facilita la compilación de los programas, siendo capaz de saber qué hay que compilar y qué no hay que recompilar. También guarda los comandos de compilación con todos los parámetros establecidos. Las normas por defecto se encuentran en un fichero llamado `make.rules`.

Si deseamos crear nuestras propias reglas, lo haremos mediante el fichero sin extensión `Makefile` o `makefile`, aunque por convención se recomienda `Makefile`. Podríamos llamarle de otra manera, pero en ese caso, debemos indicarlo al comando `make` con uno de estos flags:

- `make -f NOMBRE_ARCHIVO`
- `make -file = NOMBRE_ARCHIVO`
- `make --makefile = NOMBRE_ARCHIVO`

El comando `make` tiene la peculiaridad de que puede detectar archivos más nuevos mediante la comparativa por fechas, eso hace que, a la hora de tomarlos, sepa cuales debe usar y cuales no para evitando rehacer el proyecto entero. Eso es importante, pues en proyectos grandes, evita perder mucho tiempo repitiendo lo que ya es correcto.

Por tanto, debemos tener claro que existe el comando `make` y el archivo `Makefile`.

- **make:** Comando con comandos predefinidos que pueden alterarse.
- **Makefile:** Archivo que contiene nuestros propios comandos.

Makefile es un archivo que contiene cuatro elementos muy importantes:

- Reglas.
- Definiciones de variables.
- Directivas y funciones.
- Comentarios.

Reglas: Indican a `make` qué archivos dependen de otros archivos. Las reglas tienen la siguiente sintaxis:

```
objetivo: dependencias
    receta
```

La siguiente regla explícita indica que para crear el archivo 'main' (objetivo) deben existir `main.c` y `funciones.h` (dependencias). Si eso se cumple, ejecuta el comando `gcc -o main main.c funciones.c` (receta). Para que esto suceda, *el objetivo* no debe existir o bien *las dependencias* han sido modificadas.

```
main: main.c funciones.h
    gcc -o main main.c funciones.c
```

La receta debe estar siempre precedida por una tabulación.

También podemos aprovechar las reglas por defecto y decirle mediante una regla implícita lo mismo:

```
main: main.c funciones.h
```

Para esto, es importante saber qué flags se están usando y poder sobrescribirlas. En el siguiente enlace encontraremos las variables que se usan. Por ejemplo, `CFLAGS` tiene el valor `-g`, y podemos sobrescribirla así en nuestro `Makefile`:

```
CFLAGS = -c
```

Si no queremos sobrescribirla, simplemente creamos otra variable con un nombre distinto al reservado y le atribuimos el valor deseado:

```
C_FLAGS = -c
```

Por defecto, `make` busca los archivos en el mismo directorio raíz, así que será normal que nos incluya correctamente nuestra librería `.h` sin indicarle el directorio. Pero si retomamos el ejemplo anterior: imaginemos que `funciones.h` se encuentra en la carpeta `\include\`. Para incluirlo correctamente en la compilación, deberíamos hacerlo así

```
main: main.c include/funciones.h
    gcc -I./include main.c -o main
```


Definiciones de variables: Aunque hay varias maneras de declarar variables, nos ceñiremos únicamente a todas aquellas utilizadas en el proyecto¹²⁰.

La principal característica de `make` es que es `CaseSensitive`. Par crear una variable y darle valor se puede hacer de las siguientes maneras:

- `VAR_REC = valor_variable` (recursividad)
- `VAR_NRC := valor_variable` (expansión simple)

Cuando usamos la recursividad en las variables, implica que no podemos reusarlas consigo mismas, ya que caeríamos en un bucle infinito. Son muy útiles, pero hay que tener mayor control sobre ellas para no caer en errores o lentitud de ejecución:

- `VAR_REC = $(VAR_REC) -g <<< error`

En el caso de expansión simple, la variable adopta el último valor dado, facilitando la escritura de `Makefiles` altamente complejos. Los siguientes ejemplos son equivalentes:

Ejemplo 1			Ejemplo 2		
<code>SALUDO</code>	<code>:=</code>	<code>Hola</code>	<code>NOMBRE</code>	<code>:=</code>	<code>Hola Agustín</code>
<code>NOMBRE</code>	<code>:=</code>	<code>\$(SALUDO) Agustín</code>	<code>SALUDO</code>	<code>:=</code>	<code>Buenos días</code>
<code>SALUDO</code>	<code>:=</code>	<code>Buenos días</code>			

Otro tipo de variables, son aquellas específicas a un patrón. Estas se utilizan cuando varios objetos tienen un valor distinto, así como un aspecto en común. La sintaxis es la siguiente:

- `patrón: asignación-variable`

El patrón debe contener el símbolo `%` para representar la parte distinta del objeto.

```
%.o: %.c
gcc -c $@ $<
```

En este último ejemplo, se habrá notado que existen un par de símbolos extraños. Estas son las variables especiales. Será fácil identificarlas con esta tabla:

Nombre	Descripción
<code>\$@</code>	El nombre del objetivo de la regla. En caso de que existan más de un objetivo, se refiere al objetivo específico que ha invocado la receta.
<code>\$<</code>	El nombre del primer pre-requisito. Si la regla es implícita, será el primer pre-requisito añadido por la regla implícita.
<code>\$?</code>	Los nombres de todos los pre-requisitos más nuevos que el objetivo, separados por espacios en blanco.
<code>\$^</code>	Los nombres de todos los pre-requisitos, separados por espacios en blanco. Esta lista no contiene los pre-requisitos de sólo orden.
<code>\$ </code>	Los nombres de todos los pre-requisitos de sólo orden, separados por espacios.

¹²⁰ Makefiles, DGIIMUnderground, diagmatrix, Github

De esta manera sabremos que la recta dice así:

```
%o: %.c
      gcc -c file1.o file2.o file3.o[...] file1.c file2.c file3.c[...]
              $@                      $<
```

Para tener en cuenta los cambios ocurridos en `Makefile` y `libft.h`, es la siguiente:

```
%o: %.c Makefile funciones.h
      gcc -c $@ $<
```

También existen las variables reservadas, donde únicamente citaré un par que pueden ser útiles para nuestro proyecto:

Nombre	Descripción
MAKEFILE_LIST	Contiene el nombre del último archivo <code>makefile</code> abierto por la orden <code>make</code>
.PHONY	Esta variable se usa como objetivo de una regla sin recetas. Los pre-requisitos especificados en esa regla se ejecutan siempre que se realice una orden que puede construirlos o que ejecuta estas reglas, sin importar que no haya habido ninguna actualización desde la última vez que <code>make</code> fue ejecutado.

Finalmente hay algunas funciones que pueden ser de mucha ayuda a la hora de optimizar el archivo `Makefile`, así como la organización eficiente de nuestros archivos a la hora de compilarlos.

Nombre	Descripción
<code>\$(dir names...)</code>	Extrae el directorio de cada archivo identificado en <i>names</i>
<code>\$(abspath names)</code>	Por cada archivo identificado en <i>names</i> , retorna un nombre absoluto que no contiene punto (.) o dos puntos (..).
<code>\$(firstword names)</code>	El argumento <i>names</i> se retorna como una serie de nombres separados por un espacio.
<code>\$(notdir names)</code>	Por cada archivo identificado en <i>names</i> , extrae todo excepto la parte del directorio.
<code>\$(wildcard pattern)</code>	Expande la variable a todas las ocurrencias del patrón.

Finalmente, si queremos usar una variable, recurriremos a la siguiente acción:

- `$(NOMBRE_VARIABLE)`

El siguiente ejemplo obtiene dinámicamente la ruta de nuestro proyecto y, por tanto, nuestro archivo `Makefile`. Esta base puede servir como punto de inicio para un proyecto:

```
# Source files
CURRENT_DIR      := $(dir $(abspath $(firstword $(MAKEFILE_LIST))))
CURRENT_FILES    := $(notdir $(wildcard $(CURRENT_DIR)*.c))
```

En el archivo `Makefile`, podemos crear una nueva regla que nos imprima los resultados para comprobar lo que se ha hecho:

```
Imprimir:
    @echo $(CURRENT_DIR)
    @echo $(CURRENT_FILES)
```

En la consola ejecutamos `make Imprimir`.

Ruta de archivos

A continuación, se presentan dos versiones de `Makefile` que son compatibles entre ellas, excepto que una de las dos, obtiene los archivos de forma dinámica.

Ejemplo 1:

```
# Source files
CURRENT_DIR      := $(dir $(abspath $(firstword $(MAKEFILE_LIST))))
CURRENT_FILES    := $(notdir $(wildcard $(CURRENT_DIR)*.c))
OBJECTS          := $(CURRENT_FILES:.c=.o)
```

Ejemplo 2:

```
# Source files
CURRENT_FILES    := ft_isalpha.c ft_memchr.c ft_putchar_fd.c \
                   ft_strchr.c ft_strlcpy.c ft_strrchr.c ft_atoi.c

OBJECTS          := $(CURRENT_FILES:.c=.o)
```

Makefile 1.1.4

```
# Program name
NAME                := libft.a

# Source files
CURRENT_FILES       := ft_isalpha.c ft_memchr.c ft_putchar_fd.c ft_strchr.c \
BONUS_FILES         := ft_lstadd_back.c ft_lstclear.c ft_lstiter.c

OBJECTS              := $(CURRENT_FILES:.c=.o)
BONUS                := $(BONUS_FILES:.c=.o)

# Include header
INCLUDE_DIR          := ./include/
INCLUDE              := $(INCLUDE_DIR)libft.h

# Compiler
GCC                  := gcc
GCC_FLAGS             := -Wall -Wextra -Werror

# Console
AR                   := ar
AR_FLAGS              := -crs

RM                   := rm
RM_FLAGS              := -rf

#pattern vars
%.o: %.c $(INCLUDE) Makefile
    $(GCC) $(GCC_FLAGS) -o $@ -c $< -I$(INCLUDE_DIR)

# Rules
all: $(NAME)

$(NAME): $(OBJECTS) $(BONUS)
    $(AR) $(AR_FLAGS) $@ $?

bonus: $(OBJECTS) $(BONUS)
    $(AR) $(AR_FLAGS) $(NAME) $?

clean:
    $(RM) $(RM_FLAGS) $(OBJECTS) $(BONUS)

fclean: clean
    $(RM) -f $(NAME)

re: fclean all

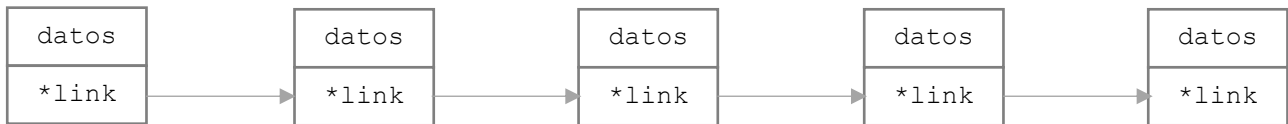
.PHONY: all clean fclean re
```

- Incorporad `libft.h` en el directorio `/include/` dentro del proyecto `libft`.
- Ejecutad en Shell: `make -r` para comprobar que funciona.
- Finalmente, ejecutad en Shell: `make clean` y `make fclean`.
- Posiblemente Include no funcione con *francinette* y el comando `paco`.

BONUS – Listas Enlazadas (básico)

Las listas son colecciones de elementos, o nodos, dispuestos uno detrás de otro. Cada uno de ellos se enlaza con el siguiente mediante un “Enlace” o “Puntero”.

Cada nodo está compuesto por datos de distintos tipos y un puntero que enlaza con el siguiente nodo. El último nodo debe apuntar a `NULL` para dar la lista por finalizada. A continuación, una representación gráfica de varios nodos generando una lista.



Así pues, las listas contienen estructuras de datos dinámicos, y eso hace que puedan cambiar de tamaño durante la ejecución de un programa. Además, podemos insertar o eliminar nodos en cualquier parte de la lista ya que no existen reglamentos que nos restrinjan tales acciones.

Definición

Para definir un nodo con dos campos debemos utilizar el tipo `struct` y le indicaremos su identificador como tipo de datos.

```

typedef struct                <id_estructura> {
    char                      *<variable_o_puntero>;
    struct <id_estructura>    *<enlace_al_siguiete_nodo>;
}                             <tipo_de_dato>;
  
```

Quedando de la siguiente manera:

```

typedef struct                s_elementoLista {
    char                      *datos;
    struct s_elementoLista    *siguiente;
}                             t_elemento;
  
```

Insertar un elemento en la lista

Desarrollamos la función. Esta debe estar presente antes de cualquier otra operación sobre la lista. Comprobamos que inicializa los valores de los punteros y de la variable:

```

Elemento    *InsertarNodo(void *contenido)
{
    t_elemento *new_nodo; //Creamos un puntero de tipo Elemento

    new_nodo = (t_elemento *)malloc(sizeof(t_elemento)); //Reservamos memoria
    if (!new_nodo) //Comprobamos que el puntero no sea NULL
        return (NULL);
    new_nodo->datos = contenido; //Asignamos el valor de *contenido a datos
    new_nodo->siguiente = NULL; //Cerramos la lista
    return (new_nodo);          //*new_nodo pasa a ser el último nodo
}
  
```

Aplicación de la función

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

typedef struct          s_elementoLista {
    char                *datos;
    struct s_elementoLista *siguiente;
}                      t_elemento;

t_elemento *InsertarNodo(void *contenido)
{
    t_elemento *nodo;

    nodo = (t_elemento *)malloc(sizeof(t_elemento));
    if (!nodo)
        return (NULL);
    nodo->datos = (void *)contenido;
    nodo->siguiente = NULL;
    return (nodo);
}

int main(void)
{
    t_elemento *nodo;
    char        saludo[5];
    char        *ptr_saludo;

    strcpy(saludo, "Hola");
    ptr_saludo = saludo;
    nodo = InsertarNodo(ptr_saludo);
    printf("%s", nodo->datos);
    return (0);
}
```

Referencias: [121](#) [122](#) [123](#) [124](#)

¹²¹ Listas en C, Monografías

¹²² Listas y colas en C, Programación en C, unican

¹²³ Listas enlazadas, ArgiesDario, Github

¹²⁴ Strings en C, Edgardo Hames

ft_lstnew()

Prototipo: `t_list *ft_lstnew(void *content);`

Descripción: Crea un nuevo nodo utilizando `malloc()`. La variable miembro `content` se inicializa con el contenido del parámetro `*content`. La variable `next`, con `NULL`.

Parámetros:

- ***content:** Contenido del nodo creado.

Valores devueltos: El nuevo nodo.

Algoritmo:

```
t_list      *ft_lstnew(void *content)
{
    t_list      *node;

    node = (t_list *)malloc(sizeof(t_list));
    if (!node)
        return (NULL);
    node->content = content;
    node->next = NULL;
    return (node);
}
```

Programa completo¹²⁵:

```

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

typedef struct          s_ficha {
    void                *nombre;
    void                *apellidos;
    struct s_ficha      *next;
}                       t_ficha;

t_ficha *nuevo_cliente(void *nombre, void *apellidos)
{
    t_ficha *nodo;
    nodo = (t_ficha *)malloc(sizeof(t_ficha));
    if (!nodo)
        return (NULL);
    nodo->nombre = nombre;
    nodo->apellidos = apellidos;
    nodo->next = NULL;
    return (nodo);
}

int main(void)
{
    t_ficha *cliente;
    char    name[] = "Guybrush";
    char    surname[] = "Threepwood";
    void    *ptr_name;
    void    *ptr_surname;

    ptr_name = name;
    ptr_surname = surname;
    cliente = nuevo_cliente(ptr_name, ptr_surname);
    printf("Nombre:\t\t%s\n", (char *)cliente->nombre);
    printf("Apellidos:\t%s\n", (char *)cliente->apellidos);
    return (0);
}

```

Output:

```

Nombre:      Guybrush
Apellidos:   Threepwood

```

¹²⁵ se han definido los punteros expresamente como void para experimentar cast/conversor en printf()

ft_lstadd_front()

Prototipo: void ft_lstadd_front(t_list **lst, t_list *new);

Descripción: Añade el nodo `new` al principio de la lista `lst`.

Parámetros:

- ****lst:** La dirección de un puntero al primer nodo de una lista.
- ***new:** Un puntero al nodo que añadir al principio de la lista.

Valores devueltos: Ninguno.

Algoritmo:

```
void ft_lstadd_front(t_list **lst, t_list *new)
{
    if (!lst || !new)
        return ;
    new->next = *lst;
    *lst = new;
}
```

Programa:

```
int main(void)
{
    t_list *lst1 = ft_lstnew("Hello");
    t_list *lst2 = ft_lstnew("World");

    printf("After to add front:\n");
    printf("  lst1:\t%s\n", (char *)lst1->content);
    ft_lstadd_front(&lst1, lst2);
    printf("Before to add front:\n");
    printf("  lst1:\t%s\n", (char *)lst1->content);
    return (0);
}
```

ft_lstsize()

Prototipo: `int ft_lstsize(t_list *lst);`

Descripción: Cuenta el número de nodos de una lista.

Parámetros:

- ***lst:** El principio de la lista.

Valores devueltos: La longitud de la lista.

Algoritmo:

```
int ft_lstsize(t_list *lst)
{
    int i;

    i = 0;
    while (lst)
    {
        lst = lst->next;
        i++;
    }
    return (i);
}
```

Notas sobre la línea: `lst = lst->next;`

El bucle `while` se ejecutará mientras `lst` no sea nulo. El campo que determinará si el siguiente nodo es nulo es `next`. Por ello, cuando detecte `NULL` saldrá del bucle.

Programa:

```
int main(void)
{
    t_list *lst1 = ft_lstnew("Node 1");
    t_list *lst2 = ft_lstnew("Node 2");
    t_list *lst3 = ft_lstnew("Node 3");
    t_list *lst4 = ft_lstnew("Node 4");
    lst1->next = lst2;
    lst2->next = lst3;
    lst3->next = lst4;
    printf ("Number of Items: %d\n", ft_lstsize(lst1));
    return (0);
}
```

Output: Number of Items: 4

ft_lstlast()

Prototipo: `t_list *ft_lstlast(t_list *lst);`

Descripción: Devuelve un puntero que apunta al último nodo de la lista.

Parámetros:

- ***lst:** El principio de la lista.

Valores devueltos: Puntero apuntando al último nodo de la lista.

Algoritmo:

```
t_list      *ft_lstlast(t_list *lst)
{
    if (!lst)
        return (NULL);
    while (lst->next)
        lst = lst->next;
    return (lst);
}
```

Notas sobre la línea: `lst = lst->next;`

El bucle `while` se ejecutará mientras el campo `next` no sea nulo. Cuando detecte `NULL` saldrá del bucle, devolviendo el último nodo de la lista.

```
int    main(void)
{
    t_list *lst;
    t_list *lst1 = ft_lstnew("Node 1");
    t_list *lst2 = ft_lstnew("Node 2");
    t_list *lst3 = ft_lstnew("Node 3");
    t_list *lst4 = ft_lstnew("Node 4");

    lst1->next = lst2;
    lst2->next = lst3;
    lst3->next = lst4;

    lst = (void *)malloc(sizeof(t_list));
    lst = ft_lstlast(lst1);

    printf ("Last Node:\t%s", (char *)lst->content);
    return (0);
}
```

ft_lstadd_back()

Prototipo: void ft_lstadd_back(t_list **lst, t_list *new);

Descripción: Añade el nodo `new` al final de la lista `lst`.

Parámetros:

- ***lst:** el puntero al primer nodo de una lista.
- ***new:** el puntero a un nodo que añadir a la lista.

Valores devueltos: Ninguno.

Algoritmo:

```
void ft_lstadd_back(t_list **lst, t_list *new)
{
    t_list    *temp;

    if (!lst || !new)
        return ;
    if (!(*lst))
    {
        *lst = new;
        return ;
    }
    temp = *lst;
    while (temp->next)
        temp = temp->next;
    temp->next = new;
}
```

ft_lstdelone()

Prototipo: `void ft_lstdelone(t_list *lst, void (*del)(void *));`

Descripción: Toma como parámetro un nodo `lst` y libera la memoria del contenido utilizando la función `del` dada como parámetro, además de liberar el nodo. La memoria de `next` no debe liberarse.

Parámetros:

- ***lst:** El nodo a liberar.
- **(*del)(void *):** Un puntero a la función utilizada para liberar el contenido del nodo.

Valores devueltos: Ninguno.

Algoritmo:

```
void ft_lstdelone(t_list *lst, void (*del)(void *))
{
    if (!lst || !del)
        return ;
    (del)(lst->content);
    free(lst);
    lst = NULL;
}
```

Programa:

```
void del_content(void *content)
{
    free(content);
}

int main(void)
{
    char str[] = "Hello World";
    char *ptr;

    ptr = str;
    t_list *lst = ft_lstnew(ptr);
    printf("Content:\t%s\n", (char *)lst->content);
    ft_lstdelone(lst, del_content);
    printf("Content:\t%s\n", (char *)lst->content);
    return (0);
}
```

Output:

```
Content:      Hello World
[error >> free(): invalid pointer]
```

ft_lstclear()

Prototipo: void ft_lstclear(t_list **lst, void (*del)(void *));

Descripción: Elimina y libera el nodo `lst` dado y todos los consecutivos de ese nodo, utilizando la función `del` y `free`. Al final, el puntero a la lista debe ser `NULL`.

Parámetros:

- ***lst:** la dirección de un puntero a un nodo.
- **(*del)(void *):** un puntero a función utilizado para eliminar el contenido de un nodo.

Valores devueltos: Ninguno.

Algoritmo:

```
void ft_lstclear(t_list **lst, void (*del)(void *))
{
    if (!lst || !del || !(*lst))
        return ;
    ft_lstclear(&(*lst)->next, del);
    (del)((*lst)->content);
    free(*lst);
    *lst = NULL;
}
```

Programa:

```
void del_content(void *content)
{
    free(content);
}

int main(void)
{
    char str1[] = "Hello";
    char str2[] = "World";
    char *ptr1;
    char *ptr2;

    ptr1 = str1;
    ptr2 = str2;
    t_list *lst = ft_lstnew(ptr1);
    ft_lstadd_back(&lst, ft_lstnew(ptr2));
    ft_lstclear(&lst, del_content);
    return (0);
}
```

ft_lstiter()

Prototipo: void ft_lstiter(t_list *lst, void (*f)(void *));

Descripción: Itera la lista `lst` y aplica la función `f` en el contenido de cada nodo.

Parámetros:

- ***lst:** un puntero al primer nodo.
- **(*f)(void *):** un puntero a la función que utilizará cada nodo.

Valores devueltos: Ninguno.

Algoritmo:

```
void ft_lstiter(t_list *lst, void (*f)(void *))
{
    if (!lst || !f)
        return ;
    while (lst)
    {
        f(lst->content);
        lst = lst->next;
    }
}
```

Programa:

```
static void changeContent(void *content)
{
    if (!content)
        return ;
    while(*(unsigned char *)content != '\0')
    {
        if (*(unsigned char *)content != ' ')
            *(unsigned char *)content = '$';
        content++;
    }
}

int main(void)
{
    char str[] = "Hello World";
    char *ptr;
    t_list *node;

    ptr = str;
    node = ft_lstnew(ptr);
    printf("Node content:\t%s\n", (char *)node->content);
    ft_lstiter(node, changeContent);
    printf("Node content:\t%s", (char *)node->content);
    return (0);
}
```

ft_lstmap()

Prototipo: `t_list *ft_lstmap(t_list *lst, void *(*f)(void *), void (*del)(void *));`

Descripción: Itera la lista `lst` y aplica la función `f` al contenido de cada nodo. Crea una lista resultante de la aplicación correcta y sucesiva de la función `f` sobre cada nodo. La función `del` se utiliza para eliminar el contenido de un nodo, si hace falta.

Parámetros:

- ***lst:** Un puntero a un nodo.
- **(*f):** La dirección de un puntero a una función usada en la iteración de cada elemento de la lista.
- **(*del):** Un puntero a función utilizado para eliminar el contenido de un nodo, si es necesario.

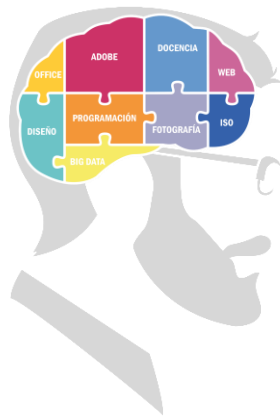
Valores devueltos:

- La nueva lista.
- NULL: Si falla la reserva de memoria.

Algoritmo:

```
t_list      *ft_lstmap(t_list *lst, void *(*f)(void *), void (*del)(void *))
{
    t_list      *new_list;
    t_list      *new_node;
    void        *set;

    if (!lst || !f || !del)
        return (NULL);
    new_list = NULL;
    while (lst)
    {
        set = f(lst->content);
        new_node = ft_lstnew(set);
        if (!new_node)
        {
            del(set);
            ft_lstclear(&new_list, (*del));
            return (new_list);
        }
        ft_lstadd_back(&new_list, new_node);
        lst = lst->next;
    }
    return (new_list);
}
```

[LinkedIn](#)

[GitHub](#)

[YouTube](#)