PROGRAMACIÓN C:

**Cuando pasamos array, la recibimos siempre con un puntero, ya que las arrays se reciben con un puntero al primer valor:**LLAMAMOS A LA FUNCIÓN: longitud = ft\_strlen(cadena);  
RECIBIMOS: size\_t ft\_strlen(const char \*s)

* Se define un puntero llamado “s” que apunta al primer parámetro de “cadena”
* Es “const” porque no se alterará en la función el contenido de “cadena”, no se realizará ninguna modificación en la cadena de caracteres que se recibe como argumento.
* es size\_t: solo números positivos y se utiliza para contar bytes, calcular desplazamientos, representar tamaños o longitudes… mantiene la consistencia y representa tamaños grandes de memoria.

**¿Qué significa el \* del inicio?: void \*ft\_memset(void \*b, int c, size\_t len);***En C, cuando una función devuelve un puntero, se utiliza el asterisco \* al principio de la declaración de la función para indicar que el valor de retorno es un puntero a un tipo específico.*

El \* indica que la función devuelve un puntero al tipo de dato especificado, en este caso es de tipo VOID por lo que el tipo de puntero es genérico y puede apuntar a cualquier tipo de dato.  
un puntero genérico (void\*), significa que no tiene un tipo de dato específico asociado. No puedes realizar operaciones aritméticas de punteros o acceder a los bytes individuales de memoria a través de un puntero genérico.

**Lo mismo pasa con:**  
que significa el \* al inicio de: char \*ft\_itoa(int n);  
indica que la función ft\_itoa devuelve un puntero a un carácter (char \*).

**Lo mismo para:**char \*ft\_strdup(const char \*s1);indica que la función ft\_strdup devuelve un puntero a un carácter (char \*).

**Y dos asteriscos?: char \*\*ft\_split(char const \*s, char c);**indica que la función ft\_split devuelve un puntero a un puntero a un carácter (char \*\*)

EJEMPLO REAL:  
MEMSET:  
void \*ft\_memset(void \*b, int c, size\_t len)

{

size\_t i;

unsigned char \*ptr;

ptr = b;

i = 0;

while (i < len)

{

ptr[i] = (unsigned char)c;

i++;

}

return (b);

}

El parámetro b se declara como void\* porque se pretende que la función ft\_memset sea genérica y pueda trabajar con diferentes tipos de datos.

Al declarar b como void\*, se le permite apuntar a cualquier tipo de dato, ya sea char, int, float u otros. Esto ofrece flexibilidad en el uso de la función, ya que puede ser aplicada a bloques de memoria de diferentes tipos sin necesidad de crear múltiples versiones de la función.

Dentro de la función, cuando se asigna la dirección de b a ptr y se realiza la manipulación de bytes individuales, se utiliza ptr como un puntero a unsigned char para asegurar que los bytes se traten de manera adecuada. Sin embargo, el parámetro b sigue siendo de tipo void\* para mantener la flexibilidad y la capacidad de recibir cualquier tipo de puntero.

**Qué diferencia hay en poner el size\_t al inicio:  
size\_t** ft\_strlen(const char \*s)  
  
**o en la variable:**int ft\_strncmp(const char \*s1, const char \*s2, **size\_t** n);

Cuando se coloca size\_t al inicio de la declaración de una función, como en size\_t ft\_strlen(const char \*s), indica que el tipo de dato de retorno de la función es size\_t. Esto significa que la función ft\_strlen devuelve un valor de tipo size\_t.

Por otro lado, cuando se coloca size\_t en la variable dentro de la declaración de la función, como en int ft\_strncmp(const char \*s1, const char \*s2, size\_t n), indica el tipo de dato del tercer parámetro n. En este caso, n es un valor de tipo size\_t que se utiliza para especificar la cantidad de caracteres a comparar en la función ft\_strncmp.

size\_t ft\_strlen(const char \*s) {

size\_t i = 0; // Variable contador

while (s[i] != '\0') {

i++; // Incrementar el contador mientras se accede a los caracteres de la cadena

}

return i; // Retornar la longitud de la cadena

}  
  
Fíjate que cuando pasamos el puntero del primer valor de la cadena por la función, la cadena completa ya la tenemos en realidad en “s”. Solo creamos otra variable para recorrer la string s[i] y listos, **en este caso un size\_t de i, ya que la función es llamada con size\_t por lo que la variable que devuelve es de tipo size\_t. la función ft\_strlen se especifica que el tipo de dato de retorno es size\_t, entonces es necesario que la variable i también sea de tipo size\_t para ser coherente.**

CUANDO DECLARAMOS ARRAY:  
  
¿Qué diferencia hay de esto:  
const char \*str = "Hola, mundo";

const char str[] = “Hola, mundo”;  
  
Imagina que tienes una caja de caramelos que dice "Hola, mundo" en ella. Ahora, hay dos formas de decirle a alguien que tome esos caramelos.

1. La primera forma es decir: "Los caramelos están en esa caja. Puedes tomarlos, pero no puedes cambiarlos". Esto es similar a const char \*str = "Hola, mundo";. Aquí, la caja representa una región de memoria de solo lectura donde se almacena la cadena "Hola, mundo". Puedes leerla y usarla, pero no puedes modificarla. El puntero str es como una flecha que apunta a la caja para que puedas acceder a los caramelos.
2. La segunda forma es decir: "Aquí tienes una caja con caramelos. Puedes tomarlos, pero no puedes cambiarlos". Esto es similar a const char str[] = "Hola, mundo";. Aquí, la caja misma representa un arreglo de caracteres que contiene la cadena "Hola, mundo". La caja es constante, lo que significa que no puedes cambiar ninguno de los caramelos dentro de ella. Puedes usar la caja para acceder a los caramelos uno por uno o mostrarlos a alguien, pero no puedes modificarlos.

**Entonces, la diferencia radica en cómo se almacena y se accede a la cadena de caracteres**. En el primer caso, tienes un puntero que apunta a una cadena en una región de memoria de solo lectura, y en el segundo caso, tienes un arreglo constante que almacena la cadena directamente. A efectos prácticos, ambos enfoques pueden proporcionar resultados similares en muchas situaciones. Ambos te permiten acceder a la cadena de caracteres "Hola, mundo" y utilizarla en tu código

La gran diferencia entre pasarlo de una manera u otra es que el valor se pierde:

**NO SE PIERDE:** int result1 = ft\_isalnum('c');

**SE PIERDE:**int result1;   
result = 'c';   
ft\_isalnum(result1);

El valor no se "pierde" en sí mismo, pero se produce una conversión implícita de char a int. Cuando asignas el carácter 'c' a la variable result1, se realiza una conversión implícita para ajustar el valor del carácter al tipo int. En otras palabras, el carácter se promociona a su valor ASCII correspondiente antes de asignarse a la variable result1.

Sin embargo, cuando llamas a la función ft\_isalnum con result1 como argumento, ocurre otra conversión implícita en la llamada a la función. La función ft\_isalnum espera recibir un argumento de tipo int, pero en este caso, el valor de result1 se convierte nuevamente en un char antes de ser pasado a la función.

es correcto:  
int result1 = ft\_isalnum('c'); // Pasando el carácter 'c' directamente

int result2 = ft\_isalnum(65); // Pasando el código ASCII del carácter 'A'