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'

DIFERENCIA ENTRE DEVOLVER UN VALOR Y NO DEVOLVERLO
VOID – NO DEVUELVE

Función que devuelve un valor:

- Imagina una máquina que tiene una entrada donde puedes ingresar un número y una salida donde se muestra el cuadrado de ese número. Cuando ingresas un número y presionas un botón, la máquina realiza el cálculo y devuelve el resultado en la salida. Esto representa una función que devuelve un valor, ya que la salida es una respuesta útil y significativa.

Función que no devuelve un valor:

- Ahora, imagina una máquina donde puedes ingresar una pelota en un orificio y la máquina la aplasta. No hay una salida específica o útil en este caso, ya que la pelota simplemente se aplasta y desaparece. Esto representa una función que no devuelve un valor, ya que no hay un resultado o respuesta utilizable.

En resumen, una función que devuelve un valor proporciona una salida significativa que se puede utilizar en el programa, mientras que una función que no devuelve un valor realiza una acción específica pero no proporciona un resultado específico o útil para su uso posterior.

UNSIGNED INT rango de 0 a 65,535. (NO NEGATIVOS)

UNSIGNED CHAR rango de 0 a 255 (no negativos) un byte de memoria.

Este error se produce porque estás intentando acceder a los datos apuntados por un puntero `void *`, que no tiene un tipo de dato específico.

En C, `void *` se utiliza para representar un puntero genérico que puede apuntar a cualquier tipo de datos. Sin embargo, no se puede desreferenciar directamente un puntero `void *` porque no se conoce el tipo de datos al que apunta.

Para solucionar este error, debes realizar un casting explícito del puntero `void *` a un tipo de puntero adecuado antes de desreferenciarlo. En tu caso, parece que estás tratando de trabajar con datos de tipo `unsigned char`.

Aquí tienes un ejemplo de cómo podrías corregir el código:

```
c
const unsigned char *ptr = (const unsigned char *)s;  // Casting
explícito
return &ptr[i];
```

En este caso, se realiza un casting del puntero `void *` a `const unsigned char *` antes de acceder a los datos apuntados por el puntero.

FUNCIONES

- isalpha
- isdigit
- isalnum
- isascii
- isprint
- strlen
- memset
- bzero
- memcpy
- memmove
- strlcpy
- strlcat
- toupper
- tolower
- strchr
- strrchr
- strncmp
- memchr
- memcmp
- strnstr
- atoi

Deberás programar las siguientes funciones, utilizando la función "malloc":

- calloc
- strdup

ISALPHA: COMPRUEBA SI ES ALFABETO. (1 es alfabeto y 0 no lo es).
ISDIGIT: COMPRUEBA SI ES UN DÍGITO. (1 es un dígito y 0 no lo es).
ISALNUM: COMPRUEBA SI ES ALFANUMÉRICO (MAY Y MINUSC) (1 es un alfanumérico y 0 no lo es).
ISASCII: COMPRUEBA SI UN CARÁCTER ES ASCII (0 AL 127) (1 es ascii y 0 no lo es).
ISPRINT: CARÁCTER ES IMPRIMIBLE O NO (0 A 127)SI, NO IMPRIMIBLES(RETORNOS CARRO, \0…

STRLEN (RECORRIDO DE UN ARRAY, SABER EL Nº TOTAL DE ELEMENTOS).

MEMSET: ESTABLECE UN BLOQUE DE MEMORIA A UN VALOR ESPECÍFICO (3 ARGUMENTOS: PUNTERO BLOQUE MEMORIA, VALOR QUE DESEAMOS Y CANTIDAD DE BYTES QUE SE ESTABLECEN.

BZERO: ESTABLECE BLOQUE DE MEMORIA A CEROS (NULO).

MEMOVE: COPIAR UN BLOQUE DE MEMORIA A OTRO BLOQUE, INCLUSO SI LOS BLOQUES SE SOLAPAN. ESTA FUNCIÓN GARANTIZA QUE LOS DATOS SE COPIEN CORRECTAMENTE, INCLUSO SI HAY SOLAPAMIENTO ENTRE LAS ÁREAS DE ORIGEN Y DESTINO.

MEMCPY: COPIA BLOQUE DE MEMORIA A OTRA UBICACIÓN (BLOQUE) PERO NO TIENE EN CUENTA EL SOLAPAMIENTO A DIFERENCIA DEL MEMOVE.

STRLCPY: MANIPULACIÓN DE CADENAS, COPIA CADENA(STRING)ORIGEN A UNA CADENA DE DESTINO CON UN LÍMITE MÁXIMO DE LONGITUD, EVITA DESBORDAMIENTOS DEL BÚFER AL COPIAR CADENAS.

STRLCAT: MANIPULACIÓN DE CADENAS (CONCATENA 2 CADENAS)

TOUPPER Y TOLOWER: MAYÚSC Y MINÚS (+32 Y -32)

STRCHR: (DE INICIO A FIN) BUSCAR LA PRIMERA APARICIÓN DE UN CARÁCTER ESPECÍFICO EN UNA CADENA DE CARACTERES. DEVUELVE UN PUNTERO AL PRIMER CARÁCTER ENCONTRADO O UN PUNTERO NULO SI EL CARÁCTER NO ESTÁ PRESENTE EN LA CADENA.

STRRCHR: LO MISMO QUE ARRIBA PERO DE (FIN A INICIO).

STRNCMP: COMPARAR LOS PRIMEROS `N` CARACTERES DE DOS CADENAS DE CARACTERES. COMPARA LAS CADENAS DE MANERA LEXICOGRÁFICA Y DEVUELVE UN VALOR ENTERO QUE INDICA SI LAS CADENAS SON IGUALES, SI LA PRIMERA CADENA ES MENOR QUE LA SEGUNDA O SI LA PRIMERA CADENA ES MAYOR QUE LA SEGUNDA.

MEMCHR: SE UTILIZA PARA BUSCAR LA PRIMERA APARICIÓN DE UN BYTE ESPECÍFICO EN UN BLOQUE DE MEMORIA. DEVUELVE UN PUNTERO AL PRIMER BYTE ENCONTRADO O UN PUNTERO NULO SI EL BYTE NO ESTÁ PRESENTE EN EL BLOQUE DE MEMORIA.

MEMCMP: SE UTILIZA PARA COMPARAR DOS BLOQUES DE MEMORIA BYTE A BYTE. COMPARA LOS PRIMEROS `N` BYTES DE DOS BLOQUES DE MEMORIA Y DEVUELVE UN VALOR ENTERO QUE INDICA SI LOS BLOQUES SON IGUALES, SI EL PRIMER BLOQUE ES MENOR QUE EL SEGUNDO O SI EL PRIMER BLOQUE ES MAYOR QUE EL SEGUNDO.

STRNSTR: `no crea una nueva cadena` BUSCA UNA SUBCADENA DENTRO DE UNA CADENA DE CARACTERES, TENIENDO UN LÍMITE MÁXIMO DE LONGITUD PARA LA BÚSQUEDA.

```
char haystack[] = "Hola, mundo!";
    char needle[] = "mundo";
```

busca la subcadena `needle` dentro de la cadena `haystack`.

ATOI: CONVERTIR UNA CADENA DE CARACTERES QUE REPRESENTA UN NÚMERO EN SU EQUIVALENTE ENTERO. EL NOMBRE "ATOI" PROVIENE DE "ASCII TO INTEGER" (ASCII A ENTERO), YA QUE INTERPRETA LA CADENA DE CARACTERES COMO UNA SECUENCIA DE DÍGITOS ASCII QUE REPRESENTA UN NÚMERO ENTERO.

NO ENTRA: MALLOC: asignar memoria dinámica durante tiempo de ejecución. Permite reservar un bloque de memoria de un tamaño especificado en bytes y devuelve un puntero al inicio de dicho bloque.

```
int *arr = (int *)malloc(num_elements * element_size);
```

1. `int *arr`: Esto declara una variable llamada `arr` que es un puntero a un entero (`int *`). El puntero `arr` se utilizará para acceder y trabajar con el arreglo de enteros que se va a asignar.
2. `malloc(num_elements * element_size)`: La función `malloc` se llama con un argumento que indica el tamaño total de la memoria que se desea asignar. En este caso, el tamaño total se calcula multiplicando `num_elements` (número de elementos en el arreglo) por `element_size` (tamaño en bytes de cada elemento).

   Por ejemplo, si `num_elements` es igual a 5 y `element_size` es igual al tamaño de un entero (`sizeof(int)`), entonces el tamaño total será de 20 bytes (5 enteros * tamaño de un entero).

   La función `malloc` intenta reservar un bloque de memoria contigua de tamaño `num_elements * element_size` y devuelve un puntero al inicio de dicho bloque. El tipo de retorno de `malloc` es `void *`, por lo que se realiza un casting explícito `(int *)` para convertir el puntero a un puntero a entero (`int *`).

   Es importante destacar que el puntero devuelto por `malloc` puede ser `NULL` si la asignación de memoria falla debido a la falta de memoria disponible.

CALLOC: se utiliza para asignar memoria dinámica para un arreglo de elementos y establecer todos los bytes asignados en cero. Es una variante de la función `malloc`. Reserva suficiente memoria para alojar `num_elements` elementos, cada uno con un tamaño de `element_size` bytes. Luego, inicializa todos los bytes de la memoria asignada en cero. Devuelve un puntero al inicio del bloque de memoria asignado o un puntero nulo (`NULL`) si la asignación falla.

```
int *arr = (int *)calloc(num_elements, element_size);
```

STRDUP: utiliza para duplicar una cadena de caracteres. Permite crear una copia exacta de una cadena existente, incluyendo su contenido y tamaño, asignando memoria dinámicamente para la **nueva cadena**. La función `strdup` crea una copia de la cadena `str` y devuelve un puntero al inicio de la nueva cadena. La nueva cadena es asignada dinámicamente utilizando `malloc` para reservar suficiente memoria para almacenar el contenido de la cadena `str`, incluyendo el carácter nulo de terminación (`'\0'`).

Recibe como argumento un puntero a una cadena y crea una copia exacta de esa cadena en una nueva ubicación de memoria asignada dinámicamente.

SUBSTR

SI crea una nueva cadena en C. se utiliza para obtener una porción o subcadena de una cadena más grande. Toma como argumentos la cadena original, el índice de inicio y la longitud de la subcadena que se desea extraer.

`ft_substr` se utiliza para extraer una subcadena de una cadena dada, mientras que `strnstr` se utiliza para buscar la primera aparición de una subcadena en una cadena dada.

STRJOIN

concatenar (unir) varias cadenas en una sola cadena. concatena dos cadenas sin verificar el límite de tamaño. En el ejemplo, se asume que hay suficiente espacio en el búfer de destino para acomodar la concatenación de las dos cadenas. No se realiza una verificación explícita del tamaño del búfer. COMO SI PASA CON STRLCAT

STRTRIM

se utiliza para eliminar los espacios en blanco al inicio y al final de una cadena.

```
char *strtrim(const char *str) {
    size_t start = 0;
    size_t end = strlen(str) - 1;

    // Buscar el primer carácter no espaciado desde el inicio
    while (isspace(str[start])) {
        start++;
    }

    // Buscar el último carácter no espaciado desde el final
    while (end > start && isspace(str[end])) {
        end--;
    }

    // Calcular la longitud de la cadena resultante
    size_t length = end - start + 1;

    // Asignar memoria para la nueva cadena resultante
    char *trimmed_str = (char *)malloc((length + 1) * sizeof(char));

    if (trimmed_str != NULL) {
        // Copiar la porción no espaciada de la cadena original a la nueva cadena
        strncpy(trimmed_str, str + start, length);
        trimmed_str[length] = '\0';  // Agregar el carácter nulo de terminación
    }

    return trimmed_str;
}
```

SPLIT
dividir una cadena en subcadenas más pequeñas, utilizando un delimitador como referencia.

ITOA (CONTRARIO DE ATOI)
convertir un número entero en una cadena de caracteres.

STRMAPI
aplicar una función a cada carácter de una cadena y generar una nueva cadena con los resultados de dicha función(POR EJEMPLO MAYÚSCULAS). La función aplicada toma como argumento cada carácter y su índice en la cadena original.

STRITERI
funcionalidad similar a `strmapi`, pero sin el índice como argumento adicional. La función `striteri` se utilizaría para aplicar una función a cada carácter de una cadena sin tener en cuenta su posición. En este ejemplo, la función `striteri` toma una cadena (`str`) y una función (`f`) como argumentos. La función `f` se aplica a cada carácter de la cadena. La función `striteri` genera una nueva cadena (`iterated_str`) con los resultados de la función aplicada a cada carácter.

PUTCHAR_FD
se utiliza para escribir un carácter en un archivo descriptor dado. Cuando hablo de "descriptor dado" en el contexto de la función `putchar_fd`, me refiero a un descriptor de archivo específico proporcionado como argumento a la función. En el lenguaje C, los descriptores de archivo son enteros que identifican los archivos abiertos o los flujos de entrada/salida estándar.

En la función `putchar_fd`, el descriptor de archivo indica el destino donde se escribirá el carácter. En lugar de escribir en la salida estándar (stdout), que es el comportamiento predeterminado de la función `putchar`, se especifica un descriptor de archivo específico.

Un descriptor de archivo puede ser obtenido al abrir un archivo utilizando la función `fopen`, o también se pueden utilizar valores predefinidos para los flujos estándar:

- `stdin` (entrada estándar) tiene el descriptor de archivo 0.
- `stdout` (salida estándar) tiene el descriptor de archivo 1.
- `stderr` (salida de error estándar) tiene el descriptor de archivo 2.

En el ejemplo anterior, se utiliza `fopen` para abrir el archivo "output.txt" en modo de escritura y se pasa el puntero de archivo (`FILE*`) como argumento al llamar a la función `putchar_fd`. Esto permite que el carácter se escriba en el archivo en lugar de la salida estándar.

```
int main() {
    char c = 'A';
    FILE* file = fopen("output.txt", "w");

    if (file != NULL) {
        putchar_fd(c, file);

        fclose(file);
    }

    return 0;
}
```

PUTSTR_FD

escribir una cadena de caracteres en un archivo descriptor dado.

```
void putstr_fd(const char* str, FILE* stream) {
    fputs(str, stream);
}

int main() {
    const char* message = "Hola, mundo!";
    FILE* file = fopen("output.txt", "w");

    if (file != NULL) {
        putstr_fd(message, file);

        fclose(file);
    }

    return 0;
}
```

PUTENDL_FD

escribir una cadena de caracteres seguida de un salto de línea en un archivo descriptor dado.

```
void putendl_fd(const char* str, FILE* stream) {
    fputs(str, stream);
    fputc('\n', stream);
}

int main() {
    const char* message = "Hola, mundo!";
    FILE* file = fopen("output.txt", "w");

    if (file != NULL) {
        putendl_fd(message, file);

        fclose(file);
    }

    return 0;
}
```

PUTNBR_FD

escribir un número entero en un archivo descriptor dado.

```
void putnbr_fd(int num, FILE* stream) {
    fprintf(stream, "%d", num);
}

int main() {
    int number = 12345;
    FILE* file = fopen("output.txt", "w");

    if (file != NULL) {
        putnbr_fd(number, file);

        fclose(file);
    }

    return 0;
}
```