get_next_line(int fd)

Prototipo: char *get_next_line(int fd)

fd = File description. Es un número único que se asigna a cada archivo abierto en un programa. Sirve como una forma de identificar y acceder a un archivo especifico.

char *get_next_line(int fd) {	'get_next_line' recibe como argumento 'fd' que es el archive del cuál se lleerán las líneas.
char *line;	Se declara la variable 'line' que se utilizará para almacenar la línea leída.
static char *buffer = NULL;	Se declara 'buffer' que se utilizará para almacenar el contenido del archivo.
line = NULL;	Establecemos 'line' en 'NULL'. Así indicamos que aún no se ha leído ninguna línea.
if (fd < 0 || BUFFER_SIZE <= 0 || read(fd, 0, 0) < 0) {	PROTECCION: Verificamos: 1. Que 'fd' sea válido. Es decir que el resultado sea mayor o igual a 0. 2. Que 'BUFFER_SIZE' sea válido. Es decir que el resultado sea mayor a 0. 3. Que se pueda leer el archivo 'fd'. Es decir que el resultado sea mayor o igual a 0.
if (buffer != NULL) {	Si ninguna de las condiciones se cumplen se liberará el 'buffer' sieme y cuando no sea 'NULL'.
free(buffer);	Se libera el 'buffer'
buffer = NULL;	Se establece 'buffer' en 'NULL'.
} return (NULL);	Devolvemos 'NULL' indicando que ha habido un error.
} buffer = file_reader(fd, buffer);	Si se cumplen todas las condiciones, Se llamará la función 'file_reader' para leer los datos en bloques y manejar el almacenamiento en el 'buffer'.
if (!buffer) return (NULL);	Si 'buffer' es 'NULL' significará que no se pudieron leer los datos en el archivo y se devolverá 'NULL' indicando que ha habido un error
line = line_isolator(buffer);	Se llama la función 'line_isolator' que aislará la línea del contenido del 'buffer'. Si no se encuentra una línea completa, se devolverá 'NULL' indicando un error.
buffer = next_line(buffer);	Se llama la función 'next_line' para mover el 'buffer' al inicio de la siguiente línea o al final del archivo si no quedan más líneas. Esta función se encargará de ajustar el 'buffer' para que apunte al siguiente contenido después de la línea leído.
return (line); }	Devolveremos el resultado asignado a la variable 'line'. Contendrá la línea leído del archivo 'NULL' en el caso que la función 'line_isolator' haya devuelto 'NULL'.

file_reader(int fd, char *buffer)

Prototipo: char *file_reader(int fd, char *buffer)

fd: File description
buffer: reserva de ‘buffer’ que se ha hecho en la función main.

char *file_reader(int fd, char *buffer) {	‘file_reader’ recibe como argumento ‘fd’ y un puntero a un buffer.
char *data;	Se declara un puentor ‘data’ que se utilizará para almacenar de forma temporal los datos leídos del archivo
int bytes;	Se declara un entero ‘bytes’ que se utilizará para almacenar el número de bytes leídos del archivo.
data = (char *)malloc(sizeof(char) * (BUFFER_SIZE + 1));	Reservamos memoria: - La cantidad de memoria asignada es ‘BUFFER_SIZE + 1’ donde ‘BUFFER_SEIZE’ es un valor definido por el usuario. - Se utiliza ‘sizeof(Char)’ para asegurar un tamaño correcto para un elemento ‘char’ (1byte)
if (!data) return (NULL);	Si la asignación de memoria falla, se devuelve ‘NULL’ para indicar que ha habido un error.
data[0] = '\0';	Se define la primera posición de ‘data’ como un carácter nulo. - Esto iniciliza ‘data ‘ como una cadena vacía.
bytes = 1;	Se inicializa ‘bytes’ con el valor 1.
if (!buffer) buffer = ft_calloc(1, 1);	Se verifica que ‘buffer’ es ‘NULL’. Lo que indicaría que aun no se le ha asignado memoria. - Si es así, se llama la función ‘ft_calloc()’ para asignar memoria y establecer todos los bytes de memoria a cero. - Esto garantiza que ‘buffer’ se inicie como una cadena vacía si todavía no se le ha asignado previamente memoria.
while (bytes != 0 && !ft_strchr(buffer, '\n')) {	Mientras ‘bytes’ no sea igual a 0 y no se encuentre el carácter ‘\n’ dentro del ‘buffer’ se inicia el bucle.
bytes = read(fd, data, BUFFER_SIZE);	Se llama la función la función ‘read’ para leer los datos del ‘fd’ - Los datos se leen en ‘data’ con un tamaño máximo especificado por ‘BUFFER_SIEZ’. - El número de ‘bytes’ leídos se asigna a la variable ‘bytes’.

if (bytes == -1)
{
free(data);
return (NULL);
}
	Verificamos si ‘bytes’ es igual a -1.
	- Si es true, indica que hay un error en la lectura del archivo.
	- Si es true, se libera la memoria asignada en ‘data’ y devolvemos ‘NULL’.

data[bytes] = '\0';	Al final de los datos leídos en ‘data’ establecemos el carácter ‘\0’. - Esto asegura que ‘data’ se trate como una cadena de caracteres válido.
buffer = ft_join_and_free(buffer, data); }	Se llama la función ‘ft_join_and_free()’ para concatenar ‘buffer’ y ‘data’ y liberar la memoria previamente en ‘buffer’. - El resultado de la concatenación se asigna a ‘buff’er’.
free(data);	Se libera la memoria asignada en ‘data’. - Ya no se necesita ‘data’ después de concatenar los datos y asingarlos a ‘buffer’.
return (buffer); }	Se devuelve el puntero ‘buffer’ que contiene los datos leídos del archivo hasta el momento. - Si no se ha leído ningún dato o ha ocurrido un error, ‘buffer’ puede ser ‘NULL’.

line_isolator(char *buffer)

Prototipo: char *line_isolator(char *buffer)

buffer: reserva de ‘buffer’ que se ha hecho en la función main.

char *line_isolator(char *buffer) {	‘line_isolator’ recibe como argumento un puntero ‘buffer’
char *line;	Se declara un puntero ‘line’ que se utilizará para almacenar la línea aislada del buffer
int i;	Se declara un entero ‘i’ que se utilizará como contador.
i = 0;	Iniciamos ‘i’ a 0.
if (!buffer[i]) return (NULL);	Verificamos si el primer carácter de ‘buffer’ es ‘NULL’ lo cual indica que el ‘buffer’ está vacío. - Si es así, se devuelve ‘NULL’ para indicar que no hay línea para aislar y que hay un error.
while (buffer[i] && buffer[i] != '\n') i++;	Mientras haya caracteres en ‘buffer’ y no encontramos un ‘\n’ Sumaremos una posición al contador.
line = (char *)malloc(sizeof(char) * (i + 2));	Reservamos en el ‘heap’ - La cantidad de memoria a reservar es (i + 2). - ‘i’ es la longitudo de la línea en el ‘buffer’. - ‘2’ es para garantizar que haya suficiente espacio para incluir ‘\n’ y ‘\0’
if (!line) return (NULL);	Verificamos la asignación de memoria. - Si ha habido un error, devolveremos ‘NULL’.
i = 0;	Reiniciamos el contador a 0
while (buffer[i] && buffer[i] != '\n') {	Mientras haya caracteres en ‘buffer’ y no encontramos un ‘\n’
line[i] = buffer[i];	Copiamos el carácter actual de ‘buffer’ a ‘line’
i++; }	Incrementamos el Contador al siguiente carácter.
if (buffer[i] == '\n') {	Verificamos si se ha encontrado un ‘\n’ en ‘buffer’
line[i] = buffer[i];	Se copia el carácter ‘\n’ a ‘line’
i++; }	Incrementamos el contador al siguiente carácter.
line[i] = '\0';	Se agrega el carácter ‘\0’ al final de ‘line’ - Esto asegura que ‘line’ se trate como una cadena de caracteres válida.
return (line); }	Se devuelve el puntero ‘line’ que contiene la línea aislada del ‘buffer’ - Si no se encontró ninguna línea o se produjo un error, ‘line’ puede ser ‘NULL’.

next_line(char *buffer)

Prototipo: char *next_line(char *buffer)

buffer: reserva de 'buffer' que se ha hecho en la función main.

char *next_line(char *buffer) {	'next_line' recibe como argumento un puntero 'buffer'
int i;	Declaramos la variable 'i' que se utilizará como contador
int j;	Declaramos la variable 'j' que se utilizará como contador
char *next_line;	Se declara el puntero 'next_line' que se utilizará par almacenar la siguiente línea después de la actual
i = 0;	Iniciamos el contado 'i' a 0.
while (buffer[i] && buffer[i] != '\n') {	Mientras haya caracteres en 'buffer' y no se encuentre '\n'
i++; }	Se incrementará el valor de 'i' para saber la posición del carácter del '\n' en el 'buffer'
if (!buffer[i]) {	Si al final del 'buffer' no hay más líneas.
free(buffer);	Liberamos la memoria asignada a 'buffer'
return (NULL); }	Se devuelve 'NULL'.
next_line = (char *)malloc(sizeof(char) * (ft_strlen(buffer) -i + 1));	Reservamos memoria en el 'heap' - La cantidad de memoria es igual a la longitud restante del 'buffer' después de la línea actual más 1. - 'i' representa la posición del primer carácter después de '\n' en el 'buffer', la resta '-i' se realiza para obtener la longitud restante del buffer desde esa posición hasta el final. - '1' se agrega para reservar espacio para el carácter '\0' - En resumen (-i + 1) se utiliza para garantizar que se reserve suficiente memoria para almacenar la parte restante del buffer y el carácter que indicar el final de la cadena.
if (!next_line) return (NULL);	Verificamos la asignación de memoria. - Si ha habido un error, devolvemos 'NULL'
i++;	Se incrementa el valor de 'i' para saltar el primer carácter después de '\n'
j = 0;	Iniciamos 'j' como contador para el próximo bucle.
while (buffer[i]) {	Ejecutamos un bucle 'while' mientras haya caracteres en 'buffer'
next_line[j++] = buffer[i++]; }	Se copian los caracteres del 'buffer' a 'nex_line' incrementando el siguiente carácter y posición.
next_line[j] = '\0';	Se agrega el carácter '\0' al final de 'next_line' - Así aseguramos que sea tratado como una cadena de caracteres válido.
free(buffer);	Se libera la memoria asignada para 'buffer' original, ya que no es necesario.
return (next_line); }	Se devuelve un puntero 'next_line' que contiene el contenido del 'buffer' después de la línea actual. - Si hubo un error o no hay más contenido, 'next_line' puede ser 'NULL'

ft_join_and_free(char *buffer, char *data)

Prototipo: char *ft_join_and_free(char *buffer, char *data)

buffer: reserva de 'buffer' que se ha hecho en la función 'file_reader'.
Data: reserva de 'data' que se ha hecho en la función 'file_reader'.

char *ft_join_and_free(char *buffer, char *data) {	'ft_join_and_free' recibe dos punter a caracteres como argumentos: 'buffer' y 'data'.
char *temp;	Se declara la variable 'temp' que será utilizada para almacenar el resultado de la concatenación
temp = ft_strjoin(buffer, data);	Se utilizará la función 'ft_strjoin()' para concatenar los strings 'buffer' y 'data' y el resultado se asigna a la variable 'temp'
free (buffer);	Se libera la memoria ocupada por 'buffer' para evitar fugas de memoria.
return (temp); }	Se devuelve el puntero 'temp' que contiene el resultado de la concatenación.