GUIA EXAM RANK 02

by Davmoren

malloc memoria dinamica

En este texto, voy a explicar de manera profunda y detallada el funcionamiento de `malloc`, `realloc` y `calloc`, así como las diferencias entre ellos y cuándo conviene usar cada uno.

Funcionamiento de `malloc`

`malloc` es una función que se utiliza para solicitar memoria dinámica en un programa. Cuando se llama a `malloc`, el sistema operativo busca un bloque de memoria libre que sea lo suficientemente grande para satisfacer la solicitud. Si se encuentra un bloque de memoria adecuado, se devuelve un puntero a la dirección de inicio de ese bloque. Si no se encuentra un bloque de memoria adecuado, se devuelve un puntero nulo (`NULL`).

Aquí hay una descripción paso a paso de cómo funciona `malloc`:

1. *Solicitud de memoria*: El programa solicita memoria dinámica llamando a `malloc` y pasando como argumento el tamaño de memoria que se necesita.
2. *Búsqueda de memoria libre*: El sistema operativo busca un bloque de memoria libre que sea lo suficientemente grande para satisfacer la solicitud.
3. *Verificación de la disponibilidad de memoria*: Si se encuentra un bloque de memoria adecuado, el sistema operativo verifica si la memoria está disponible para su uso.
4. *Asignación de memoria*: Si la memoria está disponible, el sistema operativo asigna el bloque de memoria al programa y devuelve un puntero a la dirección de inicio de ese bloque.
5. *Actualización de la tabla de asignación de memoria*: El sistema operativo actualiza la tabla de asignación de memoria para reflejar la nueva asignación de memoria.

Funcionamiento de `realloc`

`realloc` es una función que se utiliza para cambiar el tamaño de un bloque de memoria que ya ha sido asignado con `malloc`. Cuando se llama a `realloc`, el sistema operativo busca un nuevo bloque de memoria que sea lo suficientemente grande para satisfacer la nueva solicitud de tamaño. Si se encuentra un bloque de memoria adecuado, se copian los datos del bloque de memoria original al nuevo bloque de memoria y se devuelve un puntero a la dirección de inicio del nuevo bloque.

Aquí hay una descripción paso a paso de cómo funciona `realloc`:

1. *Solicitud de cambio de tamaño*: El programa solicita cambiar el tamaño de un bloque de memoria que ya ha sido asignado con `malloc` lla-

mando a `realloc` y pasando como argumentos el puntero al bloque de memoria original y el nuevo tamaño de memoria que se necesita.

2. *Búsqueda de memoria libre*: El sistema operativo busca un nuevo bloque de memoria que sea lo suficientemente grande para satisfacer la nueva solicitud de tamaño.
3. *Verificación de la disponibilidad de memoria*: Si se encuentra un bloque de memoria adecuado, el sistema operativo verifica si la memoria está disponible para su uso.
4. *Copia de datos*: Si la memoria está disponible, el sistema operativo copia los datos del bloque de memoria original al nuevo bloque de memoria.
5. *Asignación de memoria*: El sistema operativo asigna el nuevo bloque de memoria al programa y devuelve un puntero a la dirección de inicio del nuevo bloque.
6. *Actualización de la tabla de asignación de memoria*: El sistema operativo actualiza la tabla de asignación de memoria para reflejar la nueva asignación de memoria.

Funcionamiento de `calloc`

`calloc` es una función que se utiliza para solicitar memoria dinámica y inicializarla con ceros. Cuando se llama a `calloc`, el sistema operativo busca un bloque de memoria libre que sea lo suficientemente grande para satisfacer la solicitud y lo inicializa con ceros. Si se encuentra un bloque de memoria adecuado, se devuelve un puntero a la dirección de inicio de ese bloque.

Aquí hay una descripción paso a paso de cómo funciona `calloc`:

1. *Solicitud de memoria*: El programa solicita memoria dinámica llamando a `calloc` y pasando como argumentos el número de elementos y el tamaño de cada elemento.

2. *Búsqueda de memoria libre*: El sistema operativo busca un bloque de memoria libre que sea lo suficientemente grande para satisfacer la solicitud.
3. *Verificación de la disponibilidad de memoria*: Si se encuentra un bloque de memoria adecuado, el sistema operativo verifica si la memoria está disponible para su uso.
4. *Inicialización de memoria*: Si la memoria está disponible, el sistema operativo inicializa el bloque de memoria con ceros.
5. *Asignación de memoria*: El sistema operativo asigna el bloque de memoria al programa y devuelve un puntero a la dirección de inicio de ese bloque.
6. *Actualización de la tabla de asignación de memoria*: El sistema operativo actualiza la tabla de asignación de memoria para reflejar la nueva asignación de memoria.

Diferencias entre `malloc`*,* `realloc` *y* `calloc`

Aquí hay una tabla que resume las diferencias entre `malloc`, `realloc` y `calloc`:

Función	**Descripción**	**Argumentos**	**Valor devuelto**
`malloc`	Solicita memoria dinámica	Tamaño de memoria	Puntero a la dirección de inicio de la memoria
`realloc`	Cambia el tamaño de un bloque de memoria	Puntero al bloque de memoria original, nuevo tamaño de memoria	Puntero a la dirección de inicio del nuevo bloque de memoria
`calloc`	Solicita memoria dinámica y la inicializa con ceros	Número de elementos, tamaño de cada elemento	Puntero a la dirección de inicio de la memoria

Ejercicio de malloc calloc realloc

¡Claro! Aquí te presento un ejercicio complejo que muestra el funcionamiento de `malloc`, `realloc` y `calloc`.

Ejercicio:

Crear un programa que permita gestionar una lista de empleados, donde cada empleado tenga un nombre, un apellido, un edad y un salario. El programa debe permitir agregar empleados a la lista, eliminar empleados de la lista, mostrar la lista de empleados y calcular el salario total de todos los empleados.

Código:

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

// Definición de la estructura Empleado
typedef struct Empleado {
    char nombre[20];
    char apellido[20];
    int edad;
    float salario;
} Empleado;

// Función para agregar un empleado a la lista
void agregarEmpleado(Empleado** lista, int* tamano) {
    // Solicitar memoria para el nuevo empleado
    Empleado* nuevoEmpleado = malloc(sizeof(Empleado));

    // Pedir datos del empleado
    printf("Ingrese el nombre del empleado: ");
    fgets(nuevoEmpleado->nombre, 20, stdin);
    printf("Ingrese el apellido del empleado: ");
    fgets(nuevoEmpleado->apellido, 20, stdin);
    printf("Ingrese la edad del empleado: ");
    scanf("%d", &nuevoEmpleado->edad);
    printf("Ingrese el salario del empleado: ");
    scanf("%f", &nuevoEmpleado->salario);
```

```
    // Agregar el empleado a la lista
    if (*tamano == 0) {
        *lista = nuevoEmpleado;
    } else {
        // Reasignar memoria para la lista
        *lista = realloc(*lista, (*tamano + 1) *
sizeof(Empleado));
        // Agregar el empleado al final de la lista
        (*lista)[*tamano] = *nuevoEmpleado;
    }
    (*tamano)++;
}

// Función para eliminar un empleado de la lista
void eliminarEmpleado(Empleado** lista, int* tamano, int indice)
{
    if (indice < 0 || indice >= *tamano) {
        printf("Índice inválido\n");
        return;
    }

    // Reasignar memoria para la lista
    Empleado* nuevaLista = malloc((*tamano - 1) *
sizeof(Empleado));
    for (int i = 0; i < indice; i++) {
        nuevaLista[i] = (*lista)[i];
    }
    for (int i = indice + 1; i < *tamano; i++) {
        nuevaLista[i - 1] = (*lista)[i];
    }

    // Liberar memoria del empleado eliminado
    free(*lista);
    *lista = nuevaLista;
    (*tamano)--;
}

// Función para mostrar la lista de empleados
void mostrarEmpleados(Empleado* lista, int tamano) {
    for (int i = 0; i < tamano; i++) {
        printf("Nombre: %s\n", lista[i].nombre);
        printf("Apellido: %s\n", lista[i].apellido);
```

```
        printf("Edad: %d\n", lista[i].edad);
        printf("Salario: %.2f\n\n", lista[i].salario);
    }
}

// Función para calcular el salario total de todos los empleados
float calcularSalarioTotal(Empleado* lista, int tamano) {
    float salarioTotal = 0;
    for (int i = 0; i < tamano; i++) {
        salarioTotal += lista[i].salario;
    }
    return salarioTotal;
}

int main() {
    Empleado* lista = NULL;
    int tamano = 0;

    // Agregar empleados a la lista
    agregarEmpleado(&lista, &tamano);
    agregarEmpleado(&lista, &tamano);
    agregarEmpleado(&lista, &tamano);

    // Mostrar la lista de empleados
    printf("Lista de empleados:\n");
    mostrarEmpleados(lista, tamano);

    // Calcular el salario total de todos los empleados
    float salarioTotal = calcularSalarioTotal(lista, tamano);
    printf("Salario total: %.2f\n", salarioTotal);

    // Eliminar un empleado de la lista
    eliminarEmpleado(&lista, &tamano, 1);

    // Mostrar la lista de empleados después de eliminar uno
    printf("Lista de empleados después de eliminar uno:\n");
    mostrarEmpleados(lista, tamano);

    // Utilizar calloc para crear una lista de empleados

// Utilizar calloc para crear una lista de empleados
    Empleado* listaCalloc = calloc(5, sizeof(Empleado));
```

```
    if (listaCalloc == NULL) {
        printf("Error al asignar memoria\n");
        return 1;
    }

    // Inicializar la lista de empleados con datos
    for (int i = 0; i < 5; i++) {
        sprintf(listaCalloc[i].nombre, "Empleado %d", i);
        sprintf(listaCalloc[i].apellido, "Apellido %d", i);
        listaCalloc[i].edad = i * 10;
        listaCalloc[i].salario = i * 1000;
    }

    // Mostrar la lista de empleados creada con calloc
    printf("Lista de empleados creada con calloc:\n");
    mostrarEmpleados(listaCalloc, 5);

    // Liberar la memoria asignada con malloc y calloc
    free(lista);
    free(listaCalloc);

    return 0;
}
```

Explicación del código:

- La función `agregarEmpleado` solicita memoria para un nuevo empleado utilizando `malloc` y lo agrega a la lista de empleados.
- La función `eliminarEmpleado` elimina un empleado de la lista de empleados y reasigna la memoria utilizando `realloc`.
- La función `mostrarEmpleados` muestra la lista de empleados.
- La función `calcularSalarioTotal` calcula el salario total de todos los empleados.
- En la función `main`, se crea una lista de empleados utilizando `malloc` y se agregan empleados a la lista.
- Luego, se muestra la lista de empleados y se calcula el salario total.

- Después, se elimina un empleado de la lista y se muestra la lista de empleados actualizada.
- Finalmente, se utiliza `calloc` para crear una lista de empleados y se muestra la lista de empleados creada con `calloc`.
- Al final, se libera la memoria asignada con `malloc` y `calloc` utilizando `free`.

Espero que este ejemplo te haya ayudado a entender cómo se utilizan `malloc`, `realloc` y `calloc` en un programa en C. ¡Si tienes alguna pregunta o necesitas más ayuda, no dudes en preguntar!

Liberar memoria Free

La función `free` en C es una de las funciones más importantes y fundamentales en la gestión de memoria dinámica. En este explicación, te voy a mostrar cómo funciona, cómo se usa y para qué sirve `free` en C, de manera detallada y a bajo nivel.

¿Qué es la memoria dinámica?

La memoria dinámica es un tipo de memoria que se asigna y se libera durante la ejecución de un programa. A diferencia de la memoria estática, que se asigna en tiempo de compilación, la memoria dinámica se asigna en tiempo de ejecución. Esto significa que el programa puede solicitar más memoria o liberar memoria que ya no necesita durante su ejecución.

¿Cómo se asigna la memoria dinámica?

La memoria dinámica se asigna mediante la función `malloc` (o `calloc` o `realloc`, que son variantes de `malloc`). Cuando se llama a `malloc`, el sistema operativo busca un bloque de memoria disponible que tenga el tamaño solicitado

y lo asigna al programa. El programa puede entonces utilizar esta memoria para almacenar datos.

¿Qué es `free`*?*

`free` es una función que se utiliza para liberar la memoria dinámica que ya no se necesita. Cuando se llama a `free`, el sistema operativo marca el bloque de memoria como disponible para ser reutilizado por otros programas o por el mismo programa.

¿Cómo funciona `free`*?*

Cuando se llama a `free`, el sistema operativo realiza los siguientes pasos:

1. *Buscar el bloque de memoria*: El sistema operativo busca el bloque de memoria que se quiere liberar. Esto se hace mediante la dirección de memoria que se pasó como argumento a `free`.
2. *Verificar la validez del bloque*: El sistema operativo verifica si el bloque de memoria es válido y si pertenece al programa que lo está liberando.
3. *Marcar el bloque como disponible*: Si el bloque es válido, el sistema operativo marca el bloque como disponible para ser reutilizado.
4. *Actualizar la lista de bloques libres*: El sistema operativo actualiza la lista de bloques libres para que otros programas o el mismo programa puedan encontrar y utilizar el bloque liberado.

¿Cómo se usa `free`*?*

`free` se usa de la siguiente manera:

```
void* ptr = malloc(tamaño);
// Utilizar la memoria
free(ptr);
```

Donde `ptr` es la dirección de memoria que se quiere liberar y `tamaño` es el tamaño de la memoria que se asignó con `malloc`.

¿Para qué sirve `free`*?*

`free` sirve para liberar la memoria dinámica que ya no se necesita. Esto es importante por varias razones:

- *Evitar fugas de memoria*: Si no se libera la memoria que ya no se necesita, el programa puede consumir cada vez más memoria, lo que puede llevar a problemas de rendimiento y estabilidad.
- *Reutilizar la memoria*: Al liberar la memoria, se permite que otros programas o el mismo programa la reutilicen, lo que puede mejorar la eficiencia del sistema.
- *Prevenir errores de memoria*: Si no se libera la memoria que ya no se necesita, se pueden producir errores de memoria, como intentar acceder a memoria que ya no existe.

Consejos y advertencias

- *No se puede liberar la misma memoria dos veces*: Si se intenta liberar la misma memoria dos veces, se puede producir un error de memoria.
- *No se puede liberar memoria que no se asignó con* `malloc`: Si se intenta liberar memoria que no se asignó con `malloc`, se puede producir un error de memoria.
- *Es importante verificar si la memoria se asignó correctamente*: Antes de utilizar la memoria, es importante verificar si se asignó correctamente con `malloc`.

`free` es una función fundamental en la gestión de memoria dinámica que se utiliza para liberar la memoria que ya no se necesita. Es importante utilizar `free` correctamente para evitar fugas de memoria, reutilizar la memoria y prevenir errores de memoria.

Ejercicio FT*get*next*line*

El proyecto `ft_get_next_line` es un proyecto clásico en la escuela 42 que consiste en crear una función que lee una línea de un archivo y la devuelve como una cadena de caracteres. A continuación, te voy a mostrar cómo se puede implementar el proyecto `ft_get_next_line` utilizando solo las funciones `open`, `write`, `malloc` y `free`, y creando las funciones auxiliares necesarias.

ft_get_next_line.c

```
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

#define BUFFER_SIZE 1024

char *ft_strjoin(char *s1, char *s2);
char *ft_strchr(char *s, char c);
char *ft_strdup(char *s);
int ft_strlen(char *s);

int ft_get_next_line(int fd, char **line)
{
 char *buffer = malloc(BUFFER_SIZE + 1);
 char *tmp = NULL;
 int ret = 0;
 int len = 0;

 if (fd < 0 || !line || BUFFER_SIZE <= 0)
 return -1;

 while ((ret = read(fd, buffer, BUFFER_SIZE)) > 0)
 {
 buffer[ret] = '\0';
 if (tmp == NULL)
 tmp = ft_strdup(buffer);
 else
 tmp = ft_strjoin(tmp, buffer);
 if (ft_strchr(tmp, '\n') != NULL)
 break;
```

```
 }
 if (ret == -1)
 {
 free(buffer);
 return -1;
 }
 *line = ft_strdup(tmp);
 free(tmp);
 free(buffer);
 return 1;
}

char *ft_strjoin(char *s1, char *s2)
{
 char *new = malloc(ft_strlen(s1) + ft_strlen(s2) + 1);
 int i = 0;
 int j = 0;

 while (s1[i] != '\0')
 {
 new[i] = s1[i];
 i++;
 }
 while (s2[j] != '\0')
 {
 new[i + j] = s2[j];
 j++;
 }
 new[i + j] = '\0';
 return new;
}

char *ft_strchr(char *s, char c)
{
 int i = 0;

 while (s[i] != '\0')
 {
 if (s[i] == c)
 return &s[i];
 i++;
 }
```

```
 return NULL;
}

char *ft_strdup(char *s)
{
 char *new = malloc(ft_strlen(s) + 1);
 int i = 0;

 while (s[i] != '\0')
 {
 new[i] = s[i];
 i++;
 }
 new[i] = '\0';
 return new;
}

int ft_strlen(char *s)
{
 int i = 0;

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

Explicación

La función `ft_get_next_line` toma como argumentos un descriptor de archivo `fd` y un puntero a un puntero de caracteres `line`. La función devuelve un entero que indica si se leyó una línea correctamente (1) o si hubo un error (-1).

La función utiliza un buffer de tamaño `BUFFER_SIZE` para leer el archivo. Si el archivo se abre correctamente, la función lee el archivo en bloques de tamaño `BUFFER_SIZE` y los concatena en una cadena de caracteres utilizando la función `ft_strjoin`. Si se encuentra un carácter de nueva línea (`\n`) en la cadena, la función devuelve la línea leída.

Las funciones auxiliares `ft_strjoin`, `ft_strchr`, `ft_strdup` y `ft_strlen` se utilizan para manipular las cadenas de caracteres.

- `ft_strjoin`: concatena dos cadenas de caracteres.
- `ft_strchr`: busca un carácter en una cadena de caracteres.
- `ft_strdup`: crea una copia de una cadena de caracteres.
- `ft_strlen`: devuelve la longitud de una cadena de caracteres.

Uso

Para utilizar la función `ft_get_next_line`, se debe abrir un archivo y pasar el descriptor de archivo y un puntero a un puntero de caracteres a la función. La función devuelve un entero que indica si se leyó una línea correctamente.

```
int main()
{
 int fd = open("archivo.txt", O_RDONLY);
 char *line = NULL;
 int ret = 0;

 while ((ret = ft_get_next_line(fd, &line)) > 0)
 {
 printf("%s\n", line);
 free(line);
 }
 close(fd);
 return 0;
}
```

Espero que esto te ayude a comprender cómo se pueden implementar los free correctamente y que ocurre si no lo haces

Array

¡Claro! Los arrays son una estructura de datos fundamental en la programación en C. Un array es una colección de elementos del mismo tipo que se almacenan en memoria de manera contigua.

Definición de un array

Un array se define mediante el tipo de dato y el número de elementos que contendrá. La sintaxis para definir un array es la siguiente:

```
tipo nombre_array[tamaño];
```

- `tipo`: es el tipo de dato que contendrá el array (por ejemplo, `int`, `char`, `float`, etc.).
- `nombre_array`: es el nombre que se le da al array.
- `tamaño`: es el número de elementos que contendrá el array.

Ejemplos de arrays

A continuación, te presento algunos ejemplos de arrays:

```
// Array de enteros
int numeros[5];

// Array de caracteres
char letras[10];

// Array de flotantes
float temperaturas[3];
```

Acceso a los elementos de un array

Para acceder a un elemento de un array, se utiliza el índice del elemento. El índice es un número entero que indica la posición del elemento en el array. La sintaxis para acceder a un elemento de un array es la siguiente:

```
nombre_array[índice]
```

- `nombre_array`: es el nombre del array.
- `índice`: es el número entero que indica la posición del elemento en el array.

Ejemplos de acceso a elementos de un array

A continuación, te presento algunos ejemplos de acceso a elementos de un array:

```
// Acceder al primer elemento del array
int primer_elemento = numeros[0];

// Acceder al tercer elemento del array
char tercer_letra = letras[2];

// Acceder al último elemento del array
float ultima_temperatura = temperaturas[2];
```

Inicialización de un array

Un array se puede inicializar con valores cuando se define. La sintaxis para inicializar un array es la siguiente:

```
tipo nombre_array[tamaño] = {valor1, valor2, ..., valorN};
```

- `tipo`: es el tipo de dato que contendrá el array.
- `nombre_array`: es el nombre que se le da al array.
- `tamaño`: es el número de elementos que contendrá el array.
- `valor1`, `valor2`, ..., `valorN`: son los valores que se asignan a cada elemento del array.

Ejemplos de inicialización de un array

A continuación, te presento algunos ejemplos de inicialización de un array:

```
// Inicializar un array de enteros
int numeros[5] = {1, 2, 3, 4, 5};

// Inicializar un array de caracteres
char letras[10] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i',
'j'};

// Inicializar un array de flotantes
float temperaturas[3] = {20.5, 25.8, 30.2};
```

Operaciones con arrays

Los arrays se pueden utilizar en operaciones aritméticas y lógicas. A continuación, te presento algunos ejemplos de operaciones con arrays:

```
// Sumar dos arrays
int suma[5];
for (int i = 0; i < 5; i++) {
 suma[i] = numeros[i] + otros_numeros[i];
}

// Comparar dos arrays
int iguales = 1;
for (int i = 0; i < 5; i++) {
 if (numeros[i] != otros_numeros[i]) {
 iguales = 0;
 break;
 }
}
```

Espero que esta explicación te haya ayudado a entender los arrays en C.

desplazando por el Array

hay dos formas de moverse por arrays: utilizando subíndices y utilizando punteros. A continuación, te explicaré cómo hacerlo de ambas formas.

Moverse por arrays utilizando subíndices

Un subíndice es un número que se utiliza para acceder a un elemento específico de un array. En C, los subíndices se escriben entre corchetes `[]` después del nombre del array.

Por ejemplo, si tenemos un array `int arreglo[5] = {1, 2, 3, 4, 5}`, podemos acceder a cada elemento de la siguiente manera:

- `arreglo[0]` accede al primer elemento (1)
- `arreglo[1]` accede al segundo elemento (2)
- `arreglo[2]` accede al tercer elemento (3)
- `arreglo[3]` accede al cuarto elemento (4)
- `arreglo[4]` accede al quinto elemento (5)

Para moverse por el array, simplemente cambiamos el valor del subíndice. Por ejemplo, si queremos acceder al siguiente elemento, simplemente incrementamos el subíndice en 1:

- `arreglo[0]` -> `arreglo[1]`
- `arreglo[1]` -> `arreglo[2]`
- `arreglo[2]` -> `arreglo[3]`
- `arreglo[3]` -> `arreglo[4]`

Moverse por arrays utilizando punteros

Un puntero es una variable que almacena la dirección de memoria de otra variable. En C, podemos utilizar punteros para acceder a los elementos de un array.

Por ejemplo, si tenemos un array `int arreglo[5] = {1, 2, 3, 4, 5}`, podemos declarar un puntero `int *p` que apunte al primer elemento del array:

- `int *p = arreglo;`

Ahora, podemos acceder a cada elemento del array utilizando el puntero `p`:

- `*p` accede al primer elemento (1)
- `*(p + 1)` accede al segundo elemento (2)
- `*(p + 2)` accede al tercer elemento (3)
- `*(p + 3)` accede al cuarto elemento (4)
- `*(p + 4)` accede al quinto elemento (5)

Para moverse por el array, simplemente incrementamos el puntero en 1:

- `p -> p + 1`
- `p + 1 -> p + 2`
- `p + 2 -> p + 3`
- `p + 3 -> p + 4`

Ejemplos de código

A continuación, te presento algunos ejemplos de código que ilustran cómo moverse por arrays utilizando subíndices y punteros:

```
#include <stdio.h>

int main() {
 int arreglo[5] = {1, 2, 3, 4, 5};

 // Moverse por el array utilizando subíndices
 for (int i = 0; i < 5; i++) {
 printf("%d ", arreglo[i]);
 }

 printf("\n");

 // Moverse por el array utilizando punteros
 int *p = arreglo;
 for (int i = 0; i < 5; i++) {
 printf("%d ", *p);
 p++;
 }
```

```
 return 0;
}
```

Conclusión

En resumen, hay dos formas de moverse por arrays en C: utilizando subíndices y utilizando punteros. Ambas formas son válidas y pueden ser utilizadas dependiendo de la situación. Los subíndices son más fáciles de leer y escribir, mientras que los punteros ofrecen más flexibilidad y control sobre la memoria.

Castear variables

En C, el casting es un proceso que permite convertir una variable de un tipo de dato a otro tipo de dato. Esto se hace utilizando el operador de casting, que es un paréntesis que rodea el tipo de dato al que se quiere convertir la variable.

Sintaxis del casting

La sintaxis del casting es la siguiente:

```
(tipo_de_dato) variable;
```

- `tipo_de_dato`: es el tipo de dato al que se quiere convertir la variable.
- `variable`: es la variable que se quiere convertir.

Ejemplos de casting

A continuación, te presento algunos ejemplos de casting:

```
// Convertir un entero a un flotante
int x = 10;
float y = (float) x;

// Convertir un caracter a un entero
char c = 'A';
int d = (int) c;
```

```
// Convertir un flotante a un entero
float z = 10.5;
int w = (int) z;
```

Tipos de casting

Existen dos tipos de casting en C:

- *Casting implícito*: es cuando el compilador realiza la conversión de tipos de dato automáticamente, sin necesidad de utilizar el operador de casting.
- *Casting explícito*: es cuando el programador utiliza el operador de casting para realizar la conversión de tipos de dato.

Ejemplos de casting implícito

A continuación, te presento algunos ejemplos de casting implícito:

```
// Asignar un entero a un flotante
int x = 10;
float y = x;

// Asignar un caracter a un entero
char c = 'A';
int d = c;
```

En estos ejemplos, el compilador realiza la conversión de tipos de dato automáticamente, sin necesidad de utilizar el operador de casting.

Momentos en que se debe usar el casting

El casting se debe usar en los siguientes momentos:

- *Cuando se quiere realizar una operación aritmética entre variables de diferentes tipos de dato*: en este caso, se debe utilizar el casting para convertir las variables a un tipo de dato común.

- *Cuando se quiere asignar el valor de una variable de un tipo de dato a una variable de otro tipo de dato*: en este caso, se debe utilizar el casting para convertir la variable a el tipo de dato de la variable que se quiere asignar.
- *Cuando se quiere utilizar una función que requiere un tipo de dato específico*: en este caso, se debe utilizar el casting para convertir la variable a el tipo de dato que requiere la función.

Ejemplos de uso del casting en funciones

A continuación, te presento algunos ejemplos de uso del casting en funciones:

```
// Función que requiere un flotante
void imprimir_flotante(float x) {
 printf("%f\n", x);
}

int main() {
 int y = 10;
 // Utilizar el casting para convertir la variable a un flotante
 imprimir_flotante((float) y);
 return 0;
}
```

En este ejemplo, se utiliza el casting para convertir la variable `y` a un flotante, que es el tipo de dato que requiere la función `imprimir_flotante`.

Precauciones al utilizar el casting

Al utilizar el casting, se deben tener en cuenta las siguientes precauciones:

- *No se debe utilizar el casting para convertir un tipo de dato a otro tipo de dato que no sea compatible*: esto puede causar errores en el programa.

- *Se debe tener cuidado al utilizar el casting con variables de tipo puntero*: esto puede causar errores en el programa si no se realiza correctamente.

Punteros

Los punteros son una herramienta fundamental en la programación en C y otros lenguajes de bajo nivel. A continuación, te explicaré los punteros y su uso a muy bajo nivel.

¿Qué es un puntero?

Un puntero es una variable que almacena la dirección de memoria de otra variable. En otras palabras, un puntero es un apuntador a una ubicación en la memoria del ordenador.

Representación de un puntero

Un puntero se representa mediante un asterisco (`*`) seguido del nombre de la variable que apunta. Por ejemplo:

```
int x = 10;
int *p = &x;
```

En este ejemplo, `p` es un puntero que apunta a la variable `x`. La dirección de memoria de `x` se almacena en la variable `p`.

Tipos de punteros

Existen varios tipos de punteros, incluyendo:

- *Punteros a variables*: Estos son los punteros que apuntan a variables.
- *Punteros a constantes*: Estos son los punteros que apuntan a constantes.
- *Punteros a funciones*: Estos son los punteros que apuntan a funciones.

- *Punteros a estructuras*: Estos son los punteros que apuntan a estructuras.

Operadores de punteros

Existen varios operadores que se utilizan con punteros, incluyendo:

- *Operador de dirección* (`&`): Se utiliza para obtener la dirección de memoria de una variable.
- *Operador de indirección* (`*`): Se utiliza para obtener el valor de la variable que apunta un puntero.
- *Operador de incremento* (++): Se utiliza para incrementar la dirección de memoria de un puntero.
- *Operador de decremento* (−−): Se utiliza para decrementar la dirección de memoria de un puntero.

Uso de punteros

Los punteros se utilizan en una variedad de situaciones, incluyendo:

- *Pasar variables a funciones*: Los punteros se utilizan para pasar variables a funciones sin tener que copiar el valor de la variable.
- *Devolver valores de funciones*: Los punteros se utilizan para devolver valores de funciones sin tener que copiar el valor de la variable.
- *Acceder a estructuras*: Los punteros se utilizan para acceder a estructuras y modificar sus miembros.
- *Crear matrices dinámicas*: Los punteros se utilizan para crear matrices dinámicas que pueden crecer o decrecer en tamaño durante la ejecución del programa.

Ejemplos de uso de punteros

A continuación, te presento algunos ejemplos de uso de punteros:

```
// Ejemplo 1: Pasar una variable a una función utilizando un
puntero
void imprimir(int *p) {
 printf("%d\n", *p);
}

int main() {
 int x = 10;
 imprimir(&x);
 return 0;
}

// Ejemplo 2: Devolver un valor de una función utilizando un
puntero
int *devolver_puntero() {
 int x = 10;
 return &x;
}

int main() {
 int *p = devolver_puntero();
 printf("%d\n", *p);
 return 0;
}

// Ejemplo 3: Acceder a una estructura utilizando un puntero
typedef struct {
 int x;
 int y;
} Punto;

void imprimir_punto(Punto *p) {
 printf("(%d, %d)\n", p->x, p->y);
}

int main() {
 Punto p = {10, 20};
 imprimir_punto(&p);
 return 0;
}
```

Precauciones al utilizar punteros

Al utilizar punteros, se deben tener en cuenta las siguientes precauciones:

- *No se debe utilizar un puntero sin inicializar*: Esto puede causar errores en el programa.
- *No se debe utilizar un puntero que apunte a una variable que ya no existe*: Esto puede causar errores en el programa.
- *Se debe tener cuidado al utilizar punteros con matrices*: Esto puede causar errores en el programa si no se realiza correctamente.

Espero que esta explicación te haya ayudado a entender los punteros y su uso a muy bajo nivel.

ejercicio de punteros

Aquí te presento un ejercicio completo que muestra el funcionamiento de los punteros en C, incluyendo el uso de punteros a punteros y subíndices.

Ejercicio:

Crear un programa que permita gestionar una lista de estudiantes, donde cada estudiante tenga un nombre, una edad y un promedio de notas. El programa debe permitir agregar estudiantes a la lista, eliminar estudiantes de la lista y mostrar la lista de estudiantes.

Código:

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

// Definición de la estructura Estudiante
typedef struct Estudiante {
    char nombre[20];
    int edad;
```

```
    float promedio;
} Estudiante;

// Definición de la estructura ListaEstudiantes
typedef struct ListaEstudiantes {
    Estudiante* estudiantes;
    int tamano;
} ListaEstudiantes;

// Función para agregar un estudiante a la lista
void agregarEstudiante(ListaEstudiantes* lista, char* nombre, int
edad, float promedio) {
    Estudiante* nuevoEstudiante = malloc(sizeof(Estudiante));
    strcpy(nuevoEstudiante->nombre, nombre);
    nuevoEstudiante->edad = edad;
    nuevoEstudiante->promedio = promedio;

    lista->estudiantes = realloc(lista->estudiantes, (lista-
>tamano + 1) * sizeof(Estudiante));
    lista->estudiantes[lista->tamano] = *nuevoEstudiante;
    lista->tamano++;
}

// Función para eliminar un estudiante de la lista
void eliminarEstudiante(ListaEstudiantes* lista, int indice) {
    if (indice < 0 || indice >= lista->tamano) {
        printf("Índice inválido\n");
        return;
    }

    Estudiante* nuevoEstudiantes = malloc((lista->tamano - 1) *
sizeof(Estudiante));
    for (int i = 0; i < indice; i++) {
        nuevoEstudiantes[i] = lista->estudiantes[i];
    }
    for (int i = indice + 1; i < lista->tamano; i++) {
        nuevoEstudiantes[i - 1] = lista->estudiantes[i];
    }

    free(lista->estudiantes);
    lista->estudiantes = nuevoEstudiantes;
    lista->tamano--;
```

```
}

// Función para mostrar la lista de estudiantes
void mostrarEstudiantes(ListaEstudiantes* lista) {
    for (int i = 0; i < lista->tamano; i++) {
        printf("Nombre: %s\n", lista->estudiantes[i].nombre);
        printf("Edad: %d\n", lista->estudiantes[i].edad);
        printf("Promedio: %.2f\n\n", lista-
>estudiantes[i].promedio);
    }
}

int main() {
    ListaEstudiantes lista;
    lista.estudiantes = NULL;
    lista.tamano = 0;

    // Agregar estudiantes a la lista
    agregarEstudiante(&lista, "Juan", 20, 8.5);
    agregarEstudiante(&lista, "María", 22, 9.0);
    agregarEstudiante(&lista, "Pedro", 21, 8.0);

    // Mostrar la lista de estudiantes
    printf("Lista de estudiantes:\n");
    mostrarEstudiantes(&lista);

    // Eliminar un estudiante de la lista
    eliminarEstudiante(&lista, 1);

    // Mostrar la lista de estudiantes después de eliminar uno
    printf("Lista de estudiantes después de eliminar uno:\n");
    mostrarEstudiantes(&lista);

    // Utilizar punteros a punteros para acceder a un estudiante
    Estudiante** punteroAPuntero = &lista.estudiantes;
    printf("Nombre del primer estudiante: %s\n",
(*punteroAPuntero)->nombre);

    // Utilizar subíndices para acceder a un estudiante
    printf("Nombre del segundo estudiante: %s\n",
lista.estudiantes[1].nombre);
```

```
    return 0;
}
```

Explicación del código:

- La estructura `Estudiante` tiene tres campos: `nombre`, `edad` y `promedio`.
- La estructura `ListaEstudiantes` tiene dos campos: `estudiantes` y `tamano`. El campo `estudiantes` es un puntero a un arreglo de estructuras `Estudiante`, y el campo `tamano` es el número de estudiantes en la lista.
- La función `agregarEstudiante` agrega un nuevo estudiante a la lista. Primero, crea un nuevo estudiante con la información proporcionada y luego lo agrega al final de la lista.
- La función `eliminarEstudiante` elimina un estudiante de la lista. Primero, verifica si el índice es válido y luego elimina el estudiante de la lista.
- La función `mostrarEstudiantes` muestra la lista de estudiantes. Recorre la lista de estudiantes y imprime la información de cada estudiante.
- En la función `main`, se crea una lista de estudiantes y se agregan algunos estudiantes. Luego, se muestra la lista de estudiantes y se elimina un estudiante. Finalmente, se muestra la lista de estudiantes después de eliminar uno.
- Se utiliza un puntero a puntero para acceder a un estudiante y se utiliza un subíndice para acceder a un estudiante.

Espero que este ejemplo te haya ayudado a entender cómo funcionan los punteros en C y cómo se pueden utilizar para crear estructuras de datos complejas.

Estructuras o Listas

En C, las listas y estructuras son fundamentales para almacenar y manipular datos de manera eficiente. Aquí te presento algunos ejemplos avanzados de cómo utilizar listas y estructuras en C:

Estructuras

Una estructura es un tipo de dato que permite almacenar varios valores de diferentes tipos en una sola variable. Por ejemplo:

```
typedef struct {
    int edad;
    char nombre[20];
    float altura;
} Persona;
```

En este ejemplo, definimos una estructura llamada `Persona` que tiene tres campos: `edad`, `nombre` y `altura`. Podemos crear una variable de tipo `Persona` y acceder a sus campos de la siguiente manera:

```
Persona persona;
persona.edad = 25;
strcpy(persona.nombre, "Juan");
persona.altura = 1.75;
```

Listas enlazadas

Una lista enlazada es una estructura de datos que consiste en una secuencia de nodos, cada uno de los cuales contiene un valor y un puntero al siguiente nodo en la lista. Aquí te muestro un ejemplo de cómo implementar una lista enlazada en C:

```
typedef struct Nodo {
    int valor;
    struct Nodo* siguiente;
} Nodo;
```

```
typedef struct Lista {
    Nodo* cabeza;
} Lista;
```

En este ejemplo, definimos una estructura llamada `Nodo` que tiene dos campos: `valor` y `siguiente`. El campo `siguiente` es un puntero al siguiente nodo en la lista. También definimos una estructura llamada `Lista` que tiene un campo `cabeza` que es un puntero al primer nodo en la lista.

Podemos crear una lista enlazada y agregar nodos de la siguiente manera:

```
Lista lista;
lista.cabeza = NULL;

Nodo* nodo1 = malloc(sizeof(Nodo));
nodo1->valor = 1;
nodo1->siguiente = NULL;
lista.cabeza = nodo1;

Nodo* nodo2 = malloc(sizeof(Nodo));
nodo2->valor = 2;
nodo2->siguiente = NULL;
nodo1->siguiente = nodo2;
```

Listas doblemente enlazadas

Una lista doblemente enlazada es similar a una lista enlazada, pero cada nodo tiene dos punteros: uno al siguiente nodo y otro al nodo anterior. Aquí te muestro un ejemplo de cómo implementar una lista doblemente enlazada en C:

```
typedef struct Nodo {
    int valor;
    struct Nodo* siguiente;
    struct Nodo* anterior;
} Nodo;

typedef struct Lista {
    Nodo* cabeza;
    Nodo* cola;
```

```
} Lista;
```

En este ejemplo, definimos una estructura llamada `Nodo` que tiene tres campos: `valor`, `siguiente` y `anterior`. El campo `siguiente` es un puntero al siguiente nodo en la lista, y el campo `anterior` es un puntero al nodo anterior en la lista. También definimos una estructura llamada `Lista` que tiene dos campos: `cabeza` y `cola`, que son punteros al primer y último nodo en la lista, respectivamente.

Podemos crear una lista doblemente enlazada y agregar nodos de la siguiente manera:

```
Lista lista;
lista.cabeza = NULL;
lista.cola = NULL;

Nodo* nodo1 = malloc(sizeof(Nodo));
nodo1->valor = 1;
nodo1->siguiente = NULL;
nodo1->anterior = NULL;
lista.cabeza = nodo1;
lista.cola = nodo1;

Nodo* nodo2 = malloc(sizeof(Nodo));
nodo2->valor = 2;
nodo2->siguiente = NULL;
nodo2->anterior = nodo1;
nodo1->siguiente = nodo2;
lista.cola = nodo2;
```

Árboles binarios

Un árbol binario es una estructura de datos que consiste en nodos que tienen un valor y dos hijos: izquierdo y derecho. Aquí te muestro un ejemplo de cómo implementar un árbol binario en C:

```
typedef struct Nodo {
    int valor;
    struct Nodo* izquierdo;
    struct Nodo* derecho;
```

```
} Nodo;
```

En este ejemplo, definimos una estructura llamada `Nodo` que tiene tres campos: `valor`, `izquierdo` y `derecho`. El campo `izquierdo` es un puntero al hijo izquierdo del nodo, y el campo `derecho` es un puntero al hijo derecho del nodo.

Podemos crear un árbol binario y agregar nodos de la siguiente manera:

```
Nodo* raiz = malloc(sizeof(Nodo));
raiz->valor = 1;
raiz->izquierdo = NULL;
raiz->derecho = NULL;

Nodo* nodo2 = malloc(sizeof(Nodo));
nodo2->valor = 2;
nodo2->izquierdo = NULL;
nodo2->derecho = NULL;
raiz->izquierdo = nodo2;

Nodo* nodo3 = malloc(sizeof(Nodo));
nodo3->valor = 3;
nodo3->izquierdo = NULL;
nodo3->derecho = NULL;
raiz->derecho = nodo3;
```

Espero que estos ejemplos te hayan ayudado a entender cómo utilizar listas y estructuras en C de manera avanzada.

ejercicio estructuras

Aquí te presento un ejercicio completo que utiliza estructuras para crear un sistema de gestión de libros en una biblioteca.

Ejercicio:

Crear un programa que permita gestionar libros en una biblioteca. El programa debe tener las siguientes funcionalidades:

- Agregar un libro a la biblioteca
- Eliminar un libro de la biblioteca
- Buscar un libro por título o autor
- Mostrar todos los libros de la biblioteca

Código:

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

// Definición de la estructura Libro
typedef struct Libro {
    char titulo[50];
    char autor[50];
    int año;
    struct Libro* siguiente;
} Libro;

// Definición de la estructura Biblioteca
typedef struct Biblioteca {
    Libro* cabeza;
} Biblioteca;

// Función para agregar un libro a la biblioteca
void agregarLibro(Biblioteca* biblioteca, char* titulo, char*
autor, int año) {
    Libro* nuevoLibro = malloc(sizeof(Libro));
    strcpy(nuevoLibro->titulo, titulo);
    strcpy(nuevoLibro->autor, autor);
    nuevoLibro->año = año;
    nuevoLibro->siguiente = NULL;

    if (biblioteca->cabeza == NULL) {
        biblioteca->cabeza = nuevoLibro;
    } else {
        Libro* actual = biblioteca->cabeza;
        while (actual->siguiente != NULL) {
            actual = actual->siguiente;
        }
```

```
        actual->siguiente = nuevoLibro;
    }
}

// Función para eliminar un libro de la biblioteca
void eliminarLibro(Biblioteca* biblioteca, char* titulo) {
    Libro* actual = biblioteca->cabeza;
    Libro* anterior = NULL;

    while (actual != NULL) {
        if (strcmp(actual->titulo, titulo) == 0) {
            if (anterior == NULL) {
                biblioteca->cabeza = actual->siguiente;
            } else {
                anterior->siguiente = actual->siguiente;
            }
            free(actual);
            return;
        }
        anterior = actual;
        actual = actual->siguiente;
    }
}

// Función para buscar un libro por título o autor
Libro* buscarLibro(Biblioteca* biblioteca, char* titulo, char*
autor) {
    Libro* actual = biblioteca->cabeza;

    while (actual != NULL) {
        if (strcmp(actual->titulo, titulo) == 0 || strcmp(actual-
>autor, autor) == 0) {
            return actual;
        }
        actual = actual->siguiente;
    }
    return NULL;
}

// Función para mostrar todos los libros de la biblioteca
void mostrarLibros(Biblioteca* biblioteca) {
    Libro* actual = biblioteca->cabeza;
```

```
    while (actual != NULL) {
        printf("Título: %s\n", actual->titulo);
        printf("Autor: %s\n", actual->autor);
        printf("Año: %d\n\n", actual->año);
        actual = actual->siguiente;
    }
}

int main() {
    Biblioteca biblioteca;
    biblioteca.cabeza = NULL;

    // Agregar libros a la biblioteca
    agregarLibro(&biblioteca, "El señor de los anillos", "J.R.R.
Tolkien", 1954);
    agregarLibro(&biblioteca, "Harry Potter y la piedra
filosofal", "J.K. Rowling", 1997);
    agregarLibro(&biblioteca, "El cuaderno de Maya", "Isabel
Allende", 2011);

    // Mostrar todos los libros de la biblioteca
    printf("Libros de la biblioteca:\n");
    mostrarLibros(&biblioteca);

    // Buscar un libro por título
    Libro* libro = buscarLibro(&biblioteca, "El señor de los
anillos", "");
    if (libro != NULL) {
        printf("Libro encontrado:\n");
        printf("Título: %s\n", libro->titulo);
        printf("Autor: %s\n", libro->autor);
        printf("Año: %d\n\n", libro->año);
    } else {
        printf("Libro no encontrado\n");
    }

    // Eliminar un libro de la biblioteca
    eliminarLibro(&biblioteca, "Harry Potter y la piedra
filosofal");
```

```
    // Mostrar todos los libros de la biblioteca después de
eliminar uno
    printf("Libros de la biblioteca después de eliminar uno:\n");
    mostrarLibros(&biblioteca);

    return 0;
}
```

Explicación del código:

- La estructura `Libro` tiene cuatro campos: `titulo`, `autor`, `año` y `siguiente`. El campo `siguiente` es un puntero al siguiente libro en la lista.
- La estructura `Biblioteca` tiene un campo `cabeza` que es un puntero al primer libro en la lista.
- La función `agregarLibro` agrega un nuevo libro a la biblioteca. Primero, crea un nuevo libro con la información proporcionada y luego lo agrega al final de la lista.
- La función `eliminarLibro` elimina un libro de la biblioteca. Busca el libro por título y lo elimina de la lista.
- La función `buscarLibro` busca un libro por título o autor. Recorre la lista de libros y devuelve el libro que coincide con la búsqueda.
- La función `mostrarLibros` muestra todos los libros de la biblioteca. Recorre la lista de libros y imprime la información de cada libro.
- En la función `main`, se crea una biblioteca y se agregan algunos libros. Luego, se muestra la lista de libros, se busca un libro por título y se elimina un libro. Finalmente, se muestra la lista de libros después de eliminar uno.

Espero que este ejemplo te haya ayudado a entender cómo utilizar estructuras en C para crear un sistema de gestión de libros en una biblioteca.

LIBFT

ftputnbr

La función `ft_putnbr` se utiliza para imprimir un número entero en la salida estándar. A continuación, te presento la implementación de la función `ft_putnbr`:

```
void ft_putnbr(int n)
{
 if (n < 0)
 {
 write(1, "-", 1);
 n = -n;
 }
 if (n >= 10)
 {
 ft_putnbr(n / 10);
 }
 write(1, &"0123456789"[n % 10], 1);
}
```

Esta función utiliza una técnica llamada "recursividad" para imprimir los dígitos del número entero. La función se llama a sí misma para imprimir los dígitos más significativos del número, y luego imprime el dígito menos significativo.

Aquí hay una explicación paso a paso de cómo funciona la función:

1. Si el número es negativo, imprime un signo menos y convierte el número a positivo.
2. Si el número es mayor o igual a 10, llama a sí misma con el número dividido entre 10. Esto imprime los dígitos más significativos del número.
3. Imprime el dígito menos significativo del número utilizando el arreglo de caracteres "`0123456789`".

La función `write` se utiliza para imprimir los caracteres en la salida estándar. El primer argumento es el descriptor de archivo (1 para la salida estándar), el segundo argumento es la dirección de memoria del caracter a imprimir, y el tercer argumento es el número de caracteres a imprimir.

Puedes probar la función `ft_putnbr` con el siguiente código:

```
int main()
{
 ft_putnbr(12345);
 ft_putnbr(-67890);
 return 0;
}
```

Esto debería imprimir los números `12345` y `-67890` en la salida estándar.

ftmemset

La función `ft_memset` se utiliza para establecer todos los bytes de una región de memoria a un valor determinado.

A continuación, te presento la implementación de la función `ft_memset`:

```
void *ft_memset(void *s, int c, size_t n)
{
 unsigned char *ptr = (unsigned char *)s;
 while (n--)
 {
 *ptr = (unsigned char)c;
 ptr++;
 }
 return s;
}
```

Esta función utiliza un puntero `ptr` para recorrer la región de memoria `s` y establecer cada byte a el valor `c`. El tamaño de la región de memoria se especifica mediante el parámetro `n`.

Aquí hay una explicación paso a paso de cómo funciona la función:

1. Se crea un puntero `ptr` que apunta a la región de memoria `s`. Se utiliza el tipo `unsigned char` para evitar problemas de signo.
2. Se utiliza un bucle `while` para recorrer la región de memoria. El bucle se ejecuta `n` veces, donde `n` es el tamaño de la región de memoria.
3. En cada iteración del bucle, se establece el valor del byte actual a `c`. Se utiliza el operador de asignación `*` para acceder al valor del byte.
4. Se incrementa el puntero `ptr` para apuntar al siguiente byte de la región de memoria.
5. Finalmente, se devuelve el puntero original `s` para permitir la concatenación de llamadas.

La función `ft_memset` no utiliza ninguna función externa, solo utiliza operadores y tipos de datos básicos de C.

Puedes probar la función `ft_memset` con el siguiente código:

```
int main()
{
 char str[10] = "Hola Mundo";
 ft_memset(str, 'X', 5);
 printf("%s\n", str);
 return 0;
}
```

Esto debería imprimir la cadena `"XXXXXundo"`, donde los primeros 5 caracteres han sido establecidos a `'X'`.

ftstrlcpy

La función `ft_strlcpy` se utiliza para copiar una cadena de caracteres en otra, asegurándose de que la cadena destino tenga suficiente espacio para almacenar la cadena origen.

A continuación, te presento la implementación de la función `ft_strlcpy`:

```
size_t ft_strlcpy(char *dst, const char *src, size_t size)
{
 size_t len = 0;
 while (len < size - 1 && *src != '\0')
 {
 *dst = *src;
 dst++;
 src++;
 len++;
 }
 if (len < size)
 {
 *dst = '\0';
 }
 return len + (*src != '\0');
}
```

Esta función utiliza dos punteros, `dst` y `src`, para recorrer las cadenas de caracteres destino y origen, respectivamente. El tamaño de la cadena destino se especifica mediante el parámetro `size`.

Aquí hay una explicación paso a paso de cómo funciona la función:

1. Se inicializa una variable `len` para contar el número de caracteres copiados.
2. Se utiliza un bucle `while` para recorrer la cadena origen. El bucle se ejecuta mientras `len` sea menor que `size - 1` y el caracter actual de la cadena origen no sea el carácter nulo (`'\0'`).

3. En cada iteración del bucle, se copia el caracter actual de la cadena origen en la cadena destino.
4. Se incrementan los punteros `dst` y `src` para apuntar al siguiente caracter de las cadenas destino y origen, respectivamente.
5. Se incrementa la variable `len` para contar el número de caracteres copiados.
6. Si `len` es menor que `size`, se establece el último caracter de la cadena destino a `'\0'` para asegurarse de que la cadena esté terminada correctamente.
7. Finalmente, se devuelve la longitud de la cadena origen, incluyendo el carácter nulo (`'\0'`) si no se copió completamente.

La función `ft_strlcpy` no utiliza ninguna función externa, solo utiliza operadores y tipos de datos básicos de C.

Puedes probar la función `ft_strlcpy` con el siguiente código:

```
int main()
{
 char dst[10];
 char src[] = "Hola Mundo";
 ft_strlcpy(dst, src, 10);
 printf("%s\n", dst);
 return 0;
}
```

Esto debería imprimir la cadena `"Hola Mund"`, donde se han copiado los primeros 9 caracteres de la cadena origen en la cadena destino.

ftstrlcat

La función `ft_strlcat` se utiliza para concatenar una cadena de caracteres en otra, asegurándose de que la cadena destino tenga suficiente espacio para almacenar la cadena origen.

A continuación, te presento la implementación de la función `ft_strlcat`:

```
size_t ft_strlcat(char *dst, const char *src, size_t size)
{
 size_t len_dst = 0;
 size_t len_src = 0;

 // Calcular la longitud de la cadena destino
 while (len_dst < size && dst[len_dst] != '\0')
 {
 len_dst++;
 }

 // Calcular la longitud de la cadena origen
 while (src[len_src] != '\0')
 {
 len_src++;
 }

 // Concatenar la cadena origen en la cadena destino
 if (len_dst + len_src < size)
 {
 for (size_t i = 0; i < len_src; i++)
 {
 dst[len_dst + i] = src[i];
 }
 dst[len_dst + len_src] = '\0';
 }
 else
 {
 for (size_t i = 0; i < size - len_dst - 1; i++)
 {
 dst[len_dst + i] = src[i];
 }
 dst[size - 1] = '\0';
```

```
 }

 return len_dst + len_src;
}
```

Esta función utiliza dos variables, `len_dst` y `len_src`, para contar la longitud de las cadenas destino y origen, respectivamente. El tamaño de la cadena destino se especifica mediante el parámetro `size`.

Aquí hay una explicación paso a paso de cómo funciona la función:

1. Se calcula la longitud de la cadena destino (`len_dst`) y se almacena en la variable `len_dst`.
2. Se calcula la longitud de la cadena origen (`len_src`) y se almacena en la variable `len_src`.
3. Si la suma de las longitudes de las cadenas destino y origen es menor que el tamaño de la cadena destino (`size`), se concatenan las cadenas origen y destino.
4. Si la suma de las longitudes de las cadenas destino y origen es mayor o igual que el tamaño de la cadena destino (`size`), se concatenan las cadenas origen y destino, pero se truncará la cadena origen para que quepa en el espacio disponible.
5. Finalmente, se devuelve la longitud total de la cadena destino después de la concatenación.

La función `ft_strlcat` no utiliza ninguna función externa, solo utiliza operadores y tipos de datos básicos de C.

Puedes probar la función `ft_strlcat` con el siguiente código:

```
int main()
{
 char dst[20] = "Hola ";
 char src[] = "Mundo";
 ft_strlcat(dst, src, 20);
```

```
 printf("%s\n", dst);
 return 0;
}
```

Esto debería imprimir la cadena "Hola Mundo", donde se han concatenado las cadenas destino y origen.

ftstrchr y *ftstrrchr*

Las funciones `ft_strchr`, `ft_strrchr` se utilizan para buscar un caracter en una cadena de caracteres.

ft_strchr

La función `ft_strchr` busca el primer caracter que coincide con el caracter especificado en la cadena de caracteres.

A continuación, te presento la implementación de la función `ft_strchr`:

```
char *ft_strchr(const char *s, int c)
{
 while (*s != '\0')
 {
 if (*s == (char)c)
 return (char *)s;
 s++;
 }
 if (c == '\0')
 return (char *)s;
 return NULL;
}
```

Esta función utiliza un puntero `s` para recorrer la cadena de caracteres y busca el primer caracter que coincide con el caracter especificado `c`.

Aquí hay una explicación paso a paso de cómo funciona la función:

1. Se recorre la cadena de caracteres utilizando un bucle `while`.

2. En cada iteración del bucle, se compara el caracter actual con el caracter especificado `c`.
3. Si se encuentra una coincidencia, se devuelve un puntero al caracter encontrado.
4. Si no se encuentra una coincidencia y se llega al final de la cadena de caracteres (`'\0'`), se devuelve `NULL`.
5. Si el caracter especificado es `'\0'`, se devuelve un puntero al final de la cadena de caracteres.

ft_strrchr

La función `ft_strrchr` busca el último caracter que coincide con el caracter especificado en la cadena de caracteres.

A continuación, te presento la implementación de la función `ft_strrchr`:

```
char *ft_strrchr(const char *s, int c)
{
 char *ptr = NULL;
 while (*s != '\0')
 {
 if (*s == (char)c)
 ptr = (char *)s;
 s++;
 }
 if (c == '\0')
 return (char *)s;
 return ptr;
}
```

Esta función utiliza un puntero `s` para recorrer la cadena de caracteres y busca el último caracter que coincide con el caracter especificado `c`.

Aquí hay una explicación paso a paso de cómo funciona la función:

1. Se recorre la cadena de caracteres utilizando un bucle `while`.

2. En cada iteración del bucle, se compara el caracter actual con el caracter especificado `c`.
3. Si se encuentra una coincidencia, se almacena el puntero al caracter encontrado en la variable `ptr`.
4. Si se llega al final de la cadena de caracteres (`'\0'`) y no se ha encontrado una coincidencia, se devuelve `NULL`.
5. Si el caracter especificado es `'\0'`, se devuelve un puntero al final de la cadena de caracteres.

Diferencias entre ft_strchr y ft_strrchr

Las principales diferencias entre las funciones `ft_strchr` y `ft_strrchr` son:

- `ft_strchr` busca el primer caracter que coincide con el caracter especificado, mientras que `ft_strrchr` busca el último caracter que coincide con el caracter especificado.
- `ft_strchr` devuelve un puntero al primer caracter encontrado, mientras que `ft_strrchr` devuelve un puntero al último caracter encontrado.

Puedes probar las funciones `ft_strchr` y `ft_strrchr` con el siguiente código:

```
int main()
{
 char str[] = "Hola Mundo";
 char *ptr1 = ft_strchr(str, 'o');
 char *ptr2 = ft_strrchr(str, 'o');
 printf("Primer caracter 'o' encontrado: %s\n", ptr1);
 printf("Último caracter 'o' encontrado: %s\n", ptr2);
 return 0;
}
```

Esto debería imprimir la cadena `"o Mundo"` para el primer caracter 'o' encontrado y la cadena `"o"` para el último caracter 'o' encontrado.

ftstrncmp

Las funciones `ft_strcmp` y `ft_strncmp` se utilizan para comparar dos cadenas de caracteres.

ft_strcmp

La función `ft_strcmp` compara dos cadenas de caracteres y devuelve un valor entero que indica si las cadenas son iguales o no.

A continuación, te presento la implementación de la función `ft_strcmp`:

```
int ft_strcmp(const char *s1, const char *s2)
{
 while (*s1 != '\0' && *s2 != '\0')
 {
 if (*s1 != *s2)
 return (unsigned char)*s1 - (unsigned char)*s2;
 s1++;
 s2++;
 }
 if (*s1 == '\0' && *s2 == '\0')
 return 0;
 else if (*s1 == '\0')
 return -1;
 else
 return 1;
}
```

Esta función utiliza dos punteros `s1` y `s2` para recorrer las cadenas de caracteres y comparar los caracteres uno a uno.

Aquí hay una explicación paso a paso de cómo funciona la función:

1. Se recorren las cadenas de caracteres utilizando un bucle `while`.
2. En cada iteración del bucle, se comparan los caracteres actuales de las dos cadenas.
3. Si se encuentra una diferencia, se devuelve el valor entero que indica si la primera cadena es menor, igual o mayor que la segunda cadena.

4. Si se llega al final de las cadenas de caracteres (`'\0'`) y no se ha encontrado una diferencia, se devuelve 0 para indicar que las cadenas son iguales.
5. Si la primera cadena es más corta que la segunda, se devuelve -1 para indicar que la primera cadena es menor que la segunda.
6. Si la primera cadena es más larga que la segunda, se devuelve 1 para indicar que la primera cadena es mayor que la segunda.

ft_strncmp

La función `ft_strncmp` compara dos cadenas de caracteres hasta un número determinado de caracteres y devuelve un valor entero que indica si las cadenas son iguales o no.

A continuación, te presento la implementación de la función `ft_strncmp`:

```
int ft_strncmp(const char *s1, const char *s2, size_t n)
{
 size_t i = 0;
 while (i < n && *s1 != '\0' && *s2 != '\0')
 {
 if (*s1 != *s2)
 return (unsigned char)*s1 - (unsigned char)*s2;
 s1++;
 s2++;
 i++;
 }
 if (i == n)
 return 0;
 else if (*s1 == '\0')
 return -1;
 else if (*s2 == '\0')
 return 1;
 else
 return 0;
}
```

Esta función utiliza dos punteros `s1` y `s2` para recorrer las cadenas de caracteres y comparar los caracteres uno a uno hasta un número determinado de caracteres `n`.

Aquí hay una explicación paso a paso de cómo funciona la función:

1. Se recorren las cadenas de caracteres utilizando un bucle `while`.
2. En cada iteración del bucle, se comparan los caracteres actuales de las dos cadenas.
3. Si se encuentra una diferencia, se devuelve el valor entero que indica si la primera cadena es menor, igual o mayor que la segunda cadena.
4. Si se llega al número determinado de caracteres `n` y no se ha encontrado una diferencia, se devuelve 0 para indicar que las cadenas son iguales hasta ese punto.
5. Si la primera cadena es más corta que la segunda, se devuelve -1 para indicar que la primera cadena es menor que la segunda.
6. Si la primera cadena es más larga que la segunda, se devuelve 1 para indicar que la primera cadena es mayor que la segunda.

Diferencias entre ft_strcmp y ft_strncmp

Las principales diferencias entre las funciones `ft_strcmp` y `ft_strncmp` son:

- `ft_strcmp` compara las cadenas de caracteres hasta el final (`'\0'`), mientras que `ft_strncmp` compara las cadenas de caracteres hasta un número determinado de caracteres `n`.
- `ft_strcmp` devuelve un valor entero que indica si las cadenas son iguales o no, mientras que `ft_strncmp` devuelve un valor entero que indica si las cadenas son iguales o no hasta el número determinado de caracteres `n`.

Puedes probar las funciones `ft_strcmp` y `ft_strncmp` con el siguiente código:

```
int main()
{
 char str1[] = "Hola";
 char str2[] = "Hola Mundo";
 printf("Comparación de cadenas: %d\n", ft_strcmp(str1, str2));
 printf("Comparación de cadenas hasta 5 caracteres: %d\n",
ft_strncmp(str1, str2, 5));
 return 0;
}
```

Esto debería imprimir la cadena "`Comparación de cadenas: -1`" para indicar que la primera cadena es menor que la segunda, y la cadena "`Comparación de cadenas hasta 5 caracteres: 0`" para indicar que las cadenas son iguales hasta los primeros 5 caracteres.

ft*strstr* y ft*strnstr*

Las funciones `ft_strstr` y `ft_strnstr` se utilizan para buscar una subcadena dentro de una cadena de caracteres.

ft_strstr

La función `ft_strstr` busca la primera ocurrencia de una subcadena dentro de una cadena de caracteres y devuelve un puntero a la posición donde se encuentra la subcadena.

A continuación, te presento la implementación de la función `ft_strstr`:

```
char *ft_strstr(const char *haystack, const char *needle)
{
 size_t len_needle = ft_strlen(needle);
 if (len_needle == 0)
 return (char *)haystack;
 while (*haystack != '\0')
 {
 if (ft_strncmp(haystack, needle, len_needle) == 0)
 return (char *)haystack;
 haystack++;
```

```
 }
 return NULL;
}
```

Esta función utiliza dos punteros `haystack` y `needle` para recorrer la cadena de caracteres y buscar la subcadena.

Aquí hay una explicación paso a paso de cómo funciona la función:

1. Se calcula la longitud de la subcadena `needle` utilizando la función `ft_strlen`.
2. Si la longitud de la subcadena es 0, se devuelve un puntero a la posición actual de la cadena `haystack`.
3. Se recorre la cadena de caracteres `haystack` utilizando un bucle `while`.
4. En cada iteración del bucle, se compara la subcadena `needle` con la cadena `haystack` utilizando la función `ft_strncmp`.
5. Si se encuentra una coincidencia, se devuelve un puntero a la posición donde se encuentra la subcadena.
6. Si no se encuentra una coincidencia y se llega al final de la cadena `haystack`, se devuelve `NULL`.

ft_strnstr

La función `ft_strnstr` busca la primera ocurrencia de una subcadena dentro de una cadena de caracteres hasta un número determinado de caracteres y devuelve un puntero a la posición donde se encuentra la subcadena.

A continuación, te presento la implementación de la función `ft_strnstr`:

```
char *ft_strnstr(const char *haystack, const char *needle, size_t
len)
{
 size_t len_needle = ft_strlen(needle);
 if (len_needle == 0 || len == 0)
 return (char *)haystack;
```

```
 while (len >= len_needle && *haystack != '\0')
 {
 if (ft_strncmp(haystack, needle, len_needle) == 0)
 return (char *)haystack;
 haystack++;
 len--;
 }
 return NULL;
}
```

Esta función utiliza dos punteros `haystack` y `needle` para recorrer la cadena de caracteres y buscar la subcadena hasta un número determinado de caracteres `len`.

Aquí hay una explicación paso a paso de cómo funciona la función:

1. Se calcula la longitud de la subcadena `needle` utilizando la función `ft_strlen`.
2. Si la longitud de la subcadena es 0 o el número determinado de caracteres `len` es 0, se devuelve un puntero a la posición actual de la cadena `haystack`.
3. Se recorre la cadena de caracteres `haystack` utilizando un bucle `while` hasta que se llegue al número determinado de caracteres `len`.
4. En cada iteración del bucle, se compara la subcadena `needle` con la cadena `haystack` utilizando la función `ft_strncmp`.
5. Si se encuentra una coincidencia, se devuelve un puntero a la posición donde se encuentra la subcadena.
6. Si no se encuentra una coincidencia y se llega al final de la cadena `haystack` o se llega al número determinado de caracteres `len`, se devuelve `NULL`.

Diferencias entre ft_strstr y ft_strnstr

Las principales diferencias entre las funciones `ft_strstr` y `ft_strnstr` son:

- `ft_strstr` busca la subcadena en toda la cadena de caracteres, mientras que `ft_strnstr` busca la subcadena hasta un número determinado de caracteres.
- `ft_strstr` devuelve un puntero a la posición donde se encuentra la subcadena, mientras que `ft_strnstr` devuelve un puntero a la posición donde se encuentra la subcadena hasta el número determinado de caracteres.

Puedes probar las funciones `ft_strstr` y `ft_strnstr` con el siguiente código:

```
int main()
{
 char str[] = "Hola Mundo";
 char sub[] = "Mundo";
 printf("Posición de la subcadena: %s\n", ft_strstr(str, sub));
 printf("Posición de la subcadena hasta 10 caracteres: %s\n",
ft_strnstr(str, sub, 10));
 return 0;
}
```

Esto debería imprimir la cadena "`Mundo`"

ftatoi

¡Claro! La función `ft_atoi` se utiliza para convertir una cadena de caracteres en un entero.

A continuación, te presento la implementación de la función `ft_atoi`:

```
int ft_atoi(const char *str)
{
 int sign = 1;
 int res = 0;
 while (*str == ' ' || *str == '\t' || *str == '\n' || *str ==
'\r' || *str == '\f' || *str == '\v')
 str++;
```

```
 if (*str == '-')
 {
 sign = -1;
 str++;
 }
 else if (*str == '+')
 str++;
 while (*str >= '0' && *str <= '9')
 {
 res = res * 10 + (*str - '0');
 str++;
 }
 return res * sign;
}
```

Esta función utiliza un puntero `str` para recorrer la cadena de caracteres y convertir los caracteres en un entero.

Aquí hay una explicación paso a paso de cómo funciona la función:

1. Se inicializa la variable `sign` a 1, que se utilizará para almacenar el signo del entero.
2. Se inicializa la variable `res` a 0, que se utilizará para almacenar el resultado de la conversión.
3. Se recorre la cadena de caracteres hasta encontrar el primer caracter que no sea un espacio en blanco.
4. Si se encuentra un signo negativo (−), se establece la variable `sign` a -1 y se avanza al siguiente caracter.
5. Si se encuentra un signo positivo (+), se avanza al siguiente caracter.
6. Se recorre la cadena de caracteres hasta encontrar un caracter que no sea un dígito.
7. En cada iteración del bucle, se multiplica el resultado actual por 10 y se suma el valor del dígito actual.
8. Finalmente, se devuelve el resultado multiplicado por el signo.

Puedes probar la función `ft_atoi` con el siguiente código:

```
int main()
{
 char str[] = "12345";
 int res = ft_atoi(str);
 printf("Resultado: %d\n", res);
 return 0;
}
```

Esto debería imprimir la cadena "`Resultado: 12345`".

ft*itoa*

La función `ft_itoa` se utiliza para convertir un entero en una cadena de caracteres.

A continuación, te presento la implementación de la función `ft_itoa`:

```
char *ft_itoa(int n)
{
 int sign = 1;
 int len = 0;
 int tmp = n;
 if (n < 0)
 {
 sign = -1;
 n = -n;
 }
 while (tmp != 0)
 {
 len++;
 tmp /= 10;
 }
 if (sign == -1)
 len++;
 char *str = (char *)malloc(sizeof(char) * (len + 1));
 if (str == NULL)
 return NULL;
```

```
 str[len] = '\0';
 while (len > 0)
 {
 str[len - 1] = (n % 10) + '0';
 n /= 10;
 len--;
 }
 if (sign == -1)
 str[0] = '-';
 return str;
}
```

Esta función utiliza una variable `sign` para almacenar el signo del entero, una variable `len` para almacenar la longitud de la cadena de caracteres y una variable `tmp` para almacenar el valor temporal del entero.

Aquí hay una explicación paso a paso de cómo funciona la función:

1. Se inicializa la variable `sign` a 1, que se utilizará para almacenar el signo del entero.
2. Se inicializa la variable `len` a 0, que se utilizará para almacenar la longitud de la cadena de caracteres.
3. Se inicializa la variable `tmp` a `n`, que se utilizará para almacenar el valor temporal del entero.
4. Si `n` es negativo, se establece la variable `sign` a -1 y se cambia el signo de `n`.
5. Se recorre el entero `tmp` hasta que sea 0, incrementando la variable `len` en cada iteración.
6. Si `sign` es -1, se incrementa la variable `len` en 1 para tener en cuenta el signo negativo.
7. Se reserva memoria para la cadena de caracteres utilizando la función `malloc`.
8. Se inicializa la última posición de la cadena de caracteres a `\0`.

9. Se recorre la cadena de caracteres desde la última posición hasta la primera, asignando el valor de cada dígito a la posición correspondiente.
10. Si `sign` es -1, se asigna el signo negativo a la primera posición de la cadena de caracteres.
11. Finalmente, se devuelve la cadena de caracteres.

Puedes probar la función `ft_itoa` con el siguiente código:

```
int main()
{
 int n = 12345;
 char *str = ft_itoa(n);
 printf("Resultado: %s\n", str);
 free(str);
 return 0;
}
```

Esto debería imprimir la cadena `"Resultado: 12345"`. Recuerda liberar la memoria reservada utilizando la función `free` para evitar fugas de memoria.

ftcalloc

La función `ft_calloc` se utiliza para reservar memoria para un arreglo de elementos y inicializar todos los elementos a cero.

A continuación, te presento la implementación de la función `ft_calloc`:

```
void *ft_calloc(size_t nmemb, size_t size)
{
 void *ptr = malloc(nmemb * size);
 if (ptr == NULL)
 return NULL;
 char *tmp = (char *)ptr;
 size_t i = 0;
 while (i < nmemb * size)
 {
 tmp[i] = '\0';
```

```
 i++;
 }
 return ptr;
}
```

Esta función utiliza la función `malloc` para reservar memoria para el arreglo y luego inicializa todos los elementos a cero utilizando un bucle.

Aquí hay una explicación paso a paso de cómo funciona la función:

1. Se reserva memoria para el arreglo utilizando la función `malloc`.
2. Si la reserva de memoria falla, se devuelve `NULL`.
3. Se convierte el puntero `ptr` a un puntero `char` llamado `tmp`.
4. Se inicializa un índice `i` a 0.
5. Se recorre el arreglo utilizando un bucle `while`, inicializando cada elemento a cero.
6. Finalmente, se devuelve el puntero `ptr` al arreglo inicializado.

Puedes probar la función `ft_calloc` con el siguiente código:

```
int main()
{
 int *arr = (int *)ft_calloc(5, sizeof(int));
 if (arr == NULL)
 {
 printf("Error al reservar memoria\n");
 return 1;
 }
 int i = 0;
 while (i < 5)
 {
 printf("%d ", arr[i]);
 i++;
 }
 printf("\n");
 free(arr);
 return 0;
```

```
}
```

Esto debería imprimir la cadena "0 0 0 0 0", indicando que todos los elementos del arreglo han sido inicializados a cero. Recuerda liberar la memoria reservada utilizando la función `free` para evitar fugas de memoria.

ftstrdup

La función `ft_strdup` se utiliza para crear una copia de una cadena de caracteres.

A continuación, te presento la implementación de la función `ft_strdup`:

```
char *ft_strdup(const char *s)
{
 size_t len = ft_strlen(s);
 char *dup = (char *)malloc(sizeof(char) * (len + 1));
 if (dup == NULL)
 return NULL;
 size_t i = 0;
 while (i < len)
 {
 dup[i] = s[i];
 i++;
 }
 dup[i] = '\0';
 return dup;
}
```

Esta función utiliza la función `ft_strlen` para obtener la longitud de la cadena de caracteres original, luego reserva memoria para la copia utilizando la función `malloc` y finalmente copia los caracteres de la cadena original a la copia.

Aquí hay una explicación paso a paso de cómo funciona la función:

1. Se obtiene la longitud de la cadena de caracteres original utilizando la función `ft_strlen`.

2. Se reserva memoria para la copia utilizando la función `malloc`, sumando 1 a la longitud para tener en cuenta el carácter nulo (`\0`) que marca el final de la cadena.
3. Si la reserva de memoria falla, se devuelve `NULL`.
4. Se inicializa un índice `i` a 0.
5. Se recorre la cadena original utilizando un bucle `while`, copiando cada carácter a la copia.
6. Se agrega el carácter nulo (`\0`) al final de la copia.
7. Finalmente, se devuelve la copia.

Puedes probar la función `ft_strdup` con el siguiente código:

```
int main()
{
 char *orig = "Hola Mundo";
 char *dup = ft_strdup(orig);
 if (dup == NULL)
 {
 printf("Error al duplicar la cadena\n");
 return 1;
 }
 printf("Cadena original: %s\n", orig);
 printf("Cadena duplicada: %s\n", dup);
 free(dup);
 return 0;
}
```

Esto debería imprimir la cadena `"Cadena original: Hola Mundo"` y `"Cadena duplicada: Hola Mundo"`, indicando que la copia ha sido creada correctamente. Recuerda liberar la memoria reservada utilizando la función `free` para evitar fugas de memoria.

ftsubstr

La función `ft_substr` se utiliza para extraer una subcadena de una cadena de caracteres original.

A continuación, te presento la implementación de la función `ft_substr`:

```
char *ft_substr(char const *s, unsigned int start, size_t len)
{
 if (start >= ft_strlen(s) || len == 0)
 return ft_strdup("");
 char *substr = (char *)malloc(sizeof(char) * (len + 1));
 if (substr == NULL)
 return NULL;
 size_t i = 0;
 while (i < len)
 {
 substr[i] = s[start + i];
 i++;
 }
 substr[i] = '\0';
 return substr;
}
```

Esta función utiliza la función `ft_strlen` para obtener la longitud de la cadena de caracteres original, luego reserva memoria para la subcadena utilizando la función `malloc` y finalmente copia los caracteres de la cadena original a la subcadena.

Aquí hay una explicación paso a paso de cómo funciona la función:

1. Si el índice de inicio `start` es mayor o igual que la longitud de la cadena original, o si la longitud de la subcadena `len` es 0, se devuelve una cadena vacía utilizando la función `ft_strdup`.
2. Se reserva memoria para la subcadena utilizando la función `malloc`, sumando 1 a la longitud para tener en cuenta el carácter nulo (`\0`) que marca el final de la cadena.
3. Si la reserva de memoria falla, se devuelve `NULL`.

4. Se inicializa un índice `i` a 0.
5. Se recorre la cadena original utilizando un bucle `while`, copiando cada carácter a la subcadena a partir del índice de inicio `start`.
6. Se agrega el carácter nulo (`\0`) al final de la subcadena.
7. Finalmente, se devuelve la subcadena.

Puedes probar la función `ft_substr` con el siguiente código:

```
int main()
{
 char *orig = "Hola Mundo";
 char *substr = ft_substr(orig, 5, 5);
 if (substr == NULL)
 {
 printf("Error al extraer la subcadena\n");
 return 1;
 }
 printf("Cadena original: %s\n", orig);
 printf("Subcadena: %s\n", substr);
 free(substr);
 return 0;
}
```

Esto debería imprimir la cadena "`Cadena original: Hola Mundo`" y "`Subcadena: Mundo`", indicando que la subcadena ha sido extraída correctamente. Recuerda liberar la memoria reservada utilizando la función `free` para evitar fugas de memoria.

ftstrjoin

La función `ft_strjoin` se utiliza para concatenar dos cadenas de caracteres.

A continuación, te presento la implementación de la función `ft_strjoin`:

```
char *ft_strjoin(char const *s1, char const *s2)
{
```

```
 size_t len_s1 = ft_strlen(s1);
 size_t len_s2 = ft_strlen(s2);
 char *join = (char *)malloc(sizeof(char) * (len_s1 + len_s2 +
1));
 if (join == NULL)
 return NULL;
 size_t i = 0;
 while (i < len_s1)
 {
 join[i] = s1[i];
 i++;
 }
 while (i < len_s1 + len_s2)
 {
 join[i] = s2[i - len_s1];
 i++;
 }
 join[i] = '\0';
 return join;
}
```

Esta función utiliza la función `ft_strlen` para obtener la longitud de cada cadena de caracteres, luego reserva memoria para la cadena resultante utilizando la función `malloc` y finalmente copia los caracteres de cada cadena a la cadena resultante.

Aquí hay una explicación paso a paso de cómo funciona la función:

1. Se obtiene la longitud de cada cadena de caracteres utilizando la función `ft_strlen`.
2. Se reserva memoria para la cadena resultante utilizando la función `malloc`, sumando 1 a la longitud total para tener en cuenta el carácter nulo (`\0`) que marca el final de la cadena.
3. Si la reserva de memoria falla, se devuelve `NULL`.
4. Se inicializa un índice `i` a 0.

5. Se recorre la primera cadena utilizando un bucle `while`, copiando cada carácter a la cadena resultante.
6. Se recorre la segunda cadena utilizando un bucle `while`, copiando cada carácter a la cadena resultante a partir de la posición donde se dejó la primera cadena.
7. Se agrega el carácter nulo (`\0`) al final de la cadena resultante.
8. Finalmente, se devuelve la cadena resultante.

Puedes probar la función `ft_strjoin` con el siguiente código:

```
int main()
{
 char *s1 = "Hola ";
 char *s2 = "Mundo";
 char *join = ft_strjoin(s1, s2);
 if (join == NULL)
 {
 printf("Error al unir las cadenas\n");
 return 1;
 }
 printf("Cadena 1: %s\n", s1);
 printf("Cadena 2: %s\n", s2);
 printf("Cadena unida: %s\n", join);
 free(join);
 return 0;
}
```

Esto debería imprimir la cadena `"Cadena 1: Hola "`, `"Cadena 2: Mundo"` y `"Cadena unida: Hola Mundo"`, indicando que las cadenas han sido unidas correctamente. Recuerda liberar la memoria reservada utilizando la función `free` para evitar fugas de memoria.

ftstrtrim

La función `ft_strtrim` se utiliza para eliminar los caracteres especificados al principio y al final de una cadena de caracteres.

A continuación, te presento la implementación de la función `ft_strtrim`:

```
char *ft_strtrim(char const *s1, char const *set)
{
 size_t len_s1 = ft_strlen(s1);
 size_t len_set = ft_strlen(set);
 size_t start = 0;
 size_t end = len_s1 - 1;

 while (start < len_s1 && ft_strchr(set, s1[start]) != NULL)
 start++;
 while (end >= start && ft_strchr(set, s1[end]) != NULL)
 end--;

 if (start > end)
 return ft_strdup("");

 char *trim = (char *)malloc(sizeof(char) * (end - start + 2));
 if (trim == NULL)
 return NULL;

 size_t i = 0;
 while (start <= end)
 {
 trim[i] = s1[start];
 start++;
 i++;
 }
 trim[i] = '\0';
 return trim;
}
```

Esta función utiliza la función `ft_strlen` para obtener la longitud de cada cadena de caracteres, y la función `ft_strchr` para buscar los caracteres especificados en la cadena `set`. Luego reserva memoria para la cadena resultante utilizando la fun-

ción `malloc` y finalmente copia los caracteres de la cadena original a la cadena resultante.

Aquí hay una explicación paso a paso de cómo funciona la función:

1. Se obtiene la longitud de cada cadena de caracteres utilizando la función `ft_strlen`.
2. Se inicializan dos índices, `start` y `end`, a 0 y a la longitud de la cadena `s1` menos 1, respectivamente.
3. Se busca el primer carácter de la cadena `s1` que no esté en la cadena `set`, y se actualiza el índice `start`.
4. Se busca el último carácter de la cadena `s1` que no esté en la cadena `set`, y se actualiza el índice `end`.
5. Si el índice `start` es mayor que el índice `end`, se devuelve una cadena vacía utilizando la función `ft_strdup`.
6. Se reserva memoria para la cadena resultante utilizando la función `malloc`.
7. Se copian los caracteres de la cadena original a la cadena resultante, desde el índice `start` hasta el índice `end`.
8. Se agrega el carácter nulo (`\0`) al final de la cadena resultante.
9. Finalmente, se devuelve la cadena resultante.

Puedes probar la función `ft_strtrim` con el siguiente código:

```
int main()
{
 char *s1 = "   Hola Mundo   ";
 char *set = " ";
 char *trim = ft_strtrim(s1, set);
 if (trim == NULL)
 {
 printf("Error al recortar la cadena\n");
 return 1;
```

```
 }
 printf("Cadena original: %s\n", s1);
 printf("Cadena recortada: %s\n", trim);
 free(trim);
 return 0;
}
```

Esto debería imprimir la cadena "`Cadena original: Hola Mundo `" y "`Cadena recortada: Hola Mundo`", indicando que la cadena ha sido recortada correctamente. Recuerda liberar la memoria reservada utilizando la función `free` para evitar fugas de memoria.

ft*split*

La función `ft_split` se utiliza para dividir una cadena de caracteres en un arreglo de cadenas, utilizando un carácter especificado como separador.

A continuación, te presento la implementación de la función `ft_split`:

```
char **ft_split(char const *s, char c)
{
 size_t len = ft_strlen(s);
 size_t count = 0;
 size_t i = 0;

 while (i < len)
 {
 if (s[i] == c)
 count++;
 i++;
 }
 count++;

 char **split = (char **)malloc(sizeof(char *) * (count + 1));
 if (split == NULL)
 return NULL;

 size_t j = 0;
```

```
 size_t start = 0;
 while (j < count)
 {
 size_t end = start;
 while (end < len && s[end] != c)
 end++;
 split[j] = ft_substr(s, start, end - start);
 if (split[j] == NULL)
 {
 ft_free_split(split, j);
 return NULL;
 }
 start = end + 1;
 j++;
 }
 split[count] = NULL;
 return split;
}

void ft_free_split(char **split, size_t len)
{
 size_t i = 0;
 while (i < len)
 {
 free(split[i]);
 i++;
 }
 free(split);
}
```

Esta función utiliza la función `ft_strlen` para obtener la longitud de la cadena de caracteres, y la función `ft_substr` para crear cada una de las cadenas del arreglo. Luego reserva memoria para el arreglo de cadenas utilizando la función `malloc`.

Aquí hay una explicación paso a paso de cómo funciona la función:

1. Se obtiene la longitud de la cadena de caracteres utilizando la función `ft_strlen`.

2. Se cuenta el número de veces que aparece el carácter separador en la cadena.
3. Se reserva memoria para el arreglo de cadenas utilizando la función `malloc`.
4. Se recorre la cadena de caracteres, creando cada una de las cadenas del arreglo utilizando la función `ft_substr`.
5. Se agrega un puntero nulo al final del arreglo para indicar su fin.
6. Finalmente, se devuelve el arreglo de cadenas.

La función `ft_free_split` se utiliza para liberar la memoria reservada por la función `ft_split`.

Puedes probar la función `ft_split` con el siguiente código:

```
int main()
{
 char *s = "Hola,Mundo,Esto,Es,Un,Ejemplo";
 char c = ',';
 char **split = ft_split(s, c);
 if (split == NULL)
 {
 printf("Error al dividir la cadena\n");
 return 1;
 }
 size_t i = 0;
 while (split[i] != NULL)
 {
 printf("%s\n", split[i]);
 i++;
 }
 ft_free_split(split, i);
 return 0;
}
```

Esto debería imprimir cada una de las cadenas del arreglo, indicando que la cadena ha sido dividida correctamente. Recuerda liberar la memoria reservada utilizando la función `ft_free_split` para evitar fugas de memoria.

ftstrmapi

La función `ft_strmapi` se utiliza para aplicar una función a cada carácter de una cadena de caracteres y devuelve una nueva cadena con los resultados.

A continuación, te presento la implementación de la función `ft_strmapi`:

```
char *ft_strmapi(char const *s, char (*f)(unsigned int, char))
{
 size_t len = ft_strlen(s);
 char *str = (char *)malloc(sizeof(char) * (len + 1));
 if (str == NULL)
 return NULL;
 size_t i = 0;
 while (i < len)
 {
 str[i] = f(i, s[i]);
 i++;
 }
 str[i] = '\0';
 return str;
}
```

Esta función utiliza la función `ft_strlen` para obtener la longitud de la cadena de caracteres, y luego reserva memoria para la nueva cadena utilizando la función `malloc`.

Aquí hay una explicación paso a paso de cómo funciona la función:

1. Se obtiene la longitud de la cadena de caracteres utilizando la función `ft_strlen`.
2. Se reserva memoria para la nueva cadena utilizando la función `malloc`.

3. Se recorre la cadena de caracteres, aplicando la función `f` a cada carácter y almacenando el resultado en la nueva cadena.
4. Se agrega el carácter nulo (`\0`) al final de la nueva cadena.
5. Finalmente, se devuelve la nueva cadena.

La función `f` debe tener la siguiente forma:

```
char f(unsigned int index, char c)
{
 // Código que se ejecutará para cada carácter
}
```

Donde `index` es el índice del carácter en la cadena original y `c` es el carácter en sí.

Puedes probar la función `ft_strmapi` con el siguiente código:

```
char f(unsigned int index, char c)
{
 return c + 1; // Incrementa cada carácter en 1
}

int main()
{
 char *s = "Hola Mundo";
 char *str = ft_strmapi(s, f);
 if (str == NULL)
 {
 printf("Error al aplicar la función\n");
 return 1;
 }
 printf("%s\n", str);
 free(str);
 return 0;
}
```

Esto debería imprimir la cadena `"Ipmb Nvope"`, que es el resultado de incrementar cada carácter de la cadena original en 1. Recuerda liberar la memoria reservada utilizando la función `free` para evitar fugas de memoria.

ftstriteri

La función `ft_striteri` se utiliza para aplicar una función a cada carácter de una cadena de caracteres, pero a diferencia de `ft_strmapi`, no devuelve una nueva cadena, sino que modifica la cadena original.

A continuación, te presento la implementación de la función `ft_striteri`:

```
void ft_striteri(char *s, void (*f)(unsigned int, char *))
{
 size_t i = 0;
 while (s[i] != '\0')
 {
 f(i, &s[i]);
 i++;
 }
}
```

Esta función recorre la cadena de caracteres, aplicando la función `f` a cada carácter y modificando la cadena original.

Aquí hay una explicación paso a paso de cómo funciona la función:

1. Se recorre la cadena de caracteres, carácter por carácter.
2. Se aplica la función `f` a cada carácter, pasando el índice del carácter y un puntero al carácter en sí.
3. La función `f` puede modificar el carácter en la cadena original.

La función `f` debe tener la siguiente forma:

```
void f(unsigned int index, char *c)
{
 // Código que se ejecutará para cada carácter
}
```

Donde `index` es el índice del carácter en la cadena original y `c` es un puntero al carácter en sí.

Puedes probar la función `ft_striteri` con el siguiente código:

```
void f(unsigned int index, char *c)
{
 *c = *c + 1; // Incrementa cada carácter en 1
}

int main()
{
 char s[] = "Hola Mundo";
 ft_striteri(s, f);
 printf("%s\n", s);
 return 0;
}
```

Esto debería imprimir la cadena `"Ipmb Nvope"`, que es el resultado de incrementar cada carácter de la cadena original en 1. Ten en cuenta que la cadena original se modifica directamente.

Recuerda que, al contrario de `ft_strmapi`, `ft_striteri` no devuelve una nueva cadena, sino que modifica la cadena original. Por lo tanto, no es necesario liberar memoria después de llamar a esta función.

ft*lstnew*

La función `ft_lstnew` se utiliza para crear un nuevo elemento de una lista enlazada.

A continuación, te presento la implementación de la función `ft_lstnew`:

```
t_list *ft_lstnew(void *content)
{
 t_list *new = (t_list *)malloc(sizeof(t_list));
 if (new == NULL)
 return NULL;
 new->content = content;
 new->next = NULL;
 return new;
```

```
}
```

Esta función reserva memoria para un nuevo elemento de la lista enlazada y lo inicializa con el contenido proporcionado.

Aquí hay una explicación paso a paso de cómo funciona la función:

1. Se reserva memoria para un nuevo elemento de la lista enlazada utilizando la función `malloc`.
2. Si la reserva de memoria falla, se devuelve `NULL`.
3. Se inicializa el contenido del nuevo elemento con el contenido proporcionado.
4. Se inicializa el puntero `next` del nuevo elemento a `NULL`, indicando que no hay otro elemento después de este.
5. Finalmente, se devuelve el nuevo elemento.

La estructura `t_list` debe estar definida de la siguiente manera:

```
typedef struct s_list
{
 void *content;
 struct s_list *next;
} t_list;
```

Puedes probar la función `ft_lstnew` con el siguiente código:

```
int main()
{
 char *content = "Hola Mundo";
 t_list *new = ft_lstnew(content);
 if (new == NULL)
 {
 printf("Error al crear el nuevo elemento\n");
 return 1;
 }
 printf("%s\n", (char *)new->content);
 free(new);
 return 0;
```

```
}
```

Esto debería imprimir la cadena "Hola Mundo", que es el contenido del nuevo elemento. Recuerda liberar la memoria reservada utilizando la función `free` para evitar fugas de memoria.

Ten en cuenta que la función `ft_lstnew` no realiza una copia del contenido, sino que almacena el puntero al contenido original. Por lo tanto, si se modifica el contenido original, se modificará también el contenido del elemento de la lista. Si deseas crear una copia del contenido, debes hacerlo manualmente antes de llamar a `ft_lstnew`.

ft*lstadd*front y ft*lstadd*back

Las funciones `ft_lstadd_front` y `ft_lstadd_back` se utilizan para agregar un nuevo elemento a una lista enlazada, ya sea al principio o al final de la lista.

ft_lstadd_front

La función `ft_lstadd_front` se utiliza para agregar un nuevo elemento al principio de la lista enlazada.

A continuación, te presento la implementación de la función `ft_lstadd_front`:

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

Esta función agrega el nuevo elemento al principio de la lista enlazada, actualizando el puntero `next` del nuevo elemento para que apunte al primer elemento de la lista original.

Aquí hay una explicación paso a paso de cómo funciona la función:

1. Si la lista enlazada está vacía (`*lst == NULL`), se asigna el nuevo elemento a la lista enlazada y se devuelve.
2. Se actualiza el puntero `next` del nuevo elemento para que apunte al primer elemento de la lista original.
3. Se actualiza el puntero `*lst` para que apunte al nuevo elemento, que ahora es el primer elemento de la lista enlazada.

ft_lstadd_back

La función `ft_lstadd_back` se utiliza para agregar un nuevo elemento al final de la lista enlazada.

A continuación, te presento la implementación de la función `ft_lstadd_back`:

```
void ft_lstadd_back(t_list **lst, t_list *new)
{
 if (*lst == NULL)
 {
 *lst = new;
 return;
 }
 t_list *last = *lst;
 while (last->next != NULL)
 last = last->next;
 last->next = new;
}
```

Esta función agrega el nuevo elemento al final de la lista enlazada, actualizando el puntero `next` del último elemento de la lista original para que apunte al nuevo elemento.

Aquí hay una explicación paso a paso de cómo funciona la función:

1. Si la lista enlazada está vacía (`*lst == NULL`), se asigna el nuevo elemento a la lista enlazada y se devuelve.
2. Se busca el último elemento de la lista enlazada, recorriendo la lista desde el primer elemento hasta encontrar el elemento que tiene `next == NULL`.
3. Se actualiza el puntero `next` del último elemento para que apunte al nuevo elemento.
4. El nuevo elemento se convierte en el último elemento de la lista enlazada.

Puedes probar las funciones `ft_lstadd_front` y `ft_lstadd_back` con el siguiente código:

```
int main()
{
 t_list *lst = NULL;
 t_list *new1 = ft_lstnew("Hola");
 t_list *new2 = ft_lstnew("Mundo");
 ft_lstadd_front(&lst, new1);
 ft_lstadd_back(&lst, new2);
 while (lst != NULL)
 {
 printf("%s\n", (char *)lst->content);
 lst = lst->next;
 }
 return 0;
}
```

Esto debería imprimir las cadenas `"Hola"` y `"Mundo"`, que son los contenidos de los elementos de la lista enlazada.

ft*lstsize*

La función `ft_lstsize` se utiliza para obtener el número de elementos en una lista enlazada.

A continuación, te presento la implementación de la función `ft_lstsize`:

```
int ft_lstsize(t_list *lst)
{
 int size = 0;
 while (lst != NULL)
 {
 size++;
 lst = lst->next;
 }
 return size;
}
```

Esta función recorre la lista enlazada, contando el número de elementos hasta llegar al final de la lista.

Aquí hay una explicación paso a paso de cómo funciona la función:

1. Se inicializa una variable `size` a 0, que se utilizará para contar el número de elementos en la lista.
2. Se recorre la lista enlazada, comenzando desde el primer elemento (`lst`).
3. En cada iteración, se incrementa la variable `size` en 1.
4. Se mueve al siguiente elemento de la lista (`lst = lst->next`).
5. Se repite el proceso hasta llegar al final de la lista (`lst == NULL`).
6. Finalmente, se devuelve el valor de `size`, que es el número de elementos en la lista.

Puedes probar la función `ft_lstsize` con el siguiente código:

```
int main()
{
```

```
 t_list *lst = NULL;
 t_list *new1 = ft_lstnew("Hola");
 t_list *new2 = ft_lstnew("Mundo");
 t_list *new3 = ft_lstnew("!");
 ft_lstadd_back(&lst, new1);
 ft_lstadd_back(&lst, new2);
 ft_lstadd_back(&lst, new3);
 printf("Tamaño de la lista: %d\n", ft_lstsize(lst));
 return 0;
}
```

Esto debería imprimir el mensaje "`Tamaño de la lista: 3`", que es el número de elementos en la lista enlazada.

La función `ft_lstsize` es útil para obtener información sobre la lista enlazada, como el número de elementos que contiene. También se puede utilizar para realizar operaciones que dependen del tamaño de la lista, como la reserva de memoria para una matriz o la creación de una estructura de datos que requiere un tamaño específico.

ftlstdelone

La función `ft_lstdelone` se utiliza para eliminar un elemento de una lista enlazada y liberar la memoria asociada con él.

A continuación, te presento la implementación de la función `ft_lstdelone`:

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

Esta función elimina un elemento de la lista enlazada y libera la memoria asociada con él, utilizando la función `del` para liberar el contenido del elemento.

Aquí hay una explicación paso a paso de cómo funciona la función:

1. Se verifica si el elemento `lst` es `NULL`, en cuyo caso se devuelve inmediatamente.
2. Se llama a la función `del` para liberar el contenido del elemento `lst`.
3. Se libera la memoria asociada con el elemento `lst` utilizando la función `free`.

La función `del` debe ser una función que sepa cómo liberar el contenido del elemento, ya que la función `ft_lstdelone` no conoce el tipo de contenido que se almacena en el elemento.

Puedes probar la función `ft_lstdelone` con el siguiente código:

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

int main()
{
 t_list *lst = NULL;
 char *str = malloc(sizeof(char) * 10);
 strcpy(str, "Hola Mundo");
 t_list *new = ft_lstnew(str);
 ft_lstadd_back(&lst, new);
 ft_lstdelone(lst, del);
 return 0;
}
```

En este ejemplo, la función `del` simplemente libera la memoria asociada con el contenido del elemento utilizando la función `free`. La función `ft_lstdelone` llama a la función `del` para liberar el contenido del elemento y luego libera la memoria asociada con el elemento en sí.

Recuerda que la función `ft_lstdelone` no elimina el elemento de la lista enlazada, solo libera la memoria asociada con él. Si deseas eliminar el elemento de la

lista enlazada, debes actualizar los punteros de la lista enlazada para que no apunten al elemento eliminado.

La función `ft_lstdelone` es útil para liberar la memoria asociada con un elemento de la lista enlazada cuando ya no se necesita. También se puede utilizar en conjunto con otras funciones para eliminar elementos de la lista enlazada de manera segura y eficiente.

ft*lstclear*

La función `ft_lstclear` se utiliza para eliminar todos los elementos de una lista enlazada y liberar la memoria asociada con ellos.

A continuación, te presento la implementación de la función `ft_lstclear`:

```
void ft_lstclear(t_list **lst, void (*del)(void*))
{
 t_list *temp;
 while (*lst != NULL)
 {
 temp = *lst;
 *lst = (*lst)->next;
 del(temp->content);
 free(temp);
 }
}
```

Esta función elimina todos los elementos de la lista enlazada y libera la memoria asociada con ellos, utilizando la función `del` para liberar el contenido de cada elemento.

Aquí hay una explicación paso a paso de cómo funciona la función:

1. Se verifica si la lista enlazada `*lst` es `NULL`, en cuyo caso se devuelve inmediatamente.

2. Se crea un puntero temporal `temp` que apunta al primer elemento de la lista enlazada.
3. Se actualiza el puntero `*lst` para que apunte al siguiente elemento de la lista enlazada (`(*lst)->next`).
4. Se llama a la función `del` para liberar el contenido del elemento `temp`.
5. Se libera la memoria asociada con el elemento `temp` utilizando la función `free`.
6. Se repite el proceso hasta que la lista enlazada esté vacía (`*lst == NULL`).

La función `del` debe ser una función que sepa cómo liberar el contenido de cada elemento, ya que la función `ft_lstclear` no conoce el tipo de contenido que se almacena en los elementos.

Puedes probar la función `ft_lstclear` con el siguiente código:

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

int main()
{
 t_list *lst = NULL;
 char *str1 = malloc(sizeof(char) * 10);
 strcpy(str1, "Hola Mundo");
 t_list *new1 = ft_lstnew(str1);
 ft_lstadd_back(&lst, new1);
 char *str2 = malloc(sizeof(char) * 10);
 strcpy(str2, "Adiós Mundo");
 t_list *new2 = ft_lstnew(str2);
 ft_lstadd_back(&lst, new2);
 ft_lstclear(&lst, del);
 return 0;
}
```

En este ejemplo, la función `del` simplemente libera la memoria asociada con el contenido de cada elemento utilizando la función `free`. La función `ft_lstclear` llama a la función `del` para liberar el contenido de cada elemento y luego libera la memoria asociada con cada elemento en sí.

Recuerda que la función `ft_lstclear` elimina todos los elementos de la lista enlazada y libera la memoria asociada con ellos. Si deseas conservar algunos elementos de la lista enlazada, debes crear una nueva lista enlazada y copiar los elementos que deseas conservar.

La función `ft_lstclear` es útil para liberar la memoria asociada con una lista enlazada cuando ya no se necesita. También se puede utilizar en conjunto con otras funciones para eliminar elementos de la lista enlazada de manera segura y eficiente.

ft*strlmap*

La función `ft_lstmap` se utiliza para aplicar una función a cada elemento de una lista enlazada y crear una nueva lista enlazada con los resultados.

A continuación, te presento la implementación de la función `ft_lstmap`:

```
t_list *ft_lstmap(t_list *lst, void *(*f)(void *), void (*del)
(void *))
{
 t_list *new_lst = NULL;
 t_list *new_elem;
 while (lst != NULL)
 {
 new_elem = ft_lstnew(f(lst->content));
 if (new_elem == NULL)
 {
 ft_lstclear(&new_lst, del);
 return NULL;
 }
 ft_lstadd_back(&new_lst, new_elem);
```

```
 lst = lst->next;
 }
 return new_lst;
}
```

Esta función aplica la función `f` a cada elemento de la lista enlazada `lst` y crea una nueva lista enlazada `new_lst` con los resultados. La función `del` se utiliza para liberar la memoria asociada con los elementos de la lista enlazada en caso de error.

Aquí hay una explicación paso a paso de cómo funciona la función:

1. Se crea una nueva lista enlazada `new_lst` vacía.
2. Se recorre la lista enlazada `lst` elemento por elemento.
3. Para cada elemento, se aplica la función `f` al contenido del elemento y se crea un nuevo elemento `new_elem` con el resultado.
4. Si la creación del nuevo elemento falla (`new_elem == NULL`), se libera la memoria asociada con la lista enlazada `new_lst` utilizando la función `ft_lstclear` y se devuelve `NULL`.
5. Se agrega el nuevo elemento `new_elem` a la lista enlazada `new_lst` utilizando la función `ft_lstadd_back`.
6. Se repite el proceso hasta que se hayan procesado todos los elementos de la lista enlazada `lst`.
7. Finalmente, se devuelve la nueva lista enlazada `new_lst`.

La función `f` debe ser una función que sepa cómo procesar el contenido de cada elemento, ya que la función `ft_lstmap` no conoce el tipo de contenido que se almacena en los elementos.

Puedes probar la función `ft_lstmap` con el siguiente código:

```
void *f(void *content)
{
 char *str = malloc(sizeof(char) * 10);
 strcpy(str, "Hola ");
 strcat(str, (char *)content);
```

```
 return str;
}

void del(void *content)
{
 free(content);
}

int main()
{
 t_list *lst = NULL;
 char *str1 = malloc(sizeof(char) * 10);
 strcpy(str1, "Mundo");
 t_list *new1 = ft_lstnew(str1);
 ft_lstadd_back(&lst, new1);
 char *str2 = malloc(sizeof(char) * 10);
 strcpy(str2, "Adiós");
 t_list *new2 = ft_lstnew(str2);
 ft_lstadd_back(&lst, new2);
 t_list *new_lst = ft_lstmap(lst, f, del);
 while (new_lst != NULL)
 {
 printf("%s\n", (char *)new_lst->content);
 new_lst = new_lst->next;
 }
 return 0;
}
```

En este ejemplo, la función `f` concatena la cadena "Hola " con el contenido de cada elemento. La función `ft_lstmap` aplica la función `f` a cada elemento de la lista enlazada `lst` y crea una nueva lista enlazada `new_lst` con los resultados.

Recuerda que la función `ft_lstmap` crea una nueva lista enlazada con los resultados de la función `f`. Si deseas conservar la lista enlazada original, debes crear una copia de la lista enlazada antes de llamar a la función `ft_lstmap`.

La función `ft_lstmap` es útil para procesar los elementos de una lista enlazada y crear una nueva lista enlazada con los resultados. También se puede utilizar

en conjunto con otras funciones para realizar operaciones complejas sobre listas enlazadas.