UNIVERSIDAD DE COSTA RICA

Escuela de Ingeniería Eléctrica

IE0117: Programación Bajo Plataformas Abiertas

Reporte

Laboratorio # 4

Prof.Carolina Trejos Quiros

Estudiante: Jafet Guillermo Cruz Salazar - C02520

18 de mayo de 2025

Índice

1. Introducción **2**

2. Implementación **3**

3. Resultados **8**

4. Conclusiones y Recomendaciones **10**

Referencias **11**

1. Introducción

El presente informe corresponde al laboratorio numero cuatro. Este laboratorio se enfoca en el fortalecimiento de conceptos clave del manejo avanzado de memoria dinámica, aritmética de punteros, procesamiento de texto y la gestión eficiente y segura de recursos mediante practicas de programación robustas.

En este laboratorio se hizo uso del estándar POSIX.1-2008 mediante la definición explícita de la macro:

Listing 1: Macro

```
#define _POSIX_C_SOURCE 200809L
```

Esto se realizó para garantizar la portabilidad y compatibilidad del código con funciones específicas, como `strdup()`, que forman parte del estándar `POSIX` pero no están disponibles de manera predeterminada en el estándar **ISO C11**. El estándar `POSIX (Portable Operating System Interface)` está definido por **The Open Group** y proporciona una especificación clara y robusta para asegurar la interoperabilidad entre distintos sistemas operativos basados en `Unix` o similares[1].

Asimismo, se utilizó la herramienta **Valgrind** para validar la correcta gestión dinámica de memoria y asegurar la ausencia de fugas (memory leaks) y errores relacionados con accesos inválidos o memoria no inicializada. **Valgrind** es una suite integral de depuración y perfilado que ayuda a mejorar la robustez y eficiencia del software desarrollado, permitiendo identificar rápidamente errores difíciles de detectar por otros medios[2].

Finalmente como guía de funciones se tomo de referencia el manual de programación para `C` proporcionado por la **IBM**[3].

El código fuente de los scripts desarrollados se encuentra disponible en el siguiente `repositorio`; en este también se encuentran instrucciones para la ejecución de cada script: https://github.com/MavrosAilouros/Lab4_Memory_and_Pointers

A continuación se detallan los resultados obtenidos, análisis respectivas y conclusiones derivadas del desarrollo de los ejercicios planteados.

2. Implementación

Ejercicio 1: Búsqueda de secuencia más larga de unos en una matriz binaria mediante aritmética de punteros

Objetivo: Crear un programa que permita determinar la longitud máxima de secuencias consecutivas de números binarios iguales a 1 dentro de una matriz cuadrada, utilizando exclusivamente aritmética de punteros para acceder a los elementos de la matriz. Además, el programa debe ser capaz de manejar memoria dinámica de forma segura, eficiente y sin fugas.

Desarrollo: A continuación se detallan los aspectos más importantes en la implementación del programa desarrollado para encontrar la secuencia más larga de 1s en una matriz binaria cuadrada, cumpliendo con los requerimientos del enunciado del laboratorio.

Primero, se creó una función `(allocateMatrix)` para reservar memoria dinámica para la matriz cuadrada, permitiendo que su dimensión sea especificada por el usuario en tiempo de ejecución. Esto garantiza flexibilidad y eficiencia en el uso de memoria.

Listing 2: Reserva dinamica de memoria

```
/**
 * @brief Reserva memoria din mica para una matriz cuadrada.
 * @param[out] matrix triple puntero que recibir  la direcci n de memoria.
 * @param[in] size dimensi n (filas y columnas) de la matriz.
 */
int allocateMatrix(int ***matrix, int size) {
    if (!matrix || size <= 0) return -1;

    *matrix = malloc(size * sizeof(int *));
    if (!*matrix) return -1;

    for (int i = 0; i < size; ++i) {
        (*matrix)[i] = calloc(size, sizeof(int));
        if (!(*matrix)[i]) return -1;
    }

    return 0;
}
```

Mediante la función `(fillMatrix)` se llenó la matriz previamente reservada con números binarios aleatorios (0 y 1). Esto se realizó usando funciones estándar `(rand() y srand(time(NULL)))` para obtener valores pseudoaleatorios en cada ejecución.

Listing 3: Llenado aleatorio de la Matriz

```
    /**
 * @brief Llena la matriz cuadrada con valores binarios aleatorios.
 * @param matrix puntero doble que referencia a la matriz.
 * @param size dimensi n de la matriz.
 */
void fillMatrix(int **matrix, int size) {
    if (!matrix || size <= 0) return;
```

```
    srand(time(NULL));
    for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; ++j) {
            *(*(matrix + i) + j) = rand() % 2;
        }
    }
}
```

Se desarrolló una función (`findLargestLine`) para buscar y determinar la longitud de la secuencia más larga de números consecutivos iguales a uno. Este algoritmo se diseñó específicamente para permitir que las secuencias puedan continuar entre filas sucesivas, y se implementó usando únicamente aritmética de punteros, tal como exige el enunciado.

Listing 4: Búsqueda de la secuencia más larga

```
/**
 * @brief Busca la secuencia m s larga de 1s consecutivos en la matriz.
 * @param matrix puntero doble que referencia a la matriz.
 * @param size dimensi n de la matriz.
 * @param[out] result longitud de la secuencia m s larga encontrada.
 */
void findLargestLine(int **matrix, int size, int *result) {
    if (!matrix || !result || size <= 0) return;

    int count = 0;
    *result = 0;

    for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; ++j) {
            if (*(*(matrix + i) + j) == 1) {
                count++;
                if (count > *result) {
                    *result = count;
                }
            } else {
                count = 0;
            }
        }
    }
}
```

Finalmente, se creó una función que asegura la correcta liberación de memoria reservada, evitando así fugas detectadas mediante **Valgrind**:

Listing 5: Liberación segura de memoria

```
/**
```

```
 * @brief Libera memoria din mica reservada por la matriz.
 * @param matrix referencia triple a la matriz que se desea liberar.
 * @param size dimensi n de la matriz.
 */
void freeMatrix(int ***matrix, int size) {
    if (!matrix || !*matrix) return;

    for (int i = 0; i < size; ++i) {
        free((*matrix)[i]);
        (*matrix)[i] = NULL;
    }

    free(*matrix);
    *matrix = NULL;
}
```

Cada etapa se realizó respetando estrictamente las buenas prácticas de programación vistas en clase, asegurando un código legible, modular, seguro y eficiente.

Ejercicio 2: Identificación del palíndromo más largo desde un archivo de texto

Objetivo: Implementar un programa que lea un archivo de texto, analice cada palabra contenida en él, elimine caracteres no alfanuméricos, convierta todas las palabras a minúscula y determine cuál es el palíndromo más largo encontrado.

Desarrollo:

La solución del ejercicio se llevó a cabo mediante una estructura modular dividida en etapas claramente definidas:

Utilizando funciones estándar como `fopen()` y `fscanf()`, se procedió a leer cada palabra desde un archivo llamado **input.txt**. Cada palabra leída fue sometida a un preprocesamiento especial para eliminar caracteres no alfanuméricos y convertir todos los caracteres a minúsculas. Para este propósito, se creó una función específica llamada `clean_token()`:

Listing 6: Funcion clean$_{t}oken$

```
/**
 * @brief Limpia una cadena de caracteres eliminando caracteres no alfanum rico
 *        y convirtiendo los restantes a min scula.
 * @param[in] raw Cadena original le da desde el archivo.
 * @return Cadena limpia en memoria din mica, o NULL si la entrada es inv lida
 */
char *clean_token(const char *raw);
```

Seguidamente se implementó una función llamada `is_palindrome()` para verificar si una palabra limpia es un palíndromo o no. La función utiliza dos punteros avanzando desde los extremos hacia el centro para comparar caracteres de manera eficiente:

Listing 7: Deteccion de palindromos

```
/**
 * @brief Determina si una cadena es un pal ndromo.
 * @param[in] s Cadena limpia (solo caracteres alfanum ricos en min sculas).
 * @return true si la cadena es pal ndromo, false en caso contrario.
 */
bool is_palindrome(const char *s);
```

Para la selección del palíndromo mas largo se utilizó una función específica, `replace_longest()`, que compara cada palíndromo encontrado con el palíndromo más largo detectado hasta el momento, actualizando esta referencia si el nuevo palíndromo es más largo que el actual:

Listing 8: Actualizacion del palindromo mas largo

```
/**
 * @brief Actualiza el pal ndromo m s largo encontrado.
 * @param[in,out] longest Pal ndromo actual m s largo (puede actualizarse).
 * @param[in] candidate Nuevo pal ndromo candidato a comparar.
 */
void replace_longest(char **longest, const char *candidate);
```

Durante todo el desarrollo, se utilizaron técnicas seguras de asignación (`malloc`) y liberación (`free`) de memoria dinámica, validando mediante la herramienta **Valgrind** que no existieran fugas de memoria ni errores relacionados con accesos inválidos.

Finalmente, tras finalizar la lectura y procesamiento de todas las palabras, el programa mostró en pantalla el palíndromo más largo encontrado en el archivo. Además, se implementaron controles robustos para asegurar que el programa se comporte correctamente aún en casos límite.

3. Resultados

Ejercicio 1: Búsqueda de secuencia más larga de unos en una matriz binaria mediante aritmética de punteros

A continuación se presentan los resultados obtenidos después de la compilación y ejecución del programa `ejercicio1.c` en diferentes escenarios:

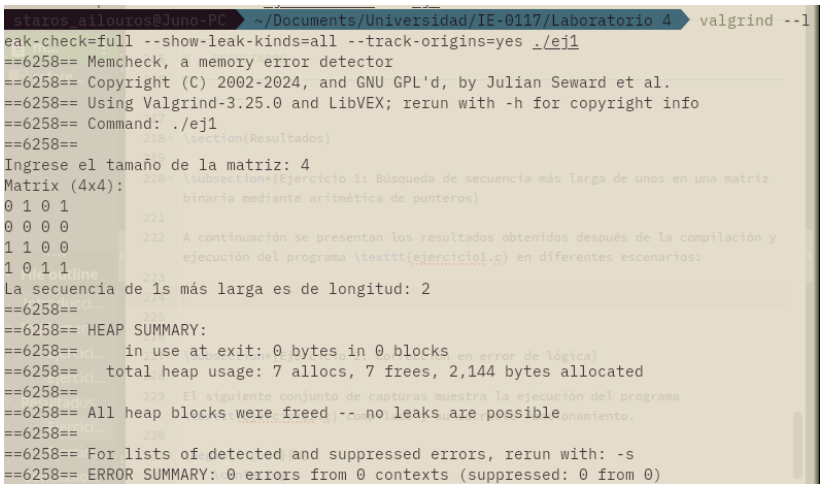

Figura 1: Caso 1: Ejecución del programa con matriz 4X4

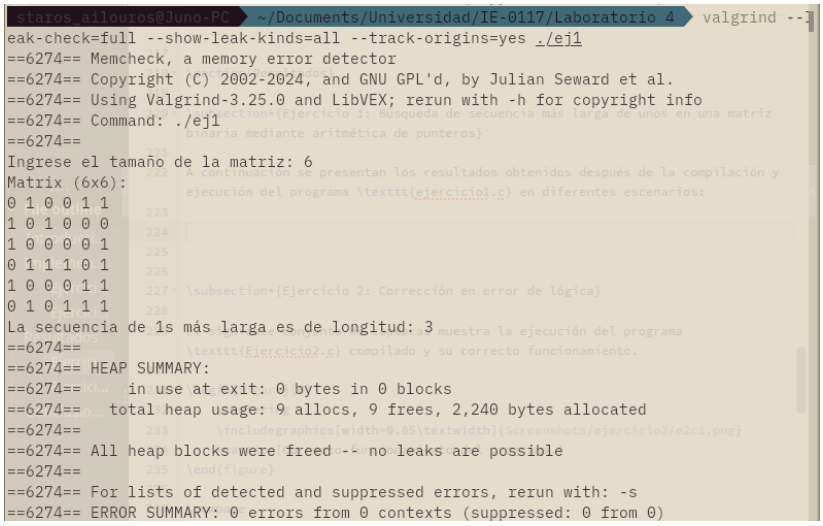

Figura 2: Caso 2: Ejecución del programa con matriz 6X6

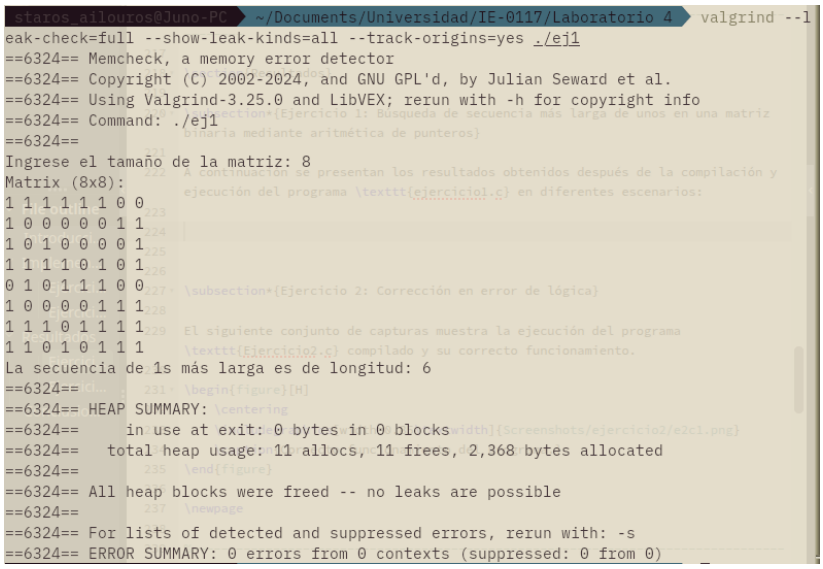

Figura 3: Caso 3: Ejecución del programa con matriz 8X8

Ejercicio 2: Identificación del palíndromo más largo desde un archivo de texto

El siguiente conjunto de capturas muestra la ejecución del programa `ejercicio2.c` compilado y su correcto funcionamiento.

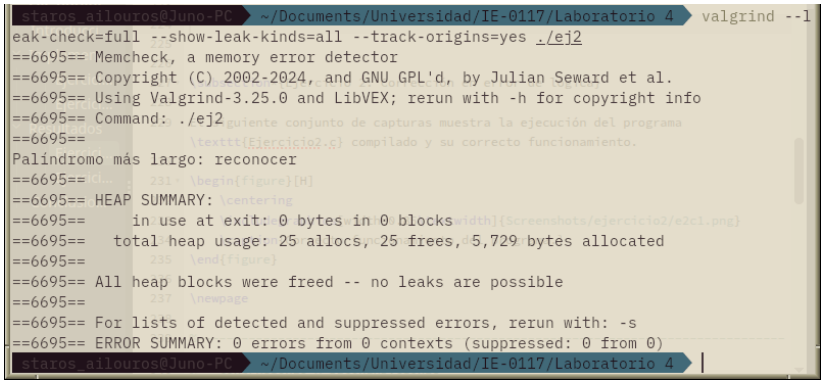

Figura 4: Resultado obtenido del análisis de palíndromo

4. Conclusiones y Recomendaciones

Conclusiones

- Se logró implementar exitosamente ambos ejercicios cumpliendo todos los requisitos planteados, destacando especialmente la gestión segura y eficiente de memoria dinámica, validada mediante **Valgrind**.
- El uso exclusivo de aritmética de punteros para el manejo de matrices demostró ser un enfoque viable y eficiente, permitiendo flexibilidad y continuidad en la búsqueda de secuencias.
- El uso adecuado de estructuras condicionales y guardas en la programación permitió evitar errores comunes como accesos inválidos o fugas de memoria, fortaleciendo así la robustez del programa.

Recomendaciones

- Continuar aplicando las buenas prácticas utilizadas en este laboratorio en futuros desarrollos, especialmente la validación constante con herramientas como **Valgrind**, para asegurar código robusto, eficiente y seguro.
- Mantener la práctica de documentación detallada mediante comentarios estructurados, particularmente en proyectos colaborativos, facilitando así el trabajo en equipo y la claridad del código.
- Para ejercicios futuros, se recomienda también incluir pruebas automatizadas o casos de prueba más exhaustivos que permitan validar aún más rigurosamente el comportamiento correcto del software desarrollado bajo diferentes condiciones y casos límite.

Referencias

1. The Open Group, "The Open Group Base Specifications Issue 7, 2018 edition (POSIX.1-2017)," 2018. Accedido: 16 de mayo de 2025.

2. Valgrind Developers, "Valgrind Documentation," 2024. Accedido: 16 de mayo de 2025.

3. IBM Corporation, "C and c++ language support," 2023. Accedido: 16 de mayo de 2025.