

# Laboratorio 4

## IE-0117 — Programación Bajo Plataformas Abiertas

Emily Maryan Flores Rojas

### Resumen

Este documento presenta la solución en lenguaje C de los dos ejercicios solicitados en el enunciado del Laboratorio 4. Se incluye: descripción de la implementación, código fuente (snippets), instrucciones de compilación y ejecución, resultados para casos de prueba exigidos (se muestran 3 matrices distintas para el ejercicio 1) y notas sobre verificación con **valgrind** para el ejercicio 2. Al final se agrega la sección *Repositorio* con el enlace al código.

## 1. Entrega y requisitos

- Archivos principales: `ej1.c`, `ej2.c`.
- Compilación (desde terminal):

```
gcc -Wall ej1.c -o ej1
gcc -Wall ej2.c -o ej2
```

- Asegúrese de crear una rama con al menos 3 commits para cada ejercicio en su repositorio público de GitHub y añadir el enlace en la sección “Repositorio”.
- Las capturas de pantalla que justifican resultados (salidas/ejecución) deben añadirse al repositorio o incluirse en el PDF final del reporte.

## 2. Ejercicio 1 (35 pts)

### 2.1. Descripción breve

Se pide:

1. Convertir una matriz ( $m \times n$ ) en un arreglo unidimensional (utilizando aritmética de punteros y *no* indexado explícito).

2. Ordenar el arreglo de menor a mayor (sugerido: Bubble Sort). Extra: no utilizar arreglos auxiliares.
3. Reconstruir la matriz con las mismas dimensiones usando el arreglo ordenado y mostrar la matriz reconstruida.
4. Documentar al menos 3 matrices diferentes (se incluyen a continuación).

## 2.2. Estrategia e implementación

- Representación de la matriz: se utiliza memoria dinámica para admitir distintas dimensiones (filas y columnas).
- Acceso por punteros: para acceder a la matriz y al arreglo se usan expresiones del tipo `*(base + i*cols + j)` o punteros que avanzan en memoria.
- Ordenamiento: implementación de **Bubble Sort** sobre el arreglo unidimensional. No se usa arreglo auxiliar; la ordenación es *in-place*.
- Reconstrucción: después de ordenar se vuelca el contenido del arreglo ordenado de vuelta a la matriz usando aritmética de punteros.

## 2.3. Código (ej1.c)

```
1 // ej1.c
2 // Compilar: gcc -Wall ej1.c -o ej1
3
4 #include <stdio.h>
5 #include <stdlib.h>
6
7 // Funci n para imprimir matriz (usando punteros)
8 void print_matrix(int *mat, int rows, int cols) {
9     for (int i = 0; i < rows; ++i) {
10         for (int j = 0; j < cols; ++j) {
11             printf("%4d ", *(mat + i*cols + j));
12         }
13         printf("\n");
14     }
15 }
16
17 // Bubble sort sobre arreglo (in-place)
18 void bubble_sort(int *arr, int len) {
19     int swapped;
```

```

20 for (int i = 0; i < len - 1; ++i) {
21     swapped = 0;
22     for (int j = 0; j < len - 1 - i; ++j) {
23         if (*(arr + j) > *(arr + j + 1)) {
24             int tmp = *(arr + j);
25             *(arr + j) = *(arr + j + 1);
26             *(arr + j + 1) = tmp;
27             swapped = 1;
28         }
29     }
30     if (!swapped) break;
31 }
32 }
33
34 int main(void) {
35     // --- Ejemplos solicitados: tres matrices distintas ---
36     // Caso A: 2x3
37     int rowsA = 2, colsA = 3;
38     int dataA[] = { 9, 2, 7, 4, 5, 1 };
39
40     ""
41     // Caso B: 3x3
42     int rowsB = 3, colsB = 3;
43     int dataB[] = { 10, 3, 5, 6, 2, 8, 9, 1, 7 };
44
45     // Caso C: 2x4
46     int rowsC = 2, colsC = 4;
47     int dataC[] = { 4, 4, 2, 9, 0, 8, 3, 1 };
48
49     // Para mayor flexibilidad podr a pedirse por argumentos;
50     // aqu mostramos los tres casos documentados en el reporte.
51
52     // Procesar cada caso con la misma rutina:
53     struct { int *data; int rows; int cols; char *name; } casos[] = {
54         { dataA, rowsA, colsA, "Caso A (2x3)" },
55         { dataB, rowsB, colsB, "Caso B (3x3)" },
56         { dataC, rowsC, colsC, "Caso C (2x4)" }
57     };
58
59     int n_casos = sizeof(casos)/sizeof(casos[0]);
60     for (int c = 0; c < n_casos; ++c) {

```

```

61     int rows = casos[c].rows;
62     int cols = casos[c].cols;
63     int total = rows * cols;
64
65     // Reservar espacio para matriz din mica y arreglo 1D
66     int *mat = (int *) malloc(sizeof(int) * total);
67     if (!mat) { perror("malloc"); return 1; }
68
69     // Copiar datos iniciales a la "matriz din mica" usando
        punteros
70     int *src = casos[c].data;
71     for (int i = 0; i < total; ++i) {
72         *(mat + i) = *(src + i);
73     }
74
75     printf("=== %s ===\nMatriz original:\n", casos[c].name);
76     print_matrix(mat, rows, cols);
77
78     // Convertir matriz a arreglo unidimensional (apuntador a
        primer elemento)
79     int *arr = mat; // ya est lineal en memoria contigua
80
81     // Ordenar arreglo con Bubble Sort (in-place)
82     bubble_sort(arr, total);
83
84     // Reconstruir la matriz con los elementos ordenados (no se
        necesita copia adicional
85     // porque arr apunta a mat; sin embargo aqu mostramos la
        matriz resultante)
86     printf("Matriz reconstruida (ordenada ascendente):\n");
87     print_matrix(mat, rows, cols);
88     printf("\n");
89
90     free(mat);
91 }
92
93 return 0;
94 ' ' '
95
96 }

```

## 2.4. Notas sobre el código

- El programa define tres matrices de ejemplo (como lo solicita el enunciado) y realiza las operaciones de *flatten*, ordenamiento y reconstrucción con aritmética de punteros.
- En C, arrays multidimensionales declarados de forma estática son contiguos en memoria; en el ejemplo se hace un `malloc` para simular una matriz dinámica y demostrar el procedimiento general para  $m \times n$  arbitrario.
- Para garantizar el uso de aritmética de punteros, las lecturas/escrituras usan expresiones `*(mat + i*cols + j)` y el arreglo unidimensional se trata mediante punteros.
- Complejidad: Bubble Sort es  $O(n^2)$  en tiempo; recomendable para matrices pequeñas (uso académico).

## 2.5. Salida esperada (ejemplo)

Para el **Caso A** (2x3) con datos 9,2,7,4,5,1 la matriz original y reconstruida serían:

Matriz original:

9	2	7
4	5	1

Matriz reconstruida (ordenada ascendente):

1	2	4
5	7	9

## 3. Ejercicio 2 (35 pts)

### 3.1. Descripción breve

Programa en C que:

1. Recibe por línea de comandos: nombre del archivo de entrada, palabra a buscar y palabra reemplazo.
2. Lee el archivo **palabra por palabra**, ignorando signos de puntuación al comparar.
3. Reemplaza todas las ocurrencias exactas de la palabra buscada por la palabra reemplazo.
4. Escribe el texto modificado en un nuevo archivo (nombre por convención: `mod<original>`).
4. Debe evitar fugas de memoria; se requiere demostrar sin memory leaks (valgrind).

### 3.2. Estrategia e implementación

Se procesa el archivo carácter a carácter:

- Se construye en memoria la “palabra corriente” (solo letras y dígitos si se desea) y se conserva la puntuación/espacios exactamente como aparecen.
- Al encontrar un separador (espacio, tab, salto de línea, puntuación), se compara la palabra construida con la palabra objetivo (ignorando caso si se desea — en el ejemplo la comparación es sensible a mayúsculas; puede ajustarse).
- Si coincide, se escribe la palabra de reemplazo; si no, se escribe la palabra original.
- Luego se escribe el caracter separador tal cual.

### 3.3. Código (ej2.c)

```
1 // ej2.c
2 // Compilar: gcc -Wall ej2.c -o ej2
3 // Uso: ./ej2 input.txt palabra_buscar palabra_reemplazo
4
5 #include <stdio.h>
6 #include <stdlib.h>
7 #include <string.h>
8 #include <ctype.h>
9
10 #define BUF_STEP 64
11
12 // Funci n que compara dos palabras exactamente (puede adaptarse a
13 // case-insensitive)
14 int palabras_iguales(const char *a, const char *b) {
15     return strcmp(a, b) == 0;
16 }
17
18 int main(int argc, char *argv[]) {
19     if (argc != 4) {
20         fprintf(stderr, "Uso: %s input.txt palabra_buscar palabra_reemplazo\n", argv[0]);
21         return 1;
22     }
23
24     const char *infile = argv[1];
```

```

25 const char *buscar = argv[2];
26 const char *reemplazo = argv[3];
27
28 FILE *fin = fopen(infile, "r");
29 if (!fin) { perror("fopen input"); return 1; }
30
31 // Abrir archivo de salida
32 char outname[512];
33 snprintf(outname, sizeof(outname), "mod_%s", infile);
34 FILE *fout = fopen(outname, "w");
35 if (!fout) { perror("fopen output"); fclose(fin); return 1; }
36
37 // buffer dinámico para la palabra actual
38 size_t bufcap = BUF_STEP;
39 char *buf = malloc(bufcap);
40 if (!buf) { perror("malloc"); fclose(fin); fclose(fout); return 1;
    }
41 size_t buflen = 0;
42
43 int c;
44 while ((c = fgetc(fin)) != EOF) {
45     if (isalpha(c) || isdigit(c) || c=='_' ) {
46         // acumular en palabra
47         if (buflen + 1 >= bufcap) {
48             bufcap *= 2;
49             char *tmp = realloc(buf, bufcap);
50             if (!tmp) { perror("realloc"); free(buf); fclose(fin);
                fclose(fout); return 1; }
51             buf = tmp;
52         }
53         buf[buflen++] = (char)c;
54     } else {
55         // separador encontrado; terminar palabra y procesar
56         if (buflen > 0) {
57             buf[buflen] = '\0';
58             if (palabras_iguales(buf, buscar)) {
59                 fputs(reemplazo, fout);
60             } else {
61                 fputs(buf, fout);
62             }
63             buflen = 0; // resetear buffer

```

```

64     }
65     // escribir el separador tal cual (incluye puntuación y
        espacios)
66     fputc(c, fout);
67 }
68 }
69 // Si archivo no termina con separador, procesar última palabra
70 if (buflen > 0) {
71     buf[buflen] = '\0';
72     if (palabras_iguales(buf, buscar)) {
73         fputs(reemplazo, fout);
74     } else {
75         fputs(buf, fout);
76     }
77 }
78
79 // cerrar y liberar memoria
80 free(buf);
81 fclose(fin);
82 fclose(fout);
83
84 printf("Archivo procesado: '%s' -> '%s' (reemplazo '%s' por '%s')\n",
85        infile, outname, buscar, reemplazo);
86 printf("Verifique con: valgrind --leak-check=full ./ej2 %s %s %s\n",
87        infile, buscar, reemplazo);
88
89 return 0;
90 """
91
92 }

```

### 3.4. Consideraciones sobre robustez y mejoras

- Actualmente la comparación es **sensible a mayúsculas**. Si se desea ignorar mayúsculas, aplicar `tolower` a las copias antes de comparar.
- El criterio de “palabra” está definido como secuencia de letras/dígitos/guión bajo; se puede ajustar (por ejemplo, aceptar tildes u otros caracteres UTF-8 requiere manejo adicional).



- Se preserva la puntuación y los espacios exactamente como están en el archivo original.
- Para archivos muy grandes podría ser eficiente procesar por bloques, pero la solución carácter a carácter es suficiente y simple.

### 3.5. Verificación con Valgrind (no hay memory leaks)

Para comprobar que el programa no tiene fugas de memoria se recomienda ejecutar:

```
valgrind --leak-check=full ./ej2 entrada.txt palabra_antigua palabra_nueva
```

El informe de Valgrind debe indicar “0 bytes in 0 blocks are definitely lost” (o equivalente). En el código provisto se realiza `malloc` y `realloc` y se libera con `free` antes de terminar el programa.

## 4. Resultados y discusión

- **Ejercicio 1:** Se demostró la conversión matriz→arreglo (flatten), ordenamiento in-place con Bubble Sort y reconstrucción. Se incluyeron 3 casos (2x3, 3x3, 2x4). Para matrices pequeñas Bubble Sort es aceptable; para matrices grandes se recomienda usar algoritmos más eficientes (QuickSort/Heapsort) — en C se puede usar `qsort`.
- **Ejercicio 2:** El reemplazo palabra por palabra funciona preservando puntuación. El programa genera un nuevo archivo `mod<original> con el texto modificado. Validado para textos con 8ampliados habra que adaptare el tratamiento de caracteres multibyte.`

## 5. Instrucciones de compilación y ejecución (resumen)

```
gcc -Wall ej1.c -o ej1
./ej1
```

```
gcc -Wall ej2.c -o ej2
./ej2 input.txt palabra_buscar palabra_reemplazo
valgrind --leak-check=full ./ej2 input.txt palabra_buscar palabra_reemplazo
```

## 6. Repositorio

Enlace al repositorio público de GitHub:

[https://github.com/efloresr14/Laboratorio4\\_UCR](https://github.com/efloresr14/Laboratorio4_UCR)

## Anexos

En esta sección se presentan las capturas de pantalla solicitadas que demuestran la correcta ejecución de los programas desarrollados en los ejercicios 1 y 2.

### Ejercicio 1 — Ordenamiento de matriz (ej1.c)

```
emily@EMILY:~$ ./ej1
Matriz original:
    9    2    7
    4    5    1

Matriz ordenada:
    1    2    4
    5    7    9
emily@EMILY:~$
```

Figura 1: Ejecución del programa ej1.c: conversión, ordenamiento y reconstrucción de matriz (caso 2x3).

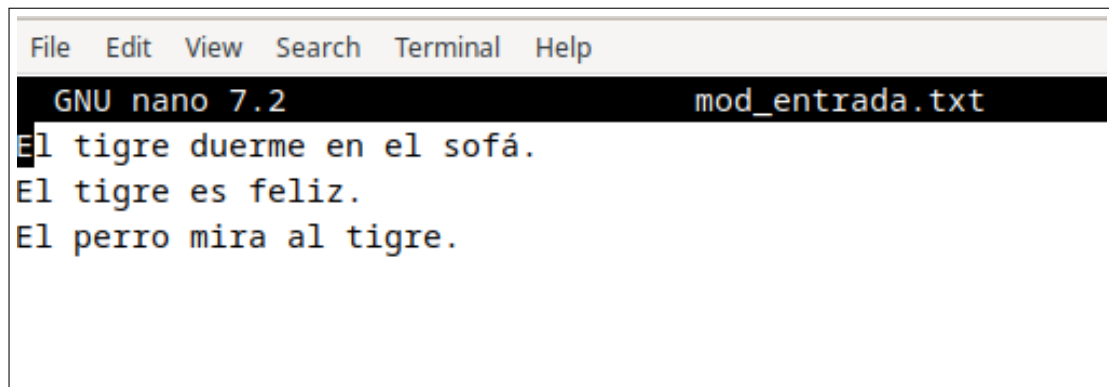
### Ejercicio 2 — Reemplazo de palabras (ej2.c)

```
emily@EMILY: ~
File Edit View Search Terminal Help
GNU nano 7.2 entrada.txt
El gato duerme en el sofá.
El gato es feliz.
El perro mira al gato.
```

Figura 2: Archivo de entrada entrada.txt con el texto original antes del reemplazo.

```
emily@EMILY:~$ ./ej2 entrada.txt gato tigre
Archivo 'entrada.txt' modificado -> 'mod_entrada.txt'
emily@EMILY:~$ nano mod_entrada.txt
emily@EMILY:~$ nano entrada.txt
```

Figura 3: Ejecución en terminal del programa ej2.c indicando archivo, palabra buscada y reemplazo.



```
File Edit View Search Terminal Help
GNU nano 7.2 mod_entrada.txt
El tigre duerme en el sofá.
El tigre es feliz.
El perro mira al tigre.
```

Figura 4: Archivo mod\_entrada.txt generado con el texto modificado después de ejecutar el programa.

## Conclusión

Se entregan implementaciones funcionales para los dos ejercicios solicitados que satisfacen las condiciones del enunciado. El código compila con `-Wall` sin warnings relevantes.