

Session 7

C program for matrix multiplication using pointers

This session produces a working C program: it reads two matrices, checks that they can be multiplied, multiplies them using pointer arithmetic only, and prints the product. It is the first coding session in MCS-217 and the program you write here is the thing you test in Session 8. Write it cleanly, because every test case in the next session refers to a function in this file by name. The manual places this under Testing for a reason: a program with clear functions and validation is a program you can test.

Objectives

✗ Do not copy. Read for understanding and the viva

- Store a two-dimensional matrix in one contiguous block and reach any element with pointer arithmetic
- Allocate matrices at run time with `malloc` and release them with `free`
- Validate dimensions before reading elements, so the program never multiplies incompatible matrices
- Split the program into small functions (`read_matrix` , `multiply` , `print_matrix` , `free_matrix`) that Session 8 can test one at a time
- Compile with `gcc -Wall -Wextra` and get zero warnings

Problem Statement

■ Write in lab record

Write a program in 'C' language for the multiplication of two matrices using pointers.

Concept

✗ Do not copy. Read for understanding and the viva

Row-major layout

C stores a two-dimensional array row after row in memory. A 2×3 matrix occupies six consecutive integers: row 0 first, then row 1.

```

1  index:   0    1    2    3    4    5
2  value: a00  a01  a02  a10  a11  a12
3          |--- row 0 ---| |--- row 1 ---|
```

If `m` points to the first element and the matrix has `cols` columns, element `(i, j)` is at offset `i * cols + j`. So `*(m + i * cols + j)` is the same value as `m[i][j]` would be in a static array. The program never writes `m[i][j]`; it only moves pointers.

Why one block and not an array of row pointers

Two layouts are common: one `malloc` of `rows * cols` ints, or an array of `rows` pointers each pointing to its own `malloc`. The single block is simpler to allocate, simpler to free (one call), and matches how the compiler lays out a static `int a[2][3]`. It also makes the inner loop of multiplication a plain pointer walk, which is the point of this session.

Walking a column with a stride

The product `C = A x B` needs, for each `C(i, j)`, the dot product of row `i` of `A` with column `j` of `B`. Row `i` of `A` is contiguous, so a pointer that starts at `a + i * c1` and increments by 1 walks it. Column `j` of `B` is not contiguous: consecutive elements are `c2` apart. A pointer that starts at `b + j` and increments by `c2` walks down the column. That stride is the only non-obvious piece of pointer arithmetic in the program.

Why the dimension check comes first

`A` ($r_1 \times c_1$) times `B` ($r_2 \times c_2$) is defined only when `c1 = r2`. The result is $r_1 \times c_2$. If you skip the check, the pointer walk reads past the end of the block and the output is garbage or a crash. The program therefore reads the four dimensions, checks them, and only then asks for elements. Each dimension is also limited to 1 to 100, which rules out zero, negative, and absurdly large sizes.

malloc and free in pairs

Every `malloc` needs exactly one `free` on every path out of `main`, including the error paths. The program frees `a` when reading `b` fails, and frees `a` and `b` when allocating `c` fails. `free_matrix` is a thin wrapper so the test cases in Session 8 have a named function to call.

Program

■ Write in lab record

matrix_multiply.c

```
1  /*
2   * matrix_multiply.c
3   * MCS-217 Session 7: multiplication of two matrices using pointers.
4   *
5   * Each matrix is stored as one contiguous block of rows * cols ints
6   * (row-major order). Element (i, j) lives at *(m + i * cols + j).
7   * No arr[i][j] indexing is used anywhere; every access goes through
8   * pointer arithmetic.
9   *
10  * Build: gcc -Wall -Wextra -o matrix_multiply matrix_multiply.c
11  */
12  #include <stdio.h>
13  #include <stdlib.h>
14
15  #define MAX_DIM 100
16
17  /* Read one positive dimension. Returns 0 on failure. */
18  static int read_dim(const char *label, int *out)
19  {
20      printf("%s", label);
21      if (scanf("%d", out) != 1) {
22          printf("Error: dimension must be an integer.\n");
23          return 0;
24      }
25      if (*out < 1 || *out > MAX_DIM) {
26          printf("Error: dimension must be between 1 and %d.\n", MAX_DIM);
27          return 0;
28      }
29      return 1;
30  }
31
32  /* Allocate rows * cols ints and fill them from stdin, row by row.
33   Returns NULL if allocation or input fails. */
34  int *read_matrix(const char *name, int rows, int cols)
35  {
36      int *m = malloc((size_t)rows * (size_t)cols * sizeof(int));
37      int *p;
38      int i, j;
39
40      if (m == NULL) {
41          printf("Error: out of memory.\n");
42          return NULL;
43      }
```

```
44     printf("Enter %d x %d elements of matrix %s, row by row:\n", rows, cols,
45     p = m;
46     for (i = 0; i < rows; i++) {
47         for (j = 0; j < cols; j++) {
48             if (scanf("%d", p) != 1) {
49                 printf("Error: element (%d,%d) is not an integer.\n", i + 1,
50                 free(m);
51                 return NULL;
52             }
53             p++; /* next element in row-major order */
54         }
55     }
56     return m;
57 }
58
59 /* C = A x B where A is r1 x c1 and B is c1 x c2. C must hold r1 * c2 ints.
60 Only pointer arithmetic is used to walk the three matrices. */
61 void multiply(const int *a, const int *b, int *c, int r1, int c1, int c2)
62 {
63     int i, j, k;
64     const int *arow; /* start of row i of A */
65     const int *bp; /* walks down column j of B */
66     int *cp = c; /* walks C in row-major order */
67     long sum;
68
69     for (i = 0; i < r1; i++) {
70         arow = a + i * c1;
71         for (j = 0; j < c2; j++) {
72             sum = 0;
73             bp = b + j; /* B(0, j) */
74             for (k = 0; k < c1; k++) {
75                 sum += (long)*(arow + k) * *bp; /* A(i,k) * B(k,j) */
76                 bp += c2; /* down one row in B */
77             }
78             *cp = (int)sum;
79             cp++;
80         }
81     }
82 }
83
84 /* Print a rows x cols matrix, one row per line. */
85 void print_matrix(const char *title, const int *m, int rows, int cols)
86 {
87     const int *p = m;
```

```
88     int i, j;
89
90     printf("%s (%d x %d):\n", title, rows, cols);
91     for (i = 0; i < rows; i++) {
92         for (j = 0; j < cols; j++) {
93             printf("%6d", *p);
94             p++;
95         }
96         printf("\n");
97     }
98 }
99
100
101 0 /* Release a matrix. Safe to call with NULL. */
102
103 1 void free_matrix(int *m)
104
105 2 {
106
107 3     free(m);
108
109 4 }
110
111 5
112 6 int main(void)
113
114 7 {
115
116 8     int r1, c1, r2, c2;
117
118 9     int *a = NULL, *b = NULL, *c = NULL;
119
120 0
121 11
122 1     if (!read_dim("Rows of A: ", &r1) || !read_dim("Columns of A: ", &c1) ||
123 11
124 2         !read_dim("Rows of B: ", &r2) || !read_dim("Columns of B: ", &c2))
125 11
126 3         return 1;
127
128 11
129 4
130 11
131 5     if (c1 != r2) {
```

```
11
6     printf("Error: columns of A (%d) must equal rows of B (%d). "
11
7         "Multiplication not possible.\n", c1, r2);
11
8     return 1;
11
9 }
12
0
12
1     a = read_matrix("A", r1, c1);
12
2     if (a == NULL)
12
3         return 1;
12
4     b = read_matrix("B", r2, c2);
12
5     if (b == NULL) {
12
6         free_matrix(a);
12
7         return 1;
12
8     }
12
9     c = malloc((size_t)r1 * (size_t)c2 * sizeof(int));
13
0     if (c == NULL) {
13
1         printf("Error: out of memory.\n");
13
2         free_matrix(a);
13
3         free_matrix(b);
13
4         return 1;
13
5     }
13
6
13
7     multiply(a, b, c, r1, c1, c2);
```

```
13
8
13
9     print_matrix("Matrix A", a, r1, c1);
14
0     print_matrix("Matrix B", b, r2, c2);
14
1     print_matrix("Product A x B", c, r1, c2);
14
2
14
3     free_matrix(a);
14
4     free_matrix(b);
14
5     free_matrix(c);
14
6     return 0;
14
7 }
```

Build

```
1 gcc -Wall -Wextra -o matrix_multiply matrix_multiply.c
2 ./matrix_multiply
```

Sample Output

Input: A is 2×3 , B is 3×2 . Elements typed row by row.

```

1 Rows of A: 2
2 Columns of A: 3
3 Rows of B: 3
4 Columns of B: 2
5 Enter 2 x 3 elements of matrix A, row by row:
6 1 2 3
7 4 5 6
8 Enter 3 x 2 elements of matrix B, row by row:
9 7 8
10 9 10
11 11 12
12 Matrix A (2 x 3):
13     1     2     3
14     4     5     6
15 Matrix B (3 x 2):
16     7     8
17     9    10
18    11    12
19 Product A x B (2 x 2):
20    58    64
21   139   154

```

Check by hand: $C(0,0) = 1*7 + 2*9 + 3*11 = 58$, $C(1,1) = 4*8 + 5*10 + 6*12 = 154$.

Incompatible dimensions (A is 2×3 , B is 2×2):

```

1 Rows of A: 2
2 Columns of A: 3
3 Rows of B: 2
4 Columns of B: 2
5 Error: columns of A (3) must equal rows of B (2). Multiplication not possible

```

Zero size:

```

1 Rows of A: 0
2 Error: dimension must be between 1 and 100.

```

Explanation of the key lines

Line	What it does
<code>int *m = malloc((size_t)rows * (size_t)cols * sizeof(int));</code>	One contiguous block for the whole matrix. The casts to <code>size_t</code> stop the multiplication overflowing before it reaches <code>malloc</code> .
<code>if (scanf("%d", p) != 1) then p++</code>	<code>scanf</code> writes straight into the block through <code>p</code> ; <code>p++</code> moves to the next element in row-major order. No index variable is used to address memory.
<code>arow = a + i * c1;</code>	Start of row <code>i</code> of <code>A</code> .
<code>bp = b + j; then bp += c2;</code>	Start at <code>B(0, j)</code> and step one full row each time, which walks down column <code>j</code> .
<code>sum += (long)*(arow + k) * *bp;</code>	<code>A(i,k) * B(k,j)</code> accumulated in a <code>long</code> so intermediate sums do not overflow for typical inputs.
<code>*cp = (int)sum; cp++;</code>	Store <code>c(i,j)</code> and advance to the next output cell. <code>cp</code> walks <code>c</code> in the same row-major order the loops produce.
<code>printf("%6d", *p); p++;</code>	Print with a fixed width so columns line up, then advance.
<code>free_matrix(a); free_matrix(b); free_matrix(c);</code>	One free per malloc.

Complexity

Metric	Value	Reason
Time	$O(r1 \times c1 \times c2)$	Three nested loops; for square $n \times n$ matrices this is $O(n^3)$.
Space	$O(r1 \times c1 + r2 \times c2 + r1 \times c2)$	The two inputs and the product. No extra buffers.
Input	$O(r1 \times c1 + r2 \times c2)$	Every element is read exactly once.

Viva Questions

✗ Do not copy. Read for understanding and the viva

- **Q:** Why is `*(m + i * cols + j)` equal to `m[i][j]`? **A:** Because C stores rows consecutively, so row `i` starts `i * cols` elements after the base pointer and column `j` is `j` more.
- **Q:** Why does `bp` increase by `c2` and not by 1? **A:** It walks down a column of `B`; consecutive elements in a column are one full row, `c2` ints, apart.
- **Q:** What happens if `c1` is not equal to `r2` and you skip the check? **A:** The pointer walk reads beyond the allocated block: undefined behaviour, usually garbage or a crash.
- **Q:** Why accumulate `sum` in a `long`? **A:** Products of two ints can exceed the int range; a wider accumulator delays overflow. The final cast to `int` can still truncate for very large values.
- **Q:** Why is `free_matrix` a separate function when it just calls `free`? **A:** It names the operation so it can be tested and replaced, and it keeps `main` symmetrical with `read_matrix`.
- **Q:** What does `scanf("%d", p) != 1` detect? **A:** Non-numeric input or end of input; both mean the element could not be read.
- **Q:** Why cast to `size_t` inside `malloc`? **A:** `rows * cols * sizeof(int)` must be computed in an unsigned type wide enough for the result; multiplying two ints first could overflow.
- **Q:** What is the time complexity for square $n \times n$ matrices? **A:** $O(n^3)$.

Common Mistakes

✗ Do not copy. Read for understanding and the viva

- Writing `a[i][j]` inside `multiply` and calling it a pointer program. The manual asks for pointers; use pointer arithmetic throughout.
- Reading elements before checking `c1 == r2`, so the user types a whole matrix and only then sees the error.
- Forgetting to `free` on the error paths. Every `return 1` after a `malloc` must release what was allocated.
- Using `int` for `rows * cols * sizeof(int)`, which can overflow before `malloc` sees it.
- Printing with `%d` and no width, so columns do not line up and the examiner cannot read the result.
- Leaving compiler warnings. `gcc -Wall -Wextra` must be clean; an unused variable or a signed/unsigned comparison is a defect you fix, not ignore.

Session Summary

■ Write in lab record

- Source listing of `matrix_multiply.c` with the file header comment
- The build command and a note that it compiles with no warnings
- Sample run for a compatible pair (2×3 times 3×2) with the input and the full output
- Sample run showing the dimension mismatch error
- The table explaining the pointer arithmetic lines
- The complexity table

 [Edit this page](#)

[< Previous](#)
Session 6

Session 8 [Next >](#)