

Session 8

Unit, module, and integration test cases for the matrix multiplication program

This session produces a set of test cases for the program written in Session 7, organised at three levels: unit tests for each function, module tests for the multiplication module as a whole, and integration tests for the complete program with its input and output. Every test case has an expected output written before the run and an actual output recorded after it, so the record shows evidence, not intentions. The program is small, but the same table structure is what you would use for the Railway Reservation System in Session 15.

Objectives

✕ Do not copy. Read for understanding and the viva

- Tell the three levels apart: what a unit, a module, and an integration test each check
- Write a test case with a precondition, an input, an expected output, and a status
- Cover boundary values (1×1 , the maximum dimension), invalid input (zero, mismatch, letters), and large values
- Run every case, record the actual output, and report defects honestly
- Produce a test summary and a defect log the examiner can verify against the program

Problem Statement

■ Write in lab record

Develop a set of test cases that will completely test the program in session 7. The test case should be separately developed for Unit testing, Module testing and Integration testing.

Concept

✕ Do not copy. Read for understanding and the viva

The three levels

Unit testing checks one function in isolation, called directly from a small test driver, with the rest of the program out of the way. Module testing checks a group of functions that work together, here the

multiplication module: `read_dim`, `read_matrix`, `multiply`, `print_matrix`, and `free_matrix` operating on real allocated matrices. Integration testing checks the whole program as the user runs it: input through `stdin`, output on `stdout`, exit status.

How unit tests are run on a C program

`main` reads from the keyboard, so a unit test cannot go through it. The driver is a second C file that includes `matrix_multiply.c` with `main` renamed (`#define main prog_main` before the include) and then calls `multiply`, `print_matrix`, and `free_matrix` on arrays it builds itself. For `read_matrix` and `read_dim`, the driver is fed input through a pipe: `printf '1 2\n3 4\n' | ./driver`. Record the driver command with the test case so the examiner can repeat it.

Choosing inputs

For every function, pick one normal case, the smallest legal size, an invalid input, and, where numbers are involved, a large value. The point of the large value is to find the overflow limit of `int`, not to avoid it. A test that only uses 1, 2, 3 proves very little.

What “actual output” means

Copy the real output, not a paraphrase. If a case fails, the actual output column shows what went wrong, and the defect log explains it. A test record with 100 percent pass and no defects on a first run is usually a sign of weak tests, not perfect code.

Unit Test Cases

■ Write in lab record

Driver: `ut.c` includes `matrix_multiply.c` with `main` renamed, builds arrays in code, and calls each function. Built with `gcc -Wall -Wextra -o ut ut.c`.

TC id	Function	Precondition	Input	Expected output	Actual output	Status
UT-01	read_dim	stdin piped	5	returns 1, stores 5	ok=1 d=5	Pass
UT-02	read_dim	stdin piped	100 (upper bound)	returns 1, stores 100	ok=1 d=100	Pass
UT-03	read_dim	stdin piped	0	prints range error, returns 0	"Error: dimension must be between 1 and 100." ok=0	Pass
UT-04	read_dim	stdin piped	101	prints range error, returns 0	"Error: dimension must be between 1 and 100." ok=0	Pass
UT-05	read_dim	stdin piped	abc	prints integer error, returns 0	"Error: dimension must be an integer." ok=0	Pass
UT-06	read_dim	stdin piped	2.5	rejected as not a whole number	ok=1 d=2, .5 left in buffer	Fail (D-02)
UT-07	read_matrix	rows=2, cols=2, stdin piped	1 2 newline 3 4	block holds 1 2 3 4 in row-major order	stored: 1 2 3 4	Pass
UT-08	read_matrix	rows=2, cols=2, stdin piped	1 2 x 4	error naming element (2,1), returns NULL	"Error: element (2,1) is not an integer."	Pass

TC id	Function	Precondition	Input	Expected output	Actual output	Status
					returned NULL	
UT-09	read_matrix	rows=2, cols=2, stdin piped	1 2 then end of input	error, returns NULL, no crash	"Error: element (2,1) is not an integer." returned NULL	Pass (see D-03)
UT-10	read_matrix	rows=2, cols=2, stdin piped	9 -9 0 2147483647	negative, zero and INT_MAX stored exactly	stored: 9 -9 0 2147483647	Pass
UT-11	multiply	a = 1 2 3 4 5 6 (2 × 3), b = 7 8 9 10 11 12 (3 × 2), c allocated 2 x 2	r1=2 c1=3 c2=2	c = 58 64 139 154	58 64 139 154	Pass
UT-12	multiply	a = 7, b = 6 (1 x 1 each)	r1=c1=c2=1	c = 42	42	Pass
UT-13	multiply	a = identity 2 x 2, b = 5 6 7 8	r1=c1=c2=2	c equals b: 5 6 7 8	5 6 7 8	Pass
UT-14	multiply	a = all zeros 2 × 2, b = 5 6 7 8	r1=c1=c2=2	c = 0 0 0 0	0 0 0 0	Pass
UT-15	multiply	a = -3 4 (1 x 2), b = 2 -5 (2 × 1)	r1=1 c1=2 c2=1	c = -26	-26	Pass
UT-16	multiply	a = 100000, b = 100000 (1 x 1)	r1=c1=c2=1	10000000000 or an overflow error	1410065408	Fail (D-01)

TC id	Function	Precondition	Input	Expected output	Actual output	Status
UT-17	print_matrix	m = 7 (1 × 1)	title "UT-P1"	header line then 7	"UT-P1 (1 x 1):" newline "7"	Pass
UT-18	print_matrix	m = 1 2 3 4 5 6 (2 × 3)	title "UT-P2"	two rows of three right-aligned width-6 numbers	" 1 2 3" newline " 4 5 6"	Pass
UT-19	print_matrix	m = -3 4 (1 x 2)	title "UT-P3"	negative printed with sign, aligned	" -3 4"	Pass
UT-20	free_matrix	none	NULL	no crash, returns	returned normally	Pass
UT-21	free_matrix	m from malloc of 4 ints	valid pointer	memory released, no crash	returned normally	Pass

Module Test Cases

■ Write in lab record

Module under test: the multiplication module, that is `read_dim` + `read_matrix` + `multiply` + `print_matrix` + `free_matrix` working together on matrices that were really allocated and really read from input. These are run through `main` but each case targets one path through the module rather than the user experience.

TC id	Feature	Precondition	Input	Expected output	Actual output	Status
MT-01	Read, multiply, print, free for rectangular matrices	program built	dims 2 3 3 2; A = 1 2 3 4 5 6; B = 7 8 9 10 11 12	Product printed as 58 64 over 139 154; exit 0	Product A x B (2 x 2): 58 64 / 139 154; exit 0	Pass
MT-02	Smallest matrices flow through every function	program built	dims 1 1 1 1; A = 7; B = 6	Product (1 × 1): 42; exit 0	Product A x B (1 x 1): 42; exit 0	Pass
MT-03	Dimension check stops the module before allocation	program built	dims 2 3 2 2	mismatch error naming 3 and 2; no element prompt; exit 1	"Error: columns of A (3) must equal rows of B (2). Multiplication not possible."; exit 1	Pass
MT-04	Zero dimension rejected before anything is read	program built	dims 0	range error after first prompt; exit 1	"Rows of A: Error: dimension must be between 1 and 100."; exit 1	Pass
MT-05	Bad element in A frees A and stops	program built	dims 1 1 1 1; A = x	element error; no prompt for B; exit 1	"Error: element (1,1) is not an integer."; exit 1	Pass
MT-06	Negative elements pass through unchanged	program built	dims 1 1 1 1; A = -3; B = 4	Product -12	Product A x B (1 x 1): -12	Pass
MT-07	Large product	program built	dims 1 1 1 1; A =	10000000000 or an overflow message	1410065408	Fail (D-01)

TC id	Feature	Precondition	Input	Expected output	Actual output	Status
			100000; B = 100000			
MT-08	Maximum dimension allocation	program built	dims 100 100 100 100, all elements 1	100 × 100 product of all 100s; exit 0	100 × 100 matrix of 100 printed; exit 0	Pass

Integration Test Cases

■ Write in lab record

Whole program as the user runs it: `./matrix_multiply` from a terminal, typed input, observed output and exit code.

TC id	Feature	Precondition	Input	Expected output	Actual output	Status
IT-01	Normal end-to-end run	binary built with no warnings	2, 3, 3, 2 then the two matrices from the Session 7 sample	A, B and the product all printed with labels and sizes; exit 0	Matrix A (2 x 3), Matrix B (3 x 2), Product A x B (2 x 2) with 58 64 / 139 154; exit 0	Pass
IT-02	Square matrices	binary built	2, 2, 2, 2; A = 1 2 3 4; B = 5 6 7 8	19 22 / 43 50	19 22 / 43 50	Pass
IT-03	Prompts appear in order and stop at the first bad value	binary built	2, 3, 2, 2	four prompts, then mismatch error, no element prompt	Prompts for all four dims then the mismatch error; exit 1	Pass
IT-04	Letters for a dimension	binary built	abc	integer error at the first prompt; exit 1	"Rows of A: Error: dimension must be an integer."; exit 1	Pass
IT-05	Dimension above the limit	binary built	101	range error; exit 1	"Error: dimension must be between 1 and 100."; exit 1	Pass
IT-06	Input ends early	binary built	2, 2, 2, 2 then only three elements	error, no crash, exit 1	"Error: element (2,2) is not an integer."; exit 1	Pass (see D-03)

TC id	Feature	Precondition	Input	Expected output	Actual output	Status
IT-07	Output alignment	binary built	1, 1, 1, 1; A = 100000; B = 100000	numbers wider than 6 still printed on their own line	"100000" and "1410065408" printed; alignment holds, value wrong	Fail (D-01)
IT-08	Build check	source file	gcc -Wall -Wextra -o matrix_multiply matrix_multiply.c	no warnings, binary produced	compiled, no output from gcc	Pass

Test Summary

■ Write in lab record

Level	Cases written	Cases run	Passed	Failed	Pass rate
Unit	21	21	19	2	90%
Module	8	8	7	1	88%
Integration	8	8	7	1	88%
Total	37	37	33	4	89%

The four failures are two defects (D-01 appears at all three levels, D-02 at unit level only). D-03 is an observation, not a failure.

Defect Log

■ Write in lab record

Defect id	Found by	Description	Severity	Root cause	Suggested fix	Status
D-01	UT-16, MT-07, IT-07	100000 x 100000 prints 1410065408 instead of 100000000000.	Medium	<code>sum</code> is a <code>long</code> but is cast to <code>int</code> when stored in <code>c</code> , and <code>c</code> is an <code>int</code> block. Any product above 2147483647 wraps.	Store the result matrix as <code>long</code> and print with <code>%ld</code> , or detect <code>sum</code> outside the <code>int</code> range and report an overflow error.	Open
D-02	UT-06	Dimension <code>2.5</code> is accepted as 2.	Low	<code>scanf("%d")</code> reads the <code>2</code> and leaves <code>.5</code> in the input buffer, which then breaks the next read.	Read the line with <code>fgets</code> and parse with <code>strtoul</code> , rejecting any trailing characters.	Open
D-03	UT-09, IT-06	When input ends early, the message says the element "is not an integer", which is misleading; the real cause is missing input.	Low (observation)	<code>scanf</code> returns EOF and the code treats every non-1 return as non- numeric.	Check for <code>EOF</code> separately and print "unexpected end of input".	Noted

No crashes, hangs, or memory errors were found in any case. The dimension check and the row-major pointer walk behaved correctly in every valid and invalid case.

Viva Questions

✗ Do not copy. Read for understanding and the viva

- **Q:** What is the difference between a unit test and a module test here? **A:** A unit test calls one function from a driver with hand-built arrays; a module test runs the read, multiply, print, and free functions together on real input.
- **Q:** Why does the unit driver rename `main`? **A:** So the driver can have its own `main` and call the program's functions directly.
- **Q:** What is a precondition? **A:** The state that must hold before the test runs, such as "binary built" or "rows=2, cols=2".
- **Q:** Why include 1×1 matrices? **A:** It is the smallest legal size; loops that run once expose off-by-one errors.
- **Q:** Why is D-01 medium and not low? **A:** It produces a wrong answer silently; the user has no way to tell the output is wrong.
- **Q:** Does a failing test mean the session failed? **A:** No. Finding and recording a real defect is the purpose of testing.
- **Q:** What does the pass rate measure? **A:** The share of test cases whose actual output matched the expected output; it says nothing about how good the cases are.
- **Q:** Why record the exit status? **A:** It is the program's contract with the shell; a script calling this program would rely on 0 for success and 1 for error.

Common Mistakes

✗ Do not copy. Read for understanding and the viva

- Writing expected output after seeing the actual output, so every case passes by construction.
- Testing only happy paths. The mismatch, zero, letters, and large-value cases are where the defects live.
- Putting the same case at all three levels and calling it three tests. Each level must target something the others cannot see.
- Leaving the actual output column blank or writing "as expected". Copy the real text.
- Ignoring compiler warnings as if they were not defects. IT-08 exists so the build itself is a test.
- Marking an overflow as pass because "the program did not crash". Wrong output is a failure.

Session Summary

■ Write in lab record

- Unit test table (21 cases) with the driver command
- Module test table (8 cases)

- Integration test table (8 cases)
- Test summary table with counts per level
- Defect log with D-01 to D-03, each with severity and suggested fix

 [Edit this page](#)

[< Previous](#)
Session 7

Session 9 [Next >](#)