

Session 12

A correct but unreliable program, the same program made reliable, and the difference between the two

This session produces two versions of a small C program that reads a list of integers, prints their mean, and looks up one value by index. The first version is correct: give it valid input and it prints the right answer every time. It is also unreliable: give it input that is slightly off and it prints garbage, crashes, or quietly returns a wrong number. The second version does the same job but checks every input before using it. Session 11 was about code that reads badly; this session is about code that fails badly.

Objectives

✗ Do not copy. Read for understanding and the viva

- State the difference between correctness (right answer on valid input) and reliability (acceptable behaviour on all input, over time, on any machine).
- Identify the specific lines where a program trusts input it has not checked.
- Show, with real runs, how the same program behaves on valid and invalid input.
- Rewrite the program so that every failure becomes a clear message and a non-zero exit status.

Problem Statement

■ Write in lab record

Session 12: Write a Program that is correct but still not reliable. Justify your answer. Make necessary assumptions.

Concept

✗ Do not copy. Read for understanding and the viva

Correctness versus reliability

Correctness is a statement about the specification: for every input the specification allows, the output matches. Reliability is a statement about operation: the probability that the program runs without failure for a given period under stated conditions, including conditions the specification never mentioned. A program can be 100 percent correct on its test set and still crash the first day a real user types a letter instead of a number. The manual's section 1.6 asks for "validation checks" and "error and exception handling" for exactly this reason.

Where unreliability hides in C

C does not check anything for you. Each of these is a place where a correct program stays silent and then fails:

- `scanf` returns how many items it converted; if you ignore it, a bad token leaves the variable unset and the program continues with whatever was in memory.
- An array of size 10 accepts index 50 without complaint; reading gives garbage, writing corrupts the stack.
- `int` wraps around on overflow; two billion plus two billion is a negative number.
- Integer division by zero is undefined behaviour: it crashes on x86, and on ARM64 (Apple Silicon) the hardware returns 0, so the program prints a wrong answer and carries on. Undefined means "any of these", which is the definition of unreliable.

What a reliable version does instead

It follows one rule: never use a value you have not checked. Check the count is a number and in range, check every value converted, use a wider type for the sum, and check the index against the count. Each failure prints what went wrong and exits with `EXIT_FAILURE`, so a calling script can detect it.

Assumptions

■ Write in lab record

1. The program reads a count N , then N integers into an array of capacity 10, prints the integer mean, then reads an index and prints the value at that index.
2. Valid input is: 1 to 10 for N , each value an `int`, and an index from 0 to N minus 1. On valid input both versions print identical output (checked with `diff`).

3. “Unreliable” means the program does not detect invalid input and its behaviour is then undefined. The runs below were done on macOS (Apple Silicon, clang as `gcc`); on another machine the garbage values and crash messages will differ, which is itself the point.
4. The reliable version treats every invalid input as an error, prints one line saying why, and exits with status 1.

Program A: Correct but Unreliable

■ Write in lab record

unreliable.c

```
1  /*
2   * unreliable.c
3   * Reads N integers, prints their mean, then prints the value at a
4   * requested index. Correct for well-formed input. Not reliable: it
5   * trusts every value the user types.
6   */
7  #include <stdio.h>
8
9  #define CAPACITY 10
10
11 int main(void)
12 {
13     int values[CAPACITY];
14     int n;
15     int i;
16     int index;
17     int sum = 0;
18
19     printf("How many values: ");
20     scanf("%d", &n);          /* return value ignored, n unchecked */
21
22     for (i = 0; i < n; i++) { /* writes past values[9] when n > 10 */
23         printf("Value %d: ", i + 1);
24         scanf("%d", &values[i]);
25     }
26
27     for (i = 0; i < n; i++) {
28         sum += values[i];      /* int overflow for large values */
29     }
30     printf("Mean: %d\n", sum / n); /* divides by zero when n is 0 */
31
32     printf("Index to look up (0 to %d): ", n - 1);
33     scanf("%d", &index);
34     printf("values[%d] = %d\n", index, values[index]); /* no range check */
35
36     return 0;
37 }
```

Compiles with `gcc -Wall unreliable.c` with no warnings. The comments mark the five places where it trusts the user.

Program B: Reliable

■ Write in lab record

reliable.c

```
1  /*
2   * reliable.c
3   * Same task as unreliable.c: read N integers, print their mean, print the
4   * value at a requested index. Every input is checked before it is used,
5   * so the program either prints a correct answer or a clear error message.
6   */
7  #include <stdio.h>
8  #include <stdlib.h>
9
10 #define CAPACITY 10
11
12 /* Reads one int from stdin into *out. Returns 1 on success, 0 on failure. */
13 static int read_int(const char *prompt, int *out)
14 {
15     printf("%s", prompt);
16     return scanf("%d", out) == 1;
17 }
18
19 int main(void)
20 {
21     int values[CAPACITY];
22     int n;
23     int i;
24     int index;
25     long long sum = 0;           /* cannot overflow for 10 int values */
26     char prompt[40];
27
28     if (!read_int("How many values: ", &n)) {
29         printf("Error: count is not a number\n");
30         return EXIT_FAILURE;
31     }
32     if (n < 1 || n > CAPACITY) {
33         printf("Error: count must be between 1 and %d\n", CAPACITY);
34         return EXIT_FAILURE;
35     }
36
37     for (i = 0; i < n; i++) {
38         snprintf(prompt, sizeof prompt, "Value %d: ", i + 1);
39         if (!read_int(prompt, &values[i])) {
40             printf("Error: value %d is not a number\n", i + 1);
41             return EXIT_FAILURE;
42         }
43     }
44 }
```

```
45     for (i = 0; i < n; i++) {
46         sum += values[i];
47     }
48     printf("Mean: %lld\n", sum / n); /* n ≥ 1 here, never divides by zero */
49
50     snprintf(prompt, sizeof prompt, "Index to look up (0 to %d): ", n - 1);
51     if (!read_int(prompt, &index)) {
52         printf("Error: index is not a number\n");
53         return EXIT_FAILURE;
54     }
55     if (index < 0 || index ≥ n) {
56         printf("Error: index %d is out of range 0 to %d\n", index, n - 1);
57         return EXIT_FAILURE;
58     }
59     printf("values[%d] = %d\n", index, values[index]);
60
61     return EXIT_SUCCESS;
62 }
```

Reliability Failures

■ Write in lab record

Each row is one condition, what `unreliable.c` actually did when run, and what `reliable.c` does instead.

Condition	Line in unreliable.c	What the unreliable version does	What the reliable version does
N larger than capacity (N = 12)	22 to 25	Writes values 11 and 12 past the end of the array, corrupting the stack; prints a mean, then aborts on return from <code>main</code> with <code>Abort trap: 6</code>	Prints <code>Error: count must be between 1 and 10</code> and exits with status 1 before reading any value
Index out of range (index = 50 with N = 3)	33 to 34	Prints <code>values[50] = 1809555360</code> ; a second run printed <code>1868292000</code> ; the number is whatever is on the stack	Prints <code>Error: index 50 is out of range 0 to 2</code> and exits with status 1
N = 0	30	Integer division by zero; on this ARM64 machine it printed <code>Mean: 0</code> and continued; on x86 it would crash with <code>Floating point exception</code>	Rejected by the same range check as above
Sum overflow (values 2000000000 and 2000000000)	28 to 30	Prints <code>Mean: -147483648</code> , a wrong answer with no warning	Sum is <code>long long</code> ; prints <code>Mean: 2000000000</code>
Non-numeric count (<code>abc</code>)	20	<code>scanf</code> converts nothing; <code>n</code> holds a leftover stack value; printed <code>Index to look up (0 to -316699233)</code> and continued	Prints <code>Error: count is not a number</code> and exits with status 1
Non-numeric value in the list	24	Same as above for that element; the mean is computed on garbage	Prints <code>Error: value k is not a number</code> and exits

Correctness and Reliability

■ Write in lab record

Aspect	Correctness	Reliability
Question it answers	Does the output match the specification for valid input?	Does the program keep behaving acceptably when conditions are not ideal?
How it is measured	Test cases with expected outputs (Session 8 style)	Failure rate over time or over inputs; mean time between failures
Where it fails	Wrong algorithm, wrong formula, off-by-one	Unchecked input, overflow, resource exhaustion, environment differences
Program A	Correct: all valid-input tests pass	Unreliable: five ways to make it crash or lie
Program B	Correct: same output as A on valid input	Reliable within its stated limits: every bad input becomes a message and a status code

Program A is correct because for valid input it is indistinguishable from Program B. It is unreliable because the word “valid” is doing all the work, and nothing in the program enforces it.

Sample Runs

■ Write in lab record

Run 1: valid input, both versions agree

```

1 $ gcc -Wall -o unreliable unreliable.c
2 $ gcc -Wall -o reliable reliable.c
3 $ printf '4\n10\n20\n30\n40\n2\n' | ./unreliable > a.out
4 $ printf '4\n10\n20\n30\n40\n2\n' | ./reliable > b.out
5 $ diff a.out b.out && echo IDENTICAL
6 IDENTICAL

```

At the terminal, either program shows:

```
1 How many values: 4
2 Value 1: 10
3 Value 2: 20
4 Value 3: 30
5 Value 4: 40
6 Mean: 25
7 Index to look up (0 to 3): 2
8 values[2] = 30
```

Run 2: invalid input, N larger than capacity and index out of range

Unreliable version, N = 12 into an array of 10, then index 50:

```
1 $ ./unreliable
2 How many values: 12
3 Value 1: 1
4 Value 2: 2
5 ...
6 Value 10: 10
7 Value 11: 11
8 Value 12: 12
9 Mean: 6
10 Index to look up (0 to 11): 50
11 values[50] = 1829724128
12 Abort trap: 6
13 $ echo $?
14 134
```

What happened: values 11 and 12 were written past the end of `values[]` onto the stack. The program did not notice; it printed a mean and a garbage lookup, and only when `main` returned did the stack-protector detect the corruption and abort the process (signal 6, exit status 134). Nothing in the program's own output says anything went wrong. When the output is redirected to a file instead of a terminal, even the prompts are lost, because the process was killed before flushing its buffer.

Reliable version, same input:

```

1 $ ./reliable
2 How many values: 12
3 Error: count must be between 1 and 10
4 $ echo $?
5 1

```

Same index error with a valid count:

```

1 $ printf '3\n5\n6\n7\n50\n' | ./unreliable
2 How many values: Value 1: Value 2: Value 3: Mean: 6
3 Index to look up (0 to 2): values[50] = 1809555360
4
5 $ printf '3\n5\n6\n7\n50\n' | ./reliable
6 How many values: Value 1: Value 2: Value 3: Mean: 6
7 Index to look up (0 to 2): Error: index 50 is out of range 0 to 2

```

Viva Questions

✗ Do not copy. Read for understanding and the viva

- **Q:** Give a one-line definition of reliability. **A:** The probability that software operates without failure for a stated time under stated conditions.
- **Q:** Program A passes every test in Session 8 style. Is it reliable? **A:** No; tests cover valid input, reliability is about what happens outside that set.
- **Q:** Why did the N = 12 run print a mean before crashing? **A:** The out-of-bounds writes corrupted the stack silently; the abort came only when `main` returned and the stack canary was checked.
- **Q:** Why did the N = 0 run not crash on this machine? **A:** Integer division by zero is undefined behaviour; ARM64 hardware returns 0 instead of trapping. Different machine, different failure.
- **Q:** What does `scanf` return, and why does it matter? **A:** The number of items converted; if you do not check it, a non-numeric token leaves the variable unset.
- **Q:** Why `long long` for the sum? **A:** Ten `int` values can exceed the `int` range; `long long` holds at least 63 bits, so the sum cannot wrap.
- **Q:** Is Program B reliable in every situation? **A:** No; it is reliable within its stated limits (10 values, `int` input). Reliability is always relative to stated conditions.
- **Q:** Which is cheaper, adding the checks now or later? **A:** Now; every check in Program B is two or three lines, and the crash in Program A would be a bug report with no useful message.

Common Mistakes

✗ Do not copy. Read for understanding and the viva

- Submitting a program that is simply wrong and calling it unreliable. It must give correct output on valid input first.
- Describing the failure in theory only. Run the invalid case and paste what actually appeared, including the exit status.
- Claiming “it crashes” when it did not. On this machine $N = 0$ printed a wrong mean; say what you observed and why it may differ elsewhere.
- Fixing reliability by adding one check and stopping. Each of the five failure rows is a separate defect and needs its own check.
- Making the reliable version change its output on valid input. Check with `diff`.

Session Summary

■ Write in lab record

- Assumptions, including the machine the runs were done on.
- Program A (`unreliable.c`) and Program B (`reliable.c`).
- The reliability failures table with line references and observed behaviour.
- The correctness versus reliability comparison.
- Run 1 (valid, identical) and Run 2 (invalid, unreliable misbehaves, reliable reports) with real output.

[✎ Edit this page](#)

[< Previous](#)
Session 11

Session 13 [Next >](#)