

Session 11

A correct program of poor quality, and the same program rewritten to a coding standard

This session produces two versions of one small C program. The first computes the right answer but is written badly on purpose. The second computes the same answer and follows a coding standard. The point is that “it works” is the lowest bar a program can clear: a reviewer, a tester, or the person who maintains the code next year needs far more than correct output. You will justify, attribute by attribute, why the first version is poor quality even though every test passes.

Objectives

✗ Do not copy. Read for understanding and the viva

- Separate the idea of correctness (right output) from quality (readable, maintainable, standard-conforming code).
- Write a program that passes its tests but fails a code review, and name the exact lines that fail.
- Rewrite the same program to a stated coding standard without changing its behaviour.
- Defend the quality attributes (readability, maintainability, modularity, naming, comments, robustness) in a viva.

Problem Statement

■ Write in lab record

Session 11: Write a Program that is correct but of not good quality. Justify your answer. Make necessary assumptions.

Concept

✗ Do not copy. Read for understanding and the viva

Correctness is not quality

A program is correct when it produces the specified output for every valid input. Quality is everything else a reader cares about: can someone understand it in five minutes, change one rule without breaking another, test one piece in isolation, and trust it with bad input. The manual (section 1.6) lists comments, validation checks, error handling, and coding standards as the things an implementation “must contain”. None of them changes the output for a valid run, which is exactly why students skip them.

What a coding standard covers

The manual gives one sample clause: “A consistent naming pattern is one of the most important elements of predictability and discoverability”. A usable standard for a C lab program covers at least:

- Naming: constants in UPPER_CASE, functions and variables as descriptive lower_snake_case, no single letters except loop counters.
- Layout: one statement per line, fixed indentation (four spaces), braces on every block.
- Structure: one job per function, `main()` only coordinates, no duplicated logic.
- Constants: no magic numbers; every threshold is a named constant defined once.
- Comments: a header comment per file and per function saying what it does and returns.
- Robustness: check what you read before you use it.

The quality attributes you will be asked about

Attribute	Question the reviewer asks
Readability	Can a new reader follow the flow without running it?
Maintainability	If the grade boundary for B changes from 75 to 70, how many lines change?
Modularity	Is each task in its own function with a clear input and output?
Naming	Does every identifier say what it holds or does?
Comments	Is the intent written down where the code is not obvious?
Robustness	What happens on $N = 0$, $N = 200$, or a letter instead of a number?
Adherence to standard	Does it follow the named rules above, consistently?

Assumptions

■ Write in lab record

1. The program grades a class: it reads N (at most 100) and then N integer marks out of 100, and prints the average, the highest marks, a letter grade per student, and the grade of the class average.
2. Grade rule: A for 90 and above, B for 75 to 89, C for 60 to 74, D for 40 to 59, F below 40. The same rule applies to the class average.
3. "Correct" means: for every valid input (1 to 100 students, marks 0 to 100) both versions print exactly the same output. This was checked by running both with the same input and comparing with `diff`.
4. The coding standard is the one listed under Concept. The poor version violates it on purpose; it was not written carelessly by accident.

Program A: Correct but Poor Quality

■ Write in lab record

poor_quality.c

```
1 #include <stdio.h>
2 int t,u;
3 int main(){int n,i,a[100],s=0,h=0;float v;
4 printf("Enter number of students: ");scanf("%d",&n);
5     for(i=0;i<n;i++){printf("Marks of student %d: ",i+1);scanf("%d",&a[i]);}
6     for(i=0;i<n;i++)s=s+a[i];
7     v=(float)s/n;
8     for(i=0;i<n;i++)if(a[i]>h)h=a[i];
9     printf("Average marks: %.2f\n",v);printf("Highest marks: %d\n",h);
10    for(i=0;i<n;i++){
11        if(a[i] ≥ 90)printf("Student %d: %d → A\n",i+1,a[i]);
12        else if(a[i] ≥ 75)printf("Student %d: %d → B\n",i+1,a[i]);
13        else if(a[i] ≥ 60)printf("Student %d: %d → C\n",i+1,a[i]);
14        else if(a[i] ≥ 40)printf("Student %d: %d → D\n",i+1,a[i]);
15        else printf("Student %d: %d → F\n",i+1,a[i]);}
16    if(v ≥ 90)printf("Class average grade: A\n");
17    else if(v ≥ 75)printf("Class average grade: B\n");
18    else if(v ≥ 60)printf("Class average grade: C\n");
19    else if(v ≥ 40)printf("Class average grade: D\n");
20    else printf("Class average grade: F\n");
21    return 0;}
```

It compiles with `gcc -Wall poor_quality.c` with no warnings, which is the first lesson: the compiler measures syntax, not quality. The unused globals `t` and `u` on line 2 are not even reported.

Program B: Same Behaviour, Coding Standard Applied

■ Write in lab record

good_quality.c

```
1  /*
2   * good_quality.c
3   * Reads the marks of N students, prints the class average, the highest
4   * marks, the grade of every student and the grade of the class average.
5   *
6   * Coding standard followed:
7   *   - constants in UPPER_CASE, functions and variables in lower_snake_case
8   *   - one task per function, main() only coordinates
9   *   - four-space indentation, braces on every block
10  *   - every function has a comment saying what it does and returns
11  */
12  #include <stdio.h>
13
14  #define MAX_STUDENTS 100
15
16  /* Grade boundaries (marks out of 100). Change here, nowhere else. */
17  #define GRADE_A_MIN 90
18  #define GRADE_B_MIN 75
19  #define GRADE_C_MIN 60
20  #define GRADE_D_MIN 40
21
22  /* Reads marks for `count` students into `marks`. Returns nothing. */
23  static void read_marks(int marks[], int count)
24  {
25      int i;
26
27      for (i = 0; i < count; i++) {
28          printf("Marks of student %d: ", i + 1);
29          scanf("%d", &marks[i]);
30      }
31  }
32
33  /* Returns the arithmetic mean of `count` marks. Caller ensures count > 0. */
34  static float average_marks(const int marks[], int count)
35  {
36      int i;
37      int total = 0;
38
39      for (i = 0; i < count; i++) {
40          total += marks[i];
41      }
42      return (float) total / count;
43  }
```

```
44
45 /* Returns the largest value among `count` marks. */
46 static int highest_marks(const int marks[], int count)
47 {
48     int i;
49     int highest = marks[0];
50
51     for (i = 1; i < count; i++) {
52         if (marks[i] > highest) {
53             highest = marks[i];
54         }
55     }
56     return highest;
57 }
58
59 /* Maps a mark (or an average) to a letter grade. One place for the rule. */
60 static char grade_of(float marks)
61 {
62     if (marks ≥ GRADE_A_MIN) {
63         return 'A';
64     }
65     if (marks ≥ GRADE_B_MIN) {
66         return 'B';
67     }
68     if (marks ≥ GRADE_C_MIN) {
69         return 'C';
70     }
71     if (marks ≥ GRADE_D_MIN) {
72         return 'D';
73     }
74     return 'F';
75 }
76
77 int main(void)
78 {
79     int marks[MAX_STUDENTS];
80     int count;
81     int i;
82     float average;
83
84     printf("Enter number of students: ");
85     if (scanf("%d", &count) ≠ 1 || count < 1 || count > MAX_STUDENTS) {
86         printf("Number of students must be between 1 and %d\n", MAX_STUDENTS);
87         return 1;
```

```
88     }
89
90     read_marks(marks, count);
91     average = average_marks(marks, count);
92
93     printf("Average marks: %.2f\n", average);
94     printf("Highest marks: %d\n", highest_marks(marks, count));
95
96     for (i = 0; i < count; i++) {
97         printf("Student %d: %d → %c\n", i + 1, marks[i], grade_of(marks[i]))
98     }
99     printf("Class average grade: %c\n", grade_of(average));
100
101     0
102
103     1     return 0;
104
105     2 }
```

Justification

■ Write in lab record

Every row cites a line of `poor_quality.c` as evidence, then says how `good_quality.c` fixes it. This table is the answer to "justify".

Attribute	Evidence in poor_quality.c	Why it is a defect	Fix in good_quality.c
Readability	Line 3 declares seven variables and opens <code>main</code> on one line; lines 5 to 8 mix tabs, two spaces and no indentation	The eye cannot find where a loop starts or ends; the reader must count braces	Four-space indentation, one declaration per line, braces on every block
Maintainability	The thresholds 90, 75, 60, 40 appear twice (lines 11 to 14 and 16 to 19)	Changing the B boundary means editing two places; missing one produces a silent inconsistency	<code>GRADE_A_MIN</code> to <code>GRADE_D_MIN</code> defined once; <code>grade_of()</code> is the single place the rule lives
Modularity	Everything is inside <code>main</code> (lines 3 to 21); reading, summing, maximum and grading are interleaved	Nothing can be tested or reused on its own	<code>read_marks()</code> , <code>average_marks()</code> , <code>highest_marks()</code> , <code>grade_of()</code> ; <code>main</code> only calls them
Naming	<code>n</code> , <code>a</code> , <code>s</code> , <code>h</code> , <code>v</code> , <code>t</code> , <code>u</code> (lines 2 and 3)	The names carry no meaning; <code>h</code> could be “hours” or “highest”; <code>t</code> and <code>u</code> are never used at all	<code>count</code> , <code>marks</code> , <code>total</code> , <code>highest</code> , <code>average</code> ; unused variables removed
Comments	None in 21 lines	The grade rule and the meaning of <code>v</code> exist only in the author’s head	File header states purpose and the standard; each function has a one-line contract
Duplicated code	The five-way if chain is written out twice (lines 11 to 19)	Two copies of one rule always drift apart eventually	One function <code>grade_of()</code> called for each student and for the average

Attribute	Evidence in poor_quality.c	Why it is a defect	Fix in good_quality.c
Magic numbers	<code>a[100]</code> on line 3, and the thresholds on lines 11 to 19	The reader does not know if 100 is a limit, a percentage, or a coincidence	<code>MAX_STUDENTS</code> and the <code>GRADE_*_MIN</code> constants have names
Robustness	Line 4 ignores the return value of <code>scanf</code> and never checks <code>n</code> ; line 7 divides by <code>n</code>	<code>N = 0</code> divides by zero; <code>N = 200</code> writes past the array; a letter leaves <code>n</code> uninitialised	<code>main</code> checks the <code>scanf</code> result and the 1 to <code>MAX_STUDENTS</code> range before reading marks
Adherence to standard	No rule from the standard is followed consistently	A team cannot review or merge code that has no shared shape	Header comment names the standard and the body follows it throughout

What did not change

The output. Both programs print the same prompts, the same numbers and the same grades for every valid input. Quality was improved without touching behaviour, which is what refactoring means.

Sample Output

■ Write in lab record

Compile and run both with the same input and compare:

```

1 $ gcc -Wall -o poor poor_quality.c
2 $ gcc -Wall -o good good_quality.c
3 $ printf '5\n85\n92\n38\n67\n74\n' | ./poor > poor.out
4 $ printf '5\n85\n92\n38\n67\n74\n' | ./good > good.out
5 $ diff poor.out good.out && echo IDENTICAL
6 IDENTICAL

```

Interactive run of either program with five students:

```
1 Enter number of students: 5
2 Marks of student 1: 85
3 Marks of student 2: 92
4 Marks of student 3: 38
5 Marks of student 4: 67
6 Marks of student 5: 74
7 Average marks: 71.20
8 Highest marks: 92
9 Student 1: 85 → B
10 Student 2: 92 → A
11 Student 3: 38 → F
12 Student 4: 67 → C
13 Student 5: 74 → C
14 Class average grade: C
```

The good version differs only on invalid input, which the poor version does not handle:

```
1 Enter number of students: 0
2 Number of students must be between 1 and 100
```

Viva Questions

× Do not copy. Read for understanding and the viva

- **Q:** The poor program gives the correct answer. Why is it still a problem? **A:** Correctness is checked once; the code is read, changed and tested many times, and every one of those costs more when the code is unreadable.
- **Q:** Name one quality defect the compiler cannot detect. **A:** Duplicated logic (the grade rule written twice); `-Wall` is silent about it.
- **Q:** What is a magic number? **A:** A literal such as 75 whose meaning is not stated; the fix is a named constant defined in one place.
- **Q:** Why is putting everything in `main` bad if the program is only 20 lines? **A:** Nothing can be unit tested or reused, and the 20 lines become 200 the moment a feature is added.
- **Q:** How does the good version prove it has the same behaviour? **A:** Both binaries were run on the same input and their outputs compared with `diff`.

- **Q:** Which attribute did `grade_of()` improve most? **A:** Maintainability: the grade rule exists once, so a boundary change is a one-line edit.
- **Q:** Is `good_quality.c` fully robust? **A:** No; it validates the count but not each mark, and `scanf` on a non-number still leaves the value unset. Robustness is a scale, not a switch.
- **Q:** What part of the manual asks for this? **A:** Section 1.6 Implementation: comments, validation checks, error handling and coding standards.

Common Mistakes

✗ Do not copy. Read for understanding and the viva

- Writing a program that is wrong and calling it “poor quality”. The problem asks for a correct program; check the output before you argue about style.
- Listing defects without line numbers. “No comments” is an opinion; “lines 1 to 21 contain no comment” is evidence.
- Making the poor version so bad it does not compile, or so mild that a reviewer cannot see the difference. Aim for clearly wrong style, clearly right output.
- Changing behaviour in the rewrite (different prompts, different rounding). Then it is a new program, not a quality improvement.
- Forgetting the assumptions. The examiner needs to know what “correct” was measured against.

Session Summary

■ Write in lab record

- Assumptions: the task, the grade rule, and the coding standard used.
- Program A (`poor_quality.c`) with its output.
- Program B (`good_quality.c`) with the same output.
- The justification table with line references for every attribute.
- The `diff` evidence that both programs behave identically on valid input.

 Edit this page

< Previous
Session 10

Next >
Session 12