**Part A — Theory Component**

**1. Coding Best Practices**

**Task:**  
Write a short essay (1–2 pages) on the **coding best practices** for the C Language:

**C is a powerful, general-purpose, structured programming language developed by Dennis Ritchie at Bell Laboratories in the early 1970s.**

Writing clean, efficient, and maintainable code is essential for every C programmer. Following coding best practices not only improves readability but also reduces bugs, enhances performance, and makes collaboration easier. Below are some key practices to follow when developing programs in C.

### ****1. Code Formatting and Naming Conventions****

* **Indentation and spacing:** Proper indentation improves readability. Use consistent spacing and indentation (usually 4 spaces per level).
* **Naming conventions:** Use meaningful names for variables and functions. For example, use calculateSum() instead of cs() or s(). Constants and macros are usually written in **UPPERCASE**.
* **Braces and layout:** Always use braces {} for conditional and loop blocks, even for single statements, to avoid logical errors.
* **Comments:** Use // for short explanations and /\* ... \*/ for multi-line descriptions. Every function should include a brief comment describing its purpose and parameters.

Example: Variables: studentCount, totalMarks

Functions: calculateAverage(), findMaximum()

Constants: #define MAX\_SIZE 100

### ****2. Error Handling Techniques****

Error handling in C is often done through return values and conditional checks since exceptions are not supported natively.

C does not have built-in exception handling, so error handling must be done manually,

Check the return values of library functions such as scanf(), malloc(), and fopen().

* Always **validate user input** and handle edge cases.
* Use **return codes** to indicate success or failure (e.g., return 0 for success, return -1 for error).
* Check the return values of standard library functions such as malloc(), fopen(), or scanf() to ensure they succeed before proceeding.
* Implement error messages using fprintf(stderr) for better debugging.
* Use proper error messages for failed operations using:

→ fprintf(stderr, "Error: File not found.\n");

### ****3. Memory Management****

Since C does not provide automatic garbage collection, memory management is crucial.

* Use malloc() or calloc() for dynamic memory allocation and always **free** the memory using free() when done.
* Avoid **memory leaks** by ensuring every allocated block is freed.
* Avoid using uninitialized pointers and ensure pointer arithmetic is done safely.

Example: int \*arr = malloc(10 \* sizeof(int));

if (arr != NULL) {

// use arr

free(arr);

}

### ****4. Modularity and File Organization****

Good programs are modular and easy to extend.

* Divide large programs into smaller **functions** that each perform a single task.
* Group related functions into separate source files (.c) and use **header files** (.h) for function declarations and constants.
* This promotes code reusability and makes debugging easier.
* Example project structure:

main.c

operations.c

operations.h

This improves reusability, readability, and debugging.

### ****5. Readability and Maintainability****

Readable code is easier to maintain and modify.

* Write self-documenting code with clear naming and minimal complexity.
* Avoid deep nesting of loops and conditions.
* Keep functions short and focused on a single responsibility.
* Use consistent style throughout the project to help teams collaborate effectively.
* Include sufficient comments and documentation.

**2. Advantages of Code Reviews**

**Task:**  
Write a one-page summary explaining:

* What a **code review** is.
* Why it is an essential step in software development.
* The **benefits** for developers and teams (e.g., knowledge sharing, early bug detection, consistent coding standards).
* Common **best practices** during code reviews (like focusing on readability, correctness, and performance).

**Optional:**  
Include an example of a short piece of code (in any language) with **poor style**, then show how a reviewer’s feedback could improve it.

### ****1. What a Code Review Is****

A **code review** is the process where one or more developers systematically examine another developer’s source code before it is merged into the main project.  
The main goal is to ensure that the code is **accurate, readable, efficient, and consistent** with project standards.  
Code reviews are not limited to finding bugs — they also focus on improving design, maintainability, and understanding of the code across the team.

Example:  
Before new code is added to a shared repository, team members review it line by line to check logic, formatting, and adherence to guidelines.

### ****2. Why It Is an Essential Step in Software Development****

Code reviews are a **critical stage** in modern software development because they help maintain **high-quality standards** and **prevent errors early**.  
They act as a quality checkpoint, ensuring that every line of code entering the main project is thoroughly verified.

Key reasons why code reviews are essential:

* **Early Error Detection:** Bugs and logic issues are found before deployment.
* **Improved Code Consistency:** Ensures the same style and structure across all modules.
* **Better Design Decisions:** Reviewers can suggest alternative and more efficient solutions.
* **Increased Reliability:** Fewer runtime errors and better performance in final applications.

### ****3. Benefits for Developers and Teams****

Code reviews offer several benefits that go beyond just improving code quality. They promote teamwork, skill enhancement, and long-term productivity.

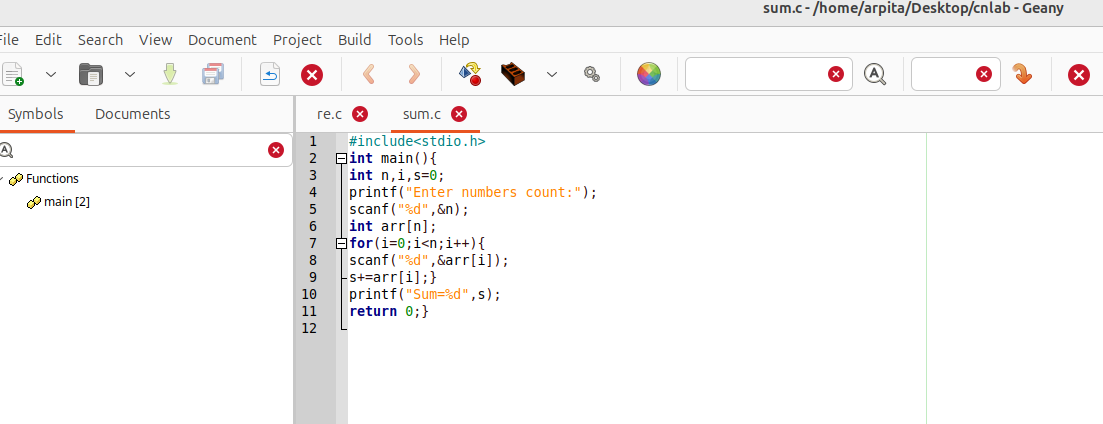
* **Knowledge Sharing:** Developers learn new techniques and coding styles from peers.
* **Skill Development:** Junior programmers gain insights from experienced reviewers.
* **Team Collaboration:** Encourages communication and understanding among team members.
* **Reduced Debugging Time:** Detecting issues early saves hours of future debugging.
* **Consistent Coding Standards:** Maintains uniform structure and readability across the project.
* **Better Maintainability:** Clean and well-documented code is easier to update or enhance later.

### ****4. Common Best Practices During Code Reviews****

To make the review process productive and respectful, developers should follow a set of best practices:

* **Focus on Readability:** Ensure the code is easy to understand, with proper indentation, naming, and comments.
* **Check for Correctness:** Validate that the logic, syntax, and data handling are accurate.
* **Consider Performance:** Look for unnecessary computations, nested loops, or inefficient algorithms.
* **Give Constructive Feedback:** Be polite, specific, and explain why a change is suggested.
* **Review Small Portions:** Smaller and frequent reviews are more effective than large ones.
* **Encourage Positive Discussion:** Treat reviews as learning opportunities rather than criticism.

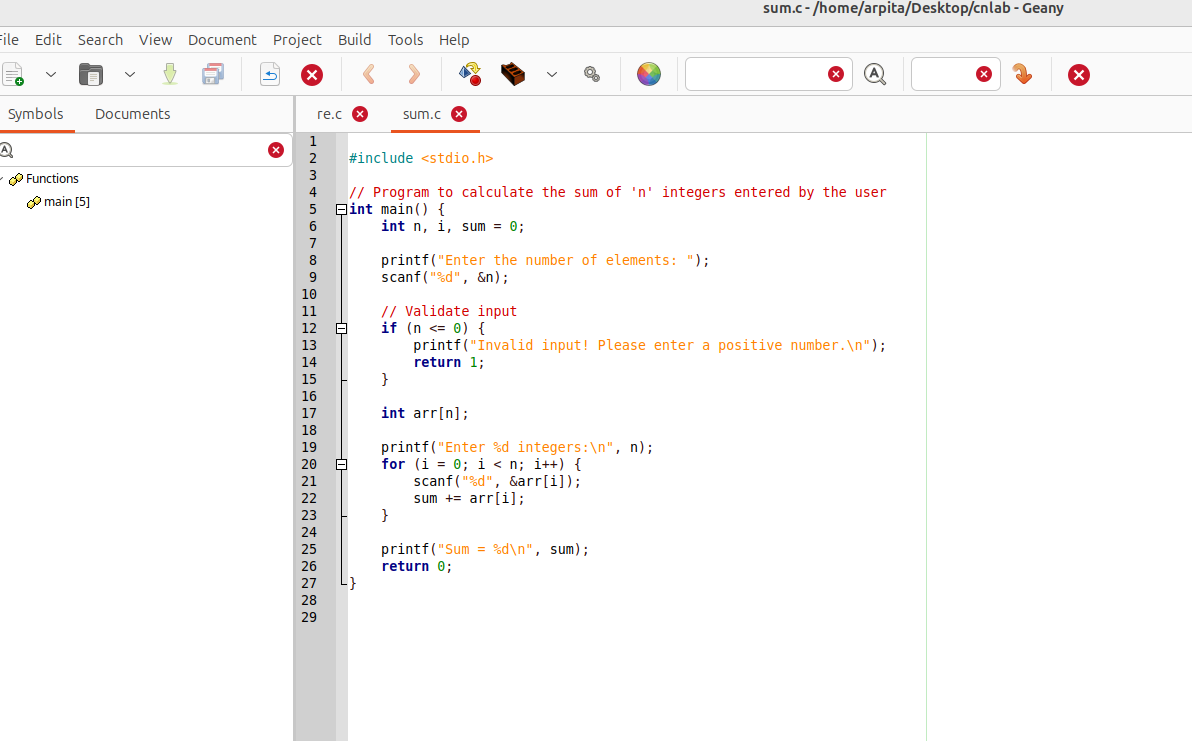
Before Code Review — Poor Coding Style:



**Reviewer’s Feedback:**

🔹 **Poor Formatting:** The entire code is crammed together without indentation or spacing.  
🔹 **Meaningless Variable Names:** s is not descriptive — should be sum.  
🔹 **Lack of Comments:** No explanation of what the program does.  
🔹 **No Input Validation:** The program doesn’t check whether n is a valid positive number.  
🔹 **Readability Issues:** Hard to read and maintain in the long run.

**After Code Review — Improved Version**



**3. Advantages of Unit Testing**

**Task:**  
Explain in your own words:

* What **unit testing** means and why it is important.
* The **advantages** of unit testing (like preventing regressions, improving design, saving debugging time).
* The tools used for unit testing in assigned language:
* C → Unity or Cunit

1. **What** **unit testing**

In software development, **unit testing** is a process where individual modules or functions of a program are tested independently to ensure that each part works as expected. A unit refers to the smallest testable part of an application — in the case of C, this is usually a function or a small group of related functions.

Unit testing is performed during the **development phase** to detect and fix bugs early in the coding cycle before they grow into larger issues during system testing or integration.

## ****2. Importance of Unit Testing****

Unit testing is an essential step in **software quality assurance**.  
It helps verify that every function in the C program behaves correctly under various conditions. By testing functions separately, developers can identify logic errors, handle edge cases, and ensure robustness.

In C, where memory management and pointer handling are manual, unit testing plays a crucial role in **preventing memory leaks, segmentation faults, and undefined behaviors.**

## ****3. Advantages of Unit Testing****

### ****a) Early Detection of Bugs****

Unit tests are written and executed during development, allowing developers to catch bugs at an early stage.  
Fixing issues during development is much cheaper and faster than debugging them after integration.

### ****b) Prevents Regressions****

Whenever a developer modifies existing code, there is a risk of breaking previously working features.  
With unit tests in place, rerunning the test suite immediately identifies whether new changes have unintentionally introduced bugs.

### ****c) Improves Code Quality and Design****

Writing testable code encourages developers to write modular, reusable, and well-structured functions.  
This leads to better software architecture because smaller and independent units are easier to test and maintain.

### ****d) Saves Debugging and Maintenance Time****

When a unit test fails, developers instantly know which specific function is causing the issue.  
This saves hours of manual debugging and simplifies long-term maintenance.

### ****e) Increases Developer Confidence****

Having a complete set of automated tests gives developers confidence to make changes or add new features, knowing that the existing functionality remains intact.

### ****f) Simplifies Integration Testing****

Since every unit has already been verified individually, integration testing becomes much smoother — any remaining issues are likely due to module interactions, not internal logic errors.

## ****4. Unit Testing Tools for C****

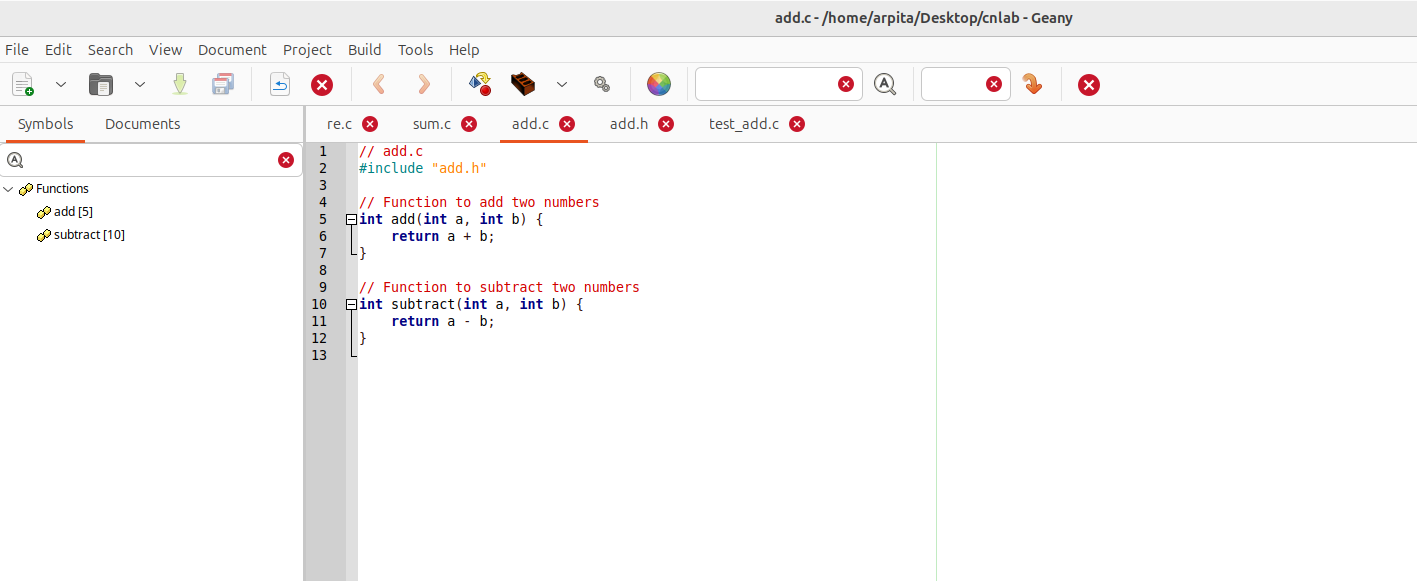
C does not have built-in testing frameworks like modern languages such as Python or Java. However, several open-source frameworks make it easy to perform unit testing in C.

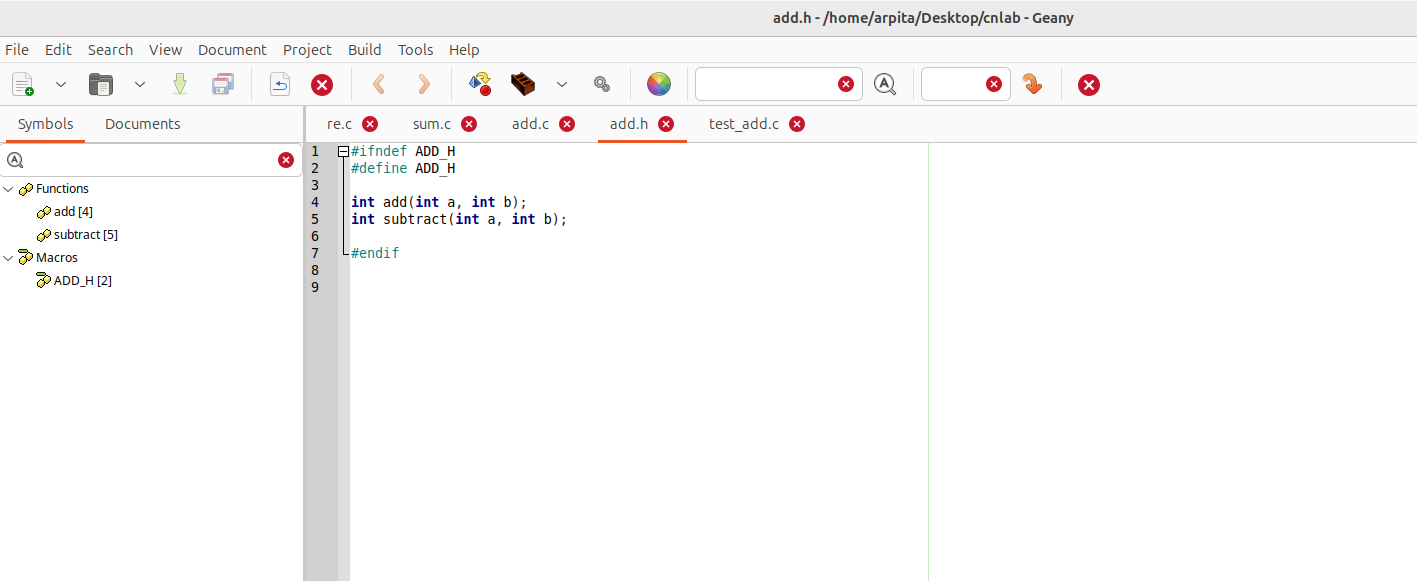
### ****b) CUnit****

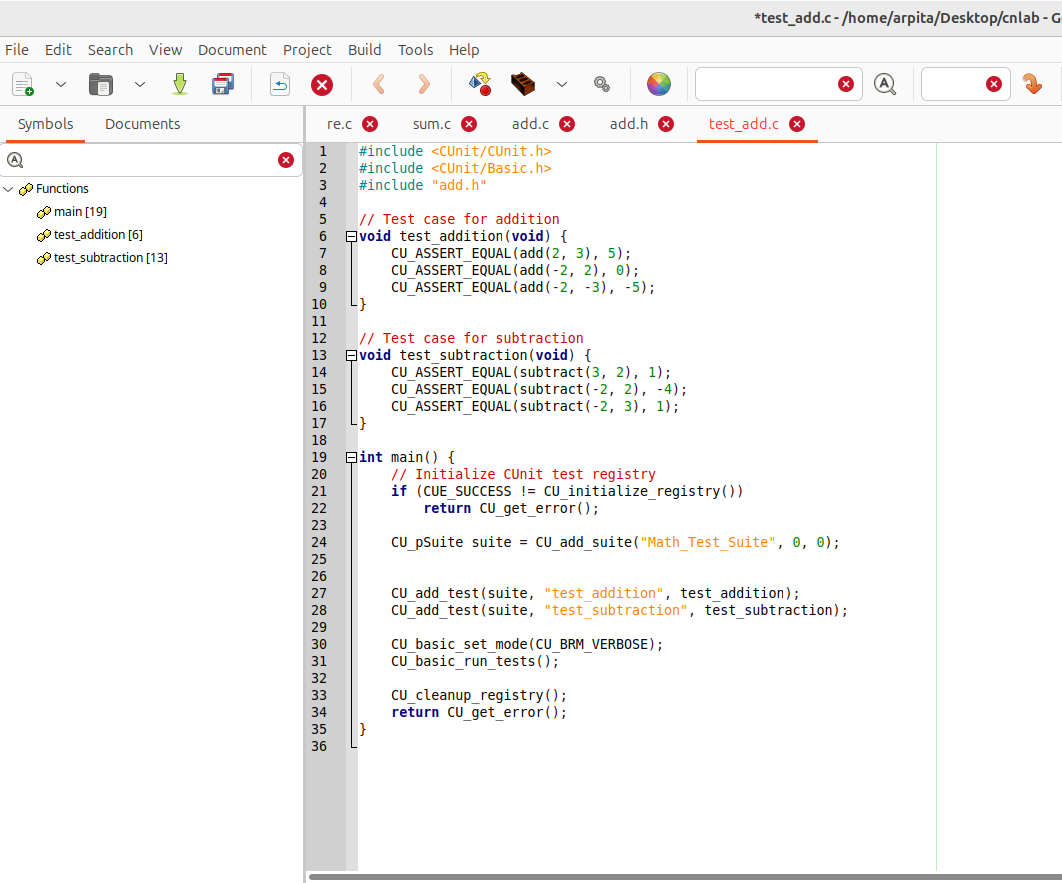
**CUnit** is popular framework for unit testing in C.  
It provides an automated way to execute tests, organize them into suites, and generate test reports.

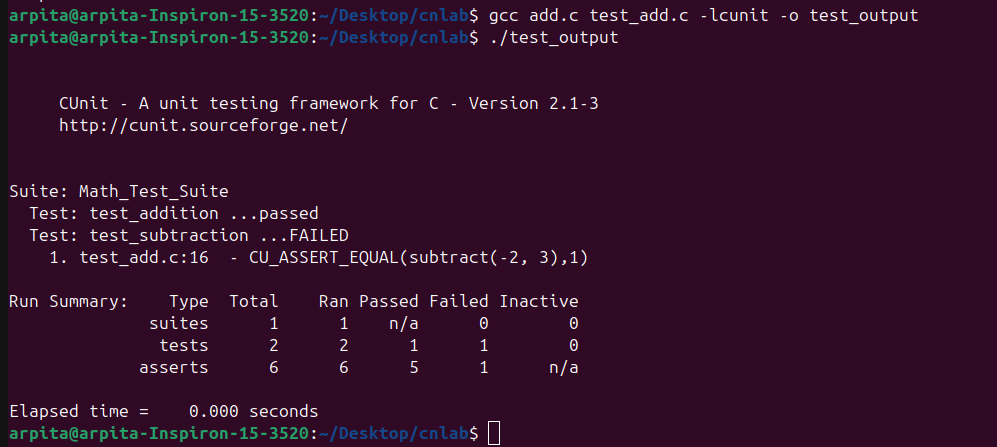
**Features:**

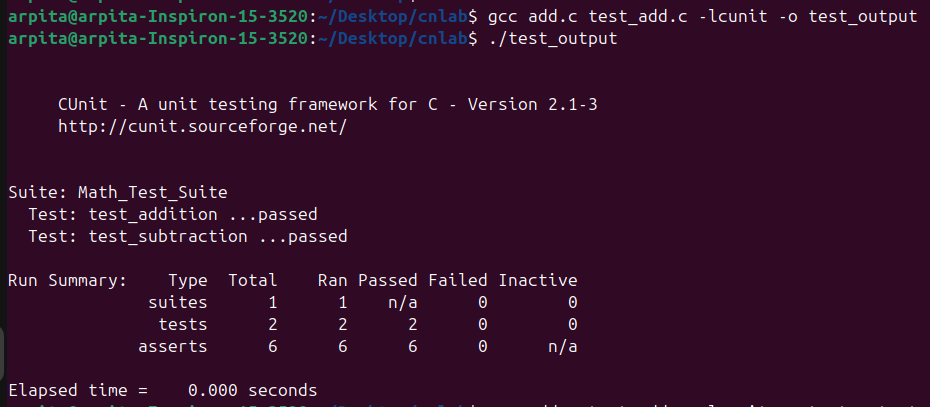
* Supports both console and GUI interfaces
* Easy integration with CI/CD tools
* Allows grouping of tests into logical suites
* Provides assertions like CU\_ASSERT\_EQUAL, CU\_ASSERT\_TRUE, etc.











**Part B — Practical Component**

**Task 1 — Implement a simple Program in assigned language**

**Choose a simple problem statement, for example:**

“Write a program that reads a list of integers and prints the sum, average, and maximum value.”

**Requirements:**

* Implement the same logic in **C**
* Follow **coding best practices** for C language:
  + Proper indentation and naming conventions
  + Modular structure (functions)
  + Error handling
  + Comments and readability

#include<stdio.h>

// Function to calculate sum

int calculateSum(int arr[],int n)

{

int sum=0;

for(int i=0;i<n;i++)

{

sum+=arr[i];

}

return sum;

}

// Function to calculate average

float calculateAverage(int sum,int n)

{

return (float)sum/n;

}

// Function to find maximum

int findmaximum(int arr[],int n)

{

int max=arr[0];

for(int i=1;i<n;i++)

{

if(arr[i]>max)

{

max=arr[i];

}

}

return max;

}

int main()

{

int n,i;

int arr[100];

printf("Enter how many numbers: ");

scanf("%d",&n);

if(n<=0 || n>100)

{

printf("Invalid no of elements!\n");

return 1;

}

printf("Enter %d integers:\n",n);

for(i=0;i<n;i++)

{

scanf("%d",&arr[i]);

}

int sum=calculateSum(arr,n);

float avg=calculateAverage(sum,n);

int max=findmaximum(arr,n);

printf("\n-----Results------\n");

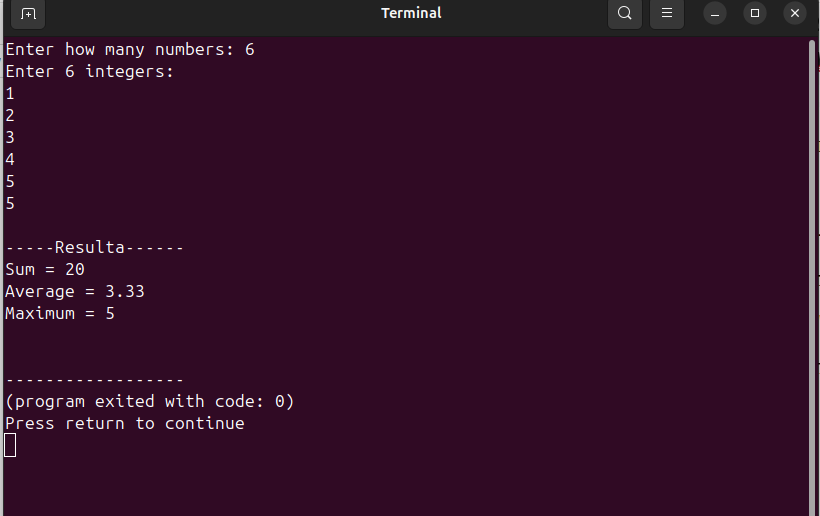
printf("Sum = %d\n",sum);

printf("Average = %.2f\n",avg);

printf("Maximum = %d\n",max);

return 0;

}



**Task 2 — Perform a Code Review**

**Exchange your code with a classmate.**

* Review their code using these criteria:
  + Code readability
  + Use of best practices
  + Correctness and error handling
  + Modularity and clarity

Code Review:

**1. Code Readability:**  
The code is neatly formatted with consistent indentation and meaningful variable names, making it easy to understand. Comments are used appropriately, but adding brief explanations inside functions would improve clarity further. Overall, the structure supports readability.

**2. Use of Best Practices:**  
The program follows good practices such as modular design and descriptive naming. However, maintaining consistent naming conventions (e.g., findMaximum instead of findmaximum) and validating user input more thoroughly would make it even better.

**3. Correctness and Error Handling:**  
The program produces correct results for valid inputs and includes basic error handling for invalid array sizes. Still, it could be enhanced by checking for invalid or non-numeric inputs to ensure robustness in real-world usage.

**4. Modularity and Clarity:**  
The code is well-divided into separate functions, each handling a specific task, which improves maintainability and clarity. This modular structure makes it easy to add new functionalities like finding the minimum or sorting in the future.

**Strengths:**

The program is well-structured and easy to follow. The use of clear variable names like sum, avg, and max improves readability. The code is properly indented, making it visually neat and consistent. The logic for calculating the sum, average, and maximum is correctly implemented, and the output is clearly displayed to the user.

#### ****Suggestions for Improvement:****

1. **Input Validation:**  
   The program could include stronger input validation (e.g., checking if the entered values are integers or if the number of inputs is positive). This would make the code more robust against invalid user inputs.
2. **Error Handling:**  
   In case the user enters invalid data, the program should display a clear error message and exit gracefully instead of continuing with incorrect values.
3. **Modularity Enhancement:**  
   Some logic (like reading input or displaying results) can be moved into separate functions to improve modularity and readability.

**Task 3 — Add Unit Tests**

**Write unit tests for at least two functions in your code (for example, calculate\_average or find\_max).**

**Use the appropriate testing framework:**

* **C:** CUnit or Unity

Program.h

#ifndef PROGRAM\_H

#define PROGRAM\_H

int calculateSum(int arr[], int size);

float calculateAverage(int sum, int n);

int findMaximum(int arr[], int size);

#endif

Program.c

#include <stdio.h>

int calculateSum(int arr[], int n) {

int sum = 0;

for (int i = 0; i < n; i++) {

sum += arr[i];

}

return sum;

}

float calculateAverage(int sum, int n) {

return (float)sum / n;

}

int findMaximum(int arr[], int n) {

int max = arr[0];

for (int i = 1; i < n; i++) {

if (arr[i] > max) {

max = arr[i];

}

}

return max;

}

test\_program.c

#include<CUnit/CUnit.h>

#include<CUnit/Basic.h>

#include "program.h"

//test for calculateAverage()

void test\_calculateAverage(void)

{

CU\_ASSERT\_DOUBLE\_EQUAL(calculateAverage(15,3),5.0,0.001);

CU\_ASSERT\_DOUBLE\_EQUAL(calculateAverage(10,2),5.0,0.001);

CU\_ASSERT\_DOUBLE\_EQUAL(calculateAverage(0,5),0.0,0.001);

}

void test\_findMaximum(void)

{

int arr1[]={1,2,3,4,5};

int arr2[]={-5,-3,-10,-1};

int arr3[]={7};

CU\_ASSERT\_EQUAL(findMaximum(arr1,5),5);

CU\_ASSERT\_EQUAL(findMaximum(arr2,4),-1);

CU\_ASSERT\_EQUAL(findMaximum(arr3,1),7);

}

int main()

{

if(CUE\_SUCCESS!=CU\_initialize\_registry())

return CU\_get\_error();

//creating test suite

CU\_pSuite suite=CU\_add\_suite("program\_Test\_suite",0,0);

if(suite==NULL)

{

CU\_cleanup\_registry();

return CU\_get\_error();

}

//Adding test cases to the suite

if((CU\_add\_test(suite,"test\_calculateAverage",test\_calculateAverage)==NULL) || (CU\_add\_test(suite,"test\_findMaximum",test\_findMaximum)==NULL))

{

CU\_cleanup\_registry();

return CU\_get\_error();

}

//Run tests using the basic interface

CU\_basic\_set\_mode(CU\_BRM\_VERBOSE);

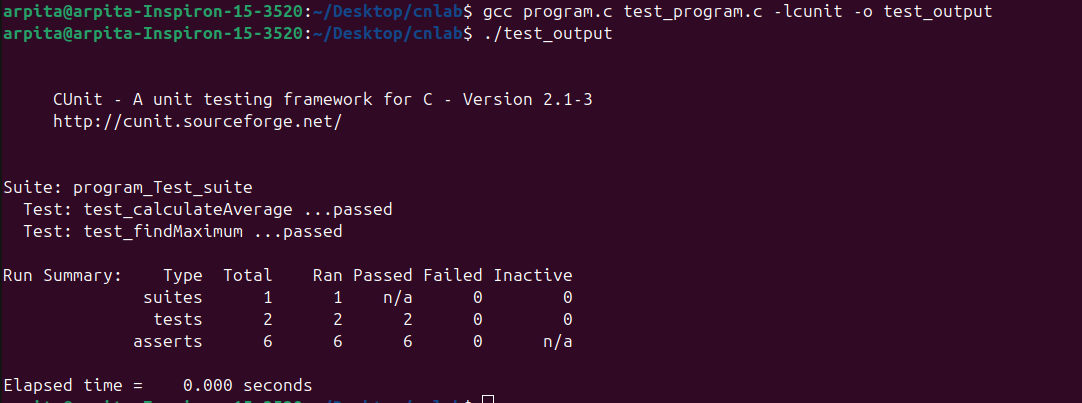
CU\_basic\_run\_tests();

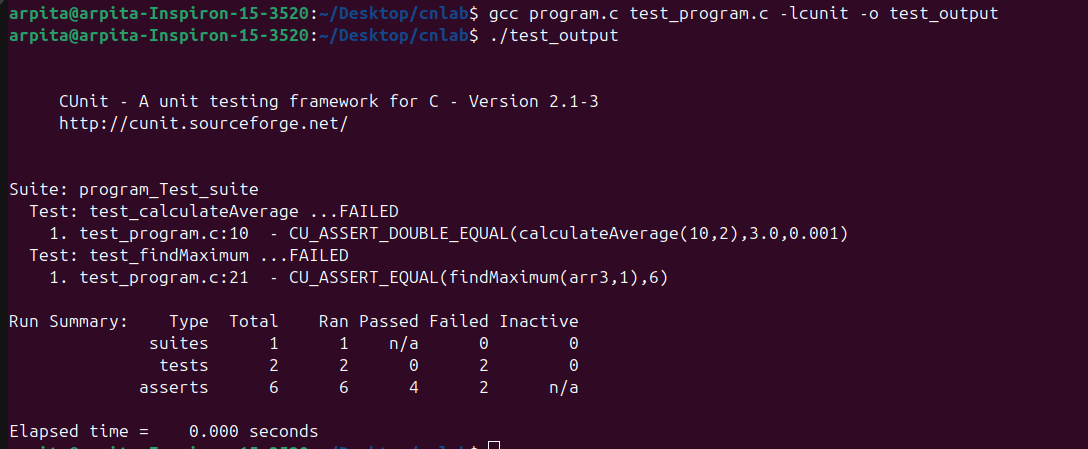
//cleanup

CU\_cleanup\_registry();

return CU\_get\_error();

}



****