Lab 7

Program Inspection, Debugging and Static Analysis

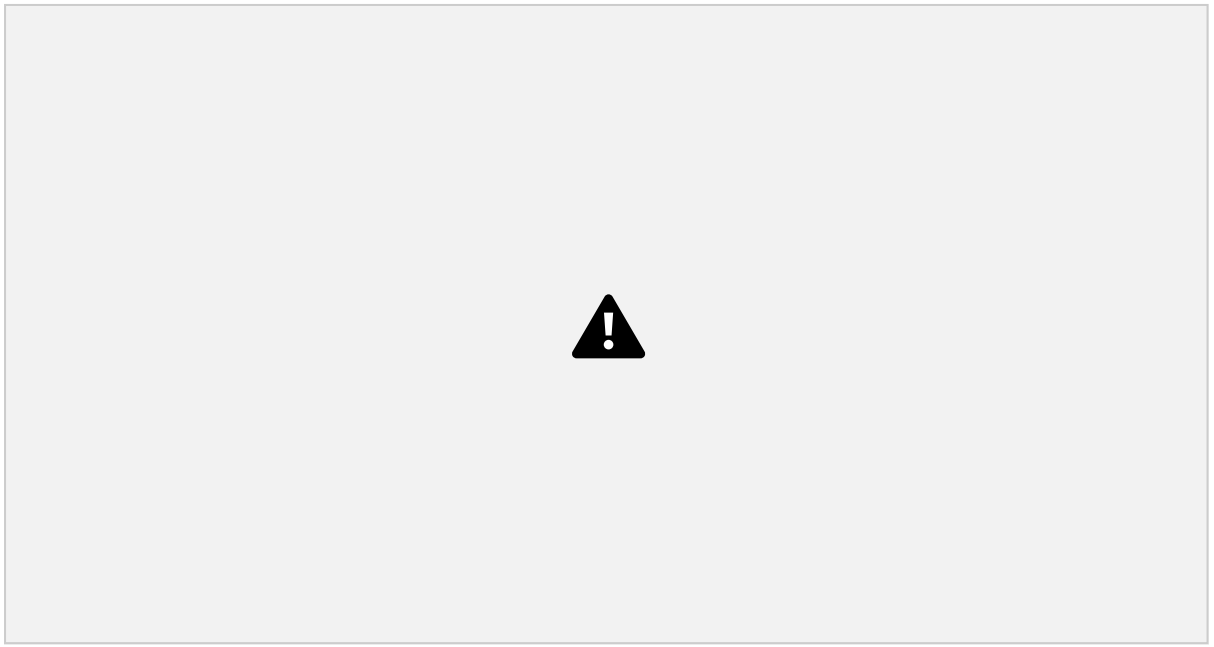
Raj Shah
202201403

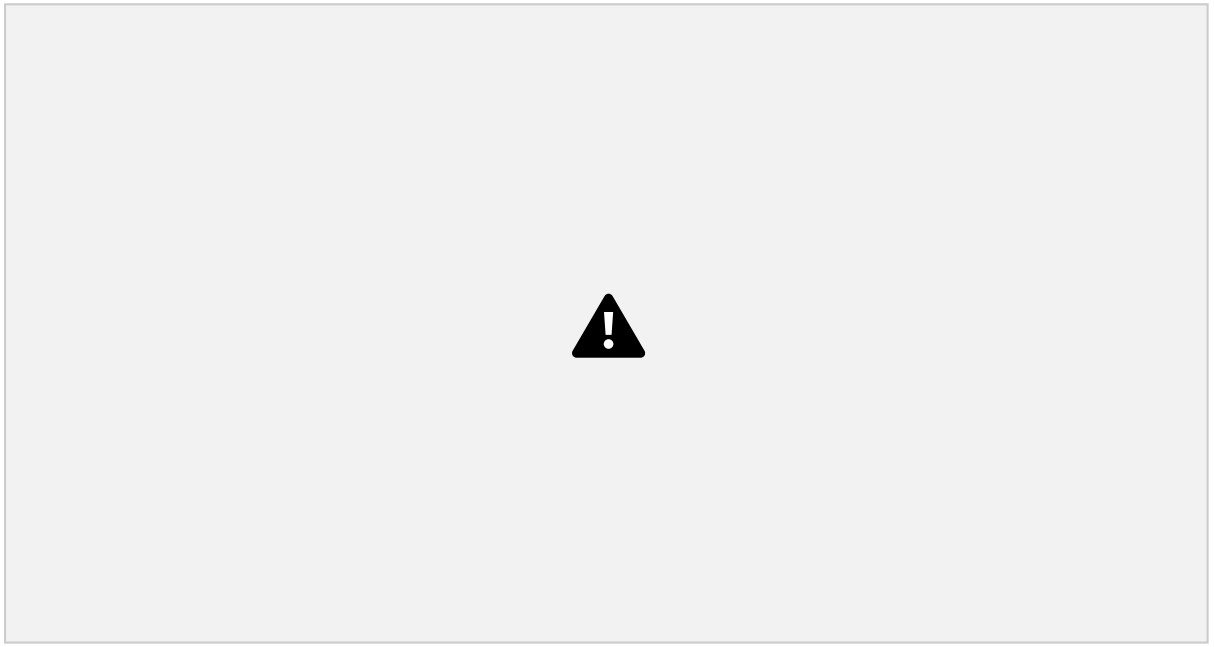
Static Analysis

Github Repository used: Covid-19 management system C++

Tool used is CPPCheck

Output of static analysis tool:





I. PROGRAM INSPECTION:

The code file (1304 LOC) is divided into fragments of about 300-400 LOC for inspection.

First Fragment:

Category A: Data Reference Errors

Error 1: Improper use of `center1`, `center2`, `center3` strings

The `center1`, `center2`, and `center3` values are hardcoded with "1center", "2center", and "3center". These values should be descriptive or stored in a way that makes them easy to reference or update (like in a separate data structure or array).

Category B: Data Declaration Errors

Error 1: String size declaration with `char[]` arrays

The use of `char name[100]`, `gender[100]`, `specialization[100]` is outdated and prone to buffer overflow issues. Modern C++ encourages the use of `std::string` for such variables to prevent errors associated with C-style strings.

Error 2: Unused member variables

Some member variables like `usn`, `add`, and `tm` are declared but never used anywhere in the class.

Category C: Computation Errors

Error 1: Incorrect vaccine dose summing

Variables like `sum_vaccine_c1`, `sum_vaccine_c2`, and `sum_vaccine_c3` are initialized to 0 but do not appear to be incremented anywhere when a vaccine is applied. This can lead to incorrect vaccine stock counts.

Category E: Control Flow and Functionality Error

Error 1: Overuse of `goto` statement

The `goto` statement used in the `admin()` function to loop back to the menu (`goto A;`) is considered bad practice because it makes the code harder to follow and maintain. Loops or function calls should be used instead.

Error 2: Redundant `break` after `goto

After the `goto` statement inside the switch cases (e.g., `goto A; break;`), the `break` statement is unnecessary since `goto` already transfers control to another part of the code.

Error 3: Exiting the program inappropriately

The program exits abruptly using `exit(0)` in the main menu. This approach skips any cleanup or final tasks that might need to run before termination.

Category G: Input Handling Errors

Error 1: Lack of input validation for critical data

- There is no input validation for critical inputs such as `choice` in the `menu()` function. This can cause the program to behave unexpectedly if invalid input is provided (e.g., non-integer values).

Error 2: Hardcoded credentials in `admin_password()`

- Hardcoding the admin username and password (`"sagar"` and `"pr123j"`) directly into the program is a security risk.

Second Fragment:

Category D: Logical Errors

Repeated `goto B`: The code uses `goto` statements frequently to return to the user menu (`B`). Using `goto` is considered poor practice as it complicates program flow and can lead to logical errors. Consider using a loop to repeatedly show the menu.

Category E: Input/Output Errors

No explicit input/output errors, but it could benefit from validation of `user_choice` input to ensure the user enters valid integer values.

Function: `covid_management::valid(string str)

Category A: Data Reference Errors

Variable `tm`: The function relies on a global or class variable `tm` without proper explanation or initialization in the code segment provided, which could lead to unexpected behavior if not handled carefully.

Category B: Data Declaration Errors

- The variable `usn` is used without context for its declaration in this specific function.

Category D: Comparison Errors

Recursive Call in `valid()`: The function uses recursion without a proper base case that handles all cases, which could lead to stack overflow if `usn` keeps being invalid.

Category G: Input/Output Errors

Input error: If the user enters a username that already exists, the prompt might be unclear on what the user should do next.

Function: `covid_management::user_password()

Category D: Comparison Errors

Lack of Password Validation: The password does not seem to have any validation (length, characters, etc.). Implementing validation for security reasons could be necessary.

Login Logic Issue: The login comparison `if (u_name == usn && u_pass == password)` is incorrect because `getline(filei, u_pass);` fetches password, but it's using the same variable for both input and file retrieval.

Category G: Input/Output Errors

Redundant Error Prompt: If the user enters an invalid option, they are sent back to the `user_password()` function. This can lead to continuous loops without clear exit criteria.

Category A: Data Reference Errors

Global variable access: The variables `sum_vaccine_c1`, `sum_vaccine_c2`, and `sum_vaccine_c3` seem to be used globally without initialization in this specific function, which could lead to data reference issues if not initialized elsewhere.

Category B: Data Declaration Errors

Ensure the variables used for the vaccine counts and `center_no` are declared with correct types and initialized before use.

Category C: Computation Errors

Vaccine Addition Logic: Repetitive code structure for adding vaccines to each center. This could be optimized to avoid repeating the logic for each case.

Category G: Input/Output Errors

The input for `center_no` could lead to issues if the user enters a non-integer value, which is not handled in the code. Input validation should be added to prevent crashes.

Third Fragment:

Category A: Data Reference Errors

1. Uninitialized Variables Variables like `sum_vaccine_c1`, `sum_vaccine_c2`, `sum_vaccine_c3`, `TOTAL_VACCINE`, `name`, `adhaar`, `phone_no`, etc., should be initialized before use to prevent undefined behavior.

2. File Handling The program assumes that files like `center1.txt`, `center2.txt`, `center3.txt`, and `Doctor_Data.dat` exist and are formatted correctly. Missing or misformatted files could lead to incorrect behavior.

Category B: Data-Declaration Errors

1. Data Type Mismatch Ensure that the data types of variables (like `adhaar`, `phone_no`, `identification_id`) are appropriate. For example, using `string` for `adhaar` is good, but make sure it is consistently treated as such across the code.

2. Buffer Overflow When using `cin.getline(name, 100)` or similar calls, ensure the input doesn't exceed the allocated buffer size. If the user enters a longer string, it may overflow and corrupt memory.

Category C: Computation Errors

1. Incorrect Computation Logic: The calculation for `s` in `display_vaccine_stock()` might not yield the expected result if any of the variables involved are uninitialized or contain incorrect values. Ensure that `TOTAL_VACCINE` is correctly defined and has a meaningful value.

Category D: Comparison Errors

1. String Comparison In `search_doctor_data()`, using `compare()` for `sadhaar`, `sidentification_id`, and `scenter` is appropriate. However, ensure that the comparisons are not case-sensitive if necessary (use `strcasecmp()` for case-insensitive comparison).

2. Character Comparisons: For the gender check, consider using `std::string` instead of a char array for better safety and handling.

Category E: Control-Flow Errors

1. Goto Statements The use of `goto` is discouraged as it can lead to hard-to-follow code and potential infinite loops. Replace with loops or function calls instead.

2. File Handling Logic There is repeated logic for file opening and checking for `!file`. This could be abstracted into a helper function to avoid duplication and improve maintainability.

Category F: Interface Errors

1. User Input Validation The input validation for `adhaar` and `phone_no` should ensure that the user is prompted again in a controlled manner rather than using `goto`.

2. Error Messages: Consistency in error messaging can improve the user experience. Ensure all error messages are user-friendly and provide guidance on the next steps.

Fourth Fragment:

Category A: Data Reference Errors

1. Uninitialized Data Members: The code uses class members (like `adhaar`, `age`, `profession`, etc.) without ensuring they are properly initialized before being accessed. This can lead to undefined behavior.

2. File Operations: There is a lack of checks after reading from the file. If the file does not contain enough data, accessing the uninitialized members could lead to unpredictable results.

Category B: Data-Declaration Errors

1. Data Types Ensure that the data types of members like `adhaar`, `age`, `gender`, etc., are declared correctly in the class definition.

2. Array Size The `sgender` array has a fixed size of 10. This may lead to buffer overflow if the user inputs a longer string.

Category C: Computation Errors

1. Incrementing Dose: In `update_patient_data`, the line `dose++` assumes `dose` is properly initialized. If not, it might lead to incorrect data being saved.

Category D: Comparison Errors

1. String Comparison: The code uses both `strcmp` and `compare` inconsistently. Using the same method for string comparison throughout would improve readability and maintainability.

Category E: Control-Flow Errors

1. Use of `goto` Statements: The `goto` statements make the flow of control hard to follow. It's advisable to use loops and functions to manage control flow, which can improve readability and maintainability.

2. Endless Loops: If a user repeatedly enters invalid data (e.g., an invalid Aadhar number), they will enter an infinite loop in `add_patient_data` and similar functions.

Category F: Interface Errors

1. User Input Validation: There is minimal validation for user inputs. While checks for the length of Aadhar and phone numbers exist, they could be more robust (e.g., checking for non-numeric characters in Aadhar and phone number).

File Not Found Handling: When the file is not found, the code merely prints a message but does not exit or handle the situation effectively, leading to further operations failing.

Category G: Input / Output Errors

1. File Handling The file operations assume success without verifying if the file opened successfully before performing read/write operations.

2. Output Messages: Messages could be improved for clarity. For example, "Press Any Key To Continue.." should be clearer (e.g., "Press any key to return to the main menu").

II. Debugging:

1. Armstrong Number Program

- Error: Incorrect computation of the remainder.
- Fix: Use breakpoints to check the remainder calculation.

Corrected Code:

```
class Armstrong {
public static void main(String args[]) { int num =
Integer.parseInt(args[0]); int n = num, check = 0, remainder; while (num
> 0) {
remainder = num % 10;
check += Math.pow(remainder, 3); num /= 10;
}
if (check == n) {
System.out.println(n + " is an Armstrong Number");
} else {
System.out.println(n + " is not an Armstrong Number");
}
}
}
```

2. GCD and LCM Program

- Errors:
    1. Incorrect while loop condition in GCD.
    2. Incorrect LCM calculation logic.
- Fix: Breakpoints at the GCD loop and LCM logic.

Corrected Code:

```
import java.util.Scanner; public class GCD_LCM {
static int gcd(int x, int y) { while (y != 0) {
```

```
int temp = y; y = x % y;
x = temp;
}
return x;
}
static int lcm(int x, int y) { return (x * y) / gcd(x, y);
}
public static void main(String args[]) { Scanner input = new
Scanner(System.in);
System.out.println("Enter the two numbers: "); int x = input.nextInt();
int y = input.nextInt();

System.out.println("The GCD of two numbers is: " + gcd(x, y));
System.out.println("The LCM of two numbers is: " + lcm(x, y));
input.close();
}
}
```

3. Knapsack Program

- Error: Incrementing n inappropriately in the loop.
- Fix: Breakpoint to check loop behavior.

Corrected Code:

```
public class Knapsack {
public static void main(String[] args) { int N = Integer.parseInt(args[0]); int
W = Integer.parseInt(args[1]);
int[] profit = new int[N + 1], weight = new int[N + 1]; int[][] opt = new int[N +
1][W + 1];
boolean[][] sol = new boolean[N + 1][W + 1]; for (int n = 1; n <= N; n++) {
for (int w = 1; w <= W; w++) { int option1 = opt[n - 1][w];
int option2 = (weight[n] <= w) ? profit[n] + opt[n - 1][w - weight[n]] :
Integer.MIN_VALUE;
opt[n][w] = Math.max(option1, option2); sol[n][w] = (option2 > option1);
}

}
}
}
```

4. Magic Number Program

- Errors:

    1. Incorrect condition in the inner while loop.
    2. Missing semicolons in expressions.

- Fix: Set breakpoints at the inner while loop and check variable values.

Corrected Code:

```
import java.util.Scanner;
public class MagicNumberCheck { public static void main(String args[]) {
Scanner ob = new Scanner(System.in); System.out.println("Enter the
number to be checked."); int n = ob.nextInt();
int sum = 0, num = n; while (num > 9) {
sum = num; int s = 0;
while (sum > 0) {
s = s * (sum / 10); // Fixed missing semicolon sum = sum % 10;
}

num = s;
}
if (num == 1) {
System.out.println(n + " is a Magic Number.");
} else {
System.out.println(n + " is not a Magic Number.");
}
}
}
```

5. Merge Sort Program

- Errors:

1. Incorrect array splitting logic.
2. Incorrect inputs for the merge method.

- Fix: Breakpoints at array split and merge operations.

Corrected Code:

```
import java.util.Scanner; public class MergeSort {
public static void main(String[] args) { int[] list = {14, 32, 67, 76, 23, 41,
58, 85};
System.out.println("Before: " + Arrays.toString(list)); mergeSort(list);
System.out.println("A er: " + Arrays.toString(list));
}


public static void mergeSort(int[] array) { if (array.length > 1) {
int[] le = le Half(array); int[] right = rightHalf(array); mergeSort(le );
mergeSort(right); merge(array, le , right);
}
}


public static int[] le Half(int[] array) { int size1 = array.length / 2;
int[] le = new int[size1]; System.arraycopy(array, 0, le , 0, size1); return le
;
}

public static int[] rightHalf(int[] array) { int size1 = array.length / 2;
int size2 = array.length - size1; int[] right = new int[size2];
System.arraycopy(array, size1, right, 0, size2); return right;
}


public static void merge(int[] result, int[] le , int[] right) { int i1 = 0, i2
= 0;
for (int i = 0; i < result.length; i++) {
if (i2 >= right.length || (i1 < le .length && le [i1] <= right[i2])) { result[i]
= le [i1];
i1++;
} else {
result[i] = right[i2]; i2++;
}
}
}
}
```

6. Multiply Matrices Program

- Errors:
    1. Incorrect loop indices.
    2. Wrong error message.
    - Fix: Set breakpoints to check matrix multiplication and correct messages.

Corrected Code:

```
import java.util.Scanner; class MatrixMultiplication {
public static void main(String args[]) {

int m, n, p, q, sum = 0, c, d, k; Scanner in = new Scanner(System.in);
System.out.println("Enter the number of rows and columns of the first
matrix");
m = in.nextInt(); n = in.nextInt();
int first[][] = new int[m][n];
System.out.println("Enter the elements of the first matrix"); for (c = 0;
c < m; c++)
for (d = 0; d < n; d++) first[c][d] = in.nextInt();
System.out.println("Enter the number of rows and columns of the second
matrix");
p = in.nextInt(); q = in.nextInt(); if (n != p)
System.out.println("Matrices with entered orders can't be multiplied.");
else {
int second[][] = new int[p][q];
int multiply[][] = new int[m][q];
System.out.println("Enter the elements of the second matrix"); for (c =
0; c < p; c++)
for (d = 0; d < q; d++) second[c][d] = in.nextInt();
for (c = 0; c < m; c++) {

for (d = 0; d < q; d++) { for (k = 0; k < p; k++) {
sum += first[c][k] * second[k][d];
}
multiply[c][d] = sum; sum = 0;
}
}
System.out.println("Product of entered matrices:"); for (c = 0; c < m;
c++) {
for (d = 0; d < q; d++) System.out.print(multiply[c][d] + "\t");
System.out.print("\n");
```

```
}
}
}
}
```

7. Quadratic Probing Hash Table Program

- Errors:

    1. Typos in insert, remove, and get methods.
    2. Incorrect logic for rehashing.

    - Fix: Set breakpoints and step through logic for insert, remove, and get methods.

Corrected Code:

```
import java.util.Scanner;
class QuadraticProbingHashTable { private int currentSize, maxSize;
private String[] keys, vals;
public QuadraticProbingHashTable(int capacity) { currentSize = 0;
maxSize = capacity;
keys = new String[maxSize]; vals = new String[maxSize];
}
public void insert(String key, String val) { int tmp = hash(key), i =
tmp, h = 1;
do {
if (keys[i] == null) { keys[i] = key; vals[i] = val; currentSize++;

return;
}
if (keys[i].equals(key)) { vals[i] = val;
return;
}
i += (h * h++) % maxSize;
} while (i != tmp);
}
```

```
public String get(String key) { int i = hash(key), h = 1; while (keys[i]
!= null) {
if (keys[i].equals(key)) return vals[i];
i = (i + h * h++) % maxSize;
}
return null;
}


public void remove(String key) { if (!contains(key)) return;
int i = hash(key), h = 1; while (!key.equals(keys[i]))
i = (i + h * h++) % maxSize;

keys[i] = vals[i] = null;
}


private boolean contains(String key) { return get(key) != null;
}

private int hash(String key) {
return key.hashCode() % maxSize;
}
}


public class HashTableTest {
public static void main(String[] args) { Scanner scan = new
Scanner(System.in);
QuadraticProbingHashTable hashTable = new
QuadraticProbingHashTable(scan.nextInt());
hashTable.insert("key1", "value1"); System.out.println("Value: " +
hashTable.get("key1"));
}
}
```

8. Sorting Array Program

- Errors:

    1. Incorrect class name with an extra space.
    2. Incorrect loop condition and extra semicolon.

- Fix: Set breakpoints to check the loop and class name.

Corrected Code:

```
import java.util.Scanner; public class AscendingOrder {
public static void main(String[] args) { int n, temp;
Scanner s = new Scanner(System.in); System.out.print("Enter the number of
elements: "); n = s.nextInt();
int[] a = new int[n];
System.out.println("Enter all the elements:"); for (int i = 0; i < n; i++) a[i]
= s.nextInt();
for (int i = 0; i < n; i++) {
for (int j = i + 1; j < n; j++) { if (a[i] > a[j]) {
temp = a[i]; a[i] = a[j]; a[j] = temp;
}
}
}

System.out.println("Sorted Array: " + Arrays.toString(a));
}
}
```

9. Stack Implementation Program

- Errors:

    1. Incorrect top-- instead of top++ in push.
    2. Incorrect loop condition in display.
    3. Missing pop method.

- Fix: Add breakpoints to check push, pop, and display methods.

Corrected Code:

```
public class StackMethods { private int top;
private int[] stack;

public StackMethods(int size) { stack = new int[size];
top = -1;
}

public void push(int value) { if (top == stack.length - 1) {
System.out.println("Stack full");
} else {
```

```
stack[++top] = value;
}

}

public void pop() { if (top == -1) {
System.out.println("Stack empty");
} else {
top--;
}
}


public void display() {
for (int i = 0; i <= top; i++) { System.out.print(stack[i] + " ");
}
System.out.println();
}
}
```

10. Tower of Hanoi Program

- Error: Incorrect increment/decrement in recursive call.
- Fix: Breakpoints at the recursive calls to verify logic.

Corrected Code:

```
public class TowerOfHanoi {
public static void main(String[] args) { int nDisks = 3;
doTowers(nDisks, 'A', 'B', 'C');
}

public static void doTowers(int topN, char from, char
inter, char to) { if (topN == 1) {
System.out.println("Disk 1 from " + from + " to " + to);
} else {
```

```
doTowers(topN - 1, from, to, inter);
System.out.println("Disk " + topN + " from " + from + "
to " + to); doTowers(topN - 1, inter, from, to);
}
}
}
```