IT314

DETROJA AVI-202201452

LAB07

INSPECTION AND DEBUGGING

CODE 1:

1. How many errors are there in the program? Mention the errors you have identified.

In the Armstrong number code, there are **two major errors**:

- **Remainder Calculation Error** (Category C: Computation Errors): The line remainder = num / 10; was incorrect because it was dividing instead of getting the last digit. It should be remainder = num % 10.
- **Incorrect Update of num** (Category C: Computation Errors): The line num = num % 10; incorrectly attempted to reduce num. Instead, it should divide the number by 10 to remove the last digit, i.e., num = num / 10;.

Both of these fall under **Category C: Computation Errors**. The errors are related to improper mathematical operations and misinterpretation of remainder logic.

2. Which category of program inspection would you find more effective?

The most effective categories for this particular program would be:

- **Category A: Data Reference Errors**: Ensuring that variables like remainder and num are correctly handled and referenced with the appropriate operations.

- **Category C: Computation Errors**: This category directly addresses issues with mathematical logic, such as remainder extraction and numeric updates.

3. Which type of error are you not able to identify using the program inspection?

A program inspection may **not easily catch logic-based errors** when they depend on specific conditions or when the logic works for some inputs but not for others. For example:

- **Boundary conditions**: Inspection may not cover test cases like 1-digit numbers or large Armstrong numbers unless manually tested.
- **Edge case handling**: The code would pass inspection but might fail in scenarios where inputs are invalid or unexpected. These errors are often identified during actual debugging or test runs.

4. Is the program inspection technique worth applicable?

Yes, **program inspection** is worth applying because it helps in identifying logical, computational, and data-reference-related issues at an early stage. By using the inspection checklist, many common errors can be found before runtime, saving time and reducing debugging complexity. However, inspections should be supplemented by actual test cases and debugging to cover edge cases and hidden logical errors.

- 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");
}
}
}
```

CODE 2;

1. How many errors are there in the program? Mention the errors you have identified.

There are **two errors** identified in the program:

1. **Error in GCD calculation (Computation Error)**:
    - The condition in the while loop in the gcd method was incorrect: while(a % b == 0). This should have been while(a % b != 0) to run the loop until the remainder becomes zero.
2. **Error in LCM calculation (Computation Error)**:
    - The condition if(a % x != 0 && a % y != 0) was incorrect. It should check if a is divisible by both numbers. The correct condition should be if(a % x == 0 && a % y == 0).

2. Which category of program inspection would you find more effective?

The most effective categories for this program are:

- **Category C: Computation Errors**: The main issues in the program are related to incorrect conditions in mathematical computations

for both the GCD and LCM methods. This category specifically focuses on errors in computations and mixed-mode operations, which is highly relevant for this program.

3. Which type of error are you not able to identify using the program inspection?

- **Boundary conditions or performance inefficiencies** might not be caught through program inspection alone. For example, if the inputs are very large, the current LCM calculation could be inefficient. Additionally, errors due to invalid or extreme input values might require runtime testing rather than inspection.
- **Edge case handling** for values such as negative inputs or zero inputs would also not be easily identified through inspection alone. Testing with a debugger or writing test cases would be needed to catch such errors.

4. Is the program inspection technique worth applying?

Yes, the **program inspection technique** is worth applying because it helps in catching **logical, computational, and structural errors** early in the process, saving debugging time later. While it might not cover all edge cases or performance inefficiencies, it is a valuable tool for ensuring correctness in computations and control flow.

- 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();

}

}
```

CODE 3:

1. How many errors are there in the program? Mention the errors you have identified.

There are **two errors** in the program:

1. **Array index error in the option1 calculation**:
    - The line int option1 = opt[n++][w]; is incorrect because the post-increment (n++) causes the code to skip items. This should be changed to int option1 = opt[n][w]; to avoid incrementing n prematurely.

2. **Incorrect index in the option2 calculation**:
    - The line option2 = profit[n-2] + opt[n-1][w-weight[n]]; is incorrect because profit[n-2] accesses the wrong index in the profit array. It should be profit[n] + opt[n-1][w-weight[n]]; to correctly reference the current item's profit.

2. Which category of program inspection would you find more effective?

The most effective categories for this program would be:

- **Category A: Data Reference Errors**: Ensuring the correct use of indices in arrays, as the errors in the code relate to improper indexing of the opt and profit arrays.
- **Category C: Computation Errors**: Since the logic for computing the maximum profit and optimal items is incorrect, this category is useful for identifying the issues related to loop conditions and calculations.

3. Which type of error are you not able to identify using the program inspection?

- **Edge cases and performance inefficiencies**: While the program inspection helps catch logical errors, it might not reveal how the code performs with large inputs (e.g., when N or W is very large), which could result in inefficiency or unexpected behavior.
- **Input validation errors**: Program inspection doesn't ensure that the code handles invalid inputs or edge cases like negative values for N or W, which would require runtime testing.

4. Is the program inspection technique worth applicable?

Yes, **program inspection** is definitely worth applying. It helps catch logic errors early in the development process, such as incorrect array indexing and flawed logic in conditionals. It can prevent costly debugging later. However, it should be supplemented with actual testing to catch runtime issues, performance problems, and edge cases that inspection alone might not identify.

- 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);

}}}}

CODE 4;

1. How many errors are there in the program? Mention the errors you have identified.

There are three errors in the program:

1. **Logical Error in Summing Digits**:
    - The inner loop incorrectly uses while (sum == 0), which causes it to run only if sum is zero. It should be while (temp > 0) to ensure all digits are summed correctly.
2. **Incorrect Calculation of the Magic Number**:
    - The program does not properly compute the sum of the digits until a single-digit number is reached. The inner loop needs to accumulate the sum of the digits of num rather than assigning sum directly to num.

3. **Missing Semicolon**:
    - There is a missing semicolon in the line sum=sum%10 (should be sum=sum%10;), which would cause a compilation error.

2. Which category of program inspection would you find more effective?

The most effective categories for this program would be:

- **Category A: Data Reference Errors**: This category is relevant due to the errors in variable use and the control of the digit summation.
- **Category C: Computation Errors**: This is essential for ensuring that the computation logic for summing digits and determining whether the number is a Magic number is implemented correctly.

3. Which type of error are you not able to identify using the program inspection?

- **Input Validation Errors**: The program does not validate inputs, such as ensuring that the number is a non-negative integer. Program inspection alone may miss these types of runtime issues.
- **Performance Inefficiencies**: While the logic can be inspected, it doesn't address how the program performs with large numbers or repeatedly calls the digit summation, which could lead to inefficiency.

4. Is the program inspection technique worth applying?

Yes, program inspection is definitely worth applying. It helps catch logical and syntactical errors early in the development process, such as issues in digit summation and incorrect loop conditions. This proactive approach can prevent costly debugging later. However, it should be supplemented with actual testing to capture runtime issues, performance problems, and edge cases that inspection alone might not identify.

- 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.");

}

}

}
```

CODE 5;

1. How many errors are there in the program? Mention the errors you have identified.

There are several errors in the program:

1. **Incorrect Array Slicing**:
    - The lines int[] left = leftHalf(array+1); and int[] right = rightHalf(array-1); incorrectly attempt to slice the array. They should be int[] left = leftHalf(array); and int[] right = rightHalf(array); without any arithmetic operations on array.
2. **Array Indices in the Merge Function**:
    - In the merge method call, the parameters left++ and right-- are incorrect. The merge function should simply use left and right without incrementing or decrementing, as this will lead to compilation errors. It should be merge(array, left, right);.
3. **Array Lengths in the Left and Right Half Functions**:
    - The leftHalf and rightHalf functions should calculate the size of the left half and right half correctly. Currently, they do so, but if the array length is odd, it will not split correctly since size1 will be array.length / 2, leading to a potential off-by-one error.
4. **Potential ArrayIndexOutOfBoundsException**:
    - In the merge function, the condition for the for-loop should ensure that result.length is equal to the sum of left.length and right.length. If the input arrays are empty or not sized correctly, this may lead to exceptions.
5. **Missing Return for Recursive Calls**:
    - The function mergeSort does not need a return value; however, it might benefit from the explicit assignment of the sorted halves back to the original array after merging, ensuring that result is actually the sorted version of the original input array.

2. Which category of program inspection would you find more effective?

The most effective categories for this program would be:

- **Category A: Data Reference Errors**: This category is relevant due to the incorrect handling of array references and slicing operations.
- **Category C: Computation Errors**: This category will help identify issues in the logic of merging and the calculations within the mergeSort method.

3. Which type of error are you not able to identify using the program inspection?

- **Performance Inefficiencies**: While the logic can be inspected, it may not address how the program performs with large inputs or how the merge operation can be optimized.
- **Edge Cases Handling**: Program inspection might miss how the algorithm handles special cases, such as empty arrays or arrays with one element, which might not be covered during inspection.

4. Is the program inspection technique worth applying?

Yes, program inspection is definitely worth applying. It helps catch logical and syntactical errors early in the development process, such as incorrect array slicing and improper handling of merging. This proactive approach can prevent costly debugging later. However, it should be supplemented with actual testing to capture runtime issues, performance problems, and edge cases that inspection alone might not identify

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++;
}
}
}
}
CODE 6;
```

1. How many errors are there in the program? Mention the errors you have identified.

There are several errors in the program:

1. **Incorrect Array Indexing**:
    - In the line sum = sum + first[c-1][c-k]*second[k-1][k-d];, the use of c-1 and k-1 will lead to incorrect indexing. Instead, it should be first[c][k] and second[k][d], which correctly accesses the elements of the matrices.
2. **Misleading Input Prompt**:
    - The second prompt for entering the number of rows and columns of the second matrix is incorrectly repeated as "first matrix." It should say "second matrix".
3. **Sum Variable Initialization**:
    - The variable sum is initialized outside the multiplication loop. Although it's reset to 0 after each calculation, it’s generally clearer to initialize it inside the loop where it’s used.
4. **Logic Error in the Loop Structure**:
    - The inner loop should iterate over n instead of p for the multiplication, as it corresponds to the number of columns in the first matrix. The correct loop structure should iterate through the shared dimension (n).
5. **Matrix Dimensions Validation**:
    - There is no check to validate if the resulting multiply matrix is appropriately sized based on the first matrix's rows and the second matrix's columns.

2. Which category of program inspection would you find more effective?

The most effective categories for this program would be:

- **Category A: Data Reference Errors**: This category applies because the program has issues with array indexing that can lead to accessing incorrect elements.
- **Category B: Logic Errors**: The multiplication logic is flawed, which falls under this category as it involves the incorrect implementation of matrix multiplication rules.

3. Which type of error are you not able to identify using the program inspection?

- **Edge Cases**: The program does not handle cases where the matrices contain invalid inputs or are empty. Program inspection might not reveal these situations.
- **Performance Inefficiencies**: While the code may work for small matrices, it may not be efficient for larger matrices, and inspection might miss optimization opportunities.

4. Is the program inspection technique worth applying?

Yes, program inspection is definitely worth applying. It helps catch logical and syntactical errors early in the development process, such as incorrect array indexing and flawed multiplication logic. This proactive approach can prevent costly debugging later. However, it should be supplemented with actual testing to capture runtime issues, performance problems, and edge cases that inspection alone might not identify.

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");
```

}

}

}

}

CODE 7;

- 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"));
}
}
```

CODE 8;

- 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));

}

}
```

CODE 9;

- 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();

}

}

CODE 10;

- 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);

}

}

}
```

I conducted the inspection of 1400 lines of code in sets of 200. I typed erroneous lines of code in each section based on categorization.

First 200 Lines Inspection:

Category A: Data Reference Errors

Uninitialized Variables: Variables like name, gender, age, and phone_no are declared but might not always be initialized when referenced, potentially causing issues if used before being assigned values.

Array Bounds: Arrays such as char specialization[100] and char name[100] lack explicit boundary checks, risking buffer overflow errors.

Category B: Data Declaration Errors

Implicit Declarations: Variables like adhaar and identification_id should be clearly declared with the correct data types before use.

Array Initialization: Arrays like char specialization[100] and char gender[100] would benefit from explicit initialization to avoid the use of undefined values.

Category C: Computation Errors

Mixed-mode Computations: Numeric strings like phone_no and adhaar are treated inconsistently. These should be handled as strings rather than integers to avoid computational errors.

Category E: Control-Flow Errors

Risky goto Usage: The goto statements in Aadhaar and mobile number validation could result in infinite loops if exit conditions aren’t well

defined. Replacing them with well-structured loops would be a safer approach.

Category F: Interface Errors

Parameter Mismatch: Functions like add_doctor() and display_doctor_data() should ensure proper matching of parameters with the caller functions.

Category G: Input/Output Errors

File Handling: Files like Doctor_Data.dat are opened without ensuring proper error handling or closure, which could lead to file access issues and runtime errors.

Second 200 Lines Inspection

Category A: Data Reference Errors

File Handling: Files such as Doctor_Data.dat and Patient_Data.dat are accessed without proper exception handling for file opening errors, like missing files or access issues.

Category B: Data Declaration Errors

Buffer Overflow Risks: Arrays like name[100], specialization[100], and gender[10] could overflow if inputs exceed their maximum size limits.

Category C: Computation Errors

Vaccine Stock Calculation: In the display_vaccine_stock() function, the total stock calculation lacks checks for negative values or overflows, which could result in miscalculations.

Category E: Control-Flow Errors

Excessive goto Usage: Functions like add_doctor() and add_patient_data() use goto statements for validation. These should be replaced with loops to enhance readability and control flow.

Category F: Interface Errors

String Comparison Issues: In the search_doctor_data() function, string comparisons could introduce errors if not managed correctly. Ensure consistent and proper string handling.

Category G: Input/Output Errors

File Closure: Files opened in functions like search_center() and display_vaccine_stock() are not always closed, which could lead to memory leaks or file lock issues.

Third 200 Lines Inspection

Category A: Data Reference Errors

File Error Handling: In functions like add_vaccine_stock() and display_vaccine_stock(), file operations for various centers (center1.txt, etc.) lack error handling after file opening. Always check that the file is successfully opened before proceeding.

Category B: Data Declaration Errors

Inconsistent Data Handling: Variables like adhaar and phone_no are inconsistently treated as numeric strings or integers across different functions, which can lead to errors.

Category C: Computation Errors

Vaccine Stock Calculation: In display_vaccine_stock(), stock calculations can fail if values are negative or uninitialized. Ensure all stock variables are initialized before use.

Category E: Control-Flow Errors

goto Usage: goto statements in functions like search_doctor_data() and add_doctor() complicate logic. Structured loops should replace these for better readability.

Category F: Interface Errors

Parameter Consistency: Ensure parameters like adhaar are passed consistently in functions like search_by_aadhar() across all subroutines.

Category G: Input/Output Errors

File Closure: Files like Doctor_Data.dat are not always closed in all branches, risking resource leaks.

Fourth 200 Lines Inspection

Category A: Data Reference Errors

Uninitialized Variables: Variables like maadhaar and file streams in functions like update_patient_data() and applied_vaccine() should be explicitly initialized to avoid issues with unset data.

Category B: Data Declaration Errors

Buffer Overflow Risk: Character arrays like sgender[10] and adhaar[12] risk overflow if input length isn't validated.

Category C: Computation Errors

Dose Incrementation: The dose++ operation in update_patient_data() could lead to invalid dose counts if not properly validated.

Category E: Control-Flow Errors

Overuse of goto: Heavy reliance on goto in functions like search_doctor_data() and add_patient_data() makes the control flow complex. Replacing goto with structured loops improves maintainability.

Category F: Interface Errors

String Comparison Issues: String comparisons in functions like search_by_aadhar() may not handle all cases properly. Ensure consistent validation logic for string operations.

Category G: Input/Output Errors

File Handling: Files like Patient_Data.dat and Doctor_Data.dat should include error checks when opened to avoid runtime failures.

Fifth 200 Lines Inspection

Category A: Data Reference Errors

Uninitialized Variables: In functions like update_patient_data() and search_doctor_data(), variables such as maadhaar should be properly initialized to avoid uninitialized value usage.

Category B: Data Declaration Errors

Buffer Overflow Risk: Arrays like sgender[10] are susceptible to overflow if input validation is not performed.

Category C: Computation Errors

Dose Increment Issues: The direct increment (dose++) in update_patient_data() needs validation to avoid erroneous dose counts.

Category E: Control-Flow Errors

Complex goto Statements: The use of goto in functions like search_doctor_data() and add_doctor() makes the code difficult to follow. Consider using loops for improved control flow.

Category F: Interface Errors

Parameter Handling: Functions like search_by_aadhar() should ensure correct and consistent parameter types are passed between functions.

Category G: Input/Output Errors

File Closure Issues: Files such as Patient_Data.dat and Doctor_Data.dat are not always properly closed, leading to resource management problems.

Final 300 Lines Inspection:

Category A: Data Reference Errors

- File Handling:
    - Files like center1.txt, center2.txt, and center3.txt are used across the add_vaccine_stock() and display_vaccine_stock() functions without proper error handling. Ensure error handling mechanisms are added in case of file access issues.

Category B: Data Declaration Errors

- Data Initialization:
    - Variables such as sum_vaccine_c1, sum_vaccine_c2, and sum_vaccine_c3 used in vaccine stock display should be initialized explicitly to avoid unintended behavior if le uninitialized.

Category C: Computation Errors

- Vaccine Stock Calculation:
    - In functions like add_vaccine_stock(), ensure that stock values are always positive and valid to avoid potential errors during subtraction in display_vaccine_stock().

Category E: Control-Flow Errors

- Excessive Use of goto Statements:

    - Throughout functions like add_doctor() and add_patient_data(), goto statements dominate the control flow. These should be replaced with loop constructs (while, for) for better readability and maintainability.

Category G: Input/Output Errors

- Inconsistent File Closing:
    - Several branches of file-handling code don't always close files correctly. Ensure every opened file is properly closed a er operations to prevent resource leaks.

Static Analysis Tool

Using cppcheck, I run static analysis tool for 1200+ lines of code used above for program inspection.

Results:

[202201452_lab7_2.c:1]: (information) Include file: <stdio.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.

[202201452_lab7_2.c:2]: (information) Include file: <stdlib.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.

[202201452_lab7_2.c:3]: (information) Include file: <sys/types.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.

[202201452_lab7_2.c:4]: (information) Include file: <sys/stat.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.

[202201452_lab7_2.c:5]: (information) Include file: <unistd.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.

[202201452_lab7_2.c:6]: (information) Include file: <dirent.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.

[202201452_lab7_2.c:7]: (information) Include file: <fcntl.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.

[202201452_lab7_2.c:8]: (information) Include file: <libgen.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.

[202201452_lab7_2.c:9]: (information) Include file: <errno.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.

[202201452_lab7_2.c:10]: (information) Include file: <string.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.

[202201452_lab7_2.c:0]: (information) Limiting analysis of branches. Use

--check-level=exhaustive to analyze all branches.

[202201452_lab7_2.c:116]: (warning) scanf() without field width limits can crash with huge input data.

[202201452_lab7_2.c:120]: (warning) scanf() without field width limits can crash with huge input data.

[202201452_lab7_2.c:126]: (warning) scanf() without field width limits can crash with huge input data.

[202201452_lab7_2.c:127]: (warning) scanf() without field width limits can crash with huge input data.

[202201452_lab7_2.c:133]: (warning) scanf() without field width limits can crash with huge input data.

[202201452_lab7_2.c:34]: (style) The scope of the variable 'ch' can be reduced. [202201452_lab7_2.c:115]: (style) The scope of the variable 'path2' can be reduced. [202201452_lab7_2.c:16]: (style) Parameter 'file' can be declared as pointer to const [202201452_lab7_2.c:55]: (style) Variable 'direntp' can be declared as pointer to const

[202201452_lab7_2.c:40]: (warning) Storing fgetc() return value in char variable and then comparing with EOF.

[202201452_lab7_3.c:1]: (information) Include file: <stdio.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.

[202201452_lab7_3.c:2]: (information) Include file: <stdlib.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.

[202201452_lab7_3.c:3]: (information) Include file: <sys/types.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.

[202201452_lab7_3.c:4]: (information) Include file: <sys/stat.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.

[202201452_lab7_3.c:5]: (information) Include file: <unistd.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.

[202201452_lab7_1.c:1]: (information) Include file: <stdio.h> not found. Please note: Cppcheck does not need standard library headers to get proper results

[202201452_lab7_1.c:2]: (information) Include file: <stdlib.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.

[202201452_lab7_1.c:3]: (information) Include file: <sys/types.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.

[202201452_lab7_1.c:4]: (information) Include file: <sys/stat.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.

[202201452_lab7_1.c:5]: (information) Include file: <unistd.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.

[202201452_lab7_1.c:6]: (information) Include file: <dirent.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.

[202201452_lab7_1.c:7]: (information) Include file: <fcntl.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.

[202201452_lab7_1.c:8]: (information) Include file: <libgen.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.

[202201452_lab7_1.c:9]: (information) Include file: <errno.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.

[202201452_lab7_1.c:29]: (style) The scope of the variable 'ch' can be reduced. [202201452_lab7_1.c:11]: (style) Parameter 'file' can be declared as pointer to const [202201452_lab7_1.c:50]: (style) Variable 'direntp' can be declared as pointer to const

[202201452_lab7_1.c:35]: (warning) Storing fgetc() return value in char variable and then comparing with EOF.

[Nursing-Management-System.cpp:4]: (information) Include file: <iostream> not found. Please note: Cppcheck does not need standard library headers to get proper results.

[Nursing-Management-System.cpp:5]: (information) Include file: <cstring> not found. Please note: Cppcheck does not need standard library headers to get proper results.

[Nursing-Management-System.cpp:6]: (information) Include file: <windows.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.

[Nursing-Management-System.cpp:7]: (information) Include file: <fstream> not found. Please note: Cppcheck does not need standard library headers to get proper results.

[Nursing-Management-System.cpp:8]: (information) Include file: <conio.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.

[Nursing-Management-System.cpp:9]: (information) Include file: <iomanip> not found. Please note: Cppcheck does not need standard library headers to get proper results.

[Nursing-Management-System.cpp:10]: (information) Include file: <cstdlib> not found. Please note: Cppcheck does not need standard library headers to get proper results.

[Nursing-Management-System.cpp:11]: (information) Include file: <string> not found. Please note: Cppcheck does not need standard library headers to get proper results.

[Nursing-Management-System.cpp:12]: (information) Include file: <unistd.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.

[Nursing-Management-System.cpp:562]: (portability) fflush() called on input stream 'stdin' may result in undefined behaviour on non-linux systems.

[Nursing-Management-System.cpp:565]: (portability) fflush() called on input stream 'stdin' may result in undefined behaviour on non-linux systems.

[Nursing-Management-System.cpp:614]: (portability) fflush() called on input stream 'stdin' may result in undefined behaviour on non-linux systems.

[Nursing-Management-System.cpp:1121]: (portability) fflush() called on input stream 'stdin' may result in undefined behaviour on non-linux systems.

[Nursing-Management-System.cpp:538]: (style) C-style pointer casting [Nursing-Management-System.cpp:619]: (style) C-style pointer casting [Nursing-Management-System.cpp:641]: (style) C-style pointer casting [Nursing-Management-System.cpp:646]: (style) C-style pointer casting [Nursing-Management-System.cpp:749]: (style) C-style pointer casting [Nursing-Management-System.cpp:758]: (style) C-style pointer casting [Nursing-Management-System.cpp:788]: (style) C-style pointer casting [Nursing-Management-System.cpp:797]: (style) C-style pointer casting [Nursing-Management-System.cpp:827]: (style) C-style pointer casting [Nursing-Management-System.cpp:836]: (style) C-style pointer casting [Nursing-Management-System.cpp:866]: (style) C-style pointer casting [Nursing-Management-System.cpp:875]: (style) C-style pointer casting [Nursing-Management-System.cpp:907]: (style) C-style pointer casting [Nursing-Management-System.cpp:973]: (style) C-style pointer casting [Nursing-Management-System.cpp:982]: (style) C-style pointer casting [Nursing-Management-System.cpp:1012]: (style) C-style pointer casting [Nursing-Management-System.cpp:1021]: (style) C-style pointer casting [Nursing-Management-System.cpp:1051]: (style) C-style pointer casting [Nursing-Management-System.cpp:1060]: (style) C-style pointer casting [Nursing-Management-System.cpp:1090]: (style) C-style pointer casting [Nursing-Management-System.cpp:1099]: (style) C-style pointer casting [Nursing-Management-System.cpp:1181]: (style) C-style pointer

casting [Nursing-Management-System.cpp:1207]: (style) C-style pointer casting [Nursing-Management-System.cpp:1216]: (style) C-style pointer casting [Nursing-Management-System.cpp:1307]: (style) C-style pointer casting [Nursing-Management-System.cpp:1317]: (style) C-style pointer casting

[Nursing-Management-System.cpp:1320]: (style) C-style pointer casting

[Nursing-Management-System.cpp:427]: (style) Consecutive return, break, continue, goto or throw statements are unnecessary.

[Nursing-Management-System.cpp:443]: (style) Consecutive return, break, continue, goto or throw statements are unnecessary.

[Nursing-Management-System.cpp:459]: (style) Consecutive return, break, continue, goto or throw statements are unnecessary.

[Nursing-Management-System.cpp:892]: (style) Consecutive return, break, continue, goto or throw statements are unnecessary.

[Nursing-Management-System.cpp:306]: (style) The scope of the variable 'usern' can be reduced.

[Nursing-Management-System.cpp:48] -> [Nursing-Management-System.cpp:277]: (style)

Local variable 'user' shadows outer function

[Nursing-Management-System.cpp:40] -> [Nursing-Management-System.cpp:304]: (style)

Local variable 'c' shadows outer variable

[Nursing-Management-System.cpp:275]: (performance) Function parameter 'str' should be passed by const reference.

[Nursing-Management-System.cpp:277]: (style) Unused variable: user
[Nursing-Management-System.cpp:304]: (style) Unused variable: c.