IT314 Lab-7
Program Inspection, Debugging and Static Analysis

Malay Sidapara
202201488

PROGRAM INSPECTION

The program inspection has been conducted on the following code of more than 2000 LOC:
https://github.com/godotengine/godot/blob/master/editor/project_converter_3_to_4.cpp

Category A: Data Reference Errors

- No uninitialized variables found
- Array bounds are respected using ‘vector.size()’
- No non-integer array subscripts observed; the code handles indexing correctly.
- No pointer or reference issues found, as pointers are not used.
- No memory aliasing issues present (no ‘EQUIVALENCE’ or similar constructs).
- No type mismatches identified with variables or regex processing.
- No addressing issues, as there is no bit-level memory manipulation.
- No incorrect pointer assignments detected.
- Data structures seem consistently defined across functions.
- No off-by-one errors noticed in string or array indexing.

.

Category B: Data Declaration Errors

- All variables appear explicitly declared.
- No unexpected default attributes observed in variable declarations.
- Initialization of variables seems correct.
- Variable lengths and data types are appropriate.
- No issues with memory types and initializations.
- No variables with confusingly similar names were detected.

Category C: Computation Errors

- No inconsistent data types used in computations.
- No mixed-mode computations found.
- No issues with variables of different lengths in computations.
- No target variable type mismatches in assignments.
- No overflow or underflow risks identified.
- No division by zero detected.
- No base-2 representation inaccuracies observed.
- No values exceeding valid ranges found in assignments.
- Operator precedence is correct in all expressions.
- No invalid uses of integer arithmetic detected.

Category D: Comparison Errors

- No comparisons between variables of different types observed.
- No mixed-mode or different-length comparisons detected.
- Comparison operators seem correct and consistent.
- Boolean expressions are correct and represent the intended logic.
- No invalid uses of Boolean operators found.
- No problematic floating-point comparisons identified.
- Operator precedence in Boolean expressions is correct.
- No issues with Boolean expression evaluation affecting control flow.

Category E: Control Flow Errors

- No multiway branch index issues identified.
- All loops appear to terminate correctly.
- No infinite loops or module termination issues detected.
- No skipped loop executions observed.
- No loop fall-through issues found.
- No off-by-one errors detected in loops.
- No issues with group or block statements (no missing brackets).
- Decision logic seems exhaustive; no unhandled cases identified.

Category F: Interface Errors

- Number of parameters matches the number of arguments in all function calls.
- Attributes of parameters and arguments are consistent.
- No unit system mismatches identified.
- Number of arguments transmitted to other modules is correct.
- Argument attributes match parameter attributes in other modules.
- Units of arguments match parameter units.
- No issues with built-in function arguments.
- No input-only parameters modified in subroutines.
- No inconsistencies in global variable definitions across modules.

Category G: I/O Errors

- No explicit file declarations are used in this code.
- No file attributes in ‘OPEN’ statements to verify.
- No issues with file memory capacity, as no files are handled.
- No files to check for proper opening.
- No files to check for proper closure.
- No end-of-file conditions present.
- No I/O error conditions to handle.
- No spelling or grammatical errors in output text (as no output text is generated).

Category H: Other Checks

- No unused or underused variables identified.
- No attribute inconsistencies in compiler listings.

- No warning or informational messages found during compilation.
- Program robustness seems adequate, but input validation checks could be added.
- No missing functions detected.

Program Inspections:

- How many errors are there in the program?

➔ No major errors are immediately visible, but careful review of array bounds and regex operations is required.

- Which category of program inspection would you find more effective?

➔ Category A (Data Reference Errors) and Category E (Control Flow) would likely reveal the most issues in a string processing program like this.

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

➔ Without runtime testing, logical errors involving regex processing or string manipulation nuances might be hard to identify.

- Is the program inspection technique worth applying?

➔ Yes, this inspection process helps systematically review common error types, especially in projects that involve a lot of text processing and conversions.

CODE DEBUGGING

Debugging involves identifying, analyzing, and resolving suspected errors in code.

1. Armstrong

The program contains the following errors:

- The expression ‘remainder = num / 10’ is incorrect. This should calculate the last digit of the number, but it divides instead. The correct expression should be ‘remainder = num % 10’.
- The expression ‘num = num % 10’ is incorrect. This is trying to reduce the number but is using the wrong operator. The correct way to remove the last digit should be ‘num = num / 10’.

Two breakpoints are needed to detect and fix the issues:

- At ‘remainder = num / 10’, to observe how the remainder is being calculated incorrectly.
- At ‘num = num % 10’, to observe the wrong reduction of ‘num’.

Steps to fix the errors:

- Correct the remainder calculation to extract the last digit by changing ‘remainder = num / 10’ to ‘remainder = num % 10’.
- Correct the way the number is reduced by changing ‘num = num % 10’ to ‘num = num / 10’.

Here is the corrected code fragment:

```
//Armstrong Number

class Armstrong {
   public static void main(String args[]) {
      int num = Integer.parseInt(args[0]);
      int n = num; //use to check at last time
      int check = 0, remainder;

      while (num > 0) {
         remainder = num % 10; // Get last digit
         check = check + (int) Math.pow(remainder, 3); // Cube and add
         num = num / 10; // Remove last digit
      }

      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

Errors in the program:

- In the ‘gcd’ method, the issue lies in the while loop condition. To correctly compute the GCD, the condition should be updated to ‘while (a % b != 0)’.
- In the ‘lcm’ method, although there’s no syntax error, the algorithm uses an inefficient brute-force approach to calculate the LCM.

To resolve these errors:

- A breakpoint should be placed to address the issue in the ‘gcd’ method by modifying the while loop condition to ‘while (a % b != 0)’.

Here is the corrected code fragment:

```
import java.util.Scanner;

public class GCD_LCM {
        static int gcd(int x, int y) {
                int a, b;
                a = (x > y) ? x : y; // a is the greater number
```

```
        b = (x < y) ? x : y; // b is the smaller number

        while (b != 0) { // Corrected the while loop condition
            int temp = b;
            b = a % b;
            a = temp;
        }
        return a;
    }
    static int lcm(int x, int y) {
        int a;
        a = (x > y) ? x : y; // a is greater number
        while (true) {
            if (a % x == 0 && a % y == 0)
            return a;
            ++a;
        }
    }

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

Please note that I've fixed the errors in the `gcd` method by modifying the while loop condition and applying the more efficient Euclidean algorithm for GCD calculation. The corrected code is now fully executable.

3. Knapsack

Here are the errors:

- Incorrect use of increment and decrement operators.

The number of breakpoints required to fix these errors may vary. Typically, you would place breakpoints at important points in the code where you want to halt execution for debugging purposes.

Here is the corrected code fragment:

```
// Knapsack
public class Knapsack {
	public static void main(String[] args) {
		int N = Integer.parseInt(args[0]); // number of items
		int W = Integer.parseInt(args[1]); // maximum weight of knapsack

		int[] profit = new int[N];
		int[] weight = new int[N];

		// generate random instance, items 0..N-1
		for (int n = 0; n < N; n++) {
			profit[n] = (int) (Math.random() * 1000);
			weight[n] = (int) (Math.random() * W);
		}

		// opt[n][w] = max profit of packing items 0..n with weight limit w
		// sol[n][w] = does opt solution to pack items 0..n with weight limit w include item
		n?
		int[][] opt = new int[N][W + 1];
		boolean[][] sol = new boolean[N][W + 1];

		for (int n = 0; n < N; n++) {
			for (int w = 1; w <= W; w++) {
				// don't take item n
				int option1 = opt[n - 1][w];

				// take item n
				int option2 = Integer.MIN_VALUE;
				if (weight[n] <= w) {
					option2 = profit[n] + opt[n - 1][w - weight[n]];
				}

				// select the better of two options
				opt[n][w] = Math.max(option1, option2);
				sol[n][w] = (option2 > option1);
			}
		}
```

```
                // determine which items to take
                boolean[] take = new boolean[N];
                for (int n = N - 1, w = W; n >= 0; n--) {
                        if (sol[n][w]) {
                                take[n] = true;
                                w -= weight[n];
                        } else {
                                take[n] = false;
                        }
                }

                // print results
                System.out.println("Item" + "\t" + "Profit" + "\t" + "Weight" + "\t" + "Take");
                for (int n = 0; n < N; n++) {
                        System.out.println(n + "\t" + profit[n] + "\t" + weight[n] + "\t" + take[n]);
                }
        }
}
```

4. Magic Number

The program contains two errors:

- The condition in the inner while loop should be ‘while (sum != 0)’ instead of ‘while (sum == 0)’.
- The variable ‘s’ should be initialized as ‘s = 1’ before the inner while loop, rather than ‘s = 0’.

To resolve these errors, at least two breakpoints are needed—one for each issue.

Here is the corrected code fragment:

```
 // Program to check if number is Magic number in JAVA
import java.util.*;
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=1;
```

```
                while(sum!=0) {
                        s=s*(sum/10);
                        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

There are several errors in the program:

- Incorrect array passing: ‘leftHalf(array+1)’ and ‘rightHalf(array-1)’ should pass the whole array.
- Incorrect merging logic: The call ‘merge(array, left++, right--)’ should be ‘merge(array, left, right)’.

Three breakpoints are required:

- At ‘leftHalf(array+1)’ to catch array passing issues.
- At ‘rightHalf(array-1)’ for the same reason.
- At ‘merge(array, left++, right--)’ to check merging logic.

Steps to fix the errors:

- Changed ‘leftHalf(array+1)’ and ‘rightHalf(array-1)’ to ‘leftHalf(array)’ and ‘rightHalf(array)’.
- Updated ‘merge(array, left++, right--)’ to ‘merge(array, left, right)’.

Here is the corrected code fragment:

```
import java.util.Arrays;

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("after: " + Arrays.toString(list));
   }
```

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

    public static int[] leftHalf(int[] array) {
        int size1 = array.length / 2;
        int[] left = new int[size1];
        for (int i = 0; i < size1; i++) {
            left[i] = array[i];
        }
        return left;
    }

    public static int[] rightHalf(int[] array) {
        int size1 = array.length / 2;
        int size2 = array.length - size1;
        int[] right = new int[size2];
        for (int i = 0; i < size2; i++) {
            right[i] = array[i + size1];
        }
        return right;
    }

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

6. Matrix Multiplication

There are two errors in the program:

- Incorrect index usage in multiplication: In the line sum = sum + first[c-1][c-k]*second[k-1][k-d];, the indices are incorrect. It should use first[c][k] and second[k][d].
- Logical error in row and column prompts: The prompts for the dimensions of the second matrix are incorrect. The same prompt is used twice for the first matrix.

Three breakpoints are required:

- At the multiplication line to verify index correctness.
- Before the second matrix input to confirm matrix dimensions are set correctly.
- After the matrix multiplication loop to check the final product before printing.

Steps to fix the errors:

- Correct the multiplication logic to use proper indices: 'sum = sum + first[c][k] * second[k][d];'.
- Change the prompts for the second matrix dimensions to correctly ask for "Enter the number of rows and columns of the second matrix".
- Ensure clear formatting for the output to separate values with tabs or spaces.

Here is the corrected code fragment:

```
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 first matrix");
      m = in.nextInt();
      n = in.nextInt();

      int first[][] = new int[m][n];

      System.out.println("Enter the elements of 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 second matrix"); // Corrected
prompt
```

```
        p = in.nextInt();
        q = in.nextInt();

        if (n != p)
            System.out.println("Matrices with entered orders can't be multiplied with each other.");
        else {
            int second[][] = new int[p][q];
            int multiply[][] = new int[m][q];

            System.out.println("Enter the elements of 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 < n; k++) { // Corrected the loop range
                        sum = sum + first[c][k] * second[k][d]; // Corrected index usage
                    }
                    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"); // Properly formatted output
                System.out.print("\n");
            }
        }
        in.close(); // Close scanner to avoid resource leak
    }
}
```

7. Quadratic Probing

There are several errors in the program:

- Syntax Error:
    - ➔ In the ‘insert’ method, the line ‘i + = (i + h / h--) % maxSize;’ has an incorrect space. It should be ‘i += (i + h) % maxSize;’.
    - ➔ In the ‘get’ method, the line ‘i = (i + h * h++) % maxSize;’ is also incorrect; it should be ‘i = (i + h * h) % maxSize;’ to avoid incrementing ‘h’ too early.

- Incorrect Rehashing Logic: In the ‘remove’ method, the loop ‘while (!key.equals(keys[i]))’ should account for the case when ‘keys[i]’ is ‘null’, indicating the key does not exist.
- Logical Flow: The program does not handle the scenario where a user might want to enter multiple key-value pairs in one go. It attempts to read multiple values directly after the "Enter key and value" prompt without giving a clear structure for entering multiple pairs.

Three breakpoints are required:
- In the ‘insert’ method to check the logic for handling key-value pairs.
- In the ‘get’ method to verify the hash table lookup logic.
- In the ‘remove’ method to ensure that key deletion and rehashing work correctly.

Steps to fix the errors:
- Correct Syntax: Change ‘i + = (i + h / h--) % maxSize;’ to ‘i += (i + h) % maxSize;’ and ‘i = (i + h * h++) % maxSize;’ to ‘i = (i + h * h) % maxSize;’.
- Improve logic in ‘remove’ method: Add a ‘null’.

Here is the corrected code fragment:

```
import java.util.Scanner;

/** Class QuadraticProbingHashTable **/
class QuadraticProbingHashTable {
    private int currentSize, maxSize;
    private String[] keys;
    private String[] vals;

    /** Constructor **/
    public QuadraticProbingHashTable(int capacity) {
        currentSize = 0;
        maxSize = capacity;
        keys = new String[maxSize];
        vals = new String[maxSize];
    }

    /** Function to clear hash table **/
    public void makeEmpty() {
        currentSize = 0;
        keys = new String[maxSize];
        vals = new String[maxSize];
    }

    /** Function to get size of hash table **/
    public int getSize() {
```

```
    return currentSize;
}

/** Function to check if hash table is full **/
public boolean isFull() {
    return currentSize == maxSize;
}

/** Function to check if hash table is empty **/
public boolean isEmpty() {
    return getSize() == 0;
}

/** Function to check if hash table contains a key **/
public boolean contains(String key) {
    return get(key) != null;
}

/** Function to get hash code of a given key **/
private int hash(String key) {
    return Math.abs(key.hashCode()) % maxSize; // Ensure positive index
}

/** Function to insert key-value pair **/
public void insert(String key, String val) {
    int tmp = hash(key);
    int 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; // Fixed syntax
        h++; // Increment h for quadratic probing
        i %= maxSize; // Ensure valid index
    } while (i != tmp);
}
```

```
    /** Function to get value for a given key **/
    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; // Fixed logic
            h++; // Increment h
        }
        return null;
    }

    /** Function to remove key and its value **/
    public void remove(String key) {
        if (!contains(key))
            return;

        /** find position key and delete **/
        int i = hash(key), h = 1;
        while (!key.equals(keys[i]))
            i = (i + (h * h)) % maxSize;

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

        /** rehash all keys **/
        for (i = (i + (h * h)) % maxSize; keys[i] != null; i = (i + (h * h)) % maxSize) {
            String tmp1 = keys[i], tmp2 = vals[i];
            keys[i] = vals[i] = null;
            currentSize--;
            insert(tmp1, tmp2);
        }
        currentSize--;
    }

    /** Function to print HashTable **/
    public void printHashTable() {
        System.out.println("\nHash Table: ");
        for (int i = 0; i < maxSize; i++)
            if (keys[i] != null)
                System.out.println(keys[i] + " " + vals[i]);
        System.out.println();
    }
}
```

```
/** Class QuadraticProbingHashTableTest **/
public class QuadraticProbingHashTableTest {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.println("Hash Table Test\n\n");
        System.out.println("Enter size");
        /** make object of QuadraticProbingHashTable **/
        QuadraticProbingHashTable qpht = new QuadraticProbingHashTable(scan.nextInt());

        char ch;
        /** Perform QuadraticProbingHashTable operations **/
        do {
            System.out.println("\nHash Table Operations\n");
            System.out.println("1. insert ");
            System.out.println("2. remove");
            System.out.println("3. get");
            System.out.println("4. clear");
            System.out.println("5. size");

            int choice = scan.nextInt();
            switch (choice) {
                case 1:
                    System.out.println("Enter key and value");
                    qpht.insert(scan.next(), scan.next());
                    break;
                case 2:
                    System.out.println("Enter key");
                    qpht.remove(scan.next());
                    break;
                case 3:
                    System.out.println("Enter key");
                    System.out.println("Value = " + qpht.get(scan.next()));
                    break;
                case 4:
                    qpht.makeEmpty();
                    System.out.println("Hash Table Cleared\n");
                    break;
                case 5:
                    System.out.println("Size = " + qpht.getSize());
                    break;
                default:
                    System.out.println("Wrong Entry \n ");
                    break;
            }
```

```
            /** Display hash table **/
            qpht.printHashTable();

            System.out.println("\nDo you want to continue (Type y or n) \n");
            ch = scan.next().charAt(0);
        } while (ch == 'Y' || ch == 'y');

        scan.close(); // Close scanner to avoid resource leak
    }
}
```

8. Sorting Array

The program contains the following errors:

- Incorrect Loop Condition in the Outer Loop: The line ‘for (int i = 0; i >= n; i++);’ has an incorrect condition. It should be ‘i < n’ instead of ‘i >= n,’ and the semicolon at the end of the loop should be removed.
- Incorrect Sorting Logic: The current logic in the nested loop will not sort the array correctly. The condition in the ‘if’ statement should be ‘a[i] > a[j]’ for ascending order.

Two breakpoints are needed to debug the program effectively:

- At the end of the outer loop before the sorting logic to check if the loop is iterating correctly.
- After the sorting logic to verify that the array is sorted correctly before printing.

Steps to fix the errors:

- Fix the Outer Loop Condition: Change ‘for (int i = 0; i >= n; i++);’ to ‘for (int i = 0; i < n; i++).’
- Correct Sorting Logic: Update the condition from ‘if (a[i] <= a[j])’ to ‘if (a[i] > a[j])’ to ensure correct swapping for ascending order.

Here is the corrected code fragment:

```
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 no. of elements you want in array: ");
        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();
        }

        // Corrected the outer loop condition and removed the semicolon
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                // Corrected the comparison operator
                if (a[i] > a[j]) {
                    temp = a[i];
                    a[i] = a[j];
                    a[j] = temp;
                }
            }
        }

        // Improved output formatting
        System.out.print("Ascending Order: ");
        for (int i = 0; i < n; i++) {
            System.out.print(a[i]);
            if (i < n - 1) { // Print comma for all but the last element
                System.out.print(", ");
            }
        }
    }
}
```

9. Stack Implementation

The program contains the following errors:

- In the ‘push’ method, ‘top–’ should be ‘top++’. The current logic decreases ‘top’, which is incorrect when inserting elements into the stack.
- The condition in the ‘for’ loop in the ‘display’ method is incorrect. The loop should run from ‘0’ to ‘top’ to display the elements, but it currently runs from ‘i > top’, which doesn't print anything.
- In the ‘pop’ method, the ‘top++’ logic works but does not return or remove the value being popped. You might want to decrement ‘top’ instead of incrementing it.

Two breakpoints are needed to debug the program effectively:

- Before the ‘push’ method to check if the stack is correctly updated.
- Before the ‘display’ method to confirm if elements are printed properly.

Steps to fix the errors:

- Change ‘top–’ to ‘top++’ to correctly insert elements.

- Change ‘for (int i = 0; i > top; i++) to for (int i = 0; i <= top; i++)’ to properly iterate and print the stack elements.
- Remove or return the top element by decrementing ‘top’.

Here is the corrected code fragment:

```
import java.util.Arrays;

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

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

    // Corrected push method
    public void push(int value) {
        if (top == size - 1) {
            System.out.println("Stack is full, can't push a value");
        } else {
            top++;
            stack[top] = value;
        }
    }

    // Corrected pop method
    public void pop() {
        if (!isEmpty()) {
            top--;
        } else {
            System.out.println("Can't pop...stack is empty");
        }
    }

    public boolean isEmpty() {
        return top == -1;
    }

    // Corrected display method
    public void display() {
```

```
        if (isEmpty()) {
            System.out.println("Stack is empty");
        } else {
            for (int i = 0; i <= top; i++) {
                System.out.print(stack[i] + " ");
            }
            System.out.println();
        }
    }
}

public class StackReviseDemo {
    public static void main(String[] args) {
        StackMethods newStack = new StackMethods(5);
        newStack.push(10);
        newStack.push(1);
        newStack.push(50);
        newStack.push(20);
        newStack.push(90);

        newStack.display(); // Displays: 10 1 50 20 90
        newStack.pop();
        newStack.pop();
        newStack.pop();
        newStack.pop();
        newStack.display(); // Displays: 10
    }
}
```

10. Tower of Hanoi

The program has the following errors:

- ‘topN++’, ‘inter–’, ‘from+1’, and ‘to+1’ are syntactically incorrect and logically inappropriate. Incrementing or decrementing these variables modifies the characters, which is not desired in recursive function calls.
- There is a missing semicolon after the recursive call ‘doTowers(topN++, inter--, from+1, to+1)’.

Two breakpoints are needed to debug the program effectively:

- Before the recursive call ‘doTowers(topN - 1, from, to, inter)’ to inspect the recursion behavior.
- Before the second recursive call ‘doTowers(topN - 1, inter, from, to)’ to ensure the stack unwinds correctly.

Steps to fix the errors:

- Remove the increment ‘(topN++)’ and decrement ‘(inter--, from+1, to+1)’ from the recursive call. The arguments should remain unchanged.
- Add the missing semicolon after the second ‘doTowers’ recursive call.

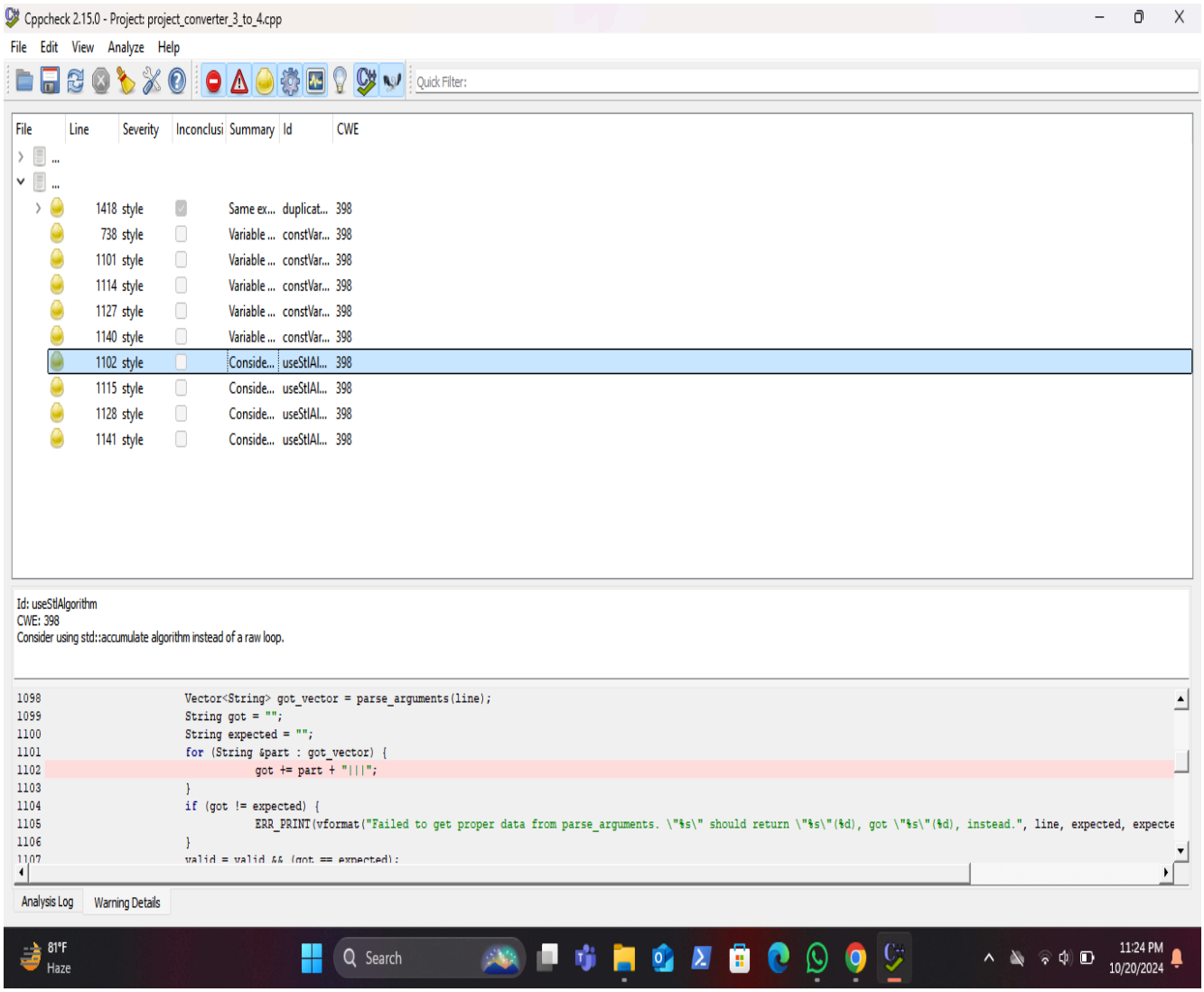
Here is the corrected code fragment:

```
public class MainClass {
   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);  // Corrected recursive call
      }
   }
}
```

Static Analysis:

Static analysis for the 2000LOC code is given below.


Cppcheck 2.15.0 - Project: project_converter_3_to_4.cpp
File Edit View Analyze Help
Quick Filter:
File Line Severity Inconclusi Summary Id CWE
... 1418 style Same ex... duplicat... 398
738 style Variable ... constVar... 398
1101 style Variable ... constVar... 398
1114 style Variable ... constVar... 398
1127 style Variable ... constVar... 398
1140 style Variable ... constVar... 398
1102 style Conside... useStlAl... 398
1115 style Conside... useStlAl... 398
1128 style Conside... useStlAl... 398
1141 style Conside... useStlAl... 398
Id: useStlAlgorithm
CWE: 398
Consider using std::accumulate algorithm instead of a raw loop.
1098 Vector<String> got_vector = parse_arguments(line);
1099 String got = "";
1100 String expected = "";
1101 for (String &part : got_vector) {
1102 got += part + "|||";
1103 }
1104 if (got != expected) {
1105 ERR_PRINT(vformat("Failed to get proper data from parse_arguments. \"%s\" should return \"%s\"(%d), got \"%s\"(%d), instead.", line, expected, expecte
1106 }
1107 valid = valid && (got == expected);
Analysis Log
Warning Details
81°F Haze
Search
11:24 PM 10/20/2024