IT314 Lab-7
Program Inspection, Debugging and Static Analysis

Kevin Shingala
202201465

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/animation_bezier_editor.cpp

Category A: Data Reference Errors

1. There are several instances where variables are used without clear initialization, such as moving_selection_offset, moving_handle_left, and moving_handle_right. It's important to ensure these are properly initialized before use.
2. Array bounds checking is not explicitly visible in the provided code snippet.
3. No obvious issues with non-integer array subscripts.
4. Potential dangling reference issues could occur with the animation and timeline pointers if they're not properly managed.
5. Not applicable to this code snippet.
6. The code assumes certain types for variables like animation and timeline. Ensure these match the expected types.
7. Not applicable to this code snippet.
8. The code uses pointers like animation.ptr(). Ensure the referenced memory has the expected attributes.
9. Not visible in this snippet.
10. Potential for off-by-one errors in loops, e.g., for (int i = 0; i < edit_points.size(); ++i).
11. Not applicable to this snippet.

Category B: Data-Declaration Errors

1. Most variables seem to be explicitly declared, but it's worth double-checking all usages.
2. Default attributes are not explicitly visible in this snippet.
3. Initialization of complex data structures (like edit_points) should be carefully reviewed.
4. Variable types and lengths should be verified, especially for critical data like animation and timeline.
5. Not explicitly visible in this snippet.
6. Similar variable names like moving_handle_left and moving_handle_right could potentially lead to confusion.

Category C: Computation Errors

1. No obvious inconsistent data type computations.
2. Mixed-mode computations are not immediately apparent but should be checked in arithmetic operations.
3. Not immediately visible in this snippet.
4. Assignment type mismatches should be checked, especially in operations involving timeline_v_zoom and timeline_v_scroll.
5. Potential for overflow in loop counters and arithmetic operations should be considered.
6. Division operations should be checked for potential zero divisors.
7. Not immediately apparent in this snippet.
8. Range checking for variables like timeline_v_scroll should be verified.
9. Complex expressions, especially in the _draw_track method, should be carefully reviewed for operator precedence.
10. Integer arithmetic in loop counters and index calculations should be verified for correctness.

Category D: Comparison Errors

1. No obvious comparisons between different data types.
2. Mixed-mode comparisons are not immediately apparent but should be checked.
3. Comparison operators seem to be used correctly, but should be double-checked, especially in complex conditions.
4. Boolean expressions, particularly in if statements, should be reviewed for correctness.
5. No obvious mixing of comparison and Boolean operators, but complex conditions should be carefully reviewed.
6. Floating-point comparisons, if present, should be checked for potential issues with precision.
7. Complex Boolean expressions should be reviewed for correct order of evaluation.
8. Not immediately apparent in this snippet.

Category E: Control-Flow Errors

1. Not applicable to this snippet.
2. Loop termination conditions, especially in methods like _draw_track, should be verified.
3. Overall termination of methods should be ensured.
4. Loop entry conditions should be checked to ensure loops execute as expected.
5. While loops with complex conditions should be reviewed for potential infinite loops.
6. Potential for off-by-one errors in loops should be carefully checked.
7. Bracket matching seems correct but should be verified, especially in nested structures.
8. Switch statements or equivalent constructs should be checked for exhaustive handling of cases.

Category F: Interface Errors
1-9. These points require a more comprehensive view of the entire codebase and how this class interacts with others. They should be carefully reviewed in the context of the full application.

Category G: Input/Output Errors
1-8. These points are not directly applicable to the given snippet, which doesn't show explicit file operations. However, if this class interacts with files or I/O streams elsewhere, these points should be carefully considered.

Category H: Other Checks
1-5. These checks require access to compiler output and a more comprehensive view of the entire program. They should be performed as part of a broader code review process.

1)How many errors are there in the program? Mention the errors you have identified.
Ans: It's difficult to give an exact number of errors without a full context and being able to run the code, but here are some potential issues identified:

- Possible uninitialized variables: moving_selection_offset, moving_handle_left, moving_handle_right
- Potential dangling references with animation and timeline pointers
- Possible off-by-one errors in loops, e.g., for (int i = 0; i < edit_points.size(); ++i)
- Potential for overflow in loop counters and arithmetic operations
- Possible issues with floating-point comparisons
- Potential for infinite loops in complex while conditions

2)Which category of program inspection would you find more effective?
Ans:

- Category A: Data Reference Errors
- Category C: Computation Errors
- Category E: Control-Flow Errors

3)Which type of error you are not able to identified using the program inspection?
Ans:

- Runtime errors that depend on specific input data
- Performance issues
- Concurrency-related bugs
- Some types of logical errors that don't manifest as obvious code issues
- Errors related to the broader system architecture or design

4) Is the program inspection technique worth applicable?

Ans: Yes, the program inspection technique is definitely worth applying. It helps identify many potential issues before runtime and can catch errors that might not be immediately apparent during testing. However, it should be used in conjunction with other techniques like:

- Static code analysis tools
- Unit testing
- Integration testing
- Code reviews by other developers

CODE DEBUGGING

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

1. Armstrong

The program contains the following issues:

1. The expression remainder = num / 10 is incorrect because it divides the number instead of extracting the last digit. The correct expression should be remainder = num % 10.
2. The line num = num % 10 is also incorrect, as it uses the wrong operation to reduce the number. Instead of finding the remainder, it should be num = num / 10 to remove the last digit.

To address these errors:

- Place a breakpoint at remainder = num / 10 to observe the incorrect remainder calculation.
- Place another breakpoint at num = num % 10 to see how the number is being improperly reduced.

Fixing the errors involves:

- Correcting the calculation for the last digit by changing remainder = num / 10 to remainder = num % 10.
- Fixing the reduction of the number 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

The program contains the following errors:

1. In the gcd method, the issue is with the condition in the while loop. The correct condition to compute the GCD should be while (a % b != 0), but it is currently incorrect.
2. In the lcm method, although there are no syntax errors, the algorithm is inefficient, relying on a brute-force approach to compute the LCM.

To fix these errors:

- Place a breakpoint in the gcd method to observe the issue in the loop and modify the condition to while (a % b != 0) for proper GCD computation.
- Consider optimizing the lcm method to avoid the brute-force approach for better performance.

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();
}
}
```

3. Knapsack

The program has the following errors:

- Incorrect use of increment (++) and decrement (--) operators, which may cause unexpected behavior.

The number of breakpoints will depend on the complexity of the code. Typically, breakpoints should be placed at critical points where these operators are used, allowing you to observe their impact on the variables and identify where the logic goes wrong.

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 the following two errors:

1. The condition in the inner while loop is incorrect. It should be while (sum != 0) instead of while (sum == 0).
2. The variable s is incorrectly initialized as s = 0 before the inner while loop. It should be initialized as s = 1.

To fix these issues:

- Place a breakpoint at the inner while loop to observe and correct the condition from while (sum == 0) to while (sum != 0).
- Place another breakpoint before the inner while loop to ensure the variable s is initialized as s = 1.

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

The program has the following errors:

1. Incorrect array passing: The functions leftHalf(array+1) and rightHalf(array-1) should pass the entire array rather than modifying the index. They need to be updated to leftHalf(array) and rightHalf(array).
2. Incorrect merging logic: The call merge(array, left++, right--) incorrectly increments and decrements the indices. It should be merge(array, left, right).

To address these issues:

- Place a breakpoint at leftHalf(array+1) to observe the array passing error.
- Place another breakpoint at rightHalf(array-1) for the same issue.
- Place a third breakpoint at merge(array, left++, right--) to examine the merging logic.

Steps to fix:

- Change leftHalf(array+1) and rightHalf(array-1) to leftHalf(array) and rightHalf(array) to pass the entire array.
- Update merge(array, left++, right--) to merge(array, left, right) to correct the merging logic without modifying the indices during the call.

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

The program contains two key errors:

1. 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 be sum = sum + first[c][k] * second[k][d]; to properly reference the matrices.
2. Logical error in row and column prompts: The program incorrectly repeats the prompt for the dimensions of the first matrix when entering the dimensions for the second matrix. The second matrix dimensions need to be prompted separately.

To fix these errors:

- Place a breakpoint at the multiplication line to verify the correctness of the indices during matrix multiplication.
- Place another breakpoint before the second matrix input to ensure that the matrix dimensions are entered correctly.
- A third breakpoint should be set after the matrix multiplication loop to inspect the final product before printing.

Steps to resolve the errors:

1. Correct the multiplication logic by updating the indices: sum = sum + first[c][k] * second[k][d];.
2. Modify the prompt for the second matrix to correctly ask: "Enter the number of rows and columns of the second matrix".
3. Ensure the output is formatted clearly, using tabs or spaces to separate the printed matrix values.

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

The program contains several errors:

1. Syntax Errors:
    - In the insert method, the line i + = (i + h / h--) % maxSize; contains an incorrect space. It should be corrected to i += (i + h) % maxSize;.
    - In the get method, the line i = (i + h * h++) % maxSize; is incorrect. The increment of h happens too early, and it should be i = (i + h * h) % maxSize; to ensure correct calculations.
2. Incorrect Rehashing Logic:
    - In the remove method, the loop while (!key.equals(keys[i])) does not account for cases where keys[i] is null. If keys[i] is null, the key does not exist, and the loop should break.
3. Logical Flow:
    - The program does not properly handle the input of multiple key-value pairs. After the "Enter key and value" prompt, it does not provide a clear structure for accepting and processing multiple entries.

Breakpoints to fix the issues:

1. In the insert method to check the logic of handling key-value pairs.
2. In the get method to verify the correctness of the hash table lookup logic.
3. In the remove method to ensure proper key deletion and rehashing.

Steps to resolve the errors:

1. Correct the syntax:
    - Change i + = (i + h / h--) % maxSize; to i += (i + h) % maxSize; in the insert method.
    - Change i = (i + h * h++) % maxSize; to i = (i + h * h) % maxSize; in the get method.
2. Improve rehashing logic:
    - In the remove method, modify the loop to handle null values properly: while (keys[i] != null && !key.equals(keys[i])).
3. Fix logical flow:
    - Modify the user input handling to clearly allow multiple key-value pairs by structuring the input prompts or using a loop that guides the user through multiple entries.

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:

1. Incorrect Loop Condition in the Outer Loop: The line for (int i = 0; i >= n; i++); has two issues:
    - The condition should be i < n instead of i >= n for proper iteration.
    - The semicolon at the end of the loop terminates the loop prematurely and must be removed.
2. Incorrect Sorting Logic: The condition in the if statement within the nested loop is incorrect for ascending order. It should be if (a[i] > a[j]) to ensure that the array is sorted properly.

Breakpoints to debug the program:

1. Set a breakpoint at the end of the outer loop, just before the sorting logic, to verify that the loop iterates correctly.
2. Set a second breakpoint after the sorting logic to check if the array is correctly sorted before printing.

Steps to fix the errors:

1. Fix the Outer Loop Condition: Change for (int i = 0; i >= n; i++); to for (int i = 0; i < n; i++), and remove the semicolon at the end of the loop.
2. Correct the Sorting Logic: Update the condition from if (a[i] <= a[j]) to if (a[i] > a[j]) to ensure proper sorting 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 contains the following errors:

1. Incorrect Increment/Decrement in Recursive Calls:
    - The lines topN++, inter--, from+1, and to+1 are both syntactically incorrect and logically inappropriate in the context of recursive function calls. Modifying the arguments directly during the call will alter the values unintentionally, leading to incorrect behavior in the recursion.
2. Missing Semicolon:
    - There is a missing semicolon after the recursive call doTowers(topN++, inter--, from+1, to+1), which leads to a syntax error.

Breakpoints to debug the program:

1. Set a breakpoint before the recursive call doTowers(topN - 1, from, to, inter) to check the recursive behavior and ensure the function operates as expected.

2. Place another breakpoint before the second recursive call doTowers(topN - 1, inter, from, to) to observe how the stack unwinds and verify the logic.

Steps to fix the errors:

1. Remove incorrect increments/decrements:
    - Remove the topN++, inter--, from+1, and to+1 in the recursive call. The arguments should remain unchanged, so the recursive call should be doTowers(topN, inter, from, to) without modifying the values.
2. Add the missing semicolon:
    - Add the missing semicolon after the second doTowers recursive call to ensure correct syntax.

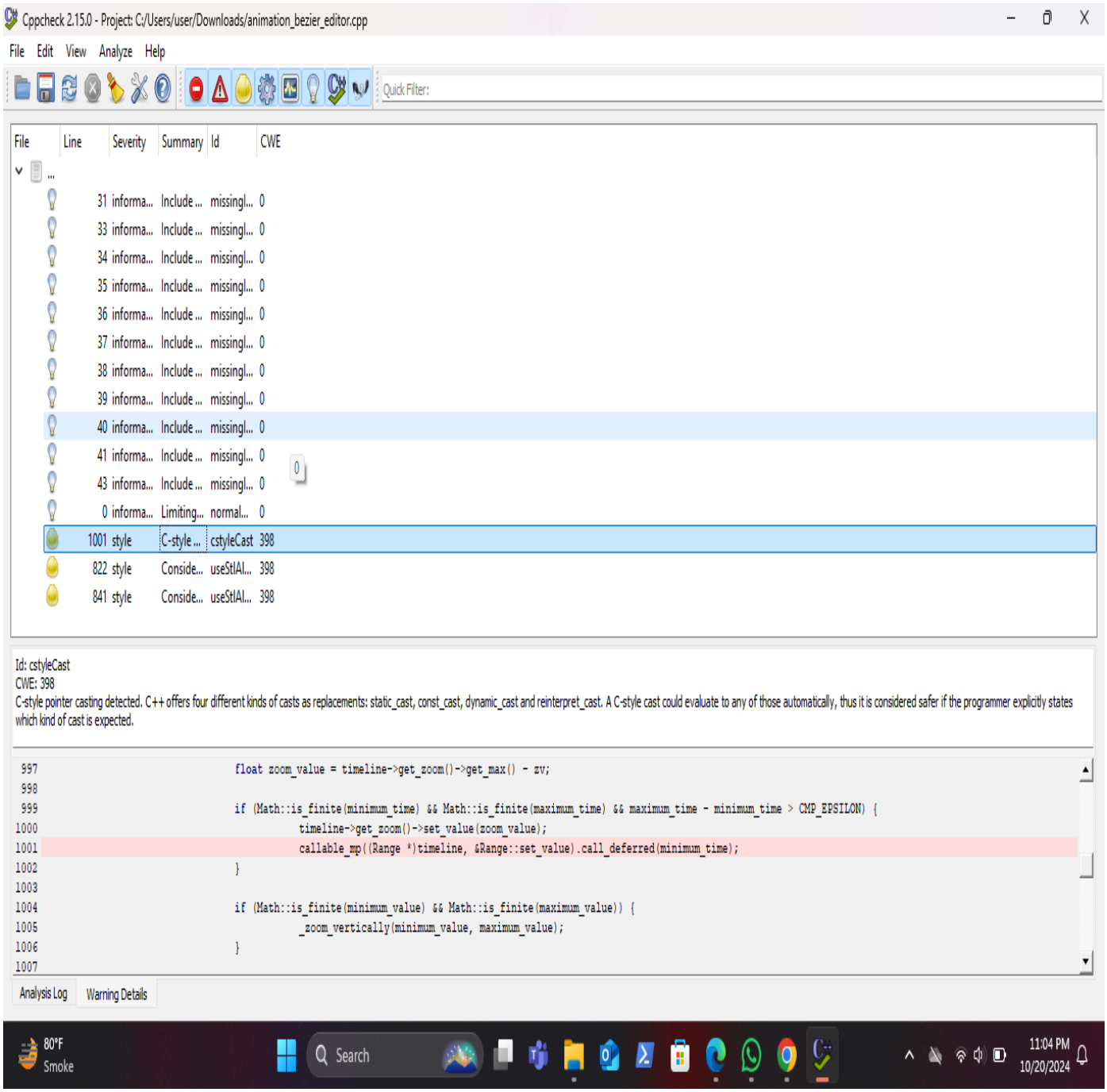
Here is the corrected code fragment:

```
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:

Cppcheck is used as a static analysis tool. Here is the attached screenshot for static analysis of the code.


Cppcheck 2.15.0 - Project: C:/Users/user/Downloads/animation_bezier_editor.cpp
File Edit View Analyze Help
Quick Filter:
File Line Severity Summary Id CWE
... 31 informa... Include ... missingI... 0
33 informa... Include ... missingI... 0
34 informa... Include ... missingI... 0
35 informa... Include ... missingI... 0
36 informa... Include ... missingI... 0
37 informa... Include ... missingI... 0
38 informa... Include ... missingI... 0
39 informa... Include ... missingI... 0
40 informa... Include ... missingI... 0
41 informa... Include ... missingI... 0
43 informa... Include ... missingI... 0
0 informa... Limiting... normal... 0
1001 style C-style ... cstyleCast 398
822 style Conside... useStlAl... 398
841 style Conside... useStlAl... 398
Id: cstyleCast
CWE: 398
C-style pointer casting detected. C++ offers four different kinds of casts as replacements: static_cast, const_cast, dynamic_cast and reinterpret_cast. A C-style cast could evaluate to any of those automatically, thus it is considered safer if the programmer explicitly states which kind of cast is expected.
997 float zoom_value = timeline->get_zoom()->get_max() - zv;
998
999 if (Math::is_finite(minimum_time) && Math::is_finite(maximum_time) && maximum_time - minimum_time > CMP_EPSILON) {
1000 timeline->get_zoom()->set_value(zoom_value);
1001 callable_mp((Range *)timeline, &Range::set_value).call_deferred(minimum_time);
1002 }
1003
1004 if (Math::is_finite(minimum_value) && Math::is_finite(maximum_value)) {
1005 _zoom_vertically(minimum_value, maximum_value);
1006 }
1007
Analysis Log
Warning Details
80°F
Smoke
Search
11:04 PM
10/20/2024