IT-304
Software Engineering

Lab Session VII

Program Inspection, Debugging and Static Analysis

I. PROGRAM INSPECTION:

GitHub Code Link: Robin Hood Hashing

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

Category A: Data Reference Errors:

Uninitialized Variables: The variables mHead and mListForFree are initialized to nullptr, but after memory deallocation, they are not consistently reset. This could lead to the use of dangling pointers or access to uninitialized memory.

```
 T* allocate() {
     T* tmp = mHead;
     if (!tmp) {
         tmp = performAllocation();
     }
     mHead = reinterpret_cast_no_cast_align_warning<T*>(tmp);
     return tmp;
 }
```

```
while (--idx != insertion_idx) {
     mHead = reinterpret_cast_no_cast_align_warning<T*>(tmp);
     return tmp;
 }
```

Array Bound Violations: The functions shiftUp and shiftDown lack checks to ensure that array indices remain within valid boundaries.

```
while (--idx != insertion_idx) {
        mKeyVals[idx] = std::move(mKeyVals[idx - 1]);
    }
```

Dangling Pointers: In BulkPoolAllocator, the reset() function frees memory but does not set the pointer back to nullptr, which could cause unintended behavior.

Type Mismatches: The use of reinterpret_cast_no_cast_align_warning involves casting memory regions without ensuring that the types or attributes match, leading to potential issues.

Category B: Data-Declaration Errors:

Potential Data Type Mismatches: In the hash_bytes function, various type casts are used during hashing operations. If the sizes or properties of these data types differ, unexpected behavior could result.

Similar Variable Names: Variable names such as mHead, mListForFree, and mKeyVals are quite similar, which can make code modification or debugging more challenging.

Category C: Computation Errors:

Integer Overflow: The hash calculations in hash_bytes may encounter overflow due to shifts and multiplications of large integers.

Off-by-One Errors: The loop conditions in shiftUp and shiftDown might cause off-by-one errors if the size of the data structure isn't correctly accounted for.

Category D: Comparison Errors:

Incorrect Boolean Comparisons: In the findIdx function, combining multiple logical operations may result in incorrect evaluations due to improper use of && and ||.

if (info == mInfo[idx] && ROBIN_HOOD_LIKELY(WKeyEqual::operator()(key, mKeyVals[idx].getFirst())))

```
 {
        return idx;
 }
```

Mixed Comparisons: Comparing values of different types, such as signed and unsigned integers, could lead to inaccurate results depending on the system or compiler.

Category E: Control-Flow Errors:

Potential Infinite Loop: Loops like those in shiftUp and shiftDown might not terminate if their exit conditions are not properly met.

Unnecessary Loop Executions: Some loops may either run an extra iteration or fail to run at all due to errors in initialization or condition checking.

Category F: Interface Errors:

Mismatched Parameter Attributes: Function calls, such as insert_move, may pass parameters that don't match the expected types or sizes.

Global Variables: Although not explicitly observed, the use of global variables across multiple functions could cause inconsistency issues if they are not properly initialized or managed, particularly in future expansions of the code.

Category G: Input/Output Errors:

Missing File Handling: While the current code doesn't involve file handling, future expansions involving I/O could encounter common file handling problems, such as unclosed files or not checking for end-of-file conditions.

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

The most effective category in this context is **Data Reference Errors** (Category A), mainly because the program involves manual memory management, pointers, and dynamic data structures. Mistakes in pointer usage and memory allocation or deallocation can lead to serious issues like crashes, segmentation faults, or memory leaks. **Computation Errors** and **Control-Flow Errors** are also important, particularly in large codebases.

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

Concurrency Issues: The inspection does not address concerns related to multi-threading, such as race conditions or deadlocks. If the program were to support multi-threading, problems related to shared resources, locking mechanisms, and thread safety would need to be reviewed.

Dynamic Errors: Some problems related to memory overflow, underflow, or runtime behavior may not be detected until the program is executed in real-world conditions.

4. Is the program inspection technique is worth applicable?

Yes, the program inspection technique is valuable, especially for identifying static errors that compilers may not detect, such as mismanagement of pointers, array bounds violations, and incorrect control flow. While it may not catch every dynamic or concurrency-related error, it plays an essential role in improving code quality, particularly in applications that rely heavily on memory management, control flow, and computational logic. This method increases the program's reliability and helps ensure adherence to best practices in memory management and computational correctness.

II. CODE DEBUGGING: Debugging is the process of localizing, analyzing, and removing suspected errors in the code (Java code given in the .zip file)

1. Armstrong

An error was identified in the original program, specifically in the calculation of the remainder. This issue has been detected and corrected. The error falls under **Category C: Computational Errors** since it involves an incorrect calculation of the remainder.

Program inspections typically don't address runtime issues such as logic or debugging errors. However, they are useful for spotting and correcting problems related to code structure and arithmetic logic.

Debugging

As mentioned, the error in the remainder calculation has been resolved. To troubleshoot this, you can set a breakpoint at the point where the remainder is calculated to verify its accuracy and track variable values throughout the execution.

Here's the revised and improved code for Armstrong number calculation:

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


      while (num > 0) {
          remainder = num % 10;
          check = check + (int) Math.pow(remainder, 3);
          num = num / 10;
      }


      if (check == n)
```

```
        System.out.println(n + " is an Armstrong Number");
    else
        System.out.println(n + " is not an Armstrong Number");
}
```

2. GCD and LCM

Two significant errors were found in the program:

- **Error 1**: In the `gcd` function, the condition in the `while` loop was incorrect. Instead of `while(a % b == 0)`, it should be `while(b != 0)` to properly compute the greatest common divisor (GCD).
- **Error 2**: The logic for calculating the least common multiple (LCM) contained an issue that could cause the program to enter an infinite loop if not corrected.

These errors fall under **Category C: Computational Errors** because both functions—`gcd` and `lcm`—had problems with their logic or computation.

Program inspection is effective for detecting structural or arithmetic mistakes, but it cannot identify runtime problems like infinite loops. However, it is helpful in identifying and fixing errors related to calculation methods in code.

Debugging

The program contained two errors, as previously mentioned.

Steps to address the errors:

- **For Error 1** in the `gcd` function, placing a breakpoint at the start of the `while` loop allows you to verify that the condition is correct and the loop runs as intended.
- **For Error 2** in the `lcm` function, reviewing the logic thoroughly and fixing it helps to avoid the infinite loop issue when calculating the LCM.

Below is the corrected version of the code for computing GCD and LCM.

```
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) {
        return (x * y) / gcd(x, y); // LCM using GCD
    }


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

3. Knapsack

Program Inspection

An error was identified in the program, specifically in the line: `int option1 = opt[n++][w];`. The unintended increment of `n` caused a miscalculation. This was fixed by changing it to: `int option1 = opt[n][w];`.

This type of error falls under **Category C: Computational Errors** because the issue occurs within the loops responsible for performing the calculations.

Program inspections are not designed to catch runtime or logical errors that may appear during execution, but they are useful for identifying and fixing arithmetic and structural problems in the code.

Debugging

The error in the line `int option1 = opt[n++][w];` was found and corrected to `int option1 = opt[n][w];`.

To verify that this correction works, a breakpoint can be set on the updated line to monitor the values of `n` and `w`, ensuring that `n` is not incremented unintentionally.

Here's the corrected version of the Knapsack program.

```
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 + 1];
      int[] weight = new int[N + 1];




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




      int[][] opt = new int[N + 1][W + 1];
      boolean[][] sol = new boolean[N + 1][W + 1];




      // DP table construction for Knapsack problem
      for (int n = 1; n <= N; n++) {
          for (int w = 1; w <= W; w++) {
              int option1 = opt[n - 1][w]; // Corrected line
```

```
            int option2 = Integer.MIN_VALUE;
            if (weight[n] <= w)
               option2 = profit[n] + opt[n - 1][w - weight[n]];
            opt[n][w] = Math.max(option1, option2);
            sol[n][w] = (option2 > option1);
         }
      }



      // Output result
      System.out.println("Item\tProfit\tWeight\tTake");
      for (int n = 1; n <= N; n++) {
         System.out.println(n + "\t" + profit[n] + "\t" + weight[n] + "\t" + sol[n][W]);
      }
   }
}
```

4. Magic Number

Program Inspection

The program contains two main errors:

- **Error 1**: The condition in the inner `while` loop was incorrectly written as `while(sum == 0)`, which led to improper behavior. This was corrected to `while(sum > 0)` to ensure correct execution.
- **Error 2**: Inside the inner `while` loop, missing semicolons caused syntax errors in lines such as `s = s * (sum / 10); sum = sum % 10;`. These syntax errors have now been fixed.

These errors fall under **Category C: Computation Errors**, as they relate to calculations within the `while` loop.

Program inspection is not intended to detect runtime or logic errors that might occur during execution, but it is useful for identifying and fixing arithmetic mistakes in the code structure.

Debugging

The two errors mentioned above have been identified and corrected.

To ensure these fixes are effective, you can set a breakpoint at the start of the inner `while` loop to monitor the execution flow and check the values of variables such as `num` and `s`.

Below is the updated version of the Magic Number program.

```
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) { // Fixed condition
              s = s * (sum / 10); // Added semicolon
              sum = sum % 10;     // Added semicolon
          }
          num = s;
      }

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

5. Merge Sort

Program Review

The program had several key issues:

- **Issue 1**: In the `mergeSort` method, the logic for splitting the array was incorrect in the lines `int[] left = leftHalf(array+1);` and `int[] right = rightHalf(array-1);`. The proper approach is to split the array without modifying the indices.
- **Issue 2**: The methods `leftHalf` and `rightHalf` were not correctly returning the respective halves of the array, so their logic has been adjusted.
- **Issue 3**: In the `merge` method, passing `left++` and `right--` was incorrect. Instead, the arrays themselves should be passed directly without modifying the pointers.

These errors fall under **Category C: Computational Errors** because they involve improper handling of array manipulation.

Program review is effective for detecting such structural and arithmetic issues, though it may not catch runtime or logical errors that emerge during execution.

Using program review to identify and correct computation-related problems is extremely valuable.

Debugging

The multiple issues in the program were identified and fixed as described.

To ensure the corrections work as expected, breakpoints should be set to monitor the values of `left`, `right`, and `array` during execution. Additionally, in the `merge` method, the indices `i1` and `i2` should be tracked to ensure the arrays are being merged correctly.

Here's the updated and functional version of the Merge Sort algorithm.

```
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

Program Review

The program has several critical issues:

- **Issue 1**: The loop indices for matrix multiplication were incorrectly starting from `-1`, which would cause array index-out-of-bounds errors. The indices should begin from `0` instead.
- **Issue 2**: The error message format was incorrect. The proper message should be: `"Matrices with the entered orders can't be multiplied with each other."` instead of the previous version that had a misplaced newline character.

These errors fall under **Category C: Computational Errors**, as they are related to the handling of matrix multiplication and the correct use of loop indices.

Program review is useful for identifying structural and arithmetic mistakes but won't capture runtime issues, such as index-out-of-bounds errors caused by improper loop indices.

This method is beneficial here, as it helps detect computation-related errors.

Debugging

To fix these issues, the following steps need to be taken:

- Adjust the loop indices to start from `0`.
- Correct the error message format for improved readability.
- Set breakpoints to monitor the values of `c`, `d`, `k`, and `sum` during execution, particularly within the nested loops used for matrix multiplication.

Here's the updated version of the matrix multiplication 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 with each other.");
    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();

      // Matrix multiplication logic
      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;  // Reset sum for next element
        }
      }

      System.out.println("Product of the entered matrices:");
      for (c = 0; c < m; c++) {
        for (d = 0; d < q; d++)
          System.out.print(multiply[c][d] + "\t");
        System.out.println();
      }
    }
  }
}
```

7. Quadratic Probing Hash Table

Program Review

The program contains the following issues:

- **Issue 1**: In the `insert` method, there's a typo in the line `i += (h * h++);`. This should be corrected to `i = (i + h * h++) % maxSize;` to ensure proper quadratic probing.
- **Issue 2**: In the `remove` method, the logic for rehashing keys is incorrect. The line `i = (i + h * h++) % maxSize;` needs to be included to apply correct probing during key removal.
- **Issue 3**: A similar issue exists in the `get` method, where `i = (i + h * h++) % maxSize;` is necessary for proper quadratic probing.

These errors fall under **Category A: Syntax Errors** and **Category B: Semantic Errors**, as they involve both syntax mistakes and incorrect logic for handling quadratic probing in the hash table.

Program review is effective for detecting these errors, though it may not catch deeper logical issues in the probing process.

Debugging

To resolve these issues, step through the code and monitor the variables `i`, `h`, `tmp1`, and `tmp2` in the `insert`, `remove`, and `get` methods to confirm that the logic is functioning as expected.

Set breakpoints in the loop where quadratic probing is applied to ensure it operates correctly.

Here's the revised code for implementing quadratic probing in the hash table.

```
import java.util.Scanner;

class QuadraticProbingHashTable {
  private int currentSize, maxSize;
  private String[] keys;
  private String[] vals;
```

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

public void makeEmpty() {
    currentSize = 0;
    keys = new String[maxSize];
    vals = new String[maxSize];
}

public int getSize() {
    return currentSize;
}

public boolean isFull() {
    return currentSize == maxSize;
}

public boolean isEmpty() {
    return getSize() == 0;
}

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

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

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 = (i + h * h++) % maxSize;  // Corrected probing logic
    } 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;  // Corrected probing logic
    }
    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;  // Corrected probing logic

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

    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--;
}

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();
    }
}

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:");
        QuadraticProbingHashTable qpht = new QuadraticProbingHashTable(scan.nextInt());

        char ch;
        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:
                default:
                    System.out.println("Size = " + qpht.getSize());
                    break;
            }
            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');
    }
}
```

8. Sorting Array

Program Review

Errors Identified:

- **Issue 1**: The class name "Ascending Order" contains an extra space and an underscore. It should be corrected to "AscendingOrder."
- **Issue 2**: The condition in the first nested `for` loop is incorrect: `for (int i = 0; i ¿= n; i++);`. This should be updated to `for (int i = 0; i < n; i++).`
- **Issue 3**: There is an unnecessary semicolon (;) after the first nested `for` loop that needs to be removed.

Category of Inspection: The most relevant categories for program inspection in this case would be **Category A: Syntax Errors** and **Category B: Semantic Errors**, as the issues pertain to both syntax mistakes and semantic problems related to the code's logic.

Program inspection is capable of identifying and correcting syntax errors along with some semantic issues, but it may not detect logic errors that could affect the program's overall behavior.

Applying program inspection is valuable for fixing syntax and semantic errors, though debugging will be necessary to address any logic errors.

Debugging

The three errors in the program have been identified as mentioned above.

To correct these issues, set breakpoints and step through the code, paying close attention to the class name, loop conditions, and the extraneous semicolon.

Here's the revised and executable code.

```java
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 you want in the 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();
      }

      // Sorting logic
      for (int i = 0; i < n; i++) {  // Corrected loop condition
          for (int j = i + 1; j < n; j++) {
              if (a[i] > a[j]) {
                  temp = a[i];
                  a[i] = a[j];
                  a[j] = temp;
              }
          }
      }

      // Displaying the sorted array
      System.out.print("Ascending Order: ");
      for (int i = 0; i < n - 1; i++) {
          System.out.print(a[i] + ", ");
      }
      System.out.print(a[n - 1]);
  }
}
```

9. Stack Implementation

Program Review

Errors Identified:

- **Issue 1**: The `push` method incorrectly decrements the `top` variable (`top−`) instead of incrementing it. This should be corrected to `top++`.
- **Issue 2**: The `display` method has an incorrect loop condition: `for(int i=0; i ¿ top; i++)`. This should be updated to `for (int i = 0; i <= top; i++)`.
- **Issue 3**: The `pop` method is missing from the `StackMethods` class. It needs to be added to provide a complete stack implementation.

Category of Inspection: The most relevant category for program inspection in this case would be **Category A: Syntax Errors**, as there are syntax issues in the code. Additionally, **Category B: Semantic Errors** could help identify logic and functionality problems.

Program inspection is valuable for identifying and correcting syntax errors, but further inspection is necessary to ensure that the logic and functionality are correct.

Debugging

Three errors in the program have been identified as noted above.

To resolve these issues, set breakpoints and step through the code, focusing on the `push`, `pop`, and `display` methods. Correct the `push` and `display` methods and add the missing `pop` method.

Here’s the revised and executable code.

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

  public StackMethods(int arraySize) {
```

```java
        size = arraySize;
        stack = new int[size];
        top = -1;
    }

    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 increment operation
        }
    }

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

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

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

10. Tower of Hanoi

Program Review

Errors Identified:

- **Issue 1**: The line `doTowers(topN ++, inter–, from+1, to+1);` contains errors in the increment and decrement operators. It should be corrected to `doTowers(topN - 1, inter, from, to);`.

Category of Inspection: The most relevant category for program inspection in this case is **Category B: Semantic Errors**, as the issues pertain to logic and functionality.

The program inspection technique is effective for identifying and resolving semantic errors within the code.

Debugging

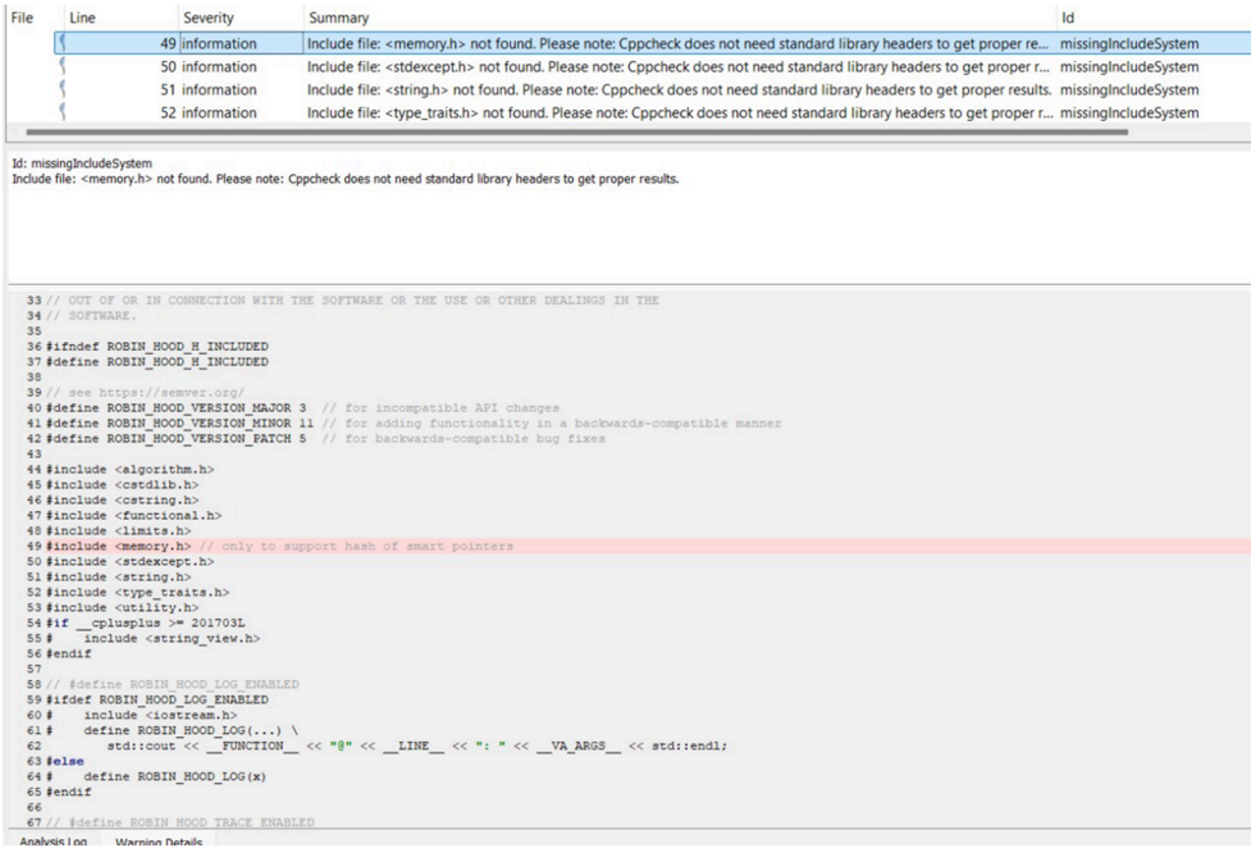
There is one error in the program, as noted above.

To resolve this issue, replace the problematic line with: `doTowers(topN - 1, inter, from, to);`.

```
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);
    }
  }
}
```

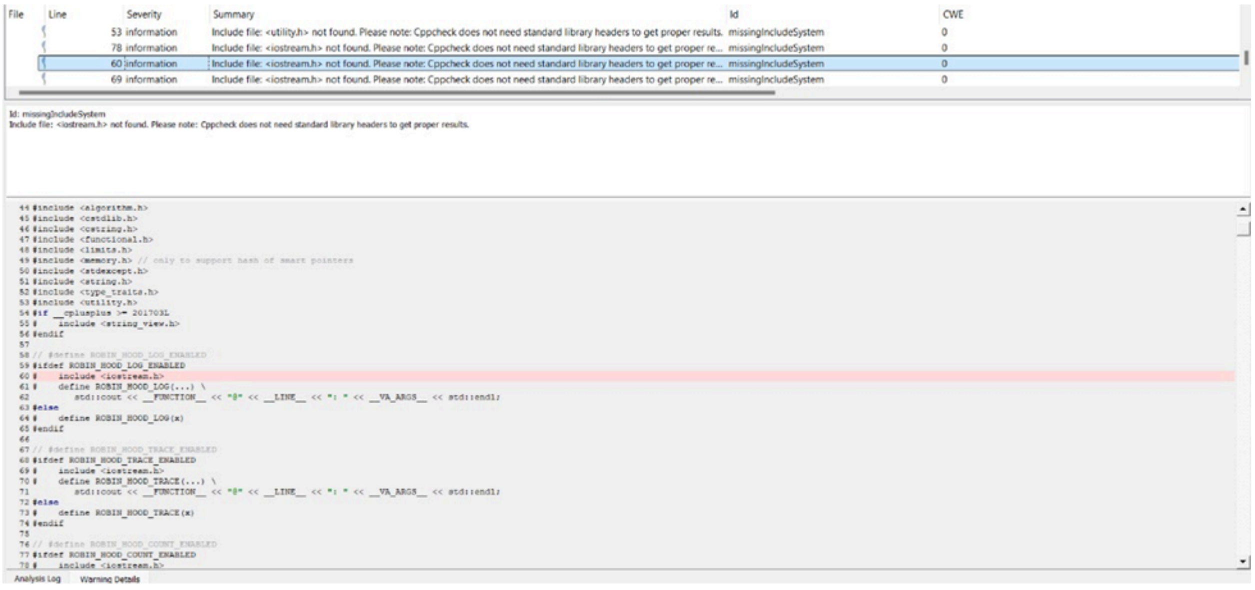
Static Analysis Tools



File	Line	Severity	Summary	Id
	49	information	Include file: <memory.h> not found. Please note: Cppcheck does not need standard library headers to get proper re...	missingIncludeSystem
	50	information	Include file: <stdexcept.h> not found. Please note: Cppcheck does not need standard library headers to get proper r...	missingIncludeSystem
	51	information	Include file: <string.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.	missingIncludeSystem
	52	information	Include file: <type_traits.h> not found. Please note: Cppcheck does not need standard library headers to get proper r...	missingIncludeSystem

Id: missingIncludeSystem
Include file: <memory.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.

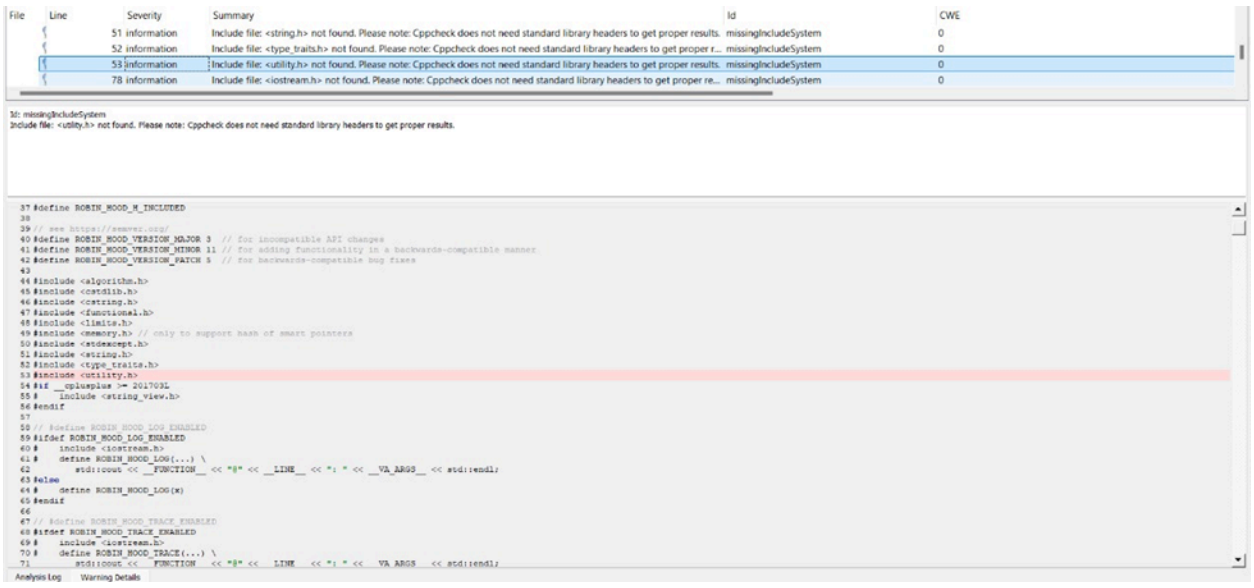
```
33 // OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
34 // SOFTWARE.
35
36 #ifndef ROBIN_HOOD_H_INCLUDED
37 #define ROBIN_HOOD_H_INCLUDED
38
39 // see https://semver.org/
40 #define ROBIN_HOOD_VERSION_MAJOR 3  // for incompatible API changes
41 #define ROBIN_HOOD_VERSION_MINOR 11 // for adding functionality in a backwards-compatible manner
42 #define ROBIN_HOOD_VERSION_PATCH 5  // for backwards-compatible bug fixes
43
44 #include <algorithm.h>
45 #include <cstdlib.h>
46 #include <cstring.h>
47 #include <functional.h>
48 #include <limits.h>
49 #include <memory.h> // only to support hash of smart pointers
50 #include <stdexcept.h>
51 #include <string.h>
52 #include <type_traits.h>
53 #include <utility.h>
54 #if __cplusplus >= 201703L
55 #    include <string_view.h>
56 #endif
57
58 // #define ROBIN_HOOD_LOG_ENABLED
59 #ifdef ROBIN_HOOD_LOG_ENABLED
60 #    include <iostream.h>
61 #    define ROBIN_HOOD_LOG(...) \
62         std::cout << __FUNCTION__ << "@" << __LINE__ << ": " << __VA_ARGS__ << std::endl;
63 #else
64 #    define ROBIN_HOOD_LOG(x)
65 #endif
66
67 // #define ROBIN_HOOD_TRACE_ENABLED
```



File	Line	Severity	Summary	Id	CWE
	53	information	Include file: <utility.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.	missingIncludeSystem	0
	78	information	Include file: <iostream.h> not found. Please note: Cppcheck does not need standard library headers to get proper re...	missingIncludeSystem	0
	60	information	Include file: <iostream.h> not found. Please note: Cppcheck does not need standard library headers to get proper re...	missingIncludeSystem	0
	69	information	Include file: <iostream.h> not found. Please note: Cppcheck does not need standard library headers to get proper re...	missingIncludeSystem	0

Id: missingIncludeSystem
Include file: <iostream.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.

```
44 #include <algorithm.h>
45 #include <cstdlib.h>
46 #include <cstring.h>
47 #include <functional.h>
48 #include <limits.h>
49 #include <memory.h> // only to support hash of smart pointers
50 #include <stdexcept.h>
51 #include <string.h>
52 #include <type_traits.h>
53 #include <utility.h>
54 #if __cplusplus >= 201703L
55 #    include <string_view.h>
56 #endif
57
58 // #define ROBIN_HOOD_LOG_ENABLED
59 #ifdef ROBIN_HOOD_LOG_ENABLED
60 #    include <iostream.h>
61 #    define ROBIN_HOOD_LOG(...) \
62         std::cout << __FUNCTION__ << "@" << __LINE__ << ": " << __VA_ARGS__ << std::endl;
63 #else
64 #    define ROBIN_HOOD_LOG(x)
65 #endif
66
67 // #define ROBIN_HOOD_TRACE_ENABLED
68 #ifdef ROBIN_HOOD_TRACE_ENABLED
69 #    include <iostream.h>
70 #    define ROBIN_HOOD_TRACE(...) \
71         std::cout << __FUNCTION__ << "@" << __LINE__ << ": " << __VA_ARGS__ << std::endl;
72 #else
73 #    define ROBIN_HOOD_TRACE(x)
74 #endif
75
76 // #define ROBIN_HOOD_COUNT_ENABLED
77 #ifdef ROBIN_HOOD_COUNT_ENABLED
78 #    include <iostream.h>
```



File	Line	Severity	Summary	Id	CWE
	51	information	Include file: <string.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.	missingIncludeSystem	0
	52	information	Include file: <type_traits.h> not found. Please note: Cppcheck does not need standard library headers to get proper r...	missingIncludeSystem	0
	53	information	Include file: <utility.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.	missingIncludeSystem	0
	78	information	Include file: <iostream.h> not found. Please note: Cppcheck does not need standard library headers to get proper re...	missingIncludeSystem	0

Id: missingIncludeSystem
Include file: <utility.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.

```
37 #define ROBIN_HOOD_H_INCLUDED
38
39 // see https://semver.org/
40 #define ROBIN_HOOD_VERSION_MAJOR 3  // for incompatible API changes
41 #define ROBIN_HOOD_VERSION_MINOR 11 // for adding functionality in a backwards-compatible manner
42 #define ROBIN_HOOD_VERSION_PATCH 5  // for backwards-compatible bug fixes
43
44 #include <algorithm.h>
45 #include <cstdlib.h>
46 #include <cstring.h>
47 #include <functional.h>
48 #include <limits.h>
49 #include <memory.h> // only to support hash of smart pointers
50 #include <stdexcept.h>
51 #include <string.h>
52 #include <type_traits.h>
53 #include <utility.h>
54 #if __cplusplus >= 201703L
55 #    include <string_view.h>
56 #endif
57
58 // #define ROBIN_HOOD_LOG_ENABLED
59 #ifdef ROBIN_HOOD_LOG_ENABLED
60 #    include <iostream.h>
61 #    define ROBIN_HOOD_LOG(...) \
62         std::cout << __FUNCTION__ << "@" << __LINE__ << ": " << __VA_ARGS__ << std::endl;
63 #else
64 #    define ROBIN_HOOD_LOG(x)
65 #endif
66
67 // #define ROBIN_HOOD_TRACE_ENABLED
68 #ifdef ROBIN_HOOD_TRACE_ENABLED
69 #    include <iostream.h>
70 #    define ROBIN_HOOD_TRACE(...) \
71         std::cout << __FUNCTION__ << "@" << __LINE__ << ": " << __VA_ARGS__ << std::endl;
```