IT-304 : Software Engineering

Lab Assignment: 7

Program Inspection, Debugging and Static Analysis

Name: Darshak Kukadiya
Id: 202201180

Task 1: 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:

1. **Uninitialized Variables:**
   - The variables mHead and mListForFree are set to nullptr, but they aren't consistently reset after memory deallocation, which can lead to dangling pointers or uninitialized memory access.

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

2. **Array Bound Violations:**
   - The shiftUp and shiftDown functions lack checks to confirm that indices stay within valid array bounds.

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

3. **Dangling Pointers:**
   - In BulkPoolAllocator, the reset() function releases memory but does not set the pointer back to nullptr.
4. **Type Mismatches:**
   - The use of reinterpret_cast_no_cast_align_warning results in casting memory areas without verifying types or attributes, which can cause subtle issues.

Category B: Data-Declaration Errors:

1. **Potential Data Type Mismatches:**
    - The hash_bytes function involves several type casts during hashing operations. If the data types' sizes or characteristics vary, this can lead to unexpected behavior.
2. **Similar Variable Names:**
    - Variables such as mHead, mListForFree, and mKeyVals have similar names, which can create confusion when modifying or debugging the code.

Category C: Computation Errors:

1. **Integer Overflow:**
    - The hash calculations in hash_bytes may experience overflow due to multiple shifts and multiplications of large integers.
2. **Off-by-One Errors:**
    - The loop conditions in shiftUp and shiftDown may produce off-by-one errors if the data structure's size isn't managed correctly.
    - while (--idx != insertion_idx)

Category D: Comparison Errors:

1. **Incorrect Boolean Comparisons:**
    - In functions like findIdx, combining multiple logical operations could result in improper evaluations due to mishandling of && and ||.

```
if (info == mInfo[idx] &&
        ROBIN_HOOD_LIKELY(WKeyEqual::operator()(key,
mKeyVals[idx].getFirst()))) {
        return idx;
    }
```

2. **Mixed Comparisons:**
    - Comparing different types, such as signed and unsigned integers, could yield incorrect results depending on the system/compiler.

Category E: Control-Flow Errors:

1. **Potential Infinite Loop:**
    - Loops such as those in shiftUp and shiftDown might not terminate correctly if their exit conditions are never satisfied.

2. **Unnecessary Loop Executions:**
    - Certain loops may run an extra time or fail to execute due to incorrect initialization or checks in their conditions.

Category F: Interface Errors:

1. **Mismatched Parameter Attributes:**
    - Function calls, like those to insert_move, might have parameters that do not align with expected types or sizes.
    - insert_move(std::move(oldKeyVals[i]));
2. **Global Variables:**
    - The presence of global variables used across various functions can lead to inconsistencies if they aren't managed and initialized properly. While this isn't directly observed, it poses a potential risk in future code expansions.

Category G: Input/Output Errors:

1. **Missing File Handling:**
    - Although the code itself doesn't handle files, any future extensions involving I/O may lead to common file handling errors, such as unclosed files or failure to check end-of-file conditions.

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

- **Category A: Data Reference Errors** is deemed the most effective in this context, primarily due to the manual memory management, pointers, and dynamic data structures involved. Errors related to pointer dereferencing and memory allocation/deallocation can result in severe issues like crashes, segmentation faults, or memory leaks, making it crucial to focus on this category. Additionally, **Computation Errors** and **Control-Flow Errors** are also significant, especially in large projects.

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

- **Concurrency Issues:** The inspection doesn't address multi-threading concerns, such as race conditions or deadlocks. If the program were expanded to support multiple threads, issues with shared resources, locks, and thread safety would require attention.

- **Dynamic Errors:** Some errors related to memory overflow, underflow, or the behavior of the runtime environment may not be caught until the code is executed in real-world scenarios.

4. Is the program inspection technique is worth applicable?

- Yes, the program inspection technique is worthwhile, particularly for identifying static errors that compilers may not flag, such as pointer mismanagement, array boundary violations, and flawed control flow. While it might not uncover every dynamic issue or concurrency-related bug, it is an essential step in ensuring code quality, especially in memory-critical applications like this C++ hash table implementation. This method enhances the code's reliability and promotes adherence to best practices in memory management, control flow, and computational logic.

Task 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

Program Inspection

1. An error was present in the original program, particularly in calculating the remainder. This issue has been detected and resolved.
2. The error falls under **Category C: Computational Errors**, as it involves incorrect computation of the remainder.
3. Program inspections do not address runtime-related issues like logic errors or debugging errors.
4. The program inspection method helps identify and fix problems related to code structure and arithmetic logic.

Debugging

1. As identified earlier, the issue with the remainder calculation has been resolved.
2. To debug the issue, you can use a breakpoint at the remainder calculation to ensure it's correct, and then monitor variable values through each step.
3. Here's the corrected and restructured code for Armstrong numbers:

```
// 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

Program Inspection

1. There were two major errors in the program:
2. **Error 1:** In the gcd function, the condition in the while loop was incorrect. It should have been while(b != 0) instead of while(a % b == 0) to properly calculate the GCD.
3. **Error 2:** The logic for calculating the LCM had an issue that would lead to an infinite loop if not corrected.
4. This type of error falls under **Category C: Computational Errors**, as both functions—gcd and lcm—contained issues with the logic or computation.
5. Program inspections are useful for identifying structural or arithmetic issues, but they cannot spot runtime errors like infinite loops.
6. Using program inspection helps to catch errors related to how calculations are performed in code.

Debugging

1. The program had two errors, as described earlier.
2. Steps to fix the errors:
3. For **Error 1** in the gcd function, setting a breakpoint at the start of the while loop helps ensure that the loop is executing with the correct condition.
4. For **Error 2** in the lcm function, reviewing and correcting the logic prevents the infinite loop issue when calculating the LCM.
5. Below is the corrected 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

1. There was an error in the program, specifically in the line: int option1 = opt[n++][w];. The unintended increment of n caused a computation error. This was corrected to: int option1 = opt[n][w];.
2. This type of error falls under **Category C: Computational Errors**, as the issue arises within the loops handling computation.
3. Program inspection does not catch runtime or logical errors that may occur during the execution phase.

4. Applying program inspection helps to detect and resolve arithmetic and structural issues in the program.

Debugging

1. The error in the line int option1 = opt[n++][w]; was identified and corrected to int option1 = opt[n][w];.
2. To ensure this error is fixed, a breakpoint should be placed on the corrected line to track the values of n and w and ensure n isn't incremented unintentionally.
3. The corrected version of the Knapsack program is as follows:

```
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

1. The program contains two key errors:
2. **Error 1:** The inner while loop had a condition while(sum == 0), which caused incorrect behavior. It was corrected to while(sum > 0) to allow proper execution.
3. **Error 2:** Inside the inner while loop, missing semicolons caused a syntax error in lines like s = s * (sum / 10); sum = sum % 10;, which have now been corrected.
4. This program falls under **Category C: Computation Errors** because the issues are related to computations within the while loop.
5. Program inspection is not equipped to find runtime or logical errors that manifest during program execution.
6. Using program inspection effectively helps to identify and rectify arithmetic errors in the code structure.

Debugging

1. The two errors were identified and corrected as mentioned above.
2. To fix these, you should set a breakpoint at the start of the inner while loop to track the execution and ensure correct values for variables like num and s.
3. Below is the corrected 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 Inspection

1. The program has multiple errors:
2. **Error 1:** In the mergeSort method, incorrect array splitting logic was used in int[] left = leftHalf(array+1); and int[] right = rightHalf(array-1);. The correct approach splits the array without modifying indices.
3. **Error 2:** The methods leftHalf and rightHalf were not correctly returning the two halves of the array, and their logic was adjusted.
4. **Error 3:** In the merge method, passing left++ and right-- was incorrect. Instead, the arrays themselves should be passed.
5. The errors fall under **Category C: Computation Errors**, as they involve incorrect handling of array manipulation.
6. Program inspection, while valuable for identifying such issues, does not detect runtime or logical errors that may occur during execution.

7. Applying program inspection to identify and resolve computation-related issues in the code is highly beneficial.

Debugging

1. The multiple errors in the program were identified and corrected as described.
2. Breakpoints should be used to observe the values of left, right, and array during execution. Additionally, the merge method requires tracking i1 and i2 to ensure correct merging.
3. Here's the corrected executable code for Merge Sort:

```
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 Inspection

1. The program contains the following key issues:
2. **Error 1:** The loop indices for matrix multiplication should start from 0 instead of -1, as starting from -1 would lead to array index out-of-bounds errors.
3. **Error 2:** The error message has an incorrect format. It should be: "Matrices with entered orders can't be multiplied with each other." instead of "Matrices with entered orders can't be multiplied with each other." (notice the misplaced newline character).
4. The **Category C: Computation Errors** is the most appropriate for this case as the errors relate to how matrix multiplication is handled.

5. Program inspection won't catch runtime or logical errors that emerge during program execution, such as index-out-of-bounds errors due to incorrect loop indices.
6. The program inspection technique is valuable here, as it helps detect issues related to computation.

Debugging

1. To resolve these errors, the following steps are required:
    - Fix the loop indices to start at 0.
    - Correct the error message for better readability.
2. Use breakpoints to examine the values of c, d, k, and sum during execution, especially in the nested loops for matrix multiplication.
3. Here's the corrected code for matrix multiplication:

```
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 Inspection

1. The program has the following errors:
2. **Error 1:** In the insert method, there's a typo in the line i += (h * h++). It should be corrected to i = (i + h * h++) % maxSize;.
3. **Error 2:** In the remove method, the logic for rehashing keys is incorrect. The line i = (i + h * h++) % maxSize; is required to ensure correct probing during key removal.
4. **Error 3:** The same issue appears in the get method, where i = (i + h * h++) % maxSize; ensures proper quadratic probing.

5. The errors fall under **Category A: Syntax Errors** and **Category B: Semantic Errors**, as they involve both incorrect syntax and faulty logic for probing in the hash table.
6. Program inspection helps in identifying such errors, though it might not catch deeper logical errors in the probing process.

Debugging

1. You need to step through the code and monitor the variables i, h, tmp1, and tmp2 in the insert, remove, and get methods to ensure that the logic works as intended.
2. Set breakpoints in the loop where quadratic probing is applied to verify that it's functioning correctly.
3. Below is the corrected executable code for quadratic probing:

```
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 Inspection

1. **Errors Identified:**
    - **Error 1:** The class name "Ascending Order" has an extra space and an underscore. It should be corrected to "AscendingOrder".
    - **Error 2:** The first nested for loop has an incorrect loop condition: for (int i = 0; i ¿= n; i++);. This should be changed to for (int i = 0; i < n; i++).
    - **Error 3:** There is an extra semicolon (;) after the first nested for loop that should be removed.
2. **Category of Inspection:** The most effective category of program inspection would be **Category A: Syntax Errors** and **Category B: Semantic Errors**, since the issues involve both syntax errors and semantic problems related to the code's logic.
3. Program inspection can identify and fix syntax errors and some semantic issues but may not detect logic errors affecting the program's behavior.
4. Applying program inspection is worthwhile to fix the syntax and semantic errors, but debugging is needed to address logic errors.

Debugging

1. There are three errors in the program, as identified above.
2. To fix these errors, set breakpoints and step through the code, focusing on the class name, loop conditions, and the unnecessary semicolon.
3. Here's the corrected executable code:

```
import java.util.Scanner;

public class AscendingOrder {
  public static void main(String[] args) {
    int n, temp;
    Scanner s = new Scanner(System.in);

    System.out.print("Enter the number of elements 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 Inspection

1. **Errors Identified:**
    - **Error 1:** The push method incorrectly decrements the top variable (top–) instead of incrementing it. It should be corrected to top++.
    - **Error 2:** The display method has an incorrect loop condition: for(int i=0; i ¿ top; i++). It should be corrected to for (int i = 0; i <= top; i++).
    - **Error 3:** The pop method is missing in the StackMethods class. It should be added to provide a complete stack implementation.
2. **Category of Inspection:** The most effective category of program inspection would be **Category A: Syntax Errors**, as there are syntax errors in the code. Additionally, **Category B: Semantic Errors** could help identify logic and functionality issues.
3. Program inspection is worthwhile for identifying and fixing syntax errors, but additional inspection is needed to ensure logic and functionality are correct.

Debugging

1. There are three errors in the program, as identified above.
2. To fix these errors, 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.
3. Here's the corrected executable code:

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

  public StackMethods(int arraySize) {
     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 Inspection

1. **Errors Identified:**
    - **Error 1:** The line doTowers(topN ++, inter–, from+1, to+1); has errors in the increment and decrement operators. It should be corrected to doTowers(topN - 1, inter, from, to);.
2. **Category of Inspection:** The most effective category of program inspection would be **Category B: Semantic Errors**, as the errors relate to logic and functionality.
3. The program inspection technique is valuable for identifying and fixing semantic errors in the code.

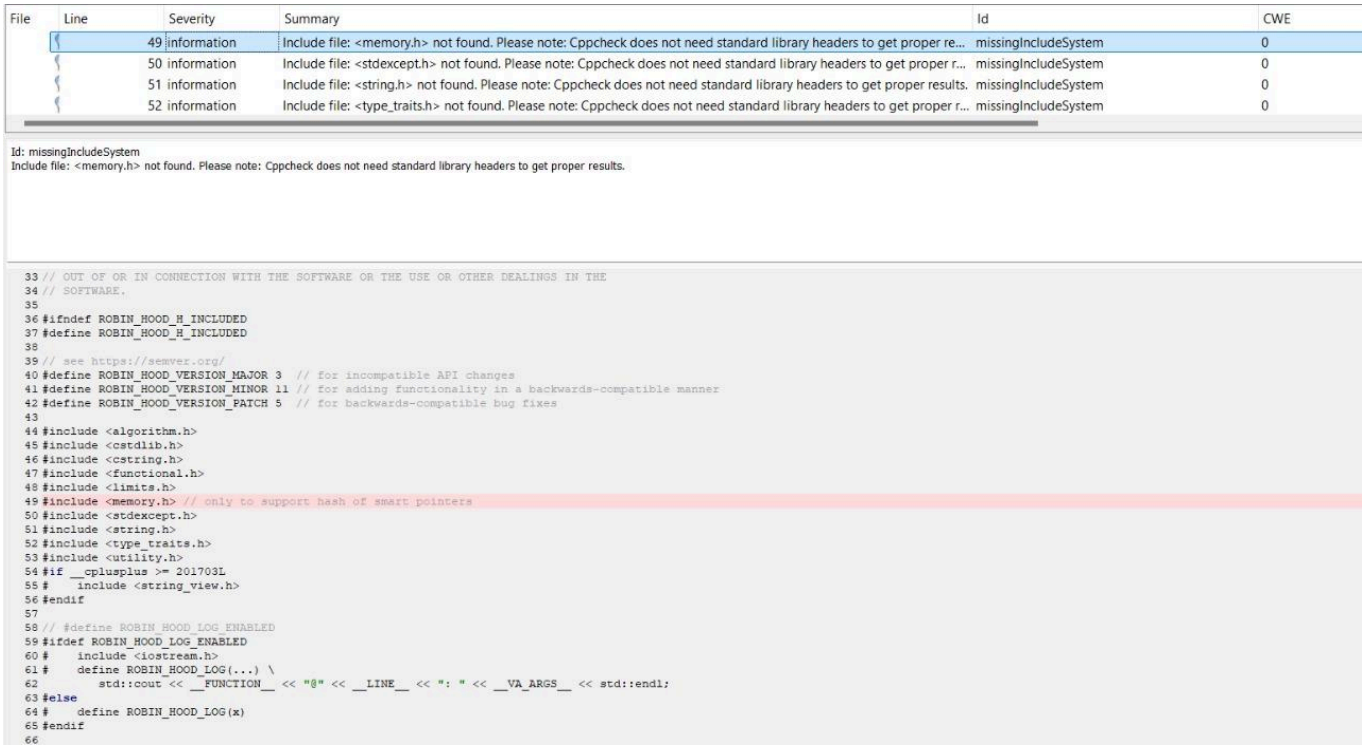
Debugging

1. There is one error in the program, as identified above.
2. To fix this error, replace the 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);
    }
```

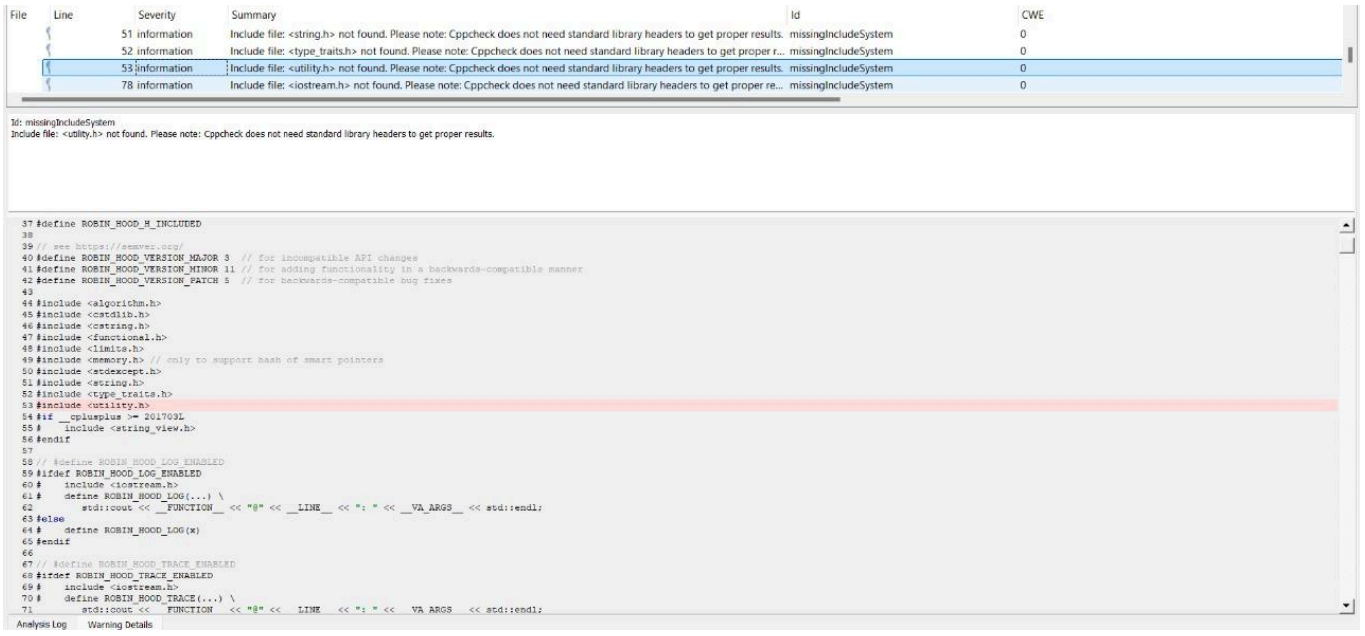
Task III: Static Analysis Tools



File	Line	Severity	Summary	Id	CWE
	49	information	Include file: <memory.h> not found. Please note: Cppcheck does not need standard library headers to get proper re...	missingIncludeSystem	0
	50	information	Include file: <stdexcept.h> not found. Please note: Cppcheck does not need standard library headers to get proper r...	missingIncludeSystem	0
	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

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
```

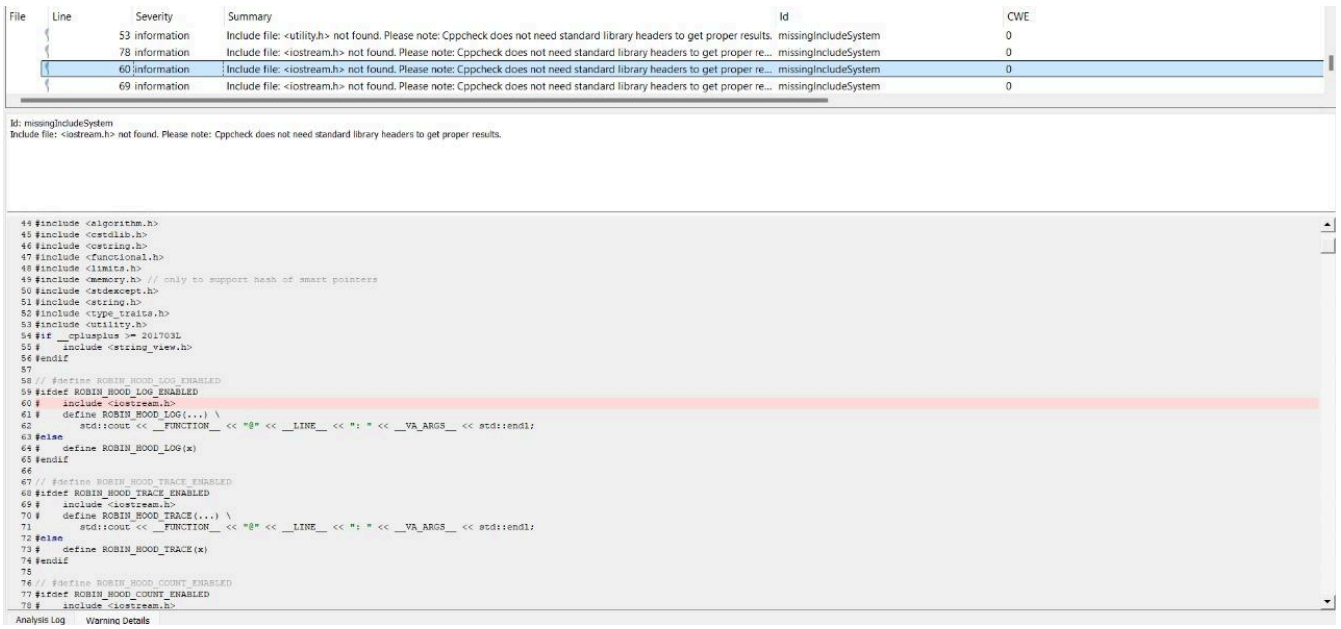


File	Line	Severity	Summary	Id	CWE
	51	information	Include file: <string.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.	missingIncludeSystem	0
	52	information	Include file: <type_traits.h> not found. Please note: Cppcheck does not need standard library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

Analysis Log Warning Details



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 <cmrdlib.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>
```

Analysis Log | Warning Details