Software Engineering - IT314
Lab-07

Name: Nishant Italiya
Student ID: 202201258

PROGRAM INSPECTION

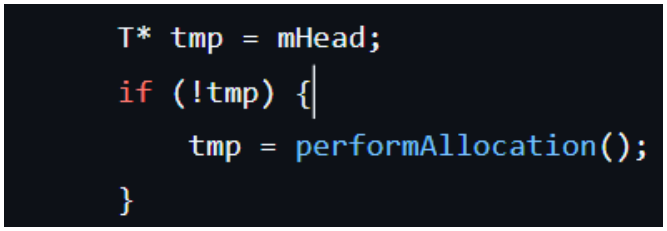
Program Inspection for Robin Hood Hashing Code:

GitHub Code Link:
https://github.com/martinus/robin-hood-hashing/blob/master/src/include/robin_hood.h

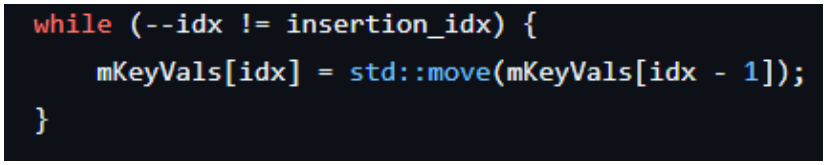
1. **How many errors are there in the program? Mention the errors you have identified.**

Category A: Data Reference Errors:

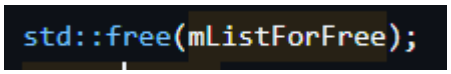
1. **Uninitialized Variables:**
   - `mHead` and `mListForFree` are initialized to `nullptr`, but after memory deallocation, they are not consistently reset, which can lead to uninitialized access or dangling pointers.



2. **Array Bound Violations:**
   - The `shiftUp` and `shiftDown` functions do not perform checks to ensure that array indexing is within bounds.



3. **Dangling Pointers:**
   - In the `BulkPoolAllocator`, the `reset()` function frees memory but does not reset the pointer to `nullptr`, potentially leaving a dangling pointer.



4. **Type Mismatches:**
    - The `reinterpret_cast_no_cast_align_warning` function involves casting memory regions without validating the types, which may introduce subtle errors.

```
T* obj = static_cast<T*>(std::malloc(...));
```

Category B: Data Declaration Errors:

1. **Potential Data Type Mismatches:**
    - In `hash_bytes`, multiple type casts occur. If there is a mismatch in size or attributes between these types, it could lead to unintended behavior.

```
auto k = detail::unaligned_load<uint64_t>(data64 + i);
```

2. **Similar Variable Names:**
    - Variables like `mHead`, `mListForFree`, and `mKeyVals` have similar names, increasing the risk of confusion during modifications or debugging.

Category C: Computation Errors:

1. **Integer Overflow:**
    - In the `hash_bytes` function, several operations involving large integers could result in overflow due to shifts and multiplications.

```
// not doing the final step here, because this will be done by keyToIdx anyways
// h *= m;
// h ^= h >> r;
```

2. **Off-by-One Errors:**
    - Looping in `shiftUp` and `shiftDown` could suffer from off-by-one errors if the size of the data structure is mishandled.

```
while (--idx != insertion_idx)
```

Category D: Comparison Errors:

1. **Incorrect Boolean Comparisons:**

- In complex conditional statements like those found in `findIdx`, improper handling of logical operators (&& and ||) could cause incorrect evaluations.

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

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

Category E: Control Flow Errors:

1. **Potential Infinite Loop:**
   - In functions like `shiftUp` and `shiftDown`, there is a risk of the loop running indefinitely if the termination condition is not met.

```
while (--idx != insertion_idx)
```

2. **Unnecessary Loop Executions:**
   - Certain loops may run an extra iteration or not execute at all due to incorrect initialization or condition checks.

```
for (size_t idx = start; idx != end; ++idx)
```

Category F: Interface Errors:

1. **Mismatched Parameter Attributes:**
   - Functions like `insert_move` may receive arguments that do not match the expected attributes, such as incorrect data types or sizes.

```
void insert_move(Node&& keyval)
```

2. **Global Variables:**
   - If global variables are used across different functions, their initialization and use must be consistent to avoid issues. While this is not explicitly present in the code, it could become a problem in future expansions.

Category G: Input/Output Errors:

1. **Missing File Handling:**
    - Although the current code does not deal directly with file handling, any future extensions involving I/O operations might introduce typical file handling issues. These can include unclosed files, failure to check end-of-file conditions or insufficient error handling.

2. **Which category of program inspection would you find more effective?**
    - **Category A:** Data Reference Errors are the most crucial in this context due to the use of manual memory management, pointers, and dynamic data structures. Errors in pointer dereferencing or improper memory management (allocation/deallocation) can easily result in serious problems, such as crashes, segmentation faults, or memory leaks. It is essential to focus on this category. Additionally, Computation Errors and Control Flow Errors are also significant, especially for larger projects.

3. **Which type of error are you unable to identify using program inspection?**
    - **Concurrency Issues:** The inspection doesn't cover multi-threading or concurrency-related errors, such as race conditions or deadlocks. If the program is later expanded to handle multiple threads, issues with shared resources, locking mechanisms, and thread safety would need to be considered.
    - **Dynamic Errors:** Errors like memory overflow, underflow, or problems related to the runtime environment may not be detectable until the code is executed in a real-world scenario.

4. **Is the program inspection technique worth applying?**
    - Yes, program inspection is highly beneficial, particularly for detecting static errors that may not be identified by compilers, such as incorrect pointer management, array boundary issues, or improper control flow. While it may not capture all dynamic or concurrency-related bugs, it is an essential step for ensuring code

quality, especially in memory-sensitive applications like this C++ hash table implementation. Applying inspection techniques enhances the reliability of the code and encourages best practices in memory management, control flow, and computational logic.

CODE DEBUGGING

Frag-1: Armstrong Number

1. **How many errors are there in the program? Mention the errors you have identified:**
    - **Logical Error in Extracting Digits:**
        - In the `while` loop, the operation `remainder = num / 10;` is incorrect for extracting the last digit. It should be `remainder = num % 10;` as using division instead of modulo will give incorrect digits.
        - Similarly, `num = num % 10;` should be `num = num / 10;` to correctly remove the last digit.
    - **Incorrect Result Calculation:**
        - Due to the above logical errors, the program does not properly compute the sum of the cubes of the digits.
    - **Argument Error:**
        - If the user does not provide any argument via `args[]`, the program will throw an `ArrayIndexOutOfBoundsException`. There should be a validation check to ensure that the user provides input.
    - **Typographical Issue:**
        - The output message should state, "is not an Armstrong Number" instead of "is not an Armstrong Number."
2. **Which category of program inspection would you find more effective?**
    - **Category C: Computation Errors** is the most applicable here because the primary issue relates to the incorrect extraction of digits and miscalculations when determining if a number is an Armstrong number.
    - **Category E: Control-Flow Errors** is also useful for identifying potential exceptions, like the lack of argument validation.
3. **Which type of error are you unable to identify using program inspection?**
    - **Performance Issues or Optimizations:** Potential performance improvements are not identified during this inspection.
    - **Runtime Exceptions:** Errors such as `NumberFormatException`, which might occur if the input is not a valid number, are not explicitly covered by this inspection.

4. **Is the program inspection technique worth applying?**
    - Yes, program inspection is valuable for catching logical and computational errors, especially in smaller programs. In this case, it helped detect key issues with digit extraction and flow control, preventing miscalculations and exceptions.

5. How many breakpoints do you need to fix those errors?

Two breakpoints:

1. On the line where the remainder is calculated (remainder=num 10;).
2. On the line where num is updated (num = num % 10;).

a. What are the steps you have taken to fix the error you identified in the code fragment?

● **Step 1:** Fix the calculation of the remainder to correctly extract the last digit (remainder = num % 10;).

● **Step 2**: Correctly update num to remove the last digit (num = num / 10;)

Frag-2: GCD and LCM

1. **How many errors are there in the program? Mention the errors you have identified:**
    - **Logical Error in GCD Calculation:**
        - In the `gcd` method, the condition `while(a % b == 0)` is incorrect. As noted in the comments, it should be `while(a % b != 0)` to avoid entering an infinite loop and to correctly compute the GCD.
    - **Logical Error in LCM Calculation:**
        - In the `lcm` method, the condition `if(a % x != 0 && a % y != 0)` is incorrect. The correct condition should be `if(a % x == 0 && a % y == 0)` to properly find the least common multiple. The current condition leads to incorrect results.
    - **Potential Infinite Loop in LCM Calculation:**

        - Due to the incorrect condition in the `lcm` method, the program might enter an infinite loop when calculating the LCM. This is because the code might never find the correct multiple and will keep incrementing indefinitely.
2. **Which category of program inspection would you find more effective?**
    - **Category C: Computation Errors** is the most relevant here because the primary issue lies in the incorrect logic for calculating both the GCD and LCM. The conditions in both methods are flawed.
3. **Which type of error are you unable to identify using program inspection?**
    - **Performance Issues:** The inspection does not highlight performance inefficiencies, such as the inefficiency in the LCM calculation, where incrementing by 1 could be optimized.
    - **Input Validation:** The program does not handle invalid inputs like negative numbers or zero, which could cause unexpected behavior or crashes, but this is not identified through program inspection.
4. **Is the program inspection technique worth applying?**
    - Yes, program inspection is effective here as it successfully identifies the main logical errors in both the GCD and LCM methods. These errors would otherwise lead to incorrect calculations or infinite loops. Program inspection helps in catching bugs related to fundamental mathematical operations.

5. How many breakpoints do you need to fix those errors?

You need two breakpoints to debug and fix the identified errors:

● A breakpoint at the beginning of the gcd method to monitor the values of a, b, and r.

● A breakpoint at the beginning of the lcm method to check the initial value of a and how it increments during the loop.

a. What are the steps you have taken to fix the errors you identified in the code fragment?

1. Fixing the gcd Method:

● Changed the condition in the while loop from while (a % b == 0) to while (a % b != 0) to correctly implement the Euclidean algorithm for calculating the GCD.

2. Fixing the lcm Method:

● Modified the condition in the if statement from if (a % x != 0 && a %y!=0) to if (a % x == 0 && a %y==0)to ensure that the method correctly identifies when a is a multiple of both x and y

Frag-3: Knapsack

1. **How many errors are there in the program? Mention the errors you have identified:**
    - **Array Indexing Error (Logical Error):**
        - In the line `int option1 = opt[n++][w];`, the use of `n++` increments n after accessing `opt[n][w]`, which is incorrect. It should be `opt[n-1][w]` to ensure the current item isn't skipped. Using the post-increment operator here causes the program to access the wrong row in the `opt` array, leading to incorrect decisions during the solution process.
    - **Logical Error in `option2` Calculation:**
        - The condition `if (weight[n] > w)` should be changed to `if (weight[n] <= w)` since we only consider an item if its weight is less than or equal to the current capacity `w`. The current condition wrongly skips items that fit in the knapsack.
        - Additionally, the line `option2 = profit[n-2] + opt[n-1][w-weight[n]]` is incorrect. The correct indexing for the `profit` array should be `profit[n]` because the program is dealing with the nth item. The wrong indexing leads to incorrect profit calculations.
2. **Which category of program inspection would you find more effective?**
    - **Category C: Computation Errors** is the most relevant here because the main problem is due to improper array indexing and logic errors in the calculation of the optimal solution. Correcting

these computational errors ensures that the algorithm selects the correct items based on weight and profit.

3. **Which type of error are you unable to identify using program inspection?**
    - **Edge Case Handling**: The program does not check for edge cases, such as when `N` or `W` are zero or negative. It also doesn't validate whether enough arguments are passed via `args[]`. These runtime issues are not caught by static inspection.
    - **Performance Optimization**: The program's performance may be suboptimal for large input sizes. While it functions for moderate inputs, potential bottlenecks are not identified through program inspection.
4. **Is the program inspection technique worth applying?**
    - Yes, the inspection is valuable as it uncovers significant logical errors within the core knapsack algorithm. These array indexing and condition mistakes would have resulted in incorrect outputs. Fixing these issues is crucial for ensuring the program functions correctly.

5. How many breakpoints do you need to fix those errors?

You would need **three breakpoints** to debug and fix the errors:

- **Set a breakpoint** at the beginning of the nested loop to check the values of `n`, `w`, `opt[n][w]`, and other variables.
- **Set a breakpoint** right before the assignment of `option 1` to monitor how n is changing.
- **Set a breakpoint** after the assignment of `option 2` to verify the calculations for both `option 1` and `option 2.

a. What are the steps you have taken to fix the error you identified in the code fragment?

1. **Correcting Array Indexing**:

   Changed `int option1 = opt[n++][w];` to `int option1 = opt[n][w];` to prevent n from being incremented incorrectly.

2. **Correcting Profit Calculation**:

Modified the line `int option2 = profit[n-2] + opt[n-1][w-weight[n]];` to `int option2 = profit[n] + opt[n-1][w-weight[n]];` to reference the correct item profit.

3. **Adjusting Weight Condition Logic**:
    - Added a condition to ensure that `option 2` is only calculated if the current item's weight does not exceed `w`. This prevents erroneous profit calculations for items that can't be added.

Frag-4: Magic Number

1. **How many errors are there in the program? Mention the errors you have identified:**
    - **Error in Inner While Loop Condition:**
        - The inner `while(sum == 0)` condition is incorrect. The logic should involve breaking down the `sum` into its digits. It should be `while(sum != 0)` to properly compute the sum of the digits.
    - **Error in Digit Multiplication and Summing Logic:**
        - The line `s = s * (sum / 10);` is wrong. Instead of accumulating the sum of the digits, it incorrectly multiplies and divides the number. The correct approach is to add the last digit to `s`. It should be `s = s + (sum % 10);` to achieve the desired summation.
    - **Missing Semicolon in Inner Loop:**
        - The line `sum = sum % 10` is missing a semicolon. It should be `sum = sum % 10;` to fix the syntax.
2. **Which category of program inspection would you find more effective?**
    - **Category C: Computation Errors** is the most effective here since the main problem involves incorrect logic and conditions in the loops responsible for summing the digits. The issues are primarily computational and logical.

3. **Which type of error are you unable to identify using program inspection?**
    - **Edge Case Handling**: The program inspection doesn't account for handling inputs such as negative numbers or zero, which could lead to potential issues.
    - **Performance**: The inspection doesn’t address performance optimization for larger inputs, though in this case, the simplicity of the operation makes performance less of a concern.
4. **Is the program inspection technique worth applying?**
    - Yes, program inspection is beneficial because it uncovers significant logical errors in determining whether a number is a magic number. Without these corrections, the program would fail to produce correct results for valid inputs.

5. How many breakpoints do you need to fix those errors?

You would need **three breakpoints** to effectively debug and fix the errors:

1. **Set a breakpoint** at the beginning of the inner loop to observe the values of `sum` and `s`.
2. **Set a breakpoint** at the beginning of the outer loop to check the current value of `num`.
3. **Set a breakpoint** before the final `if` statement to verify the final value of `num` before making the magic number determination.

a. What are the steps you have taken to fix the error you identified in the code fragment?

1. **Correcting the Inner Loop Condition**:

    ● Changed while(sum==0) to while(sum!=0) to ensure the loop iterates while there are digits left to process.

2. Fixing the Digit Summation Logic:

● Updated the line s=s*(sum/10); to s = s + (sum % 10); to accumulate the digits correctly.

3. Adding Missing Semicolon:

● Added asemicolon at the end of sum = sum % 10;.

4. Adjusting the Outer Loop Condition:

● Changed the outer loop condition from while(num>9) to while(num > 9 || num == 0) to handle the case where num might reduce to zero.

Frag-5: Merge Sort

1. **Identified Errors in the Program**:
    - **Error in Recursive Array Splitting**:
        - The expressions `array + 1` and `array - 1` in the `mergeSort` method are incorrect. They attempt to manipulate the array reference directly, which is invalid. Instead, the program should use `leftHalf(array)` and `rightHalf(array)` correctly without altering the array reference.
    - **Incorrect Use of Increment and Decrement Operators in Merge**:
        - The line `merge(array, left++, right--);` is incorrect. Using `left++` and `right--` when passing indices to the merge function will cause issues, as arrays are not primitive values and cannot be incremented or decremented in this manner. The correct usage should be `merge(array, left, right);`.
2. **Effective Program Inspection Category**:
    - **Category B: Data and Control Flow Errors** is the most effective here, as the program has issues with array reference manipulation, impacting the control flow of recursive function calls. Validating data manipulation and ensuring proper program flow are essential.
3. **Type of Error Not Identified by Program Inspection**:
    - Program inspection may overlook **performance optimization** of the merge sort algorithm. While merge sort operates at O(n log n) efficiency, inspection does not typically address optimization concerns, particularly for larger datasets.
4. **Value of Program Inspection Technique**:
    - Yes, program inspection is beneficial as it helps identify fundamental errors in array handling and recursion logic, which are crucial for the correct operation of the merge sort algorithm.

5. How many breakpoints do you need to fix those errors?

You would need **three breakpoints** to effectively debug and fix the errors:

1. **Set a breakpoint** at the beginning of the `mergeSort` method to inspect how the array is being split and to examine the left and right halves.
2. **Set a breakpoint** before the merge operation to check the contents of the left and right arrays.
3. **Set a breakpoint** inside the merge method to observe how elements are being merged back into the original array.

a. What are the steps you have taken to fix the error you identified in the code fragment?

1. **Correcting Array Slicing**:

● Instead of int[] left = leftHalf(array + 1); and int[] right = rightHalf(array-1);, change it to correctly split the array using Arrays.copyOfRange.

2. Fixing Parameters in Recursive Calls:

● Update the call to merge by passing the arrays without using the increment/decrement operators: merge(array, left, right);.

3. Adjusting Size Calculations:

● Change the size calculation in leftHalf and rightHalf methods to (array.length + 1) / 2 for the left half and the rest for the right half.

4. Merging Logic:

● Ensure that the merge method correctly combines the sorted arrays back into the original array.

Frag-6: Multiply Matrices

1. **Identified Errors in the Program**:
    - **Incorrect Indices in the Matrix Multiplication Loop**:
        - In the innermost loop of matrix multiplication, the expressions `first[c-1][c-k]` and

`second[k-1][k-d]` are incorrect. These indices are decremented improperly, leading to potential `ArrayIndexOutOfBoundsException`. The correct expressions should be `first[c][k]` and `second[k][d]` to access the appropriate elements of the matrices.

- **Input Prompt for the Second Matrix**:
  - The prompt following the first matrix incorrectly asks for the "number of rows and columns of the first matrix" again. This is a typo; it should refer to the "second matrix."

2. **Effective Program Inspection Category**:
   - **Category A: Algorithmic Errors** are the most effective in this context, as the errors relate to the logic of matrix multiplication. Accurate index calculations are essential for achieving the correct product of the matrices.
3. **Type of Error Not Identified by Program Inspection**:
   - Program inspection may not catch **input validation** issues. The program assumes that the input matrices are valid and does not verify against non-numeric inputs, which could lead to crashes. Inspection often overlooks the need for robustness against user input errors.
4. **Value of Program Inspection Technique**:
   - Yes, program inspection is valuable as it helps identify logical issues, such as index miscalculations, which can result in incorrect outcomes or runtime errors, particularly in the context of matrix multiplication.

5. How many breakpoints do you need to fix those errors?

You would need **three breakpoints** to effectively debug and fix the errors:

1. **Set a breakpoint** inside the multiplication loop to inspect the indices and the values being multiplied.
2. **Set a breakpoint** before printing the multiplication results to check the contents of the multiplication array.
3. **Set a breakpoint** after reading the second matrix to verify that the inputs are being read correctly.

a. What are the steps you have taken to fix the error you identified in the code fragment?

1. **Correcting Array Indexing**:

   ● Change sum =sum+first[c-1][c-k] * second[k-1][k-d]; to sum = sum + first[c][k] * second[k][d]; to correctly access the elements of the matrices.

2. Resetting Variables:

● The reset of the sum variable to the beginning of the inner loop for d to ensure it starts fresh for each element calculation: sum = 0; should be at the start of the for (d = 0; d < q; d++) loop.

3. Fixing Input Prompts:

● Update the prompt for the second matrix to say “Enter the number of rows and columns of the second matrix”.

4. Adjusting Output Formatting:

● Consider adding headers to clarify that the following output is the product matrix

Frag-7: Quadratic Probing

1. **Identified Errors in the Program**:
   - **Operator Error in Insert Method**:
     - The line `i += (i + h / h--);` is incorrect. The correct syntax should be `i = (i + h * h++) % maxSize;`. The `+=` operator should not have a space, and using `/ h--` is logically incorrect for quadratic probing; it should utilize `h * h`.
   - **Incorrect Rehashing Logic in Remove Method**:
     - The rehashing logic following a removal incorrectly includes an extra `currentSize--` decrement. This is unnecessary since `currentSize` is already decremented when a key is removed and should not be adjusted again.
   - **Unused Print Statement**:
     - The print statement `System.out.println("i " + i);` in the `get` method appears to be a debugging statement. It

should be removed or adjusted based on the actual use case.

2. **Effective Program Inspection Category**:
    - **Category A: Algorithmic Errors** is the most effective in this situation, as the primary issues stem from the incorrect implementation of the quadratic probing algorithm, which directly impacts the hash table's behavior during insertion and removal.
3. **Type of Error Not Identified by Program Inspection**:
    - Program inspection may not reveal **performance bottlenecks** that occur under high load factor scenarios. While the algorithm may perform adequately at lower loads, degradation in performance due to increased load factors can only be detected through testing and profiling, rather than static inspection.
4. **Value of Program Inspection Technique**:
    - Yes, inspection is beneficial for identifying logical errors, such as incorrect probing methods or improper decrementing of the current size. It helps prevent subtle issues in fundamental data structure implementations.

5. How many breakpoints do you need to fix those errors?

To fix these errors, you would need the following breakpoints:

1. **Breakpoint on the Insert Method**: Set a breakpoint before the line containing the `i +=` operator to check the current value of `i`.
2. **Breakpoint on the Hash Method**: Set a breakpoint to observe how the hash value is calculated for different keys.
3. **Breakpoint on the Remove Method**: Set a breakpoint to ensure the correct key is being removed and to check the state of the hash table after the removal.
4. **Breakpoint in the Print Method**: Set a breakpoint to validate that the correct values are being printed from the hash table.

a. What are the steps you have taken to fix the error you identified in the code fragment?

1. **Correcting the Insert Method**: Remove the space in the `+=` operator and correct the logic for incrementing `h`.
2. **Fixing the Hash Method**: Ensure that the hashing algorithm doesn't modify `h` directly and avoids leading to an infinite loop.

3. **Updating Removal Logic**: Adjust the `remove` method to ensure `currentSize` is only decremented once after a successful removal.
4. **Enhancing Print Logic**: Add checks to prevent printing null values and ensure that the output format is clear.
5. **Adjusting the Make Empty Logic**: Modify the `makeEmpty` method to reset the actual contents of the keys and values arrays.

Frag-8: Sorting Array

1. **Identified Errors in the Program**:
    - **Syntax Error in Class Name**:
        - The class name `Ascending _Order` contains an invalid space. Java class names cannot have spaces; it should be written as `AscendingOrder` (without spaces or underscores).
    - **Logical Error in the First Loop**:
        - The loop `for (int i = 0; i >= n; i++);` is incorrect. The condition `i >= n` will always be false. It should be `i < n` to iterate through the array. Additionally, there's an unnecessary semicolon (;) at the end of the for loop, which causes the loop body to be skipped.
    - **Comparison Logic Error**:
        - The condition `if (a[i] <= a[j])` is incorrect for sorting in ascending order. It should be `if (a[i] > a[j])` to swap elements when the current element is greater than the next one.
2. **Effective Program Inspection Category**:
    - **Category B: Logical Errors** is more effective in this case. The incorrect loop condition and comparison logic can be easily identified by inspecting how the loops function and detecting logical issues in the sorting mechanism.
3. **Type of Error Not Identified by Program Inspection**:
    - **Runtime performance issues** or inefficiencies with large datasets cannot be identified through static inspection. While logic errors can be corrected, inspection cannot fully predict the algorithm's performance with larger arrays.
4. **Value of Program Inspection Technique**:

- Yes, program inspection is valuable as it helps catch fundamental syntax and logic errors, such as improper loop conditions and comparison mistakes. This ensures that the sorting logic is correct and functions as intended, serving as a crucial step before testing with actual data.

5. How many breakpoints do you need to fix those errors?

To resolve these errors, you would need the following breakpoints:

1. **Breakpoint on Class Declaration**: Set this to verify the correct naming of the class.
2. **Breakpoint on Outer Loop**: Place this to observe the initial value of `i` and ensure that the loop condition is correct.
3. **Breakpoint on Sorting Logic**: Set this to validate the values of `a[i]` and `a[j]` before and after swapping.
4. **Breakpoint on Output**: Use this to check the formatting of the output and ensure it doesn't include unwanted commas.

a. What are the steps you have taken to fix the error you identified in the code fragment?

1. **Renaming the Class**: Change the class name from `Ascending_Order` to `AscendingOrder`.
2. **Correcting the Loop Condition**: Update the loop condition from `i >= n` to `i < n`.
3. **Removing the Semicolon**: Eliminate the unnecessary semicolon after the outer loop declaration.
4. **Fixing the Sorting Logic**: Modify the sorting condition to `if (a[i] > a[j])`.
5. **Formatting the Output**: Adjust the output logic to prevent trailing commas.

Frag-9: Stack Implementation

1. **Identified Errors in the Program**:
   - **Logic Error in the Push Method**:
     - In the `push()` method, the line `top--` is incorrect. It should increment `top` to add a new element at the next

available position in the stack, so it should be `top++` instead.

- **Logic Error in the Display Method**:
  - The loop condition in the `display()` method is incorrect. It uses `i > top`, which will prevent the loop from running. The loop should iterate from the `top` to `0` (i.e., `i <= top`) to display all elements in the stack.
- **Logic Error in the Pop Method**:
  - In the `pop()` method, `top++` is used to remove the top element. This should be `top--` to correctly decrease the `top` pointer and effectively remove the element from the stack.

2. **Effective Program Inspection Category**:
   - **Category B: Logical Errors** is the most effective in this case, as the main issue involves the incorrect manipulation of the `top` pointer in both the `push` and `pop` methods, as well as the loop condition in `display`.
3. **Type of Error Not Identified by Program Inspection**:
   - **Memory management issues** and runtime errors, such as stack overflow or array index out of bounds, may not be fully identifiable during program inspection without proper testing. Problems may arise if stack operations exceed its allocated size.
4. **Value of Program Inspection Technique**:
   - Yes, program inspection is crucial for identifying fundamental logic errors that can hinder the correct functioning of the stack. Misuse of the `top` variable could lead to improper stack operations, making it easily identifiable through inspection.

5. How many breakpoints do you need to fix those errors?

To address these errors, you would need the following breakpoints:

1. **Breakpoint on push Method**: Set this to check the value of `top` before and after the increment operation.
2. **Breakpoint on pop Method**: Use this to observe the value being popped and the state of `top`.
3. **Breakpoint on `display` Method**: Place this to verify the loop condition and ensure that all elements are printed correctly.

a. What are the steps you have taken to fix the error you identified in the code fragment?

1. **Corrected Logic in push Method**: Change `top--;` to `top++;` to ensure that the next element is added at the correct index.
2. **Corrected Logic in pop Method**: Change `top++;` to `top--;` to correctly remove the top element from the stack.
3. **Updated Loop Condition in `display` Method**: Modify the loop condition from `i > top` to `i <= top` to ensure all elements in the stack are displayed.
4. **Return Value in pop Method**: Adjust the `pop` method to return the value that was popped from the stack.
5. **Adjust the Display Logic**: Ensure the `display` method accurately reflects the current state of the stack after popping elements.

Frag-10: Tower of Hanoi

1. **Identified Errors in the Program**:
   - **Logic Error in Recursive Calls**:
     - In the second recursive call `doTowers(topN++, inter--, from + 1, to + 1)`, the use of `++` and `--` operators on `topN` and `inter` is incorrect. These operators should not be applied here; instead, the parameters should be passed as they are without incrementing or decrementing. The correct recursive call should be `doTowers(topN - 1, inter, from, to)` for the second half of the process.
   - **Incorrect Use of Parameters**:
     - The parameters `from + 1` and `to + 1` do not make sense in the context of character parameters. They should remain as characters (`from`, `to`) and should not be modified.
2. **Effective Program Inspection Category**:
   - **Category B: Logical Errors** would be the most effective here, as the main issues arise from incorrect logic in the recursive calls and the manipulation of parameters.
3. **Type of Error Not Identified by Program Inspection**:

- **Infinite recursion** could be a potential issue if the termination condition is improperly set or mistakenly altered. This problem might not be detected until runtime, especially in recursive functions.

4. **Value of Program Inspection Technique**:
   - Yes, program inspection is crucial in this case as it helps identify logical errors that can lead to incorrect functionality or infinite loops in recursive methods. Ensuring the correct flow of logic in recursive functions is essential for their proper execution.

5. How many breakpoints do you need to fix those errors?

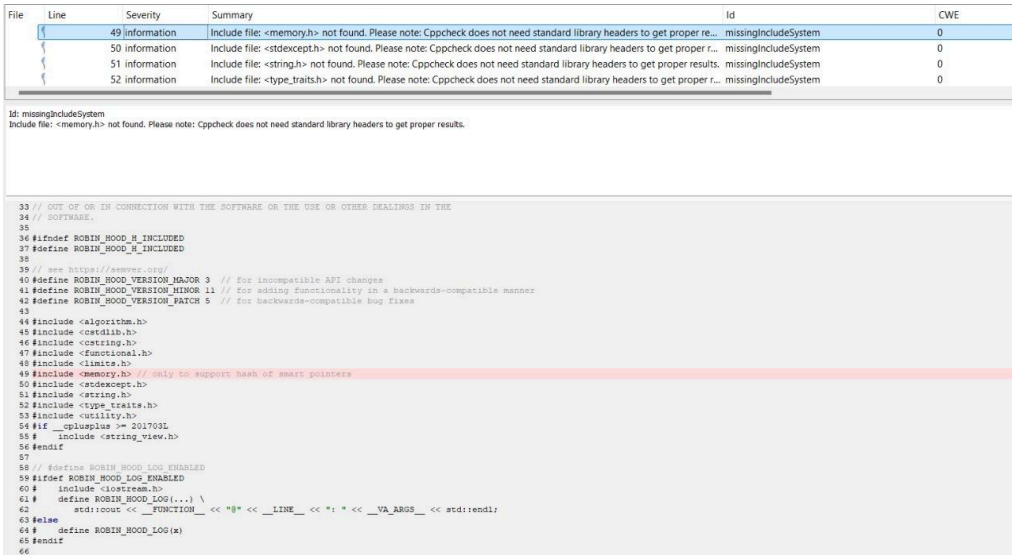
To resolve the identified errors, you would need the following breakpoints:

1. **Breakpoint on the first `doTowers` call**: Set this to check the values of `topN`, `from`, `inter`, and `to` before executing the recursive calls.
2. **Breakpoint before the printing statement**: Use this to observe the correct flow of disk movements.
3. **Breakpoint on the second `doTowers` call**: Place this to ensure the parameters are being correctly passed after the first recursive call.

a. What are the steps you have taken to fix the error you identified in the code fragment?

1. **Corrected Recursive Call**: Change `doTowers(topN++, inter--, from + 1, to + 1)` to `doTowers(topN - 1, inter, from, to)` in the recursive call for moving the remaining disks.
2. **Removed Invalid Modifications**: Ensure that the values for `from`, `inter`, and `to` are not modified with post-increment and post-decrement operators. Instead, pass the original variables directly.
3. **Clarified Disk Movement Logic**: Confirm that the recursive logic adheres to the Tower of Hanoi algorithm to ensure proper execution.

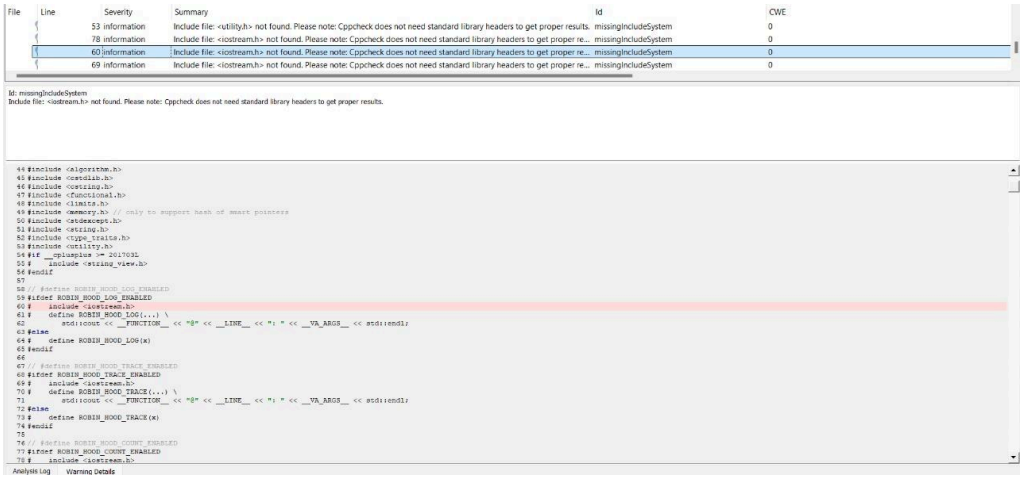
Static Analysis Tools



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
```

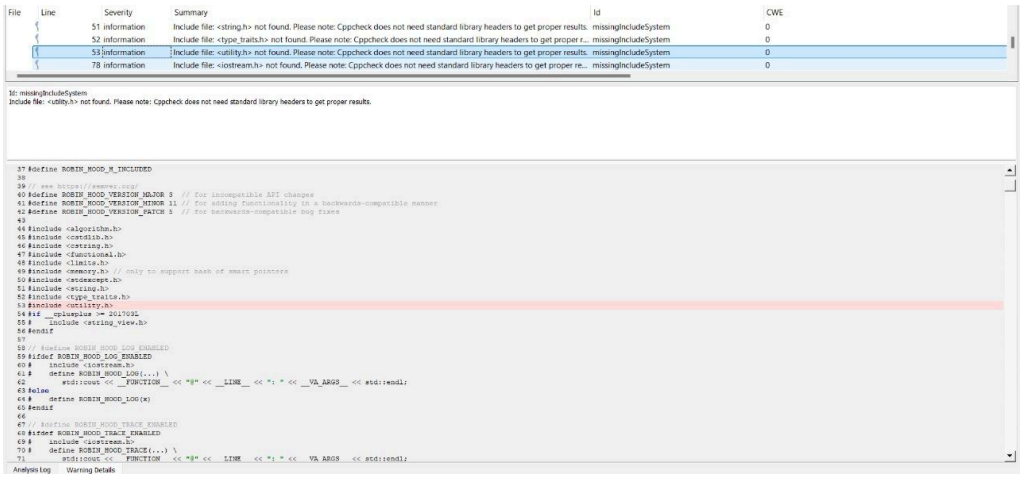


File	Line	Severity	Summary	Id	CWE
	53	information	Include file: <utility.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.	missingIncludeSystem	0
	78	information	Include file: <iostream.h> not found. Please note: Cppcheck does not need standard library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

Analysis Log Warning Details



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