IT-314: Software Engineering

Lab-07: Program Inspection, Debugging and Static Analysis

Aaditya Pithadiya - 202201224

Part-II Code Debugging

Debugger used : GDB Debugger
Debugger workspace : Online

Armstrong Problem:

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

Solution:

- ➔ In the corrected code, the method **getDigits(int num)** is introduced. It calculates the number of digits in the given input number. This is important because an Armstrong number's calculation depends on the number of digits in the number. In the original versions, this assumption was hardcoded for 3-digit numbers.
- ➔ There was correction in the calculator of remainder, num / 10 to num % 10.
- ➔ Also, the termination of the last digit is done by num = num / 10.

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

```
public static int getDigits(int num) {
    int digits = 0;
    while(num != 0) {
        digits++;
        num /= 10;
    }
    return digits;
}
```

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

Solution:

- ➔ 2 Breakpoints needed
  - Before the getDigits method to ensure that the number of digits is being calculated correctly.
  - After the while loop to check the value of check and verify that each digit raised to the power of digits is summed correctly.

3. **Complete Executable Code:** In attached “Java Corrected Codes” folder.

GCD and LCM Problem:

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

Solution:

- ➔ Here, in the gcd function, a will be the greatest number.
- ➔ Also, in the gcd function, the while loop will run till a % b != 0.
- ➔ Also, in lcm function, the corrected condition under loop is (a % x == 0 && a % y == 0).

```
while(true) {
    if(a % x == 0 && a % y == 0)
        return a;
    ++a;
}
```

```
while(a % b != 0) {
    r = a % b;
    a = b;
    b = r;
}
return r;
```

```
a = (x > y) ? x : y;
b = (x < y) ? x : y;
```

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

Solution:

- ➔ 3 Breakpoints needed
    - Inside the gcd function, right before the while loop to ensure that a is the greater number and b is the smaller number before entering the loop.
    - Inside the gcd function, within the while loop for the value of r and ensure that the condition a % b != 0 correctly reflects the state of the calculation. This helps to verify that the loop continues until the GCD is found.
    - Inside the lcm function, right before returning a to Confirm that the condition if (a % x == 0 && a % y == 0) is being evaluated correctly and that a is indeed the least common multiple when it is returned.

3. Complete Executable Code: In attached “Java Corrected Codes” folder.

Knapsack Problem:

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

Solution:

- ➔ The condition for not taking the item is wrong, as it incorrectly used n++, leading to an out-of-bounds error and incorrect access to the opt array. So, Changed to int option1 = opt[n-1][w]; to correctly access the previous row in the opt array without modifying n.
- ➔ if (weight[n] > w) checked if the current item's weight was greater than the remaining capacity, but we Changed to if (weight[n] <= w) to ensure the option for taking the item is considered when it fits within the remaining capacity.
- ➔ option2 = profit[n-2] + opt[n-1][w-weight[n]]; used the wrong index for the profit array, which could lead to incorrect calculations, So, Updated to option2 = profit[n] + opt[n-1][w - weight[n]]; to ensure it uses the current item's profit.

```
for (int n = 1; n <= N; n++) {
    for (int w = 1; w <= W; w++) {
        int option1 = opt[n-1][w];

        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);
    }
}
```

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

Solution:

- ➔ 5 Breakpoints needed
    - At the beginning of the main method, after reading user input to verify that the inputs N (number of items) and W (maximum weight) are correctly read from the user.
    - Inside the loop where random profits and weights are generated (just after the for loop that initializes profit and weight), to check that the profit and weight arrays are being populated correctly. This allows you to see the random values being assigned and ensures they fit within the expected ranges.
    - Inside the first nested for loop before calculating option1, to Examine the values of n and w before calculating the maximum profit options. This helps ensure that the loop is iterating correctly through all items and weight limits.

- Inside the nested loop after calculating option1 and option2 (after the conditional check for weight[n] <= w), to Check the values of option1, option2, opt[n][w], and sol[n][w] to ensure that the calculations are producing the expected results.
- After the loop that determines which items to take (after updating the take array), to Verify the final state of the take array before printing the results. This ensures that the correct items are flagged for inclusion in the knapsack.

3. **Complete Executable Code:** In attached “Java Corrected Codes” folder.

Magic Number Problem:

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

Solution:

- ➔ The inner while loop condition sum == 0 is incorrect, as it should continue looping while num > 0 to break down the number digit by digit.
- ➔ Multiplying s = s * (sum / 10) is incorrect for sum calculation. It should be sum += (num % 10) to sum the digits.
- ➔ There was also a missing semicolon ; after sum = sum % 10, and it should be replaced with num /= 10;
- ➔ Hence the overall magic number calculation logic was corrected.

```
int n = num;
while(num > 9) {
    int sum = 0;

    while(num > 0) {
        sum += (num % 10);
        num /= 10;
    }

    num = sum;
}
```

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

Solution:

- ➔ 5 Breakpoints needed
    - At the Start of the main Method, to verify that the program is correctly reading the input value. wecan check the value of num to ensure that the input is stored correctly before moving into the logic.
    - Before Entering the Outer while Loop, which ensures that the condition num > 9 is correctly evaluated. At this point, we can check the initial value of num and see if the loop will begin for numbers greater than 9 (since the loop only runs if num is greater than 9).

- Inside the Inner while Loop (Summing Digits), where the program extracts digits and calculates the sum of digits. We can check the values of sum and num to ensure that each digit is being correctly extracted and added to the sum.
- After the Inner while Loop, where the sum of digits is reassigned back to num. We can check the value of num at this point to ensure that the sum of digits has been correctly assigned to num before the outer while loop runs again.
- Before the Final Output, so that it helps to verify whether the final value of num is equal to 1, indicating that the number is a Magic Number. We can check the value of num at this point to ensure the program is correctly deciding if the number is magical or not.

3. **Complete Executable Code:** In attached “Java Corrected Codes” folder.

Merge Sort Problem:

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

Solution:

- ➔ In the incorrect code, array+1 and array-1 do not make sense since you can't add or subtract integers to an array directly. In the corrected code, we simply pass the array as it is to the leftHalf and rightHalf functions.
- ➔ The incorrect code uses left++ and right--, which is incorrect as we don't want to modify the array references. We need to pass the left and right arrays as they are. In the correct code, we simply pass the left and right arrays to the merge method.

```
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);
    }
}
```

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

Solution:

- ➔ 4 Breakpoints needed
  - At the Start of the mergeSort Method, This will allow us to check the state of the array at the start of each recursive call. We can observe how the array is progressively broken down into smaller parts (left and right halves).
  - Inside the leftHalf Method, Here, we can check if the left part of the array is being correctly calculated and if the values are copied correctly. The left half should contain the first half of the current array passed in the recursive call.
  - Inside the rightHalf Method, we can check if the right part of the array is correctly calculated and copied. The right half should contain the second half of the current array passed in the recursive call.
  - At the Start of the merge Method, At this breakpoint, we can verify how the sorted left and right arrays are merged into the result array. We can check the intermediate values of the indices (i1 and i2) and the result array.

3. Complete Executable Code: In attached “Java Corrected Codes” folder.

Matrix Multiplication Problem:

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

Solution:

- ➔ Note, for the convenience and to enhance the code quality, the variable names are changed, where, m, n, p, q were used as variable names for matrix dimensions. More descriptive variable names like row_1, col_1, row_2, and col_2 respectively are used to represent matrix dimensions, making the code clearer.
- ➔ The indices are incorrectly modified with c-1, c-k, k-1, and k-d, which would cause array out-of-bound errors. Corrected the logic for matrix multiplication where the row from the first matrix and the column from the second matrix are multiplied correctly by using [c][k] and [k][d] respectively, as per standard matrix multiplication logic.
- ➔ Here, in the multiplication loop, the sum variable is not correctly handled.

```
for (int c = 0 ; c < row_1 ; c++ ) {
    for (int d = 0 ; d < col_2 ; d++ ) {
        sum = 0;
        for (int k = 0 ; k < col_1 ; k++ ) {
           sum += first[c][k] * second[k][d];
        }

        multiply[c][d] = sum;
    }
}
```

```
int row_1, col_1, row_2, col_2;

Scanner in = new Scanner(System.in);
System.out.println("Input: ");
System.out.println("Enter the number of rows and columns of first matrix");
row_1 = in.nextInt();
col_1 = in.nextInt();
```

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

Solution:

- ➔ 4 Breakpoints needed
    - First Matrix Element Input, to confirm that each element of the first matrix (first[][]) is being input correctly.
    - Compatibility Check for Matrix Multiplication, To ensure that the condition checking whether the matrices can be multiplied (i.e., if the number of columns in

the first matrix equals the number of rows in the second matrix) is being evaluated correctly.

- SecondMatrix Element Input, to confirm that each element of the first matrix (second[][]) is being input correctly.
- Matrix Multiplication Loop, to confirm that the indices (c, d, k) are being handled correctly, and that the partial products and the sum are computed as expected. Checking the correct values of first[c][k] and second[k][d] at each iteration, If the accumulation into sum is happening correctly, If the resulting multiply[c][d] is correctly assigned.

3. **Complete Executable Code:** In attached “Java Corrected Codes” folder.

Quadratic Probing Problem:

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

Solution:

- ➔ Handling of Hash Function, The correct code uses Math.abs() to ensure the hash code is always non-negative. This prevents issues with negative hash codes, which can cause invalid array indices.
- ➔ Fixing Probing Logic in Insert Function, The original code had a syntax error (i += (i + h / h--)). The correct code replaces this with a proper quadratic probing formula: i = (tmp + h * h) % maxSize, and increments h for each probe.
- ➔ Corrected variable names to improve readability in remove function. In the correct code, tmp1 and tmp2 are renamed to tmpKey and tmpVal for clarity.
- ➔ Closing the Scanner in the QuadraticProbingHashTest Class, The scan.close() method is used in the correct code to properly close the scanner and prevent resource leaks.

```
        qpht.printHashTable();
        System.out.print("Do you want to continue (Type y or n): ");
        ch = scan.next().charAt(0);
    } while (ch == 'Y' || ch == 'y');

    scan.close();
}
```

```
for (i = (i + h * h++) % maxSize; keys[i] != null; i = (i + h * h++) % maxSize) {
    String tmpKey = keys[i], tmpVal = vals[i];
    keys[i] = vals[i] = null;
    currentSize--;
    insert(tmpKey, tmpVal);
}
```

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

```
i = (tmp + h * h) % maxSize;
h++;
```

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

Solution:

➔ 10 Breakpoints needed

- After the keys and vals arrays are initialized in the constructor, ensuring that the hash table is being initialized with the correct capacity and that both keys and vals arrays are empty. We are verifying, maxSize matches the input, keys and vals arrays have been properly allocated.
- After tmp = hash(key)in insert function, checking the hash value for the input key and whether it falls within the correct range (0 <= hash < maxSize).
- Inside the if (keys[i] == null) condition in insert function, ensuring that the key-value pair is inserted correctly when the calculated position is empty (keys[i] == null). We are checking if the insertion position is correct based on the hash and that the currentSize is incremented after insertion.
- Inside the if (keys[i].equals(key)) condition in insert function, checking if an existing key is being replaced with a new value, as the hash table supports updating values for existing keys. We are verifying that the value for the key is updated, not duplicated.
- After the probe index i is updated, in insert function, observing how quadratic probing is handled by monitoring the value of i in each iteration.
- After int i = hash(key), in the get function, checking that the key hash value corresponds to the correct index.
- Inside the while (keys[i] != null) loop, in get function, verifying that the key is being found correctly, and if not, that the quadratic probing logic correctly moves to the next potential bucket
- After if (!contains(key)) return, in remove function, ensuring that the key is checked for existence before attempting removal. If the key doesn't exist, the method should return early.
- Inside the while (!key.equals(keys[i])) loop in remove function, to track the steps as the algorithm probes to find the key.
- Inside the for loop after rehashing starts, checking the rehashing process after a key is removed to avoid clustering.

3. Complete Executable Code: In attached “Java Corrected Codes” folder.

Sorting Problem:

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

Solution:

- ➔ The class name has a typo. The underscore (_) is removed, and the class name is corrected to follow Java naming conventions (no spaces).
- ➔ The original code mistakenly had a semicolon (;) at the end of the for loop, which terminated the loop prematurely, making the inner block not execute. The loop condition was also incorrect (i >= n should be i < n).
- ➔ The condition a[i] <= a[j] was incorrect for ascending order. It should be a[i] > a[j], so that when the element at index i is greater than j, they are swapped.

```
for (int i = 0; i < n; i++) {
    for (int j = i + 1; j < n; j++) {
        if (a[i] > a[j]) {
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
        }
    }
}
```

```
class AscendingOrder {
    public static void main(String[] args) {
```

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

Solution:

- ➔ 3 Breakpoints needed
  - At array input to verify whether the user inputs each element correctly and that the array a is being filled as expected. By inspecting a[i] for each iteration of the loop, we can confirm the values are properly assigned.
  - At Outer loop to confirm that the outer loop for sorting is correctly iterating through all elements. By stepping into the loop, we can observe whether each element of the array is being compared to the subsequent ones.
  - At Inner loop (comparison and swapping), to verify that the program correctly compares two elements (a[i] and a[j]) and swaps them if necessary. We can inspect the values of a[i] and a[j] before and after the swap to ensure the swapping is done as expected.

3. Complete Executable Code: In attached “Java Corrected Codes” folder.

Stack Implementation Problem:

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

Solution:

- ➔ The incorrect code decremented top (top--), which moves the stack pointer incorrectly. The correct behavior is to increment top (top++) when adding a new value to the stack.
- ➔ The incorrect code incremented top when popping, which was the opposite of what is needed. The correct behavior is to print the popped value (stack[top]) and then decrement top (top--) to remove the element from the stack.
- ➔ The incorrect code used a wrong condition in the for loop (i > top). This caused the loop to never execute, so the stack contents were never printed. The correct code uses i <= top to loop through all stack elements and also includes a condition to check if the stack is empty.
- ➔ The correct code adds a check to see if the stack is empty before attempting to display the stack elements, ensuring better handling of edge cases.

```
public void display() {
    if (isEmpty()) {
        System.out.println("\nStack is empty.");
    } else {
        System.out.print("\nStack contents: ");
        for (int i = 0; i <= top; i++) {
            System.out.print(stack[i] + " ");
        }
        System.out.println("\n");
    }
}
```

```
public void pop() {
    if (!isEmpty()) {
        System.out.println("Popped value: " + stack[top]);
        top--;
    } else {
        System.out.println("Can't pop...stack is empty");
    }
}
```

```
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;
    }
}
```

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

Solution:

➔ 3 Breakpoints needed

- At StackMethods Constructor, to ensure the stack is initialized properly with the correct size and the top is set to -1 and to verify that the stack's memory allocation (stack = new int[size]) is correct.
- At push Method, to check how the value of top changes when pushing elements onto the stack.
- At pop Method, to analyze how top changes and how the stack behaves when attempting to pop values.

3. **Complete Executable Code:** In attached “Java Corrected Codes” folder.

Tower of Hanoi Problem:

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

Solution:

- ➔ Incorrect Increment/Decrement Operators in doTowers Method Cal, Post-increment causes the argument to increment after the recursive call, which is incorrect for recursion logic, Post-increment causes the argument to increment after the recursive call, which is incorrect for recursion logic. Hence, replaced topN ++ with topN - 1 to decrement the number of disks being moved correctly and removed the inter--, from+1, and to+1 manipulations. The recursive call should use the correct rods (from, inter, to) in the order they are meant to represent.
- ➔ The first recursive call uses incorrect order for rods (from, to, inter), which changes the recursive logic. The second recursive call, as already pointed out, uses the wrong arguments with post-increment and post-decrement. Ensured the recursive calls are made with the correct order of rods and no increment or decrement operators should be used on topN of the rods (from, inter, to).

```
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);
    }
}
```

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

Solution:

- ➔ 5 Breakpoints needed
    - At the start of the doTowers() function, to check the initial call and verify the input parameters (number of disks, rods).
    - At the base case condition if (topN == 1), to ensure that the base case is being reached correctly and that disk 1 is being moved from the from rod to the to rod.
    - Just before the first recursive call doTowers(topN - 1, from, to, inter), to check how the recursive stack is growing, and to ensure that the parameters for the recursive call are correct.
    - After the first recursive call, at the System.out.println() statement for moving a disk, To verify that the move for the current disk (topN) is being printed correctly and that the logic for the recursive movement is working as intended.
    - Just before the second recursive call doTowers(topN - 1, inter, from, to), To verify the next phase of recursion, where the program moves the smaller disks (topN - 1) to the target rod after the largest disk has been moved.

3. Complete Executable Code: In attached “Java Corrected Codes” folder.

Part-I Program Inspection

❖ About Github Repository:

- **Github repository Name**: GoDot Engine
- **Github repository**: https://github.com/godotengine/godot

- **Repository Brief Description**: 2D and 3D cross-platform game Engine.
  - ➔ Godot is completely free and open source under the very permissive MIT license.
  - ➔ Godot Engine is a feature-packed, cross-platform game engine to create 2D and 3D games from a unified interface.
  - ➔ It provides a comprehensive set of common tools, so that users can focus on making games without having to reinvent the wheel.

- **Godot Engine Code Used**: https://github.com/godotengine/godot/blob/master/core/input/input_event.cpp
- **LOC**: 1528

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

❖ **Potential list of Errors and warnings:**

1)

Code Line Number	51, 56, 62, 68, 74
Code Statement	Ref<InputEvent>(const_cast<InputEvent *>(this))
Reason	The code uses Ref<InputEvent>(const_cast<InputEvent *>(this)), which could lead to issues if this is null.
Category	Category A: Data Reference Errors

2)

Code Line Number	69
Code Statement	return valid ? strength : 0.0f;
Reason	Functions get_action_strength default to returning 0.0f when valid is false, instead it could use only 0 or false (boolean)
Category	Category C: Computation Errors

3)

Code Line Number	75
Code Statement	return valid ? strength : 0.0f;
Reason	Functions get_action_raw_strength default to returning 0.0f when valid is false, instead it could use only 0 or false (boolean)
Category	Category C: Computation Errors

4)

Code Line Number	55, 61
Code Statement	bool pressed_state
Reason	This could lead to undefined behavior if not initialized before use.
Category	Category B: Data-Declaration Errors

5)

Code Line Number	232, 233
Code Statement	BitField<KeyModifierMask>
Reason	If the BitField<KeyModifierMask> is not appropriately instantiated and used, this could lead to runtime errors due to type mismatches.
Category	Category B: Data-Declaration Errors

6)

Code Line Number	712
Code Statement	button_index = p_index;
Reason	The setter methods do not validate their inputs. if an invalid MouseButton is set, it might lead to undefined behavior.
Category	Category H: Other Checks

Apart from the above potential inspection for warnings or checks, there is a common lack for the input error-handling. For many functions, the values for the inputs are not checked to be within a certain bound or to be from a certain specific set of values.

For example:

Code line: 1327

```
void InputEventScreenTouch::set_index(int p_index) { index
        = p_index;
}
```

Here, the p_index must be checked to be within some bound before assigning it to the index variable.

Code line: 1339

```
Vector2 InputEventScreenTouch::get_position() const {
        return pos;
}
```

Here, the function must check that the value for pos must not be NULL.

And there are many more…

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

For this program, Functional Inspection and Code Analysis Inspection seem to be the most effective:

- Category A: Data reference error checking is essential for preventing crashes, memory leaks, and data corruption by ensuring that memory locations being accessed are valid. It helps maintain application stability and data integrity, ensuring reliable performance.
- Category C: Computation error checking is important to ensure accurate results by validating mathematical operations and preventing issues like overflows, underflows, and division by zero. It helps maintain the correctness of calculations and avoids unpredictable behavior in the program.
- Category H: Other checks are useful because it helps ensure that each function behaves as expected, e.g., handling edge cases, validating inputs, and ensuring data consistency.
- Code Analysis Inspection helps identify structural issues, such as redundancy, lack of comments, and overall organization. The lack of input validation across the program is something that stands out in this case, which Code Analysis Inspection would likely flag early on.

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

Errors that program inspection might not effectively detect the following:

- **Runtime Errors**: Here the code depends on external hardware (like MIDI devices), errors related to the actual interaction with hardware might not be easily identified through code inspection. Because, invalid or unexpected data from a MIDI device could still pass through unchecked without triggering errors during code inspection.
- **Concurrency Issues**: If the code uses asynchronous functions, race conditions and deadlocks might not be obvious through code inspection alone.
- **Performance Issues**: Without execution profiling, issues like inefficient algorithms, memory leaks, or unnecessary object allocations would not be obvious.
- **Boundary Case Errors**: For example, sending values right at the edges of acceptable MIDI ranges could cause issues that aren't immediately apparent from just reading the code.

4) Is the program inspection technique worth applicable?

Solution:

Yes, program inspection techniques are worth applying for this type of project, for several reasons:

- Code inspection allows early identification of potential bugs before the software reaches the testing or deployment stages. Issues like missing input validation, null checks, and code duplication can be caught early.
- Regular inspections lead to better code organization, clearer logic, and improved maintainability. For instance, refactoring redundant code or adding necessary input validations.
- Code inspection gives us a deeper understanding of the system's internal workings, which can help us identify potential areas for optimization.

Part-III Static Analysis Tools

❖ About Static Analysis:

- **Static Analysis Tool used**: CppCheck
- **Language Analyzed**: Cpp

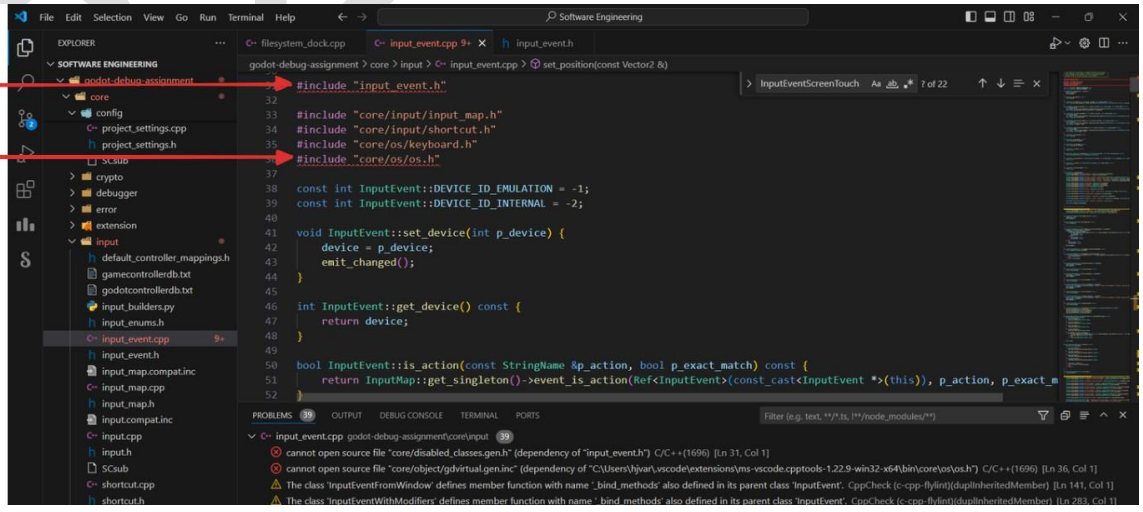
Errors Identified	Warnings Identified	Info/optional-ways Identified
2	37	0

➔ Some of the warnings and errors snippets are shown below.
➔ And attached it the link for the static_analysis_report.xls.

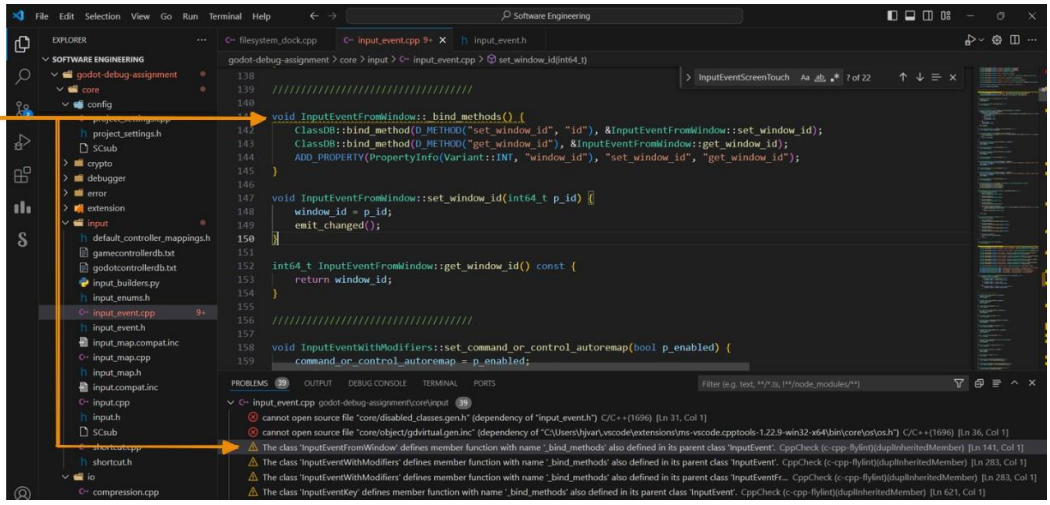
Google Drive:
https://drive.google.com/drive/folders/1s3pm94dSlycOeTItZpExsabeK_BZQelR?usp=sharing

❖ Static Analysis Output:

Error: Cannot open source File



Warning: Defines _bind_methods which is defined in its parent class



Warning: Defines _bind_methods which is defined in its parent class