Lab 7: Software Engineering

Program Inspection, Debugging and Static Analysis

Nandini Mandaviya (202201487)

Q.1 Manual static analysis

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

1. Error in File Handling:

- Issue: In the load() function, the file is opened using fs.open() and immediately read without checking if the file was successfully opened. Although there is a condition to check if (!fs.isOpened()), it's done after the node is accessed, which could lead to accessing uninitialized memory or triggering exceptions.
- Potential Fix: Add a file-open check immediately after trying to open the file and before accessing node.

```
fs.open( cascadeDirName + buf + ".xml", FileStorage::READ );

if ( !fs.isOpened() ) {

   cerr << "Error: Unable to open file: " << cascadeDirName + buf + ".xml" << endl;

   return false;

}

node = fs.getFirstTopLevelNode();
```

2. Potential Memory Leak:

- Issue: In the load() function, memory is allocated for Ptr<CvCascadeBoost> tempStage = makePtr<CvCascadeBoost>(); but if the reading process fails (!tempStage->read(node, featureEvaluator, *stageParams)), there's no proper release or cleanup of the previously allocated memory. This could lead to a memory leak if the function is called multiple times without a full cleanup.
- Potential Fix: Make sure to release memory explicitly in case of an error. Using smart pointers (Ptr) mitigates this issue somewhat, but handling failure scenarios properly is still necessary.

3. Uninitialized Variables:

- Issue: Variables like numStages, numPos, and numNeg are used without any explicit initialization before being referenced in calculations or loops. This could lead to undefined behavior if these values are not properly set before usage.
- Potential Fix: Ensure proper initialization or validation of these variables before they are used.

```
int numStages = 0;

int numPos = 0;

int numNeg = 0;
```

4. Inadequate Error Handling in scanAttr():

- Issue: The function scanAttr() sets res = false when an unrecognized parameter is encountered, but it doesn't provide any feedback or error logging. This could lead to silent failures, making debugging harder.
- Potential Fix: Log an error message when an invalid parameter is encountered to give the programmer feedback.

```
else {

   res = false;

   cerr << "Error: Unrecognized parameter: " << prmName << endl;

}
```

5. Possible Off-by-One Error in Loops:

- Issue: In multiple places, such as loops over stageTypes and featureTypes, array indices may have an off-by-one error. For instance, the loop in printDefaults() checks i < (int)(sizeof(stageTypes)/sizeof(stageTypes[0])), but this calculation may be incorrect if the arrays are not correctly sized or initialized. Also, stageType could index out of bounds if the stage number is incorrect or exceeds the array size.
- Potential Fix: Add proper bounds checking to ensure that i does not exceed the array size, and validate the index before accessing elements.

```
if ( stageType < 0 || stageType >= (int)(sizeof(stageTypes)/sizeof(stageTypes[0])) ) {

   cerr << "Error: stageType out of bounds!" << endl;

}
```

6. Implicit Type Conversion Errors:

- Issue: In various parts of the code, implicit type conversions between int, size_t, and unsigned int occur. This could lead to truncation or unexpected results, particularly in loops. For instance, winSize.width is likely an integer, but in some conditions, it may be passed to functions expecting different types.
- Potential Fix: Use explicit casting

Here is a list of 15 potential errors identified in the program, covering various areas such as memory management, file handling, initialization, and type safety:

1. Uninitialized Variables (Data Reference Error)

- Issue: Variables like numStages, numPos, and numNeg are used without being explicitly initialized.
- Fix: Initialize them to a default value before use.

```
int numStages = 0;

int numPos = 0;

int numNeg = 0;
```

2. No File Opening Check (Input/Output Error)

- Issue: In the load() function, there is no check for successful file opening immediately after trying to open a file.
- Fix: Add a file-open check before accessing any file-related nodes.

```
if (!fs.isOpened()) {

    cerr << "Error: Unable to open file" << endl;

    return false;

}
```

3. Memory Leak on Failure (Memory Management Error)

- Issue: Memory is allocated for CvCascadeBoost, but there's no cleanup if the read() operation fails, leading to a memory leak.
- Fix: Use smart pointers (Ptr), or ensure proper cleanup on failure.

```
if (!tempStage->read(node, featureEvaluator, *stageParams)) {
    return false; // Should also handle memory cleanup here.
}
```

4. Potential Dangling Pointer (Data Reference Error)

- Issue: There is a possibility of dangling pointers if memory is allocated dynamically but not freed properly, particularly with tempStage.
- Fix: Ensure that the allocated memory is always freed or use smart pointers.

5. Improper Array Bounds Check (Data Reference Error)

- Issue: Array indexing (stageTypes, featureTypes) does not ensure that the index stays within bounds, risking access violations.
- Fix: Add explicit bounds checks before accessing array elements.

```
if (stageType < 0 || stageType >= sizeof(stageTypes)/sizeof(stageTypes[0])) {
    cerr << "Error: Array index out of bounds" << endl;
}
```

6. Implicit Type Conversion (Computation Error)

- Issue: Implicit type conversion between int, size_t, and unsigned int could lead to data loss or unexpected behavior.
- Fix: Use explicit casting for better control.

```
int value = (int) someSizeTValue; // Ensure explicit casting.
```

7. Off-by-One Error in Loops (Control Flow Error)

- Issue: Loops might have an off-by-one error due to incorrect bounds, especially in conditions like i <= array_size.
- Fix: Make sure to use i < array_size for zero-based indexing.

8. Insufficient Error Handling in scanAttr() (Data Reference Error)

- Issue: When an unrecognized parameter is encountered, the function sets res = false but provides no feedback.
- Fix: Add error logging to report unrecognized parameters.

```
if (!recognizedParam) {
    res = false;
    cerr << "Error: Unrecognized parameter" << endl;
}
```

9. No Check for Null Pointers (Data Reference Error)

- Issue: There are places where pointers are used without checking if they are nullptr. For instance, after malloc or new operations.
- Fix: Always check if the pointer is nullptr before dereferencing it.

```
if (ptr == nullptr) {
    cerr << "Error: Memory allocation failed" << endl;
}
```

10. Unchecked Return Values (Input/Output Error)

- Issue: Functions like read() return values indicating failure, but these are not consistently checked.
- Fix: Always check the return value and handle errors accordingly.

11. Global Variables Used Without Synchronization (Concurrency Error)

- Issue: If the program is executed in a multi-threaded environment, global variables could cause race conditions.
- Fix: Use proper synchronization (e.g., mutex locks) when accessing global variables.

12. Possible Integer Overflow (Computation Error)

- Issue: Integer overflow could occur during arithmetic operations (e.g., numStages++ or large loop counters).
- Fix: Add checks to prevent overflow.

```
if (numStages == INT_MAX) {
    cerr << "Error: numStages overflow" << endl;
}
```

13. Incorrect Use of Boolean Expressions (Comparison Error)

- Issue: The expression 2 < i < 10 does not work as intended. It should be split into two comparisons.
- Fix: Break down the comparison into two parts.

```
if (2 < i && i < 10) {
   // Correctly evaluate the condition.
}
```

14. Unclear Variable Names (Data Declaration Error)

- Issue: Variables like fs, buf, and node are not self-explanatory, which could lead to confusion or misuse.
- Fix: Use more descriptive variable names to make the code clearer.

```
FileStorage fileStorageHandler;
```

15. No Handling for Division by Zero (Computation Error)

- Issue: If there are any divisions in the code (though not evident in the snippet), there are no checks to prevent division by zero.
- Fix: Ensure the divisor is never zero before performing division.

```
if (divisor == 0) {
   cerr << "Error: Division by zero" << endl;
   return false;
}
```

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

- Category A (Data Reference Errors): Checking if variables are initialized and if references (especially pointers) are valid is crucial in this C++ code, particularly for memory and file handling. This category would catch dangling references and uninitialised values.

3. Which type of error are you unable to identify using the program inspection?

- Dynamic behaviour and runtime errors: Issues like memory leaks, heap corruption, or multi-threaded race conditions are difficult to detect just by inspection.

4. Is the program inspection technique worth applying?

- Yes, inspection techniques help catch several logical and potential runtime errors early in development. Even though some errors might not be evident by mere inspection, addressing common issues like uninitialized variables, file handling errors, and bounds checking prevents a majority of common bugs.

Q.2 Debugging

1. Armstrong Number

1. Incorrect digit extraction:

- In the statement remainder = num / 10, this is dividing the number by 10, but it should extract the last digit of the number using num % 10 instead.
- The current logic is incorrectly dividing the number instead of extracting the digits. It will not correctly sum the cube of the digits.

2. Incorrect update of num:

- In the loop, num = num % 10 is incorrect for moving to the next digit. It should reduce the number by dividing it by 10 to remove the last digit (num = num / 10).

```
class Armstrong {

  public static void main(String args[]) {

    int num = Integer.parseInt(args[0]);

    int n = num;  // Save original number for comparison

    int check = 0, remainder;


    while (num > 0) {

      remainder = num % 10;  // Extract the last digit

      check = check + (int) Math.pow(remainder, 3);  // Add the cube of the digit

      num = num / 10;  // Remove the last digit from num

    }
```

```
        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**

- **GCD Calculation Error**: The current condition while(a % b == 0) is incorrect; it should be while(a % b != 0) to correctly implement the Euclidean algorithm.
- **LCM Calculation Error**: The condition in the LCM method if(a % x != 0 && a % y != 0) should be if(a % x == 0 && a % y == 0) to check if both numbers divide a.

Corrected LCM Method:

```
static int lcm(int x, int y)
{
    int a;
    a = (x > y) ? x : y; // a is greater number
    while(true)
    {
        if(a % x == 0 && a % y == 0)
            return a;
        ++a;
    }
}
```

Knapsack Program

- **Index Increment Error**: The line int option1 = opt[n++][w]; increments n, which causes incorrect behavior. It should be opt[n][w].

- **Profit Calculation Error**: The condition profit[n-2] should be profit[n], as you're evaluating the current item n.
- **Weight Check Error**: The line if (weight[n] > w) should allow items with weight less than or equal to w. Fix the condition.

  Corrected Knapsack Logic:

  ```
  int option1 = opt[n-1][w]; // Correct the n increment

  int option2 = Integer.MIN_VALUE;

  if (weight[n] <= w) option2 = profit[n] + opt[n-1][w-weight[n]];
  ```

  Magic Number Program

- **Sum Calculation Error**: The line while(sum == 0) should be while(sum > 0) to ensure the digits are processed correctly.
- **Multiplication and Sum Logic Error**: The line s = s * (sum / 10) is incorrect; it should sum up digits rather than multiply.

  Corrected Magic Number Logic:

  ```
  while(sum > 0)

  {

    s = s + (sum % 10); // Summing digits

    sum = sum / 10;

  }
  ```

  Merge Sort Program

- **Array Partitioning Error**: In leftHalf(array+1) and rightHalf(array-1), you're attempting to shift the array index, which is incorrect. It should simply pass the array.
- **Merge Logic Error**: The left++ and right-- in the merge() method are incorrect. These should just pass left and right as arguments.

  Corrected Merge Sort Partitioning:

  ```
  int[] left = leftHalf(array);

  int[] right = rightHalf(array);
  ```

  Matrix Multiplication Program

- **Index Error in Matrix Multiplication**: The line sum = sum + first[c-1][c-k] * second[k-1][k-d]; contains incorrect indices. It should be first[c][k] * second[k][d].
- **Matrix Index Bound Error**: Ensure indices are within bounds during multiplication.

  Corrected Matrix Multiplication Logic:

  sum = sum + first[c][k] * second[k][d]; // Corrected index access

  Quadratic Probing Hash Table

- **Insert Method Error**: The statement i += (i + h / h--) % maxSize; is incorrect. It should use quadratic probing properly, like i = (i + h * h++) % maxSize.
- **Size Decrement Error**: In the remove() method, currentSize-- is decremented twice, which will cause incorrect size tracking.

  Corrected Insert and Remove Logic:

  i = (i + h * h++) % maxSize; // Quadratic probing

  currentSize--; // Only decrement size once

Tower of Hanoi:

The Tower of Hanoi program should recursively move disks between three rods, but there are syntax and logical errors in the code:

1. **Increment Error**:
   - In the doTowers function, the line doTowers(topN ++, inter--, from+1, to+1) incorrectly uses ++ and -- which modifies the values unnecessarily. It should just pass the updated arguments directly to follow the recursive logic.

   Fix: Instead of modifying the variables like topN++ or inter--, use just topN-1, inter, etc.
2. **Recursive Call Logic**:
   - The parameters being passed in the recursive call for the second call doTowers(topN ++, inter--, from+1, to+1) are not correctly updating the rod references. The logic should follow the correct rod movements.

   Fixed Code:

```
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);  // Move top disks from 'from' to 'inter'

      System.out.println("Disk " + topN + " from " + from + " to " + to);

      doTowers(topN - 1, inter, from, to);  // Move disks from 'inter' to 'to'

    }

  }

}
```

Stack Implementation:

There are multiple issues with the **StackMethods** class:

1. **Push Method**:

    - The push method decrements top before inserting the element. It should increment top to push elements properly. Also, it doesn’t check if the stack is full properly since it doesn’t increment the top value correctly.

Fix: Increment top when inserting an element.

2. **Pop Method**:

    - The pop method increments top instead of decrementing it. When popping elements from the stack, top should be decremented.

Fix: Decrement top when popping an element.

3. **Display Method**:

    - The for loop in the display method uses the wrong comparison (i > top). It should be i <= top to display the stack correctly.

Fix: Use the correct loop condition to iterate and display the stack.

Fixed Code:

```
import java.util.Arrays;


public class StackMethods {

   private int top;

   int size;

   int[] stack;


   public StackMethods(int arraySize) {

      size = arraySize;

      stack = new int[size];

      top = -1;

   }


   public void push(int value) {

      if (top == size - 1) {

         System.out.println("Stack is full, can't push a value");

      } else {

         top++;  // Increment top before inserting

         stack[top] = value;

      }

   }


   public void pop() {
        if (!isEmpty()) {
            top--;  // Decrement top after popping
        } 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++) {  // Correct condition to display elements
            System.out.print(stack[i] + " ");
        }
        System.out.println();
    }
}

public class StackReviseDemo {
    public static void main(String[] args) {
        StackMethods newStack = new StackMethods(5);
        newStack.push(10);
        newStack.push(1);
        newStack.push(50);
        newStack.push(20);

        newStack.push(90);


        newStack.display();

        newStack.pop();

        newStack.pop();

        newStack.pop();

        newStack.pop();

        newStack.display();

    }

}
```

Corrected Output for Stack Implementation:

```
10 1 50 20 90

10
```