IT-314

Amin Fenil

202201236

LAB-7

Program Insepection:

Category A: Data Reference Errors

Uninitialized Variables: Ensure that each variable is initialized before being used in any operation.

In the code: The variables pci, rDst, and rOff might face uninitialized issues if not assigned values before their use in operations like file output or calculations.

Array Boundaries: Check that the indices used for arrays are within bounds.

In the code: Look at arrays like szMonth and ensure all subscripts are within the array's declared length.

Pointer References: Ensure memory is properly allocated and that there is no dangling reference.

In the code: Memory referencing issues may arise, particularly if pointers to file or szFileOut are not handled correctly.

Aliased Memory: The code seems clear, though languages like FORTRAN may introduce errors here. This doesn't directly apply here.

Q1: Errors Identified: Potential uninitialized variables like rDst, pci in the context of file operations.

Q2: Most Effective Inspection Category: Data reference checks are critical because uninitialized values are a common issue.

Q3: Missing Error Identification: The checks might not capture unhandled null pointer dereferencing.

Q4: Applicability: Yes, this is critical as uninitialized variables cause significant runtime errors.

Category B: Data-Declaration Errors

Variable Declarations: All variables should be explicitly declared.

In the code: Variables such as pci, rDst are declared, but their initialization isn't always obvious.

Attributes: Ensure default attributes or types are correctly understood.

In the code: No errors observed with Java-like initialization defaults.

Q1: Errors Identified: Potential variable shadowing issues or mismatches of variable types could exist if variables are redefined.

Q2: Most Effective Inspection Category: Data declaration checks prevent shadowing and mismatches.

Q3: Missing Error Identification: Handling global and local scope inconsistencies might be missed.

Q4: Applicability: Essential for catching type mismatches in complex data structures.

Category C: Computation Errors

Mixed-Type Computations: Ensure no incorrect operations between types (e.g., integer division).

In the code: Integer division of pci->lon / 15.0 could introduce unintended precision issues.

Overflow/Underflow: Handle potential overflows in calculations.

In the code: Ensure precision remains intact when handling rOff or RAbs.

Q1: Errors Identified: Integer division precision issues could arise if not properly handled.

Q2: Most Effective Inspection Category: Computation error checks, especially with floating-point calculations.

Q3: Missing Error Identification: Overflow issues in edge cases (e.g., large coordinate values).

Q4: Applicability: Yes, since floating-point precision and integer handling are critical.

Category D: Comparison Errors

Mixed-Type Comparisons: Ensure no comparison between different types, such as strings and numbers.

In the code: Comparisons of pci->dst and other types seem correct, but FBetween(pci->yea, -9999, 99999) could be risky without clear type casting.

Boolean Logic: Check Boolean expressions for correctness.

In the code: Boolean expressions like pci->zon == zonLAT are clear.

Q1: Errors Identified: Risk of incorrect boolean evaluations (e.g., comparisons of pci->lon without explicit typecasting).

Q2: Most Effective Inspection Category: Boolean and comparison operations.

Q3: Missing Error Identification: Complex chained comparisons might still be risky without further review.

Q4: Applicability: Yes, necessary for ensuring all Boolean logic behaves as expected.

Category E: Control-Flow Errors

Loop Termination: Ensure that all loops terminate correctly.

In the code: Loops such as for (i = 0; i < iMax; i++) are structured correctly but need review for proper termination.

Q1: Errors Identified: No explicit off-by-one errors, but loop conditions could fail if iMax is incorrectly calculated.

Q2: Most Effective Inspection Category: Loops and termination checks.

Q3: Missing Error Identification: Off-by-one errors in nested loops might still be tricky to catch.

Q4: Applicability: Critical, as improper loop termination can lead to infinite loops.

Category F: Interface Errors

Parameter Matching: Ensure that all function arguments match the expected parameters.

In the code: Ensure consistency when calling fprin and szFile.

Q1: Errors Identified: No parameter mismatches observed.

Q2: Most Effective Inspection Category: Interface checks, especially in file handling.

Q3: Missing Error Identification: Complex argument mismatches, especially in recursive calls.

Q4: Applicability: Yes, necessary to prevent mismatches between calling and receiving functions.

Category G: Input/Output Errors

File Handling: Ensure that all files are opened and closed correctly.

In the code: The file pointer file is closed in all cases, ensuring no memory leak.

I/O Errors: Properly handle file opening and closing.

In the code: The error messages for file opening (if (file == NULL)) are present but should be carefully tested for edge cases.

Q1: Errors Identified: No critical file handling errors found.

Q2: Most Effective Inspection Category: File handling, especially in the context of file opening/closing.

Q3: Missing Error Identification: Some advanced edge cases might cause issues during file reads.

Q4: Applicability: Essential, as proper file handling prevents resource leakage.

Category H: Other Checks

Warning Messages: Inspect all warning messages to ensure correctness.

In the code: Handle all compiler warnings, especially related to typecasting or file handling.

Q1: Errors Identified: No obvious warning-triggering code sections.

Q2: Most Effective Inspection Category: Miscellaneous checks to ensure no overlooked minor issues.

Q3: Missing Error Identification: Some edge cases not covered (e.g., rare memory overflow conditions).

Q4: Applicability: Yes, these checks add robustness to the code.

Code link:

https://github.com/CruiserOne/Astrolog/blob/85b4381efd8cb2254c07d6605f15f26f2c8317da/io.cpp#L1683

SECTION – II

Armstrong Number: Errors and Fixes

1. How many errors are there in the program?
   →There are 2 errors in the program.

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

→Steps Taken to Fix the Errors:

Error 1: The division and modulus operations were mistakenly swapped in the while loop.

Fix: Ensure that the modulus operation is used to retrieve the last digit of the number, while the division operation reduces the number for the next iteration.

Error 2: The check variable was not properly accumulating the sum.

Fix: Correct the logic to ensure that the check variable accurately reflects the sum of each digit raised to the power of the total number of digits.

```
class Armstrong {
    public static void main(String args[]) {
        int num = Integer.parseInt(args[0]);
        int n = num; // use to check at 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");
    }
}
```

GCD and LCM: Errors and Fixes

1. How many errors are there in the program?
   →There is 1 error in the program.

2. How many breakpoints do you need to fix this error?
   →We need 1 breakpoint to fix this error.

→Steps Taken to Fix the Error:

Error: The condition in the while loop of the GCD method was incorrect.

Fix: Change the condition to `while (a % b != 0)` instead of `while (a % b == 0)`. This ensures the loop continues until the remainder is zero, allowing the GCD to be calculated correctly.

```java
import java.util.Scanner;

public class GCD_LCM {
    static int gcd(int x, int y) {
        int r = 0, a, b;
        a = (x > y) ? x : y; // a is greater number
        b = (x < y) ? x : y; // b is smaller number

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

```
        return r;
    }

    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;
        }
    }

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

Knapsack Problem: Errors and Fixes

1. How many errors are there in the program?
   →There are 3 errors in the program.

2. How many breakpoints do you need to fix these errors?
   →We need 2 breakpoints to fix these errors.

→Steps Taken to Fix the Errors:

Error 1: The condition in the "take item n" case was incorrect.

Fix: Change ‘if (weight[n] > w)’ to ‘if (weight[n] <= w)’ to ensure the profit is calculated when the item can be included.

Error 2: The profit calculation was incorrect.

Fix: Change ‘profit[n-2]’ to ‘profit[n]’ to ensure the correct profit value is used.

Error 3: In the "don't take item n" case, the indexing was incorrect.

Fix: Change ‘opt[n++][w]’ to ‘opt[n-1][w]’ to properly index the items.

```java
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);
        }

        // opt[n][w] = max profit of packing items 1..n
with weight limit w
        // sol[n][w] = does opt solution to pack items
1..n with weight limit w include item n?
        int[][] opt = new int[N+1][W+1];
        boolean[][] sol = new boolean[N+1][W+1];

        for (int n = 1; n <= N; n++) {
            for (int w = 1; w <= W; w++) {

                // don't take item n
                int option1 = opt[n-1][w];

                // take item n
                int option2 = Integer.MIN_VALUE;
                if (weight[n] <= w) option2 = profit[n]
+ opt[n-1][w-weight[n]];

                // select better of two options
                opt[n][w] = Math.max(option1, option2);
                sol[n][w] = (option2 > option1);
```

```
            }
        }

        // determine which items to take
        boolean[] take = new boolean[N+1];
        for (int n = N, w = W; n > 0; n--) {
            if (sol[n][w]) { take[n] = true;  w = w -
weight[n]; }
            else           { take[n] = false;
}
        }

        // print results
        System.out.println("item" + "\t" + "profit" +
"\t" + "weight" + "\t" + "take");
        for (int n = 1; n <= N; n++) {
            System.out.println(n + "\t" + profit[n] +
"\t" + weight[n] + "\t" + take[n]);
        }
    }
}
```

Magic Number Check: Errors and Fixes

1. How many errors are there in the program?
   →There are 3 errors in the program.

2. How many breakpoints do you need to fix these errors?
   →We need 1 breakpoint to fix these errors.

→Steps Taken to Fix the Errors:

Error 1: The condition in the inner while loop was wrong.

Fix: Change ‘while(sum == 0)’ to ‘while(sum != 0)’ so the loop processes the digits properly.

Error 2: The way s was calculated in the inner loop was incorrect.

Fix: Change ‘s = s * (sum / 10)’ to ‘s = s + (sum % 10)’ to correctly add the digits.

Error 3: The order of steps in the inner while loop was wrong.

Fix: Reorder the steps to ‘s = s + (sum % 10); sum = sum / 10;’ so the digits are added correctly.

```
import java.util.*;
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)
            {
                s=s+(sum%10);
                sum=sum/10;
            }
            num=s;
        }
        if(num==1)
        {
            System.out.println(n+" is a Magic
Number.");
        }
        else
        {
            System.out.println(n+" is not a Magic
Number.");
        }
    }
}
```

Merge Sort: Errors and Fixes

1. How many errors are there in the program?
   →There are 3 errors in the program.

2. How many breakpoints do you need to fix these errors?
   →We need 2 breakpoints to fix these errors.

→Steps Taken to Fix the Errors:

Error 1: Incorrect array indexing when splitting the array in mergeSort.

Fix: Change 'int[] left = leftHalf(array + 1)' to 'int[] left = leftHalf(array)' and 'int[] right = rightHalf(array - 1)' to 'int[] right = rightHalf(array)' to pass the array correctly without skipping elements.

Error 2: Incorrect use of increment (++) and decrement (--) in merge.

Fix: Remove the ++ and -- from merge(array, left++, right--) and use merge(array, left, right) to pass the arrays directly.

Error 3: The merge function is accessing elements beyond the array bounds.

Fix: Adjust the indexing in the merge logic to make sure the array boundaries are respected.

```java
import java.util.*;

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 + 1) / 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++;
            }
```

```
        }
    }
}
```

Matrix Multiplication: Errors and Fixes

1. How many errors are there in the program?
   →There is 1 error in the program.

2. How many breakpoints do you need to fix this error?
   →We need 1 breakpoint to fix these errors.

<u>→Steps Taken to Fix the Errors:</u>

Error: Incorrect array indexing in the matrix multiplication logic.

Fix: Change first[c-1][c-k] and second[k-1][k-d] to first[c][k] and second[k][d]. This ensures that the correct elements of the matrices are referenced during multiplication.

```
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);
        }
    }
}
```

Quadratic Probing Hash Table

Errors and Fixes:

1.How many errors are in the program?

→There is 1 error in the program.

2.How many breakpoints are needed to fix this error?

→1 breakpoint* is needed to fix the error.

<u>→Steps Taken to Fix the Errors:</u>

Error: In the insert method, the line i += (i + h / h--) % maxSize; is incorrect.

Fix: Change it to i = (i + h * h++) % maxSize; to properly use quadratic probing logic.

```java
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; // Fixed
quadratic probing
        } 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;
        }
        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;

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

        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);
        }
```

```
    }

    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:
                    System.out.println("Size = " +
qpht.getSize());
                    break;
                default:
                    System.out.println("Wrong Entry
```

```
\n");
                    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');
    }
}
```

Sorting Array

Errors and Fixes:

1.How many errors are there in the program?

→There are 2 errors in the program.

2.How many breakpoints do you need to fix this error?

→We need 2 breakpoints to fix these errors.

→Steps Taken to Fix the Errors:

Error 1: The loop condition for (int i = 0; i >= n; i++) is wrong.

Fix: Change it to for (int i = 0; i < n; i++) to correctly loop through the array.

Error 2: The condition in the inner loop if (a[i] <= a[j]) is reversed.

Fix: Change it to if (a[i] > a[j]) to correctly sort the array in ascending order.

```java
import java.util.Scanner;

public class Ascending_Order {
    public static void main(String[] args) {
        int n, temp;
        Scanner s = new Scanner(System.in);
        System.out.print("Enter no. of elements you
want in 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();
        }

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

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

Stack Implementation (from Stack Implementation.txt)(Stack Implementation)

Errors and Fixes:

1.How many errors are there in the program?

→There are 2 errors in the program.

2.How many breakpoints do you need to fix this error?

→We need 2 breakpoints to fix these errors.

→Steps Taken to Fix the Errors:

Errors and Fixes

Error 1: In the push method, the line top-- is incorrect.

Fix: Change it to top++ to correctly move the stack pointer up.

Error 2: In the display method, the loop condition for (int i = 0; i > top; i++) is wrong.

Fix: Change it to for (int i = 0; i <= top; i++) to make sure all elements are displayed.

```java
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++; // Fixed increment
            stack[top] = value;
        }
    }

    public void pop() {
        if (!isEmpty()) {
            top--;
        } 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();
    }
}

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();
    }
}
```

Tower of Hanoi (from Tower of Hanoi.txt)(Tower of Hanoi)

Errors and Fixes:

1.How many errors are there in the program?

→There is 1 error in the program.

2.How many breakpoints do you need to fix this error?

→We need 1 breakpoint to fix this error

→Steps Taken to Fix the Errors:

Error: In the recursive call doTowers(topN ++, inter--, from + 1, to + 1);, the increments and decrements for the variables are wrong.

Fix: Change it to doTowers(topN - 1, inter, from, to); to ensure proper recursion and follow the Tower of Hanoi rules.

```java
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); //
Corrected recursive call
        }
    }
}
```