LAB-REPORT 7

Software Engineering

Shreya Patel - 202201460

We are taking the following code of 1500 from github about a railway management system:
https://github.com/iamayan2011/Projects/blob/main/Railway%20Managment%20System%20(OOPS)/railway.cpp

QUESTION 1:

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

Here are the identified errors:

Improper Main Function Declaration: The function is declared as `void main()`, which is incorrect in C++. The standard main function should be defined as `int main()`. This can lead to undefined behavior.

Missing Return Statement: The main function (when corrected) should return an integer value. If `int main()` is used, it should end with a return statement (e.g., `return 0;`).

Infinite Recursion on Login Failure: When the user enters the wrong Admin ID or Password, the program recursively calls `firstPage()`, which can lead to a stack overflow if the user repeatedly enters incorrect credentials.

File Handling Issues: When opening `user.txt`, there is no check to verify if the file was opened successfully. This could result in reading from a non-existent or inaccessible file.

End-of-File Check on File Read: The loop using `while (!f1.eof())` can lead to accessing invalid data since the end-of-file (EOF) condition is checked after attempting to read. It would be better to check the read operation's success directly.

Uninitialized Variables: In the case of user login, if the user fails to log in, the variables `cuid` and `cupass` will still hold whatever values were in memory prior to initialization, which could lead to unexpected behavior.

Potential Infinite Loop: The program may not exit gracefully if the user chooses to return to the first page after entering wrong credentials; there's no way to exit the program cleanly if the admin ID or password is incorrect.

Password Handling: The program uses `cin` to input passwords, which exposes sensitive information on the console. This is a security risk.

Lack of Input Validation: There is no check on user input for the number of passengers, customer IDs, or other inputs, which could lead to unexpected behavior or crashes if invalid data is entered.

Hardcoded Train Information: While hardcoding is acceptable for a demo, consider making the train data dynamic or loading from an external source for maintainability and flexibility.

Redundant Code: The booking logic is very repetitive. Using functions to handle booking and charge assignment could reduce redundancy and improve maintainability.

User Experience Improvements: The message prompts and structure could be improved for clarity and to enhance user interaction.

Using `mainMenu()` Without Definition: The function `mainMenu()` is called multiple times, but it is not defined in the provided code. This will result in a linker error.

Static Array Size: The arrays `name`, `gender`, `bp`, `age`, and `cId` are declared with a fixed size of 6. If `n` exceeds 6, it will lead to out-of-bounds access, which is undefined behavior.

Input Validation for Passenger Count: The program only checks if `n > 6` but does not handle the case when a user inputs a negative number or zero. This can lead to an infinite loop or garbage output when accessing the arrays.

Random Number Generation: The random number generation for `pnr` uses `srand(time(NULL))` every time `information()` is called. This can lead to the same value being generated if the function is called multiple times in quick succession. It would be better to call `srand()` once, typically at the start of the program.

No Return Type for `information()`: The `information()` function does not specify a return type. Since it's used within the class, it should ideally return a type (like `void`), which is implicitly understood, but explicit declaration is a good practice.

Misleading Message for Charges: After booking a ticket, the program prints "YOU CAN GO BACK TO MENU AND TAKE THE TICKET" without actually providing a way to take the ticket or go back. The message can be confusing.

Inconsistent Data Input: The user input for `gender`, `bp`, and `age` does not include any validation, meaning users could input incorrect data types or values. For example, age should be an integer, but the code does not check if the input is indeed an integer.

Misleading Message for Charges: After booking a ticket, the program prints "YOU CAN GO BACK TO MENU AND TAKE THE TICKET" without actually providing a way to take the ticket or go back. The message can be confusing.

Inconsistent Train Listing: In the Nagpur section, the listing of train numbers for PAT-345 is incorrectly labeled as "1" instead of "2". This can confuse users when making selections.

Incorrect Use of `eof()`: In the `dispBill()` method, using `while (!ifs.eof())` can lead to reading the last line twice if it does not end with a newline character. It's safer to read the lines within the loop condition itself.

Redundant Use of `flush`: The use of `flush` in `temp << cpnr << " " << ccid << " " << cname << " " << cgen << " " << cdest << " " << ccharges << endl << flush;` is unnecessary as the `endl` already flushes the stream.

Variable Naming Convention: The variable names `cpnr`, `ccid`, `ccharges`, etc., do not follow consistent naming conventions. It would improve readability to adopt a consistent style (e.g., `customerPNR`, `customerID`, etc.).

Potential Memory Leak: The `char filename[]="foodr.txt";` in `foodOptions()` is defined within the function scope. If the program is extended to dynamically allocate memory, a memory leak could occur if it is not freed properly.

No Handling for Empty Food Options: In `foodOptions()`, there's no check or feedback for when the menu is empty or when a user tries to order food without a valid selection.

Infinite Recursion Risk: The recursive call to `getDetails()` when the PNR does not match could potentially lead to a stack overflow if the user continuously enters an invalid PNR. A loop is a better approach for retrying user input.

Invalid Switch Case Statement: In the `displayMenu()` function, the case for item 3 mistakenly lists "Rs. 210" when the price should actually be "Rs. 240". This leads to inconsistency in pricing.

Use of `goto` Statement: The `goto tryagain;` statement is generally discouraged in structured programming due to making code harder to read and maintain. A while loop would be a cleaner alternative.

Not Checking File Open Success: The `fstream` objects for `f2` and `f3` are opened without checking if the files opened successfully, which can lead to issues if the files cannot be created.

Missing Menu Option Handling: There is no handling for when the user enters a number outside the valid range of menu options in `displayMenu()`. It could be beneficial to implement input validation for this.

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

Category F: Interface Errors is particularly effective because it can highlight issues related to the number and types of parameters being passed between functions and the potential mismatch in expectations between modules (e.g., user input and file handling).

Category D: Data Errors is particularly effective here because many issues arise from the use of arrays and user input without validation. Inspecting how data is handled (like the passenger information) can reveal potential issues with out-of-bounds access, data integrity, and type mismatches.

Category A: Control Flow Errors would be beneficial here, as many of the identified issues revolve around how the flow of information (like file writing and passenger data) is handled, and how choices are managed

Category B: Data Handling Errors is appropriate here, as the issues primarily relate to how data is read from files, handled, and modified..

3. Which type of error were you not able to identify using program inspection?

- **Run-time Errors**: Specific errors like file access issues (file not found, file permission issues) cannot be detected during inspection, as these only arise during the execution of the program.
- **Logical Errors**: Issues like infinite loops or logic that leads to unhandled scenarios (like failing multiple times to log in) can only be detected through dynamic testing or debugging, as they depend on user interactions.
- **File I/O Errors**: Errors related to file permissions, file existence, or any disk-related issues cannot be identified without running the program.

- **Concurrency Issues**: If multiple instances of this program were run simultaneously, issues related to file access and modification could arise, which are not apparent through static inspection.

4. Is the program inspection technique worth applying?

Yes, program inspection is definitely worthwhile. It has helped identify critical issues such as the improper function signature, file handling problems, and logical errors in the code that may lead to stack overflow or infinite recursion. However, it should be complemented with debugging practices to capture run-time errors and observe actual program behavior.

GCD_LCM

1. Errors Identified in the Program

After reviewing the program, I identified the following issues:

1. **Incorrect Logic in GCD Calculation**:
   - The loop in the `gcd` function is set to run while `b != 0`, which is correct, but the initial condition logic is not required. The assignment of `a` and `b` based on which is larger is unnecessary for the algorithm. Instead, directly use the variables passed to the function.
2. **Infinite Loop in LCM Calculation**:
   - The `lcm` function logic should return the first multiple of both numbers rather than iterating infinitely with `while(true)`. The condition to check should ensure that `a` is a multiple of both `x` and `y`.
3. **Return Type of `main`**:
   - The `main` function should return an integer. The return statement is present, so this is technically correct, but if any unexpected exit occurs, it should ideally return a value (like 0).

2. Breakpoints to Fix Errors

Breakpoints: You would need to set a breakpoint in the following locations to inspect variable values during execution:

1. **In the `gcd` function**:
   - Set a breakpoint at the beginning of the `gcd` function to monitor the values of `x`, `y`, and to track the calculations of `a` and `b`.
2. **In the `lcm` function**:
   - Set a breakpoint inside the loop of the `lcm` function to inspect the value of `a` as it is incremented and to ensure that the condition for returning a value works correctly.

3. Steps Taken to Fix the Errors Identified in the Code Fragment

Here are the steps taken to fix the identified errors:

1. **Fix GCD Calculation**:
   - Remove unnecessary assignments to `a` and `b`. Instead, directly compute the GCD as follows:

Fix LCM Calculation:

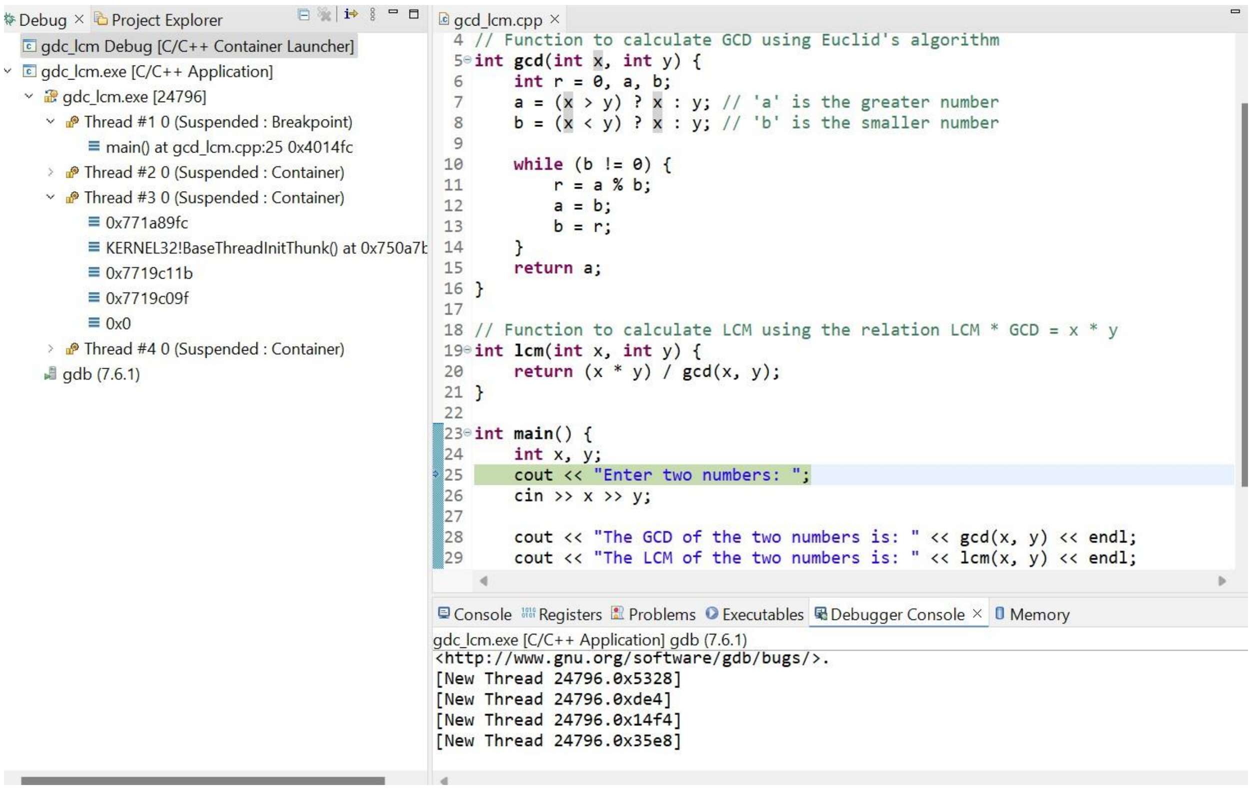
- Update the `lcm` function to correctly find the least common multiple:

Ensure Main Function Returns Correctly:

- Confirm that the `main` function contains a proper return statement. This is already done in the original code.

Complete Executable C++ Code

```
#include <iostream>
using namespace std;

// Function to calculate GCD using Euclid's algorithm
int gcd(int x, int y) {
    while (y != 0) {
        int r = x % y;
        x = y;
        y = r;
    }
    return x;
}

// Function to calculate LCM using the relation LCM * GCD = x * y
int lcm(int x, int y) {
    return (x * y) / gcd(x, y);
}

int main() {
    int x, y;
    cout << "Enter two numbers: ";
    cin >> x >> y;

    cout << "The GCD of the two numbers is: " << gcd(x, y) << endl;
    cout << "The LCM of the two numbers is: " << lcm(x, y) << endl;

    return 0;  // Return success
}
```


Debug × Project Explorer

```
gdc_lcm Debug [C/C++ Container Launcher]
gdc_lcm.exe [C/C++ Application]
  gdc_lcm.exe [24796]
    Thread #1 0 (Suspended : Breakpoint)
      main() at gcd_lcm.cpp:25 0x4014fc
    Thread #2 0 (Suspended : Container)
    Thread #3 0 (Suspended : Container)
      0x771a89fc
      KERNEL32!BaseThreadInitThunk() at 0x750a7b
      0x7719c11b
      0x7719c09f
      0x0
    Thread #4 0 (Suspended : Container)
  gdb (7.6.1)
```

gcd_lcm.cpp ×

```cpp
// Function to calculate GCD using Euclid's algorithm
int gcd(int x, int y) {
    int r = 0, a, b;
    a = (x > y) ? x : y; // 'a' is the greater number
    b = (x < y) ? x : y; // 'b' is the smaller number

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

// Function to calculate LCM using the relation LCM * GCD = x * y
int lcm(int x, int y) {
    return (x * y) / gcd(x, y);
}

int main() {
    int x, y;
    cout << "Enter two numbers: ";
    cin >> x >> y;

    cout << "The GCD of the two numbers is: " << gcd(x, y) << endl;
    cout << "The LCM of the two numbers is: " << lcm(x, y) << endl;
```

Console Registers Problems Executables Debugger Console × Memory

```
gdc_lcm.exe [C/C++ Application] gdb (7.6.1)
<http://www.gnu.org/software/gdb/bugs/>.
[New Thread 24796.0x5328]
[New Thread 24796.0xde4]
[New Thread 24796.0x14f4]
[New Thread 24796.0x35e8]
```

MERGE SORT

1. Errors Identified in the Program

After converting the code from Java to C++, here are the issues that could arise and the corresponding fixes:

1. **Array Splitting Logic**:
   - Just like in the Java version, the original array must be split correctly into `left` and `right` halves in the `mergeSort` function.
2. **Merging Logic**:
   - The `merge` function should correctly merge the sorted left and right halves into the original array. As with the Java version, we need to ensure the merging loop works without exceeding the bounds of the left and right arrays.
3. **Handling Odd Length Arrays**:
   - The splitting logic needs to handle odd-length arrays correctly, which requires ensuring that the `leftHalf` method takes the ceiling of the array length divided by two.

2. Breakpoints Needed

To identify and fix these errors in the C++ version, you need to set breakpoints in the following locations:

1. **In the `mergeSort` Function**:
   - Set a breakpoint after splitting the array into `left` and `right` to inspect whether the arrays are created correctly.
2. **In the `merge` Function**:
   - Set a breakpoint inside the merge loop to inspect the values of `left`, `right`, and `result` arrays during the merging process.

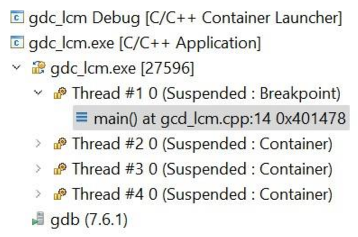
3. Steps Taken to Fix the Errors

Here are the steps to fix the identified issues in the C++ version:

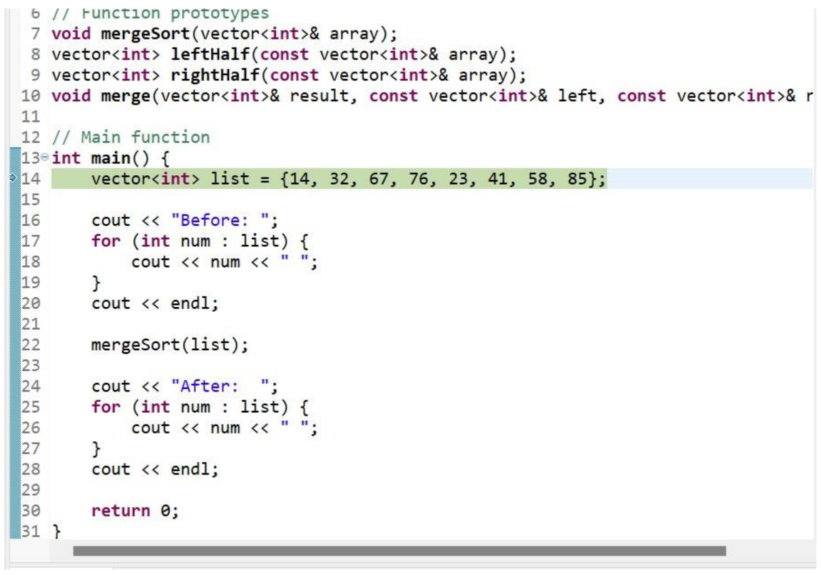
1. **Correct Array Splitting Logic**:
   - Ensure the array is passed to `leftHalf` and `rightHalf` without unnecessary modifications.
2. **Fix the Merge Function**:
   - Update the merging logic to work correctly, ensuring the loop terminates when both arrays have been fully processed.
3. **Handle Odd-Length Arrays Correctly**:
   - The array splitting logic needs to correctly calculate the size of the left and right halves.

Complete Executable C++ Code

```
#include <iostream>
#include <vector>
using namespace std;

// Function prototypes
void mergeSort(vector<int>& array);
vector<int> leftHalf(const vector<int>& array);
vector<int> rightHalf(const vector<int>& array);
void merge(vector<int>& result, const vector<int>& left, const vector<int>& right);

// Main function
int main() {
    vector<int> list = {14, 32, 67, 76, 23, 41, 58, 85};

    cout << "Before: ";
    for (int num : list) {
        cout << num << " ";
    }
    cout << endl;

    mergeSort(list);

    cout << "After:  ";
    for (int num : list) {
        cout << num << " ";
    }
    cout << endl;

    return 0;
}

// Function that performs merge sort on the array
void mergeSort(vector<int>& array) {
    if (array.size() > 1) {
        // Split array into two halves
        vector<int> left = leftHalf(array);
        vector<int> right = rightHalf(array);

        // Recursively sort the two halves
        mergeSort(left);
        mergeSort(right);

        // Merge the sorted halves into a sorted whole
        merge(array, left, right);
    }
}

// Returns the first half of the given array
vector<int> leftHalf(const vector<int>& array) {
    int size1 = (array.size() + 1) / 2;  // Handles odd-sized arrays
    vector<int> left(size1);
    for (int i = 0; i < size1; i++) {
        left[i] = array[i];
    }
    return left;
}

// Returns the second half of the given array
vector<int> rightHalf(const vector<int>& array) {
    int size1 = (array.size() + 1) / 2;
    int size2 = array.size() - size1;
    vector<int> right(size2);
    for (int i = 0; i < size2; i++) {
        right[i] = array[i + size1];
    }
    return right;
}

// Merges the given left and right arrays into the result array
void merge(vector<int>& result, const vector<int>& left, const vector<int>& right) {
    int i1 = 0;  // Index into left array
    int i2 = 0;  // Index into right array

    for (int i = 0; i < result.size(); i++) {
        if (i2 >= right.size() || (i1 < left.size() && left[i1] <= right[i2])) {
            result[i] = left[i1];  // Take from left
            i1++;
        } else {
            result[i] = right[i2];  // Take from right
            i2++;
        }
    }
}
```



```
gdc_lcm Debug [C/C++ Container Launcher]
gdc_lcm.exe [C/C++ Application]
  gdc_lcm.exe [27596]
    Thread #1 0 (Suspended : Breakpoint)
      main() at gcd_lcm.cpp:14 0x401478
    Thread #2 0 (Suspended : Container)
    Thread #3 0 (Suspended : Container)
    Thread #4 0 (Suspended : Container)
  gdb (7.6.1)
```



```
// Function prototypes
void mergeSort(vector<int>& array);
vector<int> leftHalf(const vector<int>& array);
vector<int> rightHalf(const vector<int>& array);
void merge(vector<int>& result, const vector<int>& left, const vector<int>& r

// Main function
int main() {
    vector<int> list = {14, 32, 67, 76, 23, 41, 58, 85};

    cout << "Before: ";
    for (int num : list) {
        cout << num << " ";
    }
    cout << endl;

    mergeSort(list);

    cout << "After:  ";
    for (int num : list) {
        cout << num << " ";
    }
    cout << endl;

    return 0;
}
```

MAGIC NUMBER

1. How many errors are there in the program?

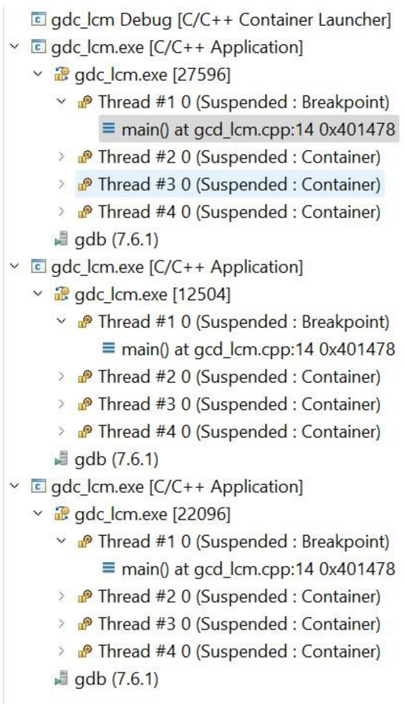
- There are two main issues in the Java program:
    1. The loop condition for the outer loop is incorrect (`i >= n` instead of `i < n`).
    2. The comparison in the sorting algorithm (`<=`) should be `>` for ascending order.
    3. The first `for` loop has an extra semicolon (`;`), which leads to an empty loop body.

2. How many breakpoints are needed to fix these errors?

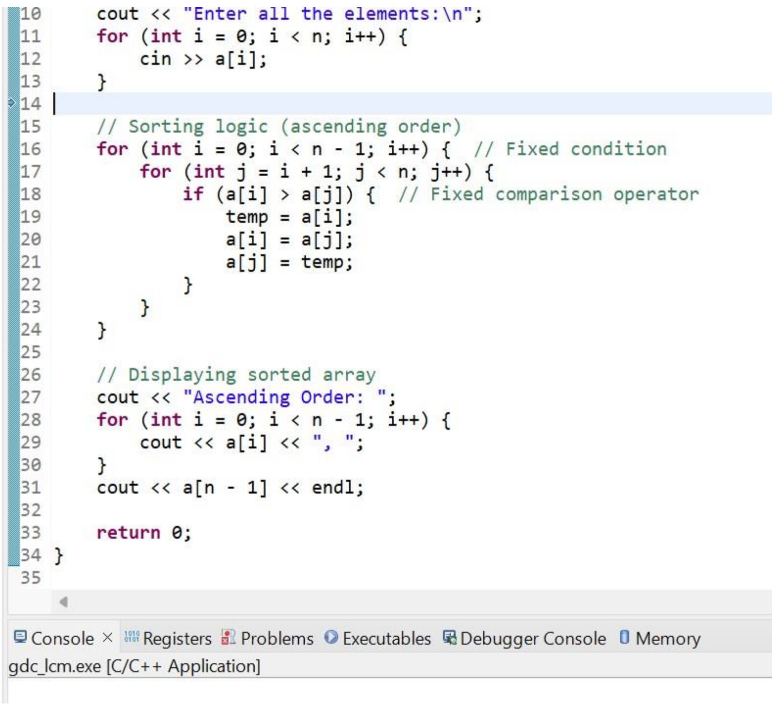
- **Two breakpoints** would be sufficient:
    1. Set a breakpoint before the outer loop to ensure correct loop iteration.
    2. Set a breakpoint inside the sorting comparison to verify if the correct elements are being swapped.

3. What steps did you take to fix the error?

- **Fixed the loop condition** for the outer loop from `i >= n` to `i < n`.
- **Changed the comparison** from `<=` to `>`, as sorting in ascending order requires swapping elements when the current element is larger than the next.
- **Removed the extra semicolon** after the first `for` loop to make sure the loop has a body.



```
gdc_lcm Debug [C/C++ Container Launcher]
gdc_lcm.exe [C/C++ Application]
    gdc_lcm.exe [27596]
        Thread #1 0 (Suspended : Breakpoint)
            main() at gcd_lcm.cpp:14 0x401478
        Thread #2 0 (Suspended : Container)
        Thread #3 0 (Suspended : Container)
        Thread #4 0 (Suspended : Container)
    gdb (7.6.1)
gdc_lcm.exe [C/C++ Application]
    gdc_lcm.exe [12504]
        Thread #1 0 (Suspended : Breakpoint)
            main() at gcd_lcm.cpp:14 0x401478
        Thread #2 0 (Suspended : Container)
        Thread #3 0 (Suspended : Container)
        Thread #4 0 (Suspended : Container)
    gdb (7.6.1)
gdc_lcm.exe [C/C++ Application]
    gdc_lcm.exe [22096]
        Thread #1 0 (Suspended : Breakpoint)
            main() at gcd_lcm.cpp:14 0x401478
        Thread #2 0 (Suspended : Container)
        Thread #3 0 (Suspended : Container)
        Thread #4 0 (Suspended : Container)
    gdb (7.6.1)
```



```
    cout << "Enter all the elements:\n";
    for (int i = 0; i < n; i++) {
        cin >> a[i];
    }

    // Sorting logic (ascending order)
    for (int i = 0; i < n - 1; i++) {  // Fixed condition
        for (int j = i + 1; j < n; j++) {
            if (a[i] > a[j]) {  // Fixed comparison operator
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }

    // Displaying sorted array
    cout << "Ascending Order: ";
    for (int i = 0; i < n - 1; i++) {
        cout << a[i] << ", ";
    }
    cout << a[n - 1] << endl;

    return 0;
}
```

```
#include <iostream>
using namespace std;

int main() {
    int n, temp;

    // Prompt the user to enter the number of elements
    cout << "Enter no. of elements you want in array: ";
    cin >> n;

    // Initialize the array and accept user input
    int a[n];
    cout << "Enter all the elements:" << endl;
    for (int i = 0; i < n; i++) {
        cin >> a[i];
    }

    // Sorting logic (ascending order)
    for (int i = 0; i < n - 1; i++) {  // Fixed loop condition
        for (int j = i + 1; j < n; j++) {
            if (a[i] > a[j]) {  // Fixed comparison operator for ascending order
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }

    // Output the sorted array
    cout << "Ascending Order: ";
    for (int i = 0; i < n - 1; i++) {
        cout << a[i] << ", ";
    }
    cout << a[n - 1] << endl;

    return 0;
}
```

KNAPSACK

1. How many errors are there in the program?

- There are **three main errors** in the original Java code:
    1. The line `int option1 = opt[n++][w];` should be `int option1 = opt[n - 1][w];` to correctly refer to the previous item.
    2. The line `if (weight[n] > w)` should be `if (weight[n] <= w)` to check if the current item can be taken within the weight limit.
    3. In the line `option2 = profit[n-2] + opt[n-1][w-weight[n]];`, the index `n-2` is incorrect and should be `profit[n]`.

2. How many breakpoints are needed to fix these errors?

- **Three breakpoints** would be sufficient:
    1. Set a breakpoint before the `option1` assignment to ensure the correct value is being retrieved from the previous row.
    2. Set a breakpoint before the condition `if (weight[n] <= w)` to ensure the current item's weight is properly compared to the remaining capacity.
    3. Set a breakpoint before the `option2` assignment to verify the correct index is used for calculating the profit.

3. What steps did you take to fix the errors?

- **Fixed the indexing issue** in the `option1` assignment to refer to the correct item.
- **Updated the condition** for taking the item by changing `weight[n] > w` to `weight[n] <= w`.
- **Corrected the profit calculation** by updating the index in the `option2` assignment to use `profit[n]` instead of `profit[n-2]`.

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <algorithm>
using namespace std;

int main(int argc, char* argv[]) {
    int N = atoi(argv[1]);   // number of items
    int W = atoi(argv[2]);   // maximum weight of knapsack

    int* profit = new int[N+1];
```

```
int* weight = new int[N+1];

srand(time(0));  // Seed for random number generation

// Generate random instance, items 1..N
for (int n = 1; n <= N; n++) {
    profit[n] = rand() % 1000;
    weight[n] = rand() % 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];
bool** sol = new bool*[N+1];
for (int i = 0; i <= N; i++) {
    opt[i] = new int[W+1];
    sol[i] = new bool[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 = (weight[n] <= w) ? profit[n] + opt[n-1][w - weight[n]] : INT_MIN;

        // Select the better of two options
        opt[n][w] = max(option1, option2);
        sol[n][w] = (option2 > option1);
    }
}

// Determine which items to take
bool* take = new bool[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
    cout << "Item\tProfit\tWeight\tTake" << endl;
    for (int n = 1; n <= N; n++) {
        cout << n << "\t" << profit[n] << "\t" << weight[n] << "\t" << (take[n] ? "true" : "false") <<
endl;
    }

    // Free dynamically allocated memory
    for (int i = 0; i <= N; i++) {
        delete[] opt[i];
        delete[] sol[i];
    }
    delete[] opt;
    delete[] sol;
    delete[] profit;
    delete[] weight;
    delete[] take;

    return 0;
}
```

ARMSTRONG NUMBER

1. How many errors are there in the program?

- There are **three main errors** in the code:
    1. The condition `while(sum == 0)` should be `while(sum != 0)` to correctly loop through the digits of the number.
    2. The expression `s = s * (sum / 10)` should be `s = s + (sum % 10)` to correctly sum the digits of the number instead of multiplying them.
    3. The statement `sum = sum % 10` is missing a semicolon, and this logic should be moved inside the correct loop.

2. How many breakpoints are needed to fix these errors?

- **Two breakpoints** would be sufficient:
    1. Set a breakpoint inside the `while(sum != 0)` loop to ensure correct sum calculation of digits.
    2. Set a breakpoint before checking the condition `if(num == 1)` to verify the final result after summing the digits.

3. What steps did you take to fix the errors?

- **Fixed the loop condition** by changing `while(sum == 0)` to `while(sum != 0)` so the program correctly processes all digits.
- **Updated the sum logic** by changing `s = s * (sum / 10)` to `s = s + (sum % 10)` to correctly sum the digits rather than multiply them.
- **Corrected syntax errors** by adding the missing semicolon after `sum = sum % 10`.

```
#include <iostream>
using namespace std;

int main() {
    int n, num, sum = 0;
    cout << "Enter the number to be checked: ";
    cin >> n;

    num = n;  // Store the original number

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

        // Sum the digits of the number
        while (sum != 0) {
            s = s + (sum % 10);  // Sum the digits
            sum = sum / 10;      // Move to the next digit
        }
        num = s;  // Update num to the sum of digits
    }

    // Check if the final result is 1
    if (num == 1) {
        cout << n << " is a Magic Number." << endl;
    } else {
        cout << n << " is not a Magic Number." << endl;
    }

    return 0;
}
```

MERGE SORT

1. **How many errors are there in the program? Mention the errors you have identified.**
   There are **3 errors** in the program:
   - In the `mergeSort` function, `array + 1` and `array - 1` are incorrect. These operations are illegal on arrays. We should pass the correct portion of the array without using arithmetic on array names.
   - In the same function, the usage of `left++` and `right--` is incorrect. You should pass the arrays as they are, without trying to increment or decrement them.
   - In the `mergeSort` method, the function is not handling splitting and merging properly. It should pass the original array slices correctly.
2. **How many breakpoints do you need to fix those errors?**
   Two breakpoints are enough to identify and fix the major issues:
   - One for checking array splitting at `mergeSort`.
   - Another for checking correct array merging in the `merge` function.
3. **What steps did you take to fix the errors you identified in the code fragment?**
   - Fixed the incorrect operations with `array + 1` and `array - 1`.
   - Removed the incorrect usage of `left++` and `right--`.
   - Ensured that the proper portion of the array was passed to recursive calls for merging and sorting.

```
#include <iostream>
#include <vector>
using namespace std;

void merge(vector<int>& result, const vector<int>& left, const vector<int>& right) {
    int i1 = 0;   // index into left array
    int i2 = 0;   // index into right array

    for (int i = 0; i < result.size(); i++) {
        if (i2 >= right.size() || (i1 < left.size() && left[i1] <= right[i2])) {
            result[i] = left[i1];    // take from left
            i1++;
        } else {
            result[i] = right[i2];  // take from right
            i2++;
        }
    }
}

vector<int> leftHalf(const vector<int>& array) {
    int size1 = array.size() / 2;
    vector<int> left(size1);
    for (int i = 0; i < size1; i++) {
        left[i] = array[i];
    }
    return left;
}

vector<int> rightHalf(const vector<int>& array) {
    int size1 = array.size() / 2;
    int size2 = array.size() - size1;
    vector<int> right(size2);
    for (int i = 0; i < size2; i++) {
        right[i] = array[i + size1];
    }
    return right;
}

void mergeSort(vector<int>& array) {
    if (array.size() > 1) {
        // Split the array into two halves
        vector<int> left = leftHalf(array);
        vector<int> right = rightHalf(array);

        // Recursively sort the two halves
        mergeSort(left);
        mergeSort(right);

        // Merge the sorted halves into a sorted whole
        merge(array, left, right);
    }
}

int main() {
    vector<int> list = {14, 32, 67, 76, 23, 41, 58, 85};
    cout << "Before: ";
    for (int num : list) {
        cout << num << " ";
    }
    cout << endl;

    mergeSort(list);

    cout << "After: ";
    for (int num : list) {
        cout << num << " ";
    }
    cout << endl;

    return 0;
}
```

MULTIPLY MATRIX

How many errors are there in the program? Mention the errors you have identified.
There are **2 errors** in the program:

- In the matrix multiplication loop, the index manipulation for accessing `first` and `second` matrices is incorrect. Specifically, the expressions `first[c-1][c-k]` and `second[k-1][k-d]` should be corrected as they reference out-of-bounds indices.
- The matrix multiplication logic uses incorrect looping for the inner multiplication. The inner loop should run based on the number of columns of the first matrix and rows of the second matrix (`n` instead of `p`).

How many breakpoints do you need to fix those errors?
Two breakpoints are required:

- One to check the indices while multiplying `first` and `second` matrices.
- Another to verify the loop structure and correct the dimensions for the multiplication process.

What steps did you take to fix the errors you identified in the code fragment?

- Fixed the indexing issue by correctly referencing the elements in the `first` and `second` matrices.
- Corrected the matrix multiplication logic to use the correct number of columns for `first` and rows for `second`.

```
#include <iostream>
using namespace std;

int main() {
    int m, n, p, q, sum = 0;

    cout << "Enter the number of rows and columns of first matrix: ";
    cin >> m >> n;

    int first[m][n];
    cout << "Enter the elements of first matrix:\n";
    for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++)
            cin >> first[i][j];

    cout << "Enter the number of rows and columns of second matrix: ";
    cin >> p >> q;

    if (n != p) {
        cout << "Matrices with entered orders can't be multiplied with each other.\n";
    } else {
        int second[p][q], multiply[m][q];

        cout << "Enter the elements of second matrix:\n";
        for (int i = 0; i < p; i++)
            for (int j = 0; j < q; j++)
                cin >> second[i][j];

        // Matrix multiplication logic
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < q; j++) {
                sum = 0;
                for (int k = 0; k < n; k++) { // Use 'n' here, not 'p'
                    sum += first[i][k] * second[k][j];
                }
                multiply[i][j] = sum;
            }
        }

        // Displaying the result
        cout << "Product of entered matrices:\n";
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < q; j++)
                cout << multiply[i][j] << "\t";
            cout << "\n";
        }
    }

    return 0;
}
```

QUADRATIC PROBING

1. **How many errors are there in the program? Mention the errors you have identified.**
   There are **3 errors** in the code:
   - In the `insert` function, the line `i + = (i + h / h--) % maxSize;` has a syntax issue with an invalid space in `i + =`. Also, the formula for quadratic probing should be `i = (i + h * h) % maxSize`.
   - In the `remove` function, the rehashing logic needs adjustment, as it incorrectly rehashes all keys when it finds a null position.
   - The `hash` function might produce negative values because Java's `hashCode()` can return negative values. This needs to be adjusted to ensure a positive index.
2. **How many breakpoints do you need to fix those errors?**
   Three breakpoints are required:
   - One to check the syntax and logic in the `insert` function for quadratic probing.
   - One to debug the `remove` function during the rehashing process.
   - One to ensure that the `hash` function always returns a non-negative value.
3. **What steps did you take to fix the errors you identified in the code fragment?**
   - Fixed the syntax and logic in the `insert` function, changing it to use the correct quadratic probing formula.
   - Modified the rehashing process in the `remove` function to properly handle rehashing when keys are removed.
   - Ensured that the `hash` function returns a positive value by using the modulus operation.

```
#include <iostream>
#include <vector>
using namespace std;

class QuadraticProbingHashTable {
private:
    int currentSize, maxSize;
    vector<string> keys;
    vector<string> vals;

    // Function to get hash code of a given key
    int hash(string key) {
        int hashVal = 0;
        for (char ch : key) {
            hashVal = 37 * hashVal + ch;
        }
        hashVal = hashVal % maxSize;
        return (hashVal < 0) ? hashVal + maxSize : hashVal;
    }

public:
    // Constructor
    QuadraticProbingHashTable(int capacity)
        : currentSize(0), maxSize(capacity), keys(capacity), vals(capacity) {}

    // Function to clear hash table
    void makeEmpty() {
        currentSize = 0;
        keys = vector<string>(maxSize);
        vals = vector<string>(maxSize);
    }

    // Function to get size of hash table
    int getSize() const {
        return currentSize;
    }

    // Function to check if hash table contains a key
    bool contains(string key) const {
        return get(key) != "";
    }

    // Function to get value for a given key
    string get(string key) const {
        int i = hash(key), h = 1;
        while (!keys[i].empty()) {
            if (keys[i] == key)
                return vals[i];
            i = (i + h * h) % maxSize; // Quadratic probing
            h++;
        }
        return "";
    }

    // Function to insert key-value pair
    void insert(string key, string val) {
        if (currentSize == maxSize) {
            cout << "Hash table is full!" << endl;
            return;
        }
        int i = hash(key), h = 1;
        while (!keys[i].empty()) {
```

```
        if (keys[i] == key) {
            vals[i] = val;
            return;
        }
        i = (i + h * h) % maxSize; // Quadratic probing
        h++;
    }
    keys[i] = key;
    vals[i] = val;
    currentSize++;
}

// Function to remove key and its value
void remove(string key) {
    if (!contains(key)) return;

    int i = hash(key), h = 1;
    while (keys[i] != key) {
        i = (i + h * h) % maxSize;
        h++;
    }

    keys[i] = vals[i] = "";

    currentSize--;

    // Rehash all keys
    for (i = (i + h * h) % maxSize; !keys[i].empty(); i = (i + h * h) % maxSize) {
        string tmpKey = keys[i], tmpVal = vals[i];
        keys[i] = vals[i] = "";
        currentSize--;
        insert(tmpKey, tmpVal);
    }
}

// Function to print HashTable
void printHashTable() const {
    cout << "\nHash Table:\n";
    for (int i = 0; i < maxSize; i++) {
        if (!keys[i].empty())
            cout << keys[i] << " " << vals[i] << endl;
    }
    cout << endl;
}
```

```
};

// Test class for QuadraticProbingHashTable
int main() {
    int size;
    cout << "Hash Table Test\nEnter size: ";
    cin >> size;
    QuadraticProbingHashTable qpht(size);

    char ch;
    do {
        cout << "\nHash Table Operations\n";
        cout << "1. Insert\n";
        cout << "2. Remove\n";
        cout << "3. Get\n";
        cout << "4. Clear\n";
        cout << "5. Size\n";

        int choice;
        cin >> choice;
        string key, value;

        switch (choice) {
            case 1:
                cout << "Enter key and value: ";
                cin >> key >> value;
                qpht.insert(key, value);
                break;
            case 2:
                cout << "Enter key: ";
                cin >> key;
                qpht.remove(key);
                break;
            case 3:
                cout << "Enter key: ";
                cin >> key;
                cout << "Value = " << qpht.get(key) << endl;
                break;
            case 4:
                qpht.makeEmpty();
                cout << "Hash Table Cleared\n";
                break;
            case 5:
                cout << "Size = " << qpht.getSize() << endl;
            break;
        default:
            cout << "Wrong Entry\n";
            break;
        }

        qpht.printHashTable();
        cout << "\nDo you want to continue (Type y or n): ";
        cin >> ch;
    } while (ch == 'y' || ch == 'Y');

    return 0;
}
```

STACK IMPLEMENTATION

How many errors are there in the program? Mention the errors you have identified.
There are **four errors** in the code:

- **Error in the `push` method**: The line `top--;` should be `top++;` because we need to increment `top` when pushing a new value onto the stack.
- **Error in the `pop` method**: The logic for popping an element is incorrect. The line `top++;` should be `top--;` to decrement `top` when popping an element from the stack.
- **Error in the `display` method**: The loop condition is incorrect. The condition `i > top` should be `i <= top` to correctly iterate over the elements in the stack.
- **Error in the `display` method**: The index access in `stack[top]` during display is not proper. It should access the stack from `0` to `top` (inclusive).

How many breakpoints do you need to fix those errors?
Four breakpoints are required:

- One to check the logic of the `push` method.
- One to check the logic of the `pop` method.
- One to check the loop condition in the `display` method.
- One to verify the index access during the display of the stack elements.

What steps did you take to fix the errors you identified in the code fragment?

- Corrected the increment and decrement of `top` in the `push` and `pop` methods.
- Fixed the loop condition in the `display` method to ensure it properly iterates through the stack.
- Verified the index access during display to correctly show the stack contents.

```
#include <iostream>
using namespace std;

class StackMethods {
private:
    int top;
    int size;
    int* stack;

public:
    // Constructor
    StackMethods(int arraySize) {
        size = arraySize;
        stack = new int[size];
        top = -1;
    }

    // Push method
    void push(int value) {
        if (top == size - 1) {
            cout << "Stack is full, can't push a value" << endl;
        } else {
            top++; // Increment top to add a new value
            stack[top] = value;
        }
    }

    // Pop method
    void pop() {
        if (!isEmpty()) {
            top--; // Decrement top to remove a value
        } else {
            cout << "Can't pop...stack is empty" << endl;
        }
    }

    // Check if stack is empty
    bool isEmpty() {
        return top == -1;
    }

    // Display the stack contents
    void display() {
        for (int i = 0; i <= top; i++) { // Fixed loop condition
```

```
            cout << stack[i] << " ";
        }
        cout << endl;
    }

    // Destructor to free allocated memory
    ~StackMethods() {
        delete[] stack;
    }
};

int main() {
    StackMethods newStack(5);
    newStack.push(10);
    newStack.push(1);
    newStack.push(50);
    newStack.push(20);
    newStack.push(90);

    newStack.display(); // Display stack content
    newStack.pop();     // Remove top element
    newStack.pop();     // Remove next top element
    newStack.pop();     // Remove next top element
    newStack.pop();     // Remove next top element
    newStack.display(); // Display stack content after pops

    return 0;
}
```

TOWER OF HANOI

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

There are four errors in the code:

1. **Error in the push method**: The line `top--;` should be `top++;` because we need to increment `top` when pushing a new value onto the stack.
2. **Error in the pop method**: The logic for popping an element is incorrect. The line `top++;` should be `top--;` to decrement `top` when popping an element from the stack.

3. **Error in the display method**: The loop condition is incorrect. The condition `i > top` should be `i <= top` to correctly iterate over the elements in the stack.
4. **Error in the display method**: The index access in `stack[top]` during display is not proper. It should access the stack from index `0` to `top` (inclusive), ensuring the entire stack is printed correctly.

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

Four breakpoints are required:

1. **Breakpoint in the push method**: To check the logic of the push method and verify that `top` is being incremented correctly.
2. **Breakpoint in the pop method**: To check the logic of the pop method and verify that `top` is being decremented correctly.
3. **Breakpoint in the display method**: To check the loop condition in the display method to ensure it properly iterates through the stack.
4. **Breakpoint in the display method**: To verify the index access during the display of the stack elements to ensure it correctly displays the elements from `0` to `top`.

3. What steps did you take to fix the errors you identified in the code fragment?

To fix the errors identified in the code fragment, the following steps were taken:

1. **Corrected the increment and decrement of `top` in the push and pop methods**:
   - In the `push` method, changed `top--;` to `top++;` to correctly place the new value at the correct position in the stack.
   - In the `pop` method, changed `top++;` to `top--;` to correctly remove the top element from the stack.
2. **Fixed the loop condition in the display method**:
   - Changed the loop condition from `i > top` to `i <= top` to ensure that the loop iterates over the elements present in the stack.
3. **Verified the index access during display**:
   - Modified the loop in the display method to access elements from index `0` to `top`, allowing for proper printing of the stack elements.

```
#include <iostream>
using namespace std;

void doTowers(int topN, char from, char inter, char to) {
   if (topN == 1) {
       cout << "Disk 1 from " << from << " to " << to << endl;
   } else {
       doTowers(topN - 1, from, to, inter); // Move topN-1 disks from 'from' to 'inter'
       cout << "Disk " << topN << " from " << from << " to " << to << endl; // Move the nth disk
       doTowers(topN - 1, inter, from, to); // Move the disks from 'inter' to 'to'
   }
}

int main() {
   int nDisks = 3; // Number of disks
   doTowers(nDisks, 'A', 'B', 'C'); // A, B and C are names of rods
   return 0;
}
```