IT - 314 Software Engineering

Lab 08

Functional Testing (Black Box)

Student ID : 202201230
Name : Rhythm Panchal

Question:

Write a set of test cases (i.e., test suite) – specific set of data – to properly test the programs. Your test suite should include both correct and incorrect inputs.

1. Enlist which set of test cases have been identified using Equivalence Partitioning and Boundary Value Analysis separately.
2. Modify your programs such that it runs, and then execute your test suites on the program.

While executing your input data in a program, check whether the identified expected outcome (mentioned by you) is correct or not.

.

Program: Consider a program for determining the previous date. Its input is triple of day, month and year with the following ranges
1 <= month <= 12, 1 <= day <= 31, 1900 <= year <= 2015.
The possible output dates would be the previous date or invalid date. Design the equivalence class test cases?

1. Equivalence Partitioning (EP):

We divide the input ranges into valid and invalid partitions (equivalence classes) for **day**, **month**, and **year**. Each equivalence class represents a set of inputs that should behave the same way when tested.

- **Day**:
  - Valid range: 1 <= day <= 31
  - Invalid ranges: day < 1 (too low) && day > 31 (too high)
- **Month**:
  - Valid range: 1 <= month <= 12
  - Invalid ranges: month < 1 (too low) && month > 12 (too high)
- **Year**:
  - Valid range: 1900 <= year <= 2015
  - Invalid ranges: year < 1900 (too low) && year> 2015 (too high)

2. Boundary Value Analysis (BVA):

In BVA, we test the boundaries of the input ranges to ensure the program handles edge cases correctly. This includes testing:

- Minimum and maximum valid values for **day**, **month**, and **year**

- Values just outside the valid range (e.g., day = 0, month = 13)

Define the Equivalence Classes:

Class Number	Description	Equivalence Class
E1	1≤day≤31	Day is valid
E2	day<1	Day is invalid (low)
E3	day>31	Day is invalid (high)
E4	1≤month≤12	Month is valid
E5	month<1	Month is invalid (low)
E6	month>12	Month is invalid (high)
E7	1900≤year≤2015	Year is valid
E8	year<1900	Year is invalid (low)
E9	year>2015	Year is invalid (high)

Test Cases Based on Equivalence Partitioning and Boundary Value Analysis:

Test Case	Day	Month	Year	Expected Outcome	Description	elevant Classe
TC1	1	1	1900	Valid	Minimum valid date.	E1, E4, E7
TC2	31	12	2015	Valid	Maximum valid date.	E1, E4, E7
TC3	0	1	2000	Invalid (low day)	Day is below the valid range.	E2
TC4	32	1	2000	Invalid (high day)	Day exceeds the valid range.	E3

TC5	15	0	2000	Invalid (low month)	Month is below the valid range.	E5
TC6	15	13	2000	Invalid (high month)	Month exceeds the valid range.	E6
TC7	15	6	1899	Invalid (low year)	Year is below the valid range.	E8
TC8	15	6	2016	Invalid (high year)	Year exceeds the valid range.	E9
TC9	15	6	2000	Valid	Middle of the valid range.	E1, E4, E7
TC10	1	12	1900	Valid	Boundary case for the day.	E1, E4, E7
TC11	31	1	2015	Valid	Boundary case for the day.	E1, E4, E7
TC12	1	1	2015	Valid	Minimum valid date in valid year.	E1, E4, E7
TC13	31	12	1900	Valid	Maximum valid date in valid year.	E1, E4, E7
TC14	29	2	2000	Valid	Leap year valid date.	E1, E4, E7
TC15	29	2	2015	Invalid (non-leap year)	February 29 in a non-leap year.	E1, E4, E9
TC16	30	4	2000	Valid	Valid date in April.	E1, E4, E7
TC17	31	4	2000	Invalid (April limit)	April has only 30 days.	E3
TC18	1	1	1899	Invalid (low year)	Year is below the valid range.	E8
TC19	31	11	2015	Valid	Valid date in November.	E1, E4, E7
TC20	30	11	2015	Valid	Valid date in November.	E1, E4, E7

Testing Code:

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

// Function to verify if a year is a leap year
bool leapYearCheck(int yearVal) {
    return (yearVal % 4 == 0 && (yearVal % 100 != 0 || yearVal % 400 == 0));
}

// Function to return the number of days in a month for a specific year
int monthDays(int monthVal, int yearVal) {
    vector<int> daysList = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
    if (monthVal == 2 && leapYearCheck(yearVal)) {
        return 29;
    }
    return daysList[monthVal - 1];
}

// Function to find the previous day's date
string getPreviousDate(int dayVal, int monthVal, int yearVal) {
    // Validate month and year
    if (!(1 <= monthVal && monthVal <= 12 && 1900 <= yearVal && yearVal <=
2015)) {
        return "Invalid date";
    }

    // Validate day based on the specific month and year
    int daysInCurrentMonth = monthDays(monthVal, yearVal);
    if (!(1 <= dayVal && dayVal <= daysInCurrentMonth)) {
        return "Invalid date";
    }

    // If the day is more than 1, simply subtract one day
    if (dayVal > 1) {
```

```
        return to_string(dayVal - 1) + ", " + to_string(monthVal) + ", " +
to_string(yearVal);
    }
    // If day is 1 but the month is greater than 1, go to the last day of the previous
month
    else if (monthVal > 1) {
        int prevMonthVal = monthVal - 1;
        return to_string(monthDays(prevMonthVal, yearVal)) + ", " +
to_string(prevMonthVal) + ", " + to_string(yearVal);
    }
    // If day is 1 and month is January, move to the last day of December in the
previous year
    else {
        return "31, 12, " + to_string(yearVal - 1);
    }
}

// Function to execute test cases
void executeTestCases() {
    vector<pair<vector<int>, string>> testCases = {
        {{1, 1, 1900}, "31, 12, 1899"},   // Test Case 1
        {{31, 12, 2015}, "30, 12, 2015"}, // Test Case 2
        {{0, 1, 2000}, "Invalid date"},   // Test Case 3
        {{32, 1, 2000}, "Invalid date"},  // Test Case 4
        {{15, 0, 2000}, "Invalid date"},  // Test Case 5
        {{15, 13, 2000}, "Invalid date"}, // Test Case 6
        {{15, 6, 1899}, "Invalid date"},  // Test Case 7
        {{15, 6, 2016}, "Invalid date"},  // Test Case 8
        {{15, 6, 2000}, "14, 6, 2000"},   // Test Case 9
        {{1, 12, 1900}, "30, 11, 1900"},  // Test Case 10
        {{31, 1, 2015}, "30, 1, 2015"},   // Test Case 11
        {{1, 1, 2015}, "31, 12, 2014"},   // Test Case 12
        {{31, 12, 1900}, "30, 12, 1900"}, // Test Case 13
        {{29, 2, 2000}, "28, 2, 2000"},   // Test Case 14
        {{29, 2, 2015}, "Invalid date"},  // Test Case 15
        {{30, 4, 2000}, "29, 4, 2000"},   // Test Case 16
        {{31, 4, 2000}, "Invalid date"},  // Test Case 17
```

```
        {{1, 1, 1899}, "Invalid date"},   // Test Case 18
        {{31, 11, 2015}, "Invalid date"}, // Test Case 19
        {{30, 11, 2015}, "29, 11, 2015"}  // Test Case 20
    };

    for (int i = 0; i < testCases.size(); i++) {
        vector<int> input = testCases[i].first;
        string expected = testCases[i].second;
        string result = getPreviousDate(input[0], input[1], input[2]);
        cout << "Test " << i + 1 << ": " << (result == expected ? "PASS" : "FAIL") <<
endl;
        cout << " Input: " << input[0] << ", " << input[1] << ", " << input[2] << endl;
        cout << " Expected: " << expected << endl;
        cout << " Actual: " << result << endl;
        cout << endl;
    }
}

int main() {
    executeTestCases();
    return 0;
}
```

Question 2 :

Program 1: The function linearSearch searches for a value v in an array of integers a. If v appears in the array a, then the function returns the first index i, such that a[i] == v; otherwise, -1 is returned.

Answer:

Define the Equivalence Classes:

Class Number	Description	Equivalence Class
1	The value v is present in the array.	Array contains v
2	The value v is not present in the array.	Array does not contain v
3	The array is empty.	Empty array
4	The array has only one element.	Single-element array

Test Cases Based on Equivalence Partitioning and Boundary Value Analysis:

Test Case	Array (a)	Value (v)	Expected Outcome	Description	evant Class
TC1	[2, 4, 6, 8]	4	1	v is present in the array	1
TC2	[2, 4, 6, 8]	5	-1	v is not present in the array	2
TC3	[]	3	-1	Empty array, no elements to search	3
TC4	[15]	15	0	Single-element array where v is present	4
TC5	[15]	20	-1	Single-element array where v is not present	5

Testing Code:

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

// Function to perform linear search
int linearSearch(vector<int>& a, int v) {
    for (int i = 0; i < a.size(); i++) {
        if (a[i] == v) {
            return i;
        }
    }
    return -1;
}

// Main function to run test cases
int main() {
    // Test cases
    vector<pair<vector<int>, int>> test_cases = {
        {{3, 5, 8, 10}, 5},    // E1: Value present
        {{3, 5, 8, 10}, 15},   // E2: Value not present
        {{}, 5},               // E3: Empty array
        {{3, 5, 8, 5, 10}, 5}, // E4: Value present multiple times
        {{3}, 3},              // BVA: Single element, present
        {{3}, 10}              // BVA: Single element, not present
    };

    // Running test cases
    for (const auto& test : test_cases) {
        vector<int> a = test.first;
        int v = test.second;
        int result = linearSearch(a, v);
        cout << "Array: {";
```

```
        for (int i = 0; i < a.size(); i++) {
            cout << a[i];
            if (i < a.size() - 1) cout << ", ";
        }
        cout << "}, Value: " << v << " => Output: " << result << endl;
    }

    return 0;
}
```

Program 2: The function countItem returns the number of times a value v appears in an array of integers a.

Answer:

Define the Equivalence Classes:

Class Number	Description	Equivalence Class
E1	The value v appears in the array at least once.	Array contains v
E2	The value v does not appear in the array.	Array does not contain v
E3	The array is empty.	Empty array
E4	The array has only one element, and that element is v.	Single-element array containing v
E5	The array has only one element, and that element is not v.	Single-element array not containing v
E6	The array contains multiple occurrences of v.	Array has multiple occurrences of v

Test Cases Based on Equivalence Partitioning and Boundary Value Analysis:

Test Case	Array (a)	Value (v)	Expected Outcome
TC1	[3, 5, 8, 10]	5	1
TC2	[3, 5, 8, 10]	15	0
TC3	[]	5	0
TC4	[3, 5, 8, 5, 10]	5	2
TC5	[3]	3	1
TC6	[3]	10	0

Testing Code:

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

// Function to count the occurrences of value v in array a
int countItem(const vector<int>& a, int v) {
    int count = 0;
    for (int i = 0; i < a.size(); i++) {
        if (a[i] == v) {
            count++;
        }
    }
    return count;
}

// Main function to run test cases
int main() {
    // Test cases
    vector<pair<vector<int>, int>> test_cases = {
        {{3, 5, 8, 10}, 5},    // E1: Value appears once
        {{3, 5, 8, 10}, 15},   // E2: Value does not appear
        {{}, 5},               // E3: Empty array
        {{3, 5, 8, 5, 10}, 5}, // E6: Value appears multiple times
        {{3}, 3},              // E4: Single element, present
```

```
        {{3}, 10}              // E5: Single element, not present
    };

    // Running test cases
    for (const auto& test : test_cases) {
        vector<int> a = test.first;
        int v = test.second;
        int result = countItem(a, v);
        cout << "Array: {";
        for (int i = 0; i < a.size(); i++) {
            cout << a[i];
            if (i < a.size() - 1) cout << ", ";
        }
        cout << "}, Value: " << v << " => Count: " << result << endl;
    }

    return 0;
}
```

Program 3: The function binarySearch searches for a value v in an ordered array of integers a. If v appears in the array a, then the function returns an index i, such that a[i] == v; otherwise, -1 is returned.
Assumption: the elements in the array a are sorted in non-decreasing order.

Answer:

Define the Equivalence Classes:

Class Number	Description	Equivalence Class

E1	v is present in the array	v exists in a[]
E2	v is not present in the array	v does not exist in a[]
E3	Array contains only one element, and v is present	a[] contains only one element == v
E4	Array contains only one element, and v is not present	a[] contains only one element != v
E5	Empty array	a[] is empty
E6	v is the first element in the array	v == a[0]
E7	v is the last element in the array	v == a[n-1]

Test Cases Based on Equivalence Partitioning and Boundary Value Analysis:

Test Case	Array (a)	Value (v)	Expected Output	Description	evant Clas
TC1	[1, 3, 5, 7, 9]	5	2	v is present in the array	E1
TC2	[1, 3, 5, 7, 9]	4	-1	v is not present in the array	E2
TC3	[7]	7	0	Single element, v is present	E3
TC4	[7]	5	-1	Single element, v is not present	E4
TC5	[]	5	-1	Empty array	E5
TC6	[1, 3, 5, 7, 9]	1	0	v is the first element	E6
TC7	[1, 3, 5, 7, 9]	9	4	v is the last element	E7

Testing Code:

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

// Function for binary search
int binarySearch(const vector<int>& a, int v) {
    int left = 0;
    int right = a.size() - 1;
```

```
    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (a[mid] == v) {
            return mid;
        }
        else if (a[mid] < v) {
            left = mid + 1;
        }
        else {
            right = mid - 1;
        }
    }

    return -1;  // Value not found
}

// Main function to test the binarySearch function
int main() {
    // Test cases
    vector<pair<vector<int>, int>> test_cases = {
        {{1, 3, 5, 7, 9}, 5},    // TC1: Value is present
        {{1, 3, 5, 7, 9}, 4},    // TC2: Value is not present
        {{7}, 7},                // TC3: Single element, present
        {{7}, 5},                // TC4: Single element, not present
        {{}, 5},                 // TC5: Empty array
        {{1, 3, 5, 7, 9}, 1},    // TC6: Value is the first element
        {{1, 3, 5, 7, 9}, 9}     // TC7: Value is the last element
    };

    // Running test cases
    for (int i = 0; i < test_cases.size(); ++i) {
        vector<int> array = test_cases[i].first;
        int value = test_cases[i].second;
        int result = binarySearch(array, value);
```

```
        cout << "Test Case " << i + 1 << ": Array = {";
        for (int j = 0; j < array.size(); ++j) {
            cout << array[j];
            if (j < array.size() - 1) cout << ", ";
        }
        cout << "}, Value = " << value << " => Output: " << result << endl;
    }

    return 0;
}
```

Program 4: The function triangle takes three integer parameters that are interpreted as the lengths of the sides of a triangle. It returns whether the triangle is equilateral (three lengths equal), isosceles (two lengths equal), scalene (no lengths equal), or invalid (impossible lengths).

Answer:

Define the Equivalence Classes:

Class Number	Description	Equivalence Class
E1	All sides are equal	Equilateral triangle
E2	Two sides are equal	Isosceles triangle
E3	No sides are equal	Scalene triangle
E4	Sides do not form a valid triangle (e.g., A + B <= C)	Invalid triangle (triangle inequality violation)
E5	One or more side lengths are zero or negative	Invalid triangle (non-positive side lengths)

Test Cases Based on Equivalence Partitioning and Boundary Value Analysis:

Test Case	A	B	C	Expected Output	Description	elevant Classe
TC1	3	3	3	Equilateral	All sides are equal	E1
TC2	4	4	7	Isosceles	Two sides are equal	E2
TC3	5	6	7	Scalene	No sides are equal	E3
TC4	1	2	3	Invalid	Triangle inequality violation (A + B <= C)	E4
TC5	0	5	5	Invalid	One side is zero (non-positive side length)	E5
TC6	-1	4	5	Invalid	One side is negative	E5

Testing Code:

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

// Function to determine the type of triangle
string triangle(int A, int B, int C) {
    // Check if the sides form a valid triangle
    if (A <= 0 || B <= 0 || C <= 0 || (A + B <= C) || (A + C <= B) || (B + C <= A)) {
        return "Invalid";
    }

    // Check for equilateral triangle
    if (A == B && B == C) {
        return "Equilateral";
    }

    // Check for isosceles triangle
    if (A == B || B == C || A == C) {
        return "Isosceles";
    }

    // If none of the above, it's a scalene triangle
    return "Scalene";
}

// Main function to test the triangle function
```

```
int main() {
    // Test cases
    int test_cases[6][3] = {
        {3, 3, 3},   // TC1: Equilateral triangle
        {4, 4, 7},   // TC2: Isosceles triangle
        {5, 6, 7},   // TC3: Scalene triangle
        {1, 2, 3},   // TC4: Invalid triangle (triangle inequality violation)
        {0, 5, 5},   // TC5: Invalid triangle (zero side length)
        {-1, 4, 5}   // TC6: Invalid triangle (negative side length)
    };

    string expected_outputs[6] = {
        "Equilateral", "Isosceles", "Scalene", "Invalid", "Invalid", "Invalid"
    };

    // Run the test cases
    for (int i = 0; i < 6; ++i) {
        int A = test_cases[i][0];
        int B = test_cases[i][1];
        int C = test_cases[i][2];

        string result = triangle(A, B, C);
        cout << "Test Case " << i + 1 << ": A = " << A << ", B = " << B << ", C = " <<
C
             << " => Output: " << result << ", Expected: " << expected_outputs[i] <<
endl;
    }

    return 0;
}
```

Program 5: The function prefix (String s1, String s2) returns whether or not the string s1 is a prefix of string s2 (you may assume that neither s1 nor s2 is null).

Answer:

Define the Equivalence Classes:

Class Number	Description	Equivalence Class
E1	s1 is a prefix of s2.	s1 is a prefix of s2
E2	s1 is not a prefix of s2.	s1 is not a prefix of s2
E3	s1 is an empty string.	s1 is empty
E4	s2 is an empty string.	s2 is empty
E5	s1 is equal to s2.	s1 equals s2

Test Cases Based on Equivalence Partitioning and Boundary Value Analysis:

Test Case	String s1	String s2	Expected Outcome
TC1	"pre"	"prefix"	TRUE
TC2	"post"	"prefix"	FALSE
TC3	""	"prefix"	TRUE
TC4	"pre"	""	FALSE
TC5	"prefix"	"prefix"	TRUE
TC6	"longprefix"	"pre"	FALSE

Testing Code:

```
#include <iostream>
```

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

// Function to check if s1 is a prefix of s2
bool prefix(const string& s1, const string& s2) {
    if (s1.size() > s2.size()) {
        return false;
    }
    for (int i = 0; i < s1.size(); i++) {
        if (s1[i] != s2[i]) {
            return false;
        }
    }
    return true;
}

// Main function to run test cases
int main() {
    // Test cases
    vector<pair<string, string>> test_cases = {
        {"pre", "prefix"},       // E1: s1 is a prefix of s2
        {"post", "prefix"},      // E2: s1 is not a prefix of s2
        {"", "prefix"},          // E3: s1 is empty
        {"pre", ""},             // E4: s2 is empty
        {"prefix", "prefix"},    // E5: s1 is equal to s2
        {"longprefix", "pre"}    // E6: s1 is longer than s2
    };

    // Expected outcomes
    vector<bool> expected_outcomes = {true, false, true, false, true, false};

    // Running test cases
    for (int i = 0; i < test_cases.size(); i++) {
        string s1 = test_cases[i].first;
        string s2 = test_cases[i].second;
        bool result = prefix(s1, s2);
        bool expected = expected_outcomes[i];
```

```
        cout << "Test Case " << i + 1 << ": ";
        cout << "s1: \"" << s1 << "\", s2: \"" << s2 << "\" => Output: " << (result ?
"true" : "false");
        cout << " (Expected: " << (expected ? "true" : "false") << ")" << endl;
    }

    return 0;
}
```

Program 6: Consider again the triangle classification program (P4) with a slightly different specification: The program reads floating values from the standard input. The three values A, B, and C are interpreted as representing the lengths of the sides of a triangle. The program then prints a message to the standard output that states whether the triangle, if it can be formed, is scalene, isosceles, equilateral, or right angled. Determine the following for the above program:

1. Identify the equivalence classes for the system
2. Identify test cases to cover the identified equivalence classes. Also, explicitly mention which test case would cover which equivalence class. (Hint: you must need to be ensure that the identified set of test cases cover all identified equivalence classes)
3. For the boundary condition A + B > C case (scalene triangle), identify test cases to verify the boundary.
4. For the boundary condition A = C case (isosceles triangle), identify test cases to verify the boundary.
5. For the boundary condition A = B = C case (equilateral triangle), identify test cases to verify the boundary.

6. For the boundary condition A2 + B2 = C2 case (right-angle triangle), identify test cases to verify the boundary.
7. For the non-triangle case, identify test cases to explore the boundary.
8. For non-positive input, identify test points.

Answer:

Triangle Classification Program:

a) Identify the Equivalence Classes

Class Number	Description	Equivalence Class
E1	A valid scalene triangle	A != B != C and forms a triangle
E2	A valid isosceles triangle	A == B or B == C or A == C
E3	A valid equilateral triangle	A == B == C
E4	A valid right-angled triangle	$A^2 + B^2 == C^2$ or similar
E5	Inputs do not form a triangle	A + B <= C or any side invalid
E6	One or more non-positive inputs	A <= 0, B <= 0, C <= 0

b) Identify Test Cases to Cover the Identified Equivalence Classes

Test Case	A	B	C	Expected Outcome	Description	levant Class
TC1	3	4	5	Right-angled triangle	Right-angled triangle (Pythagoras' theorem)	E4
TC2	3	3	3	Equilateral triangle	All sides equal	E3
TC3	4	4	5	Isosceles triangle	Two sides equal	E2

TC4	3	5	7	Scalene triangle	All sides different and valid	E1
TC5	1	2	3	Not a triangle	Sum of two sides equals the third (non-triangle)	E5
TC6	0	2	3	Invalid input (non-triangle)	Non-positive side length	E6
TC7	-1	2	3	Invalid input (non-triangle)	Negative side length	E6
TC8	1	1	2	Not a triangle	Sum of two sides equals the third (non-triangle)	E5
TC9	0.1	0.1	0.2	Not a triangle	Non-triangle with positive, small sides	E5

c) Boundary Condition A + B > C (Scalene Triangle)

Input Data (A, B, C)	Expected Outcome	Description
(2.0, 3.0, 4.0)	"Scalene"	Valid scalene triangle
(2.0, 2.0, 3.9)	"Scalene"	Valid scalene triangle
(2.0, 2.0, 4.0)	"Invalid"	Fails triangle inequality

d) Boundary Condition A = C (Isosceles Triangle)

Input Data (A, B, C)	Expected Outcome	Description
(3.0, 4.0, 3.0)	"Isosceles"	Valid isosceles triangle
(3.0, 4.0, 4.0)	"Isosceles"	Valid isosceles triangle
(3.0, 4.0, 2.9)	"Invalid"	Fails triangle inequality

e) Boundary Condition A = B = C (Equilateral Triangle)

Input Data (A, B, C)	Expected Outcome	Description
(3.0, 3.0, 3.0)	"Equilateral"	Valid equilateral triangle

(2.0, 2.0, 2.0)	"Equilateral"	Valid equilateral triangle
(2.9, 2.9, 2.9)	"Equilateral"	Valid equilateral triangle

f) Boundary Condition A2+B2=C2A^2 + B^2 = C^2A2+B2=C2 (Right-Angle Triangle)

Input Data (A, B, C)	Expected Outcome	Description
(3.0, 4.0, 5.0)	"Right-angled"	Valid right-angle triangle
(5.0, 12.0, 13.0)	"Right-angled"	Valid right-angle triangle
(3.0, 4.0, 4.9)	"Invalid"	Fails right-angle condition

g) Non-Triangle Case

Input Data (A, B, C)	Expected Outcome	Description
(1.0, 2.0, 3.0)	"Invalid"	Fails triangle inequality
(5.0, 2.0, 2.0)	"Invalid"	Fails triangle inequality
(10.0, 1.0, 1.0)	"Invalid"	Fails triangle inequality

h) Non-Positive Input

Input Data (A, B, C)	Expected Outcome	Description
(0.0, 1.0, 1.0)	"Invalid"	Non-positive side
(-1.0, 2.0, 2.0)	"Invalid"	Non-positive side
(1.0, 0.0, 1.0)	"Invalid"	Non-positive side
(1.0, 1.0, -1.0)	"Invalid"	Non-positive side

Implementation of the Triangle Classification Program:

```
#include <iostream>
```

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

// Function to check and classify the triangle
string classifyTriangle(double A, double B, double C) {
    // Check for non-positive sides
    if (A <= 0 || B <= 0 || C <= 0) {
        return "Invalid input (non-triangle)";
    }

    // Check if it's a triangle
    if (A + B <= C || A + C <= B || B + C <= A) {
        return "Not a triangle";
    }

    // Check for equilateral triangle
    if (A == B && B == C) {
        return "Equilateral triangle";
    }

    // Check for isosceles triangle
    if (A == B || B == C || A == C) {
        return "Isosceles triangle";
    }

    // Check for right-angled triangle using Pythagoras theorem
    if (fabs(A * A + B * B - C * C) < 1e-6 ||
        fabs(A * A + C * C - B * B) < 1e-6 ||
        fabs(B * B + C * C - A * A) < 1e-6) {
        return "Right-angled triangle";
    }

    // If none of the above, it's a scalene triangle
    return "Scalene triangle";
}

// Main function to run the test cases
int main() {
    // Test cases
    double test_cases[][3] = {
        {3.0, 4.0, 5.0},  // TC1: Right-angled triangle
        {3.0, 3.0, 3.0},  // TC2: Equilateral triangle
        {4.0, 4.0, 5.0},  // TC3: Isosceles triangle
        {3.0, 5.0, 7.0},  // TC4: Scalene triangle
```

```
        {1.0, 2.0, 3.0},  // TC5: Not a triangle
        {0.0, 2.0, 3.0},  // TC6: Invalid input (non-triangle)
        {-1.0, 2.0, 3.0}, // TC7: Invalid input (non-triangle)
        {1.0, 1.0, 2.0},  // TC8: Not a triangle
        {0.1, 0.1, 0.2},  // TC9: Not a triangle
        {5.0, 7.0, 11.9999}, // TC10: Scalene (boundary)
        {5.0, 7.0, 5.0},     // TC11: Isosceles (boundary)
        {5.0, 5.0, 5.0},     // TC12: Equilateral (boundary)
        {6.0, 8.0, 10.0},    // TC13: Right-angled triangle (boundary)
        {1.0, 2.0, 3.0},     // TC14: Not a triangle (boundary)
        {0.0, 1.0, 2.0},     // TC15: Invalid input (zero side)
        {-1.0, 1.0, 2.0}     // TC16: Invalid input (negative side)
    };

    // Running the test cases
    for (int i = 0; i < 16; i++) {
        double A = test_cases[i][0];
        double B = test_cases[i][1];
        double C = test_cases[i][2];
        cout << "Test Case " << i + 1 << ": A = " << A << ", B = " << B << ", C = " << C << " => " <<
classifyTriangle(A, B, C) << endl;
    }

    return 0;
}
```