IT – 314
SOFTWARE ENGINEERING

Lab – 08: Functional Testing

Student Name: Krushang Kanakad

Student ID: 202201350

Date of Submission: October 23, 2024

Question 1

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 previous date or invalid date. Design the equivalence class test cases?

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.

Answer :

Test Cases :

No.	Day	Month	Year	Valid/Invalid
1.	1	1	1900	Valid
2.	1	3	2015	Valid
3.	29	2	2013	Invalid
4.	31	4	2015	Invalid
5.	31	6	2007	Invalid

6.	0	10	2001	Invalid
7.	20	15	2005	Invalid
8.	29	2	2012	Valid
9.	31	0	1901	Invalid
10.	31	12	2014	Valid
11.	32	1	2000	Invalid
12.	1	1	1900	Valid
13.	1	1	2000	valid
14.	31	12	2014	Valid
15.	1	2	2000	Valid
16.	29	2	2004	Valid
17.	31	12	2015	Valid
18.	31	12	1900	Valid
19	1	1	2016	Invalid
20.	28	2	2015	Valid
21.	0	2	1995	Invalid
22.	32	1	2010	Invalid

1. Enlist which set of test cases have been identified using Equivalence Partitioning and Boundary Value Analysis separately.

There are three parameters, i.e., day, month and year, in this problem, so there will be separate equivalence classes for each parameter.

Equivalence Classes:

1. Date

Valid: 1 ≤ day ≤ 31
Invalid: day < 1, day > 31

2. Month

Valid: 1 ≤ month ≤ 12
Invalid: month < 1, month > 12

3. Year

Valid: 1900 ≤ year ≤ 2015
Invalid: year < 1900, year > 2015

Test Cases

1. Equivalence Partitioning

In equivalence partitioning, we divide the input data into groups, or partitions, where each group contains a set of equivalent or similar values expected to exhibit similar behavior in the system under test.

Here are some partitions based on different values:

Partition 1: Valid dates with a day between 1 and 31, a month between 1 and 12, and a year between 1900 and 2015.
Partition 2: Invalid dates with a day less than 1 or greater than 31.
Partition 3: Invalid dates with a month less than 1 or greater than 12.
Partition 4: Invalid dates with a year less than 1900 or greater than 2015.
Partition 5: Invalid dates with a day that is out of range for a given month (e.g., February 30).

Partition 6: Invalid dates with a day that is out of range for a given year (e.g., February 29 in a non-leap year).

Some sample test cases for different partitions:

Partition 1: 01/01/2009, 15/03/1990, 31/12/2004
Partition 2: 00/01/2004, -10/03/2001, 32/12/2000
Partition 3: 01/00/2001, 15/13/2011, 31/15/2010
Partition 4: 01/01/0000, 15/03/10000, 31/12/99999
Partition 5: 30/02/2022, 31/04/2023, 28/02/2100
Partition 6: 29/02/2021, 29/02/1900, 29/02/2100

No.	Day	Month	Year	Valid/Invalid
1.	1	1	1900	Valid
2.	1	3	2015	Valid
3.	29	2	2013	Invalid
4.	31	4	2015	Invalid
5.	31	6	2007	Invalid
6.	0	10	2001	Invalid
7.	20	15	2005	Invalid
8.	29	2	2012	Valid
9.	31	0	1901	Invalid
10.	31	12	2014	Valid
11.	32	1	2000	Invalid

2. Boundary Value Analysis

In boundary value analysis, we check for input values near the boundaries of valid and invalid values that are more likely to cause errors. Testing these boundary values can help identify potential problems in the software.

We first identify the boundary values for day, month, and year.

- Day: 1, 28, 29, 30, 31
- Month: 1, 2, 12
- Year: 1, 4, 100, 400 (for checking Leap Years).

We then find valid and invalid input ranges for day, month, and year.

- Day: valid input range is from 1 to 31, invalid input range is from 32 to infinity.
- Month: valid input range is from 1 to 12, invalid input range is from 13 to infinity.
- Year: valid input range is from 1900 to 2015, invalid range is anything outside that range.

Using these to generate test cases:

Test case 1: Valid date (boundary value) - Day: 1, Month: 1, Year: 2010
Test case 2: Valid date (boundary value) - Day: 31, Month: 12, Year: 2010
Test case 3: Valid date (boundary value) - Day: 29, Month: 2, Year: 2000 (leap year)
Test case 4: Invalid date (boundary value) - Day: 32, Month: 1, Year: 1990
Test case 5: Invalid date (boundary value) - Day: 13, Month: 2, Year: 1910
Test case 6: Invalid date (boundary value) - Day: 30, Month: 2, Year: 1930

Test case 7: Invalid date (boundary value) - Day: 31, Month: 4, Year: 1930
Test case 8: Valid date (within valid range) - Day: 15, Month: 6, Year: 2015
Test case 9: Invalid date (day is outside valid range) - Day: 32, Month: 6, Year: 2010
Test case 10: Invalid date (month is outside valid range) - Day: 15, Month: 13, Year: 2010
Test case 11: Invalid date (year is outside valid range) - Day: 15, Month: 6, Year: 2030

No.	Day	Month	Year	Valid/Invalid
1.	1	1	1900	Valid
2.	1	1	2000	valid
3.	31	12	2014	Valid
4.	1	2	2000	Valid
5.	29	2	2004	Valid
6.	31	12	2015	Valid
7.	31	12	1900	Valid
8.	1	1	2016	Invalid
9.	28	2	2015	Valid
10.	0	2	1995	Invalid
11.	32	1	2010	Invalid

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.

C++ code for finding Previous date:

```
bool isLeapYear(int year)

{

    return (year % 4 == 0 && year % 100 != 0) || (year % 400 == 0);

}

string CheckDateValidOrNot(int day, int month, int year)

{

    if (year < 1900 || year > 2015)

    {

        return "Invalid date";

    }

    else if (month < 1 || month > 12)

    {

        return "Invalid date";

    }

    else if (day < 1 || day > 31)

    {

        return "Invalid date";

    }

    monthDay[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

    if (isLeapYear(year))
```

```
{
    monthDay[1] = 29;
}
if (day > monthDay[month - 1])
{
    return "Invalid date";
}
string previousDate = "";
if (day > 1)
{
    previousDate += to_string(day - 1);
    previousDate += ',';
    previousDate += to_string(month);
    previousDate += ',';
    previousDate += to_string(year);
}
else
{
    if (month == 1)
    {
        previousDate += to_string(31);
        previousDate += ',';
        previousDate += to_string(12);
        previousDate += ',';
        previousDate += to_string(year - 1);
        }
        else
        {
            previousDate += to_string(monthDay[month - 2]);
            previousDate += ',';
            previousDate += to_string(month - 1);
            previousDate += ',';
            previousDate += to_string(year);
        }
    }
    return previousDate;
}
```

Question 2 : Programs

P1. 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.

```
int linearSearch(int v, int a[])
{
   int i = 0;
   while (i < a.length)
   {
      if (a[i] == v)
         return (i);
      i++;
   }
   return (-1);
}
```

Equivalence Partitioning Test cases:

No.	Test Description	Input Array(a)	v	Output
1.	Value in the middle of the array	{1, 2, 3, 4, 5}	3	2
2.	Value not in the array	{1, 2, 3, 4, 5}	6	-1
3.	Empty array	{}	5	-1
4.	Single element array, value not found	{7}	9	-1
5.	Single element array, value found	{7}	9	0
6.	Large array, value in the middle	{10, 20, 30, ..., 1000}	500	49

7.	Large array, value not present	{10, 20, 30, ..., 1000}	1500	-1
8.	Array with duplicate values, first match	{1, 3, 3, 4, 5}	3	1
9.	Array with negative numbers, value present	{-5, -4, -3, -2, -1}	-3	2
10.	Array with both positive and negative numbers	{-5, -1, 0, 1, 5}	0	2

Boundary Value Analysis Test cases:

No.	Test Description	Input Array(a)	v	Output
1.	Value at the start of the array	{1, 2, 3, 4, 5}	1	0
2.	Value at the end of the array	{1, 2, 3, 4, 5}	5	4
3.	Empty array	{}	5	-1
4.	Single element array, value not found	{7}	9	-1
5.	Boundary case, array length of 1, value not present	{1}	2	-1
6.	Large array, value at the last index	{1, 2, 3, ..., 1000}	1000	999

Modified Programm:

```
#include <iostream>

using namespace std;

int searchValue(int target, int array[], int size)

{

    for (int i = 0; i < size; i++)

    {

        if (array[i] == target)

            return i;

    }

    return -1;

}

int main()

{

    int numbers1[] = {10, 20, 30, 40, 50};

    int numbers2[] = {};

    int numbers3[] = {-10, -20, -30};

    cout << "Test 1 (target=30): " << searchValue(30, numbers1, 5) << endl;   // Output: 2

    cout << "Test 2 (target=60): " << searchValue(60, numbers1, 5) << endl;   // Output: -1

    cout << "Test 3 (Empty array): " << searchValue(30, numbers2, 0) << endl; // Output: -1

    cout << "Test 4 (Negative numbers, target=-20): " << searchValue(-20, numbers3, 3) << endl;
// Output: 1

    cout << "Test 5 (Single element, target=10): " << searchValue(10, numbers1, 1) << endl;
// Output: 0

    cout << "Test 6 (target=10, First element): " << searchValue(10, numbers1, 5) << endl;
// Output: 0
```

```
    cout << "Test 7 (target=50, Last element): " << searchValue(50, numbers1, 5) << endl;
// Output: 4

    cout << "Test 8 (Empty array): " << searchValue(20, numbers2, 0) << endl;
// Output: -1

    cout << "Test 9 (target=60, Not found): " << searchValue(60, numbers1, 5) << endl;
// Output: -1

    return 0;

}
```

P2. The function countItem returns the number of times a value v appears in an array of integers a.

```
int countItem(int v, int a[])
{
   int count = 0;
   for (int i = 0; i < a.length; i++)
   {
      if (a[i] == v)
         count++;
   }
   return (count);
}
```

Equivalence Partitioning Test cases:

No.	Test Description	Input Array(a)	v	Output
1.	Value appears multiple times in the array	{1, 2, 3, 4, 2}	2	2
2.	Value appears once in the array	{1, 2, 3, 4, 5}	5	1
3.	Value does not appear in the array	{1, 2, 3, 4, 5}	6	0
4.	Single element array, value matches	{7}	7	1
5.	Single element array, value does not match	{7}	9	0
6.	Array with all elements the same, value matches	{3, 3, 3, 3, 3}	3	5
7.	Array with all elements the same, value doesn't match	{3, 3, 3, 3, 3}	5	0
8.	Large array, value appears multiple times	{1, 2, 3, ..., 500, 2}	2	2

Boundary Value Analysis Test cases:

No.	Test Description	Input Array(a)	v	Output
1.	Boundary case, array length of 1, value does not match	{1}	2	0
2.	Minimum boundary: array of size 1, value found	{1}	1	1
3.	Empty array	{}	5	0
4.	Maximum boundary: large array, value found once	{1, 2, 3, ..., 10000}	9999	1
5.	Maximum boundary: large array, value not found	{1, 2, 3, ..., 10000}	1000 1	0
6.	Large array, value at the last index	{1, 2, 3, ..., 1000}	1000	999
7.	Minimum value: array contains smallest integer value (INT_MIN)	{INT_MIN, 0, 1, INT_MAX}	INT_ MIN	1
8.	Maximum value: array contains largest integer value (INT_MAX)	{INT_MIN, 0, 1, INT_MAX}	INT_ MAX	1

Modified Programm:

```
#include <iostream>

using namespace std;

int countItem(int target, int array[], int size)

{

   int count = 0;

   for (int i = 0; i < size; i++)

```

```
    {
        if (array[i] == target)
            count++;
    }
    return count;
}
int main()
{
    int a1[] = {1, 2, 1, 4, 1};
    int a2[] = {};
    int a3[] = {-1, -2, -1};
    int a4[] = {2};
    int a5[] = {1};
    cout << "Test 1 (v=1): " << countItem(1, a1, 5) << endl;                    // output: 3
    cout << "Test 2 (v=6): " << countItem(6, a1, 5) << endl;                    // output: 0
    cout << "Test 3 (Empty array): " << countItem(3, a2, 0) << endl;            // output: 0
    cout << "Test 4 (Negative numbers): " << countItem(-1, a3, 3) << endl;      // output: 2
    cout << "Test 5 (Single element): " << countItem(2, a4, 1) << endl;         // output: 1
    cout << "Test 6 (Single element not found): " << countItem(2, a5, 1) << endl; // output: 0
    cout << "Test 7 (v=1, First element): " << countItem(1, a1, 5) << endl;     // output: 3
    cout << "Test 8 (v=3, Last element): " << countItem(3, a1, 5) << endl;      // output: 0
    cout << "Test 9 (Empty array): " << countItem(2, a2, 0) << endl;            // output: 0
    cout << "Test 10 (v=4, Not found): " << countItem(4, a1, 5) << endl;        // output: 0
    return 0;
}
```

P3. 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 are sorted in non-decreasing order.

```
int binarySearch(int v, int a[])

{

   int lo, mid, hi;

   lo = 0;

   hi = a.length - 1;

   while (lo <= hi)

   {

      mid = (lo + hi) / 2;

      if (v == a[mid])

         return (mid);

      else if (v < a[mid])

         hi = mid - 1;

      else

         lo = mid + 1;

   }

   return (-1);

}
```

Equivalence Partitioning Test cases:

No.	Test Description	Input Array(a)	v	Output
1.	Value in the middle of the array	{1, 2, 3, 4, 5}	3	2
2.	Value not in the array	{1, 2, 3, 4, 5}	6	-1
3.	Empty array	{}	5	-1
4.	Single element array, value not found	{7}	9	-1
5.	Single element array, value found	{7}	9	0
6.	Large array, value in the middle	{10, 20, 30, ..., 1000}	500	49
7.	Large array, value not present	{10, 20, 30, ..., 1000}	1500	-1
8.	Array with duplicate values, first match	{1, 3, 3, 4, 5}	3	1
9.	Array with negative numbers, value present	{-5, -4, -3, -2, -1}	-3	2
10.	Array with both positive and negative numbers	{-5, -1, 0, 1, 5}	0	2

Boundary Value Analysis Test cases:

No.	Test Description	Input Array(a)	v	Output
1.	Value at the start of the array	{1, 2, 3, 4, 5}	1	0
2.	Value at the end of the array	{1, 2, 3, 4, 5}	5	4
3.	Empty array	{}	5	-1
4.	Single element array, value not	{7}	9	-1

	found			
5.	Boundary case, array length of 1, value not present	{1}	2	-1
6.	Large array, value at the last index	{1, 2, 3, ..., 1000}	100 0	999

Modified Programm:

```
#include <iostream>

#include <vector>

using namespace std;

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; // Value found at index mid

      }

      else if (a[mid] < v)

      {

         left = mid + 1; // Search in the right half

      }
```

```
        else

        {

            right = mid - 1; // Search in the left half

        }

    }

    return -1; // Value not found

}

int main()

{

    // Test cases

    vector<int> arr1 = {};                          // Empty array

    vector<int> arr2 = {1, 2, 3, 5, 6};              // Value is present

    vector<int> arr3 = {1, 2, 3, 4, 6};              // Value is not present

    vector<int> arr4 = {5};                          // Single element, value present

    vector<int> arr5 = {3};                          // Single element, value not present

    cout << "TC1: " << binarySearch(arr1, 5) << endl; // output -1

    cout << "TC2: " << binarySearch(arr2, 5) << endl; // output 3

    cout << "TC3: " << binarySearch(arr3, 5) << endl; // output -1

    cout << "TC4: " << binarySearch(arr4, 5) << endl; // output 0

    cout << "TC5: " << binarySearch(arr5, 5) << endl; // output -1

    return 0;

}
```

P4. The following problem has been adapted from The Art of Software Testing, by G. Myers (1979). 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 , isosceles (two lengths equal), scalene , or invalid.

```
final int EQUILATERAL = 0;

final int ISOSCELES = 1;

final int SCALENE = 2;

final int INVALID = 3;

int triangle(int a, int b, int c)

{

   if (a >= b + c || b >= a + c || c >= a + b)

      return (INVALID);

   if (a == b && b == c)

      return (EQUILATERAL);

   if (a == b || a == c || b == c)

      return (ISOSCELES);

   return (SCALENE);

}
```

Equivalence Partitioning Test cases:

No.	Test Description	Input (a,b,c)	Output
1.	Equilateral triangle (all sides equal)	(3, 3, 3)	EQUILATERAL
2.	Isosceles triangle (two sides equal)	(3, 3, 4)	ISOSCELES

3.	Scalene triangle (no sides equal)	(3, 4, 5)	SCALENE
4.	Invalid triangle (sum of two sides ≤ third)	(1, 2, 3)	INVALID
5.	Equilateral triangle with larger values	(1000, 1000, 1000)	EQUILATERAL
6.	(-1, 3, 4)	(-1, 3, 4)	INVALID

Boundary Value Analysis Test cases:

No.	Test Description	Input(a,b,c)	Output
1.	Invalid triangle (one side zero)	(0, 3, 4)	INVALID
2.	Isosceles triangle at boundary value	(2, 2, 1)	ISOSCELES
3.	Scalene triangle at boundary value	(2, 3, 4)	SCALENE
4.	Invalid triangle at boundary (sides sum = third)	(5, 5, 10)	INVALID

Modified Programm:

```
#include <iostream>

using namespace std;

const char *triangle(int a, int b, int c)

{

    if (a <= 0 || b <= 0 || c <= 0 || a + b <= c || a + c <= b || b + c <= a)

    {

        return "Invalid";

    }
    if (a == b && b == c)

    {

        return "Equilateral";

    }

    if (a == b || b == c || a == c)

    {

        return "Isosceles";

    }

    return "Scalene";

}

int main()

{

    cout << "Test 1: " << triangle(3, 3, 3) << endl;  // Output: Equilateral

    cout << "Test 2: " << triangle(4, 4, 5) << endl;  // Output: Isosceles

    cout << "Test 3: " << triangle(3, 4, 5) << endl;  // Output: Scalene

    cout << "Test 4: " << triangle(1, 2, 3) << endl;  // Output: Invalid

    cout << "Test 5: " << triangle(-1, 2, 3) << endl; // Output: Invalid

    cout << "Test 6: " << triangle(0, 2, 2) << endl;  // Output: Invalid

    cout << "Test 7: " << triangle(1, 1, 1) << endl;  // Output: Equilateral

    cout << "Test 8: " << triangle(1, 1, 2) << endl;  // Output: Invalid

    cout << "Test 9: " << triangle(-1, 1, 1) << endl; // Output: Invalid

    cout << "Test 10: " << triangle(0, 1, 1) << endl; // Output: Invalid

    cout << "Test 11: " << triangle(2, 2, 2) << endl; // Output: Equilateral

    return 0;

}
```

P5. 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).

```
public
static boolean prefix(String s1, String s2)
{
    if (s1.length() > s2.length())
    {
        return false;
    }
    for (int i = 0; i < s1.length(); i++)
    {
        if (s1.charAt(i) != s2.charAt(i))
        {
            return false;
        }
    }
    return true;
}
```

By Equivalence Class:

1. s1 is longer than s2 (impossible to be a prex).
2. E2: s1 is a valid prex of s2.
3. E3: s1 is not a prex of s2.
4. E4: s1 is an empty string (edge case).
5. E5: s2 is an empty string (edge case)

Equivalence Partitioning Test cases:

No.	Input(s1.s2)	Output	Covered Equivalence class
1.	"abcdef", "abc"	false	E1
2.	"abc", "abcdef"	true	E2
3.	"xyz", "abcdef"	false	E3
4.	"", "abcdef"	true	E4
5.	"abc", ""	false	E5

Boundary Value Analysis Test cases:

No.	Input(s1.s2)	Output	Boundary Condition
1.	“a”,” ”	false	S2 is empty
2.	"abcdef","abcdef"	true	s1 equals s2
3.	"abc", "abc"	true	Shorter but equal strings
4.	"", ""	true	Both strings are empty

Modified Programm:

```
#include <iostream>

#include <string>

using namespace std;

bool prefix(string s1, string s2)

{

    if (s1.length() > s2.length())

    {

        return false;

    }

    for (int i = 0; i < s1.length(); i++)

    {

        if (s1[i] != s2[i])

        {

            return false;

        }

    }

    return true;

}

int main()

{

    // Equivalence Partitioning Test Cases

    cout << "TC1: " << (prefix("abcdef", "abc") ? "true" : "false") << endl; // output false

    cout << "TC2: " << (prefix("abc", "abcdef") ? "true" : "false") << endl; // output true

    cout << "TC3: " << (prefix("xyz", "abcdef") ? "true" : "false") << endl; // output false
    cout << "TC4: " << (prefix("", "abcdef") ? "true" : "false") << endl;

    // output true

    cout << "TC5: " << (prefix("abc", "") ? "true" : "false") << endl;

    // output false

    // Boundary Value Test Cases

    cout << "TC6: " << (prefix("a", "") ? "true" : "false") << endl;

    // output false

    cout << "TC7: " << (prefix("abcdef", "abcdef") ? "true" : "false") << endl; // output true

    cout << "TC8: " << (prefix("abc", "abc") ? "true" : "false") << endl;

    // output true

    cout << "TC9: " << (prefix("", "") ? "true" : "false") << endl;

    // output true

    return 0;

}
```

P6: Consider again the triangle classification program (P4) with a slightly different specification: Theprogram reads floating values from the standard input. The three values A, B, and C are interpretedas representing the lengths of the sides of a triangle. The program then prints a message to thestandard 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:

a) Identify the equivalence classes for the system

b) 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)

c) For the boundary condition A + B > C case (scalene triangle), identify test cases to verify the boundary.

d) For the boundary condition A = C case (isosceles triangle), identify test cases to verify the boundary.

e) For the boundary condition A = B = C case (equilateral triangle), identify test cases to verify the boundary.

f) For the boundary condition A2 + B2 = C2 case (right-angle triangle), identify test cases to verify the boundary.

g) For the non-triangle case, identify test cases to explore the boundary.

h) For non-positive input, identify test points.

Answer :

a) Equivalence Class:

1. Valid equilateral triangle: All sides are equal.
2. Valid isosceles triangle: Exactly two sides are equal.
3. Valid scalene triangle: All sides are different.
4. Valid right-angled triangle: Follows the Pythagorean theorem.
5. Invalid triangle (non-triangle): Sides do not satisfy triangle inequalities.
6. Invalid input (non-positive values): One or more sides are non-positive.

b) Equivalence Partitioning Test cases:

No.	Input(a.b,c)	Output	Covered Equivalence class
1.	A = 3.0, B = 3.0, C =3.0	Equilateral	E1
2.	A = 4.0, B = 4.0, C = 5.0	Isosceles	E2
3.	A = 3.0, B = 4.0, C = 5.0	Scalene	E3
4.	A = 3.0, B = 4.0, C = 6.0	Invalid	E5
5.	A = -1.0, B = 2.0, C = 3.0	Invalid	E6
6.	A = 5.0, B = 12.0, C = 13.0	Right-angled	E4

Boundary Value Analysis Test cases:

c) For the boundary condition A + B > C case (scalene triangle), identify test cases to verify the boundary.

No.	Input(a.b,c)	Output
1.	A = 1.0, B = 1.0, C = 1.9999	Scalene
2.	A = 2.0, B = 3.0, C = 4.0	Scalene

d) For the boundary condition A = C case (isosceles triangle), identify test cases to verify the boundary.

No.	Input(a.b,c)	Output
1.	A = 3.0, B = 3.0, C = 4.0	Isosceles
2.	A = 2.0, B = 2.0, C = 3.0	Isosceles
3.	A = 2.0, B = 2.0, C = 2.0	Equilateral

e) For the boundary condition A = B = C case (equilateral triangle), identify test cases to verify the boundary.

No.	Input(a.b,c)	Output
1.	A = 2.0, B = 2.0, C = 2.0	Equilateral
2.	A = 1.9999, B = 1.9999, C = 1.9999	Equilateral

f) For the boundary condition A2 + B2 = C2 case (right-angle triangle), identify test cases to verify the boundary.

No.	Input(a.b,c)	Output
1.	A = 3.0, B = 4.0, C = 5.0	Right-angled
2.	A = 5.0, B = 12.0, C = 13.0	Right-angled

g) For the non-triangle case, identify test cases to explore the boundary.

No.	Input(a.b,c)	Output
1.	A = 1.0, B = 2.0, C = 3.0	Invalid
2.	A = 1.0, B = 2.0, C = 2.0	Invalid
3.	A = 1.0, B = 1.0, C = 3.0	Invalid

h) For non-positive input, identify test points.

No.	Input(a.b,c)	Output
1.	A = 0.0, B = 2.0, C = 3.0	Invalid
2.	A = -1.0, B = -2.0, C = 3.0	Invalid
3.	A = 3.0, B = 0.0, C = 2.0	Invalid

Modified Programm:

```
#include <iostream>
#include <cmath>
using namespace std;
const char *classifyTriangle(oat A, oat B, oat C)
{
   if (A <= 0 || B <= 0 || C <= 0 || A + B <= C || A + C <= B || B + C <= A)
      return "Invalid";
   if (fabs(pow(A, 2) + pow(B, 2) - pow(C, 2)) < 1e-6 ||
      fabs(pow(A, 2) + pow(C, 2) - pow(B, 2)) < 1e-6 ||
      fabs(pow(B, 2) + pow(C, 2) - pow(A, 2)) < 1e-6)
   {
      return "Right-angled";
   }
   if (A == B && B == C)
   {
      return "Equilateral";
   }
   if (A == B || B == C || A == C)
   {
      return "Isosceles";
   }
   return "Scalene";
}
int main()
{
   cout << "Test 1: " << classifyTriangle(3.0, 3.0, 3.0) << endl;   // Output:
Equilateral
   cout << "Test 2: " << classifyTriangle(4.0, 4.0, 5.0) << endl;   // Output:
Isosceles
   cout << "Test 3: " << classifyTriangle(3.0, 4.0, 5.0) << endl;   // Output:
Scalene
   cout << "Test 4: " << classifyTriangle(3.0, 4.0, 6.0) << endl;   // Output:
Invalid
   cout << "Test 5: " << classifyTriangle(-1.0, 2.0, 3.0) << endl;  // Output: Invalid
   cout << "Test 6: " << classifyTriangle(0.0, 2.0, 2.0) << endl;   // Output: Invalid
   cout << "Test 7: " << classifyTriangle(5.0, 12.0, 13.0) << endl; // Output:
Right-angled
   return 0;
}
```