IT 314

Software Engineering

Lab 8

Supervised by:
Prof. Saurabh Tiwari

Submitted by:
Kavan Kansodariya: 202201223

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.
Ans:

Equivalence Class Partitioning Test Case:

INPUT	EXPECTED OUTCOME	REASONING
15,5,2000	valid	Valid date
29,2,1900	invalid	Invalid leap year case (1900 is not a leap year)
32,5,2000	invalid	Invalid day
0,6,2005	invalid	Invalid day
1,0,1999	invalid	Invalid month
21,2,2003	Valid	Typical valid date
15,13,2000	Invalid	Invalid month
15,3,2004	valid	Valid date
7,10,2004	Valid	Valid date

Boundary value analysis test cases:

Input	Expected outcome	Reason
0,1,2000	Invalid	Invalid date
32,1,2000	Invalid	Invalid date
1,3,2000	Valid	Leap year boundary case
31,12,2015	Valid	Upper boundary for year
1,1,1900	Valid	Lower boundary for the year
29,2,1900	Invalid	Invalid leap year boundary
28,2,2001	Valid	Valid boundary for Feb in non leap year
29,2,2001	Invalid	Invalid date
31,4,2000	Invalid	Invalid date
30,4,2000	Valid	Valid boundary for 30 day month April

Code:

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

bool checkLeapYear(int yr)
{
   if (yr % 400 == 0 || (yr % 4 == 0 && yr % 100 != 0))
      return true;
   return false;
}
```

```
int getDaysInMonth(int mon, int yr)
{
   if (mon == 2)
      return checkLeapYear(yr) ? 29 : 28;
   if (mon == 4 || mon == 6 || mon == 9 || mon == 11)
      return 30;
   return 31;
}

void findPreviousDate(int d, int mon, int yr)
{
   if (yr < 1900 || yr > 2015 || mon < 1 || mon > 12 ||
      d < 1 || d > getDaysInMonth(mon, yr))
   {
      cout << "Error: Invalid date" << endl;
      return;
   }
   d--;
   if (d == 0)
   {
      mon--;
      if (mon == 0)
      {
         mon = 12;
         yr--;
      }
      d = getDaysInMonth(mon, yr);
   }
   if (yr < 1900)
   {
      cout << "Error: Invalid date" << endl;
```

```
        return;
    }
    cout << "Previous date is: " << d << "/" << mon << "/" << yr << endl;
}

int main()
{
    findPreviousDate(1, 1, 2001);
    return 0;
}
```

Q2. 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);
}
```

Value present:

- **E1: the value v is present in the array and occurs multiple times.**
- **E2: the value v is not present in the array.**
- **E3: The value v is present in the array and occurs once.**

Array Edge cases:

- **E4: The array is empty.**
- **E5: The value v is at the first or last position in the array.**

Equivalence classes:

Test cases	Input	Expected outcome	Equivalence boundary
TC1	v=1,[1,2,3,4,5]	0	E5
TC3	v=2,[]	-1	E4
TC3	v=5,[1,2,3,4,5]	4	E1
TC4	v=3,[1,3,3,3,5]	1	E2
TC5	v=6,[1,2,3,4,5]	-1	E3

Boundary points:

- **BP1:Single-element array where v is present.**
- **BP2:Single-element array where v is not present.**
- **BP3:v is at the first position.**
- **BP4:v is at the last position.**
- **BP5:Array contains negative numbers and v is a negative number.**

Test case	Input	Expected outcomes	Equivalence boundary
BP1	v=3,[3]	0	BP1
BP2	v=2,[1]	1	BP2
BP3	v=1,[1,2,3,4,5]	0	BP3
BP4	v=5,[1,2,3,4,5]	4	BP4
BP5	v=-3,[-5,-4,-3,-2,-1]	2	BP5

P2.The function counitem 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);
}
```

Value Present:

- **E1:The value v is present in the array and occurs once.**
- **E2:The value v is present in the array and occurs multiple times.**
- **E3:The value v is not present in the array.**

Array Edge Cases:

- **E4:The array is empty.**
- **E5:The value v is at the first or last position in the array.**

Equivalence Classes:

Test cases	Inputs	Expected outcomes	Equivalence boundary
TC1	v=2,[]	0	E4
TC2	v=1,[1,2,3,4,5]	1	E5
TC3	v=6,[1,2,3,4,5]	0	E3
TC4	v=3,[1,3,3,3,5]	3	E2

TC5	v=5,[1,2,3,4,5]	1	E1

Boundary points for countItem:

1. **BP1: Single-element array where v is present.**
2. **BP2: Single-element array where v is not present.**
3. **BP3: v is at the first position.**
4. **BP4: v is at the last position.**
5. **BP5: Array contains negative numbers and v is a negative number**

Test cases	Inputs	Expected outcomes	Equivalence Boundary
BP1	v=3,[3]	1	BP1
BP5	v=-3,[-5,-4,-3,-2,-1]	1	BP5
BP2	v=2,[1]	0	BP2
BP3	v=1,[1,2,3,4,5]	1	BP3
BP4	v=5,[1,2,3,4,5]	1	BP4

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.

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

    int lo, mid, hi;
    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 Classes for binarySearch:

1. **Value Present:**
   a. **E1:The value v is present in the array and is located at the first position.**
   b. **E2: The value v is present in the array and is located at the last position.**
   c. **E3:The value v is present in the array and is located somewhere in the middle.**
2. **Value not present:**
   a. **E4: The value v is less than the smallest element in the array.**
   b. **E5: The value v is greater than the largest element in the array.**
   c. **E6: The value v is not in the array but falls between two elements.**

3. **Array Edge cases:**
    a. **E7: The array is empty.**
    b. **E8: The array contains one element,which may or may not be equal to v.**

Equivalence Classes:

Test case	Input	Expected outcomes	Equivalence Boundary
TC1	v=1,[1,2,3,4,5]	0	E1
TC2	v=5,[1,2,3,4,5]	4	E2
TC3	v=3,[1,2,3,4,5]	2	E3
TC4	v=0,[1,2,3,4,5]	-1	E4
TC5	v=6,[1,2,3,4,5]	-1	E5
TC6	v=2.5,[1,2,3,4,5]	-1	E6
TC7	v=3,[]	-1	E7
TC8	v=1,[1]	0	E8
TC9	v=2,[1]	-1	E8

Test cases	Inputs	Expected outcomes	Equivalence Boundary
BP1	v=3,[3]	0	BP1
BP2	v=2,[3]	-1	BP2
BP3	v=1,[1,2,3,4,5]	0	BP3
BP4	v=5,[1,2,3,4,5]	4	BP4
BP5	v=3,[1,2,3,3,3,4,5]	2	BP5

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 (three lengths equal), isosceles (two lengths equal), scalene (no lengths equal), or invalid (impossible lengths).

Equivalence partitioning:

Input Data	Expected Outcome
3, 3, 3	EQUILATERAL (0)
3, 3, 2	ISOSCELES (1)
3, 4, 5	SCALENE (2)
1, 2, 3	INVALID (3)
1, 1, 2	INVALID (3)
5, 1, 1	INVALID (3)
2, 2, 3	ISOSCELES (1)
0, 1, 1	An Error message
1, 0, 1	An Error message

Boundary value analysis:

Input Data	Expected Outcome
1, 1, 1	EQUILATERAL (0)
1, 1, 2	INVALID (3)
2, 2, 4	INVALID (3)
2, 3, 5	INVALID (3)
3, 4, 7	INVALID (3)
1, 2, 2	ISOSCELES (1)

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).

1, 2, 3	**INVALID (3)**
0, 1, 1	**An Error message**
1, 1, 0	**An Error message**

Equivalence Partitioning:

Input Data	**Expected Outcome**
"pre", "prefix"	**true**
"pre", "postfix"	**false**
"prefix", "pre"	**false**
"test", "test"	**true**
"", "anything"	**true**
"anything", ""	**false**
"pre", "preparation"	**true**
null, "prefix"	**An Error message**
"prefix", null	**An Error message**

Boundary value analysis:

Input Data	**Expected Outcome**
"test", ""	**false**
"a", "a"	**true**
"a", "b"	**false**
"", ""	**true**
"start", "startmiddle"	**true**
"longprefix", "short"	**false**

"short", "longprefix"	**true**
null, "anything"	**An Error message**
"anything", null	**An Error message**

P6:

1. **Identify the equivalence classes:**

Equilateral Triangle: All three sides are equal.

Isosceles Triangle: Exactly two sides are equal.

Scalene Triangle: No sides are equal.

Right-Angled Triangle: Satisfies a2+b2=c2.

Invalid Triangle: Does not satisfy the triangle inequality a+b>c.

Non-positive Input: One or more sides are non-positive.

Identify Test Cases to Cover the Equivalence Classes
Equivalence Partitioning:

Input Data	Expected Outcome	Equivalence Class
3.0, 3.0, 3.0	Equilateral	Equilateral Triangle
3.0, 3.0, 2.0	Isosceles	Isosceles Triangle
3.0, 4.0, 5.0	Scalene	Scalene Triangle
3.0, 4.0, 0.0	Invalid	Invalid Triangle

0.0, 0.0, 0.0	Invalid	Non-positive Input
5.0, 1.0, 1.0	Invalid	Invalid Triangle
3.0, 4.0, 6.0	Scalene	Scalene Triangle

Boundary Condition A + B > C (Scalene- Triangle)
Boundary Value Analysis:

Input Data	Expected Outcome
2.0, 2.0, 3.99	Scalene
2.0, 2.0, 4.0	Invalid
2.0, 2.0, 4.01	Invalid

Boundary Condition A = C (Isosceles Triangle)
Boundary Value Analysis

Input Data	Expected Outcome
3.0, 4.0, 3.0	Isosceles
3.0, 3.0, 3.0	Equilateral
Isosceles	Isosceles

Boundary Condition A = B = C (Equilateral Triangle) Boundary Value Analysis:

Input Data	Expected Outcome
3.0, 3.0, 3.0	Equilateral
1.0, 1.0, 1.0	Equilateral
2.5, 2.5, 2.5	Equilateral

Boundary Condition A2+B2=C2 (Right-Angle Triangle) Boundary Value Analysis:

Input Data	Expected Outcome
3.0, 4.0, 5.0	Right Angled
6.0, 8.0, 10.0	Right Angled
5.0, 12.0, 13.0	Right Angled

Non-Triangle Case Boundary Value Analysis:

Input Data	Expected Outcome
1.0, 2.0, 3.0	Invalid
1.0, 2.0, 4.0	Invalid
1.0, 1.0, 2.0	Invalid

Non-Positive Input Boundary Value Analysis:

Input Data	Expected Outcome
0.0, 1.0, 1.0	Invalid
-1.0, 1.0, 1.0	Invalid
1.0, 0.0, 1.0	Invalid