Elmar Eisemann

Introduction to C++

Practical Exercise

Introduction

Code organization

This exercise sheet will consist of several problems that need to be solved using c++.

You will be given a **header file** which contains the skeleton of the functions and classes that you need to define in order to solve the exercises. This is the **only** file that you need to modify and hand in.

You will also be given a **code file** that will use those functions with an example input. The program in the code file will run the functions you defined in the header file for the test cases given for the exercises, so you can verify that your solution works properly. Beware, though, that during grading a different code file will be used so the functions and classes you define will receive a different set of inputs to verify that they work correctly.

Exercises

1. Statistics

This function is defined in the header file as

```
std::pair<float, float> Statistics(std::span<const float> values)
```

The input to the function will be a list of float values (which are not a special float value, such as NaN, +inf or -inf).

The output is a pair of values, where the first value is the average of the values in the list, and the second value is the standard deviation of the values. If the list is empty, it should return 0 for both values.

As an example, given the list `[-4,-2,0,0,0,2,2,2]`, the function should return the pair `(0.0, 2.0)`.

2. Tree traversal

This function is defined in the code file as

```
float TreeTraversal(const Tree& t, TreeVisitor& visitor, bool countOnlyEvenLevels)
{
    return visitor.visitTree(t, countOnlyEvenLevels);
}
```

The input to the function will be a tree object, a tree visitor object and a boolean value defining the behavior of the tree visitor.

The `Tree` structure is defined in the header file as:

```
struct Tree {
    float value;
    std::list<Tree> children;
};
```

This class encodes a tree structure, where each tree can contain other trees as children, and contains a value.

The `TreeVisitor` class is partially defined as:

```
class TreeVisitor {
    public:
        float visitTree(const Tree& tree, bool countOnlyEvenLevels);
}
```

In this exercise you do not need to modify the `TreeTraversal` function. Instead, you will need to write the function `TreeVisitor::visitTree` (and add any other members or functions you want to the class), so that when called, the visitor will go through the tree and compute the sum of all the values stored in it. If the boolean value `countOnlyEvenLevels` is set to true, then only the values on even levels of the tree should be included in the sum (the root is at level 0 and considered an even level).

3. Complex numbers

This function is defined in the code file as

```
Complex ComplexOperation(Complex c1, Complex c2)
{
  return Complex(0,0);
//    return (c1-c2) * (c1+c2);
}
```

The class `Complex` is partially defined as:

```
class Complex {
public:
    Complex(float real, float imaginary){};

    float real, im;
};
```

Note that the second line of the `ComplexOperation` function is commented. This is because until you define `Complex` appropriately to support the operations in the commented line, the program will not compile. Therefore, you need to change that line in order for the program to properly test your output.

The objective of this exercise is to define the appropriate operations in `Complex` so that the addition, the subtraction and the product of complex numbers work in the standard way. For example, if given as input the complex numbers $1 + 2i$ and $5 - 3i$, it should return $-19 + 34i$.

Do not remove or rename the variables `real` and `im` from the class `Complex` because they are used to verify the final result.

4. Water levels

This function is defined in the header file as

```
float WaterLevels(std::span<const float> heights)
```

The parameter `heights` defined in this exercise describes the height of a surface at successive points. Figure **??** shows as an example the surface defined by the list `7,6,7,3,8,7,8,9,6`.

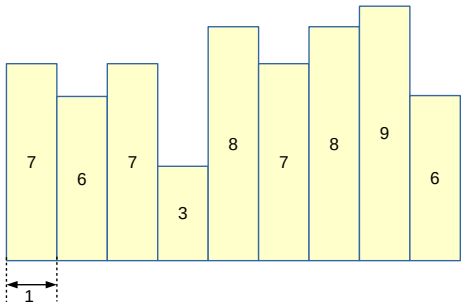

Figure 1: Surface defined by `heights`.

The goal of this exercise is to compute the volume of water that this structure can hold. As an example, see the figure **??**. In that case, the input list would be `[7,6,4,7,6,9,3,1,5,3]`, and the expected result is 11.

Note that no water can be held at both end points.

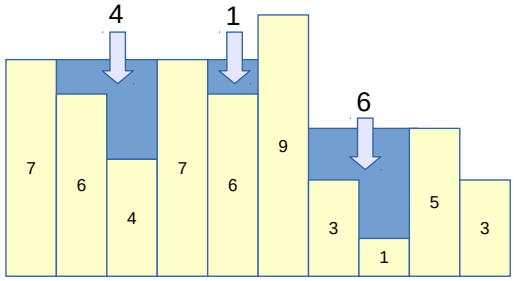

Figure 2: Example of the expected result.

5. Labyrinth

This function is defined in the header file as

```
using location = std::pair<int, int>;

int Labyrinth(const std::set<std::pair<location, location>>& labyrinth, int size)
```

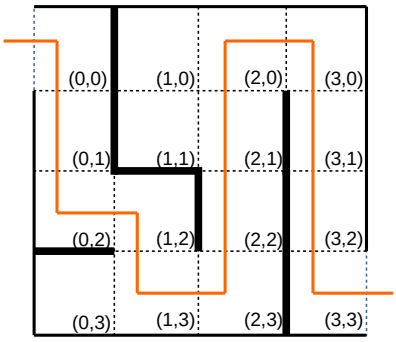

Figure 3: Example of a labyrinth.

The objective of this exercise is to find the way out of a labyrinth. The labyrinth is a square grid of values of size `size * size`. One can only move from a square to an adjacent square (not diagonally). The set `labyrinth` contains the pairs of adjacent locations that are blocked from one another by a wall. The edges of the labyrinth are assumed to be blocked as well. The goal of this exercise is to find the length of the shortest path to get from location `(0,0)` to location `(size-1,size-1)`. If no path is possible, then return 0.

As an example, figure **??** shows a labyrinth of size 4. The set that describes this labyrinth is: [1]

```
[( (0,0) , (1,0) ),
 ( (0,1) , (1,1) ),
 ( (0,2) , (0,3) ),
 ( (1,1) , (1,2) ),
 ( (1,2) , (2,2) ),
 ( (2,3) , (3,3) ),
 ( (2,2) , (3,2) ),
 ( (2,1) , (3,1) )].
```

In the example figure, the expected result is 13.

[1] Beware that for each wall only one of the combinations is in the set.