Introduction to Computer Science I

Summer Sessions 2021 - COM SCI31-1 - STAHL

Project 3

Programming Assignment 3
Elevator

Time due: 9:00 PM Wednesday, July 14th

(For clarity purposes in all the examples that follow, the character Oh has been underlined, as in: O Otherwise, anything non-underlined represents the number zero, as in: 0)

Introduction

Elevator conveyer systems date back to Roman times, as many different societies realized the need to lift and lower people and things.

For this assignment, suppose that a sensor is recording the different tasks an elevator performs. For example, consider the elevator string:

```
M1
```

This string indicates that the elevator has been called to first floor. At this point, suppose the doors open and three people come aboard. This would correspond to the elevator string:

```
M1O+3
```

Alternatively, an equivalent elevator string could say:

```
M1O+1+1+1
```

Now suppose the elevator closes it doors and moves to floor 5. This would correspond to the elevator string:

```
M1O+1+1+1CM5
```

At this point, suppose two people walk off the elevator and then four new people come aboard. This would correspond to the elevator string:

```
M1O+1+1+1CM5O-2+4
```

Alternatively, an equivalent elevator string could say:

```
M1O+3CM5O+1+1+1+1-1-1
```

Precisely, to be a valid elevator string,

- the only characters present should be M, O, C, +, - and digit characters
- by convention, elevators always start with the doors closed
- an elevator string must begin with M followed by digits
- elevators doors can only be opened if they are currently closed; elevator doors can only be closed if they are currently open
- once elevator doors are opened, passengers can board via + followed by digits
- once elevator doors are opened, passengers can leave via - followed by digits
- doors must be closed before the elevator can be called to a different floor
- elevator strings must end with either O, C or a digit character and not a +, - or M

All of the following are examples of valid elevator strings:

- M12 (elevator called to floor 12)
- M12O+1-0C (elevator called to floor 12, doors open, one passenger aboard, doors close)
- M12O+1CM3 (elevator called to floor 12, doors open, one passenger aboard, doors close, elevator called to floor 3)
- M0 (floor zero is allowed and valid)
- M1O+0004C (extra leading zeros are allowed)
- M3O+3COCOCM2 (door can be opened if presently closed, doors can be closed if presently open)
- M12O+1-1+1-1+1 (folks can get on and off)
- M10M8M9 (moving between floors)
- M2OCOC (valid)
- M2OCM1 (valid)
- M2OCO (valid)
- M2O+3CO-1CO (valid)

All of the following are examples of invalid elevator strings:

- M (elevator needs to be called to a floor)
- M1C (elevator start with doors closed; once closed, doors cannot be closed again)
- M1OCOO (once doors are open, doors cannot be opened again)
- M1O-12+50 (passenger count cannot go negative as the elevator string is processed)
- M1O+4-3-1CM2O-1 (passenger count cannot go negative as the elevator string is processed)
- M1O+-C (+ and - must be followed by digit characters)
- M-3 (only above ground floors)

Your task

For this project, you will implement the following six functions, using the exact function names, parameter types, and return types shown in this specification. (The parameter *names* may be different if you wish).

`bool isValidElevatorString(string elevatorstring)`
This function returns `true` if its parameter is a well-formed elevator string as described above, or `false` otherwise.

`bool doorsOpen(string elevatorstring)`
If the parameter is a well-formed elevator string, this function should return `true` if the elevator doors are open at the end of processing the entire elevator string; return `false` otherwise.

`int endingPassengers(string elevatorstring)`
If the parameter is a well-formed elevator string, this function should return the number of passenger aboard the elevator at the end of processing the entire elevator string. If the parameter is not a valid elevator string, return `-1`.

`int mostPassengers(string elevatorstring)`
If the parameter is a well-formed elevator string, this function should return the greatest number of passengers ever aboard the elevator as the elevator string is processed. If the parameter is not a valid elevator string, return `-1`.

`int endingFloor(string elevatorstring)`

If the parameter is a well-formed elevator string, this function should return the floor the elevator ended on at the end of processing the entire elevator string. If the parameter is not a valid elevator string, return `-1`.

`int highestFloor(string elevatorstring)`
If the parameter is a well-formed elevator string, this function should return the greatest floor the elevator moved to as the elevator string is processed. If the parameter is not a valid elevator order string, return `-1`.

These are the only six functions you are required to write. Your solution may use functions in addition to these six if you wish. While we won't test those additional functions separately, using them may help you structure your program more readably. Of course, to test them, you'll want to write a main routine that calls your functions. During the course of developing your solution, you might change that main routine many times. As long as your main routine compiles correctly when you turn in your solution, it doesn't matter what it does, since we will rename it to something harmless and never call it (because we will supply our own main routine to thoroughly test your functions).

Clearly, I am expecting the last five functions to invoke the first function as they complete their assigned task. This idea of code reuse is one of the major benefits of having functions. So please do that.

Before you ask a question about this specification, see if it has already been addressed by the Project 3 FAQ. And read the FAQ before you turn in this project, to be sure you didn't misinterpret anything. Be sure you also do the warmup exercises accompanying this project.

Additionally, I have created a testing tool called CodeBoard to help you check your code. CodeBoard enables you to be sure you are naming things correctly by running a small number of tests against your code. Passing CodeBoard tests is not sufficient testing so please do additional testing yourself. To access CodeBoard for Project 3, please click the link you see in this week named CodeBoard for Project 3. Inside the file named user_functions.cpp, copy and paste your implementation of the assigned functions. CodeBoard uses its own main( ) to run tests against your code. Click Compile and Run. However please be aware that no editing changes can be saved inside CodeBoard. In this anonymous user configuration, CodeBoard is read-only and does not allow for saving changes.

In an effort to assist CS 31 students with the contents of your .zip file archive, Howard has created a Zip File Checker which will echo back to you the contents of your .zip file. Please use this file checker to ensure you have named all your files correctly.

Programming Guidelines

The functions you write must not use any global variables whose values may be changed during execution (so global *constants* are allowed).

When you turn in your solution, neither of the six required functions, nor any functions they call, may read any input from `cin` or write any output to `cout`. (Of course, during development, you may have them write whatever you like to help you debug.) If you want to print things out for debugging purposes, write to `cerr` instead of `cout`. `cerr` is the standard error destination; items written to it by default go to the screen. When we test your program, we will cause everything written to `cerr` to be discarded instead — we will never see that output, so you may leave those debugging output statements in your program if you wish.

The correctness of your program must not depend on undefined program behavior. For example, you can assume nothing about `c`'s value at the point indicated, nor even whether or not the program crashes:

```
int main()
{
    string s = "Hello";
    char c = s[5];   // c's value is undefined
    …
```

Be sure that your program builds successfully, and try to ensure that your functions do something reasonable for at least a few test cases. That way, you can get some partial credit for a solution that does not meet the entire specification.

There are a number of ways you might write your main routine to test your functions. One way is to interactively accept test strings:

```
int main()
{
    string s;
    cout.setf( ios::boolalpha ); // prints bool values as "true" or "false"
    for(;;)
    {
        cout << "Enter a possible elevator string: ";
        getline(cin, s);
        if (s == "quit") break;
        cout << "isValidElevatorString returns " << isValidElevatorString(s) << endl;
        cout << "doorsOpen(s) returns " << doorsOpen(s) << endl;
        cout << "endingPassengers(s) returns " << endingPassengers(s) << endl;
        cout << "mostPassengers(s) returns " << mostPassengers(s) << endl;
        cout << "endingFloor(s) returns " << endingFloor(s) << endl;
        cout << "highestFloor(s) returns " << highestFloor(s) << endl;
    }
   return 0;
}
```

While this is flexible, you run the risk of not being able to reproduce all your test cases if you make a change to your code and want to test that you didn't break anything that used to work.

Another way is to hard-code various tests and report which ones the program passes:

```
int main()
{
    if (!isValidElevatorString(""))
        cout << "Passed test 1: !isValidElevatorString(\"\")" << endl;
    if (!isValidElevatorString("   "))
        cout << "Passed test 2: !isValidElevatorString(\"   \")" << endl;
    …
```

This can get rather tedious. Fortunately, the library has a facility to make this easier: `assert` . If you `#include` the header `<cassert>` , you can call `assert` in the following manner:

```
assert(some boolean expression);
```

During execution, if the expression is true, nothing happens and execution continues normally; if it is false, a diagnostic message is written to `cerr` telling you the text and location of the failed assertion, and the program is terminated. As an example, here's a very incomplete set of tests:

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

…

int main()
{
    assert( isValidElevatorString("") == false );
    assert( isValidElevatorString("    ") == false );
    assert( doorsOpen( "    " ) == false );
    assert( endingPassengers( "      " ) == -1 );
    assert( mostPassengers( "      " ) == -1 );
    assert( endingFloor( "       " ) == -1 );
    assert( highestFloor( "       " ) == -1 );
    assert( isValidElevatorString( "M1O+3-2C" ) );
    assert( doorsOpen( "M1O+3-2C" ) == false );
    assert( endingPassengers( "M1O+3-2C" ) == 1 );
    assert( mostPassengers( "M1O+3-2C" ) == 3 );
    assert( endingFloor( "M1O+3-2C" ) == 1 );
    assert( highestFloor( "M1O+3-2C" ) == 1 );
    …
    cout << "All tests succeeded" << endl;
    return 0;
}
```

The reason for writing one line of output at the end is to ensure that you can distinguish the situation of all tests succeeding from the case where one function you're testing silently crashes the program.

What to turn in

What you will turn in for this assignment is a zip file containing these two files and nothing more:

1. A text file named **elevator.cpp** that contains the source code for your C++ program. Your source code should have helpful comments that tell the purpose of the major program segments and explain any tricky code. The file must be a complete C++ program that can be built and run, so it must contain appropriate #include lines, a main routine, and any additional functions you may have chosen to write.

2. A file named **report.doc** or **report.docx** (in Microsoft Word format) or **report.txt** (an ordinary text file) that contains in addition **your name** and **your UCLA Id Number** :
   a. A brief description of notable obstacles you overcame.
   b. A description of the design of your program. You should use pseudocode in this description where it clarifies the presentation.
   c. A list of the test data that could be used to thoroughly test your program, along with the reason for each test. You don't have to include the results of the tests, but you must note which test cases your program does not handle correctly. (This could happen if you didn't have time to write a complete solution, or if you ran out of time while still debugging a supposedly complete solution.)

By July 3th, there will be a link on the class webpage that will enable you to turn in your zip file electronically. Turn in the file by the due time above. Give yourself enough time to be sure you can turn something in, because we will not accept excuses like "My network connection at home was down, and I didn't have a way to copy my files and bring them to a SEASnet machine." There's a lot to be said for turning in a preliminary version of your program and report early (You can always overwrite it with a later submission). That way you have something submitted in case there's a problem later. Notice that most of the test data portion of your report can be written from the requirements in this specification, before you even start designing your program.

G31 Build Commands

```
g31 -c elevator.cpp
g31 elevator.o -o runnable
./runnable
```

Submission status

Submission status	No attempt
Grading status	Not graded
Due date	Wednesday, 14 July 2021, 9:00 PM PDT
Time remaining	1 day 5 hours
Last modified	-
Submission comments	Comments (0)

Add submission

You have not made a submission yet.

◄ Project 3 Warmup

Jump to...

Some More Things About Strings ►