Artiom Arutiunov

ID: 504597668

2/5/16

Project Writeup

a. This was the most difficult coding project I’ve had to do. The main reason I found this project more challenging than previous ones was due to the necessity of thoroughly understanding how strings work. In particular, I initially had trouble figuring out how to approach the coding for function isWellFormedCommand, until I recognized that a helper function would make building isWellFormed a lot less confusing. The most challenging aspect of writing the helper function was dealing with the logic of what was allowed and what wasn’t, and how does the position of a certain character within the string influence what can come next. For example, an optional “+” or “-“ sign means that if there is a sign, then the position of the first digit in the command would have to be 1 as opposed to 0, since the 0 spot is occupied by the sign. Additionally, I had trouble keeping track of the fact that string size and position are slightly different, as the first character in a string has the position of 0 as opposed to one, which means that the conditions for my for loops had to sometimes seem counterintuitive. Alongside keeping track of the control flow of the helper function, another difficult aspect of developing this program was building the second function, crossedOriginOnItsPath. The challenging aspect of building this function was how to convert the string of digits into an integer, which I resolved by using the “stoi” function from the ctype.h library. Lastly, building an overarching loop around the whole function to allow it to process multiple commands within a single string required me to apply some intricate logic. I decided to make the overarching loop a “while” loop, under the premise that the loop is executed insofar as the position of the start of the command is less than the size of the string. Thus, if it can be concluded that I had reached the end of the string under a control flow, then the return statement should be that the position of the start of the command is *equal* to the string size, which takes us out of the overarching while loop. Ultimately, this project required me to really think about how to structure my program and to look for ulterior functions to help me along the way, such as “tolower” and “stoi.”

b. Pseudocode of rovingrobot.cpp:

The essential design of my program is using a bunch of control flow statements, frequently one imbedded in another, coupled with overarching loops. I used a helper function to support the isWellFormedString to make it easier to build.

bool crossedOriginOnItsPath

The program crossedOriginOnItsPath runs through a giant “while” loop while the position of the start of the command is less than the size of the command (otherwise we have reached the end of the string). The program uses an inner “for” loop to embed all the digits into the variable “last position of digit.” From there, we convert the string into an integer and go through another for loop which analyzes the rest of the substring to determine where in the subsequent “switch” statement should the converted string-to-integer variable should go.

{

while position of start of command < size of command

{

for loop while string character is not a letter

{

last position of digit = digits;

}

Temporary number = substring of command (position of start of command, last position of digit - position of start of command)

Number of steps (in a direction) = convert Temporary number from string to int

for loop that determines in which way the directions go

{

if k = size of string of command

position of start of command = k

break out of loop;

if command[k] is NOT a letter

position of start of command = k

break out of loop;

switch (tolower of command[k])

case: ‘n’

count of NS = count of NS + Number of steps

case: ‘s’

count of NS = count of NS - Number of steps

case: ‘w’

count of WE = count of WE + Number of steps

case: ‘e’

count of WE = count of WE – Number of steps

}

if (NS count == 0) and (WE count == 0)

return true;

else

return false;

}

}

bool isWellFormedCommandString

The only function this program really had was to run an infinite for loop while the helper single\_command function analyzed the parsed command strings one at a time and returned an integer value to which the isWellFormed program returned a corresponding Boolean expression.

{

string workstring = command

integer position

infinite for loop

{

position = single\_command(Workstring)

if position = -1

return false;

else if position == 0

return true;

else workstring = substring of (position, size of workstring – position)

}

}

int single\_command (helper function for isWellFormed)

This was the most complicated part of the entire program, as it embeds numerous “if” statements within one another to control for possible scenarios dealing with the position of the string, the subsequent character type of the string, or the size of the string. There is no overarching loop that runs this program over, because that is the role of the isWellFormed function (which calls this function). One tricky part of the code comes at the bottom of the function definition, because if we have not reached the end of the string and there are no more directions, then we must have hit another non-alpha character, which indicates the beginning of a new command. This resulted in writing some corresponding “else” statements to the appropriate if statements.

{

if command size = 0

return false;

if command[0] contains + or –

if the command size = 1

return -1;

else

change position of first digit to 1 and last digit to 3

else

positions of first and last digit remain standard

if first digit position of command string is a digit

if it’s equal to 0

return -1;

if the size of the command consists of only that digit

return -1;

else if first digit position of command is not a digit

return -1;

for loop that looks for additional digits

if first nondigit position of command is a letter

{

change first letter to lower case

if first letter does not equal ‘n’, ‘s’, ‘e’, or ‘w’

return -1;

}

else

return -1;

if size of command = first nondigit + 1

return 0;

if position first nondigit + 1 of command is a letter

make command char 2 a lower case

if first letter = ‘n’ or ‘s’

if command char = ‘w’ or ‘s’

if second letter if last character in string

else

index of first character after 2nd letter

else

return first non digit + 1

else

return first non digit + 1

}

c. list of test data

I began by running the examples Howard provided in the Project 3 page.

assert(isWellFormedCommandString("4N-3S+12NE6W"));

assert(isWellFormedCommandString("4e-3sw+12se7sw"));

assert(isWellFormedCommandString("+4nw1e3w")); assert(isWellFormedCommandString("1Nw-2sE7nE6Sw"));

The compiler agreed with these assertions, which is correct because these are all valid strings with multiple embedded commands.

Next, I ran the examples Howard provided of non-valid commands.

assert(isWellFormedCommandString("7q+8A"));

🡪 not valid because there are undefined letters

assert(isWellFormedCommandString("+-12s "));

🡪 not valid because you can only have 1 sign before digits

assert(isWellFormedCommandString("s12e-3"));

🡪 not valid because you need at least 1 digit before every direction

assert(isWellFormedCommandString("1000s-2000N"));

🡪 not valid because you cannot have more than 3 digits per command

The compiler disagreed with these assertions, which is correct because all of these are invalid strings with various errors.

Next, I tested isWellFormedCommand with different amounts of imbedded commands, + or –‘s, lower and upper case, and different amount of directions.

assert(isWellFormedCommandString("123n-778sw900e767nw"));

assert(isWellFormedCommandString("25n68nw89ne1s-20w+21e")); assert(isWellFormedCommandString("123n+12s-1e"));

assert(isWellFormedCommandString("12n"));

assert(isWellFormedCommandString("12n123s"));

The compiler agreed with these assertions, which is correct because all of them are valid commands.

Next, I tested isWellFormedCommand with different invalid commands.

assert(isWellFormedCommandString(" 123N-778sw900e767nw"));

🡪 Invalid because of leading space

assert(isWellFormedCommandString("25n68nw89nE1s-20We+21e"));

🡪 Invalid because of “We”

assert(isWellFormedCommandString("123n+12S--1e"));

🡪 Invalid because of “—“

assert(isWellFormedCommandString("12n "));

🡪 Invalid because of ending space

assert(isWellFormedCommandString("12n123S#"));

🡪 Invalid because of #

assert(isWellFormedCommandString("12n123S$@310w "));

🡪 Invalid because of “$@”

The compiler disagreed with all of these assertions, which is correct because each of them is invalid in their own way.

I then began to test the other function, crossedOriginOnItsPath. I first tested the function for whether it correctly returns true for commands that add up to 0,0 in the coordinate plane.

assert(isWellFormedCommandString("1n1s"));

assert(crossedOriginOnItsPath("1w1e"));

assert(crossedOriginOnItsPath("1nw1se"));

assert(crossedOriginOnItsPath("1ne1sw"));

assert(crossedOriginOnItsPath("-1w-1e"));

assert(crossedOriginOnItsPath("-1n-1s"));

assert(crossedOriginOnItsPath("-1nw-1se1n1s"));

assert(crossedOriginOnItsPath("999nw999se-999n-999s"));

The compiler agreed with all of these assertions, which is correct because all of them add up to the origin when incorporating the negating “-“ signs.

Finally, I tested crossedOriginOnItsPath for whether it correctly returns false for commands that do not add up to the origin.

assert(crossedOriginOnItsPath("1n2s"));

assert(crossedOriginOnItsPath("12w1e"));

assert(crossedOriginOnItsPath("1nw16se"));

assert(crossedOriginOnItsPath("771ne198sw"));

assert(crossedOriginOnItsPath("-1w-10e"));

assert(crossedOriginOnItsPath("-145n-1s"));

assert(crossedOriginOnItsPath("-13nw-111se1n120s"));

assert(crossedOriginOnItsPath("999nw998se-999n-999s"));

The compiler disagreed with all of these assertions, which is correct because all of them do not add up to the origin.