CSC230 Project 1

This is your first, easy project. The others will be larger and more interesting. This one will let you get you started writing, formatting, compiling, and executing simple C programs. This requires that you familiarize yourself with the Common Platform and our Coding Style Guidelines. A full list of the relevant Learning Outcomes is included at the end of this page.

This project is to be done individually.

Getting Started

See the note about the missing expected output files at the end of this section.

There is a starter file for this project, starter1.tgz. This file is a compressed tar archive containing several files you'll need. You'll need to unpack the files inside to get started. There are a few good ways to do this:

- The hard way:
  1. First, download the archive using the link above, and then use an sftp client or a tool like ExpanDrive to copy the archive to a location in your AFS space where you want to work on this assignment.
  2. Log in on a common platform machine, change directories to the path where you plan to work and then unpack the archive using a command like the following:
     ```
     $ tar xzvf starter1.tgz
     ```

- Or, the easy way:

  1. If you are logged in on one of the common platform systems, you can just unpack the starter directly from our official copy in AFS, without even making a copy of it. Change directory to a location where you want to work on the assignment, and enter the following command. This will unpack the archive, putting all the files right there in your working directory.
     ```
     $ tar xzvf /afs/eos.ncsu.edu/courses/csc/csc230/common/www/proj/p1/starter1.tgz
     ```

In general, when we grade your programs, we expect them to behave exactly as described in the assignment. This includes producing the right output with the right spacing and line termination. The project 1 starter includes some input files and expected output files to help you make sure your programs are behaving correctly. As described below, you can capture your program's output in a file and then use the `diff` command to compare the output you got with the what you were expected to get. If `diff` sees any differences, even in spacing or line termination, it will tell you where they are.

When the project was first posted, the starter didn't include the expected output files for the `textbox.c` program. Those have been added to the starter now. There's also a new file update1.tgz that just includes the previously missing files. If you just need these expected output files, you probably want to download the update, rather than a new copy of the starter, less risk of overwriting your work.

Part 0 : Registering with NCSU GitHub (a very

easy 10 pts)

Using a web browser, visit github.ncsu.edu and log in with your unity credentials. If you're not able to log in on this website, inform the instructor immediately (i.e., **before** the deadline).

We'll be using NCSU's GitHub for submitting future assignments. This is an opt-in system, which means that the initial login is required so that you are recognized as a user. If you are not in the system, we won't be able to create the repositories that you will be using for the rest of the course, and you won't be able to complete future assignments.

Maybe you've used git and github before. That's great, but keep in mind that you'll need to use your assigned repo, hosted on NC State's GitHub for future projects.

Grading

You'll get your 10 points just for logging in on the NCSU github. This is a really easy 10 points to earn. There is nothing to turn in for this part of the assignment. If you run into trouble logging in, be sure to let us know before the deadline, so we can try to help you out.

Part 1 : Correcting Style (25 pts)

The program, style.c is badly formatted. It has everything, bad or missing comments, bad curly bracket placement, some hard tabs for indentation, some bad line termination, no line termination at the end of the last line, maybe more problems. Edit this program to make it consistent with the class style guidelines.

If you'd like, you can use an IDE or some other tool to help you format the source code. If you have trouble removing the windows-style line termination, you might want to try the `dos2unix` command installed on the common platform systems. For invisible parts of the source file (like spaces or line termination), you might want to out the `hexdump` program. If you run it like the following, it will show the sequence of characters in the source file, in hexadecimal on the left and as symbols on the right where possible:

```
$ hexdump -C style.c
```

When your program runs (even before you correct the style), it should exit with a successful exit status and produce output like the following. The starter also includes a expected-s.txt file you can use to make sure your style-corrected program produces exactly the output we're expecting (a randomly-generated paragraph of text).

```
wlrb mqbhcd r owkk hiddqsc xrjm wfr sjyb dbef arcbynecd ggx pklore
nmpa qfwkho kmcoqhnw kuewhsqmgb uqcljj vsw dkqtbxi mvtrrbljpt snfwzqfj
fadrrwsof b nuvqhff saqxwpqcac hchzv rkmlno jkpqpx jxkitzyx cbhhkic
oendtomfg wdwf gpxiq kuytdlcgde htaciohor tq vwcsgspqo msb agu nny
nz gd wpbtr blnsade guumoqc rubetoky hoachwdvmx rdryxl n qtukwa mleju
wci xubume meya drmydiajxl ghiqfmz lvihjo vsuyoyp yul eim otehzri c
kpggkbb p zrzu xamludf kgruowz i oobpple lwphapjna qhdcnvwdtx bmyppp
uxnspusgd iixqmbfjxj v djsuyib ebmws q oygyxym evypzvje ebeocfu
sxdixtigsi ehkch dflilrjq nxzt rsvbspkyh enbppkqtp dbuotbbqcw vrf ju jd
```

```
tg iqvdg ijvwcya bwewpjvyg hljxepb iwuqzdzu du zv fspqp wuz f ovydd
Words: 103
```

After reformatting the program, add the required block comments at the start of the source file and before each function. You may need to read the code for each function so your comments can summarize what they each do and so you can choose meaningful named constants, instead of magic numbers.

For this part of the assignment, you'll be submitting your modified source file, `style.c`, to the Project 1 submission locker on WolfWare classic. Your submission will be graded according to the following:

- Correcting source code formatting: 10 points
- Adding the required block comments: 10 points
- Program still compiles cleanly and produces the right output: 5 points

Part 2 : Text Box (25 pts)

Have a look at the textbox.c source file. This program is intended to give you a chance to try reading input character-by-character using getchar() and printing characters using putchar(). The overall skeleton of this program is done for you, but you'll need to fill in the bodies of the functions to get it working.

The job of this program is to print text from standard input, but with a border around the text. The figure below shows how it's supposed to work.

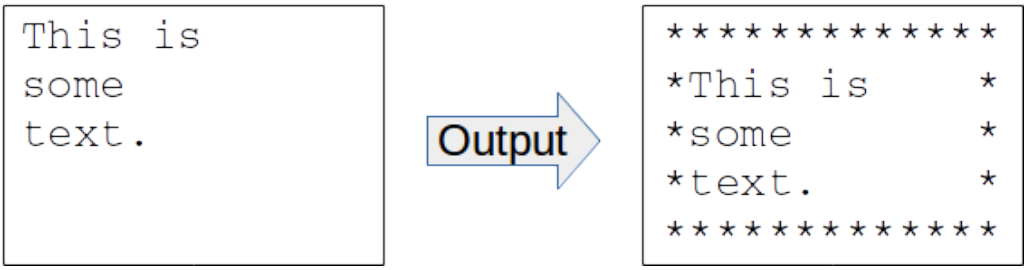

Figure: job of the textbox.c program, wrapping a border around input text.

The border drawn around the text is a fixed-width, and made from a chosen character. It should be tall enough to contain every line read from standard input. Both the box width and the character it's made of are specified via preprocessor constants, so it would be easy to change them and then re-compile the program. We'll use a border that's made of asterisks and contains lines of 60 characters (so, a lot wider than the figure above shows). This width value is the length of each line of text *inside* the border. With a border character at the start and end of each output line, these will actually be two characters longer. So, by default, output lines will all be 62 characters, with 60 characters in between he first and last character of the border.

Input lines may not all be exactly the right length (probably, they won't be). So, as you're printing, you'll have to pad with spaces on the right for lines that are too short. For lines that are too long, you'll discard extra characters on the right, printing as many as you can and reading (and ignoring) everything else up to the end of the line.

Design and Implementation

Your program will define and use the following three functions. You can have more if you want, but we'll be looking for at least these three.

- `void lineOfChars( char ch, int count )`
  This function prints out multiple copies of the given character, followed by a newline. The number of copies is determined by the count parameter. Use this function to help print the top and bottom border around the text.

- `bool paddedLine()`
  This function will read and print a single line of text inside the border. If there's no line of text to print (i.e., it hits EOF before it can read any characters), it will return false to tell the caller that there's no more text to put in the box. This function will read the text from standard input and print to standard output. If the line of text isn't long enough, it will add extra spaces at the end to make the box rectangular. If a line of text is too long, it will discard extra characters on the line (i.e., up to the end-of-line) to keep the box rectangular.

  This function is probably the most fun/tricky part of this assignment. The test cases provided with the starter will help you try out your function's behavior and make sure it's doing the right thing in all situations. When printing out a line in the middle part of the text box, you can either make main() responsible for printing the border characters at the start and the end, or you can make this function handle it, whichever seems easier to you.

- `int main()`
  This is, of course, the starting point of your program. It will use the other two functions to print the text from standard input with a border around it.

Be sure to fill in the empty block comments in this program according to the style guide. Remember, a file comment has a short summary of the file/program, along with a file and an author tag. A block comment on a function needs a short summary of what the function does, along with param tags for any parameters the function takes, and return tag if it's not a void function.

Testing

Compile your program using the following command.

```
$ gcc -Wall -std=c99 textbox.c -o textbox
```

You can run the program and type input to it yourself, but the output will look a lot better if you use redirection to read input from a file. If you just want to see what your program's output looks like, you can run it like:

```
$ ./textbox < input-t1.txt
```

If you really want to make sure your output is right, you can capture the output in file and compare it against one of our expected output examples. Try running your program as follows to tell it to read input from our first test case (input-t1.txt) and send output to a file named `output.txt`. Then, check your programs exit status to make sure it reported successful execution, and use diff to help make sure it produced output that matches what we were expecting.

```
$ ./textbox < input-t1.txt >| output.txt
$ echo $?
0
```

```
$ diff output.txt expected-t1.txt
```

You can use similar commands to test your program against each of the four test cases we're providing. Here's what they each do:

1. This test input contains five lines, each one 60 characters long, exactly long enough to fill a line of the box. It should be easy.
2. This test has lines that are all shorter than the box width, so they'll all have to be padded with extra spaces at the end to make the box the right width.
3. This test has some lines that are too long, so the extra characters at the end will have to be discarded as the line is processed.
4. This test contains a few lines of text, some that have to be padded with spaces and some (one) that has to be truncated.

Remember, you need check the exit status right after you run your program. If you get an exit status of zero and diff doesn't report any differences between the expected output and the actual output you got, then it looks like your program is behaving correctly. If diff notices any differences, it will report the line number where there's a discrepancy, along with the contents of the lines that don't match. If you have trouble interpreting the output of diff, you can try out the `sdiff` command. This works like diff, but it will display the expected and actual output files side by side, and may make it easier to see inconsistencies between the files. To use sdiff, just replace `diff` with `sdiff` in the shell code above.

Grading

For this part of the assignment, you'll be submitting your modified source file, `textbox.c` to the Project 1 submission locker on WolfWare classic. Your edits to `textbox.c` will be graded according to the following:

- Compiling cleanly: **5 points**
- Producing the right output during execution: **15 points**
- Follwing the style guidelines: **5 points**
  This includes commenting, consistent indentation (you can change the indentation from the starter code, if you want), curly bracket placement, etc.
- Following the design **Up to 20 percent deduction**
  This includes implementing and using the functions described above and preprocessor constants, no global variables.
- Late submission **20 percent deduction**

Part 3 : Ballistics Table Generator (25 pts)

One of the first jobs we came up for computers was calculating ballistics tables, charts that tell you where a projectile could be expected to land if you launch it under particular circumstances. We've come up with lots of varied (and peaceful) things to do with computers since then, but it can still be interesting to think about these problems from the early days of computing. That's what you're going to do for this program.

The following figure shows an ideal path of a projectile. On a flat surface, if there's no air resistance, it follows a parabolic path that depends on how fast it's going when it leaves the point of origin, v_0, and at what angle, a.

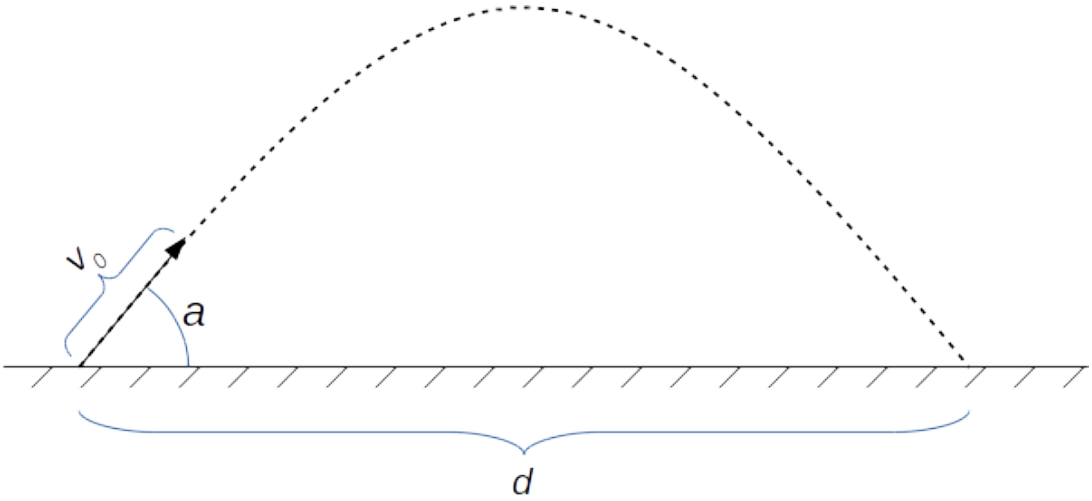

Figure: ideal, parabolic path of a projectile.

Specifically, the height of an idealized projectile at *t* seconds after launch will be determined by the following formula, where v_0 is its initial speed in *m/s*, *a* is the angle at which it begins traveling (measured from the ground), and *g* is gravitational acceleration (use $9.81\ m/s^2$).

$v_0\ t\ sin(\ a\ )\ -\ g\ t^2\ /\ 2.0$

Horizontally, the projectile travels until it hits the ground. So, after *t* seconds of flight, the following formula tells the distance, *d*, it has traveled from its point of origin.

$v_0\ t\ cos(\ a\ )$

Your `ballistics.c` program will prompt the user for a double value representing the initial projectile velocity. Use the following prompt (there's a space after the colon).

```
V0:
```

After the user types in a value and presses enter, your program should print a blank line. This extra line is to make the output look OK when it's sent to a file. Then, print a table like the following (this one's for an initial velocity of 10):

```
     angle |          v0 |        time |    distance
-----------+-------------+-------------+------------
         0 |     10.000 |       0.000 |       0.000
         5 |     10.000 |       0.178 |       1.770
        10 |     10.000 |       0.354 |       3.486
        15 |     10.000 |       0.528 |       5.097
        20 |     10.000 |       0.697 |       6.552
        25 |     10.000 |       0.862 |       7.809
        30 |     10.000 |       1.019 |       8.828
        35 |     10.000 |       1.169 |       9.579
        40 |     10.000 |       1.310 |      10.039
        45 |     10.000 |       1.442 |      10.194
        50 |     10.000 |       1.562 |      10.039
        55 |     10.000 |       1.670 |       9.579
        60 |     10.000 |       1.766 |       8.828
        65 |     10.000 |       1.848 |       7.809
```

```
 70 |     10.000 |      1.916 |      6.552
 75 |     10.000 |      1.969 |      5.097
 80 |     10.000 |      2.008 |      3.486
 85 |     10.000 |      2.031 |      1.770
 90 |     10.000 |      2.039 |      0.000
```

This table has four columns, with a header at the top. Columns are separated by a vertical bar, with a space on either side of every bar. Values in the table are printed in a 10-character field, with the value rounded to three fractional digits for real-valued fields. The first column samples different values for the angle, *a*, every five degrees from 0 to 90 degrees. The second column gives the initial velocity, entered by the users. The third column reports the number of seconds the projectile is in the air, and the last column reports the distance, *d* it travels from its point of origin.

How are you going to generate this table? Well, the first two columns are easy. The initial velocity is given by the user, and the angle is sampled every five degrees. For the flight time, you just have to figure out when the height of the projectile gets back down to zero; that's when it hits the ground. Look at the formula for the projectile height given above, and solve for the value of *t* that yields a height of zero. This gives you a quadratic equation, so solving it would be a good a good application for the quadratic formula. In fact, this quadratic equation is so simple, there are definitely some opportunities to simplify your solution.

You can assume the initial velocity given by the user is a non-negative real number. You don't have to error-check this input. There will be plenty of opportunities for that kind of thing on future projects.

Design and Implementation

The starter contains a partial implementation for the `ballistics.c` program. You get to fill in the functions and create meaningful named constants to make the program easy to understand and modify. You'll define and use the following three functions. You can have more if you want.

- `double flightTime( int angle, double v0 )`
  Given the angle in degrees that a projectile leaves the ground and its initial velocity, this function should return the time the projectile is in the air.

- `void tableRow( int angle, double v0, double t )`
  This function prints out a row of the table, given the angle and initial velocity (like the previous function) and the flight time (the output from the previous function), it should print a row of the table reporting these three values, along with the distance the projectile travels.

- `int main()`
  This is the starting point for running your program. It will read input from the user, print the table header, then use the other two functions to print the rest of the table.

Be sure to fill in the block comments for the source file and the functions you get to write. For preprocessor constants you define, be sure to give a short Javadoc-style comment that says what they're for.

Math

For this problem, we'll need to use some parts of the math library, for calculating the sine and cosine of the launch angle, *a* and for the square root calculation, if you need it. The functions you need are declared in the `math.h` header; the partial implementation from the starter already includes it for you.

You'll also need to link with the math library. The compiler option for this is given below.

The C functions for sine, cosine and square root are a lot like the static methods in the Java Math class. Here's how they work:

- `double sin( double a )`
  This function returns the sine of the angle, a, given in radians.
- `double cos( double a )`
  This function returns the cosine of the angle, a, given in radians.
- `double sqrt( double v )`
  This function returns the (non-negative) square root of the given (non-negative) value, v.

Notice that the sin() and cos() functions expect an angle in radians, but, elsewhere, angles are described in degrees. So, you'll have to convert. For this, you'll probably want to use the preprocessor constant `M_PI` (the mathematical constant PI). This value is commonly defined in the `math.h` header, but it's not actually part of the C99 standard. There's an option for defining it anyway, given below in the compile instructions.

Testing

This program may use the `M_PI` constant and it will probably need functions from the math library. You can use the `-D_GNU_SOURCE` to enable the constant for PI, and the `-lm` flag will tell the compiler to link with this library:

```
$ gcc -D_GNU_SOURCE -Wall -std=c99 ballistics.c -o ballistics -lm
```

This program is easy to run with input entered by the user. While you're developing and debugging, you may just want to type the input value yourself. Once you think it's running correctly, you can start comparing the output against expected output for our provided test cases. You can use commands like the following:

```
$ ./ballistics < input-b1.txt >| output.txt
$ echo $?
0
$ diff output.txt expected-b1.txt
```

We're providing four test cases to try your program with, each with an input file and an expected output file. They provide different values for the initial velocity, including one with an initial velocity of zero.

Grading

For this part of the assignment, you'll be submitting your modified source file, `ballistics.c` to the Project 1 submission locker on WolfWare classic. We'll grade it using the same criteria as the `textbox.c` program:

- Compiling cleanly: **5 points**
- Producing the right output during execution: **15 points**
- Follwing the style guidelines: **5 points**
  commenting, consistent indentation, curly bracket placement, magic numbers, etc.
- Following the design **Up to 20 percent deduction**

Implementing and using the exptected functions, no global variables.
- Late submission **20 percent deduction**

Notes on Completing the Assignment

Of course, before you submit, you'll want to be sure both of your programs are consistent with the CSC 230 Style Guidelines. Also, make sure they compile cleanly (no errors or warnings) on a common platform machine using the required compiler options.

There is a 24 hour window for late submissions. After the main locker closes, submit all late work to `Project_1_Late`, but remember there is an automatic *20 percent* deduction for late work.

Common Problems

- While developing, if a program enters an infinite loop, use `Ctrl+C` to stop it.
- Make sure you submit source files (ending in .c), not your executable programs.
- Make sure you've submitted all of your files using WolfWare classic. After submitting, you can go back and check to make sure your files are all there. Emailing your program isn't really a submission.
- Contact the teaching staff if you run into a problem submitting your work. Or, visit Piazza and see if someone else has run into the same kind of problem.

Learning Outcomes

The syllabus lists a number of learning outcomes for this course. This assignment is intended to support several of theses:

- Write small to medium C programs having several separately-compiled modules
- Explain what happens to a program during preprocessing, lexical analysis, parsing, code generation, code optimization, linking, and execution, and identify errors that occur during each phase. In particular, they will be able to describe the differences in this process between C and Java.
- Correctly identify error messages and warnings from the preprocessor, compiler, and linker, and avoid them.
- Find and eliminate runtime errors using a combination of logic, language understanding, trace printout, and gdb or a similar command-line debugger.
- Interpret and explain data types, conversions between data types, and the possibility of overflow and underflow
- Explain, *inspect*, and implement *programs using structures such as enumerated types, unions, and constants and arithmetic, logical, relational, assignment, and bitwise operators.*
- Trace and reason about variables and their scope in a single function, across multiple functions,

and across multiple modules.

- Write, debug, and modify programs using library utilities, including, but not limited to assert, the math library, the string library, random number generation, variable number of parameters, standard I/O, and file I/O
- Use simple command-line tools to design, document, debug, and maintain their programs.
- Distinguish key elements of the syntax (what's legal), semantics (what does it do), and pragmatics (how is it used) of a programming language.