CSC 230 Project 2

Sudoku

Sudoku is a popular number-placement puzzle, whose objective is to fill a 9 X 9 grid with digits such that each row of the grid, each column of the grid, and each of the nine 3 X 3 subgrids (the dark boxes in the figure below) contain the digits from 1 to 9. You'll find a copy of the Sudoku puzzle below along with an explanation of the rules and some variants of the game at the Wikipedia page: https://en.wikipedia.org/wiki/Sudoku

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Sudoku Puzzle

For this project, you're going to implement a program, `sudoku`, that tells whether a completed Sudoku puzzle is valid, i.e., it meets the requirements given above, or is invalid because it does not meet the requirements. The program will also solve a partially completed Sudoku puzzle or determine that it is invalid due to not meeting the requirements or because it cannot be solved. In addition to 9 X 9 puzzles, the program will work with puzzles whose size is a perfect square ranging from 4 to 16.

The user can run the program like the the following. Here, the user is entering the size of the puzzle, then the digits in each row of the puzzle. A blank space in the puzzle that has not yet been filled in with a number is indicated with the digit 0. Here is a valid completed 4 X 4 puzzle. This is the same as test input, `input-01.txt`.

```
$ ./sudoku
4
1 2 3 4
3 4 1 2
2 1 4 3
4 3 2 1
Valid
```

Here the program outputs the solution for a partially completed 4 X 4 puzzle. This is the same as test input, `input-02.txt`.

```
$ ./sudoku
4
1 2 3 4
3 0 1 2
2 1 4 3
0 3 2 0
  1  2  3  4
  3  4  1  2
  2  1  4  3
  4  3  2  1
```

Here is an invalid completed 4 X 4 puzzle. This is the same as test input, `input-03.txt`.

```
$ ./sudoku
4
1 2 3 4
3 4 1 2
2 1 2 3
4 3 2 1
Invalid
```

To help get you started, we're providing you with a test script, and several test input files and expected output files. See the Getting Started section for instructions on how to set up your development environment so that you can be sure to submit everything needed when you're done.

This project supports a number of our course objectives. See the Learning Outcomes section for a list.

Rules for Project 2

You get to complete this project individually. If you're unsure of what's permitted, have a look at the academic integrity guidelines in the course syllabus.

In the design section, you'll see some instructions for how your implementation is expected to work. Be sure you follow these rules. It's not enough to just turn in a working program; your program has to follow the design constraints we've asked you to follow. This helps to make sure you get some practice with the parts of the language we want to make sure you've seen.

Requirements

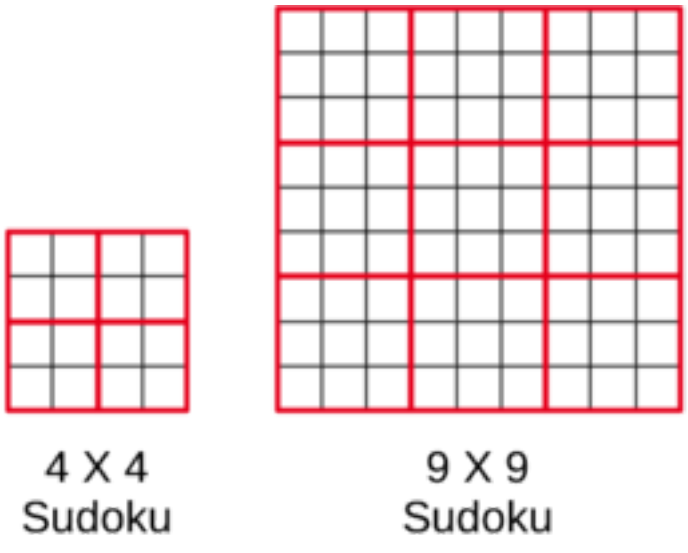
Requirements are a way of describing what a program is supposed to be able to do. In software development, writing down and discussing requirements is a way for developers and a customers to agree on the details of a system's capabilities, often before coding has even begun. Here, we're trying to demonstrate good software development practice by writing down requirements for our program, before we start talking about how we're going to implement it.

Sudoku Solution Validity

To be valid, the solution to an n X n Sudoku puzzle must have a positive number between

1 and n in every cell of the grid. There can't be more than one copy of the same number in any row of the grid and there can't be more than one copy of the same number in any column.

A Sudoku puzzle also has n subgrids. These are shown as red boxes in the example 4 X 4 and 9 X 9 grids below. If we let m be the square root of the puzzle size, n, then each of these subgrids is a m X m block of cells. The entire puzzle is covered by m rows of subgrids, each containing m subgrids (so, the entire puzzle is covered by an m X m grid of subgrids, each subgrid containing an m X m block of cells). The subgrids give one more constraint on a solution. To be valid, a solution can't contain more than one copy of the same number in any of the subgrids.



Example of subgrids in 4 X 4 and 9 X 9 puzzles

Program Input

The `sudoku` program reads input from standard input. The first value is the size, n, of the Sudoku puzzle, i.e., n is the number of rows and columns. This is followed by the values in each row, with the digit 0 representing a blank space in the puzzle. The values are separated by whitespace, but the size and the values in each row do not have to be on separate lines. This makes the input easier to read with scanf(). For example, the input below, which is the same as `input-09.txt` and represents the same completed puzzle as `input-01.txt`, is valid.

```
$ ./sudoku
4 1 2 3 4 3 4 1 2
2 1 4
3
4 3 2 1
Valid
```

Also, your program may ignore/disregard any input after reading the size and the first n X n values.

Invalid Input

If the size of the Sudoku puzzle isn't valid because it is not a perfect square between 4 and 16, the program should print (to stdout) a line with the following error message and

terminate with an exit status of 1.

```
Invalid size
```

If a value in the puzzle is not an integer, or if it is less than 0 or greater than the size of the puzzle, or if there are too few values, the program should print (to stdout) a line with the following error message and terminate with an exit status of 1.

```
Invalid input
```

Program Output

If a Sudoku puzzle does not contain any blank spaces (zeros), and it meets the validity requirement described in the Sudoku Solution Validity section, the program should print a line with the following message and exit successfully.

```
Valid
```

If a Sudoku puzzle does not contain any blank spaces (zeros), and it does not meet the Sudoku requirements described in the Solution Validity section, the program should print a line with the following message and exit successfully.

```
Invalid
```

If a Sudoku puzzle contains one or more blank spaces (zeros), and it is not possible to replace these blank spaces with numbers so that the puzzle satisfies the solution requirements (in the Solution Validity section), the program should print a line with the following message and exit successfully.

```
Invalid
```

If a Sudoku puzzle contains one or more blank spaces (zeros) and can be solved (i.e., the blanks can be replaced with numbers so that the requirements described under the Solution Validity are satisfied), the program should print out a solution and then exit successfully. To report the solution, print each row on a separate line and use a field width of 3 for each value in a row. For example, here is the solution that would be output for `input-02.txt`.

```
  1   2   3   4
  3   4   1   2
  2   1   4   3
  4   3   2   1
```

Design

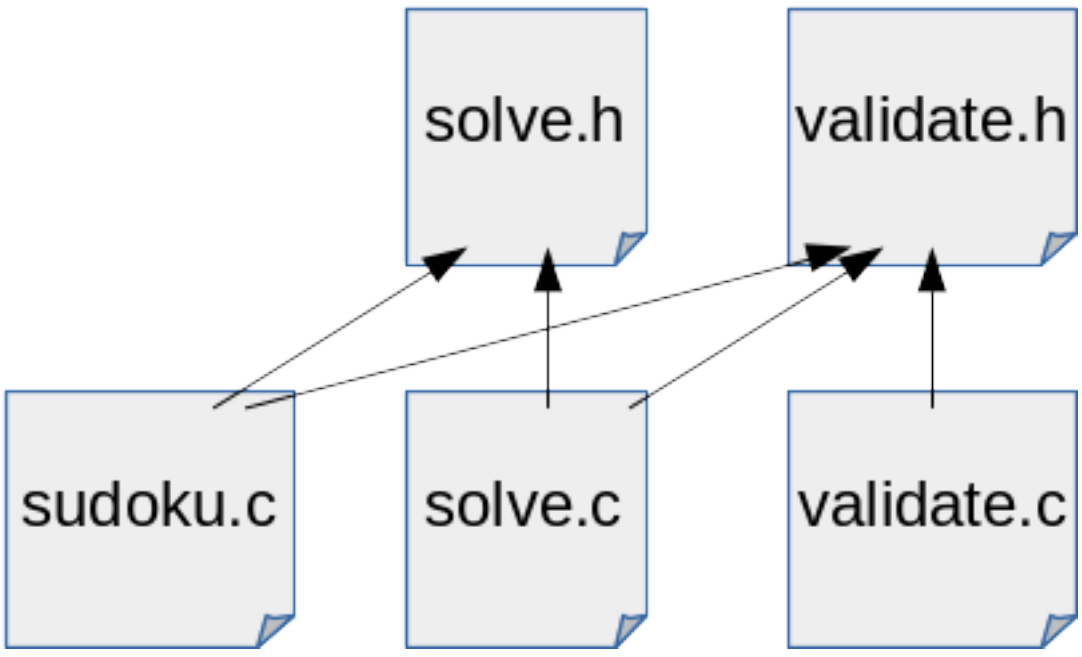
The program will be implemented in three components.

- validate.h / validate.c
  This component will be responsible for validating the rows, columns, and squares of a puzzle to determine if the Sudoku rules have been violated.

- solve.h / solve.c
  This component will solve a Sudoku puzzle or determine that it cannot be solved. It will make use of the validate component.

- sudoku.c
  This is the main component. It will contain the starting point for the program (main) will be responsible for reading in the puzzle size and the values in the puzzle, which will be stored in a 2D integer array. It will use the validate component to determine if the completed puzzle is valid/invalid. It will use the validate component to determine if an incomplete puzzle is already in violation of the rules. If not, it will use the solve component to solve the puzzle or determine that it cannot be solved.

The following figure describes the dependency structure between these components. The sudoku component uses the other two components. The solve component can use functions from validate, but it can't use functions from the main sudoku component. The validate component doesn't depend on either of the other components, so it can't call functions in sudoku.c or in solve.c.



Dependency structure for the program

Required Functions

You can use as many functions as you want to solve this problem, but you'll need to implement and use at least the following ones:

- `bool validateRows(int size, int grid[size][size])`
  This function is part of the validate component. It returns true if every row in the grid contains exactly one digit in the range 1 to size, false, otherwise

- `bool validateCols(int size, int grid[size][size])`
  This function is part of the validate component. It returns true if every column in the grid contains exactly one digit in the range 1 to size, false, otherwise.

- `bool validateSquares(int n, int size, int grid[size][size]))`
  This function is part of the validate component. It returns true if every n X n square (subgrid) in the grid contains exactly one digit in the range 1 to size.

- `bool validateRowsWithSpaces(int size, int grid[size][size])`
  This function is part of the validate component. It returns true if every row in the grid contains no more than one digit in the range 1 to size, false, otherwise

- `bool validateColsWithSpaces(int size, int grid[size][size])`
  This function is part of the validate component. It returns true if every column in the grid contains no more than one digit in the range 1 to size, false, otherwise.

- `bool validateSquaresWithSpaces(int n, int size, int grid[size][size]))`
  This function is part of the validate component. It returns true if every n X n square (subgrid) in the grid contains no more than one digit in the range 1 to size.

- `bool solve(int n, int size, int grid[size][size])`
  This function is part of the solve component. It attempts to solve the Sudoku puzzle represented by the grid. It returns true if it is successful in solving the puzzle, false, otherwise. It must use the algorithm for solving the puzzle as described below. If the puzzle is successfully solved, grid will contain the solved Sudoku puzzle when the function returns. Just as in the `validateSquares()` and `validateSquaresWithSpaces()` functions, the parameter, `n`, is the square root of the parameter, `size`, which is the number of rows and columns in the Sudoku puzzle.

NOTE: The `validate.h` file contained in the original starter2.tgz file contained the following prototypes:

```
bool validateNoSpaces(int n, int size, int grid[size][size]);
bool validateSpaces(int n, int size, int grid[size][size]);
```

They have been removed from the `validate.h` file contained in the updated starter2.tgz file. You may implement and use these functions, if you like, but you are not required to do so.

Sudoku Algorithm

Because some Sudoku puzzles may have more that one solution, you must use the "Simple Solving Algorithm" found at http://pi.math.cornell.edu/~mec/Summer2009/meerkamp/Site/Solving_any_Sudoku_I.html to ensure that your solutions matches ours. Here is a copy of the algorithm:

1. Make a list of all the all blank spaces in the puzzle, in typewriter order (left to right, top to bottom)

2. Initially, our *current cell* will be the first cell in the list of blank cells.

3. Enter a 1 into the current cell. If this violates the solution validity, try entering a 2, then a 3, and so forth, until
   a. the entry does not violate the Sudoku condition, or until
   b. you have reached n (the size of the puzzle) and still violate the solution validity.

4. In case a: if the current cell was the last blank on our list, then the puzzle is solved. If

not, then go back to step 3 with the current cell being the next cell in the list of blank spaces.
In case b: if the current cell is the first cell in the enumeration, then the Sudoku puzzle does not have a solution. If not, then erase the n from the current cell, move the current cell to the previous cell in the list of blanks, and continue with step 3.

Globals and Magic Numbers

You should be able to complete this project without creating any global variables. The function parameters give each function everything it needs.

Be sure to avoid magic numbers in your source code. Use the preprocessor to give a meaningful name to all the important, non-obvious values you need to use.

Build Automation

You get to implement your own Makefile for this project (called `Makefile` with a capital 'M', no filename extension). Its default target should build your program, compiling each source file to an object file and then linking the objects together into an executable. It should correctly describe the project's dependencies, so targets can be rebuilt selectively, based on what parts of the project have changed. The makefile should compile each source file with the -Wall, -std=c99, and -g options. This will be useful when you try to use `gdb` to help debug the program.

The Makefile should also have a `clean` rule, to let the user easily delete any temporary files or target files that can be rebuilt the next time make is run (e.g., the object files, the executable and any temporary output files).

A clean rule can look like the following. It doesn't have any prerequisites, so it's only run when it's explicitly specified on the command-line as a target. It doesn't actually build anything; it just runs a sequence of shell commands to clean up the project workspace. Your clean rule will look like the following (we used the <tab> notation to remind you where the hard tabs need to go). In your clean rule, replace things like all-your-object-files with a list of the object files that get built as part of your project.

```
clean:
<tab>rm -f all-your-object-files
<tab>rm -f your-executable-program
<tab>rm -f any-temporary-output-files
<tab>rm -f anything-else-that-doesn't-need-to-go-in-your-repo
```

To use your `clean` target, type the following at the command line, ***but first be sure you have committed all of the files that you need to your github repo***, as the -f flag forces the removal without asking you if it's OK.

```
$ make clean
```

Testing

The starter includes a test script, along with test input files and expected outputs for the program. When we grade your program, we'll test it with this script, along with a few other test inputs we're not giving you.

To run the automated test script, you should be able to enter the following:

```
$ chmod +x test.sh # probably just need to do this once
$ ./test.sh
```

This will automatically build program using your Makefile and see how it behaves on all the provided test inputs.

You probably won't pass all the tests the first time. In fact, until you have a working makefile, you won't be able to use the test script at all.

If you want to compile your program by hand, the following command should do the job.

```
$ gcc -Wall -std=c99 -g sudoku.c validate.c solve.c -o sudoku
```

To run your program, you can do something like the following. The test script prints out how it's running your program for each test case, so this should make it easy to check on individual test cases you're having trouble with. The following commands run the program with input read from the `input-01.txt` test case and with output written to the file, `output.txt`. Then, we check the exit status to make sure the program exited successfully (it should for this particular test case). Finally, we use diff to make sure the output we got looks exactly like the expected output.

```
$ ./sudoku < input-01.txt > output.txt
$ echo $?
0
$ diff output.txt expected-01.txt
```

If your program generated the correct output, diff shouldn't report anything. If your output isn't exactly right, diff will tell you where it sees differences. Keep in mind that diff may complain about differences in the output that you can't see, like differences in spacing or a space at the end of a line.

Sample Input Files

With the starter, we're providing a number of test input and expected output files to help you develop your program and test it to make sure it's working correctly. Here's what they each do. This can help you think about and examine what's going wrong if your program's behavior isn't exactly what it's supposed to be.

1. A completed 4 X 4 puzzle that is Valid.
2. A partially completed 4 X 4 puzzle that can be solved.
3. A completed 4 X 4 puzzle that is Invalid due to a bad row, column, and square.
4. A completed 9 X 9 puzzle that is Valid.

5. A completed 4 X 4 puzzle that is Invalid due to a bad square.
6. A partially completed 4 X 4 puzzle that is Invalid due to a bad row, column, and square.
7. A partially completed 4 X 4 puzzle that is Invalid because even though there are no conflicts, it cannot be solved
8. A partially completed 9 X 9 puzzle that can be solved.
9. A completed 4 X 4 puzzle that is Valid, but with size and row values that are not on different lines.
10. A completed 4 X 4 puzzle that is Valid, but has input values after the 16 puzzle digits that can be ignored.
11. A completed 16 X 16 puzzle that is Valid.
12. A partially completed 16 X 16 puzzle that can be solved.
13. A completed 16 X 16 puzzle that is Invalid
14. An invalid input, with a size of 5.
15. An invalid input, with a non-integer value.
16. An invalid input, with value that is greater than the size.

Grading

The grade for your program will depend mostly on how well it functions. We'll also expect your code to compile cleanly, we'll expect it to to follow the style guide and the expected design, and we'll expect a working makefile.

- Working Makefile, including dependencies and a make clean rule: **8 points**
- Compiling cleanly on the common platform: **10 points**
- Correct behavior on all test cases: **70 points**
- Source code follows the style guide: **20 points**
- Deductions
  - Up to **-50 percent** for not following the required design.
  - Up to **-30 percent** for failing to submit required files, submitting files with the wrong name or having extraneous files in the repo.
  - Up to **-20 percent** penalty for late submission.

Getting Started

To get started on this project, you'll need to clone your NCSU github repo and unpack the given starter into the p2 directory of your repo. You'll submit by checking files into your repo and pushing the changes back up to the NCSU github.

Cloning your Repository

Everyone in CSC 230 has been assigned their own NCSU GitHub repository to be used during the semester. It already has subdirectories (mostly empty) for working on each of the remaining projects. How do do you figure out what repo you've been assigned? Use a web browser to visit `github.ncsu.edu`. After authenticating, you should see a drop-down menu over on the left, probably with your unity ID on it. Select "engr-csc230-fall2019" from this drop-down and you should see a repo named something like "engr-csc230-fall2019/unity-id" in the box labeled Repositories over on the left. This is your repo for submitting projects.

I've had some students in the past who do not see the organization name, "engr-csc230-fall2019" in their drop-down menu. I'm not sure why that happened, but when they chose "Manage organizations" near the bottom of this drop-down, it took them to a list of all their repos, and that list did include "engr-csc230-fall2019". Clicking on the organization name from there took them to a page that listed their repo. If you're having this problem, try the "Manage organizations" link.

You will need to start by cloning this repository to somewhere in your local AFS file space using the following command (where unity-id is your unity ID, just like it appears in your repo name):

```
$ git clone https://github.ncsu.edu/engr-csc230-fall2019/unity-id.git
```

This will create a directory with your repo's name. If you `cd` into the directory, you should see directories for each of the projects for the class. You'll want to do your development for this assignment right under the `p2` directory. That's where we'll expect to find your project files when we're grading.

Unpack the starter into your cloned repo

You will need to copy and unpack the project 2 starter. We're providing this file as a compressed tar archive, starter2.tgz. You can get a copy of the starter by using the link in this document, or you can use the following curl command to download it from your shell prompt.

```
$ curl -O https://www.csc2.ncsu.edu/courses/csc230/proj/p2/starter2.tgz
```

Temporarily, put your copy of the starter in the p2 directory of your cloned repo. Then, you should be able to unpack it with the following command:

```
$ tar xzvpf starter2.tgz
```

Once you start working on the project, be sure you don't accidentally commit the starter to your repo (that would be an example of an extraneous file that doesn't need to be there). After you've successfully unpacked it, you may want to delete the starter from your p2 directory, or move it out of your repo.

```
$ rm starter2.tgz
```

Instructions for Submission

If you've set up your repository properly, pushing your changes to your assigned CSC230 repository should be all that's required for submission. When you're done, we're expecting your repo to contain the following files. You can use the web interface on github.ncsu.edu to confirm that the right versions of all your files made it.

- `sudoku.c` : Implementation file created by you.
- `validate.h` : Header file for the validate component completed by you.
- `validate.c` : Implementation file for the validate component created by you.
- `solve.h` : Header file for the solve component completed by you.
- `solve.c` : Implementation file for the solve component created by you.
- `Makefile` : Makefile for efficiently building your program, created by you.
- `input-*.txt` : Test input files, provided with the starter.
- `expected-*.txt` : Expected output files, provided with the starter.
- `test.sh` : test script, provided with the starter.
- `.gitignore` : a file for this project, telling git some files to *not* commit to the repo.

Pushing your Changes

To submit your solution, you'll need to first commit your changes to your local, cloned copy of your repository. First, you need to add any new files you've created to the index. Running the following command will stage the current versions of a file in the index. Just replace the `some-file-name` with the name of the new file you want commit. You only need to do this once for each new file. The `-am` option used with the `commit` below will tell git to automatically commit modified files that are already being tracked.

```
$ git add some-file-name
```

When you're adding new files to your repo, you can use the shell wildcard character to match multiple, similar filenames. For example, the following will add all the input files to your repo.

```
$ git add input-*.txt
```

When you start your project, don't forget to add the `.gitignore` file to your repo. Since its name starts with a period, it's considered a hidden file. Commands like `ls` won't show this file automatically, so it might be easy to forget.

```
$ git add .gitignore
```

Before you commit, you may want to run the git status command. This will report on any files you are about to commit, along with other modified files that haven't been added to the index yet.

```
$ git status
```

Once you're ready to commit, run the following command to commit changes to your local repository. The `-am` option tells git to automatically commit any tracked files that have

been modified (that's the `a` part of the option) and that you want to give a commit message right on the command line instead of starting up a text editor to write it (that's the `m` part of the option).

```
$ git commit -am "<a meaningful message about what you're committing>"
```

Beware, you haven't actually submitted anything for grading yet. You've just put these changes in your local git repo. To push changes up to your repo on github.ncsu.edu, you need to use the push command:

```
$ git push
```

Feel free to commit and push as often as you want. Whenever you've made a set of changes you're happy with, you can run the following to update your submission.

```
$ git add any-new-files
$ git status
$ git commit -am "<a meaningful message about what you're committing>"
$ git push
```

Keeping your repo clean

Be careful not to commit files that don't need to be part of your repo. Temporary files or files that can be easily re-generated will just take up space and obscure what's really changing as you modify your source code. And, the NCSU github site puts a file size limit on what you can push to your repo. Adding files you don't really need could create a problem for you later.

The `.gitignore` file helps with this, but it's always a good idea to check with `git status` before you commit, to make sure you're getting what you expect.

Checking Jenkins Feedback

It will take me a few days to get our Jenkins systems working. Once they're set up, I'll send out a note on Piazza, and these instructions should work.

We have created a Jenkins build job for you. Jenkins is a continuous integration server that is used by industry to automatically build and test applications as they are being developed. We'll be doing the same thing with your project as you push changes to the NCSU github. This will provide early feedback on the quality of your work and your progress toward completing the assignment.

The Jenkins job associated with your GitHub repository will poll GitHub every two minutes for changes. After you have pushed code to GitHub, Jenkins will notice the change and automatically start a build process on your code. The following actions will occur:

- Code will be pulled from your GitHub repository
- A testing script will be run to make sure you submitted all of the required files, to check your code against parts of the course style guidelines and to try out your solution on each of our provided test cases

Jenkins will record the results of each execution. To obtain your Jenkins feedback, do the following tasks (remember, after a push, you may have to wait a couple of minutes for the latest results to appear):

- Go to Jenkins for CSC230. Your web browser will probably complain that this site doesn't have a valid certificate. That's OK. Tell your browser to make an exception for this site and it should let you continue to the site.
- You'll need to authenticate with your unity ID and password.
- Click the project named *p2-<unityid>*
- There will be a table called *Build History* in the lower left, click the link for the latest build
- Click the *Console Output* link in the left menu (4th item)
- The console output provides the feedback from compiling your program and executing the provided test cases. If the bottom of the output says you failed some tests, there should be one or more lines earlier in the output that starts with four stars. These should say something about what parts of the test your program failed on.

Succeeding on Project 2

Be sure to follow the style guidelines and make sure your program compiles cleanly on the common platform with the required compile options. If you have your program producing the right output, it should be easy to clean up little problems with style or warnings from the compiler.

Be sure your files are named correctly, including capitalization. We'll have to charge you a few points if you submit something with the wrong name, and we have to rename it to evaluate your work.

We'll test your program with a few extra test cases when we're grading it. Maybe try to think about sneaky test cases we might try, like behaviors that are described in this write-up but not tested in any of the provided test cases. You might be able to write up a simple test case to try these out yourself.

There is a 24 hour window for **late submissions**. Use this if you need to, but try to keep all your points if you can. Getting started early can help you avoid this penalty.

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
- Correctly identify error messages and warnings from the preprocessor, compiler, and linker, and avoid them.
- Interpret and explain data types, conversions between data types, and the possibility of overflow and underflow

- Use the C preprocessor to control tracing of programs, compilation for different systems, and write simple macros.
- 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.
- Use an automatic packaging tool, such as make or ant, to distribute and maintain software that has multiple compilation units.
- Use a version control tools, such as subversion (svn) or git, to track changes and do parallel development of software.
- Distinguish key elements of the syntax (what's legal), semantics (what does it do), and pragmatics (how is it used) of a programming language.