Programming Assignment #1: Elevate

COP 3502, Summer 2018

Due: Tuesday, May 29, *before* 11:59 PM

Abstract

In this assignment, you will write a *main()* function that can process command line arguments – parameters that are typed at the command line and passed into your program right when it starts running (as opposed to using *scanf()* to get input from a user *after* your program has already started running). Those parameters will be passed to your *main()* function as an array of strings, and so this program will also jog your memory with respect to using strings in C.

On the software engineering side of things, you will learn to construct programs that use multiple source files and custom header (.h) files. This assignment will also help you hone your ability to acquire new knowledge by reading technical specifications. If you end up pursuing a career as a software developer, the ability to rapidly digest technical documentation and work with new software libraries will be absolutely critical to your work, and this assignment will provide you with an exercise in doing just that.

Finally, this assignment is specifically designed to require relatively few lines of code so that you can make your first foray into the world of Linux without a huge, unwieldy, and intimidating program to debug. As the semester progresses, the programming assignments will become more lengthy and complex, but for now, this assignment should provide you with a gentle introduction to a development environment and software engineering concepts that will most likely be quite foreign to you at first.

Deliverables

Elevate.c

Note! The capitalization and spelling of your filename matter!

Note! Code must be tested on Eustis, but submitted via Webcourses.

Totally Clickable Table of Contents

Tip: Feeling lost? Check out Appendix A (“Getting Started: A Guide for the Overwhelmed”) on pg. 12.

Another Tip: Any time you see a page number in this PDF, you can click it to jump to that page.

1. Overview

In this assignment, you'll code up an algorithm for determining the movement of elevators through a building. Here's the scenario: There's a building with nine floors (numbered 1 through 9) and two side-by-side elevators, referred to as the *left* and *right* elevators. The building has a neat feature: When someone presses a button to summon an elevator, they don't just hit an arrow indicating whether they're headed up or down. Instead, the elevator call buttons have been replaced with number pads where residents of the building punch in the number of the floor they want to travel to.

In this program, you'll have to determine which elevator to send – the *left* or *right* elevator – when someone presses a button. Any time someone presses a button to summon an elevator, you'll know the following things (each of which is simply a floor number on the range 1 through 9):

- the location of the *left* elevator
- the location of the *right* elevator
- the location of the person summoning the elevator
- the destination of the person summoning the elevator

Based on that information, here are the rules for determining which elevator to send when someone presses a button:

1. First, if one elevator is closer than the other to the floor where a button was pressed, then that closer elevator is sent to service the request. For example, if someone on floor 5 requests an elevator, and the *left* elevator is currently on floor 7 and the *right* elevator is currently on floor 1, then the *left* elevator is sent to pick up the person on floor 5 (regardless of where that person is headed), since that results in the shortest wait time for that person.

2. If a tie results from Rule #1 (i.e., if both elevators are equally far away from the floor where someone has pressed a button), then the elevator we choose to service the request is the one that leaves the elevators in the best possible configuration after delivering the passenger to their desired floor. The "best possible configuration" is defined as the configuration that results in the lowest number of floors the elevators would have to travel to service the next request, on average. (*Whaaaaaaatt?!* Yes, this is one of the more complex pieces of the puzzle, and it's described in far more detail below.)

3. If a tie results from both Rules #1 and #2 of this algorithm, then just send the elevator on the *right* to service the request.

To keep things simple, we'll assume that we're only ever processing one elevator request at any given moment. No one will request an elevator until the previous person's elevator trip is complete. So, we always move our elevators as if we have no knowledge of where the next elevator request will come from.

1.1 Rule #2: Computing the Average Wait Time

Suppose our *left* elevator is on floor 5, and our *right* elevator is on floor 7. Next, suppose someone on floor 6 requests an elevator, and they want to reach floor 2. Since both elevators are exactly one floor away from floor 6, Rule #1 results in a tie, and we must move on to Rule #2. We need to know: Which elevator (*left* or *right*) would leave us in a better overall situation if it were to pick this person up on floor 6 and deliver them to floor 2?

Here's how to compute that:

If the *left* elevator services the request, it will pick someone up on floor 6 and carry them to floor 2. This would result in the *left* elevator being on floor 2 and the *right* elevator still being on floor 7. To determine how good of a situation that is, let's look at the average number of floors the elevators would then have to travel to pick up whoever issues the very next request after that:

- If the next request comes from **floor 9**, the closest elevator (*right* = 7) will travel **2 floors** to get there.
- If the next request comes from **floor 8**, the closest elevator (*right* = 7) will travel **1 floor** to get there.
- If the next request comes from **floor 7**, the closest elevator (*right* = 7) will travel **0 floors** to get there.
- If the next request comes from **floor 6**, the closest elevator (*right* = 7) will travel **1 floor** to get there.
- If the next request comes from **floor 5**, the closest elevator (*right* = 7) will travel **2 floors** to get there.
- If the next request comes from **floor 4**, the closest elevator (*left* = 2) will travel **2 floors** to get there.
- If the next request comes from **floor 3**, the closest elevator (*left* = 2) will travel **1 floor** to get there.
- If the next request comes from **floor 2**, the closest elevator (*left* = 2) will travel **0 floors** to get there.
- If the next request comes from **floor 1**, the closest elevator (*left* = 2) will travel **1 floor** to get there.

In that scenario, the average distance we expect to travel to pick up the next passenger would be:

$$(2 + 1 + 0 + 1 + 2 + 2 + 1 + 0 + 1) / 9 \approx 1.11 \text{ floors}$$

On the other hand, if the *right* elevator services the request, it will pick someone up on floor 6 and carry them to floor 2. This would result in the *left* elevator still being on floor 5 and the *right* elevator being on floor 2. After that, if someone else issues a request for an elevator, here's what will happen:

- If the next request comes from **floor 9**, the closest elevator (*left* = 5) will travel **4 floors** to get there.
- If the next request comes from **floor 8**, the closest elevator (*left* = 5) will travel **3 floors** to get there.
- If the next request comes from **floor 7**, the closest elevator (*left* = 5) will travel **2 floors** to get there.
- If the next request comes from **floor 6**, the closest elevator (*left* = 5) will travel **1 floor** to get there.
- If the next request comes from **floor 5**, the closest elevator (*left* = 5) will travel **0 floors** to get there.
- If the next request comes from **floor 4**, the closest elevator (*left* = 5) will travel **1 floor** to get there.
- If the next request comes from **floor 3**, the closest elevator (*right* = 2) will travel **1 floor** to get there.
- If the next request comes from **floor 2**, the closest elevator (*right* = 2) will travel **0 floors** to get there.
- If the next request comes from **floor 1**, the closest elevator (*right* = 2) will travel **1 floor** to get there.

In that scenario, the average distance we expect to travel to pick up the next passenger would be:

$$(4 + 3 + 2 + 1 + 0 + 1 + 1 + 0 + 1) / 9 \approx 1.44 \text{ floors}$$

Therefore, we use the *left* elevator to service the request described above. That decision leaves the elevators in a configuration (*left* = 2, *right* = 7) that will give the next passenger faster pickup service on average (the elevators only having to travel 1.11 floors) than if we had moved the *right* elevator (which would have resulted in *left* = 5, *right* = 2, and an average travel of 1.44 floors to satisfy the next request – certainly slower than traveling just 1.11 floors).

1.2 Input and Output Specification

Plot twist! To get a sequence of elevator requests into your program, you won't read from a file, and you won't use *scanf()* to read the input from a user. Instead, all the elevator requests that your program needs to process will be passed to it via the command line. That is, we'll type the elevator requests we want your program to process before we even hit "enter" to start running your program, like so:

```
seansz@eustis:~$ ./elevate 3-2 9-5 1-2
1  1
1  2
1  5
2  5
seansz@eustis:~$ _
```

In the example above, *elevate* is the name of the program being run, and the remaining command line arguments represent all the elevator requests your program needs to process. A request will always be given in the form *X*-*Y*, where *X* is the floor where someone has requested an elevator (a number from 1 through 9), and *Y* is that passenger's destination (also a floor number, from 1 through 9). Those two numbers are always separated by a hyphen ('-'). In the example above, the first elevator request originates at floor 3, and that passenger wants to reach floor 2. The second request originates at floor 9, and that passenger wants to reach floor 5. The third request is from someone on floor 1 who wants to reach floor 2.

Each line of output produced by your program will have two integers. The first integer indicates what floor the *left* elevator is on, and the second integer indicates what floor the *right* elevator is on. The two integers must always be separated by two spaces, and each line should be terminated with a single newline character ('\n').

When your program is run, both elevators must always start on floor 1, so you should start by printing "1 1" to the screen. After that, there will be one line of output for each command line argument, showing what floors the elevators are on after each request has been processed. If you work through the example above on paper, you'll find that if both elevators are on floor 1 and someone wants to travel from floor 3 to floor 2, the *right* elevator will service the request. (So, the second line of output indicates that the *left* elevator is still on floor 1, and the *right* elevator is now on floor 2.) After that, someone summons an elevator to floor 9. The *right* elevator is closest, so it services the request and ends up on floor 5 (which is where the person on floor 9 was headed), so we print "1 5" to the screen. Finally, someone on floor 1 summons an elevator and travels to floor 2. We use the *left* elevator to satisfy that request (since it was closer to floor 1 than the *right* elevator), and the *left* elevator subsequently ends up on floor 2 (hence the final line of output, "2 5").

1.3 Processing Command Line Arguments and Extracting Integers from Strings

Instructions for getting your program to process command line arguments are included below in Appendix B, "Guide to Command Line Arguments" (pg. 16). Command line arguments are passed to C programs as strings, and so you'll need to extract two integers from each of those strings. Appendix C, "ASCII Character Values" (pg. 19), contains some helpful information on how to do that.

2. What happens next?

Here's what's left in this document:

- Section 4, "Elevate.c and Elevate.h," will tell you a bit about how to set up your source file for this assignment and what the heck is going on with the Elevate.h file we included with the assignment.
- Section 5, "Test Cases and the test-all.sh Script," talks a bit about how to use the test-all.sh script (included with this assignment) to get Eustis to automatically run your program on the attached test cases.
- Section 6, "Function Requirements," has some specific function requirements you'll want to review.
- Section 7, "Suggested Functions," has some tips on how to break your program up into functions to make it easier to get all the required components working.
- Section 8, "Special Restrictions," has – you guessed it – some special restrictions you have to follow in the code you submit for this assignment.
- Section 9, "Deliverables (Submitted via Webcourses, Not Eustis)," has some brief instructions for submitting the assignment.
- Section 10, "Grading" has some information about grading criteria and grading standards for the program.

There are also four appendices at the end of this PDF:

- Appendix A, "Getting Started: A Guide for the Overwhelmed," has some advice on how to get started with this assignment.
- Appendix B, "Processing Command Line Arguments," walks you through how to set up your program to process command line arguments.
- Appendix C, "ASCII Character Values," talks about how to extract integers from those command line argument strings.
- Appendix D, "Compilation and Testing," has all the details you need in order to run our test cases, whether you're in Eustis, working at the command line on your own system, or developing your code in CodeBlocks.

3. Important Note: Test Case Files Look Wonky in Notepad

Included with this assignment are several test cases, along with output files showing exactly what your output should look like when you run those test cases. You will have to refer to those as the gold standard for how your output should be formatted. Please note that if you open those files in Notepad, they will appear to contain one long line of text. That's because Notepad handles end-of-line characters differently from Linux and Unix-based systems. One solution is to view those files in an IDE (like CodeBlocks), or you could use a different text editor (like Atom or Sublime).

4. Elevate.c and Elevate.h

You will submit a single source file, named `Elevate.c`, that contains all required function definitions, as well as any auxiliary functions you deem necessary. In `Elevate.c`, you will have to #include any header files necessary for your functions to work, including the custom `Elevate.h` header file we have distributed with this assignment, which has functional prototypes for all the functions you will be implementing. You **must** #include that file from your `Elevate.c` file like so:

```
#include "Elevate.h"
```

The “quotes” (as opposed to <brackets>) indicate to the compiler that this header file is found in the same directory as your source file, not a system directory. Note that filenames are case sensitive in Linux, so if you `#include "elevate.h"` (with a lowercase ‘e’), your program will not compile. You must use an uppercase ‘E’.

You should not send `Elevate.h` when you submit your assignment, and you should be very careful about modifying `Elevate.h`. We will use our own unmodified copy of `Elevate.h` when compiling your program.

If you write auxiliary functions (“helper functions”) in your `Elevate.c` file, you should **not** add those functional prototypes to `Elevate.h`. Our test case programs will not call your helper functions directly, so they do not need any awareness of the fact that your helper functions even exist. (We only list functional prototypes in a header file if we want multiple source files to be able to call those functions.) So, just put the functional prototypes for any helper functions you write at the top of your `Elevate.c` file.

Think of `Elevate.h` as a bridge between source files. It contains functional prototypes for functions that might be defined in one source file (such as your `Elevate.c` file) and called from a different source file (such as the `UnitTestXX.c` files we have provided with this assignment).

5. Test Cases and the test-all.sh Script

The test cases included with this assignment are designed to show you some ways in which we might test your code and to shed light on the expected functionality of your code. We’ve also included a script, `test-all.sh`, that will compile and run all test cases for you.

Super Important: Using the `test-all.sh` script to test your code on Eustis is the safest, most sure-fire way to make sure your code is working properly before submitting.

You can run the script on Eustis by placing it in a directory with `Elevate.c`, `Elevate.h`, the `sample_output` directory, and all the test case files, and then typing:

```
bash test-all.sh
```

Some of the test cases we’ve given you are designed to test the functionality of your program when you run it with command line arguments, and others are source files that you can run by compiling them alongside your own code for this program (i.e., by compiling multiple source files into a single program). For further details about these different types of test cases, or for detailed instructions on how to run test cases in your own development environment, as well as how to run test cases individually rather than in one big batch, see Appendix D ("Compilation and Testing") on pg. 21.

Although we have included a variety of test cases to get you started with testing the functionality of your code, we encourage you to develop your own test cases, as well. Ours are by no means comprehensive. We will use much more elaborate test cases when grading your submission. In creating your own test cases, you should always ask yourself, "What kinds of inputs could be passed to this program that don't violate any of the input specifications, but which haven't already been covered in the test cases included with the assignment?"

6. Function Requirements

In the source file you submit, `Elevate.c`, you must implement the following functions. You may implement auxiliary functions (helper functions) to make these work, as well, although that is probably unnecessary for this assignment. Please be sure the spelling, capitalization, and return types of your functions match these prototypes exactly.

```
int main(int argc, char **argv);
```

Description: Start by printing "1 1" to the screen to denote that both elevators are on the first floor. Then, process each command line argument, always printing where the elevators end up after each argument is processed, as described in Section 1 ("Overview").

Note that there should be two spaces between the numbers printed on each line, and each line should terminate with a newline character ('\n') If no arguments are passed to the program at the command line, the only output should be "1 1" followed by a newline character.

Also note that input specifications are a contract; you may assume any arguments passed to `main()` will follow the format described in Section 1 ("Overview").

Return Value: You must return zero from this function. Returning any value other than zero from `main()` could result in catastrophic test case failure when we grade your program.

```
double difficultyRating(void);
```

Output: This function should not print anything to the screen.

Return Value: A double indicating how difficult you found this assignment on a scale of 1.0 (ridiculously easy) through 5.0 (insanely difficult).

```
double hoursSpent(void);
```

Output: This function should not print anything to the screen.

Return Value: A reasonable estimate (greater than zero) of the number of hours you spent on this assignment.

7. Suggested Functions

Here are some functions I found helpful when writing up my solution to this program. You don't have to write these, but breaking things up into functions makes it easier to write sophisticated pieces of code, and will also make your code easier to read and trace through.

```
double averageWaitTime(int num_floors, int left, int right);
```

Description: This function takes the number of floors in the building (`num_floors`), the floor where the left elevator is located (`left`), and the floor where the right elevator is located (`right`), and returns the average number of floors an elevator will have to travel if someone calls an elevator.

Side Note: The first parameter, `num_floors`, is not strictly necessary. I included it when I first threw together my solution to this assignment because I have a healthy aversion to hard-coding certain types of constants in my code. This ended up being helpful, because in my original draft for this assignment, there were only four floors in the building. When I changed that number to nine, it was easy to modify my code to work with the new building parameters. Having that parameter there makes my code flexible, adaptable, and reusable.

Return Value: A double indicating the expected number of floors an elevator would have to travel in order to service a request. Admittedly, "averageWaitTime" is a bit of a misnomer, because the function's return value is measured not in time, but in the number of floors traveled.

```
int smallestDelta(int pickup_floor, int left, int right);
```

Description: This function takes the floor where someone is requesting an elevator (`pickup_floor`), the floor where the left elevator is located (`left`), and the floor where the right elevator is located (`right`), and returns the number of floors the closest elevator would have to travel in order to reach `pickup_floor`. This function will not help you find which elevator needs to be dispatched to `pickup_floor`, but I did find it helpful to call this function repeatedly from my `averageWaitTime()` function.

Return Value: A non-negative integer indicating how many floors the nearest elevator would have to travel to reach `pickup_floor`.

```
int delta(int pickup_floor, int elevator_floor);
```

Description: This function takes the floor where someone is requesting an elevator (`pickup_floor`) and the floor where one of the elevators is located (`elevator_floor`), and returns the number of floors that elevator would have to travel to reach `pickup_floor`. I called this function twice for each command line argument I was processing in `main()` – once to see how far away the *left* elevator was, and once to see how far away the *right* elevator was. Outsourcing that task to a function isn't strictly necessary, but it improved the readability of my code.

Return Value: A non-negative integer indicating how many floors the given elevator would have to travel to reach `pickup_floor`.

8. Special Restrictions

You must adhere to the following special restrictions when writing this assignment:

- **No Global Variables**
  You may not use any global variables in this assignment. (A global variable is a variable that is declared outside of a function – e.g., just below a `#define` statement – and can therefore be accessed by any and all functions in your source file.)

- **No Standard C Library Function Calls Except *printf()***
  You should not need any standard C functions except for `printf()`. If you find yourself calling weird functions from C libraries (like string tokenizer functions or functions for converting strings to integers), you are over-complicating the problem. Again, feel free to come talk to us if you want some ideas on how to stop using those functions! Remember, as long as you're not sharing your code, you can also talk with other people in the class about how to approach this.

- **No Array Creation**
  You cannot create any arrays (statically or dynamically) anywhere in your code, other than the `argv` array used to pass command line arguments to `main()`. If you find yourself using an array in this assignment, you are probably using unnecessary space. Feel free to come to talk to us about your approach, and we can nudge you in a direction that will allow you to solve the problem without using an array.

- **No File I/O**
  Your program cannot read or write to any files.

- **No *goto* Statements**
  If you don't know what a `goto` statement is, awesome. You should never use `goto` statements in your code. They are explicitly banned in this class.

9. Deliverables (Submitted via Webcourses, Not Eustis)

Submit a single source file, named `Elevate.c`, via Webcourses. The source file should contain definitions for all the required functions (listed above), as well as any auxiliary functions you need to make them work. Don't forget to `#include "Elevate.h"` in your source code.

Do not submit additional source files, and do not submit a modified `Elevate.h` header file. Your source file must work with the `test-all.sh` script, and it must be able to compile and run like so:

```
gcc Elevate.c
./a.out 3-2 9-5 1-2
```

Be sure to include your name and NID as a comment at the top of your source file.

10. Grading

The *tentative* scoring breakdown (not set in stone) for this programming assignment is:

65%	correct output for test cases (not all of which have been release with this project)
10%	implementation details (manual inspection of your code)
5%	`difficultyRating()` is implemented correctly
5%	`hoursSpent()` is implemented correctly
5%	source file is named correctly (`Elevate.c`); spelling and capitalization count
10%	adequate comments and whitespace; source includes student name and NID

Note! Your program must be submitted via Webcourses, and it must compile and run on Eustis to receive credit. Programs that do not compile will receive an automatic zero.

Your grade will be based largely on your program's ability to compile and produce the *exact* output expected. Even minor deviations (such as capitalization or punctuation errors) in your output will cause your program's output to be marked as incorrect, resulting in severe point deductions. The same is true of how you name your functions and their parameters. Please be sure to follow all requirements carefully and test your program throughly.

Additional points will be awarded for style (appropriate commenting and whitespace) and adherence to implementation requirements. For example, the graders might inspect your code to ensure that you're not creating arrays or calling any functions from standard C libraries, except for `printf()`.

Start early. Work hard. Good luck!

Appendix A

Getting Started: A Guide for the Overwhelmed

Getting Started: A Guide for the Overwhelmed

Okay, so, this might all be overwhelming, and you might be thinking, “Where do I even start with this assignment?! I’m in way over my head!” First and foremost, let me say this:

DON’T

PANIC

There are plenty of TA office hours where you can get help, and here’s my general advice on starting the assignment:

1. First and foremost, start working on this assignment early. Nothing will be more frustrating than running into unexpected errors or not being able to figure out what the assignment is asking you to do on the day that it is due.

2. Secondly, glance through all the section headings in this PDF to get an idea of what kind of information it contains.

3. Thirdly, read over the PDF to get an idea of what the assignment is asking you to do. Section 1, “Overview,” will be the most important part to start out. Some sections won’t make sense right away, and it’s okay to skim them if that’s the case, but don’t blow them off entirely.

4. After you’ve read through the PDF, but before you start programming, open up the test cases and sample output files included with this assignment and trace through a few of them to be sure you have an accurate understanding of what you’re supposed to be doing. Refer back to relevant sections of the PDF (the ones you might have skimmed in your first pass through the PDF) to clarify any questions you might have about those test cases.

5. Once you’re ready to begin coding, start by creating a skeleton `Elevate.c` file. Add a header comment with your name and NID, add one or two standard `#include` directives, and be sure to

`#include "Elevate.h"` from your source file. Then copy and paste each functional prototype from `Elevate.h` into `Elevate.c`, transform those prototypes into functions by removing the semicolon and setting up curly braces for each function signature, and set up all those functions to return dummy values (some arbitrary `int` or `double`, as appropriate).

6. Set up your `main()` function to take command line arguments. (For details on that, see Appendix B: Processing Command Line Arguments on pg. 16 of this PDF). Be sure your `main()` function returns zero.

7. Test that your `Elevate.c` source file compiles. Do this before you've even taken a stab at making the functions work! If you're at the command line on a Mac or in Linux, your source file will need to be in the same directory as `Elevate.h`, and you can test its ability to compile like so:

   ```
   gcc Elevate.c
   ```

   Alternatively, you can try compiling it with one of the unit test source files using the instructions set forth in "Compiling and Running Unit Tests at the Command Line (Linux/Mac)" (Appendix D, pg. 26).

   If you're using CodeBlocks, you'll want to start by creating a project using the instructions below in "Compilation and Testing with CodeBlocks" (Appendix D, pg. 24). Import `Elevate.h` and your new `Elevate.c` source file into your project, and get the program compiling and running before you move forward. (Note that CodeBlocks is the only IDE we officially support in this class.) Instructions for compiling unit tests in CodeBlocks are included in "Running Unit Tests with CodeBlocks" (Appendix D, pg. 24).

8. Once you have your project compiling, see if you can set up a basic for-loop that prints all command line arguments to the screen, one by one. Then see if you can extract the two integers from each command line argument, store them in `int` variables, and print those to the screen correctly. You'll want to be sure that your `int` variables have the correct numbers in them for each command line argument before you start passing them to functions and trying to determine which elevators are moving in which direction.

9. Once you have that working, go back to Section 6, "Function Requirements," and read through the function requirements. Do some brainstorming on how you want to solve the problem. If you followed through with Step #4 in this section, then you've already programmed your brain to solve this problem – neat! Now you just need to take that brain code and turn it into C code. At this point, you might want to trace through some test cases again (and revisit Section 1, "Overview," if necessary). As you work through test cases on paper, pay attention to what step-by-step process your brain is using to solve this problem. The process you're following is going to involve some if-else logic, and if you pay attention to it, it'll start to look a lot like code.

10. Once you have an idea of how you want to write `main()` (even if it feels a bit vague at first), dive in fearlessly. Be bold! Don't hesitate to run code that you think might not work; you won't break anything. You also don't have to finish writing all of main() before you run your code. You should run your code frequently to see if each step is working.

General Tips for Working on Programming Assignments

Here's some general advice that might serve you well in this and all remaining assignments for the course:

1. In general, you'll want to stop to check whether your code compiles after every one or two significant blocks or chunks of code you throw down. That's rather moot in this assignment, since there isn't much actual code to write. However, throughout the semester, if you frequently stop to check whether your code is compiling (rather than writing 30-100 lines of code before checking whether any of it is even syntactically correct), you will save yourself a lot of headaches.

2. If you get stuck while working on an assignment, draw diagrams on paper or a whiteboard. Make boxes for all the variables in your program. Trace through your code carefully, step by step, using these diagrams.

3. You're bound to encounter errors in your code at some point. Use `printf()` statements liberally to verify that your code is producing the results you think it should be producing (rather than making assumptions that certain components are working as intended). You should get in the habit of being immensely skeptical of your own code and using `printf()` to provide yourself with evidence that your code does what you think it does.

4. If you encounter a segmentation fault, you should always be able to use `printf()` and `fflush()` to track down the *exact* line you're crashing on.

5. When you find a bug, or if your program is crashing on a huge test case, don't trace through hundreds of iterations of some for-loop to track down the error. Instead, try to cook up a very small, new test case (as few lines as possible) that causes your code to crash. The smaller the test case that you have to trace through, the easier your debugging task will be.

6. You will eventually want to read up on how to set break points and use a debugger. Some helpful Google queries might be: CodeBlocks debugger and gdb debugging tutorial.

Appendix B:

Processing Command Line Arguments

Overview: Command Line Arguments

All your interactions with Eustis this semester will be at the command line, where you will use a text-based interface (rather than a graphical interface) to interact with the operating system and run programs.

When we type the name of a program to run at the command line, we often type additional parameters *after* the name of the program we want to run. Those parameters are called "command line arguments," and they are passed to the program's `main()` function upon execution.

For example, in class, you've seen that I run the program called *gcc* to compile source code, and after typing "*gcc*," I always type the name of the file I want to compile, like so:

```
gcc Elevate.c
```

In this example, the string "*Elevate.c*" is passed to the *gcc* program's `main()` function as a string, which tells the program which file it's supposed to open and compile.

In this assignment, your `main()` function will have to processes command line arguments. The following sections show you how to get that set up.

Passing Command Line Arguments to *main()*

Your program must be able to process any number of command line arguments, which are the strings representing elevator requests. For example:

```
seansz@eustis:~$ ./a.out 3-1 4-2
1  1
1  1
1  2
seansz@eustis:~$ _
```

You don't have to do anything drastically complicated to get command line arguments (like the strings "3-1" and "4-2" in this example) into your program. You just have to change the function signature for the `main()` function you're writing in `Elevate.c`. Whereas we have typically seen `main()` defined using `int main(void)`, you will now use the following function signature instead:

```
int main(int argc, char **argv)
```

Within `main()`, `argc` is now an integer representing the number of command line arguments passed to the program (including the name of the executable itself – so in the example above where we ran `./a.out 3-1 4-2`, `argc` would be equal to 3). `argv` is an array of strings that stores all those command line arguments. `argv[0]` always stores the name of the program being executed ("./a.out"), and in the example given above, `argv[1]` stores the string "3-1," and `argv[2]` contains "4-2".

Example: A Program That Prints All Command Line Arguments

For example, here's a small program that would print out all the command line arguments it receives (including the name of the program being executed). Note how we use `argc` to loop through the array:

```
#include <stdio.h>

int main(int argc, char **argv)
{
    int i;
    for (i = 0; i < argc; i++)
        printf("argv[%d]: %s\n", i, argv[i]);
    return 0;
}
```

If we compiled that code into an executable file called `a.out` and ran it from the command line by typing `./a.out lol hi there!`, we would see the following output:

```
argv[0]: ./a.out
argv[1]: lol
argv[2]: hi
argv[3]: there!
```

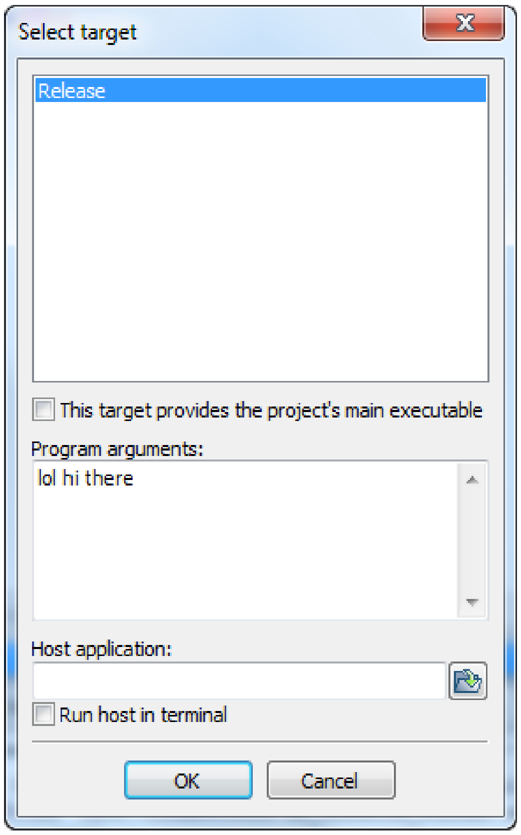
Passing Command Line Arguments in CodeBlocks

You can set up CodeBlocks to automatically pass command line arguments to your program every time you hit *Build and run (F9)* within the IDE. Simply go to *Project → Set program's arguments...*, and in the box labeled *Program arguments*, type your desired command line arguments.

For example, the setup shown here passes `lol hi there` as command line arguments to the program when compiled and run within CodeBlocks. (I don't think you need to check off the box next to "*This target provides....*" CodeBlocks seems to take care of that automatically after you hit "OK.")

Those three strings will be passed as `argv[1]`, `argv[2]`, and `argv[3]`. `argv[0]` will contain the name of your executable file automatically and should not be listed in the *Program arguments* box.

If you compile your program like this in CodeBlocks and later run your executable from the command line on Eustis (or even on your own machine), you will still need to supply fresh arguments for your program at the command line.


Select target

Release

This target provides the project's main executable

Program arguments:

lol hi there

Host application:

Run host in terminal

Appendix C:

ASCII Character Values

ASCII Character Values

As you already know, when we run your program, we'll be passing command line arguments that you need to process. Each of those arguments is a string (such as "3-5"), but you need to extract two separate integers from those strings. To do that, you need to know a bit about how C represents the individual characters in a string.

In C, each character has an underlying integer value called its ASCII value. (ASCII is an international standard for character numbering. If you want to read more about the topic, see the ASCII article on Wikipedia. For this assignment, however, you don't need to know any more about ASCII than what I've written up on this page.)

To see a character's underlying integer value, you can always just print it as an integer, like so:

```
printf("%d", 'a');
```

The ASCII value for 'a' is 97, so the above line of code should print out 97.

Lowercase letters are numbered sequentially: 'a' is 97, 'b' is 98, 'c' is 99, and so on. If for some reason you wanted to convert the characters 'a' through 'z' to integers 0 through 25 (maybe so you could use them to access indices in an array of length 26?), you could just subtract 97 from those characters. For example:

```
printf("%d", 'a' - 97);  // 'a' - 97 = 97 – 97 = 0
printf("%d", 'b' - 97);  // 'b' - 97 = 98 – 97 = 1
printf("%d", 'c' - 97);  // 'c' - 97 = 99 – 97 = 2
```

Personally, I never hard-code the value 97 when converting characters to integers (partly because I never used to be able to remember it off the top of my head). Instead, I just use 'a' in place of 97, because the math will always work out:

```
printf("%d", 'a' - 'a');  // 'a' - 'a' = 97 – 97 = 0
printf("%d", 'b' - 'a');  // 'b' - 'a' = 98 – 97 = 1
printf("%d", 'c' - 'a');  // 'c' - 'a' = 99 – 97 = 2
```

The really useful thing you need to know here is that the character '0' (that's a zero in single quotes) is not the same as the integer 0. The underlying ASCII value for '0' is actually 48:

```
printf("%d", '0');  // This prints 48
```

Similarly, '1' is 49 in ASCII, '2' is 50, '3' is 51, and so on. So, if you want to convert individual characters in a string to their corresponding values, you can simply subtract '0' (much in the same way that we subtracted 'a') above:

```
printf("%d", '0' - '0');  // '0' - '0' = 48 – 48 = 0
printf("%d", '1' - '0');  // '1' - '0' = 49 – 48 = 1
printf("%d", '2' - '0');  // '2' - '0' = 50 – 48 = 2
```

You know that a string is just a character array, so if `argv[1]` is "3-5", the '3' is stored at `argv[1][0]`, and the '5' is stored at `argv[1][2]`. You'll need to extract those characters as integers.

Appendix D:

Compilation and Testing

Overview

This appendix has instructions for compiling and running test cases in a variety of different contexts.

Dissecting the Contents of the *testcases* Folder

This project includes a folder named "*testcases*." In that folder, you'll find the following goodies:

1. The `test-all.sh` script (discussed below in Section 5, "Test Cases and the test-all.sh Script"). This little beast is made with love and can compile and run your code on all the test cases.

2. Test cases named `arguments01.txt` through `arguments10.txt`. These files contain command line arguments to pass to your program for testing. Instructions on how to use these files are given on pg. 22: "Running Standard Test Cases (arguments01.txt through arguments10.txt)."

3. Test cases named `UnitTest11.c` and `UnitTest12.c`. A unit test is a test case that runs a specific segment of your code to check that it's producing the correct results. In this case, each unit test will call one of the functions you're required to write for this assignment. These unit tests are source files that you will have to compile into a program along with your `Elevate.c` source file and the `UnitTestLauncher.c` source file included in this folder. Instructions for working with these unit tests are on pg. 23: "Running Unit Test Cases (UnitTest11.c and UnitTest12.c)."

4. A source file named `UnitTestLauncher.c`. This source file has to be compiled into your program if (and only if) you are running a unit test. This process is described on pg. 23: "Running Unit Test Cases (UnitTest11.c and UnitTest12.c)."

5. A folder named `sample_output`. This folder contains output files named `output01.txt` through `output12.txt`. These files show exactly what your output should look like for different test cases (provided that you do **<u>not</u>** open them in Notepad; see Section 3, "Important Note: Test Case Files Look Wonky in Notepad").

   The `arguments01.txt` through `arguments10.txt` test cases should produce the output given in `output01.txt` through `output10.txt`, and the `UnitTest11.c` and `UnitTest12.c` test cases should produce the output given in `output11.txt` and `output12.txt`. For details on how to ensure that your output is a 100% match for the expected output given in these text files, see Section 5 ("Test Cases and the test-all.sh Script").

Running Standard Test Cases (*arguments01.txt* through *arguments10.txt*)

There are four ways to run the standard test cases (`arguments01.txt` through `arguments10.txt`) included with this assignment:

1. Compile your program at the command line, and then copy and paste the arguments from one of those files into the command prompt after "./a.out." You can use *diff* to check the difference between your output and the expected output. For example, `arguments01.txt` contains the text "`3-2 9-5 1-2                         `." (Yes, there are spaces at the end of that text. Those spaces

help `test-all.sh` make all of its output look pretty, and spaces at the end of your command line input get ignored when the arguments are passed to `main()` anyway, so you needn't worry about them.) You can run that test case and check your output against `output01.txt` like so:

```
seansz@eustis:~$ gcc Elevate.c
seansz@eustis:~$ ./a.out 3-2 9-5 1-2                  > myoutput.txt
seansz@eustis:~$ diff myoutput.txt sample_output/output01.txt
seansz@eustis:~$ _
```

The lack of response from the *diff* command indicates that your output was correct. Yay!

2. Instead of copying and pasting text from `arguments01.txt`, you can also use the following magical incantations to make the Linux command line pass the contents of `arguments01.txt` to your program as command line arguments for you. (This works with the Mac terminal, too!)

```
seansz@eustis:~$ gcc Elevate.c
seansz@eustis:~$ ./a.out $(cat arguments01.txt) > myoutput.txt
seansz@eustis:~$ diff myoutput.txt sample_output/output01.txt
seansz@eustis:~$ _
```

   Again, the lack of any grumbling from *diff* indicates that your output was correct.

3. If you're using CodeBlocks, you can copy and paste the contents of `arguments01.txt` (or any of the other argument-based test case files) into the program arguments interface as described above in Appendix B ("Processing Command Line Arguments," pg. 16). Then, just hit the button to compile and run your program.

4. Alternatively, at a Linux or Mac command line (or in the new bash shell in Windows), you can run the `test-all.sh` script and let it do all the work for you. It will compile and run your source code on all the test cases, and give you a report of the results. For details on how to do that, see Section 5, "Test Cases and the test-all.sh Script," on pg. 7.

Running Unit Test Cases (*UnitTest11.c* and *UnitTest12.c*)

The unit tests are a bit more tricky to get running. Here are your options:

1. The easiest option is to simply let the `test-all.sh` script do all the work for you, but you'll learn more if you try one of the following options as well. For details on how to use the `test-all.sh` script, see Section 5, "Test Cases and the test-all.sh Script."

2. You can run each unit test individually at the command line (Linux, Mac, or the new bash shell in Windows) by following the detailed instructions on pg. 26 ("Compiling and Running Unit Tests at the Command Line (Linux/Mac)").

3. You can run each unit test individually with CodeBlocks by following the detailed instructions on pg. 24 ("Running Unit Tests with CodeBlocks").

Compilation and Testing with CodeBlocks

The key to getting multiple files to compile into a single program in CodeBlocks (or any IDE) is to create a project. Here are the step-by-step instructions for creating a project in CodeBlocks, which involves importing `Elevate.h` and the `Elevate.c` file you've created (even if it's just an empty file for now).

1. Start CodeBlocks.
2. Create a New Project (*File* -> *New* -> *Project*).
3. Choose "Empty Project" and click "Go."
4. In the Project Wizard that opens, click "Next."
5. Input a title for your project (e.g., "Elevate").
6. Choose a folder (e.g., Desktop) where CodeBlocks can create a subdirectory for the project.
7. Click "Finish."

Now you need to import your files. You have two options:

1. Drag your source and header files into CodeBlocks. Then right click the tab for **<u>each</u>** file and choose "Add file to active project."

   – or –

2. Go to *Project* -> *Add Files...*. Browse to the directory with the source and header files you want to import. Select the files from the list (using CTRL-click to select multiple files). Click "Open." In the dialog box that pops up, click "OK."

You should now be good to go. Try to build and run the project (F9).

Running Unit Tests with CodeBlocks

If you want to run a unit test (one of the test cases that is a source file), you must also do the following:

1. Add the unit test you want to run (e.g., `UnitTest11.c`) to your project. Note that you can only have one unit test in your project at a time. For example, if you import both `UnitTest11.c` *and* `UnitTest12.c`, the compiler will complain that you have multiple definitions for `main()`.

2. Add `UnitTestLauncher.c` to your project.

3. Find the line in `Elevate.h` that says `#define main __hidden_main__` and ensure that it is **<u>not</u>** commented out (i.e., remove the "`//`" from the beginning of the line).

If you want to go back to running normal test cases, you must remove the `UnitTestNN.c` and `UnitTestLauncher.c` files from your CodeBlocks project, and you must once again comment out the line in `Elevate.h` that says `#define main __hidden_main__`. You **<u>must</u>** use "`//`" to comment out that line, with no space after the "`//`," like so: `//#define main __hidden_main__`.

Compilation and Testing at the Command Line (Linux/Mac)

To compile your source file (`.c` file) at the command line:

```
gcc Elevate.c
```

By default, this will produce an executable file called `a.out`, which you can run with command line arguments like so:

```
./a.out 3-2 9-5 1-2
```

If you want to name the executable file something else, use:

```
gcc Elevate.c -o Elevate.exe
```

...and then run the program with command line arguments like so:

```
./Elevate.exe 3-2 9-5 1-2
```

Running the program could potentially dump a lot of text to the screen. If you want to redirect your program's output to a text file in Linux, it's easy. Just run the program using the following command, which will create a file called `whatever.txt` that contains the output from your program:

```
./Elevate.exe 3-2 9-5 1-2 > whatever.txt
```

Linux has a helpful command called `diff` for comparing the contents of two files, which is really helpful here since we've provided several sample output files. You can see whether your output matches ours exactly by typing, e.g.:

```
diff whatever.txt sample_output/output01.txt
```

If the contents of `whatever.txt` and `output01.txt` are exactly the same, `diff` won't have any output. It will just look like this:

```
seansz@eustis:~$ diff whatever.txt sample_output/output01.txt
seansz@eustis:~$ _
```

If the files differ, it will spit out some information about the lines that aren't the same. For example:

```
seansz@eustis:~$ diff whatever.txt sample_output/output01.txt
1c1
< 1, 1
---
> 1  1
seansz@eustis:~$ _
```

Compiling and Running Unit Tests at the Command Line (Linux/Mac)

Compiling a unit test at the command line requires you to compile three source files into one program. For example, if you want to compile `UnitTest11.c`, you must compile it along with your `Elevate.c` file and the `UnitTestLauncher.c` file we have included with this project.

Important: In order for this to work, you **must** ensure the `#define main __hidden_main__` line in `Elevate.h` is **not** commented out (i.e., remove the "`//`" from the beginning of that line).

Once you've done that, you can compile these three source files into a single program like so:

```
gcc Elevate.c UnitTest11.c UnitTestLauncher.c
```

By default, that gcc command will produce an executable file called `a.out`. You can now run the unit test like so:

```
./a.out
```

You can redirect the output of the unit test to a text file like so:

```
./a.out > whatever.txt
```

Linux has a helpful command called `diff` for comparing the contents of two files, which is really helpful here since we've provided several sample output files. You can see whether your output matches ours exactly by typing, e.g.:

```
diff whatever.txt sample_output/output11.txt
```

If the contents of `whatever.txt` and `output01.txt` are exactly the same, `diff` won't have any output. It will just look like this:

```
seansz@eustis:~$ diff whatever.txt sample_output/output11.txt
seansz@eustis:~$ _
```

If the files differ, it will spit out some information about the lines that aren't the same, as we saw on the previous page.

An Important Note About Messing with Elevate.h

Super Important: After running unit tests, before you can run standard test cases again, you will need to comment out the `#define main __hidden_main__` line in `Elevate.h`. You **must** use "`//`" to comment out that line, with no space after the "`//`," like so: `//#define main __hidden_main__`.

Uncommenting that line essentially kills your `main()` function in `Elevate.c` and allows one of the unit test functions to take over the program. Commenting out that line again brings your `main()` function back to life.