Programming Assignment 1 (PA1) - Display Diamond

Milestone Due: Wednesday, January 24 @ 11:59 pm
Final Due: Wednesday, January 31 @ 11:59 pm

Assignment Overview
The purpose of this assignment is to build your knowledge of the ARM assembly language, especially branching and looping logic, calling assembly routines from within a C program, calling C functions from within assembly routines, passing parameters and returning values, using Unix command line arguments, and learning some useful Standard C Library routines.

You will be writing a program that takes 4 inputs from the command line and prints a diamond design to stdout. See man ascii for a map of the ASCII character set. This assignment will require appropriate error checking and reporting (as documented below).

Start early! Remember that you can and should use man in order to lookup information on specific C functions. For example, if you would like to know what type of parameters strtol() takes, or what strtol() does to errno, type man -s3 strtol. Also, take advantage of the tutors in the lab. They are there to help you learn more on your own and help you get through the course!

Grading
- README: 10 points - See README File section
- Compiling: 5 points - Using our Makefile; no warnings. If what you turn in does not compile with the given Makefile, you will receive 0 points for this assignment. NO EXCEPTIONS!
- Style: 20 points - See Style Requirements here
- Correctness: 65 points
  - Milestone (15 points) - To be distributed across the Milestone functions (see below)
  - Make sure you have all files tracked in Git.
- Extra Credit: 5 points - View Extra Credit section for more information.
- Wrong Language: You will lose 10 points for each module in the wrong language, C vs. Assembly or vice versa.

NOTE: If what you turn in does not compile with given Makefile, you will receive 0 points for this assignment.

Getting Started
Follow these steps to acquire the starter files and prepare your Git repository.

Gathering Starter Files:
The first step is to gather all the appropriate files for this assignment. Connect to pi-cluster via ssh.

$ ssh cs30xyz@pi-cluster.ucsd.edu

Create and enter the pa1 working directory.
$ mkdir ~/pa1
$ cd ~/pa1

Copy the starter files from the public directory.
$ cp ~/../public/palStarterFiles/* ~/pa1/
Starter files provided:

- pa1.h
- pa1Strings.h
- test.h
- testcheckRange.c
- Makefile

Preparing Git Repository:
You are required to use Git with this and all future programming assignments. Refer to the PA0 writeup for how to set up your local git repository.

Sample Output
A sample executable provided for you to try and compare your output against is available in the public directory. Note that you cannot copy it to your own directory; you can only run it using the following command (where you will also pass in the command line arguments):

```
$ ~/../public/pa1test
```

If there is a discrepancy between the sample output in this document and the pa1test output, follow the pa1test output.

The output of your program MUST match exactly as it appears in the pa1test output. You need to pay attention to the order of error messages!

Important note for entering arguments on the command line:
When you enter alphanumeric and some non-alphanumeric ASCII characters on the command line, you can choose to enclose the character in single quotes ("a"), double quotes("a"), or no quotes at all.

Example:
```
./pa1 11 7 "a" 'b'
```

However, while this will work for all alphanumeric characters, this will not work for some non-alphanumeric characters! Some characters, such as >, #, <, and others, MUST be enclosed with either single or double quotes in order for the shell to read them properly.

Example:
```
./pa1 11 7 "#" '<'
```

Below are some brief example outputs of this program. Make sure you experiment with the public executable to further understand the program behavior. Bolded text is what you type in the terminal.

1. Command-line Parsing Errors
1.1. Too many arguments (extra operand).

```
cs30xyz@pi-cluster-001:pa1$ ./pa1 7 5 a b c
```

Usage: ./pa1 width height border_char diamond_char

- width (must be odd within the range of [3 - 21])
- height (must be odd within the range of [1 - 19])
  (must be less than width)
- border_char (must be single char within the ASCII range [33 - 126])
  (must be different than diamond_char)
1.2. No arguments.
Usage: ./pa1 width height border_char diamond_char
width        (must be odd within the range of [3 - 21])
height       (must be odd within the range of [1 - 19])
              (must be less than width)
border_char  (must be single char within the ASCII range [33 - 126])
              (must be different than diamond_char)
diamond_char (must be single char within the ASCII range [33 - 126])
              (must be different than border_char)

1.3 Invalid width strtol conversion (no errno).
width(7a) must be an integer

1.4 Errno set in width strtol conversion.
converting width(77777777777777777777) in base(10) to long: Numerical result out of range

2. Other Errors
2.1 Height not less than width.
height(7) must be less than width(7)

2.2 Border_char and diamond_char are the same.
border_char(a) and diamond_char(a) must be different
2.3. Border_char not within range.

```
./pal 7 5 ' ' a
```

border_char( ) must be within the ASCII range [33 - 126]

2.4. Width not odd.

```
./pal 6 5 a b
```

width(6) must be an odd number

2.5. Multiple errors.

```
./pal 6 9 a aa
```

width(6) must be an odd number

height(9) must be less than width(6)

donad_char(aa) must be a single character

3. Valid Output

3.1. width = 7, height = 5, border_char = H, diamond_char = o.

```
./pal 7 5 H o
```

HHHHHHH
HHHoHHH
HHooHHH
HooooOH
HHooHHH
HHHoHHH
HHHHHHH
```

3.2. width = 15, height = 14, border_char = '!', diamond_char = @.

```
./pal 15 13 ! @
```

!!!!!!!!!!!!!!!!
!!!!!!!!@!!!!!!!
!!!!!!!!@@!!!!!!
!!!!!!!!@@@@@@@@
!!!!@@@@@@@@@@@
@@@@@@@@@@@@@@
@@@@@@@@@@@@@@
!!!!!!!!@@@@@@@@
!!!!!!!!@@@!!!
!!!!!!!!@!!!
!!!!!!!!!!!!
!!!!!!!!!!!!
!!!!!!!!!!!!
!!!!!!!!!!!!
3.3. Using octal and hex ascii codes for legal characters.

    width = 15, height = 13, border_char = '%' diamond_char = "".

Detailed Overview

The function prototypes for the various C and Assembly functions are as follows.

**C routines:**

    int main( int argc, char * argv[] );

    Assembly routines:

    long checkRange( long value, long minRange, long maxRange );
    long rem( long dividend, long divisor );
    long isOdd( long value );
    void printChar( char ch );
    void displayDiamond( long width, long height, char border, char diamond );

For the Milestone, you will need to complete:

    checkRange.s    rem.s    isOdd.s

Process Overview:

The following is an explanation of the main tasks of the assignment, and how the individual functions work together to form the whole program.
This program takes 4 command line arguments:
   $ ./pa1 width height border_char diamond_char

Explanation of Command Line Arguments:
- **width** - the width of the diamond pattern, **including the border**
- **height** - the height of the diamond, **not including the border**
- **border_char** - the character used to draw the border
- **diamond_char** - the character used to draw the diamond

Drawing the diamond pattern consists of the following steps:
1. Parse command line arguments in `main()`, where `checkRange()` and `isOdd()` will help with the error checking.
2. Draw the diamond pattern using `displayDiamond()`, where `printChar()` will be used to print the individual characters.

C Functions to be Written
Listed below are the modules to be written in C.

**main.c**
```c
int main( int argc, char * argv[] );
```

The main function will drive the rest of the program. It will first perform input checking by parsing the command-line arguments and checking for errors. If all inputs are valid, it will call `displayDiamond()`. If any of the input checks fail, it will print the corresponding errors, the usage and then exit. Keep in mind that all the error strings have format specifiers, so be sure to add the appropriate arguments when printing error messages. Make sure you use your `checkRange()` function when checking the limits of the command line arguments.

Note that your variable names should all be in camelCase (e.g. `sampleVariableName`, not `sample_variable_name`).

First, check that the user entered the correct number of command line arguments. If they didn't, print the `STR_USAGE` and return `EXIT_FAILURE` right away.

Now we can parse the command line arguments:
1. **width**: set the global variable `errno` to 0 (see `man -s3 errno`), then use `strtol()` to convert the diamond width to a long (see `man -s3 strtol`). Check for the following errors in the order they appear below. Note that some of the error strings will need `STR_WIDTH` as part of the error message.

<table>
<thead>
<tr>
<th>Error</th>
<th>How to Handle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error converting to long (errno was set by strtol())</td>
<td>Use <code>snprintf()</code> to build the error string using <code>STR_ERR_CONVERTING</code> (make sure the string is null-terminated). Call <code>perror()</code>, passing this string as a parameter. Skip to step 2.</td>
</tr>
<tr>
<td>width contains non-numerical characters (check <code>endptr</code>)</td>
<td>Print the <code>STR_ERR_NOT_INT</code> error message. Skip to step 2.</td>
</tr>
<tr>
<td>width is not within range (not within [WIDTH_MIN - WIDTH_MAX])</td>
<td>Print the <code>STR_ERR_NUM_RANGE</code> error message.</td>
</tr>
</tbody>
</table>
width is not odd (use isOdd()) | Print the STR_ERR_ODD error message. Continue to step 2.

2. **height**: set the global variable errno to 0 (see man -s3 errno), then use strtol() to convert the diamond height to a long (see man -s3 strtol). Check for the following errors in the order they appear below. Note that some of the error strings will need STR_HEIGHT as part of the error message.

<table>
<thead>
<tr>
<th>Error</th>
<th>How to Handle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error converting to long (errno was set by strtol())</td>
<td>Use snprintf() to build the error string using STR_ERR_CONVERTING (make sure the string is null-terminated). Call perror(), passing this string as a parameter. Skip to step 3.</td>
</tr>
<tr>
<td>height contains non-numerical characters (check endptr)</td>
<td>Print the STR_ERR_NOT_INT error message. Skip to step 3.</td>
</tr>
<tr>
<td>height is not within range (not within [HEIGHT_MIN - HEIGHT_MAX])</td>
<td>Print the STR_ERR_NUM_RANGE error message.</td>
</tr>
<tr>
<td>height is not odd (use isOdd())</td>
<td>Print the STR_ERR_ODD error message.</td>
</tr>
<tr>
<td>height &gt;= width (only if no errors with height and width)</td>
<td>Print the STR_ERR_HEIGHT_LESS_THAN WIDTH error message. Continue to step 3.</td>
</tr>
</tbody>
</table>

3. **border_char**: extract the first character from the border_char argument. Check for the following errors in the order they appear below. Note that both of the error strings will need STR_BORDER_CHAR as part of the string.

<table>
<thead>
<tr>
<th>Error</th>
<th>How to Handle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not a valid single character (hint: man -s3 strlen)</td>
<td>Print the STR_ERR_SINGLE_CHAR error message. Skip to 4.</td>
</tr>
<tr>
<td>char not within range (not within [ASCII_MIN - ASCII_MAX])</td>
<td>Print the STR_ERR_ASCII_RANGE error message. Continue to step 4.</td>
</tr>
</tbody>
</table>

4. **diamond_char**: extract the first character from the diamond_char argument. Check for the following errors in the order they appear below. Note that both of the error strings will need STR_DIAMOND_CHAR as part of the string.

<table>
<thead>
<tr>
<th>Error</th>
<th>How to Handle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not a valid single character (hint: man -s3 strlen)</td>
<td>Print the STR_ERR_SINGLE_CHAR error message. Skip to 5.</td>
</tr>
<tr>
<td>char not within range (not within [ASCII_MIN - ASCII_MAX])</td>
<td>Print the STR_ERR_ASCII_RANGE error message. Continue to step 5.</td>
</tr>
</tbody>
</table>

5. If any errors were encountered, print a newline and return EXIT_FAILURE. Otherwise, make sure border_char and diamond_char are different from each other. If they are not, print the STR_ERR_BORDER_DIAMOND_DIFF error message, then print a newline, and return EXIT_FAILURE.
6. If no errors were encountered, draw the diamond pattern simply by calling the `displayDiamond()` function, passing in the appropriate parameters.

**Return Value:** If errors were encountered, return `EXIT_FAILURE`. Otherwise, return `EXIT_SUCCESS`.

---

**Assembly Functions to be Written**

Listed below are the modules to be written in Assembly.

**checkRange.s**

```assembly
long checkRange( long value, long minRange, long maxRange );
```

This assembly module will check if the `value` is within `minRange` and `maxRange`, inclusive on both ends. This means if `value` is equal to `minRange`, it is within the range, and if `value` is equal to `maxRange`, it is also within the range.

For example:

<table>
<thead>
<tr>
<th>value</th>
<th>minRange</th>
<th>maxRange</th>
<th>Within range?</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>-4</td>
<td>error</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>true</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>5</td>
<td>false</td>
</tr>
</tbody>
</table>

**Reasons for error:**
- `minRange` is greater than `maxRange` → return -1

**Return Value:** If any errors occur, return -1. Otherwise, return 1 to represent true, 0 to represent false.

**rem.s**

```assembly
long rem( long dividend, long divisor );
```

This function calculates the remainder of the division between arguments. This function should act the same as the modulus operator (%) in C.

**Return Value:** The remainder as a 32-bit signed integer.

**isOdd.s**

```assembly
long isOdd( long value );
```

This assembly module checks if `value` is an odd number, using the `rem()` function.

**Return Value:** 0 if the `value` is not odd, non-0 if the `value` is odd.

**printChar.s**

```assembly
void printChar( char ch );
```

This assembly module prints the character argument to stdout (use `printf()`). This is very similar to the example assembly module given below. The main difference is that `printChar()` just prints a single character (so think about how that might affect the format string).
Return Value: None

Example: This example assembly module takes in a string that represents a class, and prints out a message saying that that class is your favorite class.

```
printFavoriteClass("CSE 30") -> "My favorite class is CSE 30"
```

```assembly
.cpu   cortex-a53
.fpu   neon-fp-armv8
.syntax unified

.section    .rodata
fmt: .asciz   "My favorite class is %s\n"

.global printFavoriteClass
.type   printFavoriteClass, %function

.text
.align  2

printFavoriteClass:
push  {fp, lr} @ Saves registers on the stack.
add   fp, sp, 4 @ Adjusts fp to be at bottom of current stack.
    @ Uses 4 because (#regs_saved - 1) * 4.
mov   r1, r0 @ Copy of formal parameter to the second arg of printf.
ldr   r0, =fmt @ Gets address of format string.
bl    printf @ Calls printf.
sub   sp, fp, 4
pop   {fp, pc} @ Returns from function.
```

displayDiamond.s

```
void displayDiamond( long width, long height, char border, char diamond );
```

This assembly module will perform the actual outputting of individual characters (via calls to `printChar()`) such that the diamond pattern is displayed with the user-supplied values. You might have to reuse a few registers in this function.

Here is the equivalent C version:

```
http://cseweb.ucsd.edu/~ricko/pa1/displayDiamond_template
```

All of the assembly constructs you will be using will have been covered in lecture and can be referenced in your notes. You are not limited to using the above algorithm, but part of the purpose of this programming assignment is to learn how to write looping/conditional constructs (branches), use the simple `bl` instruction to branch to subroutines with parameter passings, and perform simple arithmetic instructions (`add/sub,udiv`) in assembly.

We would encourage you to use the linked algorithm for these reasons, however we do not want to suppress creative thinking - alternative solutions are welcome. You must use the "preferred" style of coding loops, backwards branching logic, as detailed in class: set up an opposite logic branch to jump over the loop body and a positive logic branch to jump backwards to the loop body. Points will be taken off for not using backwards branching logic.

Return Value: None
Unit Testing

You are provided with only one basic unit test file for the Milestone function, `checkRange()`. This file only has minimal test cases and is only meant to give you an idea of how to write your own unit test files. **You must write unit test files for each of the Milestone functions, as well as add several of your own thorough test cases to all of the unit test files. You need to add as many unit tests as necessary to cover all possible use cases of each function. You will lose points if you don’t do this!** You are responsible for making sure you thoroughly test your functions. Make sure you think about boundary cases, special cases, general cases, extreme limits, error cases, etc. as appropriate for each function. The Makefile includes the rules for compiling these tests. Keep in mind that your unit tests will not build until all required files for that specific unit test have been written. These test files are not being collected for the Milestone and will only be collected for the final turnin (however, they should already be written by the time you turn in the Milestone because you should be using them to test your Milestone functions).

**Unit tests you need to complete:**
- `testcheckRange.c`
- `testrem.c`
- `testisOdd.c`

**To compile:**

```
$ make testcheckRange
```

**To run:**

```
$ ./testcheckRange
```

(Replace “testcheckRange” with the appropriate file names to compile and run the other unit tests)

**README File**

Your README file for this and all assignments should contain:

- High level description of what your program does.
- How to compile it (be more specific than: just typing “make” - i.e., what directory should you be in?, where should the source files be?, etc.).
- How to run it (give an example).
- An example of normal output and where that normal output goes (stdout or a file or ????).
- An example of abnormal/error output and where that error output goes (stderr usually).
- How you tested your program (what test values you used to test normal and error states) and showing your tests covered all parts of your code (test coverage). (Be more specific than diff'ing your output with the solution output - i.e., what are some specific test cases you tried?, what different types of cases did you test?, etc.)
- Anything else that you would want/need to communicate with someone who has not read the assignment write-up but may want to compile and run your program.
- Answers to questions (if there are any).

**Questions to Answer in the README**

Start gdb with your pa1 executable, then set a breakpoint at strtol:

```
break strtol
```

Note that you may see this message:

```
Function "strtol" not defined.
Make breakpoint pending on future shared library load? (y or [n])
```

In this case just hit y to continue.
Run the program (in gdb) with the following command line args:

```
run 9InchNails 5 b c
```

You should be at the entry point of the Std C Lib routine strtol called from main.

1. How do you print the value of the string that is the 1st arg in strtol? (The value should be "9InchNails")
2. How do you print the decimal value of the base that is the 3rd arg in strtol? (The value should be 10)
3. How do you print the hex value of &endptr that is the 2nd arg in strtol? (The value should be something like 0x7efff---- (a high stack address - will vary))

Go to the next high level source instruction in main. This should be the next C instruction in main after the function call to strtol. Type `next`. Again, if you encountered the "strtol" not defined error earlier, you may have to enter `next` a few times until the code returns to main.

4. How do you print the value returned by strtol? (The value should be 9) Show two ways:
   a. Using the name of the local variable you use to hold the return value
   b. Displaying the value in the register used to return the value

5. How do you print the character `endptr` is pointing to? (Should be the character 'I')

6. How do you print the entire null-terminated string `endptr` is pointing to? (Should be “InchNails")

7. How do you print the decimal value of the global variable `errno` at this point? (The value should be 0)

Continue the execution of your pa1 in gdb. Type `continue`.

You should hit another breakpoint and should see the error message `width(9Inchnails) must be an integer.`

Type `continue` again to end the program.

Run the program again (in gdb) with the following command line args:

```
run 99999999999999 5 b c
```

You should be at the entry point of the Std C Lib routine strtol called from main. Go to the next source instruction in main. Type `next`. Again, if you encountered the "strtol" not defined error earlier, you may have to enter `next` a few times until the code returns to main.

It should be the source-level instruction after the call to strtol passing in 99999999999999 to convert to an int. Print the decimal value of `errno` at this point. The value of `errno` should be 34 now which is the value of `ERANGE`. See the man page for `errno` and section 2 intro. You can continue or quit.

The following questions pertain to Git:

8. What is the Git command to show the current state of your working tree?
9. What is the Git command to discard any changes made to a file since its last commit?
10. What is the Git command to display the differences between the local version of a file and the version last committed?

Academic Integrity:

11. What was your process for completing this assignment with integrity?
Extra Credit

Overview
There are 5 points total for extra credit on this assignment. You will make a modified version of your Display Diamond program where the height of the top and bottom borders are equal to the number of digits in the width command line argument. Note that in order to do this extra credit, you must have a working version of your entire PA1 assignment.

Getting Started
Copy your regular displayDiamond function to start on the extra credit version

```bash
$ cp ~/pa1/displayDiamond.s ~/pa1/displayDiamondEC.s
```

Compiling
Once you have written your extra credit code, you can compile the extra credit program using the following command from within your pa1 directory.

```bash
$ make pa1EC
```

Sample Output

```
[cs30xyz@pi-cluster-001]:pa1$ ./pa1EC
Usage: ./pa1EC width height border_char diamond_char
    width       (must be odd within the range of [3 - 21])
    height      (must be odd within the range of [1 - 19])
                  (must be less than width)
    border_char (must be single char within the ASCII range [33 - 126])
                  (must be different than diamond_char)
    diamond_char (must be single char within the ASCII range [33 - 126])
                  (must be different than border_char)
```

```
[cs30xyz@pi-cluster-001]:pa1$ ./pa1EC 21 11 a -

```

Extra Credit Assembly Functions To Be Written
Note that while the names of files to be written are numOfDigitsEC.s and displayDiamondEC.s, the names of the functions should be numOfDigits() and displayDiamond(), respectively.
**numOfDigitsEC.s**

```c
long numOfDigits( long num, long base );
```

This function counts the number of digits in `num` in the given `base`. You need to check if `base` is within the limits of 2 and 36 (inclusive) using your `checkRange()` function.

If `num` is negative, do NOT convert it to positive before counting the number of digits. The reason behind this is that if `num` is the largest negative number, this number cannot be represented as a positive number. Therefore, in the case of negative numbers, we want to just leave them negative.

Note for unit testing: To type hex values, prefix the number with `0x`. To type octal values prefix them with `0`. C does not allow you to type values in binary, so you will need to convert binary to either decimal, octal, or hex.

For example:

<table>
<thead>
<tr>
<th>num</th>
<th>base</th>
<th>Number of digits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>any</td>
<td>1</td>
</tr>
<tr>
<td>-2693</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>0x7A2</td>
<td>16</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>4 ( (8 \equiv 1000_2) )</td>
</tr>
</tbody>
</table>

**Error conditions:**
- `base` is not within [2-36] inclusive → return -1

**Return Value:** If any errors occur, return -1. Otherwise, return the number of digits in `num` of the given `base`.

**displayDiamondEC.s**

```c
void displayDiamond( long width, long height, char border, char diamond );
```

This assembly module is very similar to your `displayDiamond` function, however the top and bottom border heights, instead of always being 1, will be the number of digits in the width (use `numOfDigits()` for this).

**Return Value:** None.

---

**Milestone Turn-in Instructions**

**Milestone Turn-in - due Wednesday night, January 24 @ 11:59 pm [15 points of Correctness Section]**

Before final and complete turnin of your assignment, you are required to turnin several modules for the Milestone check. Each module must pass all of our unit tests in order to receive full credit.

**Files required for the Milestone:**

- `checkRange.s`
- `rem.s`
- `isOdd.s`

A working Makefile with all the appropriate targets and any required header files must be turned in as well. All Makefile test cases for the `milestone` functions must compile successfully via the commands `make test***`. You do not need to turn in your README with the milestone.
In order for your files to be graded for the Milestone Check, you must use the milestone specific turnin script from the raspberry pi.

$ ~/../public/bin/cse30_palmilestone_turnin

To verify your turn-in:

$ ~/../public/bin/cse30verify palmilestone

**Final Turn-in Instructions**

**Final Turn-in - due Wednesday night, January 31 @ 11:59 pm**

Once you have checked your output, compiled, executed your code, finished your README file (see above), and double-checked your style, you are ready to turn it in. Use the following names *exactly* otherwise our Makefile will not find your files.

<table>
<thead>
<tr>
<th>Files required for the Final Turn-in:</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>checkRange.s</td>
<td>main.c</td>
<td>pal.h</td>
</tr>
<tr>
<td>displayDiamond.s</td>
<td>testcheckRange.c</td>
<td>palStrings.h</td>
</tr>
<tr>
<td>isOdd.s</td>
<td>testisOdd.c</td>
<td>test.h</td>
</tr>
<tr>
<td>printChar.s</td>
<td>testrem.c</td>
<td>Makefile</td>
</tr>
<tr>
<td>rem.s</td>
<td></td>
<td>README</td>
</tr>
</tbody>
</table>

**Extra Credit Files:**

displayDiamondEC.s          numOfDigitsEC.s

**How to Turn in an Assignment**

Before turning in, run `make clean` and then `make` to double check for any compiler errors/warnings. Then use the following turnin script from the raspberry pi to submit your full assignment before the due date as follows:

$ ~/../public/bin/cse30turnin pa1

To verify your turn-in:

$ ~/../public/bin/cse30verify pa1

Up until the due date, you can re-submit your assignment via the scripts above. It is your responsibility to check that your code was turned in successfully using the `cse30verify` command. If there is an issue turning in your code, you might not get a grade for the assignment!

Failure to follow the procedures outlined here will result in your assignment not being collected properly and will result in a loss of points. **Late assignments WILL NOT be accepted.**

If there is anything in these procedures which needs clarifying, please feel free to ask any tutor, the instructor, or post on the Piazza Discussion Board.