CS16, Spring 2010

lab06: ("lab six")
structs, pointers to structs, reading from files, arrays of structs

---

Goals for this lab

By the time you have completed this lab, you should be able to:

- Understand how to define a struct, and access the members of a struct
  - You should be able to access these members, directly, and through a pointer
- Understand how to pass a struct to a function
  - You should be able to pass either the struct itself (so that the function gets a copy of the original), or a pointer to the struct (so that the function can modify the original struct)—and understand the difference.

You'll also get some practice with reading from a file into an array of structs

  - We'll explore how to read lines from a file into an array of structs
  - We'll work with that array of structs to find various values

Prior Skills/Knowledge Needed

The list of skills needed for this lab is essentially the same as that for lab04, so we won't repeat that list in detail.

The only other pre-requisite for this lab is that you should already have introduced to the basic ideas of structs—in Spring 2010 we did this in homework assignments H13, H14, H17 and H18

Continue with your same pair partner from lab05—unless there is some reason not to...

- **If (and only if) you have a new partner for lab06—**one you didn't work with previously—complete a new version of W02, and post to the lab04/05/06 pair partner forum on Gauchospace. Indicate that this is a new pairing for lab06.

Also keep in mind:

- Pair programming is *required* for this lab, not optional—see lab02 for the reasons why.
- Keep the evaluation criteria in mind—you can find those listed in the intro to lab02

Step by Step Instructions

Step 0: Get together with your assigned lab partner and do the set up steps.

Choose a pilot and a navigator, and remember to switch often.

Check your PC—if you are in Cooper, there might be two mice attached. The navigator might be able to use the second mouse to help point things out, and feel a bit more connected to what is going on.

The driver should:

- Bring up a terminal window and create a ~/cs16/lab06 directory
- Copy files from my directory at `~pconrad/public_html/cs16/10S/labs/lab06/code/`

Step 1: Understanding what we are going to do:
Finishing three programs—and writing a fourth from scratch

The object of this lab is finish three incomplete programs that are in the test-driven development (TDD) style—and then to write a fourth one (completely from scratch.) Each of these works with the idea of structs.

Since this is our first time working with structs, the first program is very easy. The next two get progressively more challenging.

The fourth one is about the same level of difficulty as the third, but it is more challenging because you need to write the entire program from scratch (although you can borrow lots of code from the third program.)

By now, you are very familiar with the outline of what we do with a TDD style program—i.e. see the tests fail, then replace the stub with working code so the tests pass. So, I'll skip right to an explanation of the three programs:

Step 2: Working with areaBoxWithTests.c

This one is super easy—you just need to fill in the stub for `areaOfBox`. (More on how to do that in just a moment.)

However, let's first review how you have to compile areaBoxWithTests.c

Step 2a: How to compile areaBoxWithTests.c—and the other programs in this lab

If you try the way we've compiled most programs so far, by just typing:

```
make areaBoxWithTests
```

we run into a problem. Try it! You'll get something like this:

```
-bash-4.1$ make areaBoxWithTests
cc     areaBoxWithTests.c   -o areaBoxWithTests
/tmp/ccO5uGOC.o: In function `main':
areaBoxWithTests.c:(.text+0x15e): undefined reference to
`checkExpectDoubleWithFileAndLine'
areaBoxWithTests.c:(.text+0x1ed): undefined reference to
`checkExpectDoubleWithFileAndLine'
areaBoxWithTests.c:(.text+0x26d): undefined reference to
`checkExpectDoubleWithFileAndLine'
areaBoxWithTests.c:(.text+0x2ed): undefined reference to
`checkExpectDoubleWithFileAndLine'
collect2: ld returned 1 exit status
make: *** [areaBoxWithTests] Error 1
-bash-4.1$
```

An error like this is called a *linking error*—it arises when we have a *function call without a matching function definition*.

- Specifically, in this case, there is a function call inside `areaBoxWithTests.c` to the function `checkExpectDoubleWithFileAndLine`, but there is no function

definition for `checkExpectDoubleWithFileAndLine`inside that file—hence the error.

So how do we fix this?

- One way would be to put a definition for `checkExpectDoubleWithFileAndLine` inside areaBoxWithTests.c, but that's not what we are going to do in this case.
- Instead, as you may recall from lab05, when we want to reuse a collection of function definitions in more than one program, we put those function definitions in a separate file. In this case, we've collected a number of functions related to test-driven development and put them into a file called `tdd.c`. We've also put the function prototypes in a file called `tdd.h`.
- The function definition for `checkExpectDoubleWithFileAndLine` is in the file `tdd.c`—so, what we need to do is compile `areaBoxWithTests.c` and `tdd.c` together.

One way to combine these two is to compile with cc instead of with make, and list both files, like this:

```
-bash-4.1$ cc areaBoxWithTests.c tdd.c
-bash-4.1$
```

If we do this, the executable gets placed in a file called a.out, so to run the program we have to specify ./a.out:

```
-bash-4.1$ cc areaBoxWithTests.c tdd.c
-bash-4.1$ ./a.out
TEST FAILED: areaBoxWithTests.c line 55 areaOfBox(r) got -42.000000 expected
12.000000 (tolerance=0.000000):
TEST FAILED: areaBoxWithTests.c line 56 areaOfBox(s) got -42.000000 expected
12.000000 (tolerance=0.000000):
TEST FAILED: areaBoxWithTests.c line 57 areaOfBox(t) got -42.000000 expected
10.000000 (tolerance=0.000000):
TEST FAILED: areaBoxWithTests.c line 58 areaOfBox(u) got -42.000000 expected
60.000000 (tolerance=0.000000):
4 tests failed
-bash-4.1$
```

This works, but it has the disadvantage that we can only have one executable at a time. If we later want to compile another program in the same way, say initBoxWithTests.c, then the ./a.out file gets overwritten with that program:

```
-bash-4.1$ cc initBoxWithTests.c tdd.c
-bash-4.1$ ./a.out
```

```
TEST FAILED: initBoxWithTests.c line 121 distanceBetween(p1,p2) got -42.300000
expected 8.062300 (tolerance=0.001000):
TEST FAILED: initBoxWithTests.c line 123 distanceBetween(p2,p1) got -42.300000
expected 8.062300 (tolerance=0.001000):

...
TEST FAILED: initBoxWithTests.c line 268 boxesApproxEqual(b2,b2Expected,tol)
 :-(  15 tests FAILED!
-bash-4.1$
```

This is inconvenient. One way around this is to specify the name of the output file on the command line with the -o flag—traditionally we use the name of the .c file that contains the main, but without the .c on it, like this:

```
-bash-4.1$ cc areaBoxWithTests.c tdd.c -o areaBoxWithTests
-bash-4.1$ ./areaBoxWithTests
TEST FAILED: areaBoxWithTests.c line 55 areaOfBox(r) got -42.000000 expected
12.000000 (tolerance=0.000000):
TEST FAILED: areaBoxWithTests.c line 56 areaOfBox(s) got -42.000000 expected
12.000000 (tolerance=0.000000):
TEST FAILED: areaBoxWithTests.c line 57 areaOfBox(t) got -42.000000 expected
10.000000 (tolerance=0.000000):
TEST FAILED: areaBoxWithTests.c line 58 areaOfBox(u) got -42.000000 expected
60.000000 (tolerance=0.000000):
4 tests failed
-bash-4.1$
```

A danger of this method is that if we accidentally forget to leave off the .c on the name of the file that follows the -o, we can end up over-writing our .c code that we worked so hard to get right. This can be supremely frustrating!

So, there is a trick that can make our life a lot easier—a trick that allows us to just use the make command the way we've been doing all quarter, but with the extra tdd.c file coming along for the ride.

What we do is tell the make program that every time you run the C Compiler, add a bit of extra stuff. Here's how we do it:

```
-bash-4.1$ export CFLAGS="tdd.c"
-bash-4.1$
```

This looks like an assignment statement, and in a way it is. This is a *bash shell command* that assigning the value `"tdd.c"` to the *environment variable* called `CFLAGS`.

Environment variables are variables that are known to the bash shell, and can be accessed inside certain programs to modify their behavior. The make program, in particular, will take anything in CFLAGS and insert it into the compile command.

We can check the value of an environment variable by using the *environment variable dereference operator*—this isn't the star (*) like in C, but instead, the dollar sign ($). Here's how we check the value of `CFLAGS`:

```
-bash-4.1$ echo $CFLAGS
tdd.c
-bash-4.1$
```

Once CFLAGS has the value tdd.c, then we can just use the make command the way we always did, and tdd.c gets included every time——like this. Note that once CFLAGS has the correct definition, we can use the make command on every single one of the C programs included in this lab:

```
-bash-4.1$ make areaBoxWithTests
cc tdd.c    areaBoxWithTests.c   -o areaBoxWithTests
-bash-4.1$ make initPointWithTests
cc tdd.c    initPointWithTests.c   -o initPointWithTests
-bash-4.1$ make initBoxWithTests
cc tdd.c    initBoxWithTests.c   -o initBoxWithTests
-bash-4.1$ make readAirports
cc tdd.c    readAirports.c   -o readAirports
-bash-4.1$
```

And each can be run in the normal way:

```
-bash-4.1$ ./areaBoxWithTests
TEST FAILED: areaBoxWithTests.c line 55 areaOfBox(r) got -42.000000 expected
12.000000 (tolerance=0.000000):
TEST FAILED: areaBoxWithTests.c line 56 areaOfBox(s) got -42.000000 expected
12.000000 (tolerance=0.000000):
TEST FAILED: areaBoxWithTests.c line 57 areaOfBox(t) got -42.000000 expected
10.000000 (tolerance=0.000000):
TEST FAILED: areaBoxWithTests.c line 58 areaOfBox(u) got -42.000000 expected
60.000000 (tolerance=0.000000):
4 tests failed
-bash-4.1$ ./initPointWithTests
TEST FAILED: initPointWithTests.c line 84 distanceBetween(p1,p2) got -42.300000
expected 8.062300 (tolerance=0.001000):
TEST FAILED: initPointWithTests.c line 86 distanceBetween(p2,p1) got -42.300000
expected 8.062300 (tolerance=0.001000):
etc..
```

So, that is what we'll do throughout this lab to compile our code—we'll define CFLAGS to be "tdd.c" with:

```
-bash-4.1$ export CFLAGS="tdd.c"
-bash-4.1$
```

And then just use `make programName` the same way we've been doing for most of the C programs we've compiled so far.

Some extra hints related to the use of CFLAGS

- You have to run the `export CFLAGS="tdd.c"` command once per login session—it doesn't carry over from session to session.
- To cancel out the definition of an environment variable, use the unset command—e.g. `unset CFLAGS`

- If you want to get all the warnings the compiler can offer—which can be very helpful if you are debugging errors—you can use this command to add `-Wall` to the `CFLAGS` along with `tdd.c`
- `-bash-4.1$ export CFLAGS="-Wall tdd.c"`
- `-bash-4.1$`
-
- Spacing is crucial in the `export` command—don't put spaces on either side of the = sign.

  For example, this command does not work:
- `-bash-4.1$ export CFLAGS = "-Wall tdd.c"`
- `-bash: export: `=': not a valid identifier`
- `-bash: export: `-Wall tdd.c': not a valid identifier`
- `-bash-4.1$`
- If you are debugging a segmentation fault using the instructions from lab06a, you'll want to define your `CFLAGS` this way:
- `-bash-4.1$ export CFLAGS="-g -Wall tdd.c"`
- `-bash-4.1$`
-
- Finally, this isn't related to CFLAGS directly—but it is helpful for compiling with make—if you want to enable filename completion with the tab key after typing make, use this command:
- `-bash-4.1$ complete -r make`
- `-bash-4.1$`

This only has to be typed once per login shell.

Step 2b: Filling in the stub for the areaOfBox function

Now you know how to compile areaBoxWithTests.c—as a reminder, the once per login commands are:

```
-bash-4.1$ complete -r make
-bash-4.1$ export CFLAGS="tdd.c"
```

And what you type each time you want to compile and run is:

```
-bash-4.1$ make areaBoxWithTests
cc tdd.c    areaBoxWithTests.c   -o areaBoxWithTests
-bash-4.1$ ./areaBoxWithTests
TEST FAILED: areaBoxWithTests.c line 55 areaOfBox(r) got -42.000000 expected
12.000000 (tolerance=0.000000):
TEST FAILED: areaBoxWithTests.c line 56 areaOfBox(s) got -42.000000 expected
12.000000 (tolerance=0.000000):
TEST FAILED: areaBoxWithTests.c line 57 areaOfBox(t) got -42.000000 expected
10.000000 (tolerance=0.000000):
TEST FAILED: areaBoxWithTests.c line 58 areaOfBox(u) got -42.000000 expected
60.000000 (tolerance=0.000000):
4 tests failed
-bash-4.1$
```

So now, you just need to fill in the stub for the areaOfBox function so that the tests pass.

This involves understanding how to access the members of a struct—in this case, the height and width of a `struct Box`—and how to multiply them together and return the result.

If you were able to complete the last few homework assignments related to structs, you should have no problem—but if you need to, review the following for more info on working with structs:

- The handouts that went with H13, H14, H17 and H18
- Sections 7.1 and 7.2 of your CS16 textbook (Engineering Problem Solving with C, 3rd Edition)

Once you've fixed the areaOfBox function, and all the tests pass—take another minute to look through the entire program. Even though this part is easy, the next few steps get progressively longer and more complex, and understanding this easy program ***first*** may really help—especially if you can understand ***in its entirety.***

Step 3: Working with initPointWithTests.c

This program is a bit more challenging. There's also something special here: when we replace the stub for distance with the formula for distance, we have to use a square root.

This may require us to add `-lm` into our `CFLAGS` as well, like this:

```
-bash-4.1$ export CFLAGS="-lm tdd.c"

-bash-4.1$
```

(This can also be combined with `-Wall` as in **`export CFLAGS="-Wall -lm tdd.c"`**)

Your job on this program is to fill in three stubs:

- **`distanceBetween`** — distance between two points
  - since we already covered this in lecture, the correct answer is provided in a comment
  - this answer is also provided as an example of how to access members of a struct
  - so don't just blindly uncomment the correct answer—try to understand how it works
- **`pointsApproxEqual`**—whether two points are equal (within some tolerance)
  - the correct answer is provided in a comment here also is also provided for you as an example of how to reuse the distance formula
  - You'll get a chance to apply this idea yourself when you write `boxApproxEqual` and `circleApproxEqual` in a later step of this lab
- **`initPoint`**—initialize a point
  - Here there are two possible correct answers, both provided. Try to understand how they work.
  - Again, you'll be required in later steps to provide answers like this on your own, so take some time with each of these—and come back to them later if you are stumped on the corresponding functions for `initBox` and `initCircle`

You should also take a look at the way the `ASSERT_TRUE` function* is used and the places it is used in the program. You'll need to understand how `ASSERT_TRUE` works to write the later steps of this lab.
Essentially, `ASSERT_TRUE` takes just one argument—an expression that should evaluate to true (i.e. a non-zero integer) for the test case to pass.

*Actually, strictly speaking, the `ASSERT_TRUE` function is a macro, not a function—but this is not a detail we need to concern ourselves with at this point. If you are interested, you can read more about macros in section 4.8 of the Etter textbook.

When your test cases pass, move on to the next step.

Step 4: Working with initBoxWithTests.c

This program is similar to `initPointWithTests.c`, but here, you are filling in stubs and the answers are NOT provided.

For some of the stubs, you can copy/paste your answers from `initPointWithTests.c`.

But there are some new ones, where you and your pair partner need to come up with the code yourself.

Have fun!

Once you've gotten all the test cases to pass, you are ready to write a program from scratch.

Step 5: Writing initCircleWithTests.c from scratch

Now, write a program from scratch called initCircleWithTests.c

You are strongly encouraged to follow the TDD process: i.e. write tests and stubs first, get all the tests to fail, then fill in the stubs to make the tests pass.

- We aren't going to ask for a transcript to prove you did it this way—this is on the honor system.
- But, if you are asking the TA or the instructor for help, we'll give you frowny faces if you aren't following the TDD process, and big smiles if you are.

To receive full credit, your program should have the following elements.

1. **A definition for `struct Circle`** that has a `struct Point` member called `center`, and a double member called `radius`.
    - You may reuse the `struct Point` definition from the example programs above—in fact, not only may you do this, you should do this.

    - In addition to reusing struct Point, you should also reuse the functions that go along with `struct Point`, such as `pointsApproxEqual()`, `testPointsApproxEqual()`.
    - That also means reusing the functions that `pointsApproxEqual()`, `testPointsApproxEqual()` depend on, such as `distanceBetween()`.
    - For now, to simplify things, reusing the code may involve copying and pasting the code from the other files.
        - As we get more comfortable with how to share functions among different programs, we get away from this copying/pasting routine.
    - If/when you reuse any functions, be sure that the tests come along with those functions and are called from your main.
2. **A function called `circleApproxEquals()`** that takes three parameters: two `struct Circle` instances, and a double value for `tolerance`. It should return true if the circles are approximately equal, i.e. the center points are approximately equal, and the radii are approximately equal.
3. **A function `testCircleApproxEquals()`** that tests the `circleApproxEquals()`. Include at least three tests. This function should be called from your `main()`.
4. **A function called `initCircle()`** that initializes a circle. For parameters, it should take a `struct Circle *`, and doubles for the values of `centerX`, `centerY` and `radius`.
5. **A function called `testInitCircle()`** that tests `initCircle`. Include at least three tests. Call it from your main.
6. **A function called `areaCircle()`** that takes a `struct Circle` as its parameter, and returns the area of that circle.
    - For the value π, you can use `#include<math.h>`, and then use the predefined constant `M_PI`.
7. **A function called `testAreaCircle()`** that tests the `areaCircle` function. Include at least three tests.

Good luck!

Step 6: Turning to the readAirports.c program

We now turn to another topic—reading from a data file into an array of structs.

To start, look at the files 5airports.csv and airports.csv

Use the `more` command to look first at the file 5airports.cs (i.e. type more 5airports.csv at the Unix command prompt.)

Then use the more command to look at airports.csv

- For this one, you may need to know how to get out of `more`, since the file is pretty long!
- You can exit `more` by typing `q` (for quit)
- `CTRL/C` should also work.

These files contain latitude and longitude information for various airports.

- As the name suggests, `5airports.csv` contains information on only five airports
- airports.csv contains information on a large number of airports.

The first line in each file gives us some clues as to what the files contain:

```
Code,Lat,Lon,City,State
```

The file is in "CSV" format, i.e. "comma separated values".

We can get an idea of how long the airports.csv file is using the command `wc -l airports.csv`

This command allows us to see how many lines there are in the file ('wc stands for word count, but when we pass the -l flag it counts lines instead of words):

```
-bash-3.2$ wc -l airports.csv
1218 airports.csv
-bash-3.2$
```

We can see that if we want to make an array big enough to store all of these airports, it will need to have at least 1217 elements in it.

Step 7: Look at the file readAirports.c

Next, look through the file readAirports.c—open it up in the text editor (i.e. emacs).

This is a fairly long program, and some parts of it present new concepts that may be unfamiliar at first.

Here are a few of the new things you'll encounter. Try to find each one, and look over it with your pair programming partner. See if you can figure out what the code is doing.

1. The struct definition is in a separate header file: airport.h
    - Take a look at that file
    - There are a few new things in there—the `#ifndef` stuff, for example—but don't worry about that for now.
    - Just look at the `struct Airport` definition
    - We put it in a separate file because we might **reuse** it in other programs.
2. In the main, we declare a `FILE *` variable
    - This is a pointer to a file
    - It allows is to read data from a file
3. In the main, we declare an array of structs
    - We are going to read every line from an input file into this array
4. Inside the function `initAirportFromString`, we see calls to `strtok`
    - `strtok` stands for "string tokenize"—this means to pull it apart into its pieces
    - In this case, we are pulling apart a string such as `"LAX,33.93,118.4,LosAngeles,CA"` and turning it into separate strings, i.e. `"LAX"`, `"33"`, `"93"`, `"118.4"`, `"LosAngeles"`, and `"CA"`.
    - Notice that "," is a parameter to each call—this is because our data is separated by commas
5. Inside the function `initAirportFromString`, we see calls to strncpy
    - The `strncpy` function is used to copy a string
    - `strncpy` allows us to copy the **contents** of the string, not just the **pointer** value
    - The `n` in `strncpy` refers to the third argument, which is the maximum number of characters to be copied.
6. In the `main` program, we find a call to `strcmp`—the string compare function
    - This function allows us to see whether two strings are equal
    - If the strings are equal, strcmp returns a value of 0 (think: there is 0 difference between them.)
    - We use strcmp to check whether the name of the file passed in is the special value: "test"
    - If `argv[1]` is `"test"`, instead of opening a file, we run some tests, and quit.
7. The test function `runTests()` also uses `strcmp`

    - Notice that we use our test-driven development approach to check whether the function `initAirportFromString` did its job properly.
8. The main program has code to open a file with `fopen`, and read data from it
    - The `fgets()` function (file get string) is used to read lines from the file
    - We combine this with a while loop that tests `feof` (end of file) and `ferror` (file error) each time through the loop to see whether to continue into the loop—in case the fgets call didn't return any data.
    - Note that `feof` is only true AFTER you try to read and find nothing there.
    - We only continue into the loop if both `feof` and `ferror` are NOT true—i.e. our attempt to read was successful and we have some data to process!
9. After reading the data into the array, we use the array to look for the easternmost airport.
    - You get to add code for finding the westernmost, northernmost, and southernmost airports.

Once you've looked over all of that, you are ready to start coding yourself!

Step 8: What you need to do—adding some tests, and some new code

You job is to add a few tests, and then add some extra code at the end of the main function. (That extra code will also require you to add three functions.)

1. Compile the readAirports.c program.
2. Run it with the command: `./readAirports test`
    - You should see that five tests fail
    - Find the "stub" line in the function `initAirportFromString()` and comment it out
        - For now, don't remove it entirely—we'll do that at a later step
    - Recompile and run again with `./readAirports test`
    - All five tests should pass
3. Next, find the function runTests()
    - Inside this function, find the comment where it indicates you should add five tests
    - Add those tests

    - Then, bring the stub back in `initAirportFromString()`
        - `i.e.` the line that just says `return;` as the first line of the function
        - this prevents the function from doing any useful work
    - You should see 10 tests fail
    - Then take the stub back out—permanently—and all the tests should pass.
4. Finally, look at the end of the main function
    - There is a place there where the program prints out the easternmost airport
    - Your job is to add code to also print out the westernmost, southernmost and northernmost airport
    - In the process of doing that, you'll need to create three new functions:
        - indexOfAirportWithLargestLongitude
        - indexOfAirportWithSmallestLatitude
        - indexOfAirportWithLargestLatitude
    - Once you've written the new functions, and the new code in the main, you can test this code by running the program first with `./readAirports 5airports.csv`
        - You should see that San Diego is the easternmost airport—look at a map if you don't believe it!
        - Similarly, by simple inspection of the data, you should be able to determine which of the airports (SBA, SAN, SFO, LAX, SMF) is the southernmost, northernmost, and westernmost.
    - Then run your program with `./readAirports airports.csv` to see if the answers make sense (i.e. you'd expect the northernmost airport to be in Alaska, the easternmost to be in Maine, etc.)

When you've done all that, you are ready to submit!

A note of caution: When I've assigned this airport program in the past, some folks clearly either didn't look at their results carefully, or they were very unfamiliar with US Geography.

As an example: if your program suggests that the Northernmost airport in the United States is in Houston, Texas, then this is a big clue that something is wrong with your code.

Please check your results against common-sense notions about where things are—and what North, South, East and West actually mean in terms of latitude and longitude. If you aren't sure about these things, look them up (e.g. Google Maps, Wikipedia articles on latitude and longitude, etc.)

Step 9: Script and submit

Script your assignment just as in previous weeks. This week, instead of walking you through it step-by-step, I'm providing only the basic outline.

- By the time the course is over, you should be able to script properly without detailed instructions (that will be the expectation in future CS courses.)

So here's a basic outline. (You can always refer back to lab05 if you want to see the detailed step-by-step version).

- You always start by doing a **pwd** and and **ls** to show what directory you are in, and what files you have.
- The idea then is to take each of your programs one at a time, i.e. `areaBoxWithTests.c`, `initPointWithTests.c`, `initBoxWithTests.c`, `initCircleWithTests.c` and `readAirports.c` for each one:
  - Show that it compiles cleanly (you may need to remove the executable first if you are using make)
  - Show that it works properly by running it at least once
  - If needed, run it a few more times if needed, on different inputs, to show that it works properly.
    - This only applies if the program reads input from a file, the user, or the command line.
- When finished, type `exit` to end the script, and check the contents of the script file with `cat` or `more`.

Your script file should be located in your **~/cs16/lab06** directory and should be called **lab06.txt**

To submit use the turnin command: **turnin lab06@cs16 lab06**

- As a reminder: to submit your assignment, **you need to be in the ~/cs16 directory**—one level higher than the previous step (use `cd ..`)

Evaluation and Grading (300 pts total)

- **Mechanics (80 pts)**
  - 30 points: submission is on time and follows submission instructions (i.e. is done via turnin by the Due Date.)
  - 50 points:
    - successfully submitting a lab06 directory with five required C programs and a lab06.txt transcript file.
    - each C file contains a header comment on first line with name "cs16 lab06" and date and name(s) of students submitting the work.
    - all @@@ comments are removed in all files
    - extraneous irrelevant comments are removed—i.e. you shouldn't have comments about a "box" in a routine that process a "circle", leftover from copying/pasting
- **Programming Tasks—`areaBoxWithTests.c`, `initPointWithTests.c`, `initBoxWithTests.c` (50 pts)**
  - 10 points—`areaBoxWithTests.c`
    - Replace stub for `areaBox` with correct code
  - 10 points—`initPointWithTests.c`
    - Replace three stubs with correct code
  - 30 points—`initBoxWithTests.c`
    - 10 points: replace three stubs that are same as those in `initPointWithTests.c`
    - 10 points: replace stub of `boxesApproxEqual` with correct code
    - 10 points: replace stub of `initBox` with correct code
- **Programming Tasks—`initCircleWithTests.c` (70 points)**
  - 10 points for each of the items 1-7 listed in Step 5
- **Programming Tasks in `readAirports.c` (100 points)**
  - 25 points—adding tests in `runTests` as indicated by the comment
    - Add tests and remove the @@@ comments
    - The results of the tests should be added into the `failure` variable (use the `+=` operator)
    - The tests should fail when there is a stub in `initAirportFromString()`, and should pass when the stub is replaced with working code.
  - 25 points—code for printing western most point
    - This code should call a new function `indexOfAirportWithLargestLongitude()` to do its calculation
  - 25 points—similar code for printing northernmost point

        - This too should call a new function with an appropriate name
    - 25 points—similar code for printing southernmost point
        - You guessed it—this should call yet another function with an appropriate name

Due Date

Due Date: You should try to complete this assignment by the end of the discussion section in which it was assigned.

If you are unable to complete it by the end of your discussion section you may continue to work on it through the week—ideally before the start of next week's discussion section.

It will be accepted without late penalty until 5pm on Friday 05/14.

Late assignments will only be accepted (with 20 point penalty) through 5PM on Thursday 05/20—however you are strongly encouraged to complete it before the Midterm Exam E02 on Wednesday 05/19, as part of your studying and preparation for the exam.

After 5PM Thursday 05/20, a zero will be recorded, and the only option is to make up the points via extra credit.
