lab05 : Fun with shapes: Pointers

num	ready?	description	assigned	due
lab05	true	Fun with shapes: Pointers	Tue 11/12 09:00AM	Mon 11/18 11:59PM

Goals of this lab

The goal of this lab is to demonstrate how we can abstract “things” in the world (such as geometric objects) into program contructs. We also hope that you will get a lot more practice with using pointers, passing pointers to functions and using pointers along with structs. You will continue to use the TDD process to develop high quality and well-tested code in a systematic manner. Lastly, we would like you to delve deep and have fun! When you run into problems with your code, remember to use the skills we have been learning in class, such as code tracing and drawing pointer diagrams, to understand the dynamics of your program and how it is interacting with memory (for now the run-time stack).

Academic Honesty

All work submitted for this lab should be your own. If you are using any hints from a previous offering of this course that was posted publicly by a CS16 instructor, you must cite your source.

Step by Step Instructions

Step 1: Getting Ready

1. If you are working as a pair, go to github and create a git repo for this lab following the naming convention specified in previous labs. If you are working with a partner only one of you needs to create the repo.

2. If you are working with a partner and you are the one who created the github repo, add your partner as a collborator on the repo

3. Decide on initial navigator and driver.

4. Driver, log on to your CSIL account.

5. Open a terminal window and log into the correct machine.

6. Change into your CS 16 directory

Remember to push your work to github at the end of EVERY work session. That way, both partners always have access to the latest version of the code even if the code is being developed on one partner’s CoE account.

Step 2: Obtain the starter code

This step is similar to lab02, first clone your github repo in the ~/cs16/ directory. Then cd into your repo directory.

```
cd ~/cs16/lab05_gaucho_ally
```

Copy the code from your starter code directory to your local lab05 repo using the following command.

```
cp /cs/faculty/dimirza/cs16/labs/lab05/* ~/cs16/lab05_agaucho_alily/
```

Typing the list (ls) command should show you the following files in your current directory

```
[dimirza@csil-03 lab05_agaucho_alily]$ ls
areaOfBoxTest.cpp         initBoxTest.cpp           pointToStringTest.cpp  shapeFuncs.h  utility.cpp
areaOfBoxTest.cpp~        initPointTest.cpp         README.md              shapes.h      utility.h
boxesApproxEqualTest.cpp  Makefile                  #shapeFuncs.cpp#       tddFuncs.cpp
distanceBetweenTest.cpp   pointsApproxEqualTest.cpp  shapeFuncs.cpp         tddFuncs.h
[dimirza@csil-03 lab05_agaucho_alily]$
```

Step 3: Reviewing the Files and what your tasks are

Here is a brief description of each of the files and expected implementation. Note that the .h/.cpp files with the same name are a pair. All the function declarations should be placed in the .h files. The definition of the functions that contains your implementation should go in the corresponding .cpp file

- utility.h/cpp : Modify to implement any of your own functions that you will need to solve the problems of this lab
- tddFuncs.h/cpp : Functions that you may use to test your code
- shapes.h : Contains the declaration of two structs: Point and Box. These data structures will be used in other files e.g. shapeFuncs.h/cpp
- shapeFuncs.h/cpp : Functions to compute metrics on geometric constructs such as points and boxes (see shapes.h)
- *Test.cpp: Each of the files that end in Test.cpp contain code to test a particular function in shapeFuncs.cpp. For example distanceBetweenTest.cpp contains test code to test your implementation of the distanceBetween() function in shapeFuncs.cpp. Note that each Test.cpp file tests contains a main function, which means that each test file along with its dependent code is meant to be compiled into a separate executable. The provided Makefile makes sure that this is infact the case. The rationale behind this approach is that each function in shapeFuncs.cpp can be developed and tested independently as much as possible.

Here is a list of your tasks for this lab:

- Run make and see the given code being compiled.
- Run ./distanceBetweenTest and see it fail.
- Edit the distanceBetween function in shapeFuncs.cpp to replace with correct code.
- Run ./distanceBetweenTest and see it pass.
- Commit and push your code to github.

- Run ./pointsApproxEqualTest and see it pass.
- Look at the code in pointsApproxEqualTest.cpp and shapeFuncs.cpp and understand how it works; Notice how the pointsApproxEqual() function uses the distanceBetween() function that you just wrote and tested, rather than writing new code within pointsApproxEqual() that repeats the logic of distanceBetween(). The takeaway here is that you want to keep your code as DRY as possible (DRY==Don't Repeat Yourself). You also want to only reuse code that has already been tested. You'll need to understand pointsApproxEqual() to get ./boxesApproxEqual to pass.

- Run ./initPointTest and see it fail.
- Looking at the test code in initPointTest.cpp figure out what the initPoint function is supposed to do and add preconditions and postconditions as comments to the start of that function. See page 275 of the book for more information on writing pre and post conditions.
- Edit the initPoint function in shapeFuncs.cpp to replace the stub with correct code.
- Run ./initPointTest and see it pass.
- Now reason about why your code works. Do this by drawing a pointer diagram that shows the state of memory right before the initPoint function returns when it is called for the very first time by the test code. Your pointer diagram should show the value of member variables x and y of the struct object 'p1' in initPointTest.cpp as well as the relationship between 'p1' and the formal parameter 'p' of the function initPoint. You should also show the formal parameters xVal and yVal in memory and indicate whether or not they are colocated in memory with any other variables (such as x and y). Make the drawing on a piece of paper or as ascii art in a text file and upload it to gradescope with the filename: pointer-diagram-initPoint. The diagram will be graded manually by us.

- Run ./boxesApproxEqualTest and see it fail.
- Edit the boxesApproxEqual function in shapeFuncs.cpp to replace the stub with correct code. As you do, consider adding an approxEqual function that takes two double values into utility.h and utility.cpp, as this will make your coding job easier, and keep you code "DRYer". Also, consider reusing the pointsApproxEqual function in your boxesApproxEqual solution. Remember that the `&&` operator is the symbol for "logical and" in C++.
- Run ./boxesApproxEqualTest and see it pass.
- Reason about why your code worked, draw a diagram to show the relationship between the formal and actual parameters. You don't need to submit the diagram but you may be asked to draw such a diagram on an exam!
- Commit and push your code to github. This may be a good time to switch partners if you haven't done so already.

- Run ./initBoxTest and see it fail
- Edit the initBox function in shapeFuncs.cpp to replace with correct code. As you do, remember that you use -> to access members of a struct through a pointer, but simply . to access members of a struct directly. You may need both in your answer.
- Run ./initBoxTest and see it pass
- Commit and push your code to github.

- Run ./areaOfBoxTest and see it fail
- Edit the areaOfBox function in shapeFuncs.cpp to replace with correct code.
- Run ./areaOfBoxTest and see it pass
- Commit and push your code to github.

- Run ./pointToStringTest and see it it pass
- Copy pointToStringTest.cpp to boxToStringTest.cpp and make tests for the boxToString function. Look in shapeFuncs.cpp at the boxToString function stub for an example of the format you need for boxToString's return values. Make tests for different precisions, just like pointToString has.
- Add code to the Makefile so that boxToString runs. Just follow the model–adding code for boxToStringTest everywhere you see code for pointToStringTest
- Run make
- Commit and push your code to github.

- Run ./boxToStringTest and see the tests fail
- Fix the definition of boxToString in shapeFuncs.cpp
- See the test ./boxToStringTest pass
- Commit and push your code to github.

- YOU ARE READY TO CHECK YOUR WORK.

Step 4: Checking your work before submitting

When you are finished, you should be able to type `make clean` and then `make tests` and see the following output:

```
-bash-4.2$ make clean
/bin/rm -f distanceBetweenTest initPointTest pointsApproxEqualTest boxesApproxEqualTest initBoxTest areaOfBoxTest pointToStringTest *.o
-bash-4.2$ make tests
g++ -Wall -Wno-uninitialized   -c -o distanceBetweenTest.o distanceBetweenTest.cpp
g++ -Wall -Wno-uninitialized   -c -o tddFuncs.o tddFuncs.cpp
g++ -Wall -Wno-uninitialized   -c -o utility.o utility.cpp
g++ -Wall -Wno-uninitialized   -c -o shapeFuncs.o shapeFuncs.cpp
g++ -Wall -Wno-uninitialized  distanceBetweenTest.o tddFuncs.o utility.o shapeFuncs.o -o distanceBetweenTest
g++ -Wall -Wno-uninitialized   -c -o initPointTest.o initPointTest.cpp
g++ -Wall -Wno-uninitialized  initPointTest.o tddFuncs.o utility.o shapeFuncs.o -o initPointTest
g++ -Wall -Wno-uninitialized   -c -o pointsApproxEqualTest.o pointsApproxEqualTest.cpp
g++ -Wall -Wno-uninitialized  pointsApproxEqualTest.o tddFuncs.o utility.o shapeFuncs.o -o pointsApproxEqualTest
g++ -Wall -Wno-uninitialized   -c -o boxesApproxEqualTest.o boxesApproxEqualTest.cpp
g++ -Wall -Wno-uninitialized  boxesApproxEqualTest.o tddFuncs.o utility.o shapeFuncs.o -o boxesApproxEqualTest
g++ -Wall -Wno-uninitialized   -c -o initBoxTest.o initBoxTest.cpp
g++ -Wall -Wno-uninitialized  initBoxTest.o tddFuncs.o utility.o shapeFuncs.o -o initBoxTest
g++ -Wall -Wno-uninitialized   -c -o areaOfBoxTest.o areaOfBoxTest.cpp
g++ -Wall -Wno-uninitialized  areaOfBoxTest.o tddFuncs.o utility.o shapeFuncs.o -o areaOfBoxTest
g++ -Wall -Wno-uninitialized   -c -o pointToStringTest.o pointToStringTest.cpp
g++ -Wall -Wno-uninitialized  pointToStringTest.o tddFuncs.o utility.o shapeFuncs.o -o pointToStringTest
./distanceBetweenTest
PASSED: distanceBetween(p1,p2)
PASSED: distanceBetween(p2,p1)
PASSED: distanceBetween(p3,p4)
PASSED: distanceBetween(p4,p5)
PASSED: distanceBetween(p5,p3)
./initPointTest
PASSED: pointsApproxEqual(p1,p1Expected)
PASSED: pointsApproxEqual(p2,p2Expected)
PASSED: pointsApproxEqual(p3,p3Expected)
PASSED: pointsApproxEqual(p4,p4Expected)
./pointsApproxEqualTest
PASSED: pointsApproxEqual(p1,p1)
PASSED: pointsApproxEqual(p1,p2)
PASSED: assertFalse(pointsApproxEqual(p2,p1)
./boxesApproxEqualTest
PASSED: boxesApproxEqual(b0,b0)
PASSED: boxesApproxEqual(b1,b0)
PASSED: boxesApproxEqual(b0,b1)
PASSED: boxesApproxEqual(b0,b2)
PASSED: boxesApproxEqual(b0,b3)
PASSED: boxesApproxEqual(b0,b4)
PASSED: boxesApproxEqual(b5,b6)
PASSED: boxesApproxEqual(b6,b5)
./initBoxTest
PASSED: boxesApproxEqual(b1,b1Expected)
PASSED: boxesApproxEqual(b2,b2Expected)
PASSED: boxesApproxEqual(b1,b2)
./areaOfBoxTest
PASSED: areaOfBox(r)
PASSED: areaOfBox(s)
PASSED: areaOfBox(t)
PASSED: areaOfBox(u)
./pointToStringTest
PASSED: pointToString(p1)
PASSED: pointToString(p2)
PASSED: pointToString(p2,1)
PASSED: pointToString(p2,4)
PASSED: pointToString(p2,5)
-bash-4.2$
```

Plus, some output at the end with the output of your boxToStringTest

```
./boxToStringTest
PASSED: boxToString(b1,1)
PASSED: boxToString(b1,2)
PASSED: boxToString(b1,3)
PASSED: boxToString(b1,4)
PASSED: boxToString(b1,5)
PASSED: boxToString(b1,6)
-bash-4.2$
```

At that point, you are ready to try submitting on gradescope.

Step 5: Submitting via gradescope

Submit all the cpp files to lab05 assignment on gradescope. Then visit gradescope and check that you have a correct score. If you are working with a partner, make sure both of you join a team on gradescope, otherwise only one of you will get credit for the lab

- You must check that you have followed these style guidelines:

1. Indentation is neat, consistent and follows good practice (see below)
2. Variable name choice: variables should have sensible names. More on indentation: Your code should be indented neatly. Code that is inside braces should be indented, and code that is at the same “level” of nesting inside braces should be indented in a consistent way. Follow the examples from lecture, the sample code, and from the textbook.

- Your submission should be on-time. If you miss the deadline, you are subject to getting a zero

Commit and push the latest version of your code on github

Grading Rubric

Points from gradescope automatic grading:

Test Group	Test Name	Value
areaOfBoxTest	areaOfBoxTest	(30 pts)
boxToStringTest	boxToStringTest (requires you to add a new file, so more points)	(60 pts)
boxesApproxEqualTest	boxesApproxEqualTest	(30 pts)
distanceBetweenTest	expected output from distanceBetweenTest	(30 pts)
initBoxTest	initBoxTest	(30 pts)
initPointTest	initPointTest	(30 pts)
pointToStringTest	pointToStringTest (should pass in base code, so no points assigned)	(0 pts)
pointsApproxEqualTest	pointsApproxEqualTest output (should pass in base code, so no points)	(0 pts)

Style: Good choice of variable names, code indented in ways that are consistent, and in line with good C++ practice. Where applicable, common code is factored out into functions (added to utility.h and utility.cpp as needed).

This last point may or may not arise, but if it does, utility.h and utility.cpp is a place where functions needed in multiple files can be put—prototypes in utility.h and function definitions in utility.cpp.

You will note that the gradescope score is worth 250 points. The grade will ultimately normalized to be out of 100 points. This lab is worth exactly the same as all the other labs done so far (i.e. the 250 points here are equivalent to 100 points in other labs).

Step 6: Done!

Once your submission receives a score of 250/250, you are done with this assignment. Remember that we will check your code for appropriate comments, formatting, and the use of required code, as stated earlier, based on your github submission.

If you are in the Phelps lab or in CSIL, make sure to log out of the machine before you leave. Also, make sure to close all open programs before you log out. Some programs will not work next time if they are not closed. Remember to save all your open files before you close your text editor.

If you are logged in remotely, you can log out using the exit command:

```
$ exit
```