lab04 : Odds and primes: Fun with arrays and makefiles

num	ready?	description	assigned	due
lab04	true	Odds and primes: Fun with arrays and makefiles	Tue 11/05 09:00AM	Tue 11/12 11:59PM

CS16: Programming Assignment 04

Introduction -- Important: Read this!

The mentors will be looking for (and grading) your programming style, such as proper use of comments, tab indentation, good variable names, and overall block and function designs. So, it is not enough for your lab to pass gradescope tests! Please read the instructions herein **carefully**.

Pair programming

If working in a pair: Choose who will be the first driver and who will start as navigator, and then remember to switch (at least once) during the lab. But you should probably know the long-term goal too: each partner should participate in both roles in approximately equal parts over the course of the assignment. We realize it is not possible to equally split time in every lab perfectly, but it is worth trying, and it is possible to make up for unequal splits in future labs. We trust you will try to meet this goal. Thanks!

PLEASE MAKE SURE YOU TRADE CONTACT INFORMATION WITH YOUR LAB PARTNER! This means emails, phone numbers, online chat handles, or whatever is necessary to continue working together when you are working remotely (like, say, if one of you goes home for the weekend).

You should now have a familiarity with:

- Creating a git repo on github
- Cloning your github to your local machine
- Integrating git command-line tools into your workflow (*git add…*, *git commit..*, *git push …*)

Be sure to commit and push or work to github at the end of EVERY work session. That way, if your pair partner bails on you, you can continue working without them. :)

Step 1: Log on to CSIL and bring up a terminal window.

I hope I can safely assume that you have all gotten a CoE account. If your account is not working, get the attention of the instructor.

Log into your account to make sure it works. As a reminder to get to the terminal go to **Application** Menu, then **System Tools**, then **Terminal Window**.

In the steps below, and in most future labs, you will create files on your own account.

Step 2: Create a new repo, add your partner as collaborator and clone it to your local directory

In lab02, we have done the same thing. So if you don't know to how to do that, please refer to lab02 for details. The basic steps are as follows:

- Create a git repo on github following the correct naming convention, e.g., if your github username is jgaucho and your partner's is alily, your should name your repo lab04_agaucho_alily (usernames appear in alphabetical order). Don't forget to make it 'PRIVATE'.
- Add your partner as a collaborator for the repo.
- Go to your CS16 directory and clone the repo locally.

Step 3: Get the starter code from a local directory

Copy the skeleton to your local lab04 repo using the following command, **REMEMBER** to change the directory name in the commands below to your own directory's name, in this lab we simply use lab04_agaucho_alily as a example for your local git directory:

```
cp /cs/faculty/dimirza/cs16/labs/lab04/* ~/cs16/lab04_agaucho_alily/
```

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

```
[-bash-4.2]$ ls
arrayBoundsDemo.cpp    maxOfArray.cpp           sumOfArray.cpp
arrayFuncs.h           maxOfArrayErrorTest.cpp  sumOfArrayTest.cpp
arrayToString.cpp      maxOfArrayTest.cpp       tddFuncs.cpp
arrayToStringTest.cpp  minOfArray.cpp           tddFuncs.h
countEvens.cpp         minOfArrayErrorTest.cpp  utility.cpp
countEvensTest.cpp     minOfArrayTest.cpp       utility.h
countPrimes.cpp        README.md                utilityTest.cpp
countPrimesTest.cpp    sumOdds.cpp
Makefile               sumOddsTest.cpp
[-bash-4.2]$
```

Push the initial version of the code on github before making any changes by typing the following commands

```
git add .
git commit -m "Initial version"
git push origin master
```

Step 4: Reviewing Separate Compilation

The files in your directory this week use separate compilation, that is each program is not necessarily taking all of its code from a single .cpp source file.

In Lecture, we will introduce the idea of separate compilation, where your C++ program may be divided among multiple source files. The following web page explains more about separate compilation, dividing your program up among multiple C++ and .h files, and using a Makefile. I strongly encourage you to read over it briefly before you proceed with the lab: Separate Compilation and Makefiles

Step 5: Writing isOdd(), isEven() and isPrime()

Your first step is very simple to describe, but somewhat challenging. The challenge here is mostly C++ coding — we will not get into the details of the separate compilation until a bit later.

To get started, do the following steps:

Step 5a: make clean

In your working directory, type **ls** and make note of the different files therein: some are .cpp types, some are .o (short for "object file"), some are .h (short for "header file"), and others do not have extensions (they are binary executables). Now, type **make clean**. This command cleans out any .o files and executables from your directory

That should look like this:

```
-bash-4.2$ make clean
/bin/rm -f arrayToStringTest arrayBoundsDemo countEvensTest minOfArrayTest minOfArrayErrorTest
countPrimesTest maxOfArrayTest maxOfArrayErrorTest sumOddsTest sumOfArrayTest utilityTest *.o

-bash-4.2$
```

Take a look at the **Makefile** file to understand why this happened.

Step 5b: make utilityTest

This command makes the executable for a main program, defined in `utilityTest.cpp`, that tests the functions defined in `utility.cpp`. Recall that for functions defined in a file such as `utility.cpp` that has no **main()**, the function prototypes are defined in the file `utility.h` Look at the source code for both `utility.cpp` and `utility.h` to see what they contain. Recall that a "stub" is place-holder code that allows an incomplete function to compile. It is designed to fail all the tests, though for a boolean function, since there are only two possible values (true and false), any stub value you choose is going to pass at least some of the tests.

That should look like this:

```
-bash-4.2$ make utilityTest
g++ -Wall -Wno-uninitialized   -c -o utilityTest.o utilityTest.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  utilityTest.o tddFuncs.o utility.o -o utilityTest

-bash-4.2$
```

Step 5c: Run ./utilityTest

Next, type `./utilityTest`

This runs the `utilityTest` program that tests the three functions isOdd, isEven and isPrime. As we noted above, some of the tests will pass, even though the implementation of the three functions is totally bogus (hard coded to return false always).

Here is what that looks like (some output truncated)

```
-bash-4.2$ ./utilityTest
        FAILED: isEven(2)
                Expected: 1 Actual: 0
        PASSED: isEven(3)
        FAILED: isEven(4)
                Expected: 1 Actual: 0
        PASSED: isEven(55)

[...      Some output omitted here... ]

PASSED: isPrime(64507)
        FAILED: isPrime(69997)
                Expected: 1 Actual: 0
PASSED: isPrime(-55)
PASSED: isPrime(-80)
PASSED: isPrime(0)
PASSED: isPrime(1)

-bash-4.2$
```

Step 5d: Repeat: edit, compile, run, until all tests pass

Now do these steps, repeatedly, until all tests pass:

edit utility.cpp (e.g. emacs utility.cpp, or vim utility.cpp)
make utilityTest
run utilityTest (e.g. ./utilityTest)

Submit working versions of your code on github using the commands:

```
git add *.cpp *.h
git commit -m "implemented utility function - nameof function()"
git push origin master
```

You must only proceed with the rest of the lab once you have implemented all the utility functions and pass the provided test cases. This is because for other files that you will be editing later, you will NEED functions isOdd, isEven and isPrime. Once you get them working, you will be able to call them in other files and KNOW that they work properly. You will not have to repeat the function definition.

When all the tests for `utlityTest` pass, do a final push to github and move on to the next step.

If you are working with a pair partner, this is a good time to switch roles.

Step 6: Reviewing the rest of the files and what your tasks are

Now, let us look at the files you actually have in your directory, and what you need to do with them. You have the following .cpp files. This table indicates what you must do with each one to get full credit on this lab.

Filename	**Your task**	**Details**
arrayBoundsDemo.cpp	NOTHING TO CHANGE OR SUBMIT.	This is here as example code only. You are encouraged to run it, study it, and learn about how array bounds work in C++
arrayToString.cpp	NOTHING TO CHANGE OR SUBMIT.	This code is part of your solution, but you do not have to submit it - we will use our own version, which matches the one in your sample directory. This file just has utiltiy funcitons for printing arrays as strings.
arrayToStringTest.cpp	NOTHING TO CHANGE OR SUBMIT.	This code is part of your solution, but you do not have to submit it - we will use our own version, which matches the one in your sample directory. This file is an example of how to test cases to determine whether the output of `arrayToString` works correctly.
countEvens.cpp	REPLACE STUB WITH CORRECT CODE.	You must replace the code in this file with correct code that returns the number of even integers in each array passed in.
countEvensTest.cpp	NO MODIFICATIONS NEEDED	This tests the changes you made in *countEvens.cpp*.
countPrimes.cpp	REPLACE STUB WITH CORRECT CODE.	You must replace the code in this file with correct code that returns the number of prime integers in each array passed in. Treat negative numbers, 0 and 1 as “not prime”. You may want to add a definition of `isPrime()` to the *utility.cpp* file and a function prototype to *utility.h* so that you can call function `isPrime` in your solution.
countPrimesTest.cpp	NO MODIFICATIONS NEEDED	This tests the changes you made in *countEvens.cpp*.
maxOfArray.cpp	REPLACE STUB WITH CORRECT CODE.	You can look at *minOfArray.cpp* for hints. This one should be easy.
maxOfArrayErrorTest.cpp	REPLACE EMPTY MAIN WITH TESTS.	Insert code to call maxOfArray with zero length array. Use *minOfArrayErrorTest.cpp* as a model.
maxOfArrayTest.cpp	REPLACE EMPTY MAIN WITH TESTS.	Insert code to call `assertEqual` exactly seven times testing whether `maxOfArray` returns correct values. Use *minOfArrayTest.cpp* as a model. It must be exactly “seven” calls to `assertEquals` to pass the gradescope tests. You should call your arrays the same things that they are called in `minOfArrayTests`, and the lengths should be the same. So the messages you get out for passed tests should match the messages from `minOfArrayTests` except that the name of the function is `maxOfArray` instead of `minOfArray`. You MAY change the values in the arrays themselves, though, to make the tests better tests, if you need to. **(Note that just hard coding a program that prints “PASSED” seven times with the apprpriate messages is not sufficient to get credit–you need to really have actual tests. Any attempt to “game the system”, i.e. to get gradescope tests to pass without a bona-fide attempt to actually solve the problem will get zero credit.)**
minOfArray.cpp	NOTHING TO CHANGE.	This is a model of correct code that can serve as a hint for how to write *maxOfArray.cpp*
minOfArrayErrorTest.cpp	NOTHING TO CHANGE.	This is an model of correct code for how to test whether a function behaves as expected when given input that should print a message to cerr and exit the progrm.
minOfArrayTest.cpp	NOTHING TO CHANGE.	This is a model of how to do unit testing on a function that returns an integer.
sumOdds.cpp	REPLACE STUB WITH CORRECT CODE.	You must replace the code in this file with correct code that returns the number of sum of the odd integers in each array passed in. Negative odd integers count as odd integers.

Filename	Your task	Details
sumOddsTest.cpp	REPLACE EMPTY MAIN WITH TESTS	Insert code to call `assertEqual` exactly seven times testing whether `sumOdds` returns correct values. Use *sumOfArrayTest.cpp* as a model. It must be exactly "seven" calls to `assertEquals` to pass the gradescope tests. You should call your arrays the same things that they are called in `sumOfArrayTests`, and the lengths should be the same. So the messages you get out for passed tests should match the messages from `sumOfArrayTests` except that the name of the function tested is `sumOdds` instead of `sumOfArray`. You MAY change the values in the arrays themselves, though, to make the tests better tests, if you need to. **(Note that just hard coding a program that prints "PASSED" seven times with the apprpriate messages is not sufficient to get credit–you need to really have actual tests. Any attempt to "game the system", i.e. to get gradescope tests to pass without a bona-fide attempt to actually solve the problem will get zero credit.)**
sumOfArray.cpp	INCORRECT CODE FOR YOU TO FIX.	The sum is not initialized properly. So the tests should fail. Your job is to see that the tests fail, then fix the sum initialization so the tests pass. Should be easy.
sumOfArrayTest.cpp	NOTHING TO CHANGE.	This is a set of tests to verify whether `sumOfArray()` works correctly.
tddFuncs.cpp	NOTHING TO CHANGE.	These are two functions that can be used to test functions that return either int or string values.
utility.cpp	ADD FUNCTIONS HERE AS NEEDED.	If you need to write your own helper functions, e.g. `isPrime`, `isOdd`, `isEven`, to use in other files, here is where you can put those definitions.

Step 7: Actually Getting Started

I suggest you start by typing: `make`

You should see a lot of activity as programs are compiled. You then will have a lot of executables you can run. Here is a list. Try running each one and see what happens.

Note these are the programs listed under BINARIES in the Makefile.

file	Anything to do?	explanation
arrayToStringTest	no	Run this and all tests should pass. Nothing to do here.
arrayBoundsDemo	no	Run this, and look at the code. This is an opportunity to learn something about how we pass arrays to functions in C++, but there is nothing you have to turn in from this program for the lab. It is just here as an example for you to learn from.
countEvensTest	***YES***	Run this, and you will see all the tests fail. YOU NEED TO FIX THE `countEvens` function and then get all these tests to pass.
minOfArrayTest	no	Just run this and see the tests pass. You can use the .cpp file *minOfArrayTest.cpp* as a model for writing *maxOfArrayTest.cpp*
minOfArrayErrorTest	no	Just run this and see the output. It should be `ERROR: minOfArray called with size < 1` printed on cerr (the standard error output stream). The gradescope system will check this as one of the acceptance tests for this lab, and it will also check that `maxOfArrayErrorTest` does the same thing. You can use the .cpp file *minOfArrayErrorTest.cpp* as a model for writing *maxOfArrayErrorTest.cpp*
countPrimesTest	***YES***	Run this, and you will see all the tests fail. YOU NEED TO FIX THE `countPrimes` function and then get all these tests to pass.
maxOfArrayTest	***YES***	Run this, and you will see that initially there is no output. That is because the main is empty. YOU NEED TO REPLACE THIS MAIN with code that tests `maxOfArray`. Use `minOfArrayTest` as a model. Initially, just put in the tests, and keep `maxOfArray` returning the stub vaue -42. See all the tests fail. Then get `maxOfArray` to return the right values and see all the tests pass.
maxOfArrayErrorTest	***YES***	Run this, and you will see that initially there is no output. That is because the main() is empty. YOU NEED TO REPLACE THIS MAIN with code that tests `maxOfArray`. Use `minOfArrayTest` as a model. Initially, just put in the tests, and keep `maxOfArray` returning the stub vaue -42. See all the tests fail. Then get `maxOfArray` to return the right values and see all the tests pass.
sumOddsTest	***YES***	Run this, and you will see that initially there is no output. That is because the main() is empty. YOU NEED TO REPLACE THIS MAIN with code that tests `sumOdds`. Use `minOfArrayTest` as a model. Initially, just put in the tests, and keep `sumOdds` returning the stub vaue -42. See all the tests fail. Then get `sumOdds` to return the right values and see all the tests pass.
sumOfArrayTest	***YES***	Run `sumOfArrayTest` and you will see that all the tests fail. Getting them to pass is probably the easiest step in this lab. Just look at the `sumOfArray` function, which is almost correct - it just needs you to initialize sum correctly. Note that in C/C++ variables are NOT automatically initialized, and failing to initialize them does not always result in an error message or warning unless you specifically ask the compiler to tell you about those. For this lab, the Makefile deliberarly turns that warning OFF so that we have to catch that ourselves.

So, if you go through that list, and do all the things indicated, you are finished with the lab and ready to submit.

Step 8: Checking your work before submitting

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

```
-bash-4.2$ make tests
./arrayToStringTest
PASSED: arrayToString(fiveThrees,5)
PASSED: arrayToString(zeros,3)
PASSED: arrayToString(empty,0)
PASSED: arrayToString(primes,10)
PASSED: arrayToString(meaning,1)
PASSED: arrayToString(mix,10)
./countEvensTest
PASSED: countEvens(fiveThrees,5)
PASSED: countEvens(zeros,3)
PASSED: countEvens(fiveInts,5)
PASSED: countEvens(empty,0)
PASSED: countEvens(primes,10)
PASSED: countEvens(meaning,1)
PASSED: countEvens(mix,10)
./countPrimesTest
PASSED: countPrimes(fiveThrees,5)
PASSED: countPrimes(zeros,3)
PASSED: countPrimes(fiveInts,5)
PASSED: countPrimes(empty,0)
PASSED: countPrimes(primes,10)
PASSED: countPrimes(meaning,1)
PASSED: countPrimes(mix,10)
./maxOfArrayTest
PASSED: maxOfArray(fiveThrees,5)
PASSED: maxOfArray(zeros,3)
PASSED: maxOfArray(fiveInts,5)
PASSED: maxOfArray(fiveInts,2)
PASSED: maxOfArray(fiveInts,3)
PASSED: maxOfArray(meaning,1)
PASSED: maxOfArray(mix,10)
./minOfArrayTest
PASSED: minOfArray(fiveThrees,5)
PASSED: minOfArray(zeros,3)
PASSED: minOfArray(fiveInts,5)
PASSED: minOfArray(fiveInts,2)
PASSED: minOfArray(fiveInts,3)
PASSED: minOfArray(meaning,1)
PASSED: minOfArray(mix,10)
./sumOddsTest
PASSED: sumOdds(fiveThrees,5)
PASSED: sumOdds(zeros,3)
PASSED: sumOdds(fiveInts,5)
PASSED: sumOdds(fiveInts,3)
PASSED: sumOdds(fiveInts,2)
PASSED: sumOdds(meaning,1)
PASSED: sumOdds(mix,10)
./sumOfArrayTest
PASSED: sumOfArray(fiveThrees,5)
PASSED: sumOfArray(zeros,3)
PASSED: sumOfArray(fiveInts,5)
PASSED: sumOfArray(fiveInts,3)
PASSED: sumOfArray(fiveInts,2)
PASSED: sumOfArray(meaning,1)
PASSED: sumOfArray(mix,10)
./utilityTest
PASSED: isEven(2)
PASSED: isEven(3)
PASSED: isEven(4)
PASSED: isEven(55)
PASSED: isEven(-55)
PASSED: isEven(-80)
PASSED: isOdd(2)
PASSED: isOdd(3)
PASSED: isOdd(4)
PASSED: isOdd(55)
PASSED: isOdd(-55)
PASSED: isOdd(-80)
PASSED: isPrime(2)
PASSED: isPrime(3)
PASSED: isPrime(4)
PASSED: isPrime(55)
PASSED: isPrime(859)
PASSED: isPrime(861)
PASSED: isPrime(863)
PASSED: isPrime(1337)
PASSED: isPrime(1373)
PASSED: isPrime(64507)
PASSED: isPrime(69997)
PASSED: isPrime(-55)
PASSED: isPrime(-80)
PASSED: isPrime(0)
PASSED: isPrime(1)

-bash-4.2$
```

And, you should be able to type `make errorTests` and see the following output:

```
-bash-4.2$ make errorTests
./minOfArrayErrorTest
ERROR: minOfArray called with size < 1
make: [errorTests] Error 1 (ignored)
./maxOfArrayErrorTest
ERROR: maxOfArray called with size < 1
make: [errorTests] Error 1 (ignored)

-bash-4.2$
```

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

Step 9: Turn in your code on gradescope

- Navigate to your ~/cs16/lab04_agaucho_alily directory, the one containing your code for this week's lab.

```
-bash-4.2$ cd ~/cs16/lab04_agaucho_alily
```

- Use the `ls` command to list your files and to be sure that you have all .cpp files (that you received as starter files) in your directory. It is ok if there are other files (*.txt, *.o, etc.) along with the executables. You only have to submit *.cpp files

```
-bash-4.2$ cd ~/cs16/lab04_agaucho_alily
-bash-4.2$ ls
arrayBoundsDemo.cpp    maxOfArray.cpp           sumOfArray.cpp
arrayFuncs.h           maxOfArrayErrorTest.cpp  sumOfArrayTest.cpp
arrayToString.cpp      maxOfArrayTest.cpp       tddFuncs.cpp
arrayToStringTest.cpp  minOfArray.cpp           tddFuncs.h
countEvens.cpp         minOfArrayErrorTest.cpp  utility.cpp
countEvensTest.cpp     minOfArrayTest.cpp       utility.h
countPrimes.cpp        README.md                utilityTest
countPrimesTest.cpp    sumOdds.cpp              utilityTest.cpp
Makefile               sumOddsTest.cpp
-bash-4.2$
```

Submit all the cpp files to lab04 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

Grading Rubric

Points from automated gradescope. system tests

Passed Tests

Test Group	Test Name	Value
countEvens	countEvensTest	25 pts
countPrimes	countPrimesTest	25 pts
maxOfArray	maxOfArrayTest	25 pts
maxOfArrayErrorTest	maxOfArrayErrorTest	25 pts
sumOdds	sumOddsTest	25 pts
sumOfArray	sumOfArrayTest	25ts
utilityTest	utilityTest	50pts

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 200 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 200 points here are equivalent to 100 points in other labs).

Step 10: Done!

Once your submission receives a score of 200/200, 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
```