Assignment 00: Practice Class Workflow

COSC 2336: Data Structures and Algorithms

Summer 2021

Objectives

- Setup and explore your VSCode DevBox Environment
- Learn some basic Git commands and workflow
- Practice submitting an example class project
- Make sure clas GitHub accounts are configured for git pushes and pull requests.
- Make sure VSCode development IDE is configured properly for C++ projects
- Learn about unit testing framework used for class asignments.

Description

The purpose of this practice assignment is to ensure that you have your development environment properly set up for the class assignments, and to learn about and practice our class assignment workflow.

There are several basic concepts and tools you need to learn about in this practice, so that you are ready for the actual class assignments. The tools we will be using for this class include:

- Visual Code studio IDE for code editing, building and debugging
- Basic Git usage, and using GitHub repositories for commiting work and pull requests.
- Unit test frameworks, we use the Catch2 unit test framework for our C++ coding assignments.

This practice assignment will walk you through the basic procedures and setup of assignments for our class assignments. Videos of the instructor doing all of the tasks of this practice can be found at the bottom of this readme, as well as additional suggested sources for learning the basics about the tools we will be using.

Pre-Setup and Configuration

Before performing this practice assignment workflow, you should have the following tasks already completed.

DevBox Configuration

1. You need to have your working class DevBox installed and be able to access the VSCode IDE from your system for development of class assignments. Go to the following URL and follow the instructions to setup and install your class DevBox. https://github.com/tamuc-class/cosc2336-devbox There is a video link on the README that walks you through installing and setting up the development environment.
2. You will need to create a GitHub account if you do not already have one. Go to the following URL and create a GitHub account to use. You may want to keep using this account in the future, so use your most useful e-mail address (doesn't necessarily need to be your TAMUC e-mail address). Also select a good username, and consider configuring your GitHub bio, icon and other properties. https://github.com/
3. A ssh public/private key will be created for you to use in your class DevBox. Open up the file name `/home/vagrant/.ssh/id_ed25519.pub`. You need to add this public key to your GitHub account. Login to GitHub, navigate to `settings -> SSH and GPG keys`. Create a `New SSH key` and copy/paste the text from the public key file into your GitHub key. Give the key a meaningful name, for example `COSC 2336 DevBox Public Git Key`.

4. There is one VSCode extension that you will need to install by hand in your class DevBox VSCode environment. From your class DevBox VSCode open up your `Extensions`. Click on the `...` at the top of the extensions and slect `Install from VSIX`. There should be a vile named `cpptools-linux.vsix` in your home directory. Select this extension to install it. This extension adds in basic C++ IDE features like intellisense, build problem parsing and notifications, context sensitive completions, etc.

Copy assg00 Repository on GitHub

Once you have completed these tasks, you are ready to begin working on the practice assignment 00. For all of our class assignments, you will first need to clone a repository I will give you from GitHub classrooms. I need to give you an invitation like. The following is the invitation link to use:

- Assg 00 (Summer 2021): https://classroom.github.com/a/dURspxBr

By following that link, you will be taken to GitHub, where you will be asked to accept the assignment. If this is the first assignment you are accepting, you need to associate your GitHub account with the class ID that identifies you as a member of this class. Please let me know if you do not see yourself listed as a student here. Once you associate your account and accept the assignment, a copy of the assignment will be created for you in your GitHub account.

Clone Repository using SSH to Class DevBox

The copied repository will be named something like `assg00-username` where it appends your GitHub username to the repository. Once you have this repository copied in GitHub, you need to clone the repository to your DevBox locally so that you can work on it.

In your class DevBox, open the Git section. Select the `Clone Repository` button to clone your repository. You need to copy/paste your GitHub ssh url into your DevBox to clone the repository. You find your GitHub url by opening your repository on GitHub, navigating to `Code`, and pulling down the `Code` download button. Make sure that you clone using ssh, not https. If you clone using ssh you will not be able to push your changes back to the repository. If you get an error message that the ssh clone failed because you do not have permission to access the repository, then you have probably not yet successfully created your ssh public key in step 3 above in your GitHub account yet.

You should clone this and all assignments into your `~/sync/assg` directory. This will allow you to be able to access the files on your host system as well.

Checkout the Feedback branch

Once the repository is cloned, you should make sure that you switch to the `feedback` branch of your repository, which should already be created for you. In your DevBox VSCode IDE, select the Git section. At the top of the Git source control section, open the `...` additional actions and select `Checkout to...`. Select the `origins/feedback` branch. Once you have checked out this branch, you should see that `feedback` is your working branch on the bottom status bar of your VSCode window.

Practice Assignment 00 Ready

At this point you should be ready to begin working on the practice Assignment 00. For all of the assignments for this class you will follow the same steps in the previous section when you get started.

1. Copy the assignment repository on GitHub using the provided invitation link.
2. Clone the repository using SSH url to your local class DevBox
3. Checkout the `origins/feedback` branch of to your local working repository.

At this point for each assignment, you will be ready to begin reading the assignment description and working on the assignment tasks.

Overview and Setup

For all assignment, it will be assumed that you have a working class DevBox, and that you have copied and cloned the assignment before beginning to work on it. We will start each assignment with a description of the general setup of the assignment, and an overview of the assignment tasks.

For this practice assignment you have been given the following files (among many others):

File Name	Description
`src/assg00-tests.cpp`	Unit tests for the two functions you are to write.
`include/assg00-functions.hpp`	Header file for function prototypes you are to add.
`src/assg00-functions.cpp`	Implementation file for the functions you are to write for this assignment.

All assignments for this class are in the form of multi-file projects. All source files will be in the `src` subdirectory, and all header files which are needed so you can share and include code will be in the `include` directory.

For this and all class assignments, we will be using a testing framework. The GitHub repository has been set up to perform a build and test action automatically whenever you push a commit of your code to the GitHub repository. This commit task will run the same tests that you have in your assignment, and that you need to get working for the assignment.

For this practice assignment, you will need to add codes into two files, named `assg00-functions.hpp` and `assg00-functions.cpp`. In addition, the unit tests you need to pass for the practice assignment are given to you in `assg00-tests.cpp`. Some or all of these tests will be commented out to begin with. You will uncomment the tests and write the code to get the tests to pass as the main work for each assignment.

The code you are initially given should always be compilable and runnable. You should check this out, and try to make sure that for every small change or addition you make, that you code still compiles and runs the given tests.

Before starting on the tasks below, confirm that your starting code is compiling and running. From your VSCode DevBox, open the `assg00` folder if it is not currently open. Then perform a `make clean / make all / make run`. You can use VSCode command palate to perform the `Run Task` command, and slelect these tasks from the command palette. Keyboard shortcuts should already be assigned to these common tasks, so you could do them as follows

- `ctrl-shift-1` make clean

```
> Executing task: make clean <

rm -rf ./test ./debug *.o *.gch
rm -rf output html latex
rm -rf obj
```

- `ctrl-shift-2` or `ctrl-shift-b` make build

```
> Executing task: make all <

mkdir -p obj
g++ -Wall -Werror -pedantic -g -Iinclude -c src/tests-main.cpp -o obj/tests-main.o
g++ -Wall -Werror -pedantic -g -Iinclude -c src/assg00-tests.cpp -o obj/assg00-tests.o
g++ -Wall -Werror -pedantic -g -Iinclude -c src/assg00-functions.cpp -o obj/assg00-functions.o
g++ -Wall -Werror -pedantic -g  obj/tests-main.o  obj/assg00-tests.o  obj/assg00-functions.o -o test
g++ -Wall -Werror -pedantic -g -Iinclude -c src/assg00-main.cpp -o obj/assg00-main.o
g++ -Wall -Werror -pedantic -g  obj/assg00-main.o  obj/assg00-functions.o -o debug
```

- `ctrl-shift-3` make run

```
> Executing task: make run <

././test --use-colour yes
===============================================================================
No tests ran
```

The project should compile cleanly with no erors when you do the `make build`, and the tests should run from `make run`, though all tests are currently commented out, so there are not actual tests available to run yet.

Assignment Tasks

Now we will walk through the typical tasks and workflow you will perform for the class assignments. In this section you will normally have a list of tasks you should complete. You should complete the tasks in the given order here, and do not move on to the next task until you have successfully completed the previous one. I will encourage you, and maybe even require you, to commit and push your changes after completing each task (at a minimum).

For this practice assignment, the goal is to create two functions named `isPrime()` and `findPrimes()`. We will start with the first function.

Task 1: Implement `isPrime()` Function

Open up the `assg00-tests.cpp` file. In this file you will find two `TEST_CASE` sections that are both currently commented out. The first of these has a set of checks to test the `isPrime()` function. Uncomment just this first `TEST_CASE`. After uncommenting this test case, perform a `make build` of your code. You should find that the build will fail with the following message:

```
$ make all

g++ -Wall -Werror -pedantic -g -Iinclude -c src/assg00-tests.cpp -o obj/assg00-tests.o
In file included from src/assg00-tests.cpp:22:
src/assg00-tests.cpp: In function ‘void ____C_A_T_C_H____T_E_S_T____0()’:
src/assg00-tests.cpp:33:10: error: ‘isPrime’ was not declared in this scope
   33 |    CHECK(isPrime(1) );
      |          ^~~~~~~
In file included from src/assg00-tests.cpp:22:
src/assg00-tests.cpp:33:10: error: ‘isPrime’ was not declared in this scope
   33 |    CHECK(isPrime(1) );
      |          ^~~~~~~
make: *** [include/Makefile.inc:51: obj/assg00-tests.o] Error 1
```

What is happenning here is that these tests are trying to test the implementation of an `isPrime()` function, but there is no implementation yet of this function. You are going to implement this function. Lets start by creating a stub for the function, so that we can get our code to compile and run correctly.

We need to put a function prototype for this function into the `assg00-functions.hpp` header file. A function prototype can be used in a header file so that we can include the header, and the compiler will then know what the signature of the function is and can thus figure out how to compile code that wants to call this function.

In `assg00-functions.hpp` at the appropriate place, add the following function signature:

```
bool isPrime(int value);
```

The `isPrime()` function signature here simply says that there is some function named `isPrime` that will be implemented in the implementation file. This function takes a single `int` parameter (called `value`) as input to the function. The function returns a `bool` boolean result (`true` or `false`).

Once you have added this function prototype to the header file, try to build your program again.

```
$ make all

g++ -Wall -Werror -pedantic -g -Iinclude -c src/assg00-tests.cpp -o obj/assg00-tests.o
g++ -Wall -Werror -pedantic -g  obj/tests-main.o  obj/assg00-tests.o  obj/assg00-functions.o -o test
/usr/bin/ld: obj/assg00-tests.o: in function `____C_A_T_C_H____T_E_S_T____0()':
/home/dash/repos/cosc2336-github-classroom/assg00/src/assg00-tests.cpp:33: undefined reference to `isPrime(
/usr/bin/ld: /home/dash/repos/cosc2336-github-classroom/assg00/src/assg00-tests.cpp:34: undefined reference
/usr/bin/ld: /home/dash/repos/cosc2336-github-classroom/assg00/src/assg00-tests.cpp:35: undefined reference
/usr/bin/ld: /home/dash/repos/cosc2336-github-classroom/assg00/src/assg00-tests.cpp:38: undefined reference
```

```
/usr/bin/ld: /home/dash/repos/cosc2336-github-classroom/assg00/src/assg00-tests.cpp:41: undefined reference
/usr/bin/ld: obj/assg00-tests.o:/home/dash/repos/cosc2336-github-classroom/assg00/src/assg00-tests.cpp:42:
collect2: error: ld returned 1 exit status
make: *** [include/Makefile.inc:42: test] Error 1
```

The output of the compilation is a bit complex here, but you should take a moment to look at it. The `assg00-tests.cpp` file actually compiles successfully now. This is because it includes the signature you added from `assg00-functions.hpp` header file, and so the `c++` compiler knows how to compile the test into an object file named `assg00-tests.o`. The compilation then continues. It tries to build the test executable, but we get a series of errors here when it tries to link together the test executable. All of the errors are because of an `undefined reference to isPrime(int)`. The function signature you added tells the compiler how another file can use the function. But we need to provide an actual **implementation** of the function somewhere, so that when the compiler tries to link together the code, there is an implementation that can be called to compute the `isPrime()` function results.

So lets add in the implementation. In the `assg00-functions.cpp` file, add in the following stub implementation.

```
/** isPrime
 * Determine if a given (positive) integer value >= 1 is prime.
 * Prime numbers are numbers that are divisible only by 1 and
 * themselves.  Thus in this implementation, we use a brute force
 * method and test to see if any number from 2 up to the value-1 is
 * a divisor of the number.  If we find such a divisor, then the
 * number is not prime.  If we fail to find such a divisor, then the
 * number is prime.
 *
 * @param value The value to be tested to see if prime or not.
 *
 * @returns bool Returns true if the value is prime, false if it
 *   is not.
 */
bool isPrime(int value)
{
  return true;
}
```

Make sure that you add in the implementation directly under the documentation for the `isPrime()` function, which should already be available to you in the starting code for this practice assignment.

Notice that this is not really a correct implementation. It is a stub function, it always just answers `true` whenever it is asked if any `value` is a prime number or not.

Compile your program once again. You should find that it will not correctly compile and link together your `test` executable, if you have correctly entered the stub function and function prototype.

```
$ make all

g++ -Wall -Werror -pedantic -g -Iinclude -c src/assg00-functions.cpp -o obj/assg00-functions.o
g++ -Wall -Werror -pedantic -g  obj/tests-main.o  obj/assg00-tests.o  obj/assg00-functions.o -o test
g++ -Wall -Werror -pedantic -g  obj/assg00-main.o  obj/assg00-functions.o -o debug
```

Always make sure your program is in a compilable state. If it can compile the `test` executable, we can run the unit tests, and see how well our implementation is working so far.

```
$ make run
././test --use-colour yes


test is a Catch v2.12.2 host application.
Run with -? for options
```

```
<isPrime()> function tests

src/assg00-tests.cpp:30


src/assg00-tests.cpp:38: FAILED:
  CHECK_FALSE( isPrime(4) )
with expansion:
  !true

src/assg00-tests.cpp:42: FAILED:
  CHECK_FALSE( isPrime(6) )
with expansion:
  !true

src/assg00-tests.cpp:44: FAILED:
  CHECK_FALSE( isPrime(8) )
with expansion:
  !true

src/assg00-tests.cpp:45: FAILED:
  CHECK_FALSE( isPrime(9) )
with expansion:
  !true

src/assg00-tests.cpp:46: FAILED:
  CHECK_FALSE( isPrime(10) )
with expansion:
  !true

src/assg00-tests.cpp:48: FAILED:
  CHECK_FALSE( isPrime(12) )
with expansion:
  !true

src/assg00-tests.cpp:50: FAILED:
  CHECK_FALSE( isPrime(14) )
with expansion:
  !true

src/assg00-tests.cpp:51: FAILED:
  CHECK_FALSE( isPrime(15) )
with expansion:
  !true

src/assg00-tests.cpp:52: FAILED:
  CHECK_FALSE( isPrime(16) )
with expansion:
  !true

src/assg00-tests.cpp:54: FAILED:
  CHECK_FALSE( isPrime(18) )
with expansion:
  !true

src/assg00-tests.cpp:56: FAILED:
  CHECK_FALSE( isPrime(20) )
```

```
with expansion:
  !true

src/assg00-tests.cpp:63: FAILED:
  CHECK_FALSE( isPrime(101007) )
with expansion:
  !true

src/assg00-tests.cpp:64: FAILED:
  CHECK_FALSE( isPrime(101831) )
with expansion:
  !true

===============================================================================
test cases:  1 |  0 passed |  1 failed
assertions: 24 | 11 passed | 13 failed

make: *** [include/Makefile.inc:60: run] Error 13
```

Here you should see, that while the tests run, many of the tests are failing, as can be seen from the output of running the tests. If you look closely back at the `assg00-tests.cpp` file, you will see that not all of the tests are failing. It actually passes the first 3 tests that check if 1, 2 and 3 are prime. This should make sense, because your stub function always returns `true`, so if the number being tested is prime, then the test passes.

Commit Changes, Make Pull Request

Part of the workflow of assignments is that you should commit your work once you get tasks (or milestones within tasks) completed and working. Uncommenting the first set of unit tests, and getting the code to compile and run the tests is a good milestone. Lets add these changes to the `feedback` branch, make a commit, and push the commit to your repository.

Open the Git section in VSCode. You will see that the 3 files you modified are now listed as changed in your repository. Perform an `add all` changes to add all of these 3 file changes to the commit you are creating. Once the files are added to the commit, write a commit message and then select the checkmark to commit these changes to your current `feedback` branch. Try and always use good commit messages. Read the following: https://git-scm.com/book/en/v2/Distributed-Git-Contributing-to-a-Project

Here is an example following the guidelines for this commit

```
Task 1 isPrime() compiling and running tests

Partial implementation of Task 1 to write the isPrime() function.
Code is compiling and running a stub function.  Stub function
always returns true, so passes all tests where tester is expecting
the answer to be true for a given value.
```

Once your changes are committed, a new commit version is created. However, this commit is only local to your DevBox. You need to push your commit in the `feedback` branch to your remote repository, so that the instructor can see and evaluate your work.

At the bottom of your VSCode window is some status information about your repository. Next to the `feedback` branch indication, should be a status indicator. It will be indicating that you now have 1 commit on the `feedback` branch that is ahead of the remote repository. If you push this indicator, your commit will be pushed to the `feedback` branch on your GitHub repository.

Once you have pushed your comit changes, we can now create a pull request. You will need to create a pull request for each assignment, and have at least 1 commit (though you should have more) that will be part of the pull request in the `feedback` branch. The following actions need to be performed on your GitHub account. For each project, the `feedback` branch should be created for you initially already. In addition, an initial pull request will also have been created. However, you should go ahead and close this pull request, as we need to open a new one for your commits.

After closing this pull request, navigate on GitHub to find the changes that the most recent commit on your `feedback` branch have made. This is one location in GitHub that you can create a pull request from. Create the pull request for the changes on this first commit you made.

At this point you should see that you have 1 open pull request. If you go to this request, you will see that the build status for this commit failed. If you look in there, you will see that the commit does build, but it fails to pass all of the tests for the `isPrime()` and also for the other function you will write next.

If you have questions or need help, you can create your pull request and add comments onto the pull request. The instructor will be looking at the pull requests students are creating for assignments, and giving feedback and answering question on the pull request mechanism in GitHub.

Complete isPrime() Task 1

At this point lets create a correct implementation of the `isPrime()` function that will pass the unit tests. Modify your `isPrime()` function to look like the following.

```
/** isPrime
 * Determine if a given (positive) integer value >= 1 is prime.
 * Prime numbers are numbers that are divisible only by 1 and
 * themselves.  Thus in this implementation, we use a brute force
 * method and test to see if any number from 2 up to the value-1 is
 * a divisor of the number.  If we find such a divisor, then the
 * number is not prime.  If we fail to find such a divisor, then the
 * number is prime.
 *
 * @param value The value to be tested to see if prime or not.
 *
 * @returns bool Returns true if the value is prime, false if it
 *   is not.
 */
bool isPrime(int value)
{
  // check all possible divisors of the value from 2 up to value-1 to
  // see if we can find a valid divisor
  for (int divisor = 2; divisor < value - 1; divisor++)
  {
    // if divisor divides the value, remainder is 0, and thus we found
    // a divisor
    if (value % divisor == 0)
    {
      // if there is a divisor other than 1 and value, then the answer is
      // false, it is not prime
      return false;
    }
  }

  // but if we check all divisors and don't find one, then the answer
  // is true, it is a prime
  return true;
}
```

This function performs a brute force search of all possible numbers that might be divisors of the `value`. A number is prime if it hs no divisors other than 1 and the `value`. So if any other `divisor` is found, the function can return an answer of `false` indicating that the number is not prime. But if we check all divisors and don't find any, then the answer is `true`, the number is prime.

Now try compiling and running your tests again. You should see that all of the tests now pass.

```
$ make
```

```
mkdir -p obj
g++ -Wall -Werror -pedantic -g -Iinclude -c src/tests-main.cpp -o obj/tests-main.o
g++ -Wall -Werror -pedantic -g -Iinclude -c src/assg00-tests.cpp -o obj/assg00-tests.o
g++ -Wall -Werror -pedantic -g -Iinclude -c src/assg00-functions.cpp -o obj/assg00-functions.o
g++ -Wall -Werror -pedantic -g  obj/tests-main.o  obj/assg00-tests.o  obj/assg00-functions.o -o test
g++ -Wall -Werror -pedantic -g -Iinclude -c src/assg00-main.cpp -o obj/assg00-main.o
g++ -Wall -Werror -pedantic -g  obj/assg00-main.o  obj/assg00-functions.o -o debug

$ make run
././test --use-colour yes
===============================================================================
All tests passed (24 assertions in 1 test case)
```

You should find that all of the (uncommented) tests of `isPrime()` now pass. If they do, create a new commit of your changes (should only be changes in the `assg00-functions.cpp` file this time), and push your changes. Make sure you provide an appropriate commit message.

Go and look at your pull request again on GitHub. You will see that this second commit has been added to the pull request. Also you should notice that the test status is still failing. If you look at the status details, you should be able to discover why. While your commit should build, and should pass the `test isPrime` tests, it will not be passing the `test findPrimes` tests yet.

Task 2: Implement findPrimes() function

Lets complete this assignment. There is still a second function that you need to write and pass the tests. As with the first task, start by uncommenting the second `TEST_CASE` set of tests in `assg00-tests.cpp`. You should try building your code now. You will of course see that the build is now failing, because the tests want to run a function named `findPrimes()` but you haven't written it yet.

As before, lets start by doing the minimal work to get the project back to a compilable state. Add a function prototype for the function again into `assg00-functions.hpp` header file:

```
int findPrimes(int start, int end, bool displayOnCout = true);
```

Once you have done this try building again. You should find that the tests can compile, but again the test executable will not build because we havn't given an implementation for `findPrimes()` yet.

And as before, lets just add a stub implementation so we can get things building again. The `findPrimes()` function is supposed to be returning the number of primes it finds. Lets just start by returning 9 (which is what the first test in our test cases is expecting).

```
/** findPrimes
 * Find primes in a range of values from start to end (inclusive).
 * This function returns a count of the number of primes found within
 * the range.  As a side effect, the primes that are found in the range
 * are displayed on standard output.
 *
 * @param start The start of the range to search.  This value is inclusive,
 *   we test the start value of the range to see if it is a prime or not.
 * @param end The end of the range to search.  This value is inclusive,
 *   we test the end value of the range to see if it is a prime or not.
 * @param displayOnCout A parameter controlling whether or not primes found
 *   in the range are displayed on the standard cout stream or not.  This
 *   parameter defaults to true, found primes will be displayed when found.
 *
 * @returns int Returns the count of the number of primes we find in the
 *   asked for range.
 */
int findPrimes(int start, int end, bool displayOnCout = true)
{
  // stub answer, return what first test expects so we can get things compiling
  return 9;
}
```

Now if you rebuild, the program should compile. If you run your tests, you should see that the tests run. All of the tests in the first test case should still be passing. But it will be failing some of the tests in the second test case.

For practice, you should create a new commit and commit it to your local repository again. Don't forget to craft an appropriate commit message. Then push this commit to your feedback branch. Then once again check your pull request. This third commit should now be a part of the pull request. Notice also that the build status should still be failing. But if you look carefully at the build task results, you will see that the build and `isPrime()` tests pass, and the `findPrimes()` tests are now running, they just are not all passing yet.

Complete findPrimes() Task 2

We are almost done with this practice assignment. Lets complete the `findPrimes()` function so that it passes all of the tests. This function is supposed to search all primes in the given range, and return the count of the number of primes it finds within the range. Modify your implementation of `findPrimes()` with the following code.

```
/** findPrimes
 * Find primes in a range of values from start to end (inclusive).
 * This function returns a count of the number of primes found within
 * the range.  As a side effect, the primes that are found in the range
 * are displayed on standard output.
 *
 * @param start The start of the range to search.  This value is inclusive,
 *   we test the start value of the range to see if it is a prime or not.
 * @param end The end of the range to search.  This value is inclusive,
 *   we test the end value of the range to see if it is a prime or not.
 * @param displayOnCout A parameter controlling whether or not primes found
 *   in the range are displayed on the standard cout stream or not.  This
 *   parameter defaults to true, found primes will be displayed when found.
 *
 * @returns int Returns the count of the number of primes we find in the
 *   asked for range.
 */
int findPrimes(int start, int end, bool displayOnCout = true)
{
  int primeCount = 0;  // count primes we see in range to return

  if (displayOnCout)
  {
    cout << "List of primes in range "
         << start << " to " << end << ": ";
  }

  // search the range of values from start to end
  for (int value = start; value <= end; value++)
  {
    // whenever we find a prime list it out to standard output
    if (isPrime(value))
    {
      if (displayOnCout)
      {
        cout << value << ", ";
      }
      primeCount++;
    }
  }
  if (displayOnCout)
  {
    cout << endl;
    cout << "Count of primes: " << primeCount << endl;
  }

  // return the count of the number of primes we found in the range
  return primeCount;
}
```

This function has some additional logic to display some status of the search. Try modifying a test to pass in `true` for the third parameter to see how this works. We may use this in class later to discuss running the debugger on your code.

Once you add in the above function, try compiling and running your tests. They should all be passing now for both test cases. Since you are done with the assignment, it is usually a good idea to do one final build from scratch to be sure. Try doing a `make clean`, and then rebuild and run the tests. If all tests are uncommented, and all of them are passing, then you are in good shape at this point.

To finish the assignment, you should create a final commit of these changes in `assg00-functions.cpp` to implement your `findPrimes()` function. Make sure to give a good and appropriate commit message. Push the changes to your GitHub repository. When you are finished, it is a good idea then to leave a final comment in your pull request stating that you think you are finished and ready to have the work evaluated. The instructor make give feedback during your developement, or at the end. You should notice that your final commit is now passing all of the build tasks on GitHub. This is a good sign for the assignment. Though passing all of the tests may not mean you have completed all of the work successfully yet. For example, at times you may need to implement some functions or code in a specific way, and if you do no do this, even though you may be passing the tests, you may not be given full credit unless you correct your implementation to follow the instructions. Likewise, you may be asked to follow particular style or formatting guidelines of your code. For example, you will be required to provide function documentation for all of your functions, like the comments before the 2 functions you were given here.

Assignment Submission

For this class, the submission process is to correctly create a pull request with changes committed and pushed to your copied repository for grading and evaluation. For the assignments, you may not be able to complete all tasks and have all of the tests successfully finishing. This is ok. However, you should endeavor to have as many of the tasks completed before the deadline for the assignment. Also, try and make sure that you only push commits that are building and able to run the tests. In this practice assignment, 50 points out of 100 were assigned to correctly building. In general, a commit will get a 0 grade if it is not building and running the tests. So make sure before you push a change it at least builds. If you incorrectly push a bad build, there are ways to revert or remove changes from the commit to get back to a state that is building, so you can start again from that point.

Program Style

At some point you will be required to follow class style and formatting guidelines. The VSCode environment has been set up to try and format your code for some of these guidelines automatically to conform to class style requirements. But not all style issues can be enforced by the IDE/Editor. The instructor may give you feedback in your pull comments and/or create issues for you for the assignment that you need to address and fix. You should address those if asked, and push a new commit that fixes the issue (or ask for clarification if you don't understand the request). In general the following style/formatting issues will be required for programs for this class:

1. All programs must be properly indented. All indentation must be consistent and lined up correctly. Class style requires 2 spaces with no embedded tabs for all code indentation levels. The editor style checker should properly indent your code when you save it, but if not you may need to check or correct this if code is misaligned or not properly indented.
2. Variable and function names must use `camelCaseNameingNotation`. All variable and function names must begin with a lowercase letter. Do not use underscores between words in the variable or function name. Often function names will be given to you, but you will need to create variables, and maybe some functions, that conform to the naming conventions.

   - Global constants should be used instead of magic numbers. Global constants are identified useing `ALL_CAPS_UNDERLINE_NAMING`.
   - User defined types, such as classes, structures and enumerated types should use camel case notation, but should begin with an initial upper case letter, thus `MyUserDefinedClass`.
3. You are required to use meaningful variable and function names. Choosing good names for code items is an important skill. The code examples and starting code tries to give examples of good and meaningful names. In general, do not use abbreviations. Single variable names should be avoided, except maybe for generic loop index variables `i`, `j`, etc. Make your code readable, think of it as writing a document to communicate with other developers (and with your instructor who will be evaluating your code).
4. There are certain whitespace requirements. In general there should usually never be more than 1 blank line in a row in your code. Likewise there should usually not be more than 1 blank space on a line. There should be 1 blank space before and after all binary operators like `+`, `*`, `=`, `or`.
5. Function documentation is required for all regular functions and all class member functions. You need to follow the correctly formatted Doxygen function documentation format. We will use function documentation generation, and you should be sure your documentation can be built without emitting warnings or errors. Likewise all files should have a file header documentation at the top. You should edit the file header of files where you add in new code (not simply uncommenting existing code). Make sure the information has your correct name, dates, and other information.
6. Practice using proper Git commit messages. You should refer to issues and tasks correctly in commit messages.