Assignment Example: Example Workflow for Class Assignments to Setup Git, GitHub and VSCode Dev Containers

CSci 430: Introduction to Operating Systems

Objectives

- Setup and explore your VSCode Dev Container Environment
- Learn some basic Git commands and workflow
- Practice submitting an example class project
- Make sure class 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 and test runner used for class assignments.

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 committing work and pull requests.
- Unit test frameworks, we use the Catch2 unit test framework for our C++ coding assignments.
- Using Dev Containers in VSCode to have a common set of development tools.

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

You need to complete the steps in the DevelopmentEnvironmentSetup.md file before continuing with this practice assignment. All assignments assume that you have the following configurations completed in order to work on the assignments.

1. You have created a GitHub account to use for this class.
2. You have set up your GitHub ssh key so that you can clone repositories that you own and have write access so you can commit and push your changes back to your repositories.
3. You have installed Microsoft VSCode IDE/Editor on your system.
4. You have installed and learned how to use the Remote Containers extension in your VSCode IDE, so that you can clone your repositories, use the provided build tools and extensions, and can push your work back to your repositories for grading.

If you have a working VSCode IDE, and have set up the Dev Containers, then you can proceed with this and all assignments for the class.

When your development environment is configured and ready, before each assignment you will perform the following steps to begin working on the assignment.

Accept the Assignment to create your repository on GitHub

For all assignments you will be given a link that will allow you to accept the GitHub classroom assignment. Accepting the assignment will make a copy of the assignment code in your GitHub account from the assignment template. Once you accept the assignment, you will have a GitHub repository that you can clone and push your work back into for grading.

By following assignment invitation 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 select your student ID, you will need to create a team name. Teams of 2 or 3 students may be allowed for the class. Ask the instructor before starting assignment 01 about forming teams. There are additional work that needs to be done to collaborate successfully using git if you form a team, so be aware of that.

Clone Repository using SSH to a VSCode Dev Container

Your new repository on GitHub will be named something like `assg00-username` where it appends your GitHub username or group name to the repository. Once you have this repository created in GitHub, you need to clone the repository to a VSCode Dev Container.

Startup the VSCode IDE, and open up the "Remote Explorer" on the left hand side. The "Remote Explorer" section will not be present unless you have correctly installed the Remote Containers extension in your VSCode IDE, as described in the more detailed environment detailed instructions.

In the "Remote Explorer", if you hover over the CONTAINERS item, there will be a "+" mark allowing yout to create a new container. Select the "+" to create a new container, then select "Clone Repository in Container Volume...". This will allow you to clone your GitHub repository into a suitable Dev Container with all development tools set up that you need for the assignments. You should copy and paste the repository ssh url when asked here for the repository url, so that you can have write access to push back your work to your GitHub repository.

TODO: determine the best procedure to clone repository into a remote container.

Check that Initial Project Files Compile and Run Tests

Before starting to implement the assignment tasks, confirm that your starting code is compiling and running. From your VSCode Dev Container, open the `assg00` folder if it is not currently open. Then perform a `make clean / make all / make tests`. You can use VSCode command palette to perform the `Run Task` command, and select 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-c` make clean

```
> Executing task: make clean <

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

- `ctrl-shift-b` make build

```
> Executing task: make all <

mkdir -p obj
g++ -Wall -Werror -pedantic -g -Iinclude -I../assg-base-0.3/include -c src/test-primes.cpp -o obj/test-prim
g++ -Wall -Werror -pedantic -g -Iinclude -I../assg-base-0.3/include -c src/primes.cpp -o obj/primes.o
g++ -Wall -Werror -pedantic -g  obj/test-primes.o  obj/primes.o ../assg-base-0.3/obj/catch2-main.o -o test
g++ -Wall -Werror -pedantic -g -Iinclude -I../assg-base-0.3/include -c src/main.cpp -o obj/main.o
g++ -Wall -Werror -pedantic -g  obj/main.o  obj/primes.o -o debug
```

- `ctrl-shift-t` make tests

```
> Executing task: make tests <
```

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

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

Practice Assignment Example Ready

At this point you should be ready to begin working on the practice 'Assignment Example'. For all of the assignments for this class you will follow the same steps in the previous section(s) when you get started on a new assignment.

1. Accept the assignment invitation from GitHub classroom to create the assignment repository in your GitHub acount.
2. Clone the repository to a VSCode Dev Container using the SSH URL of your GitHub repository for the assignment.
3. Confirm that the project builds and runs, though no tests may be be defined or run initially. If the project does not build on the first checkout, please inform the instructor.
4. You should create the issue for Task 1 and/or for all tasks for the assignment now before beginning the first task. On your GitHub account, go to issues, and create them from the issue templates for the assignment. Also make sure you link the issue(s) with the `Feedback` pull request.

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 VSCode IDE with remote Dev Containers working, and that you have accepted and cloned the assignment to a Dev Container in your VSCode IDE. 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.
`src/assg00-sim.cpp`	The `main()` function for the assignment command line simulation.
`include/primes.hpp`	Header file for function prototypes you are to add for this assignment.
`src/primes.cpp`	Implementation file for the functions you are to write for this assignment.

All assignments for this class are 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` subdirectory.

For this and all class assignments, we will be using a unit 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 local repository, and that you need to get working for the assignment.

For this practice assignment, you will need to add codes into two files, named `primes.hpp` and `primes.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 for the assignments. 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. This is known as incremental development.

NOTE: Practice incremental program development. Always make sure your code is compilable and you can run the unit tests for the assignment. Make small changes, and recompile and run the tests after every small change. If you make a change that causes the compilation to fail, revert or fix the change immediately, do not add code to a project

that is currently not running and compiling. Do not write more than 2 or 3 lines of code at a time without trying to compile and run your project.

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 current one. You will be required to, at a minimum, push 1 commit for each defined task of the assignment to your GitHub classroom repository. You can certainly make more commits, and are most likely to need to do so as you will inevitably make some mistakes and push incomplete work or work with problems. This is fine, and actually normal and expected for people doing actual work on projects using a source code revision control system like git. But at a minimum, you must make at least 1 commit for each task, and that commit must implement the task fully and pass the tests for that task.

NOTE: All assignments for this class, when you accept them from GitHub classroom, will create a `Feedback` pull request for you. This pull request is tracking all commits to the `main` branch to be pulled into this `Feedback` pull request. All work you do should be pushed to the `main` branch of the repository. The `Feedback` pull request will automatically gather all commits to the `main` branch. You can communicate with me, and I will give back comments on the `Feedback` pull request of your progress for the assignment. You should not close or merge the `Feedback` pull request. When I am satisfied you have 100% completed an assignment, I will merge and close the `Feedback` pull request for you. This is an indication that you have completed all parts of an assignment. If the `Feedback` pull request is still open, it means I have not yet accepted it fully, and there may be comments or requirements from me for additional tasks or items you need to fix before your work is considered 100% complete.

For this practice assignment, the goal is to create two functions named `isPrime()` and `findPrimes()`. Then finally these functions are used in a command line simulation tool. We will start with the first function.

Task 1: Implement `isPrime()` Function

To begin with, each task should have an issue created for it in your GitHub repository. Got to your GitHub repository and create a new issue. You should find that a template is available for each task for the assignment. You should create the Task 1 issue now, and if you prefer, you can go ahead and create the issues for all of the tasks before you begin working on them at this point after copying your sandbox repository.

When you have the issue for Task 1 created, go to the `Feedback` pull request and link this issue with the `Feedback` pull request. The issue gives a TLDR description of what needs to be done for that task of the assignment. Once you get the hang of things, you may be able to work on and complete the tasks from the issue descriptions alone.

Once you have the Task 1 issue created, open up the `assg00-tests.cpp` file. In this file you will find two `TEST_CASE` sections that are both currently undefined. The first of these has a set of checks to test the `isPrime()` function. At the top of test files you will find lines like

```
// change these to #define when ready to begin working on each task,
// once defined they will enable the tests declared below for each task
#undef task1
#undef task2
```

We want to begin working on Task 1, so we need to define task 1 so that the test cases and unit tests for Task 1 are defined and compiled. Modify the line like this to define the task 1 tests:

```
#define task1
```

After defining the task1 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:41:9: error: ‘isPrime’ was not declared in this scope
```

```
   41 |   CHECK(isPrime(1));
      |         ^~~~~~
In file included from src/assg00-tests.cpp:22:
src/assg00-tests.cpp:41:9: error: ‘isPrime’ was not declared in this scope
   41 |   CHECK(isPrime(1));
      |         ^~~~~~
make: *** [include/Makefile.inc:63: obj/assg00-tests.o] Error 1
```

What is happening 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 `primes.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. All header files in your assignments can be found in the `include` subdirectory of your repository directory.

In `primes.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`). **NOTE** don't forget the semicolon `;` at the end of the line, it is easy to miss. For a function prototype, we don't provide a body between curly braces `{` and `}` that implements the function, we simply end the line with a semicolon `;`.

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

```
$ make all

++ -Wall -Werror -pedantic -g -Iinclude -c src/assg00-tests.cpp -o obj/assg00-tests.o
g++ -Wall -Werror -pedantic -g -Iinclude -c src/primes.cpp -o obj/primes.o
g++ -Wall -Werror -pedantic -g  obj/assg00-tests.o  obj/primes.o obj/catch2-main.o  -o test
/usr/bin/ld: obj/assg00-tests.o: in function `C_A_T_C_H_T_E_S_T_0()':
/workspaces/assg00-tamuc-student-team/src/assg00-tests.cpp:41: undefined reference to `isPrime(int)'
/usr/bin/ld: /workspaces/assg00-tamuc-student-team/src/assg00-tests.cpp:42: undefined reference to `isPrime
/usr/bin/ld: /workspaces/assg00-tamuc-student-team/src/assg00-tests.cpp:43: undefined reference to `isPrime
/usr/bin/ld: /workspaces/assg00-tamuc-student-team/src/assg00-tests.cpp:46: undefined reference to `isPrime
/usr/bin/ld: /workspaces/assg00-tamuc-student-team/src/assg00-tests.cpp:49: undefined reference to `isPrime
/usr/bin/ld: obj/assg00-tests.o:/workspaces/assg00-tamuc-student-team/src/assg00-tests.cpp:50: more undefin
collect2: error: ld returned 1 exit status
make: *** [include/Makefile.inc:54: 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 `primes.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 `primes.cpp` file, located in the `src` subdirectory, add in the following stub implementation.

```
/** @brief Is value prime
 *
 * 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 placeholder function, it always just answers `true` whenever it is asked if any `value` is a prime number or not. It is ignoring the input parameter, so it is not really testing if the `value` is prime or not.

Compile your program once again. You should find that it will now 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 -I../assg-base-0.3/include -c src/primes.cpp -o obj/primes.o
g++ -Wall -Werror -pedantic -g  obj/test-primes.o  obj/primes.o ../assg-base-0.3/obj/catch2-main.o -o test
g++ -Wall -Werror -pedantic -g  obj/main.o  obj/primes.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 tests
./test --use-colour yes


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


<isPrime()> function tests

src/test-primes.cpp:30


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

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

src/test-primes.cpp:44: FAILED:
  CHECK_FALSE( isPrime(8) )
with expansion:
```

```
  !true

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

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

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

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

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

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

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

src/test-primes.cpp:56: FAILED:
  CHECK_FALSE( isPrime(20) )
with expansion:
  !true

src/test-primes.cpp:63: FAILED:
  CHECK_FALSE( isPrime(1017) )
with expansion:
  !true

src/test-primes.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: tests] 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 just happens to be prime, then the test passes.

Commit Changes to Feedback 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 for this practice assignment. Lets add these changes to the `Feedback` pull request by making a commit, and pushing 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 a `stage all changes` to add all of these 3 file changes to the commit you are creating. Once the files are staged for the commit, write a commit message and then select the check-mark to commit these changes to your current `main` branch. Try and always use good commit messages. Read the following: Git Commit Messages Guidelines

Here is an example following the guidelines for this commit. Notice good commit messages have a title line, followed by a blank line, followed by 1 or more sentences of description.

```
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 `main` branch to your remote repository 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 `main` branch indication, should be a indicator of your repository push/pull status. It will be indicating that you now have 1 commit locally on the `main` branch that is ahead of the remote repository. If you push this indicator, your commit will be pushed to the `main` branch on your GitHub repository.

Once your have pushed your commit, this commit should appear in the `main` branch of your repository. Also, it should automatically appear in the `Feedback` pull request. The same compilation and tests will be run as GitHub actions. If you open up the `Feedback` pull request, you should be able to see the commit, and the results of running the GitHub auto-grading action. You can open this up to see the details. The auto-grader will fail at this point, because while your code is compiling, you are not yet passing all of the tests defined for this assignment.

The `Feedback` pull request will automatically gather all pushed commits to the `main` branch. You should also make sure you create and associate all of the task Issues for the assignment with the `Feedback` pull request. I will be checking your pull request regularly for the assignments. You can leave comments to me if you need help or clarifications. I may leave comments back, or add in extra work, issues or requirements that you need to fix before the assignment is considered fully complete.

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.

```
/** @brief Is value prime
 *
 * 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 has 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
g++ -Wall -Werror -pedantic -g -Iinclude -c src/assg00-tests.cpp -o obj/assg00-tests.o
g++ -Wall -Werror -pedantic -g -Iinclude -c src/primes.cpp -o obj/primes.o
g++ -Wall -Werror -pedantic -g  obj/assg00-tests.o  obj/primes.o obj/catch2-main.o  -o test
g++ -Wall -Werror -pedantic -g -Iinclude -c src/assg00-sim.cpp -o obj/assg00-sim.o
g++ -Wall -Werror -pedantic -g  obj/assg00-sim.o  obj/primes.o  -o sim

$ make tests
./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 `primes.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. If you didn't do it already, create the Task 2 issue in your GitHub account before beginning work on Task 2. There is still a second function that you need to write and pass the tests for in order to complete the assignment. 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 `primes.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 haven'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).

```
/** @brief Find primes in range
 *
 * 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.

```
> Executing task: make tests <

././test --use-colour yes

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

-------------------------------------------------------------------------------
<findPrimes()> function tests
-------------------------------------------------------------------------------
src/test-primes.cpp:71
```

```
src/test-primes.cpp:79: FAILED:
  CHECK( findPrimes(1, 10, false) == 169 )
with expansion:
  9 == 169

src/test-primes.cpp:84: FAILED:
  CHECK( findPrimes(10, 1500, false) == 4256 )
with expansion:
  9 == 4256 (0x10a0)

===============================================================================
test cases:  2 |  1 passed | 1 failed
assertions: 27 | 25 passed | 2 failed

make: *** [include/Makefile.inc:60: tests] Error 2
The terminal process "/bin/bash '-c', 'make tests'" failed to launch (exit code: 2).

Terminal will be reused by tasks, press any key to close it.
```

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 'main' branch, into the `Feedback` pull request. Then once again check your pull request. This third commit should now be a part of the pull request. Notice also that the project action check will still be failing. But if you look carefully at the action check 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. Also if you didn't do it, make sure you associate the issue for Task 2 with this pull request.

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.

```
/** @brief Find primes in range
 *
 * 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 `primes.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 may give feedback during your development, 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 not 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 request 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 using `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 white space 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.

Additional Information

The following are suggested online materials you may use to help you understand the tools and topics we have introduced in this assignment.

- Git Tutorials
- Git User Manual
- Git Commit Messages Guidelines
- Test-driven Development and Unit Testing Concepts
- Catch2 Unit Test Tutorial
- Getting Started with Visual Studio Code
- Visual Studio Code Documentation and User Guide
- Make Build System Tutorial
- Markdown Basic Syntax