

Exercises for Test

Test

Testing C code

The exercise is to make sure this module (all c files and header files) are tested properly. During the development of test code you may (nudge nudge) stumble upon errors in the software (I have made sure there are some). These errors should be fixed as well.

The tests shall be written so they can be performed non-interactively.

The source code folder should like this. You will need to copy files. Windows users should copy from the windows folders. GNU/Linux and MacOS (yes both!) should copy files from the gnu-linux folder.

```
+-- Doxyfile
+-- main.c
+-- Makefile
+-- math
|   +- math.c
|   +-- math.h
|
+-- parser.c
+-- parser.h
+-- settings.mk
+-- test
    +-- Makefile
    +-- test.c
    +-- test.h
```

Black box tests

1.

Add a test (to the Makefile) to the `black-box` rule. Simply copy the existing directive.

From:

```
black-box: $(BIN)
    @echo Calling mean with 1 2 3
    ./$(BIN) 1 2 3
```

to

```
black-box: $(BIN)
    @echo Calling mean with 1 2 3
    ./$(BIN) 1 2 3

    @echo Calling mean with 1 2 3 <--- change this if you want
    ./$(BIN) 1 2 3 <--- really change this!!!
```

The second directive should call the program (`mean`) with some clever arguments. You have to think about what can possibly go wrong.

Hint: The following line `return sum/size;` contains a severe error. Is there a value for either `sum` or `size` when things go mathematically wrong :)

Hint to the hint: dividing something with 4 is ok, dividing something with 3 is ok, dividing something with 2 is ok, dividing something with 1 is ok... ;)

2.

Add a test, as in (1), where you test some bad input (such as characters).

Unit tests

10.

Add a test in `test/test.c` by. The test shall test the `mean_value` function in the following way:

- pass a `NULL` pointer as first argument
- pass 100 as second argument

The expected result shall be?

Add a call to the macro `CHECK`

Note: `CHECK` is available as a function `check` in `test.c` as well. If you prefer to use the function instead of the macro - just do it :)

11.

Add a test in `test/test.c` by. The test shall test the `mean_value` function in the following way:

- pass the pointer p as first argument
- pass 100 as second argument

The expected result shall be?

Add a call to the macro CHECK

Note: CHECK is available as a function check in test.c as well. If you prefer to use the function instead of the macro - just do it :)

13.

Add a test in test/test.c by. The test shall test the `mean_value` function in the following way:

- pass a point to in, p2
- pass 10 as second argument

p2 shall be an array with 10 ints: 1 2 2 2 2 2 2 2 2 2

The expected result shall be?

Add a call to the macro CHECK

You may notice that the program really is not good at calculating the mean value. The mean value should be rounded to 2, but the returned mean value is 1. You will need to change the source code (math/math.c)

Having to change the code in order to pass the tests is somewhat cumbersome, but hey we want a good program. So it's worth it.

14.

Ok, let's do something fun. How should the function tell us that something went wrong?

If you're thinking about returning `-1` to signal something went wrong you have to answer the question of what is the mean value of the integer `-1`. Yes, the mean value of `-1` will be the same as the error code. So we can't both return an error code and a value. One possibility would be to limit the input to only be positive integers. This way we could reserve negative return values for error codes.

Note: Look at the manual for the function `atoi`, which is not a favorite function of mine. You will see a similar problem there.

Instead think about how you call `sscanf`. Yes, you pass a pointer to that variable you want `sscanf` to store the scanned value in. `sscanf` uses the return value to

tell the caller how the scanning went. This is good. We can get verify if `sscanf` succeeded and if success we use the variable.

We need to change the `mean_value` function in a similar way.

So let's change the function to this instead:

```
int mean_value(int *p, int size, int *mean_value);
```

15.

Write tests to verify if the function `mean_value` can deal with big numbers. Pass an array with 10 integers being of size 2^{31} (2 to the power of 31).

If you find a bug in the function, correct the function, recompile and rerun the tests.

Documentation (optional)

20.

Install doxygen*

- On Debian and Ubuntu: `sudo apt-get install doxygen`
- On MacOS: See <http://www.stack.nl/~dimitri/doxygen/download.html>
- On Windows. Two options: 1) install via cygwin (run the setup program again) 2) See <http://www.stack.nl/~dimitri/doxygen/download.html>

“Doxygen is the de facto standard tool for generating documentation from annotated C++ sources, but it also supports other popular programming languages such as C, ...” - <http://www.stack.nl/~dimitri/doxygen/index.html>

Doxygen's source code if anyone wants to have a look :) <https://github.com/doxygen/doxygen>

21.

Run the `doxygen` command in the directory where the mean programs is located.

`doxygen`

This should generate a html version of the “annotated” comments we've made in the source code (`math.h` and `parser.h`).

Look in the html folder.

```
ls html/
```

Open up the documentation. Play around with the comments and regenerate the manual.