Evil Hangman

This lab assumes that you have completed all steps involved in labs 1 through 4.

Switch to your HANGMAN directory: cd /Spring2017/COMP1020/HANGMAN

Clean up your directory by typing make clean

Add the following three empty files to your directory: unit_test.c, unit_test.h, and test_def.c

Modify your Makefile so that you add a new target called unit_test that will build a new executable that we can use to test our string data structure. Feel free to use other variables in your macro and remember to modify your clean target so that it can clean up the space for these new files. You will also need to add target lines for the new .o files: unit_test.o and test_def.o

```
unit_test: my_string.o unit_test.o test_def.o
	$(CC) $(CFLAGS) -o unit_test unit_test.o test_def.o my_string.o
```

We begin by building a framework for testing our code. We are going to create a main program in unit_test.c that will initialize an array of function pointers where every function pointer will hold the address of a test function which has the following signature:

```
Status long_function_name(char* buffer, int length);
```

The idea is to write a program that will automatically run all of our test functions and report on their success or failure. As we write more functions to test our code this unit will become more and more useful to us.

The following is a sample main program for unit_test.c that places two functions in the array of function pointers to be tested.

```
#include <stdio.h>
#include "unit_test.h"

int main(int argc, char* argv[])
{
  Status (*tests[])(char*, int) =
    {
       test_init_default_returns_nonNULL,
       test_get_size_on_init_default_returns_0
    };
  int number_of_functions = sizeof(tests) / sizeof(tests[0]);
  int i;
  char buffer[500];
  int success_count = 0;
  int failure_count = 0;

  for(i=0; i<number_of_functions; i++)
  {
    if(tests[i](buffer, 500) == FAILURE)
    {
      printf("FAILED: Test %d failed miserably\n", i);
      printf("\t%s\n", buffer);
      failure_count++;
    }
    else
    {
          // printf("PASS: Test %d passed\n", i);
          // printf("\t%s\n", buffer);
      success_count++;
    }
  }
  printf("Total number of tests: %d\n", number_of_functions);
  printf("%d/%d Pass, %d/%d Failure\n", success_count,
    number_of_functions, failure_count, number_of_functions);
  return 0;
}
```

The implementation code for each of your tests should go in the file test_defs.c

```
Status test_init_default_returns_nonNULL(char* buffer, int length)
{
  MY_STRING hString = NULL;

  hString = my_string_init_default();

  if(hString == NULL)
  {
    my_string_destroy(&hString);
    strncpy(buffer, "test_init_default_returns_nonNULL\n"
            "my_string_init_default returns NULL", length);
    return FAILURE;
  }
  else
  {
    my_string_destroy(&hString);
    strncpy(buffer, "\ttest_init_default_returns_nonNULL\n", length);
    return SUCCESS;
  }
}
```

In each test notice that they return the same type and take the same two types of parameters. The name for a test should be descriptive of what it is testing. Do not shy away from long function names. In the above test we are trying to create a string object using the my_string_init_default function. We then simply check to see if we get something that is not NULL back. This function does not verify that the object that we get back is in fact a string and does not attempt to do any further testing on it. Think about these tests as tiny tests that are to test one thing at a time.

My unit_test.h file contains the following:

```
#ifndef UNIT_TEST_H
#define UNIT_TEST_H
#include "my_string.h"

Status test_init_default_returns_nonNULL(char* buffer, int length);
Status test_get_size_on_init_default_returns_0(char* buffer, int length);

#endif
```

The second test tries to verify that the string we create using the default init function has a size of zero. We expect a default string to be "empty" and so it makes sense for it to not have anything stored in it after the init. Your tests should allocate the memory they need for their test and free it up before they return to the caller.

```
Status test_get_size_on_init_default_returns_0(char* buffer, int length)
{
  MY_STRING hString = NULL;
  Status status;

  hString = my_string_init_default();

  if(my_string_get_size(hString) != 0)
  {
    status = FAILURE;
    printf("Expected a size of 0 but got %d\n", my_string_get_size(hString));
    strncpy(buffer, "test_get_size_on_init_default_returns_0\n"
            "Did not receive 0 from get_size after init_default\n", length);
  }
  else
  {
    status = SUCCESS;
    strncpy(buffer, "test_get_size_on_init_default_returns_0\n"
            , length);
  }

  my_string_destroy(&hString);
  return status;
}
```

TA CHECKPOINT 1: Demonstrate to your TA that you can build unit_test with your Makefile and can run the sample code against the above given tests. You must demonstrate that your code does not have any memory leaks using valgrind.

The second part of this lab is about learning to test your own code. You must write a total of 23 more tests (so that you have a total of 25 tests. Your tests must satisfy the following. They must have your loginid in the name of the function so my functions will now be called test_dbadams_blah_blah_blah. The naming scheme is so that you can run your code against every test that every person in the class writes. This becomes sort of a competition to see who can write the best tests. For every "valid test" that you write that someone else fails you will gain a point. The top six scorers on testing will gain + 5 points on their entire lab grade.

A valid test is a test that conforms to the specification but does not force a particular implementation. Be careful about writing tests that your code can pass because you are making assumptions about how you implemented that behavior. A good test should pass on anyone with a valid implementation and should not require them to implement it exactly as you have. You can also gain points in the next phase by detecting that a particular test is invalid and "fixing" it to be valid.

The only other information I will give you about writing your test functions is that your tests should try to cover each of the interface functions with at least one test. No function should be left out. You will find that 25 tests is not nearly enough to test all of the behaviors but we are hoping that if every student or small group of students comes up with 25 tests then we will have enough tests to confirm that our code is good.

TA CHECKPOINT 2: Demonstrate to your TA that you can pass all 25 of your own tests with no memory leaks.