CST8234 – C Programming

Lab 04: Testing

Stack

A stack is a basic data structure that can be logically thought as linear structure represented by a real physical stack or pile, a structure where insertion and deletion of items takes place at one end called top of the stack.

Wikibooks

In this lab, you are going to learn how to use the **check framework** to test part of your of Assignment #1.

You are using the following data structures:

```
typedef int data_t;

typedef struct stack {
    data_t *array;
    int size;
    int top;
} stack_t;
```

You have been asked to implement the following functions:

```
int    stack_init( stack_t *s, int size );
void   stack_free( stack_t *s );
int    get_stack_size( stack_t s );
int    get_stack_top( stack_t s );
int    get_stack_elements( stack_t s );
int    is_stack_empty( stack_t stack );
int    is_stack_full( stack_t stack );
void   stack_clear( stack_t *s) ;
void   stack_push( stack_t *s, data_t item );
data_t stack_pop( stack_t *s);
data_t stack_peek( stack_t  s);
int    stack_contains( stack_t s, data_t value);
int    stack_find( stack_t s, data_t value);
void   stack_print( stack_t stack );
```

Manual testing

In this lab, you will alternate between improving your tests and improving your code. This resembles a formal software development process called *test-driven development* (TDD). The idea behind TDD is that you should write tests for a program **before** you write the program itself. This requires you to think critically about the expected behaviour of your program even before you begin to code it.

After you write some initial tests, you then write the minimal amount of code necessary to pass the test. After you have fixed any problems and all of the current tests are passing, you write more tests and then more code. Gradually, you will build up a large list of tests that help you verify that new changes don't break older features. This technique is also sometimes called the "**test a little, code a little**" strategy.

In this lab, you are going to build a set of test part of your first assignment. You could then use the same strategy to test the rest.

Ad-hoc Testing

Ad-hoc testing is an informal way of testing. Usually done without planning or documentation. **Test is only conducted if an error is foun**d.

This is most likely the type of testing you are conducting at this point. For example, in your second lab:

```
int digits( int a ) {

      int d = 0;

      while( a ) {
            a /= 10;
            d++;
            printf(“a = %d\nd = %d\n”, a, d );
      }
      return d;
}
```

or in your main, you may have tested as follow:

```
int main( void ) {


      if ( digits( 1000) == 4 )
            printf(“It WORKED!!!!!\n”);
      else
            printf(“NO WORKING!!!!!\n”);

      return EXIT_SUCCESS;
}
```

Notice that in this case, we will need to duplicate the code to test other values and be sure the function actually works.

Automated Testing

It is much more robust to separate a program's tests into a separate module, enabling them to be maintained, compiled, and managed separately. This will allow you to handle test crashes gracefully, and to use the **`main()`** function for actual program code.

In this course we are going to be using the Check framework to do automated testing of some of your labs and assignments.

Installation

To install the check libraries in Ubuntu:

```
sudo apt-get install check
```

Starting Up with testing

Test writing using **Check** is very simple.

Create a separate **.c** file in your assignment directory (where you are developing your assignment 1). I'm suggesting your test file to be called: **test_PROGRAM_NAME.c** For your assignment #1, then the test file should be called: **test_stack.c**

Since your test is testing your **stack** assignment, it needs to have access to your data type, function prototypes and to the check framework.

Your **test_stack.c** must begin with:

```
#include <stdio.h>
#include <stdlib.h>
#include <check.h>
#include "stack.h"
```

Your **Makefile** already includes the needed directives to compile your test:

```
TEST=test_stack
MODS=stack.o
test: $(TEST)
      ./$(TEST)  # This will compile and run your test
TESTCFLAGS=$(CFLAGS) -Wno-gnu-zero-variadic-macro-arguments
TESTLIBS=-lcheck -lm -lpthread -lrt

$(TEST): $(TEST).o $(MODS)
      $(CC) $(TESTLDFLAGS) -o $(TEST) $^ $(LIBS) $(TESTLIBS)

$(TEST).o: $(TEST).c
      $(CC) -c $(TESTCFLAGS) $<
```

To compile your test file:

```
# make test
```

Writing Tests

The basic unit test looks as follows:

```
START_TEST (test_name) {
  /* unit test code */
}
END_TEST
set
```

The **START_TEST/END_TEST** pair are macros that setup basic structures to permit testing. The **Check** framework replaces these macros with code that runs the test, handles any crashes, and records the result. Every test is named, and the name is declared as a parameter to the **START_TEST** macro.

Check provides many convenience functions for actually performing tests. Here are the most important:

```
/*
 * Evaluate expr;
 * the check passes if the result is true and fails if it is false.
 */
ck_assert(expr);

/*
 * The check passes if the two integers are equal
 * and fails if they are not
 */
ck_assert_int_eq(x, y);

/*
 * The check passes if the two character strings are equal
 *(according to the strcmp function) and fails if they are not.
 */
ck_assert_str_eq(x, y)
```

Let's write a small test to check if your functions set_init( ) is working properly:

```
/**********************  INIT TEST **********************************/
/* TEST:  init function:
 *      ( 1 ) init_NORMAL
 *              stack has been initialized with size > 0
 *              No added elements, top is the size of the stack
 *              array should contain a valid address
 *             size should be STACK_A
 */

#define STACK_A 5

START_TEST ( init_NORMAL ) {

      stack_t stack  = { NULL, 0, 0 };

      stack_init(&stack, STACK_A );
      ck_assert(get_stack_size(stack) == STACK_A);
      ck_assert(stack.array != NULL );
      ck_assert(get_stack_top(stack) == STACK_A);
      stack_free(&stack);
}
END_TEST
```

Now that you have a test, you can to aggregate it into a suit and run them with a suite runner.

```
Suite * test_suite(void) {
    Suite *s;
    TCase *tc_stack;

    s = suite_create("ALL CASES");
    tc_stack = tcase_create("STACK");

     tcase_add_test(tc_stack, init_NORMAL);

     suite_add_tcase(s, tc_stack);
     return s;
}
/********************************************************************/
/* run_testsuite( )
 ********************************************************************/
int run_testsuite(){

      int fail_nr;
      Suite *s;
      SRunner *sr;

      printf("%s\n", FILLER );

      s = test_suite();
      sr = srunner_create(s);

      srunner_run_all(sr, CK_VERBOSE);

      fail_nr = srunner_ntests_failed(sr);
      srunner_free(sr);
      printf("%s\n", FILLER );
      printf("%s\n", fail_nr ? TEST_FAILURE : TEST_SUCCESS );
      printf("%s\n", FILLER );

      return ( !fail_nr ) ? EXIT_SUCCESS : EXIT_FAILURE;
}

int main(int argc, char* argv[]) {
    return run_testsuite();
}
```

where:

```
#define TEST_SUCCESS "SUCCESS: All current tests passed!"
#define TEST_FAILURE "FAILURE: At least one test failed or crashed"
```

The details are outside the scope of this course, in a nutshell, the **`test_suite( )`** function aggregates the test cases you create (add a line of each test you have) and the **`run_testsuite()`** run the test, keeps track of the number of failed assert and prints then end results.

For this lab, you are to write a test to verify the correct functionality of your assignment. We will discuss in the lab test plans and how to achieve this.