Lab 3 - Developing C Functions Using the Sput Testing Framework

Before starting this lab, please read *Important Considerations When Submitting Files to Brightspace*, at the end of this document.

Note that if you are using a different IDE, e.g. Clion, adjust the VS Code instructions as required.

Part 1 - The *sput* Testing Framework

In SYSC 2006, we use a simple testing framework named *sput* to automate the process of testing C functions. In this exercise, you will learn how to read test functions that use sput and interpret the output produced by the framework when the tests are executed.

Step 1: Depending on your operating system, download (**Lab3Part1_Windows.zip** or **Lab3Part1_macOS.zip**) the starting code from Brightspace and extract the zipped file.

Step 2: Open the unzipped folder in VS Code as you did in previous labs.

Step 3: Lab3Part1_xxx contains four files: power.c, power.h, main.c and sput.h.

- power.h contains the declarations (function prototypes) for the functions in power.c. **Do not modify power.h.**
- power.c contains flawed implementations of four functions named `power1`, `power2`, `power3` and `power4` that calculate x^n for non-negative integers n.
- main.c and sput.h implement a *test harness* (functions that will test your code). main.c contains five functions. Functions `test_power1`, `test_power2`, `test_power3` and `test_power4` contain the code for four test suites, one for each of the four functions in power.c. (These functions are explained in Step 6.) Function `main` controls the execution of the test suites. **Do not modify main.c and sput.h.**

Step 4: Run the project. It should build without any compilation or linking errors, but you will see in the TERMINAL tab that all tests failed.

Step 5: Read this step carefully.

Double click the icons for main.c and power.c to open these files in editor windows. Locate function `test_power1`. This is the function that checks if `power1` correctly calculates 2^0, 2^1, 2^2 and 2^3. Here is the code for this function:

```
static void test_power1(void)
{
    sput_fail_unless(power1(2, 0) == 1, "power1(2, 0)");
    printf("Expected result: 1, actual result: %d\n", power1(2, 0));

    sput_fail_unless(power1(2, 1) == 2, "power1(2, 1)");
    printf("Expected result: 2, actual result: %d\n", power1(2, 1));
```

```
    sput_fail_unless(power1(2, 2) == 4, "power1(2, 2)");
    printf("Expected result: 4, actual result: %d\n", power1(2, 2));

    sput_fail_unless(power1(2, 3) == 8, "power1(2, 3)");
    printf("Expected result: 8, actual result: %d\n", power1(2, 3));
}
```

The four checks are performed by `sput_fail_unless`, which has two arguments. The first argument is a condition that must be `true` in order for the test to pass. The second argument is a descriptive string that is displayed when `sput_fail_unless` is executed.

For example, the first call to `sput_fail_unless` determines if `power1` correctly calculates 2^0, which is 1. The first argument is the expression `power1(2, 0) == 1`, which compares the value that is returned by `power1` (the calculated value of 2^0) to the expected result, 1. This test fails *unless* the value returned by `power1(2, 0)` equals 1; in other words, the test passes only if the value returned by `power1` is correct.

After `sput_fail_unless` returns, `printf` is called to display the value that we expect a correct implementation of `power1` to return (that is, 1), followed by the actual value returned by the function.

The next three pairs of `sput_fail_unless`/`printf` calls check if `power1` correctly calculates 2^1, 2^2 and 2^3.

Execute the project. The test harness runs, and in the TERMINAL tab you will see the output from the test harness. It will be similar to this:

```
Running test harness for SYSC 2006 Lab 3, Part 1

== Entering suite #1, "Testing power1()" ==

[1:1]  test_power1:#1  "power1(2, 0)"  FAIL
!    Type:      fail-unless
!    Condition: power1(2, 0) == 1
!    Line:      17
Expected result: 1, actual result: 0
[1:2]  test_power1:#2  "power1(2, 1)"  FAIL
!    Type:      fail-unless
!    Condition: power1(2, 1) == 2
!    Line:      19
Expected result: 2, actual result: 0
[1:3]  test_power1:#3  "power1(2, 2)"  FAIL
!    Type:      fail-unless
!    Condition: power1(2, 2) == 4
!    Line:      23
Expected result: 4, actual result: 0
```

```
[1:4]  test_power1:#4  "power1(2, 3)"  FAIL
!    Type:      fail-unless
!    Condition: power1(2, 3) == 8
!    Line:      23
Expected result: 8, actual result: 0

--> 4 check(s), 0 ok, 4 failed (100.00%)
Tests for other functions won't be run until power1 passes all tests.

==> 4 check(s) in 1 suite(s) finished after 0.00 second(s),
    0 succeeded, 4 failed (100.00%)

[FAILURE]
```

As we review the output, we see that all four checks in `test_power1` failed; in other words, the conditions in all four `sput_fail_unless` calls are `false`. Specifically,

- Condition `power1(2, 0) == 1` in the call to `sput_fail_unless` on line 17 is `false`; `power(2, 0)` returned `0` instead of the expected value, `1`;
- Condition `power1(2, 1) == 2` in the call to `sput_fail_unless` on line 19 is `false`; `power(2, 1)` returned `0` instead of the expected value, `2`;
- Condition `power1(2, 2) == 4` in the call to `sput_fail_unless` on line 21 is `false`; `power(2, 2)` returned `0` instead of the expected value, `4`;
- Condition `power1(2, 3) == 8` in the call to `sput_fail_unless` on line 23 is `false`; `power(2, 2)` returned `0` instead of the expected value, `8`;

Step 6: Trace the code in `power1` (in file power.c) "by hand", step by step. Identify the incorrect statement or statements that cause `power1` to return 0 instead of the correct power of 2. Edit `power1` to correct the flaw. **Do not add any additional local variables to the function; they are not needed.**

Build and execute the project. The test harness will run.

After you have corrected `power1`, the output displayed by the test harness should look like this:

```
== Entering suite #1, "Testing power1()" ==

[1:1]  test_power1:#1  "power1(2, 0)"  pass
Expected result: 1, actual result: 1
[1:2]  test_power1:#2  "power1(2, 1)"  pass
Expected result: 2, actual result: 2
[1:3]  test_power1:#3  "power1(2, 2)"  pass
Expected result: 4, actual result: 4
[1:4]  test_power1:#4  "power1(2, 3)"  pass
Expected result: 8, actual result: 8

--> 4 check(s), 4 ok, 0 failed (0.00%)
```

If any of the checks in suite #1 fail, repeat this step until all checks pass. When all checks in suite #1 pass, the harness will also run test suite #2, which tests `power2`. Some of the checks in suite #2 will fail.

Step 7: Review the console output from suite #2. Use this output to help you determine the flaw in `power2`. Correct the flaw. **Do not add any additional local variables to the function; they are not needed.**

Build and execute the project. Review the console output and determine if your function passes all the tests in test suite #2. If any of the checks fail, repeat this step until all checks pass. When all checks in suite #2 pass, the harness will also run test suite #3, which tests `power3`. Some of the checks in suite #3 will fail.

Step 8: Use the output from test suite #3 to help you determine the flaw in `power3`. Correct the flaw. **Do not add any additional local variables to the function; they are not needed.**

Build and execute the project. When all checks in suite #3 pass, the harness will also run test suite #4, which tests `power4`. Some of the checks in suite #4 will fail.

Step 9: Use the output from test suite #4 to help you determine the flaw in `power4`. Correct the flaw. **Do not add any additional local variables to the function; they are not needed.**

Build and execute the project. When all checks in suite #4 pass, the summary output by the test harness will look like this:

```
==> 16 check(s) in 4 suite(s) finished after 0.00 second(s),
    16 succeeded, 0 failed (0.00%)
[SUCCESS]
```

Step 10: Save and Close the project. (Select File > Close project from the menu bar.)

Part 2 - C Functions; Function Composition

In this part, you will design and implement functions by applying *function composition*. We will illustrate this concept with a short example:

Here is a function that returns the area of a circle with a given radius:

```
/* Return the area of a circle with radius r. */
double area_of_circle(double r)
{
      return acos(-1.0) * r * r;  // acos(-1.0) computes pi
}
```

Notice that we did not assign the value computed by this expression:

```
      acos(-1.0) * r * r
```

to a local variable and return that variable. The function is more concise if the `return` statement contains the expression.

Here is a function that calculates the distance between two points (x_1, y_1) and (x_2, y_2):

```
/* Return the distance between points (x1, y1) and (x2, y2). */
double distance(int x1, int y1, int x2, int y2)
{
     int dx = x2 - x1;
     int dy = y2 - y1;
     return sqrt(dx * dx + dy * dy);
}
```

The function body could be written as a single `return` statement, but by using local variables `dx` and `dy`, we do not have to compute the differences x_2 - x_1 and y_2 - y_1 twice.

Now suppose you are asked to write a function that takes two points, the center of a disk and a point on the perimeter, and returns the area of the disk:

```
/* Return the area of a disk with center point (xc, yc) and
 * a point on the perimeter (xp, yp).
 */
double area_of_disk(int xc, int yc, int xp, int yp)
```

First, we calculate the radius of the disk, which is the distance between the two points. To do this, we call function `distance`:

```
     double radius = distance(xc, yc, xp, yp);
```

Next, we calculate the area of a circle with that radius. To do that, we will call `area_of_circle`:

```
     double area = area_of_circle(radius);
```

We encapsulate these statements in the `area_of_disk` function:

```
/* Return the area of a disk with center point (xc, yc) and
 * a point on the perimeter (xp, yp).
 */
double area_of_disk(int xc, int yc, int xp, int yp)
{
    double radius = distance(xc, yc, xp, yp);
    double area = area_of_circle(radius);
    return area;
}
```

Recall that a function call that returns a value, such as `distance(xc, yc, xp, yp)` or `area_of_circle(radius)`, is an expression. The arguments of a function call are expressions, so arguments can be function calls. While local variables like `radius` and `area` may be useful when debugging, we can revise `area_of_disk`, making it more concise, by

eliminating the local variables and *composing* the function calls:

```
/* Return the area of a disk with center point (xc, yc) and
 * a point on the perimeter (xp, yp).
 */
double area_of_disk(int xc, int yc, int xp, int yp)
{
    return area_of_circle(distance(xc, yc, xp, yp));
}
```

General Requirements for Part 2

For those students who already know C: do not use arrays, structs, or pointers. They are not necessary for this lab.

Your functions should not be recursive. Repeated actions must be implemented using C's `while`, `for`, or `do-while` loop structures.

None of the functions you write should perform console input; for example, contain `scanf` statements. None of your functions should produce console output; for example, contain `printf` statements.

You must format your C code so that it adheres to one of two commonly-used conventions for indenting blocks of code and placing braces (K&R style or BSD/Allman style). VS Code makes it easy to do this. (Instructions for formatting the code are in VS Code installation instructions.)

Finish each exercise (i.e., write the function and verify that it passes all of its tests) before you move on to the next one. Do not leave testing until after you have written all the functions.

Getting Started

Step 1: Depending on your operating system, download (**Lab3Part2_Windows.zip** or **Lab3Part2_macOS.zip**) the starting code from Brightspace and extract the zipped file.

Step 2: Open the unzipped folder in VS Code as you did in previous labs

Step 3: Lab3Part2_xxx contains four files: main.c, composition.c, composition.h and sput.h.

- composition.h contains the declarations (function prototypes) for the functions you will implement. **Do not modify composition.h.**
- composition.c contains incomplete definitions of the three functions you have to design and code.
- main.c and sput.h implement the *test harness* for Exercises 1 - 3. **Do not modify main.c and sput.h.**

Step 4: Run the project. It should build without any compilation or linking errors. The test harness will report several failures as it runs, which is what we would expect, because you have

not started working on the functions the harness tests.

Step 5: Open composition.c in the editor and do Exercises 1 through 3. Do not make any changes to main.c, composition.h or sput.h. All the code you will write must be in composition.c.

Exercise 1

The factorial *n*! is defined for a positive integer *n* as:

$$n! = n \times (n\text{-}1) \times (n\text{-}2) \times \ldots \times 2 \times 1.$$

For example, $4! = 4 \times 3 \times 2 \times 1 = 24$.

0! is defined as: 0! = 1.

An incomplete implementation of a function named `factorial` is provided in composition.c. The function header is:

```
int factorial(int n)
```

This function calculates and returns `n`!.

Finish the definition of this function. Your function should assume that `n` is 0 or positive; i.e., **it should not verify that n is ≥ 0 before calculating n!.**

Aside: for C compilers that use 32-bit integers, the largest value of type `int` is 2^{31} - 1. Because the return type of `factorial` is `int` and *n*! grows rapidly as *n* increases, this function will be unable to calculate factorials greater than 15!

Run the project. Use the console output to help you identify and correct any flaws. Verify that your function passes all the tests in test suite #1 before you start Exercise 2.

Exercise 2

Suppose we have a set of *n* distinct objects. There are *n*! ways of ordering or arranging *n* objects, so we say that there are *n*! permutations of a set of *n* objects. For example, there are 2! = 2 permutations of {1, 2}: {1, 2} and {2, 1}.

If we have a set of *n* objects, there are $n!/(n - k)!$ different ways to select an ordered subset containing *k* of the objects. That is, the number of different ordered subsets, each containing *k* objects taken from a set of *n* objects, is given by:

$$n!/(n - k)!$$

For example, suppose we have the set {1, 2, 3, 4} and want an ordered subset containing 2 integers selected from this set. There are 4! / (4 - 2)! = 12 ways to do this: {1, 2}, {1, 3}, {1, 4}, {2, 1}, {2, 3}, {2, 4}, {3, 1}, {3, 2}, {3, 4}, {4, 1}, {4, 2} and {4, 3}.

An incomplete implementation of a function named `ordered_subsets` is provided in composition.c. This function has two integer parameters, `n` and `k`, and has return type `int`. This

function returns the number of ways an ordered subset containing k objects can be obtained from a set of n objects.

Finish the definition of this function. Your function should assume that n and k are positive and that n >= k; i.e., the function should **not** check if n and k are negative values, or compare n and k.

For each factorial calculation that's required, your `ordered_subsets` function must call the `factorial` function you wrote in Exercise 1. In other words, do not copy/paste code from `factorial` into `ordered_subsets`. **Do not declare any local variables in your `ordered_subsets` function. There is no need to store the values returned by `factorial` in local variables.**

Run the project. Use the console output to help you identify and correct any flaws. Verify that your function passes all the tests in test suite #2 before you start Exercise 3.

Exercise 3

Combinations are not concerned with order. Given a set of *n* distinct objects, there is only one combination containing all *n* objects.

If we have a set of *n* objects, there are $n!/((k!)(n - k)!)$ different ways to select *k* unordered objects from the set. That is, the number of combinations of *k* objects that can be chosen from a set of *n* objects is:

$$n!/((k!)(n - k)!)$$

The number of combinations is also known as the *binomial coefficient*.

For example, suppose we have the set {1, 2, 3, 4} and want to choose 2 integers at a time from this set, without regard to order. There are 4! / ((2!) (4 - 2)!) = 6 combinations: {1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4} and {3, 4}.

An incomplete implementation of a function named `binomial` is provided in composition.c. This function has two integer parameters, n and k, and has return type `int`. This function returns the number of combinations of k objects that can be chosen from a set of n objects.

Finish the definition of this function. Your function should assume that n and k are positive and that n >= k; i.e., the function should **not** check if n and k are negative, or compare n and k.

Your `binomial` function must call your `ordered_subsets` and `factorial` functions. **Do not declare any local variables in your `binomial` function. There is no need to store the values returned by `factorial` and `ordered_subsets` in local variables.**

Run the project. Use the console output to help you identify and correct any flaws. Verify that your function passes all the tests in test suite #3.**Part 3 - Recursive function**

In this part, you will design and implement recursive functions.

General Requirements for Part 3

Your functions must be recursive. Repeated actions should not be implemented using C's `while`, `for`, or `do-while` loop structures.
None of the functions you write should perform console input; for example, contain `scanf` statements. None of your functions should produce console output; for example, contain `printf` statements.
You must format your C code so that it adheres to one of two commonly-used conventions for indenting blocks of code and placing braces (K&R style or BSD/Allman style). VS Code makes it easy to do this. (Instructions for formatting the code are in VS Code installation instructions.)

Getting Started

Step 1: Depending on your operating system, download (**Lab3Part3_Windows.zip** or **Lab3Part3_macOS.zip**) the starting code from Brightspace and extract the zipped file
Step 2: Open the unzipped folder in VS Code as you did in previous labs
Step 3: `Lab3Part3_xxx` contains 3 files:

- recursive_functions.h contains the declarations (function prototypes) for the functions you will implement. **Do not modify recursive_functions.h.**
- recursive_functions.c contains incomplete definitions of the functions you have to design and code.
- main.c and sput.h implement the *test harness* for the exercises. **Do not modify main.c and sput.h.**

Step 4: Run the project. It should build without any compilation or linking errors. The test harness will report several failures as it runs, which is what we would expect, because you have not started working on the functions the harness tests.
Step 5: Open recursive_functions.c in the editor and do Exercises 1. Do not make any changes in main .c and recursive_functions.h file. All the code you will write must be in recursive_functions.c.

Exercise 1

File recursive_functions.c contains an incomplete definition of a function named `num_digits` that returns the number of digits in integer *n*, *n* >= 0. The function prototype is:

```
int num_digits(int n);
```

If *n* < 10, it has one digit, which is *n*. Otherwise, it has one more digit than the integer *n* / 10. For example, 7 has one digit. 63 has two digits, which is one more digit than 63 / 10 (which is 6). 492 has three digits, which is one more digit than 492 / 10, which is 49.

Define a recursive formulation for `num_digits`. You'll need a formula for the recursive case

and a formula for the stopping (base) case. Using this formulation, implement `num_digits` as a recursive function. (Recall that, in C, if `a` and `b` are values of type `int`, `a / b` yields an `int`, and `a % b` yields the integer remainder when `a` is divided by `b`.) Your `num_digits` function cannot have any loops.

Function `test_exercise_1` has seven test cases for your `num_digits` function. It calls `test_num_digits` seven times, once for each test case. Notice that `test_num_digits` has two arguments: the value that will be passed to `num_digits`, and the value that a correct implementation of `num_digits` will return (the expected result).

Use the console output to help you identify and correct any flaws. Verify that `num_digits` passes all of its tests.

Extra Practice (you do not need to submit the solution for this exercise)

Exercise 2

In this exercise, you'll explore a solution to the problem of calculating x^n recursively that reduces the number of recursive calls. File recursive_functions.c contains an incomplete definition of a function named `power2` that calculates and returns x^n for $n >= 0$, using the following recursive formulation:

$x^0 = 1$

$x^n = (x^{n/2})^2$, $n > 0$ and n is even

$x^n = x * (x^{n/2})^2$, $n > 0$ and n is odd

The function prototype is:

```
double power2(double x, int n);
```

Implement `power2` as a recursive function, using the recursive formulation provided above. Your `power2` function cannot have any loops, and it cannot call the `pow` function in the C standard library or the `power` function you wrote for Exercise 1.

Function `test_exercise_2` has five test cases for your `power2` function: (a) 3.5^0, (b) 3.5^1, (c) 3.5^2, (d) 3.5^3, and (e) 3.5^4. It calls `test_power2` five times, once for each test case.

Build and execute the project. If you translate the recursive formulation into C correctly, you'll find that your `power2` function performs recursive calls "forever". Add the following statement at the start of your function, to print the values of its parameters each time it is called:

```
printf("x = %.1f, n = %d\n", x, n);
```

The information displayed on the console should help you figure out what's going on. Hint: what

happens when parameter `n` equals 2; i.e., when you call `power2` to square a value? Drawing some memory diagrams may help!

To solve this problem, we can change the recursive formulation slightly:

$x^0 = 1$

$x^n = (x^{n/2}) * (x^{n/2})$, $n > 0$ and n is even

$x^n = x * (x^{n/2}) * (x^{n/2})$, $n > 0$ and n is odd

Change your `power2` function to use the revised formulation. Are there any other changes you can make that will reduce the number of times that `power2` is called recursively?

Wrap-up

Submit power.c, composition.c and recursive_functions.c to Brightspace.

Before submitting your lab work:

- Review *Important Considerations When Submitting Files to Brightspace*, below. Remember: submit early, submit often.
- Make sure that power.c, composition.c and recursive_functions.c have been formatted to use K&R style or BSD/Allman style, as explained in *Part 2*, *General Requirements*.
- Ensure that you are submitting the files that contain your solutions, and not the unmodified files you downloaded from Brightspace!

You are permitted to make changes to your solutions and resubmit the files as many times as you want, up to the deadline. You need to demo your work to TAs during the lab for grades.

Solutions that are emailed to your instructor or a TA will not be graded, even if they are emailed before the deadline.

Important Considerations When Submitting Files to Brightspace

It is your responsibility to ensure that all submissions (e.g., lab work) are completed on time. There are a very limited number of exceptions to this policy, as described in the *Academic Regulations of the University*, Section 4.4, *Deferred Term Work*, and all such requests must have appropriate supporting documentation.

Technical problems do not exempt you from the requirement to meet the submission deadlines. You must submit your files before demoing them to TAs for marks. In addition, you are advised to:

- periodically submit your "work in progress" as a draft; for example, submit the file(s)

containing the work you've completed at least once a day.

- **Remember that your mark will be 0 if the TA did not check your work in the lab or your submission is not on the system when the TA checked your work.**

Make sure that each file you submit has the specified filename and format. For example, if you submit a C source code file with an incorrect filename, you will receive 0 for the lab. .

Make sure that every C source code file you submit can be opened, compiled, and run in the VS Code IDE. Make sure that your code does not use any non-standard C extensions or libraries. If your code does not compile and run, you will receive 0 for that submission.

We recommend that, after submitting source code files, you download the files from Brightspace and verify that:

- the downloaded files have the correct filenames (see the previous paragraph);
- the files can be opened and viewed in the IDE's editor;
- no syntax errors are reported when the code is compiled.

Some exercises were adapted from problems by Frank Carrano, Paul Helman and Robert Veroff, and Cay Horstmann