Carleton University
Department of Systems and Computer Engineering
SYSC 2006 - Foundations of Imperative Programming

Lab 2 - Introduction to Coding and Testing C Functions

Demo/Grading

After you finish all the exercises, call a TA, who will review your solutions, ask you to run the test harnesses provided on cuLearn, and assign a grade. For those who don't finish early, a TA will grade the work you've completed, starting about 30 minutes before the end of the lab period. **Any unfinished exercises should be treated as "homework"; complete these on your own time, before your next lab.**

Prerequisite Reading

zyBooks: Chapters 1 to 5

Part 1 - The *sput* Testing Framework

In SYSC 2006 we use a simple testing framework named *sput* to automate the process of testing the functions. In this exercise, you'll learn how to interpret the output produced by *sput* to help you locate and correct bugs in code.

Step 1: Download power_functions.zip from cuLearn. Unzip this file; for example, by right-clicking on the compressed folder icon, then selecting Extract All... from the pop-up menu. You should now have a folder named power_functions that contains a Pelles C project.

Step 2: Launch Pelles C. From the menu, select File > Open... Use the Open dialogue box to navigate into the power_functions folder, select file power_functions (the file of type Pelles C Project File), then click the Open button. Pelles C will open the project.

The project contains four files, which are listed in a pane in the IDE:

- power.c contains flawed implementations of four functions that calculate x^n for non-negative integers n.
- power.h (listed under Include files) contains the declarations (function prototypes) for the functions. **Do not modify power.h.**
- main.c and sput.h implement a *test harness* (functions that will test your code). **Do not modify main.c and sput.h.**

Double click the icons for `main.c` and `power.c` to open these files in editor panes.

- power.c contains four functions named `power1`, `power2`, `power3` and `power4`.
- main.c contains four *test suites*, one for each of the four functions in power.c. These suites are implemented by functions named `test_power1`, `test_power2`, `test_power3` and `test_power4`. (These functions are explained in Step 4.) Function `main` initializes *sput*, then executes the suites.

Step 3: Build the project. It should build without any compilation or linking errors.

Step 4: Read this step carefully. To use the test harness, you need to understand the output it displays.

The first test function called by the harness is `test_power1`, which checks if `power1` correctly calculates 2^0, 2^1, 2^2, and 2^3:

```
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));
}
```

Each check is performed by calling function `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 by the harness.

For example, the first call to `sput_fail_unless` is passed the value of the expression `power1(2, 0) == 1`. This means that the first test passes only if `power1` correctly calculates and returns 2^0.

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

Reading `test_power1`, we see that the four pairs of `sput_fail_unless`/`printf` calls check if `power1` correctly calculates 2^0, 2^1, 2^2, and 2^3.

Execute the project.

When the test harness runs, a Console program output window will open. The output from the test harness will be similar to this:

```
Running test harness for SYSC 2006 Winter 2018 Lab 2, Exercise 1

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

[1:1]  test_power1:#1  "power1(2, 0)"  FAIL
!    Type:      fail-unless
!    Condition: power1(2, 0) == 1
!    Line:      22
Expected result: 1, actual result: 0
[1:2]  test_power1:#2  "power1(2, 1)"  FAIL
!    Type:      fail-unless
!    Condition: power1(2, 1) == 2
!    Line:      24
Expected result: 2, actual result: 0
```

```
[1:3]  test_power1:#3  "power1(2, 2)"  FAIL
!    Type:      fail-unless
!    Condition: power1(2, 2) == 4
!    Line:      26
Expected result: 4, actual result: 0
[1:4]  test_power1:#4  "power1(2, 3)"  FAIL
!    Type:      fail-unless
!    Condition: power1(2, 3) == 8
!    Line:      28
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]
*** Process returned 1 ***
Press any key to continue...
```

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 `power(2, 0) == 1` in the call to `sput_fail_unless` on line 22 is `false`; `power(2, 0)` returned `0` instead of the expected value, `1`;
- Condition `power(2, 1) == 2` in the call to `sput_fail_unless` on line 24 is `false`; `power(2, 1)` returned `0` instead of the expected value, `2`;
- Condition `power(2, 2) == 4` in the call to `sput_fail_unless` on line 26 is `false`; `power(2, 2)` returned `0` instead of the expected value, `4`;
- Condition `power(2, 3) == 8` in the call to `sput_fail_unless` on line 28 is `false`; `power(2, 2)` returned `0` instead of the expected value, `8`;

Step 5: Examine the code for `power1`. The output from the test harness tells us that `power1` always returns 0; yet the return statement is not `return 0`. Trace the code in `power1` "by hand", step by step. Identify the incorrect statement or statements. Edit `power1` to correct the flaw.

Build the project, correcting any compilation errors, then execute the project. The test harness will run.

After you've corrected `power1`, the output displayed by the *sput* 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`.

Step 6: Some of the checks in suite #2 will fail.

Review the console output from suite #2. Use this output to help you determine the flaw in `power2`.

Correct the flaw. Build the project, correcting any compilation errors, then execute the project. Review the console output, and verify that 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`.

Step 7: Some of the checks in suite #3 will fail. Use the output from test suite #3 to help you determine and correct the flaw in `power3`. When all checks in suite #3 pass, the harness will also run test suite #4, which tests `power4`.

Step 8: Some of the checks in suite #4 will fail. Use the output from test suite #4 to help you determine and correct the flaw in `power4`. When all checks in suite #4 pass, the summary output by *sput* will look like this:

```
==> 16 check(s) in 4 suite(s) finished after 0.00 second(s),
    16 succeeded, 0 failed (100.00%)

[SUCCESS]
Press any key to continue...
```

Part 2 - Developing Some Simple C Functions

General Requirements

You have been provided with four files:

- exercises.c contains incomplete definitions of four functions you have to design and code.
- exercises.h contains the declarations (function prototypes) for the functions you'll implement. **Do not modify exercises.h.**
- main.c and sput.h implement the *test harness* for Exercises 2-5. **Do not modify main.c and sput.h.**

For those students who already know C or C++: do not use arrays, structs or pointers. They aren't 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). Pelles C makes it easy to do this. To select the formatting style:

- From the menu bar, select Tools > Options... An Options box will appear.
- Click the Tabs tab
- In the C formatting style box, click a radio button to select either Style 1 (for K&R style) or Style 2 (which appears to be close to BSD/Allman style).
- Click OK. The Options box will close.

To format the code in your editor window:

- Select Edit > Select all. Your code will be highlighted.
- Select Source > Convert to.
- From the submenu, select Formatted C code. Your highlighted code will be reformatted to conform to the selected style.

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. Don't leave testing until after you've written all the functions.

Getting Started

Step 1: Create a new project named Lab2Part2.

- If you're using the 64-bit edition of Pelles C, the project type should be Win 64 Console program (EXE). (Although the 64-bit edition of Pelles C can build 32-bit programs, you may run into difficulties if you attempt to use the debugger to debug 32-bit programs.)
- If you're using the 32-bit edition of Pelles C, the project type should be Win32 Console program (EXE).

When you finish this step, Pelles C will create a project folder named Lab2Part2.

Step 2: Download files main.c, exercises.c, exercises.h and sput.h from cuLearn. Move these files into your Lab2Part2 folder.

Step 3: You must add main.c and exercises.c to your project (moving the files to your project folder doesn't do this).

- Select Project > Add files to project... from the menu bar.
- In the dialogue box, select main.c, then click Open. An icon labelled main.c will appear in the Pelles C project window.
- Repeat this step for exercises.c.

You don't need to add exercises.h and sput.h to the project. Pelles C will do this after you've added main.c.

Step 4: Build the project. It should build without any compilation or linking errors.

Step 5: Execute the project. The test harness will report several failures as it runs, which is what we'd expect, because you haven't started working on the functions the harness tests.

Step 6: Open exercises.c in the editor and do Exercises 1 through 3. Don't make any changes to main.c, exercises.h or sput.h. All the code you'll write must be in exercises.c.

Exercise 1

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

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

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 exercises.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!

Build the project, correcting any compilation errors, then execute the project. The test harness will run. 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 exercises.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 2. In other words, don't copy/paste code from `factorial` into `ordered_subsets`.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. 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 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 `exercises.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.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Use the console output to help you identify and correct any flaws. Verify that your function passes all the tests in test suite #3.

Wrap-up

1. Remember to have a TA review your solutions to the exercises, assign a grade (Satisfactory, Marginal or Unsatisfactory) and have you initial the grading/sign-out sheet.

2. Remember to back up your project folder before you leave the lab; for example, copy it to a flash drive and/or a cloud-based file storage service. All files you've created on the hard disk will be deleted when you log out.

Extra Practice - Exercise 4

The cosine of an angle *x* can be computed from the following infinite series:

$$\cos x = 1 - x^2/2! + x^4/4! - x^6/6! + \ldots$$

Note that *x* is measured in radians, not degrees. (Recall that there are Π radians in 180 degrees.)

We can approximate the cosine of an angle by summing several terms of this series. An incomplete implementation of a function named `cosine` is provided in `exercises.c`. This function has two parameters, `x` and `n`, and has return type `double`. This function calculates and returns the cosine of angle `x` by calculating the first `n` terms of the series.

Finish the definition of this function. Your `cosine` function must call your `factorial` function.

Your `cosine` function must also call C's `pow` function. The prototype for this function is in header file `math.h`:

```
// Return x raised to the y power.
double pow(double x, double y);
```

Note that it's o.k. to pass integer arguments to `pow`. For example, if the second argument is an integer, C will convert this value to a `double` before assigning it to parameter `y`.

For this exercise, instead of using a *sput* test suite, we'll use a different approach to testing the function. The C standard library has a function named `cos`, so we'll compare the cosines calculated by this function with the values returned by your `cosine` function.

`main.c` contains a function named `test_cosine`. Here is the code that lets us check if `cosine` correctly calculates the cosine of 0 radians. It first calls C's `cos` function to calculate a correct approximation of cos 0. It then repeatedly calls your `cosine` function. The first time `cosine` is called, only the first term of the series is calculated. The second time `cosine` is called, two terms of the series are summed. During the final iteration, seven terms are summed. When you run this code and observe the output, you'll see how rapidly the value returned by `cosine` converges on the correct value (as returned by C's `cos` function).

```
printf("Calculating cosine of 0 radians\n");
printf("Calling standard library cos function: %.8f\n", cos(0));
printf("Calling cosine function\n");
for (int i = 1; i <= 7; i += 1) {
   printf("# terms = %d, result = %.8f\n", i, cosine(0, i));
}
printf("\n");
```

Notice that the character string argument in the fourth call to `printf` is:

```
"# terms = %d, result = %.8f\n"
```

When this string is displayed, the `%d` will be replaced by the value of variable `i` and the `%.8f` will be replaced by the value returned by `cosine`. `%.8f` specifies that this value should be formatted as a `double` (a real number), with 8 digits after the decimal point.

The test function calculates the cosines of 0 radians (0 degrees), Π /4 radians (45 degrees), Π/2 radians (90 degrees), and Π radians (180 degrees). For each of these values, we have `cosine` calculate 1 term of the series, 2 terms of the series, etc., all the way up to 7 terms.

Inspect the output produced by `test_cosine`. How close are the values returned by `cosine` to the values returned by `cos`?

What are the advantages and disadvantages of testing your `cosine` function using the approach followed in this exercise, compared to using a test framework like *sput*?