Carleton University
Department of Systems and Computer Engineering
SYSC 2006 - Foundations of Imperative Programming -Winter 2019

Lab 11 - Recursive Functions

Attendance/Demo

After you finish the exercises, a TA will review your solutions, ask you to run the test harness 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.**

General Requirements

You have been provided with three files:

- recursive_functions.c contains unfinished implementations of six recursive functions;
- recursive_functions.h contains the prototypes for those functions;
- main.c contains a simple *test harness* that exercises the functions in recursive_functions.c. Unlike the test harnesses provided in some of the labs, this one does not use the sput framework. As each test runs, the expected and actual results will be displayed on the console, along with a message indicating if the test passed. **Do not modify `main()` or any of the test functions.**

None of the functions you write should perform console input; i.e., contain `scanf` statements. Unless otherwise specified, none of your recursive functions should produce console output; i.e., 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). Instructions for selecting the formatting style and formatting blocks of code are in the Lab 1 handout.

Finish each exercise (i.e., write the function and verify that it passes all its tests) before you move on to the next one. Don't leave testing until after you've written all your functions.

Instructions

Step 1: Launch Pelles C and create a new Pelles C project named recursion. (Instructions for creating projects are in the handout for Lab 1.) If you're using the 64-bit edition of Pelles C, select Win 64 Console program (EXE) as the project type. If you're using the 32-bit edition of Pelles C, select Win32 Console program (EXE). **Don't click the icons for Console application wizard, Win32 Program (EXE) or Win64 Program (EXE) - these are not correct types for this project.**

Step 2: Download file main.c, recursive_functions.c and recursive_functions.h from cuLearn. Move these files into your recursion folder.

Step 3: Add main.c and recursive_functions.c to your project. (Instructions for doing this are in the handout for Lab 1.)

You don't need to add recursive_functions.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. Execute the project. The test harness will show that functions do not produce correct results (look at the output printed in the console window and, for each test case, compare the expected and actual results). This is what we'd expect, because you haven't started working on the functions that the test harness tests.

Step 6: Open recursive_functions.c and main.c in the Pelles C editor. Complete Exercises 1 - 3.

Exercise 1

File recursive_functions.c contains an incomplete definition of a function named `power` that calculates and returns x^n for $n >= 0$, using the following recursive formulation:

$$x^0 = 1$$

$$x^n = x * x^{n-1}, n > 0$$

The function prototype is:

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

Implement `power` as a recursive function. Your `power` function cannot have any loops, and it cannot call the `pow` function in the C standard library.

Read the definition of function `test_power` in main.c. function. Notice that `test_power` displays enough information for you to determine if your implementation of `power` is correct. Specifically, `test_power` prints:

- the name of the recursive function that is being tested (`power`);
- the values that are passed as arguments to `power`;
- the result we expect a correct implementation of `power` to return;
- the actual result returned by `power`;
- a short message indicating if the test passes or if there is an error.

Function `test_exercise_1` has five test cases for the `power` 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_power` five times, once for each test case.

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 `power` passes all the tests before you start Exercise 2.

Exercise 2

File recursive_functions.c contains an incomplete definition of a function named `count`. This function counts the number of integers in the first `n` elements of array `a` are that equal to `target`, and returns that count. The function prototype is:

```
int count(int a[], int n, int target);
```

For example, if array `arr` contains the 11 integers 1, 2, 4, 4, 5, 6, 4, 7, 8, 9 and 12, then `count(arr, 11, 4)` returns 3 because 4 occurs three times in `arr`.

Implement `count` as a recursive function. Your `count` function cannot have any loops. Hint: review the `sum_array` function that was presented in lectures (the lecture slides and C Tutor examples are posted on cuLearn.)

Read the definitions of `test_exercise_2` and `test_count` in main.c. Function `test_exercise_2` has six test cases for the `count` function. It calls `test_count` six times, once for each test case. Notice that `test_count` has four arguments: the three arguments that will be passed to `count`, and the value that a correct implementation of `count` will return.

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 `count` passes all the tests before you start Exercise 3.

Exercise 3

File recursive_functions.c contains an incomplete definition of a function named `occurrences`. This function counts the number of integers in the singly-linked list pointed to by `head` that are equal to `target`, and returns that count. The function prototype is:

```
int occurrences(node_t *head, int target);
```

For example, if the linked list pointed to by `list` contains the 11 integers 1, 2, 4, 4, 5, 6, 4, 7, 8, 9 and 12, then `occurrences(list, 4)` returns 3 because 4 occurs three times in the list.

Implement `occurrences` as a recursive function. Your `occurrences` function cannot have any loops. Hint: review the recursive linked list functions that were presented in lectures (C Tutor examples are posted on cuLearn.)

Function `test_exercise_3` has six test cases for the `occurrences` function. It calls `test_occurrences` six times, once for each test case. Notice that `test_occurrences` has three arguments: the two arguments that will be passed to `occurrences`, and the value that a correct implementation of `occurrences` will return.

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 `occurrences` passes all the tests before you start Exercise 4.

Exercise 4

File recursive_functions.c contains an incomplete definition of a function named `last`. This function the last element in a singly-linked list of integers. The function prototype is:

```
int last(node_t *head);
```

For example, if the linked list pointed to by `list` contains the 5 integers 1, 2, 4, 4, 6, 5, then `last(list)` returns 5 because 5 is the last element in the list.

The function must terminate (via `assert`) if it is passed an empty list.

Implement `list` as a recursive function. Your `list` function cannot have any loops.

Function `test_exercise_4` has four test cases for the `last` function. It calls `test_last` four times, once for each test case. Notice that `test_last` has two arguments: the argument that will be passed to `list`, and the value that a correct implementation of `list` will return.

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 `last` passes all the tests.

Wrap-up

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

2. Remember to backup 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.

Homework Exercise - Visualizing Program Execution

On the final exam, you will be expected to draw diagrams that depict the execution of recursive functions, using the same notation as C Tutor. This exercise is intended to help you develop your code tracing/visualization skills when working with recursive functions.

1. Launch C Tutor (the *Labs* section on cuLearn has a link to the website).

2. Copy your `power` function into C Tutor.

3. Write a short `main` function that calls `power`.

4. Use C Tutor to trace your program one statement at a time. To help you visualize the value returned by each recursive call, consider adding local variables to your function. One of the C Tutor examples on cuLearn illustrates how to do this for the recursive `factorial` function presented in class.

5. Repeat this exercise for your `count`, `occurrences` and `last` functions.

Extra Practice

Exercise 5

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 <u>cannot</u> have any loops.

Function `test_exercise_5` 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).

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 `num_digits` passes all the tests.

Exercise 6

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_6` 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 the project, correcting any compilation errors, then execute the project. The test harness will run. If you translated 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. 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?

How many recursive calls will your `power2` function make when calculating 3^{32}? 3^{19}? How much of an improvement is this, compared to the number of calls made by your `power` function from Exercise 1?

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