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

Lab 10 - Recursive Functions

Objective

To develop some simple recursive functions.

Attendance/Demo

After you finish all the exercises, call a TA, who 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 four 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. The harness doesn't compare the actual and expected results of each test and keep track of the number of tests that pass and fail. Instead, as each test runs, the expected and actual results will be displayed on the console, and you'll have to review this output to determine if your functions are correct.
  Part of the test harness has been written for you, but you will have to implement some of the test functions.

None of the recursive 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). Pelles C makes it easy to do this - instructions were provided in Labs 1 and 2.

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`.

- 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 folder named recursion.

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

Step 3: You must add main.c and recursive_functions.c to your project. To 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 for recursive_functions.c.

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. There won't be much output, because the functions in recursive_functions.c are incomplete, as are some of the test functions in main.c.

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 <u>cannot</u> have any loops, and it <u>cannot</u> call the `pow` function in the C standard library.

main.c contains a function named `test_power` that will test your `power` function. Read the definition of this function. Notice that `test_power` displays enough information for you to determine which function is being tested and whether or not the results returned by the function are 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`.

Function `test_exercise_1` has five test cases for your `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. Inspect the console output, and verify that your `power` function passes all the tests before you start Exercise 2.

Exercise 2

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_2` has seven test cases for your `num_digits` function. It calls the test function, `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). This test function has not been completed.

Finish the implementation of `test_num_digits`. The output displayed by `test_num_digits` should look like this:

```
Calling num_digits(k) with k = 5
Expected result: 1
Actual result: the value returned by your function

Calling num_digits(k) with k = 9
Expected result: 1
Actual result: the value returned by your function

Calling num_digits(k) with k = 10
Expected result: 2
Actual result: the value returned by your function
```

`....` *Output from remaining test cases not shown*

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Inspect the console output, and verify that your `num_digits` function 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 searches the first `n` integers elements of array `a` for occurrences of the specified integer `target`. The function prototype is:

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

The function returns the count of the number of integers in `a` that are equal to target. For example, if array `arr` contains the 11 integers 1, 2, 4, 4, 5, 6, 4, 7, 8, 9 and 12, then `occurrences(arr, 11, 4)` returns 3 because 4 occurs three times in `arr`.

Implement `occurrences` as a recursive function. Your `occurrences` function <u>cannot</u> have any loops. Hint: review the `sum_array` function that was presented in lectures (the lecture slides and code are posted on cuLearn.)

Function `test_exercise_3` has five test cases for your `occurrences` function. It calls the test function, `test_occurrences`, five times, once for each test case. Notice that `test_occurrences` has four arguments: the three arguments that will be passed to `occurrences`, and the value that a correct implementation of `occurrences` will return. This test function has not been completed.

Finish the implementation of `test_occurrences`. The output displayed by `test_occurrences` should look like this:

```
Calling occurrences with a = {1, 2, 4, 4, 5, 6, 4, 7, 8, 9, 12},
n = 11, target = 1
Expected result: 1
Actual result: the value returned by your function

Calling occurrences with a = {1, 2, 4, 4, 5, 6, 4, 7, 8, 9, 12},
n = 11, target = 2
Expected result: 1
Actual result: the value returned by your function

Calling occurrences with a = {1, 2, 4, 4, 5, 6, 4, 7, 8, 9, 12},
n = 11, target = 4
Expected result: 3
Actual result: the value returned by your function

     ....       Output from remaining test cases not shown
```

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Inspect the console output, and verify that your `occurrences` function passes all the tests.

Homework Exercise - Visualizing Program Execution

On the final exam, you may be asked 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 tests `power`.
4. *Without using C Tutor,* trace the execution of your program. Draw memory diagrams that depict the program's activation frames just before the `return` statement in `power` is executed. Because `power` is called recursively, there will be one diagram for each call. Use the same notation as C Tutor.
5. Use C Tutor to trace your program one statement at a time, stopping just before each `return` statement is executed. Compare your diagrams to the visualization displayed by C Tutor.
6. Repeat this exercise for your `num_digits` and `occurrences` functions.

Extra Practice

Exercise 4

How many recursive calls will your `power` function from Exercise 1 make when calculating 3^{32}? 3^{19}?

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. Your `power2` function <u>cannot</u> have any loops, and it <u>cannot</u> call the `pow` function in the C standard library or the `power` function you wrote for Exercise 1.

Hint: the most obvious solution involves translating the recursive formulation directly into C, but you may find that this implementation of `power2` performs recursive calls "forever". If this happens, 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, you will need to change the recursive formulation slightly.

Function `test_exercise_4` 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 the test function, `test_power2`, five times, once for each test case. This test function has not been completed. Using `test_power` as a model, finish the implementation of `test_power2`. The output displayed by `test_power2` should look like this:

```
Calling power2(x, k) with x = 3.50, k = 0
Expected result: 1.00
Actual result: the value returned by your function

Calling power2(x, k) with x = 3.50, k = 1
Expected result: 3.50
Actual result: the value returned by your function
```

`....` *Output from remaining test cases not shown*

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Inspect the console output, and verify that your `power2` function passes all the tests.

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?

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