Carleton University
Department of Systems and Computer Engineering
SYSC 2006 - Foundations of Imperative Programming - Winter 2014

Lab 11 - Recursive Functions

Objective

To learn how to develop recursive functions.

Attendance/Demo

To receive credit for this lab, you must make the effort to finish a reasonable number of exercises and demonstrate the code you complete. **Also, you must submit your lab work to cuLearn by the end of the lab period**. (Instructions are provided in the *Wrap Up* section at the end of this handout.)

When you have finished all the exercises, call a TA, who will review the code you wrote. For those who don't finish early, a TA will ask you to demonstrate whatever code you've completed, starting about 30 minutes before the end of the lab period. **Any unfinished exercises should be treated as "homework" and should be completed before the end of term.**

General Requirements

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 your functions.

None of the functions you write should perform console input; i.e., contain `scanf` statements. Unless otherwise specified, none of your functions should produce console output; i.e., contain `printf` statements.

You have been provided with three files:

- recursive_functions.c contains unfinished implementations of four recursive functions;
- recursive_functions.h contains the function 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 previous labs, this one does not use the sput framework. The test code doesn't compare the actual and expected results of each test and keep track of the number of tests that pass and fail. Instead, the expected and actual results will be displayed on the console, and you 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.

Instructions

1. Create a new folder named Lab 11.
2. Launch Pelles C and create a new Pelles C project named recursion inside your Lab 11 folder. The project type must be Win32 Console program (EXE). You should now have a folder named recursion inside your Lab 11 folder (check this).
3. Download file main.c, recursive_functions.c and recursive_functions.h from cuLearn. Move these files into your recursion folder.

4. You must add main.c and recursive_functions.c to your project. From the menu bar, select Project > Add files to project... 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.

5. Build the project. It should build without any compilation or linking errors.

6. 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

Exercise 1

File recursive_functions.c contains an incomplete implementation 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. Open main.c in the editor and read `test_power`. 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`.

The `main` function 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 implementation of a function named `power2` that calculates and returns x^n for $n >= 0$, using the following recursive formulation:

$$x^0 = 1$$

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

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

The function prototype is:

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

Implement `power2` as a recursive function. 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.

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.

The `main` function 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 before you start Exercise 3.

Exercise 3

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

How many recursive calls will your `power2` function from Exercise 2 make when calculating 3^{32}? 3^{19}?

Exercise 4

File recursive_functions.c contains an incomplete implementation 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);
```

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

Hint: if *n* < 10, it has one digit. Otherwise, it has one more digit than the integer *n* / 10. (Recall that dividing an integer by an integer yields an integer).

The `main` function 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. 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 5.

Exercise 5

File recursive_functions.c contains an incomplete implementation 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 `a` contains the 11 integers 1, 2, 4, 4, 4, 5, 6, 7, 8, 9 and 12, then `occurrences(a, 11, 4)` returns 3 because 4 occurs three times in `a`.

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

The `main` function 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, 4, 5, 6, 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, 4, 5, 6, 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, 4, 5, 6, 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.

Wrap-up

1. Remember to have a TA review and grade your solutions to the exercises before you leave the lab.
2. The next thing you'll do is package the project in a ZIP file (compressed folder) named recursion.
   - From the menu bar, select Project > ZIP Files... A Save As dialog box will appear. If you named your Pelles C project recursion, the zip file will have the same name by default; otherwise, you'll have to edit the File name: field and rename the file to recursion before you save it. **Do not use any other name for your zip file** (e.g., lab11.zip, my_project.zip, etc.).
   - Click Save. Pelles C will create a compressed (zipped) folder, which will contain copies of the the source code and several other files associated with the project. (The original files will not be removed). The compressed folder will be stored in your project folder (i.e., folder recursion).
3. Log in to cuLearn and submit recursion.zip.
   - Click the Submit Lab 11 link. After you click the Add submission button, drag recursion.zip to the File submissions box. Do not submit another type of file (e.g., a Pelles C .ppj file, RAR file, a .txt file, etc.)
   - After the icon for the file appears in the box, click the Save changes button. At this point, the submission status for your file is "Draft (not submitted)". You can resubmit the file by clicking the Edit my submission button.
   - Once you're sure that you don't want to make any changes, click the Submit assignment button. This will change the submission status to "Submitted for grading". **Note: after you've clicked the Submit assignment button, you cannot resubmit the file.**

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