Carleton University
Department of Systems and Computer Engineering
SYSC 2006 - Foundations of Imperative Programming - Winter 2016

Lab 11 - Recursive Functions

Objective

To learn how to develop some simple recursive functions.

Attendance/Demo

To receive credit for this lab, you must demonstrate your solutions to the exercises. **Also, you must submit your lab zipped project file to cuLearn**. (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"; you should complete these before the final exam.**

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 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 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 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, 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.

Instructions

1. Launch Pelles C and create a new Pelles C project named recursion (all letters are lowercase). The 64-bit version of Pelles C is installed on our lab computers, so the project type must 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 your own computer, the project type should be Win 64 Console program (EXE) or Win32 Console program (EXE), depending on whether you installed the

64-bit or 32-bit edition of Pelles C. After creating the project, you should have a folder named recursion. Check this. If you do not have a project folder named recursion, close this project and repeat Step 1.

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

3. 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 `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. (Hint: 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`.)

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. Your `num_digits` function <u>cannot</u> have any loops.

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 (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 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. Hint: review the `sum_array` function that was presented in lectures (the lecture slides and code are posted on the cuLearn course page.)

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, assign a grade (Satisfactory, Marginal or Unsatisfactory) and have you initial the demo/sign-out sheet.

2. The next thing you'll do is package the project in a ZIP file (compressed folder) named recursion.zip. To do this:

   2.1. 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 this 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.).

   2.2. 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. To do this:

3.1. Click the Submit Lab 11 link. A page containing instructions and your submission status will be displayed. After you've read the instructions, click the Add submission button. A page containing a File submissions box will appear. Drag recursion.zip to the File submissions box. **Do not submit another type of file (e.g., a Pelles C .ppj file, a RAR file, a .txt file, etc.)**

3.2. After the icon for the file appears in the box, click the Save changes button. At this point, the submission status of your file is "Draft (not submitted)". If you're ready to finish submitting the file, jump to Step 3.4. If you aren't ready to do this; for example, you want to do some more work on the project and resubmit it later, you can leave the file with "draft" submission status. When you're ready to submit the final version, you can replace or delete your "draft" file submission by following the instructions in Step 3.3, then finish the submission process by following the instructions in Step 3.4.

3.3. You can replace or delete the file by clicking the Edit my submission button. The page containing the File submissions box will appear.

3.3.1. To overwrite a file you previously submitted with a file having the same name, drag another copy of the file to the File submissions box, then click the Overwrite button when you are told the file exists ("There is already a file called..."). After the icon for the file reappears in the box, click the Save changes button.

3.3.2. To delete a file you previously submitted, click its icon. A dialogue box will appear. Click the Delete button., then click the OK button when you are asked, "Are you sure you want to delete this file?" After the icon for the file disappears, click the Save changes button.

3.4. Once you're sure that you don't want to make any changes, click the Submit assignment button. A Submit assignment page will be displayed containing the message, "Are you sure you want to submit your work for grading? You will not be able to make any more changes." Click the Continue button to confirm that you are ready to submit your lab work. This will change the submission status to "Submitted for grading".

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

$x^0 = 1$

$x^n = (x^{n/2})^2$, $n > 0$ 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 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.

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