Carleton University
Department of Systems and Computer Engineering
SYSC 2006 - Foundations of Imperative Programming - Fall 2022

Lab 12 - Recursive Functions

Note that if you are using a different IDE, e.g. CLion, you need to adjust the Pelles C instructions as required.

General Requirements

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 of 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.) Select Win 64 Console program (EXE) as the project type. **Don't click the icons for Console application wizard, Win64 Program (EXE) or any of the other "Empty projects" icons. These are not correct types for this project.**

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

Step 2: Download lab12.zip from Brightspace. Open lab12.zip and locate the files named main.c, main.c, recursive_functions.c and recursive_functions.h. 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.

As you add the files, icons labelled main.c and recursive_functions.c will appear in the Project window, below the Source files icon. An icon labelled recursive_functions.h will appear below the Include files icon.

- 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 several 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.**

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

Step 5: Execute the project. The test harness will show that the 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: Complete Exercises 1 - 4. If you become "stuck" while working on the exercises, consider using C Tutor to help you discover the problems in your solutions.

Exercise 1

Open recursive_functions.c and main.c in the Pelles C editor.

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.

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 and execute the project. Use the console output to help you identify and correct any flaws. Verify that `power` passes all of its tests before you start Exercise 2.

Exercise 2

File recursive_functions.c contains an incomplete definition of a function named `count_in_array`. The function prototype is:

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

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

Implement `count_in_array` as a recursive function. Your `count_in_array` function <u>cannot</u> have any loops. Hint: review the recursive `sum_array` function that was presented in lectures (the lecture slides and the C Tutor link are posted on Brightspace.) We recommend that, before writing any code. you formulate a recursive definition of the solution to the problem, "What is the number of integers in the first `n` elements of array `a` that are equal to `target`?" Then, convert that definition to C code. We also recommend that you use an iterative, incremental approach when implementing the function. For example, during the first iteration, write just enough code to handle the base case(s). Use the console output to help you identify and correct any flaws. When your function passes the tests for this case, write the code recursive case(s).

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

Build and execute the project. Use the console output to help you identify and correct any flaws. Verify that `count_in_array` passes all of its tests before you start Exercise 3.

Exercise 3

File recursive_functions.c contains an incomplete definition of a function named `count_in_sll`. The function prototype is:

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

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

Implement `count_in_sll` as a recursive function. Your `count_in_sll` function <u>cannot</u> have any loops. Hint: review the recursive `length` function that was presented in lectures (the lecture slides and the C Tutor links are posted on Brightspace.) We recommend that, before writing any code. you formulate a recursive definition of the solution to the problem, "What is the number of integers in the singly-linked list pointed to by `head` that are equal to `target`?" Then, convert that definition to C code. We also recommend that you use an iterative, incremental approach when implementing the function. For example, during the first iteration, write just enough code to handle the base case(s). Use the console output to help you identify and correct any flaws. When your function passes the tests for this case, write the code recursive case(s).

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

Build and execute the project. Use the console output to help you identify and correct any flaws. Verify that `count_in_sll` passes all of its tests before you start Exercise 4.

Exercise 4

File recursive_functions.c contains an incomplete definition of a function named `last_in_sll`. The function prototype is:

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

This function returns the last element in the singly-linked list of integers pointed to by `head`. For example, if the linked list pointed to by `head` contains the 5 integers 1, 2, 4, 4, 6, 5, then `last_in_sll(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 `last_in_sll` as a recursive function. Your `last_in_sll` function cannot have any loops. We recommend that, before writing any code. you formulate a recursive definition of the solution to the problem, "What is the last element in the singly-linked list pointed to by `head`?" Then, convert that definition to C code. We also recommend that you use an iterative, incremental approach when implementing the function. For example, during the first iteration, write just enough code to handle the base case(s). Use the console output to help you identify and correct any flaws. When your function passes the tests for this case, write the code recursive case(s).

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

Build and execute the project. Use the console output to help you identify and correct any flaws. Verify that `last_in_sll` passes all of its tests.

Wrap-up

Get your lab marked during your lab by a TA. Submit recursive_functions.c to Brightspace **immediately** after the TA has marked it. (Do not wait for the deadline as the TA may input a grade of zero if your work is not there by the end of your lab!)

Before submitting your lab work

- Make sure that recursive_functions.c has been formatted to use K&R style or BSD/Allman style, as explained in *General Requirements*.
- Ensure you're submitting the file that contains your solutions, and not the unmodified file you downloaded from Brightspace!
- **Remember that your mark will be 0 if a TA did not check your work in the lab or if you did not submit your final version immediately after the TA checked your work.**

Homework Exercise - Visualizing Program Execution

In the final exam, you will be expected to be able to understand and 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. Click on this link to open C Tutor.
2. Copy/paste your `power` function into C Tutor.
3. Write a short `main` function that calls `power`. Feel free to borrow code from this lab's test harness.
4. *Without using C Tutor*, trace the execution of your program. Draw memory diagrams that depict

the program's activation frames as `power` is called recursively. Use the same notation as C Tutor.

5. Use C Tutor to trace your program one statement at a time, stopping just before your function returns. To help you visualize the value returned by each recursive call, consider adding local variables to your function. (See the lecture slides for examples.) If C Tutor complains that your program is too long, delete the comments above the function definition. Compare your diagrams to the visualizations displayed by C Tutor.

6. Repeat this exercise for your `count_in_array`, `count_in_sll` and `last_in_sll` functions. For the last two functions, you'll need to copy the declaration of `node_t` from recursive_functions.h into C Tutor.

Extra Practice (you do not need to submit solutions for these exercises)

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 and execute the project.Use the console output to help you identify and correct any flaws. Verify that `num_digits` passes all of its 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 and execute the project. If you translate 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. Hint: 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

History

Nov. 5, 2022: Initial release.