CS 305 Lab 5: Recursion
Fall 2019

The purpose of this lab is to give you additional experience with recursion, a topic that was introduced in CS 203. You will be examining and completing the following recursive functions:

- Testing a string for being a palindrome.
- Multiplication, using the *Russian Peasant's Algorithm.*
- Print a number in a given base.

Recall that recursive functions are written such that recursive calls of the function lead toward base case(s). For each of the recursive functions in this lab, either the base case or the recursive case (or both) is incomplete. You job will be to complete the function and then examine the stack upon execution.

This lab has a total of 100 possible points (30 points from pre-lab and 70 points from lab).

Sit with your assigned partner.

Objectives

Upon completion of the laboratory exercise, you will be able to do the following:

- Identify the base case(s) and recursive case(s) for the recursive functions listed above.
- Understand why each function terminates.
- Use the debugger to examine the stack for recursive function calls.

Part 1: Logging in (choose one person)

1. Follow the procedure from prior labs to log in and open Mobaxterm:

2. Go into your cs305 directory. Make a new lab5 directory:

```
cd cs305
mkdir lab5
```

3. Get the lab5 files. Download `starter_lab5.zip` from moodle to your cs305 folder. Hopefully, you can see the zip file. To unzip it, type

```
unzip starter_lab5.zip
```

There should be three .c files.

Part 2: Palindrome test (`palindrome.c`) – Fix the recursive case

1. Examine the `is_palindrome` function in `palindrome.c`. This function is supposed to determine whether a string is a palindrome (i.e., whether it reads the same backward as forward).
The function takes three parameters:

- a string
- the index in the string of the first character to be considered to be part of the palindrome
- the index in the string of the last character in the string that is to be considered part of the palindrome

2. The algorithm is as follows:
    - base case #1 (**already done**): the first index is greater than or equal to the last index. In this case we have a one-character or zero-character string, which is a palindrome, so return 1.
    - base case #2 (**already done**): the first index is less than the last index, but the respective characters do not match. In this case return 0.
    - recursive case (**you need to do**): the first index is less than the last index, but the values at these character positions are equal. We have a palindrome only if we get a palindrome after discarding the first and last character of the string, so we can simply return the recursive call to `is_palindrome` on this shorter string (shorter by 2 characters).

The algorithm should always terminate, because at each recursive call, we are calling `is_palindrome` on a shorter substring.

3. Modify the recursive case so that it runs correctly. All other code should remain the same. Compile and run the program.

```
gcc -g -o palindrome palindrome.c
./palindrome
```

Once you are confident it is working, use the debugger to run the program. Set a breakpoint at the first line of code inside is_palindrome.

```
gdb palindrome
(gdb) run
```

Find the first line of code inside `is_palindrome`. Set a breakpoint there using the `break` command. Then, use a combination of `cont` and `info stack` to view the stack contents each time the breakpoint is hit (each time the function is recursively called).

1. What are the call stack values for first_index and last_index when running the program on input string "`racecar`" when the call stack is the largest?

first_index=_____, last_index= _____
// continue list here

2. What is the call stack values when running the program on input string "`barbed`"?
first_index=_____, last_index= _____
// continue list here

Checkpoint 1 [25 points]: Show your lab instructor/assistant your program running with various strings and show your answers to the questions above.

Part 3: Multiplication (`multiply.c`) – Fix the recursive case (switch driver)

1. Examine the `multiply` function in `multiply.c`, which is intended to perform multiplication using the *Russian Peasant's Algorithm*. This algorithm multiplies two integers as follows:

- Base case: (**already done**) if the first integer is zero, then the result is zero.
- Base case: **(already done)** If one or both are negative numbers, then recursively call `multiply` on the negated values.
- Recursive case: (**you need to do**) If the first integer is even, the result is the result of recursively multiplying half of the first number by twice the second. For example, you can multiply 24 times 7 by multiplying 12 (half of 24) by 14 (twice 7); this results in the value 168. If the first integer is odd, the result is the result of recursively multiplying half of the first number (rounded down) by twice the second, and then adding the second number to compensate for the rounding down. For example, you can multiply 25 times 7 by multiplying 12 (half of 25) by 14 (twice 7), and then adding 7; this results in the value 175.

2. To see this algorithm in action, consider multiplying 13 times 20:

- 13*20 is 6*40 + 20
    - 6*40 is 3*80
        - 3*80 is 1*160 + 80
            - 1*160 is 0*320 + 160
                - 0*320 = 0 (base case)
            - so 1*160 is 0 + 160 = 160
        - so 3*80 is 160 + 80 = 240
    - so 6*40 = 240
- so 13*30 is 240 + 20 = 260

Explain why this function will always terminate:

3. The starter file is written so that the base case is correct and some of the recursive cases are correct, but the last recursive case is incomplete. Modify the code so that the program runs correctly. All other code should remain the same. Compile and run your program.

```
gcc –o multiply multiply.c
./multiply
```

Checkpoint 2 [25 points]: Show your lab instructor/assistant your program running with various pairs of non-negative integers, including cases where one or both are zero. Show your answer to the question above.

Part 4: Number-printing (`print_num.c`) – Fix the base cases (switch driver)

1. Examine the `print_num` function in `print_num.c`. This function is intended to print a number in any given base between 2 and 36. The `main` function reads an integer and prints it in all bases between 2 and 36. The letters *a* through *z* are used to represent the digits 10 through 35, respectively.
The function takes two arguments:
- the number to print
- the base to print the number in

2. Its intended operation is as follows:
- base case 1 (**you need to do**) the number is negative, print a – character, and then recursively call the function with the negated number (which becomes a positive number)
- base case 2 (**you need to do**): if the number is less than the base, print its digit, using the `digits` array. Hint: be sure to print using formatting code `%c` for character.
- recursive case (**already done**): if the number is greater than or equal to the base
  - recursively print the value of the number divided by the base. This should print out all the digits except the last.
  - recursively print the value of the number remaindered by the base. This should print out the last digit.

3. For example, to print 173 in base 10:
- recursively print 173/10: this prints "17" by recursively calling print_num(17, 10)
- recursively print 173%10: this prints "3"

The result is that "173" is printed.

4. This function always terminates because:
- a negative value always results in a recursive call with a positive value. (We'll ignore `MIN_INT` for now)
- a positive value always results in either the base case, or recursive calls with smaller values.

5. Modify the base cases so that the program runs correctly. All other code should remain the same. Compile and run your program.

```
gcc -g -o print_num print_num.c
./print_num
```

If you want the output to go to a file, use:

```
./print_num > out.txt
```

(nothing will print to the screen, but you can still type the number here. Do `less out.txt` to see the contents of the file)

Checkpoint 3 [20 points]: Show your lab instructor/assistant your code and program running with 121.

Close Mobaxterm and any other applications. Log off the computer.

If any checkpoints are not finished, they are due in one week. You submit screenshots, code files, and answers to questions electronically (on moodle).