Lab : Recursion

Information

In-class labs are meant to introduce you to a new topic and provide some practice with that new topic.

Topics: Writing simple recursive functions

Solo work: Labs should be worked on by each individual student, though asking others for help is permitted. Do not copy code from other sources, and do not give your code to other students. **Students who commit or aid in plagiarism will receive a 0% on the assignment and be reported.**

Building and running: If you are using Visual Studio, make sure to run with debugging. **Do not run without debugging!**

Using the debugger will help you find errors.
To prevent a program exit, use this before `return 0;`

```
cin.ignore();      cin.get();
```

Turn in: Once you're ready to turn in your code, prepare the files by doing the following: **(1)** Make a copy of your project folder and name it `LASTNAME-FIRSTNAME-LABNAME`. (Example: `HOPPER-GRACE-LAB-UNIT-TESTS`) **(2)** Make sure that all source files (.cpp, .hpp, and/or .h files) and the `Makefile` files are all present. **(3)** Remove all Visual Studio files - I only want the source files and Makefiles. **(4)** Zip your project folder as `LASTNAME-FIRSTNAME-LABNAME.zip`

Never turn in Visual Studio files!

Starter files: Download from GitHub.

Grading: Grading is based on completion, if the program functions as intended, and absense of errors. **Programs that don't build will receive 0%.** Besides build errors, runtime errors, logic errors, memory leaks, and ugly code will reduce your score.

Contents

1.1 About

Recursion can be hard to think in terms of, especially early on. For this lab, we will step through some recursive functions, and then you will be challenged to try to design solutions for the others.

Reference the textbook, reading notes, and lectures for help.

1.1.1 Setting up the project

Download the starter zip file, `LAB-RECURSION-INTRO.zip`, off GitHub. This zip contains the following:

- `function1.hpp`
- `function2.hpp`
- `function3.hpp`
- `function4.hpp`
- `function5.hpp`
- `program_main.cpp`
- `Makefile`

All of these sources belong in one project and are not separate programs.

The main program has a main menu to allow the user to run one function at a time. Each of the function files contains stubs for iterative and recursive functions.

1.2 Lab specifications

1.2.1 Function 1: CountUp

```
1 void CountUp_Iter( int start, int end )
2 {
3 }
4
5 void CountUp_Rec( int start, int end )
6 {
7 }
```

For this first set of functions, you will write each of these so that they display the numbers between `start` and `end` (inclusive), going up by 1 each time. The output will look like this:

```
CountUp, Iterative:
1   2   3   4   5   6   7   8   9   10

CountUp, Recursive:
1   2   3   4   5   6   7   8   9   10
```

Iterative version

The iterative version is pretty easy - write a for-loop and display each number.

Recursive version

For the recursive version, follow these steps:

- Display `start`, plus one tab, with a `cout` statement.
- **Terminating case:** If the value of `start` and `end` are the same, then `return;` out of the function.
- **Recursive case:** Call `CountUp_Rec`, passing in `start+1` and `end`.

Once you get it working, move on to the next function.

1.2.2 Function 2: MultiplyUp

```
1 void MultiplyUp_Iter( int start, int end )
2 {
3 }
4
5 void MultiplyUp_Rec( int start, int end )
6 {
7 }
```

This function is similar to CountUp, except it multiplies the value each time. The output will end up looking like this:

```
MultiplyUp, Iterative:
2   4   16  256

MultiplyUp, Recursive:
2   4   16  256
```

Terminating case: Stop once `start` is greater than `end`. You can use `return;` to leave a function, even though its return-type is void.

Recursive case: Call the function again, passing in $start * start$ and end.

1.2.3 Function 3: Factorial

```
1 int Factorial_Iter( int n )
2 {
3     return -1; // placeholder;
4 }
5
6 int Factorial_Rec( int n )
7 {
8     return -1; // placeholder
9 }
```

For these functions, some value n is passed in. Each of these will calculate $n!$ by computing $n * (n-1) * (n-2) * ... * 2 * 1$. For the recursive case, we can think of the calculation recursively:

$$n! = n * (n-1)!$$
and
$$n! = n * (n-1) * (n-2)!$$

Terminating case: End once $n = 1$, returning the value of n. (At this point, we don't need to go past 1 in our calculation.)

Recursive case: Multiply n by $(n-1)!$
How do you calculate $(n-1)!$? You call `Factorial_Rec` again, but passing in $n-1$.

```
Factorial, Iterative:

2! = 2       3! = 6
4! = 24      5! = 120
6! = 720         7! = 5040
8! = 40320       9! = 362880

Factorial, Recursive:

2! = 2       3! = 6
4! = 24      5! = 120
6! = 720         7! = 5040
8! = 40320       9! = 362880
```

1.2.4 Function 4: CountConsonants

```
1 int CountConsonants_Iter( string text )
2 {
3     return -1; // placeholder
4 }
5
6 int CountConsonants_Rec( string text, int pos )
7 {
8     return -1; // placeholder
9 }
```

For CountConsonants, it will iterate through each letter in the `text` parameter, incrementing a counter each time it finds a consonant.

You can use the included function, `bool IsConsonant( char letter )` to check each letter.

You can also treat a string as an array of chars, so accessing `text[0]` would give you the first letter. You can also access the string's size with `text.size()`. For instance, if you wanted to display each letter from a string `text`, you could use:

```
for ( int i = 0; i < text.size(); i++ )
    cout << text[i] << endl;
```

Terminating case: If you're at the end of the string (that is, `pos = text.size()`), then return 0.

Recursive case: You will have 1 if the current letter (`text[pos]`) is a consonant, and 0 if the current letter is not. Add this value to the consonants You will add `CountConsonants_Rec`, passing in text and $pos + 1$.

```
GetConsonants, Iterative:
 * Consonants in aeiou: 0
 * Consonants in kittens: 5
 * Consonants in development: 7


GetConsonants, Recursive:
 * Consonants in aeiou: 0
 * Consonants in kittens: 5
 * Consonants in development: 7
```

1.2.5 Function 5: GetFirstUppercase

```
1 char GetFirstUppercase_Iter( string text )
2 {
3     return ' '; // placeholder
4 }
5
6 char GetFirstUppercase_Rec( string text, int pos )
7 {
8     return ' '; // placeholder
9 }
```

For this function, it will go through the entire text string. If it encounters an uppercase letter, it will return that letter. If the entire string is searched and no uppercase letter is found, it will return a space char: ' '.

For the recursive version, there will be two terminating cases: if you find an uppercase letter (then return it), and if you get to the end of the string (return ' ').

The recursive case should be similar to with the previous function.

```
GetFirstUppercase, Iterative:
 * First upper-case in how are YOU?: 'Y'
 * First upper-case in What?: 'W'
 * First upper-case in where am I?: 'I'
 * First upper-case in no caps: ' '


GetConsonants, Recursive:
 * First upper-case in how are YOU?: 'Y'
 * First upper-case in What?: 'W'
 * First upper-case in where am I?: 'I'
 * First upper-case in no caps: ''
```