Assignment Recursion: Understanding and Writing Recursive Functions

COSC 2336: Data Structures and Algorithms

Summer 2021

Objectives

- Practice writing functions.
- More practice with classes and arrays in C++, including passing in class instances to functions and declaring and using instances of classes.
- Practice writing recursive functions.
  - Identify and write base cases for recursion
  - Identify and write general recursive case breaking problem into sub-problem(s)
- Compare iterative vs. recursive implementation of functions.
- Learn to define base case and general case for recursion.
- Continue practicing test driven development and git project workflow

Description

In this assignment your primary goal is to learn about writing recursive functions. As you should have seen in this unit's materials, a recursive function is a function that calls itself. But for recursion to work, the call to itself has to be done in a particular way. Recursive functions work by specifying a problem in terms of 1 (or more) smaller sub-problems. The recursive call is on the smaller problem. Eventually the problem will be small enough that it can be solved directly, this is called the base case. The case where the problem is still not small enough to solve is the general case, where you have to break up the problem into a smaller sub-problem (or multiple smaller sub-problems). This in general is the most important aspect of writing a recursive function, identifying the base case, and correctly specifying the general case to break a problem into 1 or more smaller sub-problems.

In this assignment you will be writing 3 separate functions. But for each function, you will be writing both an iterative version and a recursive version. An iterative version simply means using iteration, like a loop, instead of recursion to solve the problem.

We will also be further looking at using Classes as we did in the previous unit. In this assignment we are using a user defined type, that has been defined for you, called the `List` class, which holds a simple list of integer values. This class is defined in the `List.[hpp|cpp]` header and implementation files. You should familiarize yourself with how this class works before beginning your assignment. The instructor should go over using this class a bit before this assignment starts. The first set of unit tests in this assignment are actually testing the `List` class, and will be initially uncommented for you, so you can examine them as well for ideas on how the `List` type is used.

You will be writing three different functions that work on this `List` type. The first will simply sum up the values in the list and return this calculated sum. The second function will reverse the order of the values in the list. The third function will be one that checks if the list is a palindrome or not. We usually think of words as palindromes, but the concept is the same here for a list of integers. If we have a list, say `[ 1 2 3 2 1 ]` we can describe this as a palindrome, because the values are the same whether we go forwards or backwards through the list. However the list `[ 1 2 3 2 ]` is not a palindrome by this definition.

Overview and Setup

For this assignment you will be given the following files that you will be using and adding code to for this assignment.

File Name	Description
`src/assg-tests.cpp`	Unit tests for the recursive functions class you are to implement.
`include/List.hpp`	Header file for the declarations of the `List` class you will be using and its defined API.
`src/List.cpp`	Implementation file for the `List` member functions that implement the API and class functionality.
`include/assg-recursion.hpp`	Header file for function prototypes you will be implementing in this assignment.
`src/assg-recursion.cpp`	Implementation files where you will write your code for this assignment.

You will not be modifying any code in the `List.[hpp|cpp]` files, you will only be using this class. All of your code will be written in the `assg-recursion.[hpp|cpp]` files. This week, though you are using a class, you will be writing just regular C functions again, not class member functions.

As usual, before starting on the assignment tasks proper, you should make sure you have completed the following setup steps.

1. Copy the assignment repository on GitHub using the provided assignment invitation link for 'Assignment Recursion' for our current class semester and section.
2. Clone the repository using the SSH url to your local class DevBox development environment.
3. Checkout the 'origins/feedback' branch to your locak working DevBox repository.
4. Configure the project by running the `configure` script from a terminal.
5. Confirm that the project builds and runs, though no tests will be defined or run initially. If the project does not build on the first checkout, please inform the instructor.
6. You should create the issue for Task 1 and/or for all tasks for the assignment now before beginning the first task. On your GitHub account, go to issues, and create it/them from the issue templates for the assignment. Also you should close the initial Pull request that should be automatically opened for you, so that you can open your own when commiting your work.

Assignment Tasks

Task 1: Implement `sumIterative()` Function

If you haven't already, create the Task 1 issue on GitHub and close the initial Pull request if it is still open.

For the first task we want you to write a function that will take in an instance of the given `List` user defined type, and sum up the values of the list and return this sum. You are to implement this task as a regular iterative function, so you will need a `for` loop that loops through the values of the list, summing them up and returning this sum (it will be considered incorrect to use recursion in this first function).

In the test file as usual there are test cases for each of the tasks of the assignment. However, the first test case this time is one that tests the functionality of the `List` data type. It will be already uncommented, and all of these tests should be passing (and should remain passing as you implement your assignment code). You should examine the tests and how the `List` data type is used in this first set of tests to understand how you will use this class in your own functions.

Start by uncommenting the next test case with the set of tests for the `sumIterative()` function. As usual you are encouraged to practice incremental programming. Uncomment this test case, create the function prototype in the `assg-recursion.hpp` header file, and create an empty stub function that simply returns a 0 result. This should allow you to have a program that compiles and runs the tests cases for your first function, though the tests will not be passing yet.

As another wrinkle, you have not been given the function documentation for the function you are to write in the `assg-recursion.cpp` header file. You are required to write correct function documentation for your functions this week. This includes a short title, a longer description, and documenting the input parameters and return value using `@param` and `@returns` tags respectively. The instructor should go over creating the function documentation correctly before this assignment.

The `sumIterative()` function takes 3 parameters. The first should be a reference to a `List` data type, e.g. `List&`. The second and third are actually `begin` and `end` indexes of the list. It is more correct to say that this function sums up the sub-list of values from the `begin` index to the `end` index of the list. So for example, if the list of values is currently `[ 3 5 2 4 8 1 ]` this list has 6 values, and the valid indexes of the list are from 0 to 5. If you want to sum up the whole list, you would call `sumIterative(list, 0, 5)`, because begin and end are inclusive, so this should sum all the values from index 0 to index 5 inclusive, for a result of 23. But you can also sum up a sub portion of the list, say `sumIterative(list, 1, 3)` which should sum up the values at indexes 1, 2 and 3 for a result of 11.

So your function should take the 3 parameters described, and return an integer result, the sum of the sub-list it is asked to sum. Don't forget to document all 3 of these parameters with `@param` tags and the return value with an `@returns` tag. Also remember that you should be using a loop to sum up the values, you will be required to use a recursive algorithm in the next task.

When you are satisfied with your work and you can compile and pass the first set of test in the test case for the `sumIterative()` function, commit your changes and push them to the repository using the `feedback` branch. Then document this task completion with a pull request and merge the changes to the main branch.

Task 2: Implement `sumRecursive()` Function

Make sure you have created Issue 2 for this task. And it would be best to close and merge your first pull request before continuing on to Task 2.

For the second task you should implement a recursive version of summing a `List` of values. The function should be named `sumRecursive()`, but it will have the same input parameters and return value as the previous function. Don't forget to update the function documentation if needed if you mention the algorithm used in the implementation of the function, though the parameters and return value can probably be the same for both of these functions.

To successfully implement the summing task using recursion, you will need to specify a base case and a recursive case correctly. The base case(s) should be the following

1. If the sub-list that is asked to be summed is size 0, return a sum of 0.
2. If the sub-list is of size 1, return the value as the sum.

For example, lets again say your list has the values `[ 1 3 2 5 ]`. I can ask you to sum up a sub-list of size 1, say `sumRecursive(list, 1, 1)`. When `begin` and `end` indexes are equal then the sub-list is of size 1. Here the value at index 1 of the list is a 3, so the sum returned should be 3.

But I could also ask you to sum up the list `sumRecursive(list, 1, 0)`. Whenever the end is less than the beginning, then the list is empty, so the sum should be 0 in this case.

Once you have the base cases working you can then implement the recursive case. However, you should try and implement the base cases first. Several of the unit tests test for these base cases where the sub-list is empty or of size 1, and you should be able to pass these tests by adding in these 2 base cases.

The recursive case can be done in more than 1 way. The easiest idea is that the sum of a list is the value at the `begin` index, plus the sum of values of the sub-list from begin + 1 to end. This is an example of tail recursion.

Implement a recursive case, and test if your sum function using recursion works to pass the given unit tests. Once you have implemented your recursive function and you have gotten your tests to pass, commit your changes and push them to the `feedback` branch. Then close out your Task #2 and pull request to documentation the completion of this task.

Task 3: Implement `reverseIterative()` Function

Make sure that you have created Task 3 and are ready for a new pull request before beginning this next task.

Our next two tasks will work to reverse the `List` of values in place. So for example, if the list is `[ 1 2 3 4 ]`, then calling reverse on the list should reverse the values to be `[ 4 3 2 1 ]`.

Uncomment the test case with the `reverseIterative()` tests, and as usual you should create the function prototype and a stub function, and make sure the code compiles and the tests run.

The input parameters will still be the same for this function as the previous two. These two functions will both be `void` functions. They will not return an explicit result. Instead as a result of calling them, they will reverse the values of the list they are given as input in place.

For your iterative solution, you should implement the following algorithm using a loop, you should do the following as a loop as long as begin is less than end `(begin < end)`:

1. Swap the values at the begin and end indexes.
2. increment begin and decrement end so next pass of loop will swap the next two values.

Notice that when `begin == end` or when `begin > end` this represents a `List` of size 1 or size 0 respectively. So the loop terminates once you get down to a list of size 1 or smaller.

Following this procedure you should be able to successfully reverse your list. Once you have done this, and you are satisfied your tests are passing for this test case, commit your work and push it to the `feedback` branch. Then document this task completion with a pull request and merge your work back to the main branch.

Task 4: Implement `reverseRecursive()` Function

Perform the usual prerequisite steps before starting task 4.

For task 4, implement a recursive version of the reversing a `List` function. This function will have the same signature as the previous function, the list and the begin and end index of the sub-list to be reversed.

The base case(s) should be similar for this recursive function as for your first one. A list of size 0 or size 1 is already the reverse of itself. So whenever asked to reverse a sub-list that is 1 or smaller in size, you can simply do nothing (return).

The general case then should be implemented again using swapping. For the general recursive case, you should first just swap the values from the begin and end indexes of the sub-list. Then you can reverse the middle portion of the list by calling your `reverseRecursive()` recursively on the sub-list from `begin + 1` to `end + 1`.

Once you are satisfied with your implementation of `reverseRecursive()` perform the usual steps to commit and push your work and document it with a pull request.

Task 5: Implement `isPalindromeIterative()` Function

Perform the usual prerequisite steps before starting task 5.

The final functions will determine if a sub-list is a palindrome as we described in the assignment description.

The `isPalindromeIterative()` takes the same 3 parameters as for the previous functions in this assignment. The `isPalindrome` functions will return a boolean result of `true` if the sub-list is a palindrome or `false` if the sub-list is not a palindrome.

The function may be a bit misnamed for the first case here, but we wanted to keep the same pattern of function names. An easy way to determine if a sub-list is a palindrome or not is to:

1. make a copy of the `List`
2. reverse the sub-list portion of this copy
3. Compare the original and the copy with the reverse sub-list.
    - If they are equal, then the reversed sup-list was a palindrome, so you can return `true`.
    - If they are not equal after the reverse, then the sub-list was not a palindrome.

You are required to reuse your `reverseIterative()` or your `reverseRecursive()` function to reverse the string. Of course if only 1 of them is working use that one (though you should not be working on this task if you don't have both of the previous tasks successfully passing all tests).

The `List` data type has member functions and operators that support the copying and comparison, so you don't have to implement these.

To copy a list, you can use the copy constructor provided by the `List` type. There are many examples of this in the tests, but for example.

```
// a list with values 1, 2  and 3
int values[] = {1, 2, 3}
List original(3, values);

// this makes a copy of the original list
List copy = original;
```

To compare if two lists are equal, the `==` operator has been defined for our `List` data type. This has been done using operator overloading, which we will discuss soon in a future unit.

```
// compare if the previous lists are equal
if (original == copy)
{
  // then they are equal
}
else
{
  // they are not equal
}
```

With these operations supported for lists, you should be able to implement the described algorithm to copy the list, reverse the sub-list portion, and then see if the results are the same or not.

Once you implement and get your `isPalindromeIterative()` function working, commit and push your changes back to your repository. Then document the completion of this task 5 with a pull request and merge.

Task 6: Implement `isPalindromeRecursive()` Function

Perform the usual prerequisite steps before starting task 6.

This function will have the same signature as the previous non-recursive version.

The base case is again going to be similar for your recursive function to determine if the list is a palindrome or not. List of size 1 or smaller are trivially considered palindromes. So `true` should just be returned immediately if the sub-list being checked is of size 1 or smaller.

Likewise the general recursive case has some similarities to the previous recursive function to reverse a list. You are required to use recursion in this implementation, so you should not be calling the `reverseRecursive()` function, that is not what we mean by a recursive implementation of the `isPalindrome` test.

To check if a list is a palindrome that does not fit the base case, first test if the `begin` and `end` values are equal or not. If they are not equal then you know the list is not a palindrome, and you can immediately return `false`.

If they are equal, then the list is a palindrome only if the sub-list from `begin + 1` to `end - 1` is a palindrome. So to find that out, you have to call your `isPalindromeRecursive()` recursive on this sub-list to test the middle part for palindromy-ness.

Once you are satisfied with your implementation of the `isPalindromeRecursive()` function, commit and push your changes to the `feedback` branch. And finish your assignment by documenting the completion of this task with a pull request.

Assignment Submission

For this class, the submission process is to correctly create pull request(s) with changes committed and pushed to your copied repository for grading and evaluation. For the assignments, you may not be able to complete all tasks and have all of the tests successfully finishing. This is ok. However, you should endeavor to have as many of the tasks completed before the deadline for the assignment as possible. Also, try and make sure that you only push commits that are building and able to run the tests. You may loose points for pushing a broken build, especially if the last build you submit is not properly compiling and running the tests.

In this problem, up to 40 points will be given for having at least 1 commit that compiles and runs the tests (and at least some attempt was made to work on the first task). Thereafter 10 points are awarded for completing each of the 6 tasks. However you should note that the autograder awards either all point for passing all tests, or no points if any test is failing for one of the tasks. Also note that even if you pass all tests, when the instructor evaluates your assignment, they may remove points if you don't follow the requirements for implementing the code (e.g. must reuse functions here as described, need to correctly declare parameters or member functions as `const` where needed, must have function documentation correct). You may also loose points for style issues. The instructor may give back comments in pull requests and/or create new issues for you if you have issues such as these, so it is good to have work committed early before the due date, so that the instructor may give feedback requesting you to fix issues with your current submission.

Program Style

At some point you will be required to follow class style and formatting guidelines. The VSCode environment has been set up to try and format your code for some of these guidelines automatically to conform to class style requirements. But not all style issues can be enforced by the IDE/Editor. The instructor may give you feedback in your pull comments and/or create issues for you for the assignment that you need to address and fix. You should address those if asked, and push a new commit that fixes the issue (or ask for clarification if you don't understand the request). In general the following style/formatting issues will be required for programs for this class:

1. All programs must be properly indented. All indentation must be consistent and lined up correctly. Class style requires 2 spaces with no embedded tabs for all code indentation levels. The editor style checker should properly indent your code when you save it, but if not you may need to check or correct this if code is misaligned or not properly indented.
2. Variable and function names must use `camelCaseNameingNotation`. All variable and function names must begin with a lowercase letter. Do not use underscores between words in the variable or function name. Often function names will be given to you, but you will need to create variables, and maybe some functions, that conform to the naming conventions.
    - Global constants should be used instead of magic numbers. Global constants are identified using `ALL_CAPS_UNDERLINE_NAMING`.
    - User defined types, such as classes, structures and enumerated types should use camel case notation, but should begin with an initial upper case letter, thus `MyUserDefinedClass`.
3. You are required to use meaningful variable and function names. Choosing good names for code items is an important skill. The code examples and starting code tries to give examples of good and meaningful names. In general, do not use abbreviations. Single variable names should be avoided, except maybe for generic loop index variables `i`, `j`, etc. Make your code readable, think of it as writing a document to communicate with other developers (and with your instructor who will be evaluating your code).
4. There are certain white space requirements. In general there should usually never be more than 1 blank line in a row in your code. Likewise there should usually not be more than 1 blank space on a line. There should be 1 blank space before and after all binary operators like `+`, `*`, `=`, `or`.
5. Function documentation is required for all regular functions and all class member functions. You need to follow the correctly formatted Doxygen function documentation format. We will use function documentation generation, and you should be sure your documentation can be built without emitting warnings or errors. Likewise all files should have a file header documentation at the top. You should edit the file header of files where you add in new code (not simply uncommenting existing code). Make sure the information has your correct name, dates, and other information.
6. Practice using proper Git commit messages. You should refer to issues and tasks correctly in commit messages.

Additional Information

The following are suggested online materials you may use to help you understand the tools and topics we have introduced in this assignment.

- Lecture U04-1 Recursion
- What is recursion?
- Recursion Basics - using Factorial
- Why recursion is not always good
- Exponentiation - Calculate pow(x,n) using recursion