CS 008
Lecture notes
3/26/24

1 Outline

- Recursion
- Git/Github

2 Recursion

A recursive function is a function that calls itself to achieve a certain task. Recursive algorithms are comprised of a **base case** and a **recursive case**. Iteration and recursion are just two ways of doing the same exact task. It is usually more memory intensive than the iterative counter part but can have some benefits:

Pros:

- Code is often more intuitive
- Easier to debug (sometimes)

Cons:

- Sometimes slower
- Could potentially use more memory in C++

Both iterative and recursive algorithms can be equally powerful depending on the implementation as shown in *Figure 1.*

Figure 1:

Recursive implementation:

```
void decrement_to_zero(int initial_number) {
    cout << "Number is " << initial_number << "!" <<
↪   endl;

    if (initial_number==0) {
        return; // base case
    } else {
        decrement_to_zero(initial_number-1); // recursive
↪   case
    }

}
```

Iterative implementation:

```
void decrement_to_zero(int initial_number) {
    while (initial_number >= 0) {
        cout << "Number is " << initial_number << "!" <<
↪   endl;
        initial_number--;
    }
}
```

2.1 Call Stack in C++

C++ has some memory known as the **call stack** or just known as the **stack**. The stack is where all variables within the local scope exists. The scope contains the variables contained within the function / chunk of code (often within). The stack contains local variables (within a scope) but does not contain dynamic variables which are stored in heap memory. It can also contain function calls which are stored until one finally returns (which ties into recursion).

Note:

Dynamic variables (ones created by the **new** keyword) are not on the call stack.

While a function is running, the variables are all stored on the stack. If that

first function calls a second function, the second function's variables go onto the top of the stack. Recall that a stack is a **LIFO** data structure and the first function's variables cannot be removed before the second function's.

Note:

The call stack is the reason why recursion usually ends up using more memory than it's iterative counterpart. When there are too many functions' variables being stored on the stack, it can use up all available memory on the call stack and creates a **segmentation fault**. Having too much on the stack can create a **stack overflow**.

Figure 2:

Recursive implementation:

```
int factorial(int number) {
    if (number==1) {
        return 1; // base case
    } else {
        return number * factorial(number-1); // recursive
↪   case
    }
}
```

Iterative implementation:

```
int factorial(int number) {
    int factorial_result = 1;
    while (number > 1) {
        factorial_result = factorial_result * number;
        number--;
    }

    return factorial_result;
}
```

2.2 Tail recursion

The function that is called recursively should have one return statement and the return should be the last line in the function. The return statement should not perform any computation of it's own (it needs to be a pure return statement that just passes in values). Some of the ways to help with writing tail recursive functions involve helper functions, ternary operators and accumulators.

Figure 3:

It's significantly more difficult to write tail recursive functions without the use of helper functions, ternary operators and an accumulator variable. Since one of the goals of tail recursion is to have only one return statement, returning to a helper function with an accumulator variable which returns a ternary operator makes tail recursion significantly easier to work with.

Note:

In tail recursion, the return statement should not perform any computation of it's own. It should be a pure return that does not handle any computation.

3 Git/github

Create a github account and complete the assignment on Canvas. Look for the github documentation if you need help.