MTH 4300, Lecture 6

Reference Parameters and Arrays; Recursion; Fibonacci, and the Dangers of Recursion

E Fink

February 19, 2025

Arrays can be made arguments to functions, but they are ALWAYS passed by reference, automatically. No & is necessary, and in fact putting one would lead to an error!

Note that arrays can declared as parameters either with a fixed size, or with no size (e.g. `int x[]`). In the latter case, it might be a good idea to have your function also accept the length of your array as an argument.

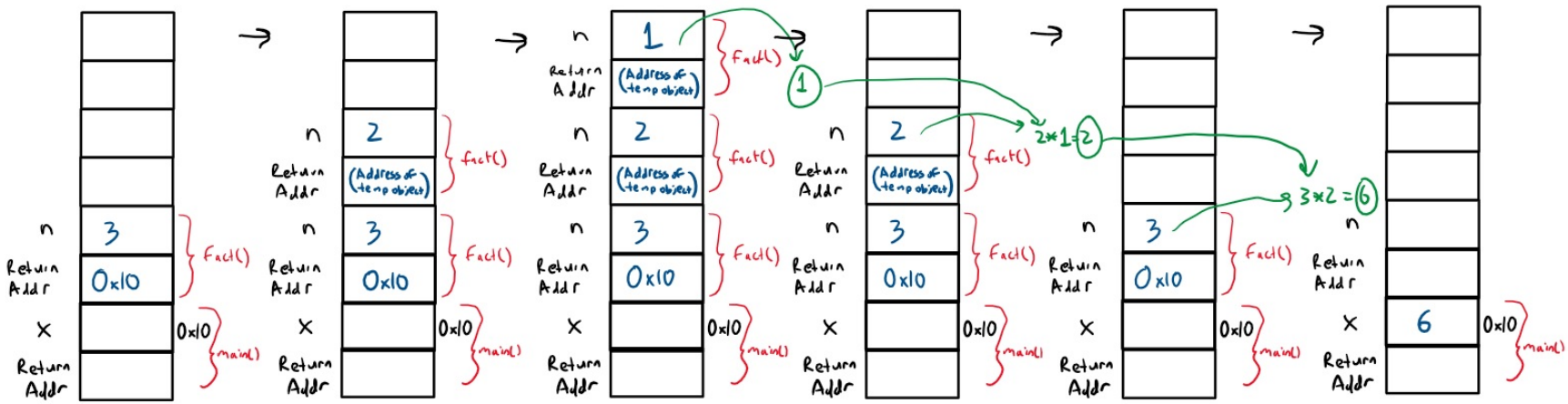
L6x1_array.cpp

Note: if a function is meant to merely read the contents of an array, rather than modify them (e.g., like the `swap()` function from last time), it is a good idea to put the qualifier `const` in front of the data type of the array. This causes a compilation error to appear if the body of the function tries to write a new value anywhere in the array. Of course, compilation errors are a GOOD thing.

Recall that a *recursive* function is a function that calls itself. Recursive functions, in addition to self-calls, usually have a *base case* which can be executed without a recursive call; and the recursive calls usually "move closer" to the base case.

Many problems can be naturally solved using recursion, like the factorial problem.

L6x2_fact.cpp (and see the depiction of the stack below)



Recursion

Let's write a recursive (and loop-free) function called `product()`, which takes an array, and computes the product of all the elements in a range specified by a lower and upper index.

For example, for an array declared as

```
int x[5] = {2, 10, 3, 1, 5};
```

the value of `product(x, 0, 4)` would be 300, the product of `x[0]` through `x[4]`;

the value of `product(x, 1, 3)` would be: 30, the product of `x[1]` through `x[3]`.

The strategy: to multiply entries `i` through `j`, **multiply `x[i]` by the product of entries `i+1` through `j`**.

Of course, what would be the base case? When `i` equals `j`, in which case the function should return entry `x[i]`.

L6x3_prod.cpp

Recursion

Next: a similarly designed recursive, loop-free function called `range_max()`, which takes an array, and computes the maximum value of all the elements in a range specified by a lower and upper index.

For example, for an array declared as

```
int x[5] = {2, 10, 6, 1, 5};
```

the value of `range_max(x, 0, 4)` would be 10, the maximum value among `x[0]` through `x[4]`;

the value of `range_max(x, 2, 3)` would be 6, the maximum value among `x[2]` through `x[3]`.

Follow a similar strategy to the last problem: namely, the max of entries `x[i]` through `x[j]` is either `x[i]` or the max of entries `x[i+1]` through `x[j]`, whichever is larger.

L6x4_max.cpp

3. Fibonacci, and the Dangers of Recursion

Common dangers when writing recursive functions include:

- Forgetting to write a base case.
- Neglecting to ensure that the recursive calls *progress towards* a base case.
- An explosion of recursive calls, as we'll see now.

Recall the Fibonacci sequence:

1, 1, 2, 3, 5, 8, 13, 21, ...

where each term after the first two is the sum of the two previous terms.

The formula for the nth Fibonacci term is:

$$fib(n) = \begin{cases} 1, & n = 1, 2 \\ fib(n-1) + fib(n-2), & n \geq 3 \end{cases}$$

This lends itself to a very direct recursive implementation.

L6x5_fib.cpp

Fibonacci, and the Dangers of Recursion

However, watch out! The fact that `fib()` makes **two** recursive calls opens up the possibility for an explosion of computation – since each of those calls can make two calls, each of which could make two calls.

Let's go to https://pythontutor.com/cpp.html to visualize **L6x6_modified.cpp** (which is slightly more unwieldy than the previous version, but has essentially the exact same complexity problem, as is easier to visualize).

Interesting: the *height* of the stack doesn't explode as n increases – but the time spent trying to clear the stack does, because the height keeps going up and down, up and down, recomputing previously computed values.

How does one improve this implementation? By *memoizing* – saving previously computed function calls to a cache, stored outside of the function. You can possibly use a global variable for this.