12. Variable scope and recursion

Herman Kamper

Two things in this lecture, and they turn out to be connected.

The first is a question we have been stepping around since we started writing functions: when a function creates a variable, where does that variable actually live? Who else can see it, and how long does it survive? That is variable scope, and it takes about a third of this lecture.

The second is recursion, which is what happens when a function calls itself. It sounds like it should not work. It does work, it is genuinely one of the more beautiful ideas in programming, and once you have seen it you cannot unsee it. But it only makes sense if you are solid on the first half, which is why the two are here together.

Variable scope: where variables live

Let's start with a question. Read the following program and write down, on paper, exactly what it prints. Do this before reading on.

```c
#include <stdio.h>

void fooFighters(int x);

int main()
{
    int x = 1;

    printf("(1) x = %d\n", x);
    fooFighters(x);
    printf("(3) x = %d\n", x);

    return 0;
}

void fooFighters(int x)
{
    x = x + 1;
    printf("(2) x = %d\n", x);
}
```

There are only two plausible answers. Either it prints 1, 2, 1 or it prints 1, 2, 2. Commit to one.

The answer is:

```
(1) x = 1
(2) x = 2
(3) x = 1
```

Here is why. The `x` declared inside `main` and the `x` declared as the parameter of `fooFighters` are not the same variable. They are two different boxes in memory that happen to share a name. I could have called the second one `apples` and the program would behave identically.

When `main` calls `fooFighters(x)`, what gets handed over is not the box, it is the value currently in the box. At that moment `x` is 1, so writing `fooFighters(x)` is exactly the same as writing `fooFighters(1)`. That value gets copied into a fresh box belonging to `fooFighters`, which also happens to be called `x`. This is what "call by value" means, and it is how every function you have written so far has behaved.

Inside the function, `x = x + 1` changes that fresh box from 1 to 2, and the 2 gets printed. Then the function ends. And when a function ends, its boxes are thrown away: the operating system reclaims that memory, and the second `x` stops existing. Back in `main`, the only `x` that ever existed there still contains 1.

Variables like these are called local variables. They live inside the function that declares them, they are invisible to every other function, and they are destroyed when the function returns. Every variable in the course up to this point has been one of these.

Global variables

There is a second kind of variable. If you declare a variable outside of all the functions, usually right at the top of the file, it belongs to the whole program:

```c
#include <stdio.h>

void fooFighters(void);

int y = 0;

int main()
{
    printf("In main, y = %d\n", y);
    fooFighters();
    printf("In main, y = %d\n", y);

    return 0;
}
```

```
void fooFighters(void)
{
    y = y + 1;
    printf("In fooFighters, y = %d\n", y);
}
```

This prints:

```
In main, y = 0
In fooFighters, y = 1
In main, y = 1
```

Follow it through. One box called `y` is created, outside of any function, and starts at 0. `main` prints it: 0. `fooFighters` runs, and the `y` it refers to is that same box, so `y = y + 1` changes the box itself to 1. The function ends and its own local boxes are thrown away, but `y` is not one of them: it does not belong to `fooFighters`. Back in `main`, printing `y` gives 1.

That is a global variable. It lives everywhere, and any function can read it or change it.

Now the warning: use these sparingly. Plenty of software engineering books will tell you flatly that if you can avoid a global variable, you should. The reason becomes obvious on a project with more than one person on it. Suppose you write half the functions and somebody else writes the other half, and there is a global variable that both halves touch. They change it somewhere, for reasons that made sense to them, and your function now sees a value it was not expecting. Nobody did anything wrong, and yet the program is broken, and finding out where is genuinely hard because the change could have come from anywhere.

The alternative is what you have been doing all along: break the problem into functions, and pass what each function needs through its parameters. Then you can see, from the function's first line, everything that can possibly affect it. Sometimes you really do need a global variable, and then you use one. Just not by default.

Static variables

The third kind is the strange one. Consider a function with an ordinary local variable in it, called twice:

```
void fooFighters(void)
{
    int z = 5;

    z = z + 1;
    printf("z = %d\n", z);
}
```

Call it twice from `main` and it prints:

```
z = 6
z = 6
```

Most people expect 6 and then 7, so make sure you see why it is 6 twice. On the first call a box called `z` is created with 5 in it, incremented to 6, printed, and then destroyed when the function ends. On the second call a brand new box is created, with 5 in it again, and the whole thing repeats. The function has no memory of having been called before.

Now change one word:

```
void fooFighters(void)
{
    static int z = 5;

    z = z + 1;
    printf("z = %d\n", z);
}
```

The same two calls now print:

```
z = 6
z = 7
```

That is a static local variable, and it is an odd hybrid. It is still local: `main` cannot see `z`, cannot read it, cannot change it, and would not compile if it tried. But it is not destroyed when the function returns. It is created once, keeps its value between calls, and lives as long as the program does. The initialisation `= 5` happens once only, the first time through, which is why the second call picks up at 6 rather than starting again at 5.

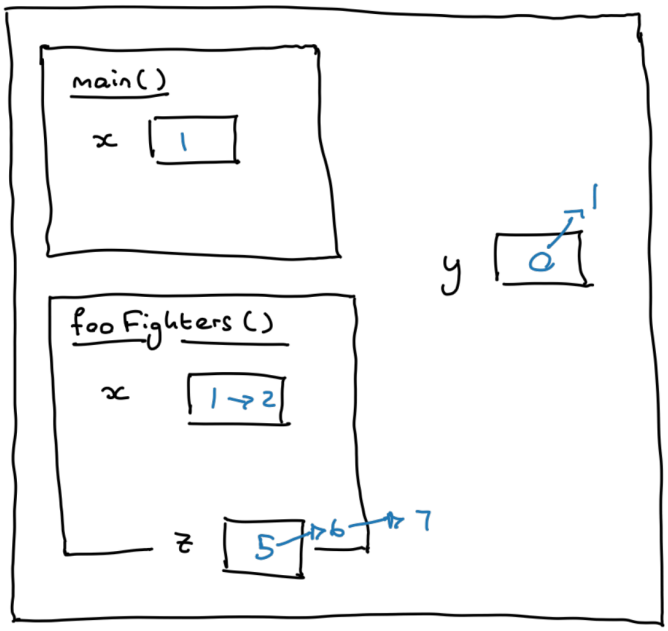

Figure 1: The three kinds of variable. Local variables live inside their function and are destroyed when it returns, so the `x` in `main` and the `x` in `fooFighters` are different boxes. The global `y` sits outside all the functions and is shared by them. The static `z` sits on the edge of its function: only `fooFighters` can reach it, but it survives from one call to the next.

I find it helps to draw the three kinds in one picture, with static variables sitting on the boundary of their function, half in and half out.

Put another way, there are two separate questions to ask about any variable, and they have different answers:

Variable	Who can see it?	How long does it live?
Local	Only its own function	Until that call returns
Global	Every function	The whole program
Static local	Only its own function	The whole program

The first column is scope. The second is lifetime. Local variables are limited on both counts, globals on neither, and static locals are restricted in scope but not in lifetime.

Keep the first row in mind. It is about to become the thing that makes recursion work at all.

Recursion

Every function call you have written so far follows the same pattern. `main` calls a function, that function calls another one, that one calls `printf` or `sqrt`, and so on. Functions call other functions, in an orderly way, and eventually everything returns.

Recursion is the case where a function calls itself.

That is the whole definition, and the first reaction to it is usually that it cannot possibly work, or that it must run forever. Neither is true. To see why, it helps to forget about C entirely for a moment.

The same idea in mathematics

Here is a function defined the way you would have seen it in school:

$$f(n) = \begin{cases} 1 & \text{if } n = 1 \\ n \cdot f(n-1) & \text{if } n > 1 \end{cases}$$

Take a piece of paper and work out $f(3)$. Genuinely do it, rather than reading on. Struggling with a question like this for a minute is worth more than watching someone else answer it.

$f(3)$ is not the $n = 1$ case, so we are in the second line: $f(3) = 3 \cdot f(2)$. To get $f(2)$ we apply the definition again: $f(2) = 2 \cdot f(1)$. And $f(1)$ is the first line, which is just 1. So $f(2) = 2$, and $f(3) = 6$.

Nothing about that felt strange, and you have just done recursion. The function was defined in terms of itself, and it worked out fine, because each time we applied the definition the argument got smaller, and eventually it hit a case that did not refer back to the function at all.

Now do $f(5)$, which is where the shape of it becomes visible.

This is where the film *Inception* is a genuinely good analogy. You go into a dream, and inside that dream you have another dream, and inside that one another, going deeper each time. Then at the bottom you wake up, one level at a time, and each level you return to carries something back with it from the level below.

$f(5)$ is the first dream. It needs $f(4)$, so we go a level deeper. $f(4)$ needs $f(3)$, deeper again. Then $f(2)$, then $f(1)$, which is the bottom: $f(1) = 1$, no further descent. Now we wake up. $f(1)$ hands 1 back to $f(2)$, which computes $2 \cdot 1 = 2$ and hands that back to $f(3)$, which computes $3 \cdot 2 = 6$ and hands that to $f(4)$, giving $4 \cdot 6 = 24$, and finally $5 \cdot 24 = 120$.

If you followed that, you understand recursion, and you have not written a line of code yet.

You may also have noticed what f actually computes. We worked out that $f(5) = 120$, and $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$. It is the factorial. Which makes sense written out:

$$5! = 5 \times (4 \times 3 \times 2 \times 1) = 5 \times 4!$$

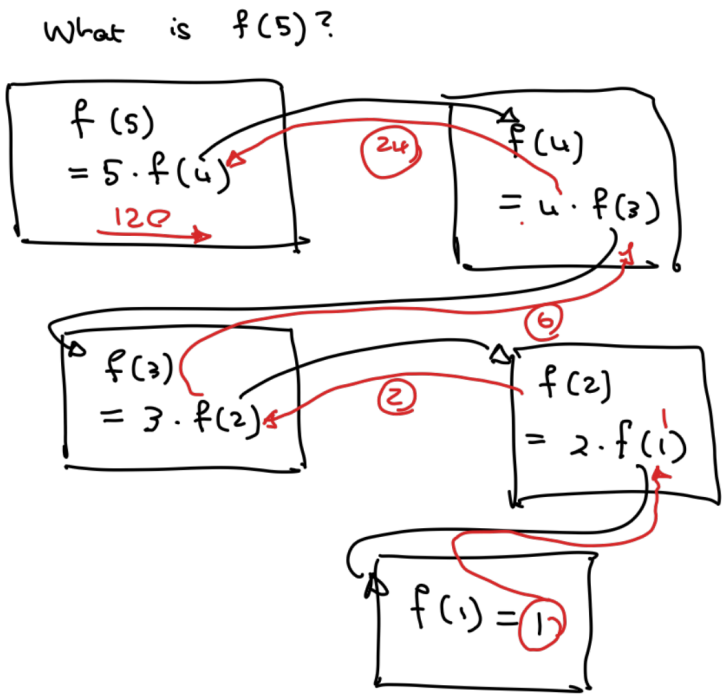

Figure 2: Working out $f(5)$ by hand. Going down the levels in black, each call waiting on a smaller one, and then coming back up in red, each level multiplying by its own n and handing the result to the level above.

and in the same way $4! = 4 \times 3!$, and $3! = 3 \times 2!$, down to $1! = 1$. A factorial is defined in terms of a smaller factorial, and that is exactly the structure a recursive function wants.

Every recursive definition has these two parts, and it is worth naming them because you will need both every time you write one. There is the base case, the simplest possible input, which is answered directly with no further calls: here, $f(1) = 1$. And there is the recursive case, which answers the general problem in terms of a smaller version of the same problem: here, $n \cdot f(n-1)$. Without a base case you never stop. Without the problem getting smaller each time you never reach the base case, which amounts to the same thing.

In code

The translation is almost mechanical:

```
/*
 * Use recursion to calculate factorials.
 * Author: Herman Kamper
 */

#include <stdio.h>

int factorial(int n);

int main()
{
    int n;

    printf("n = ");
    scanf("%d", &n);

    printf("n! = %d\n", factorial(n));

    return 0;
}


/*
 * Calculate the factorial of n.
 */
int factorial(int n)
{
    int result;

    if (n == 1)
        result = 1;
    else
        result = n * factorial(n - 1);

    return result;
}
```

Look at the `else` branch. Inside the definition of `factorial`, we call `factorial`. When students suggest this out loud for the first time there is usually a pause, as though they have proposed something illegal. It is completely legal, and it is the same thing you did on paper a page ago.

Running it:

```
n = 3
n! = 6

n = 5
n! = 120
```

Here is how I think about what happens inside. It is not exactly what the machine does, but it is close enough to reason with. When `main` calls `factorial(3)`, a copy of the function gets created, with its own `n` holding 3. That copy runs until it reaches `n * factorial(n - 1)`, at which point it needs the answer to `factorial(2)` before it can do anything else, so it stops and waits. A second copy is created, with its own `n` holding 2. That one gets as far as needing `factorial(1)`, and waits. A third copy is created with `n` holding 1, and this one hits the base case, returns 1, and finishes. Now the waiting copies can finish, innermost first: the `n = 2` copy computes 2×1 and returns 2, and then the `n = 3` copy computes 3×2 and returns 6 to `main`.

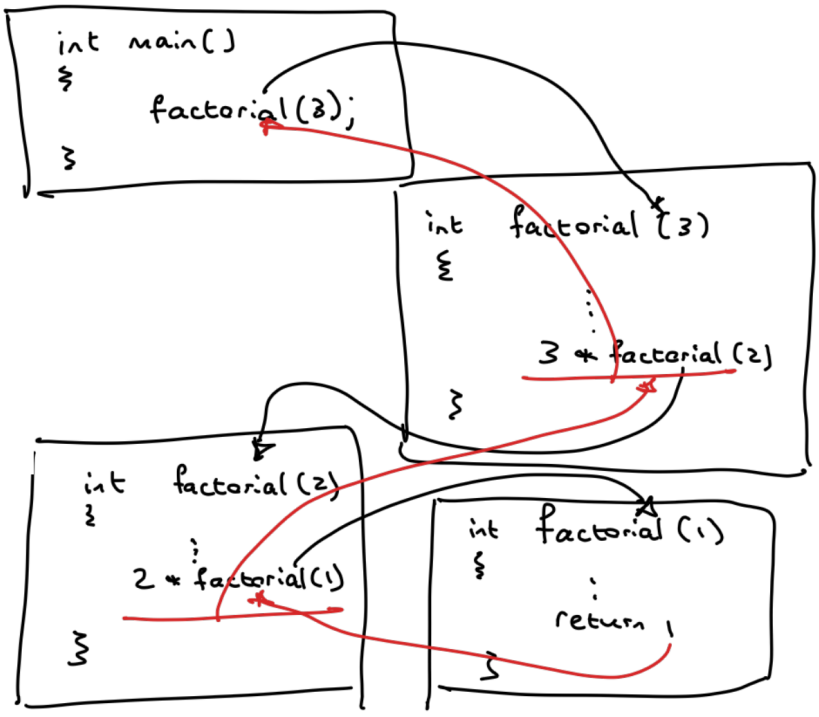

Figure 3: The same trace, in code. `main` calls `factorial(3)`, which cannot finish until `factorial(2)` returns, which cannot finish until `factorial(1)` returns. The black arrows go down, the red ones carry the answers back up.

Notice what makes this work, and it is exactly the point of the first half of this lecture. Each copy of the function has its own `n`. They are local variables, and local variables belong to a particular call, not to the function in general. Three calls to `factorial` are live at the same time and there are three separate boxes called `n`, holding 3, 2 and 1, none of which interferes with the others. If parameters were shared between calls the whole scheme would collapse.

The code itself does not get copied, incidentally. There is one `factorial` function in the program and every level runs the same instructions. What differs from level to level is the data: the value of `n`, and the value each call is waiting on.

Two practical warnings. The base case must actually be reachable. This version tests `n == 1`, so calling `factorial(0)` or `factorial(-3)` never hits it: `n` goes 0, -1, -2 and downwards forever, each call using a bit more memory, until the program dies. Real code should check the input before recursing, or use `n <= 1` as the base case. And second, factorials grow enormously fast, faster than an `int` can hold. On a typical machine `factorial(12)` gives 479001600, which is right, and `factorial(13)` gives 1932053504, which is not: the true answer is 6227020800 and it does not fit. Nothing warns you. This is not a problem with recursion, it is a problem with `int`, but it is worth knowing that a correct algorithm can still give you a wrong number.

A second example: Fibonacci

The Fibonacci series starts with 0 and 1, and every number after that is the sum of the previous two. So 0 + 1 gives 1, then 1 + 1 gives 2, then 1 + 2 gives 3, then 2 + 3 gives 5, and so on:

n	0	1	2	3	4	5	6	7	8
Fibonacci	0	1	1	2	3	5	8	13	21

The top row is the index. What we want is a function that takes an index and gives back that entry: ask it for 6 and it says 8, ask it for 100 and it says whatever the hundredth entry is.

As a mathematical definition:

$$\text{Fibonacci}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \text{Fibonacci}(n-1) + \text{Fibonacci}(n-2) & \text{if } n > 1 \end{cases}$$

Two base cases this time, because you need two starting values before the pattern can begin, and one recursive case which calls the function twice.

In C:

```c
int fibonacci(int n)
{
    if (n == 0 || n == 1)
        return n;
    else
        return fibonacci(n - 1) + fibonacci(n - 2);
}
```

That is the entire function. Four lines, and it is worth staring at for a moment: the last line calls the function it is in the middle of defining, twice, in a single expression.

Check it by hand rather than trusting it. Take `fibonacci(3)`, which from the table should be 2.

We start in `fibonacci(3)`. `n` is neither 0 nor 1, so we need `fibonacci(2) + fibonacci(1)`. Deal with `fibonacci(2)` first. Its `n` is 2, so it needs `fibonacci(1) + fibonacci(0)`. Both of those are base cases: `fibonacci(1)` returns 1 and `fibonacci(0)` returns 0, so `fibonacci(2)` returns 1. Back up in `fibonacci(3)` we still owe the second half, `fibonacci(1)`, which is a base case returning 1. So `fibonacci(3)` returns 1 + 1, which is 2, exactly as the table says.

Compare that picture with the factorial one. Factorial descends in a straight line, one call at each level. Fibonacci branches, because each call spawns two more. That difference matters more than it looks.

Notice that `fibonacci(1)` gets worked out twice in that tree, in two different places, and nothing remembers that we already knew the answer. For $n = 3$ that is a wasted call or two.

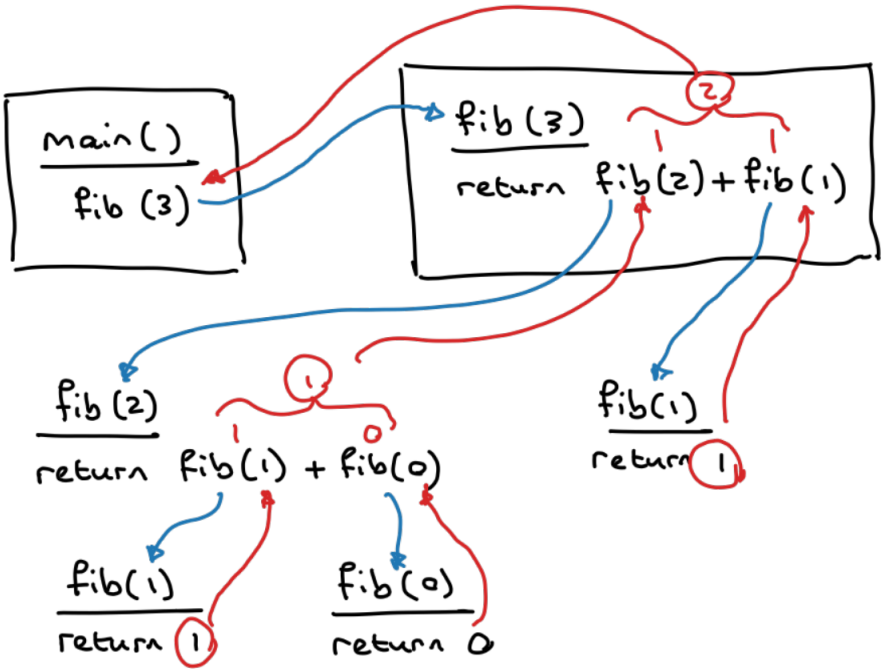

Figure 4: Tracing `fibonacci(3)`. I use `fib()` as a shorthand for `fibonacci()` in the figure. Because the recursive case makes two calls rather than one, the picture is a tree rather than a single chain. Blue goes down into the calls, red brings the answers back.

But the waste compounds: computing `fibonacci(30)` this way makes 2692537 separate calls to arrive at the answer 832040, and each step up in n roughly multiplies the work by 1.6. Ask for `fibonacci(50)` and you will have time to make coffee. The code is short and it mirrors the mathematics exactly, and it is also a spectacularly inefficient way to compute a Fibonacci number.

So should you use recursion?

The honest answer is: less often than you might think.

Anything you can write with recursion you can also write with a loop. That is not a rule of thumb, it is a theorem. Factorial with a `for` loop is no harder to write than the recursive version:

```
int factorial(int n)
{
    int i, result = 1;

    for (i = 2; i <= n; i++)
        result = result * i;

    return result;
}
```

And the loop version is cheaper. Every recursive call costs memory, because every level that is still waiting has to be kept around, along with its own copy of the parameters and local variables. A loop keeps one set of variables and reuses them. Recursion is more processor and memory intensive, essentially always.

So we use recursion when it makes the code clearer, not when it makes it faster, because it will not make it faster. For problems that are naturally defined in terms of smaller versions of themselves, the recursive version can be dramatically shorter and easier to follow. Searching through a branching structure is the standard example: at each point you want to do the same thing to each of the branches, and each branch is itself a smaller version of the same structure. Written recursively that is a few lines. Written with loops it takes real effort, because you have to keep track of where you still have to go back to, and you end up hand-building the very thing the recursive version got for free.

My own habit is to reach for a loop first and use recursion only where it genuinely earns its place. Some people use it everywhere. I think that mostly makes programs harder to read.

Where recursion shows up

Two examples, because it is more common than it sounds.

Suppose you are generating scenery for a computer game and you want trees. There is a rule that produces one: draw a branch, then at the end of that branch draw two shorter branches going off to the left and right. Then apply the same rule to each of those branches. And to each of theirs.

Figure 5: The rule, applied twice. Start with a single branch, add a branch to the left and a branch to the right, then do the same thing again to every branch you just created.

Keep going for a dozen levels, shrinking the branches slightly each time, and what comes out looks startlingly like a real tree. The rule is three words long and it is recursive: to draw a tree, draw a branch and then draw two smaller trees on the end of it.

The second example is not about computers at all. Human language is recursive, and this may be the thing that most sharply separates it from animal communication. You can put a sentence inside a sentence. “Dave thinks that he is cool” contains “he is cool”. And then you can do it

again: “Mary said that Dave thinks that he is cool”. And again: “John wonders if Mary said that Dave thinks that he is cool”.

There is no longest sentence, and the proof is recursive. Whatever sentence you claim is longest, I can say “I once said that” followed by your sentence, and now mine is longer. Linguists including Noam Chomsky have argued that this ability to nest structure inside structure without limit is the core of what makes human language what it is. Whether or not that turns out to be the whole story, it is a nice thought that the idea you just used to compute a factorial is also the reason you can keep talking.

Looking forward

That closes out functions. You can write them, split them across files, give them types of your own, and now you have seen them call themselves. Along the way this lecture answered the question of where a variable lives, which is not really about functions at all but about memory, and that is a thread that gets picked up properly later.

Recursion is one of those ideas that takes a while to settle. If the factorial trace made sense and the Fibonacci one felt like hard work, that is completely normal, and the fix is to do a few of them by hand on paper rather than to read about it again. The exercises below are mostly there for exactly that.

Exercises

1. Predict the output of the first program in this lecture without running it, then change `fooFighters` so that the change to `x` does survive back in `main`, using only what you know so far. There is a way, and finding it will tell you whether you have understood the difference between a value and a box.
2. Write a program with a function containing an ordinary local counter that is incremented and printed on each call. Call it five times. Now make the counter `static` and call it five times again. Write down both outputs before running either.
3. Write a recursive function that computes x^n for a non-negative integer n. What is the base case? Check it against a few values you can work out by hand.
4. Write a recursive function that takes a positive integer and returns the sum of its digits, so that 4172 gives 14. The recursive step needs `% 10` and `/ 10`.
5. Add a counter to the recursive `fibonacci` that increments on every call, and print it alongside the answer. Run it for $n = 5$, 10, 20 and 25 and look at how the count grows. Then write an iterative version with a loop and compare how much work each does.