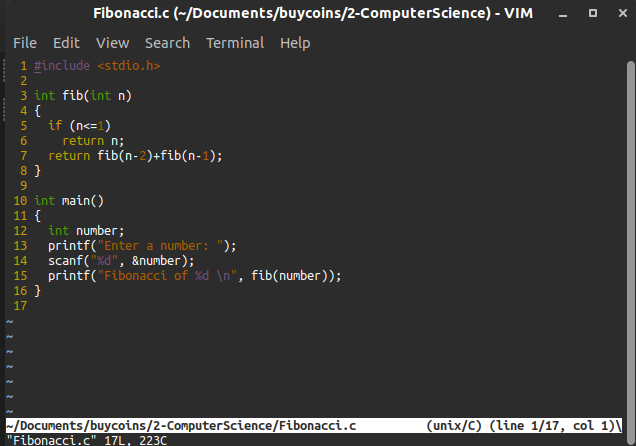
**Background**

Recursion involves a function calling itself over until the base case becomes true.

In the case of the Fibonacci sequence, at every function (fib) call, a separate activation record is created in the stack memory. This function resides in its own memory space and contains its own values.



All functions are not executed as they are called (no value is returned yet) only until the base condition is met (in this case until n becomes <=1). At this point, all previous functions starting from the last function returns its value to their calling function up to the first function that was called.

In the example above, if number = 5, then when n <= 1, fib(1) returns its value (1) to fib(2) which returns its value (1) to fib(3) which then returns its own value (2) to fib(4). Fib(4) returns its value (3) to fib(5). At the end, fib(5) returns 5.

As each fib function executes, its own activation record created within this stack memory is deleted and its value returns to its calling function.

**So why recursion is bad for finding the Fibonacci of a number?**

* **High space complexity:** The process of calling an entirely new function has a space complexity of **O**(n) because it function requires its own memory space. This can lead to a stack-overflow error which occurs when the stack memory is exceeded (most times this limit is set by the compiler/interpreter in the case of C and Python). This scenario is not good for our programs.
* **Costly time complexity:** Instantiating new functions also comes with a cost that increases the time complexity of the program. A new function creation corresponds to an increase in computational complexity. In fact, it has a time complexity (we can estimate computational complexity by analyzing time) of **O**(2n) which isn’t good for our programs too.