


Introduction to recursion

Functions and the structure of recursive problems

Functions

Functions are pieces of code you can reuse

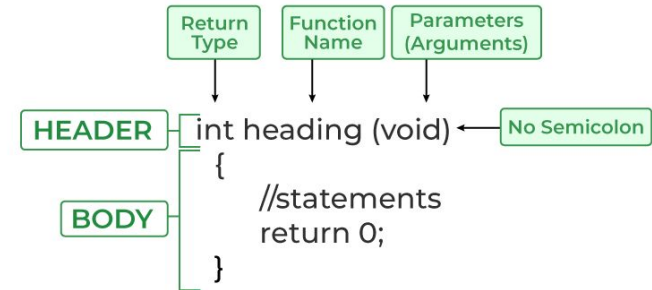


```
1 void shout() {  
2     printf("FUS RO DAH\n");  
3 }  
4  
5 int main() {  
6     shout(); // will print "FUS RO DAH" to terminal  
7     shout(); // will print "FUS RO DAH" to terminal, again  
8     shout(); // will print "FUS RO DAH" to terminal, again  
9     shout(); // will print "FUS RO DAH" to terminal, again  
10 }
```

Parts of a function

- Header: the code that defines the function
 - Return type
 - Function name
 - Parameters
- Body: the code inside the function
 - Must include a return statement...
 - ...**UNLESS** you have void return type

Function Definition

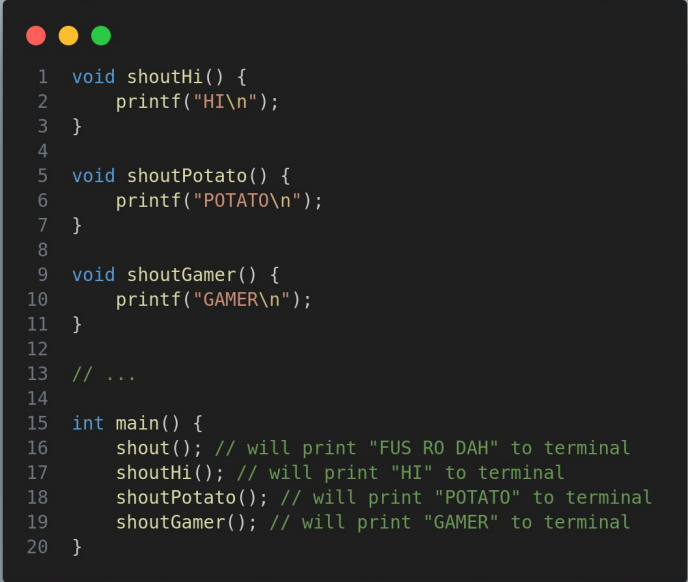


<https://media.geeksforgeeks.org/wp-content/uploads/20230302125407/C-Function-Prototype-and-C-Function-Call.png>

What if I want to shout something different?

Should I write 10 different shouting functions?

This becomes a lot of annoying back and forth work



```
1 void shoutHi() {  
2     printf("HI\n");  
3 }  
4  
5 void shoutPotato() {  
6     printf("POTATO\n");  
7 }  
8  
9 void shoutGamer() {  
10    printf("GAMER\n");  
11 }  
12  
13 // ...  
14  
15 int main() {  
16     shout(); // will print "FUS RO DAH" to terminal  
17     shoutHi(); // will print "HI" to terminal  
18     shoutPotato(); // will print "POTATO" to terminal  
19     shoutGamer(); // will print "GAMER" to terminal  
20 }
```

Functions can do different things based on the parameters

Parameters are passed into a function by the parentheses after function name

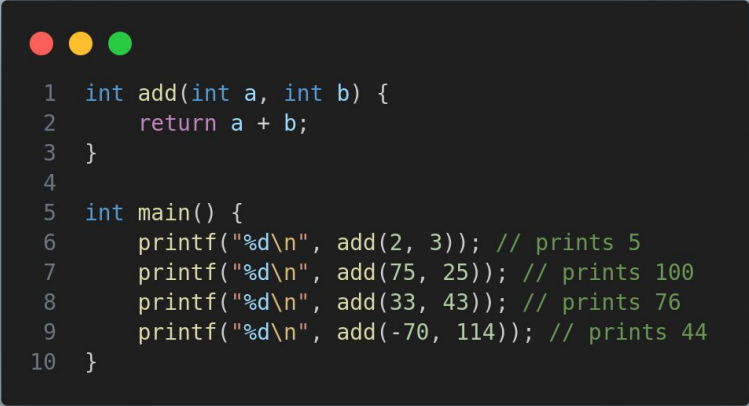
```
1 void shout(char * str) {  
2     printf("%s\n", str);  
3 }  
4  
5 int main() {  
6     shout("FUS RO DAH"); // will print "FUS RO DAH" to terminal  
7 }  
8
```

```
1 int add(int a, int b) {  
2     return a + b;  
3 }  
4  
5 int main() {  
6     printf("%d\n", add(2, 3)); // prints 5  
7     printf("%d\n", add(75, 25)); // prints 100  
8     printf("%d\n", add(33, 43)); // prints 76  
9     printf("%d\n", add(-70, 114)); // prints 44  
10 }
```

Calling a function means to use it in code

To call a function, write its name with parentheses and whatever parameters are inside


`printf` and `add` are both called in `main`



```
1  int add(int a, int b) {  
2      return a + b;  
3  }  
4  
5  int main() {  
6      printf("%d\n", add(2, 3)); // prints 5  
7      printf("%d\n", add(75, 25)); // prints 100  
8      printf("%d\n", add(33, 43)); // prints 76  
9      printf("%d\n", add(-70, 114)); // prints 44  
10 }
```

Functions often return a value


If I use a function, what do I get out of it?



```
1 int add(int a, int b) {  
2     return a + b;  
3 }  
4
```

add must return a value with `int` type.


But sometimes you don't need to return a value



```
1 void shout() {  
2     printf("FUS RO DAH\n");  
3 }
```

In this case, we use the `void` return type.

Example? `main` function!



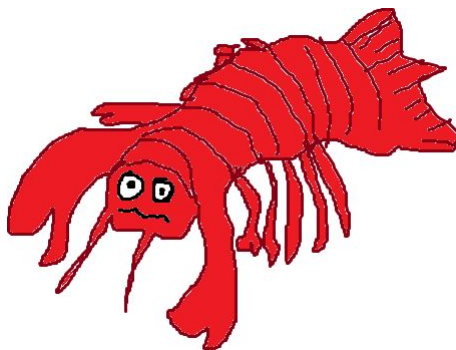
```
1  int main() {  
2      printf("hello!\n");  
3      call_some_function();  
4  }
```

Programs always “start” at the `main` function*

*unless you want to talk about kernels and operating systems, but I only had that conversation in my third year of studies

Demo

<https://lobster.eecs.umich.edu/#123> (eecs280 \Rightarrow L02.2_call_stack)



Summary on functions

- Functions are a way to write some code, then use it again whenever you need to.
- Functions can do different things depending on the parameters.
- Functions must specify the type of the value returned or specify that there is no return value (`void`).
- All programs start at the `main` function.

Further English-language resources

University of Michigan, EECS 280: [Function Calls and the Call Stack](#)

This resource goes into deeper detail about the **machine model** conceptualization of function calls.

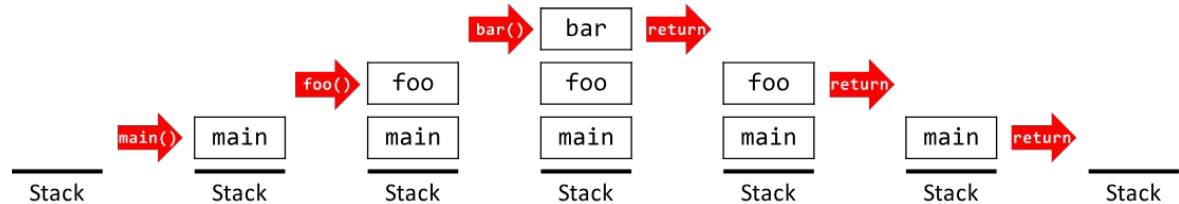
In simpler English: this is what function calls look like in the computer.

Call stack

There is an ordering in which functions are called

- Programs always start in `main()`
- When calling a function, you go to that function's body – function scope
 - If you call more functions, they start stacking into something called the call stack

```
void bar() {  
}  
  
void foo() {  
    bar();  
}  
  
int main() {  
    foo();  
}
```

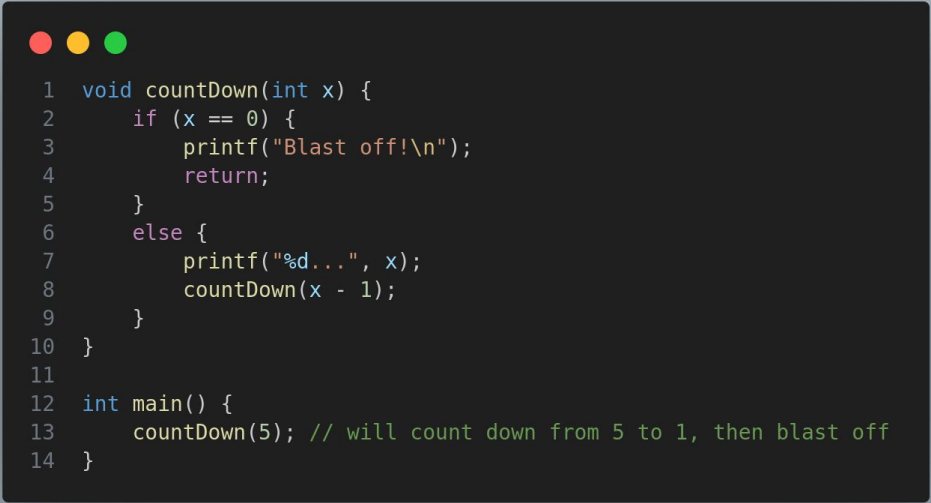


Basics of recursion

Cambridge Dictionary definition of recursion

the practice of describing numbers, expressions, etc. in terms of the numbers, expressions, etc. that come before them in a series

Simple programming definition: a function that calls itself



```
1 void countdown(int x) {  
2     if (x == 0) {  
3         printf("Blast off!\n");  
4         return;  
5     }  
6     else {  
7         printf("%d...", x);  
8         countdown(x - 1);  
9     }  
10 }  
11  
12 int main() {  
13     countdown(5); // will count down from 5 to 1, then blast off  
14 }
```

Structure of recursive problems

Fibonacci sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144

What do you notice about these numbers?

Fibonacci sequence can be defined recursively

Definition [\[edit\]](#)

The Fibonacci numbers may be defined by the [recurrence relation](#)^[6]

$$F_0 = 0, \quad F_1 = 1,$$

and

$$F_n = F_{n-1} + F_{n-2}$$

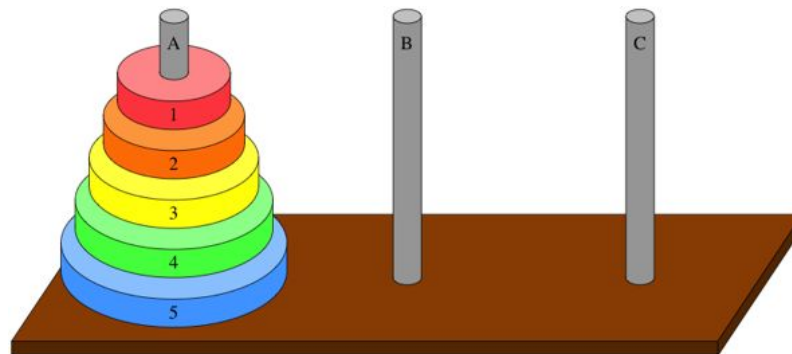
for $n > 1$.

https://en.wikipedia.org/wiki/Fibonacci_sequence

- In simple English: each number is the sum of the previous two numbers
- What about the first two numbers?

Let's play a game: Tower of Hanoi

- Your goal is to move all tower pieces from the left to the right
- The center starts empty
- There are many solutions here, but I'm looking for a recursive one!



<https://cdn.kastatic.org/ka-perseus-images/5b5fb2670c9a185b2666637461e40c805fcc9ea5.png>

A recursive solution to Tower of Hanoi

- If you only have one piece, just move it to the rightmost peg.
- If you have n pieces, use following algorithm:
 1. Move the top $n-1$ pieces to the right peg
 2. Move the bottom piece to the center peg
 3. Move the top $n-1$ pieces to the left peg
 4. Move the bottom piece to the right peg
 5. Move the top $n-1$ pieces to the right peg
- Notice how there are two parts to this solution

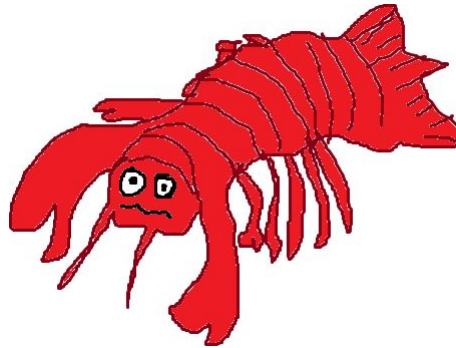
Recursive problems have two basic parts

- Base case: logic to handle the simplest case possible.
- Recursive case: logic to handle every other case
- Recursive problems should get smaller

```
1 void countDown(int x) {  
2     if (x == 0) {  
3         printf("Blast off!\n");  
4         return;  
5     }  
6     else {  
7         printf("%d...", x);  
8         countDown(x - 1);  
9     }  
10 }  
11  
12 int main() {  
13     countDown(5); // will count down from 5 to 1, then blast off  
14 }
```

Demo

<https://lobster.eecs.umich.edu/#151> (eecs280 \Rightarrow L18.3_countToN)



Reference this demo if you need to remember how recursive calls are ordered.

Further English-language resources

University of Michigan, EECS 280: [Recursion](#)

More on recursion, including a more memory-efficient version called “tail recursion”.

Exercise problems

Write a recursive function for Fibonacci numbers and factorial

Go to GitHub \Rightarrow michigan-musicer \Rightarrow al.exercise_5 for code files.

Extra note: You are expected to implement inefficient solutions. To learn about the efficient solutions, look up dynamic programming (combining recursion with storing information in arrays).