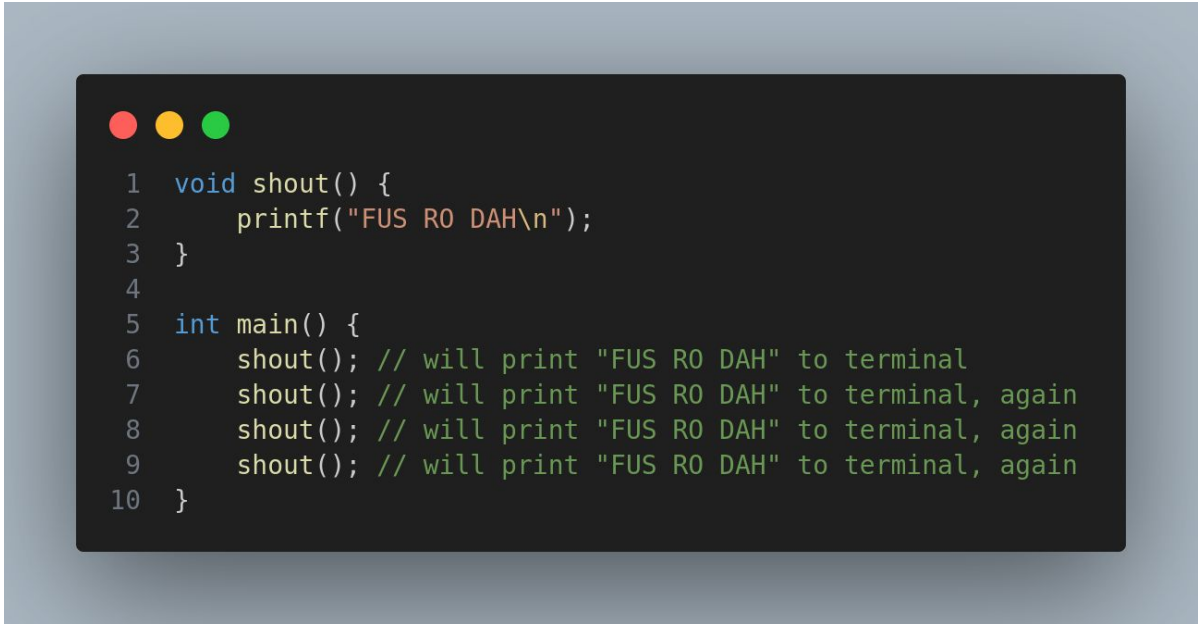
Introduction to recursion

Function calls and the structure of recursive problems

Functions

Functions are pieces of code you can reuse



```c
void shout() {
    printf("FUS RO DAH\n");
}

int main() {
    shout(); // will print "FUS RO DAH" to terminal
    shout(); // will print "FUS RO DAH" to terminal, again
    shout(); // will print "FUS RO DAH" to terminal, again
    shout(); // will print "FUS RO DAH" to terminal, again
}
```

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

```c
void shoutHi() {
    printf("HI\n");
}

void shoutPotato() {
    printf("POTATO\n");
}

void shoutGamer() {
    printf("GAMER\n");
}

// ...

int main() {
    shout(); // will print "FUS RO DAH" to terminal
    shoutHi(); // will print "HI" to terminal
    shoutPotato(); // will print "POTATO" to terminal
    shoutGamer(); // will print "GAMER" to terminal
}
```

Functions can do different things based on the **parameters**

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

```c
void shout(char * str) {
    printf("%s\n", str);
}

int main() {
    shout("FUS RO DAH"); // will print "FUS RO DAH" to terminal
}
```

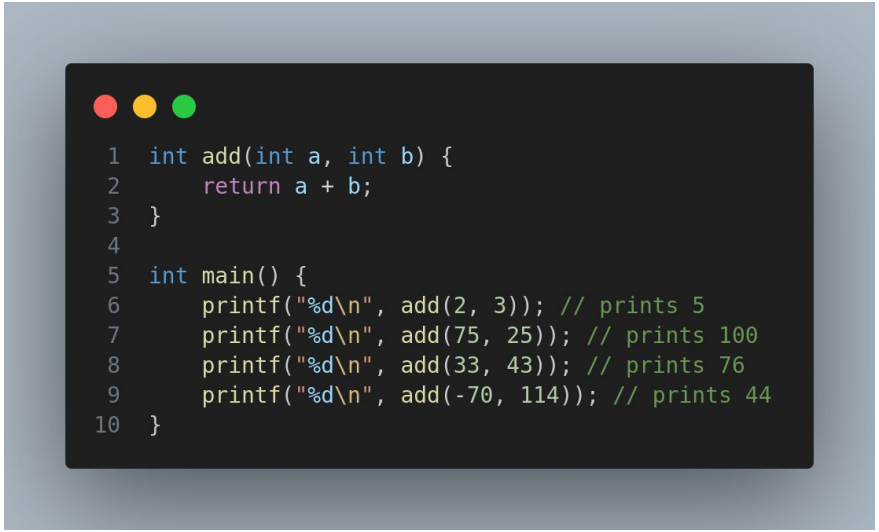
```c
int add(int a, int b) {
    return a + b;
}

int main() {
    printf("%d\n", add(2, 3)); // prints 5
    printf("%d\n", add(75, 25)); // prints 100
    printf("%d\n", add(33, 43)); // prints 76
    printf("%d\n", add(-70, 114)); // prints 44
}
```

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`

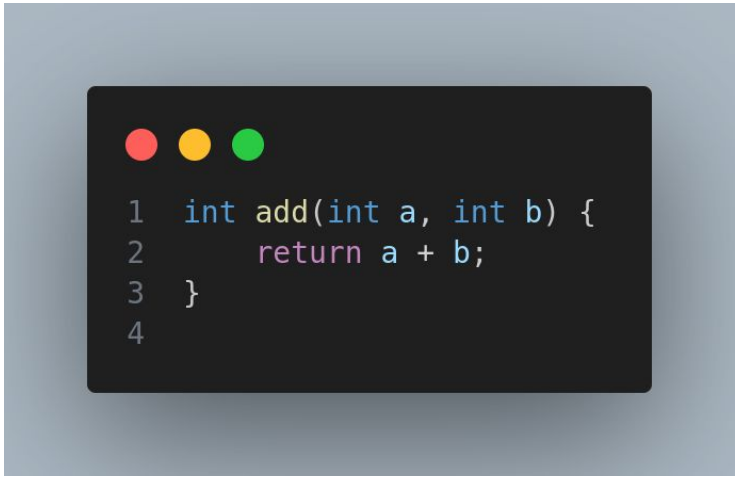


```c
int add(int a, int b) {
    return a + b;
}

int main() {
    printf("%d\n", add(2, 3)); // prints 5
    printf("%d\n", add(75, 25)); // prints 100
    printf("%d\n", add(33, 43)); // prints 76
    printf("%d\n", add(-70, 114)); // prints 44
}
```

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.

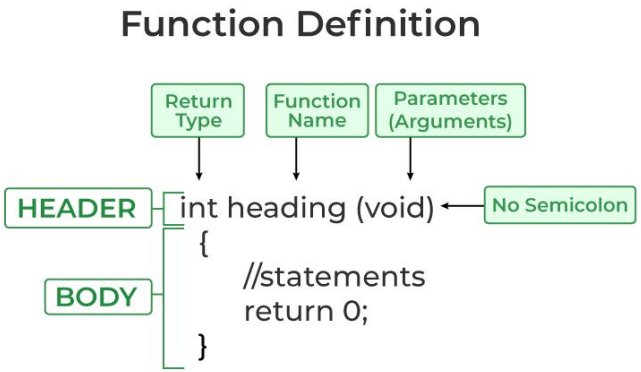
But sometimes you don't need to return a value

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

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

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



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

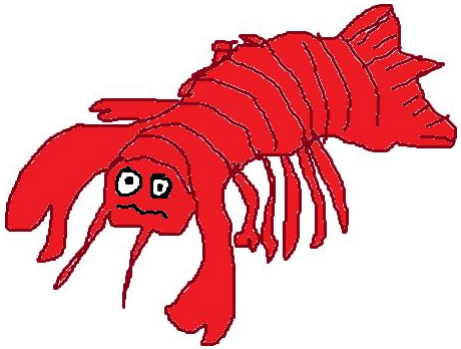
Example? `main` function!

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



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.

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

```c
void countDown(int x) {
    if (x == 0) {
        printf("Blast off!\n");
        return;
    }
    else {
        printf("%d...", x);
        countDown(x - 1);
    }
}

int main() {
    countDown(5); // will count down from 5 to 1, then blast off
}
```

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 $n > 1$.

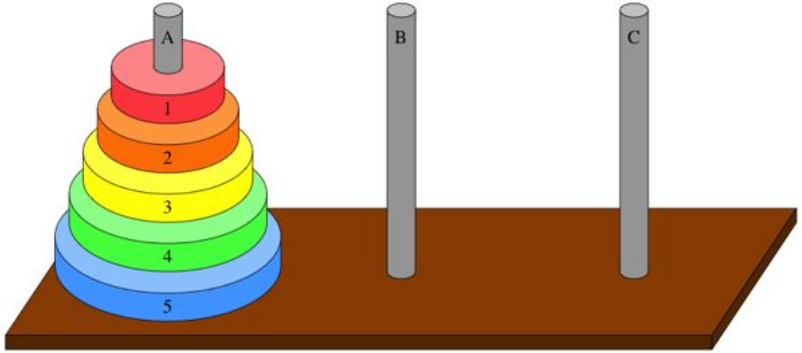
https://en.wikipedia.org/wiki/Fibonacci_sequence

- 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

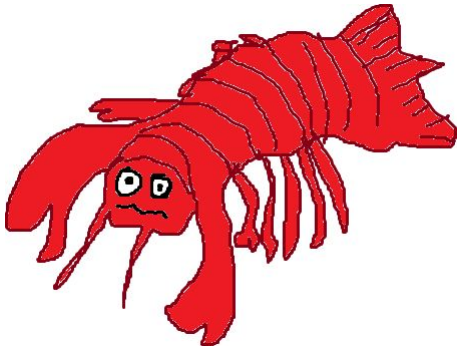
- Recursive problems should get smaller

```c
void countDown(int x) {
    if (x == 0) {
        printf("Blast off!\n");
        return;
    }
    else {
        printf("%d...", x);
        countDown(x - 1);
    }
}

int main() {
    countDown(5); // will count down from 5 to 1, then blast off
}
```

Demo

https://lobster.eecs.umich.edu/#151



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

Go to GitHub ⇒ michigan-musicer ⇒ al.exercise_4 for code files.

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