Passing Pointers to Functions

After arrays and structs, we'll see that pointers works with functions two. Another very important concept dealt with in this lesson is that of passing function arguments by reference, which allows us to alter variable outside the function scope.

We'll cover the following

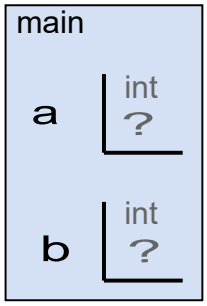
- Pointers to Functions
- Function Arguments: Passing By Value vs Passing By Reference

Pointers to Functions

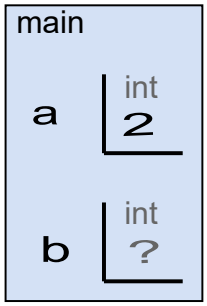
One of the handy things you can do in C, is to use a pointer to point to a function. Then you can pass this function pointer to other functions as an argument, you can store it in a struct, etc. Here is a small example:

```
#include <stdio.h>

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

int subtract( int a, int b ) {
  return a - b;
}

int multiply( int a, int b ) {
  return a * b;
}

void doMath( int (*fn)(int a, int b), int a, int b ) {
  int result = fn(a, b);
  printf("result = %d\n", result);
}

int main(void) {

  int a = 2;
  int b = 3;

  doMath(add, a, b);
  doMath(subtract, a, b);
  doMath(multiply, a, b);

  return 0;
}
```

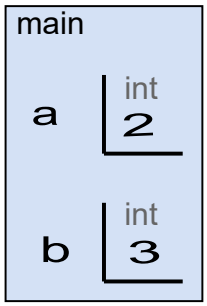
Stack



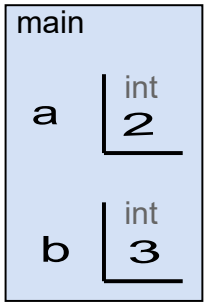
Stack



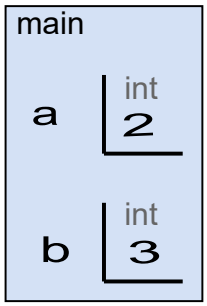
Stack



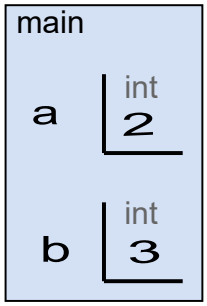
Stack



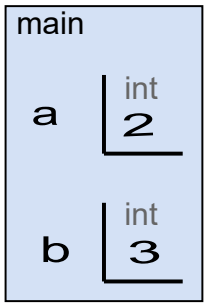
Stack



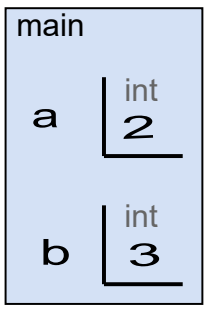
Stack



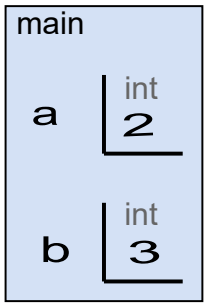
Stack



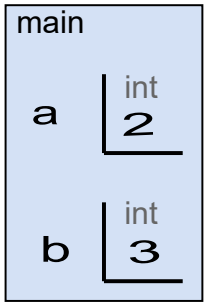
Stack



Stack

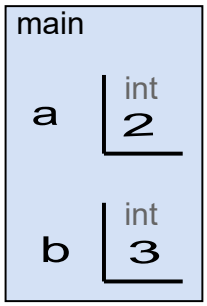


Stack

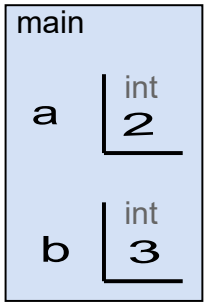


Output: Result = 5

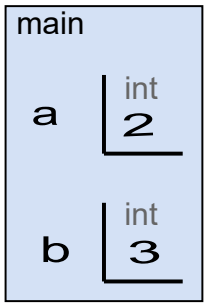
Stack



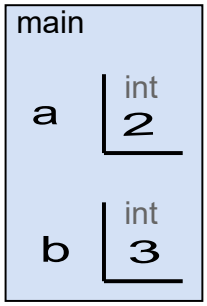
Stack



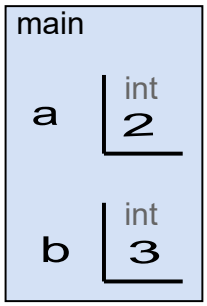
Stack



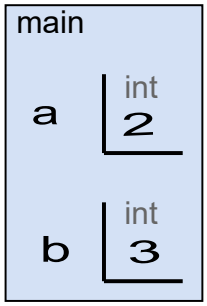
Stack



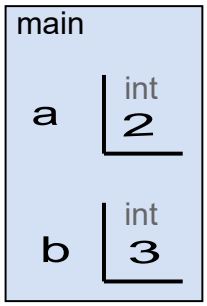
Stack



Stack

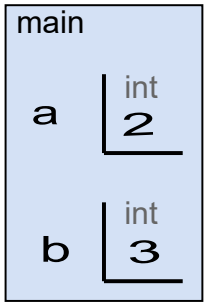


Stack

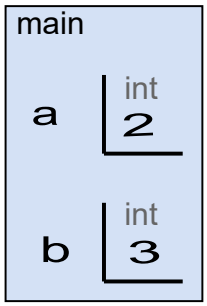


Output: Result = -1

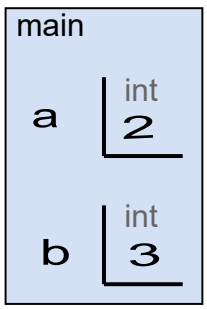
Stack



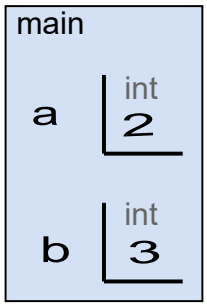
Stack



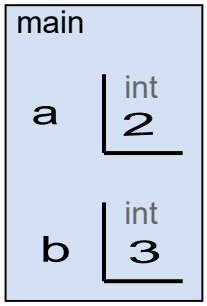
Stack



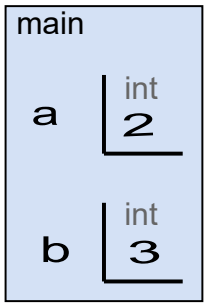
Stack



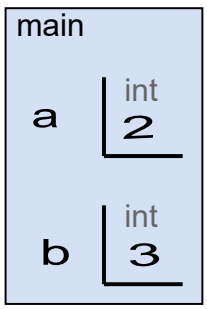
Stack

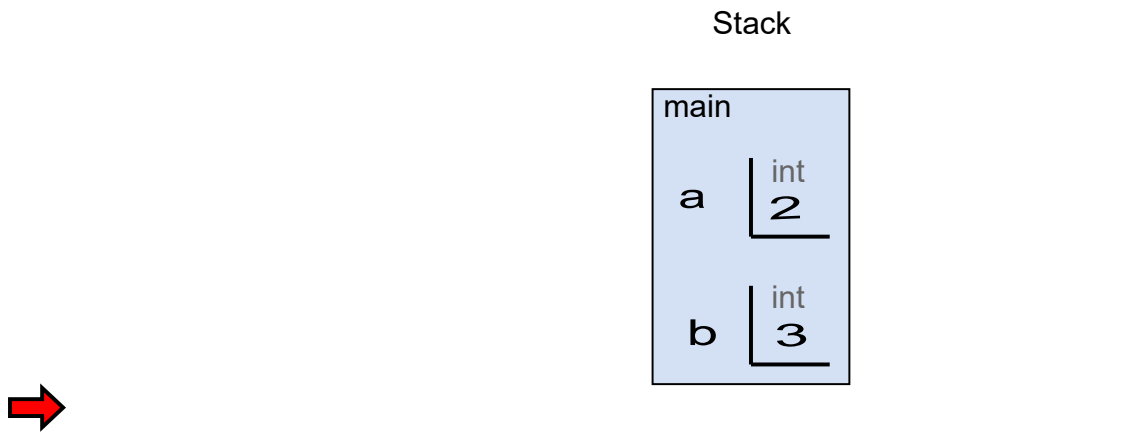


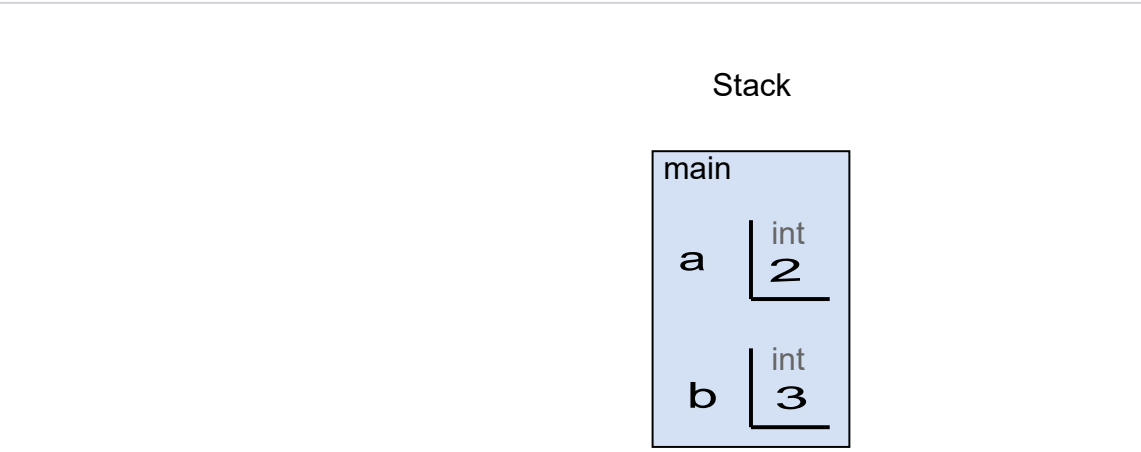
Stack



Stack







Let's go through the example above to understand what's happening. On lines **3-5**, **7-9** and **11-13**, we define functions `add`, `subtract` and `multiply`. These functions

return an `int` and take two `int` values as input arguments.

On lines **15-18** we define a function `doMath` which returns nothing (hence `void`) and which takes three input arguments. The first input argument is:

```
int (*fn)(int a, int b)
```

This first argument is a **pointer to a function**. It's actually more specific than that. It's a pointer to a specific kind of function: a function that returns an `int`, and that takes two `int` values as inputs. Let's unpack this. The `(*fn)` says this is a pointer to a function, and we shall refer to that function as `fn`. The preceding `int` says, it's a function that returns an `int`. The subsequent `(int a, int b)` says it's a function that takes two `int` arguments as inputs.

On lines 25-27, we call our `doMath()` function, each time passing it the three input arguments that it requires. First, a pointer to a function. Here we simply pass the **name** of one of the functions we defined above: `add()`, `subtract()` or `multiply()`. We are permitted to pass these functions to `doMath()` because they all satisfy the requirements of the first input argument of `doMath()`: they all return an `int`, and they all take two `int` values as inputs.

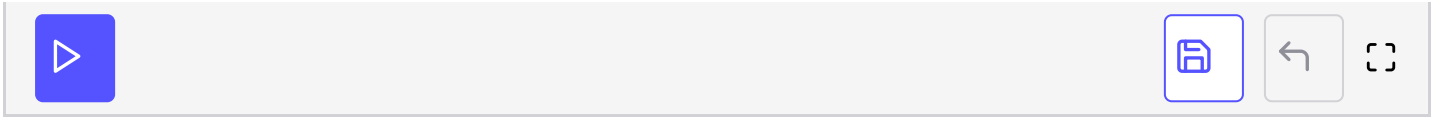
Function Arguments: Passing By Value vs Passing By Reference

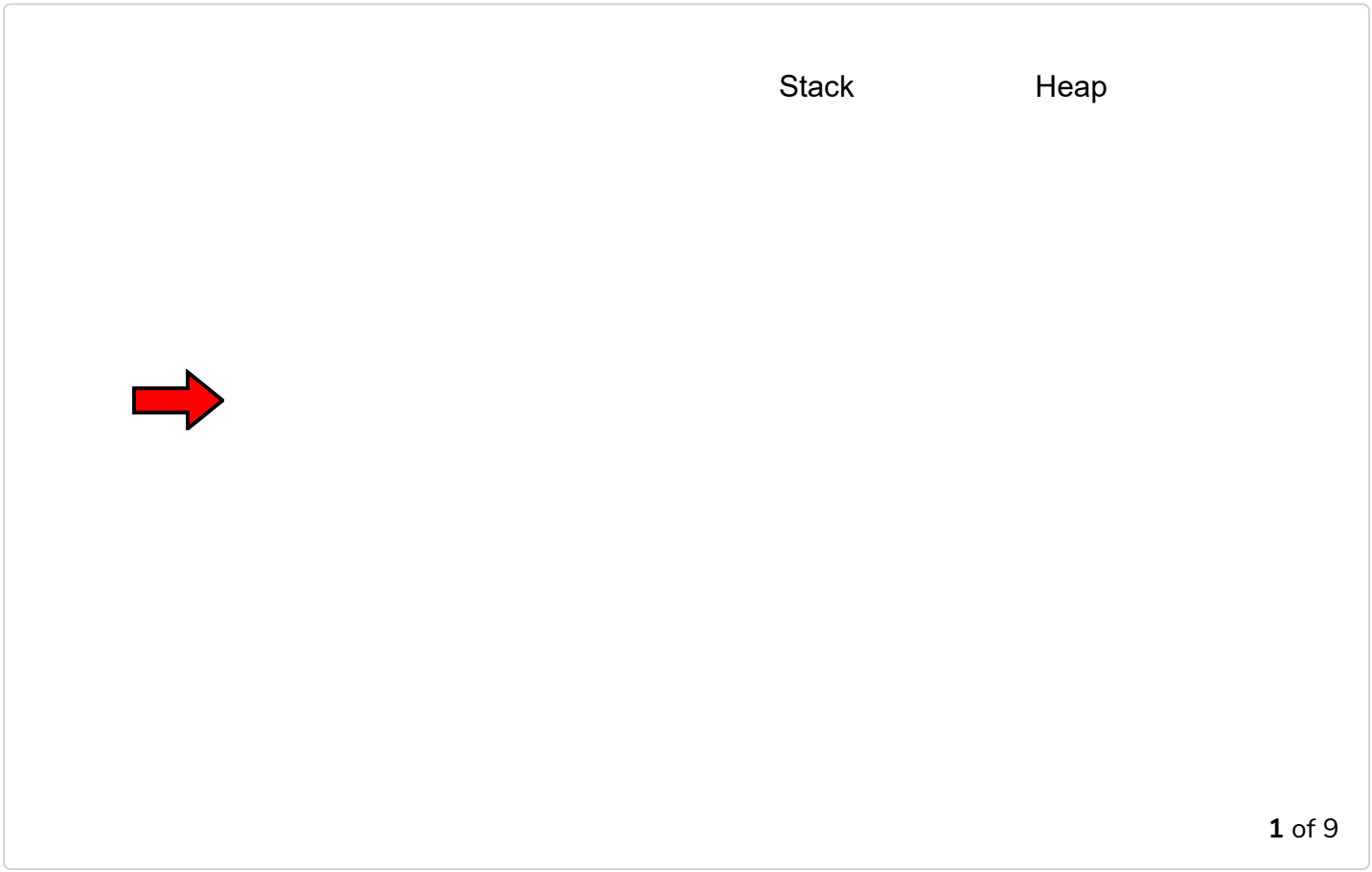
Typically when you think about passing arguments to functions, you think about passing the function the **value** of some variable. A common idiom in C however is to pass function arguments by **reference**, using pointers. This is the case in particular with large data structures like arrays and structs, for which it would be inefficient to make a copy of the whole thing, and passing that copy to a function. Instead, in passing by reference, you simply pass a pointer to the data, to the function.

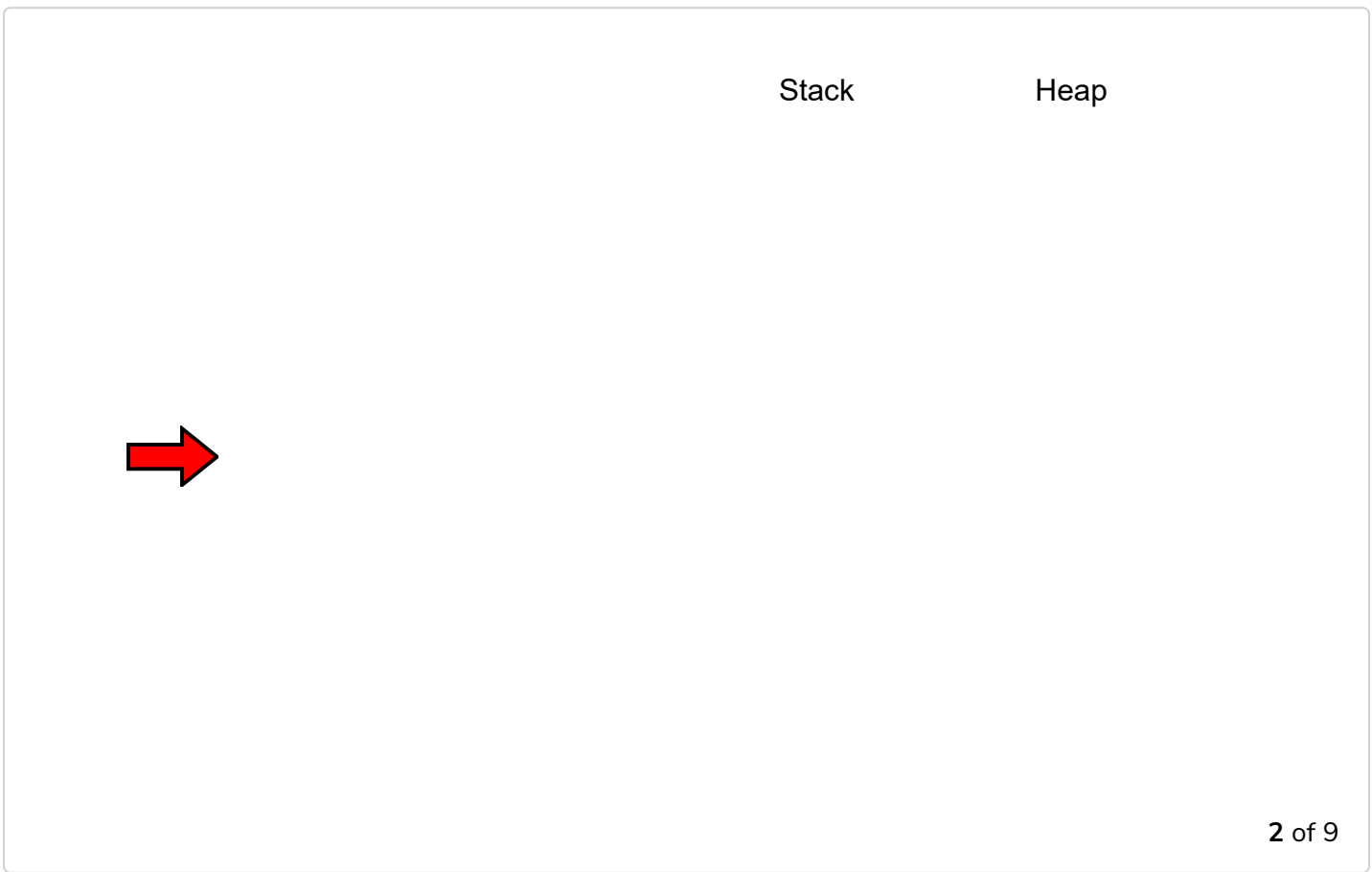
Here is some code illustrating passing by value, first:

```
#include <stdio.h>

void myFun(int x) {
  x = x * 2;
}

int main(void) {
  int y = 50;
```

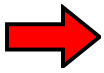
```
  printf("y=%d\n", y);
  myFun(y);
  printf("y=%d\n", y);

  return 0;
}
```







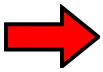
Stack Heap



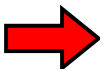
Output: y=50

Stack Heap

Stack Heap



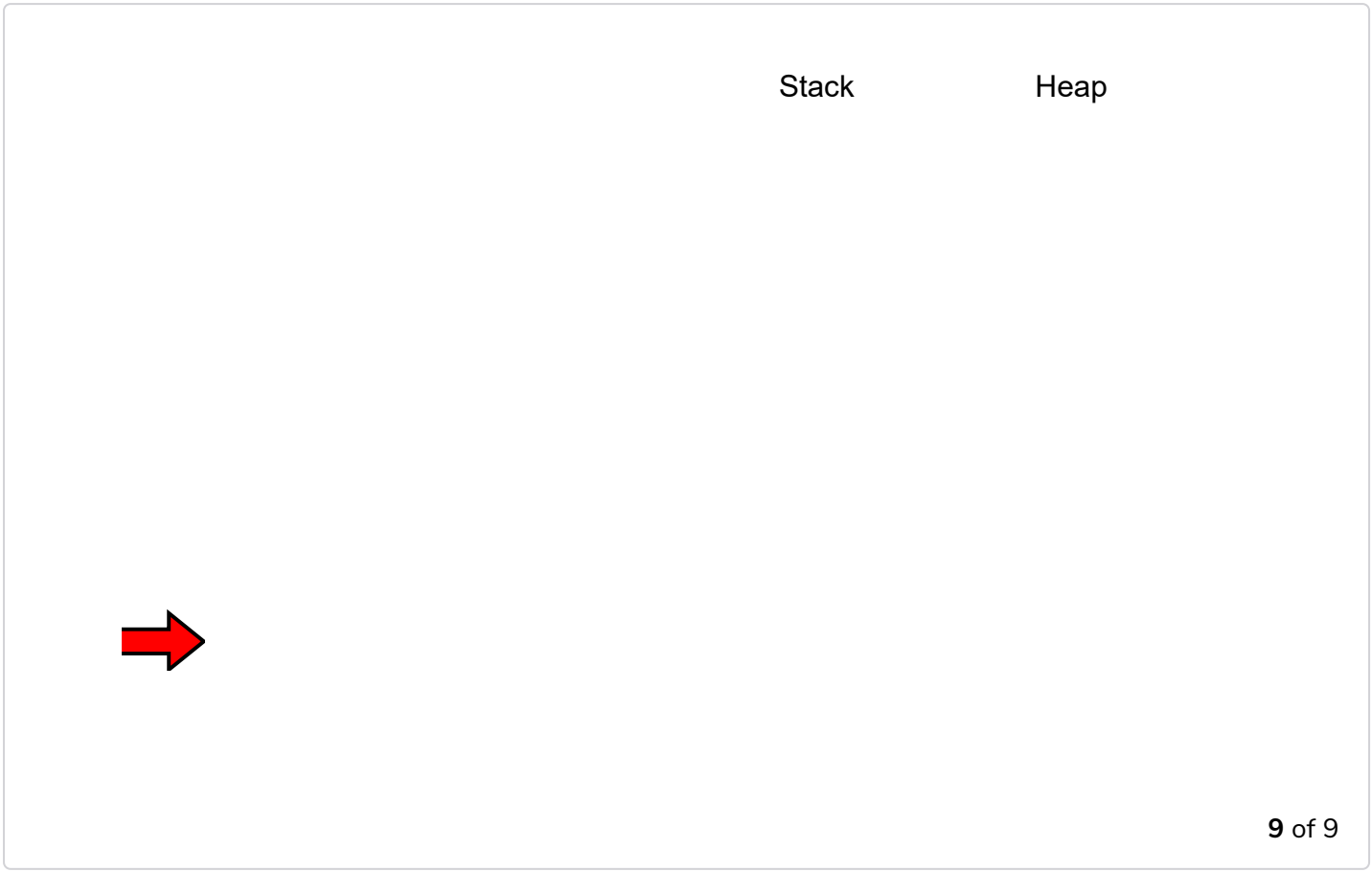
Stack Heap



Stack Heap

Stack Heap

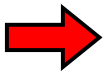
Output: y=50



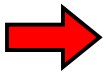
As you can see in the above code example, on line **8**, we assign the value `50` to the variable `y`. Then on line **10** we call the function `myFun()` which takes one `int` argument and we pass it our variable `y`. This is **passing by value** since we are handing over to `myFun()` the **value** of `y` (`50`). Within `myFun()` we multiply the argument passed to it by `2` and exit. In `main()` when we print the value of `y`, after the function call to `myFun()`, it is still 50 (not 100).

```
#include <stdio.h>

void myFun(int *x) {
  *x = *x * 2;
}

int main(void) {
  int y = 50;
  printf("y=%d\n", y);
  myFun(&y);
  printf("y=%d\n", y);
  return 0;
}
```

Stack Heap

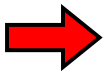


Stack Heap



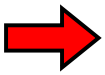
Stack Heap

Output: y=50

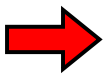


Stack Heap

Stack Heap



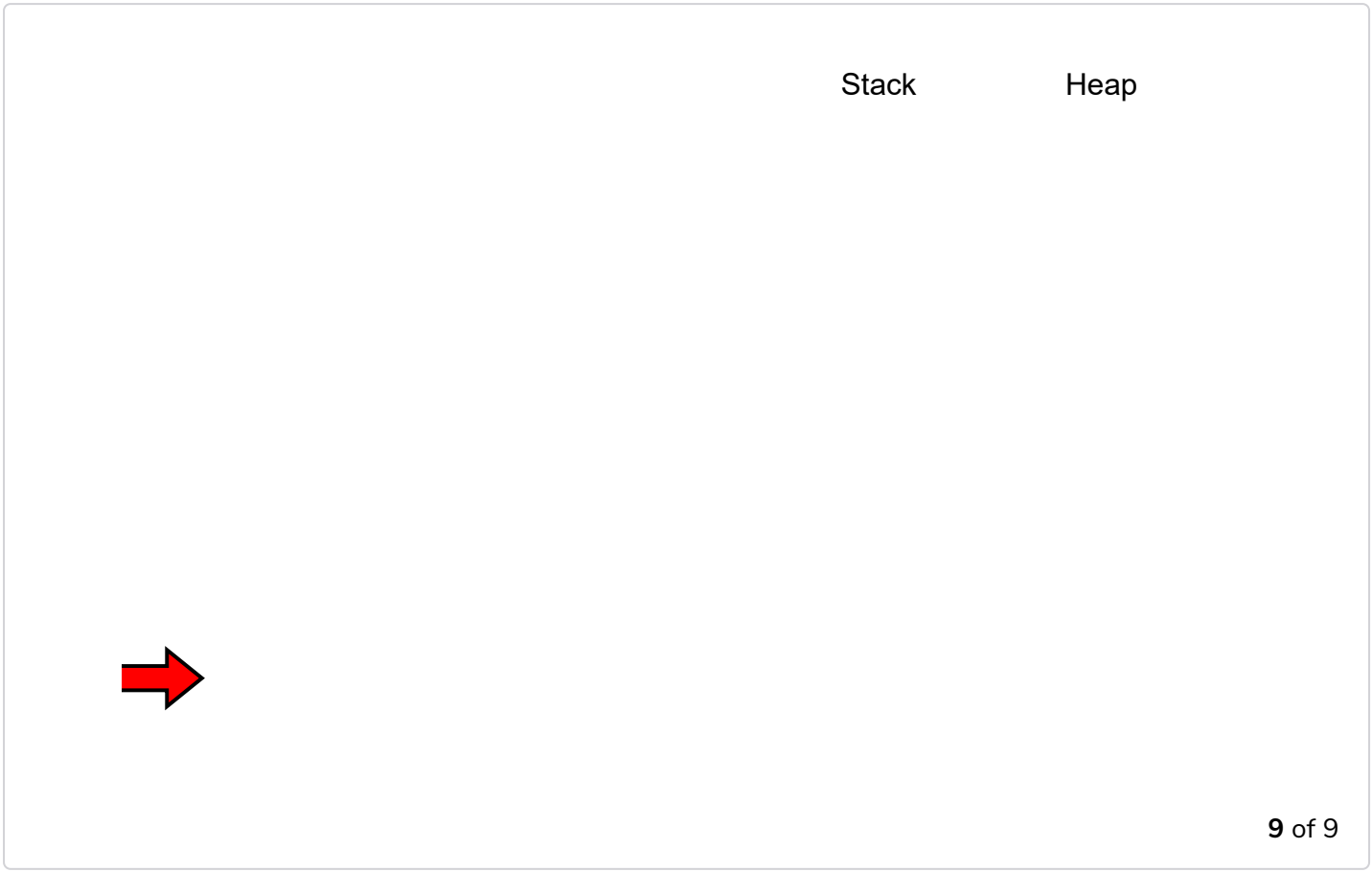
Stack Heap



Stack Heap

Stack Heap

Output: y=100



As you can see in example above, we rewrite `myFun()` so that it take an input argument that is not an `int`, but rather a **pointer to an int** (hence the star * notation). Now in our `main()` function, on line **10**, we pass to `myFun()` not the value of `y` as in the previous code example, but rather the **address** of `y`, using the `&` notation. Now when `myFun()` is called, it uses **pointer dereferencing** to multiply the value **pointed to by** its argument `x`, by `2`. Of course `x` is simply the **address** of `y`, which we passed to `myFun()`, and so the value pointed to by `x` is the value that we assigned to `y`, which is `50`. So `myFun()` multiplies that value by 2 and **assigns** it using pointer dereferencing to `*x`, which is the value associated with `y`.

Make sure you understand these two code examples above, and why they do different things. If you understand this, then you basically understand pointers.

As we discussed before, whatever a pointer points to is part of the Heap. There are several ways to interact with this “dynamic memory”. This will be the topic for our next lesson.