COMP1511 - Programming Fundamentals

Term 1, 2019 - Lecture 15
Stream B

What did we cover last week?

Memory

- Pointers and the idea of what computer memory is

Structs

- Custom variables made up of collections of variables

Professionalism

- Caring about yourself and others in your work

What are we covering today?

Memory

- How functions work in memory
- Direct use of memory in C

Structs and pointers

- Building data structures

Recap - Professionalism

Caring about the people around you and your work

- Communication
- Teamwork
- Resilience
- Technical Skills

- No time like the present to put this into practice!
- Please try to keep your interactions with others as respectful as you can

Recap - Pointers

Pointers

- A pointer is a variable that stores a memory address
- We can assign a memory location to a pointer from a variable
- We can access the memory the pointer is "aiming at"

```
int i = 100;
// create a pointer called ip that points at
// the location of i
int *ip = &i;
printf("The value of the variable at %p is %d", ip, *ip);
```

Recap - Structs

Structs

- A struct is a collection of variables that can be accessed under one name
- They're used to collect custom information together

```
struct fighter {
    char name[20];
    int strength;
    int health;
};
```

Functions and Memory

What actually gets passed to a function?

- Everything gets passed "by value"
- Variables are copied by the function
- The function will then work with their own versions of the variables

What happens to variables passed to functions?

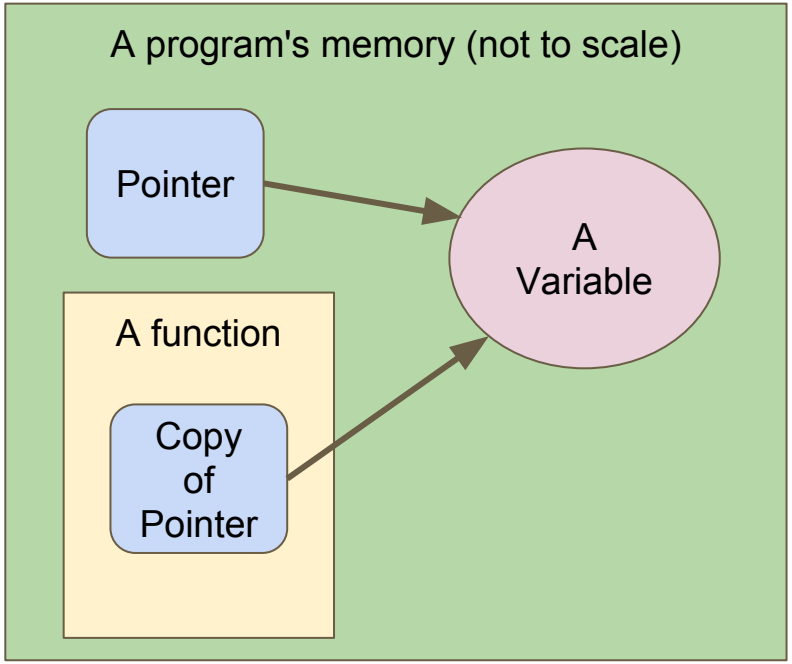
```c
int main (void) {
    int x = 5;
    int y = doubler(x);
    printf("x is %d and y is %d.\n", x, y);
    // "x is 5 and y is 10"
    // this is because the doubler function takes the value 5 from x
    // and copies it into the variable "number" which is a new variable
    // that only lasts as long as the doubler function runs
}

int doubler(int number) {
    number = number * 2;
    return number;
}
```

Functions and Pointers

What happens to pointers that are passed to functions?

- Everything gets passed "by value"
- But the value of a pointer is a memory address!
- The memory address will be copied into the function
- This means **both** pointers are accessing the same variable!



Functions and Pointers

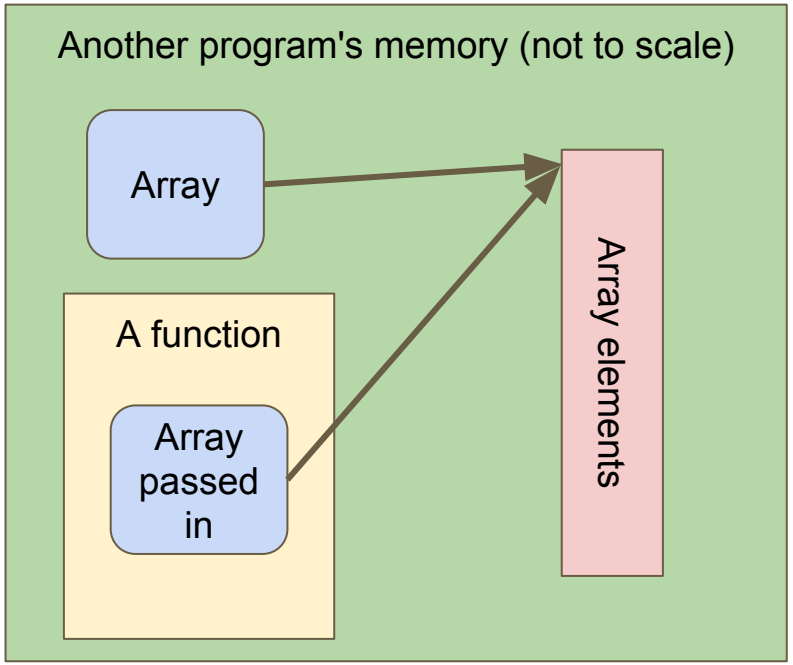
```c
int main (void) {
    int x = 5;
    int *pointerX = &x;
    doublePointer(pointerX);
    printf("x is %d.\n", x);
    // "x is 10"
    // This is because doublePointer gets given access to x via its
    // copied pointer . . . since it changes what's at the other end of
    // that pointer, it affects x
}

// Double the value of the variable the pointer is aiming at
void doublePointer(int *numPointer) {
    *numPointer = *numPointer * 2;
}
```

Arrays are represented as pointers

Arrays and pointers are very similar

- An array is a variable
- It's not actually a variable containing all the elements
- When we use the array variable (no []), it's actually the memory address of the start of the elements
- Arrays and pointers act the same!



Functions and Arrays

```
int main (void) {
    int myNums[3] = {1,2,3};
    doubleAll(3, myNums);
    printf("Array is: ");
    int i = 0;
    while(i < 3) {
        printf("%d ", myNums[i]);
        i++;
    }
    printf("\n");
    // "Array is 2 4 6"
    // Since passing an array to a function will pass the address
    // of the array, any changes made in the function will be made
    // to the original array
}
```

Functions and Arrays continued

```
// Double all the elements of a given array
void doubleAll(int length, int numbers[]) {
    int i = 0;
    while(i < length) {
        numbers[i] = numbers[i] * 2;
        i++;
    }
}
```

Break Time

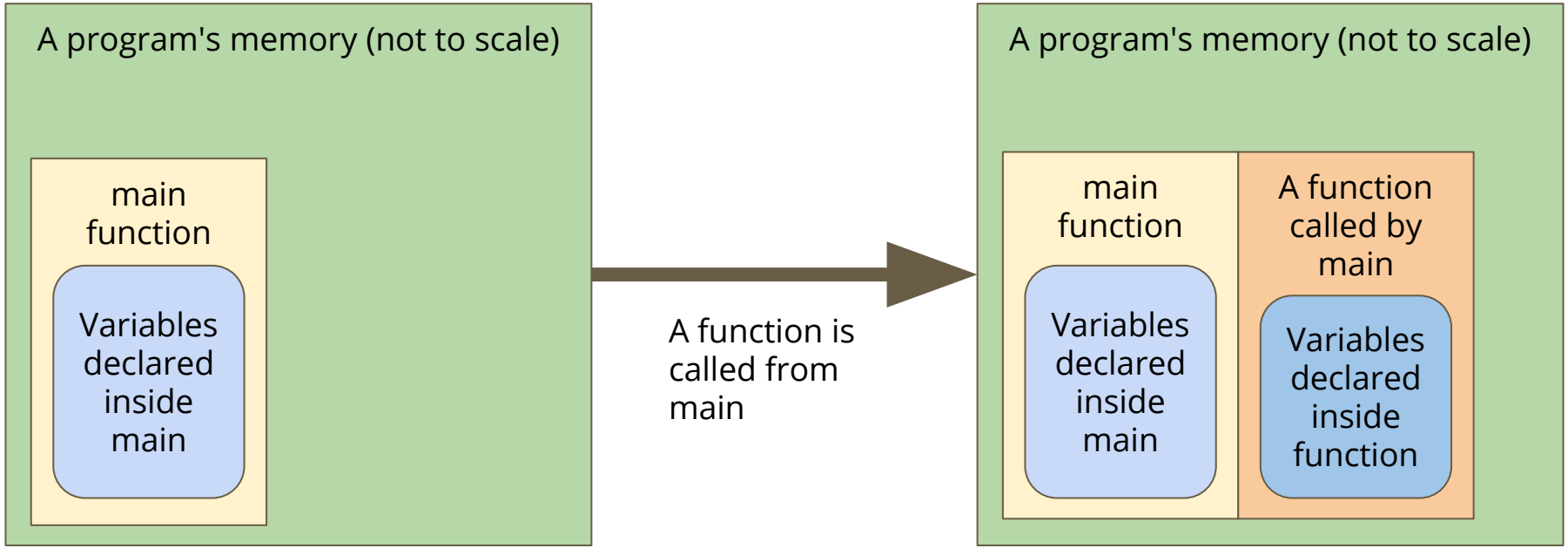
We hope everyone learnt something new while working on Coco

- Remember that competition success and being a good programmer are not necessarily correlated!

  "I don't care who you are, where you're from, what you've done . . . as long as you love C." - The Backstreet Boys

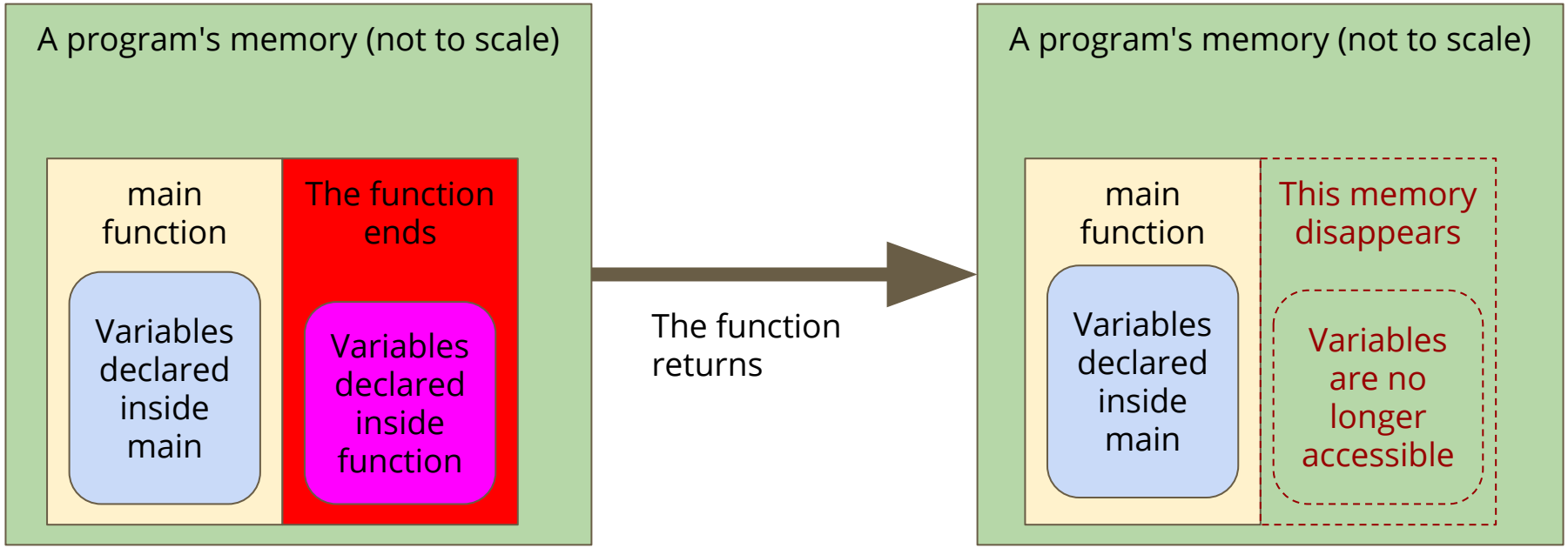
Memory in Functions

What happens to variables we create inside functions?



Memory in Functions

What happens to variables we create inside functions?



Keeping memory available

What if we want to create something in a function?

- We often want to run functions that create data
- We can't always pass it back as an output

```
// Make a number and return a pointer to them
int *createNumber() {
    int number = 10;
    return &number;
}
// This example will return a pointer to memory that we no longer have!
```

Memory Allocation

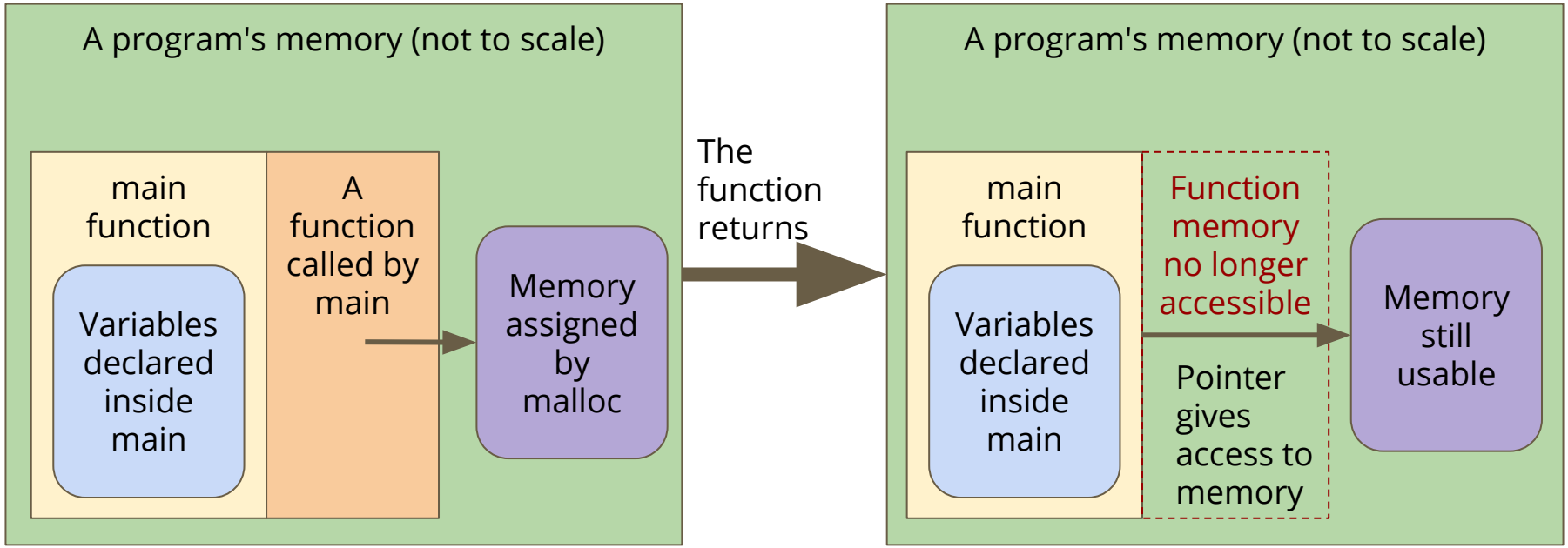
C has the ability to allocate memory

- A function called **malloc**(bytes) returns a pointer to memory
- Allows us to take control of a block of memory

- This won't automatically be cleaned up when a function ends
- To clean up the memory, we call **free**(pointer)
- free() will use the pointer to find our previous memory to clean it up

What malloc() does

Using malloc, we can assign some memory that is not tied to a function



Malloc() in code

We can assign a particular amount of memory for use

- The function sizeof() allows us to see how many bytes something needs
- We can use sizeof() to allocate the correct amount of memory

```
// Allocate memory for a number and return a pointer to them
int *mallocNumber() {
    int *intPointer = malloc(sizeof(int));
    *intPointer = 10;
    return intPointer;
}
// This example will return a pointer to memory we can use
```

Cleaning up after ourselves

Allocated memory is never cleaned up automatically

- We need to remember to use free()
- Every pointer that is aimed at allocated memory must be freed!

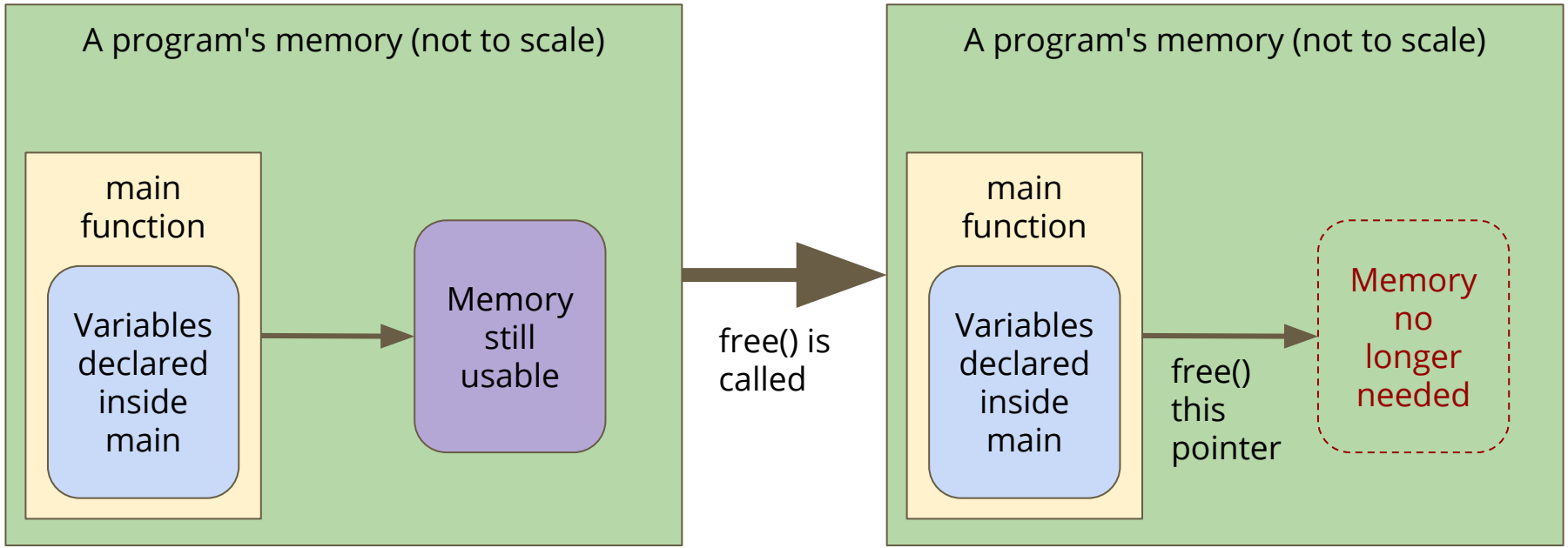
```
// Allocate memory for a number and return a pointer to them
int main(void) {
    int *iPointer = mallocNumber();

    *iPointer += 25;

    free(iPointer);
    return 0;
}
```

Freeing up memory

Calling free will clean up the allocated memory that we're finished with



Using memory

Some things to think about with malloc() and free()

- You can use sizeof() to figure out how many bytes something needs
- We can malloc arrays and structs as well as variables
- In general, always use sizeof() with malloc()

- Anything allocated with malloc() must be free() after you've finished with it
- Otherwise we get what's known as memory leaks!

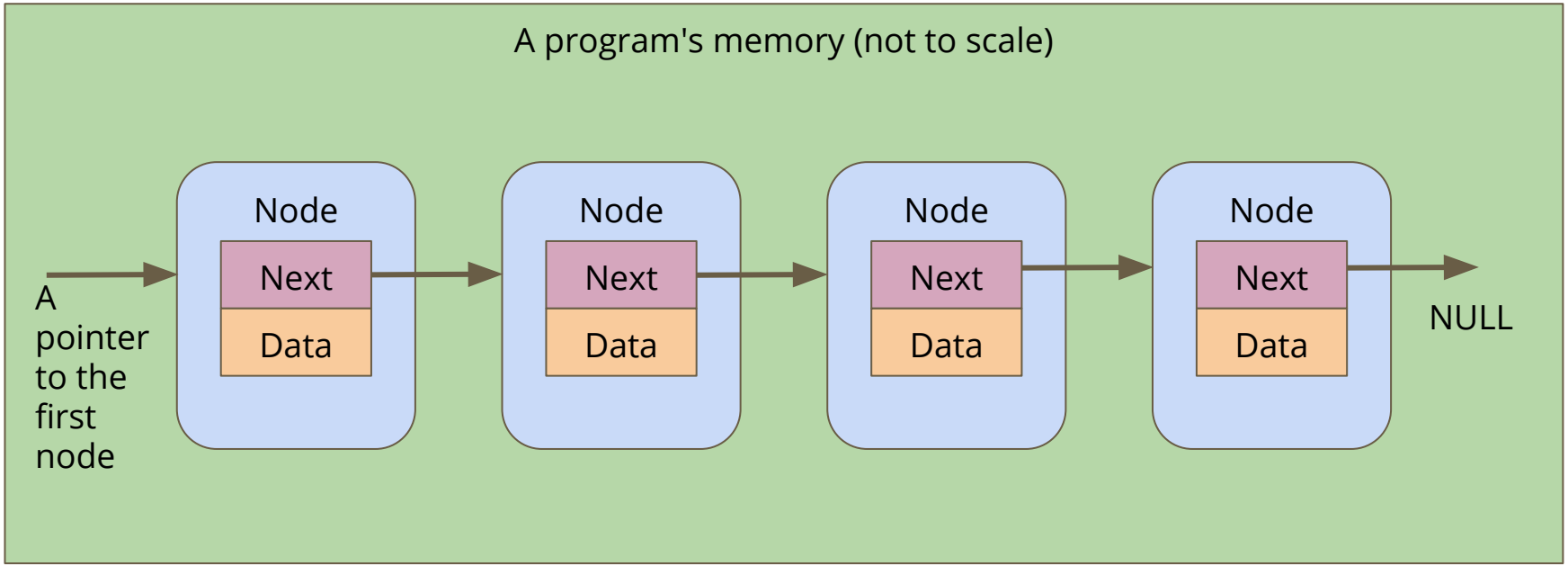
A new kind of struct

Let's make an interesting struct

- This is a node
- It contains some information
- As well as a pointer to another node!

```
struct node {
    struct node *next;
    int data;
}
```

A Chain of Nodes - a Linked List



Linked Lists

A chain of these nodes is called a Linked List

As opposed to Arrays . . .

- Not one continuous block of memory
- Items can be shuffled around by changing where pointers aim
- Length is not fixed when created
- You can add or remove items from inside the list

Let's make a simple Linked List

What do we need?

- A struct for a node
- A pointer to keep track of the start of the list
- A way to create a node and connect it

A function to add a node

```
// Create a node using the data and next pointer provided
// Return a pointer to this node
struct node *createNode(int data, struct node *next) {
    struct node *n;
    // allocate the memory for a single node
    n = malloc(sizeof (struct node));
    if (n == NULL) {
        // malloc returns NULL if there isn't enough memory
        // terminate the program
        fprintf(stderr, "out of memory\n");
        exit(1);
    }
    n->data = data;
    n->next = next;
    return n;
}
```

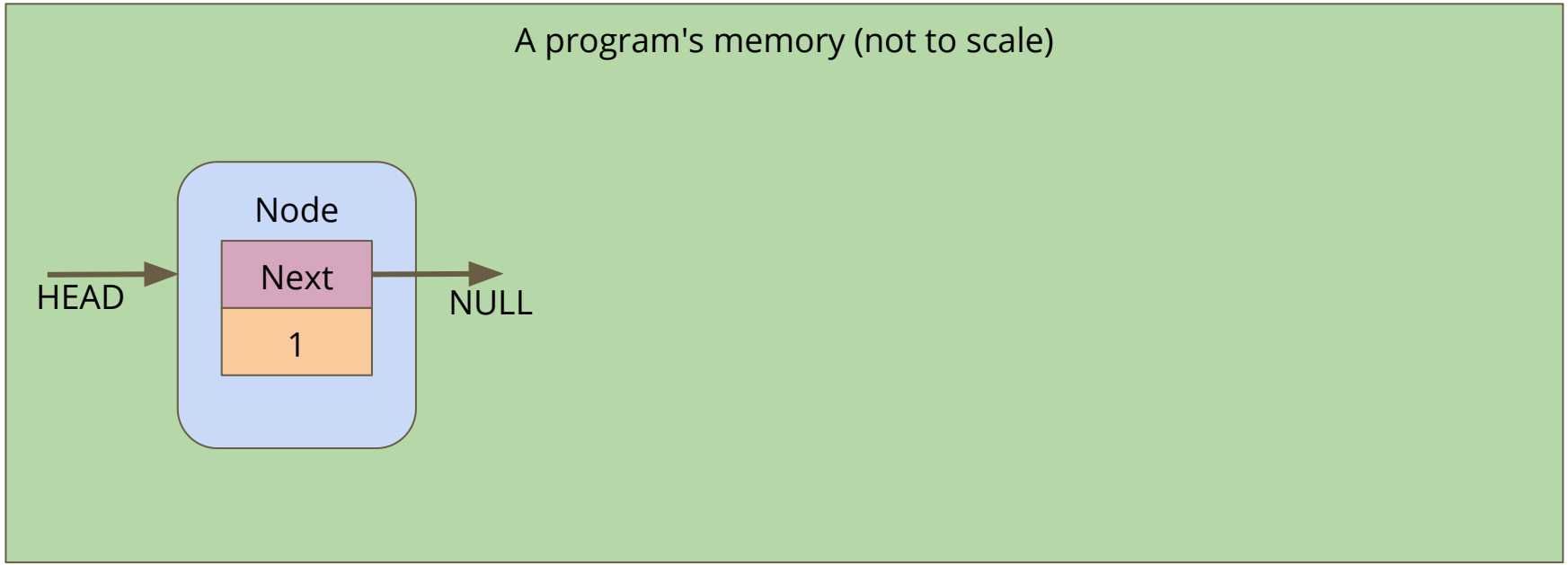
Building a list from createNode()

```
int main (void) {
    // head will always point to the first element of our list
    struct node *head = createNode(1, NULL);
    head = createNode(2, head);
    head = createNode(3, head);
    head = createNode(4, head);
    head = createNode(5, head);

    return 0;
}
```

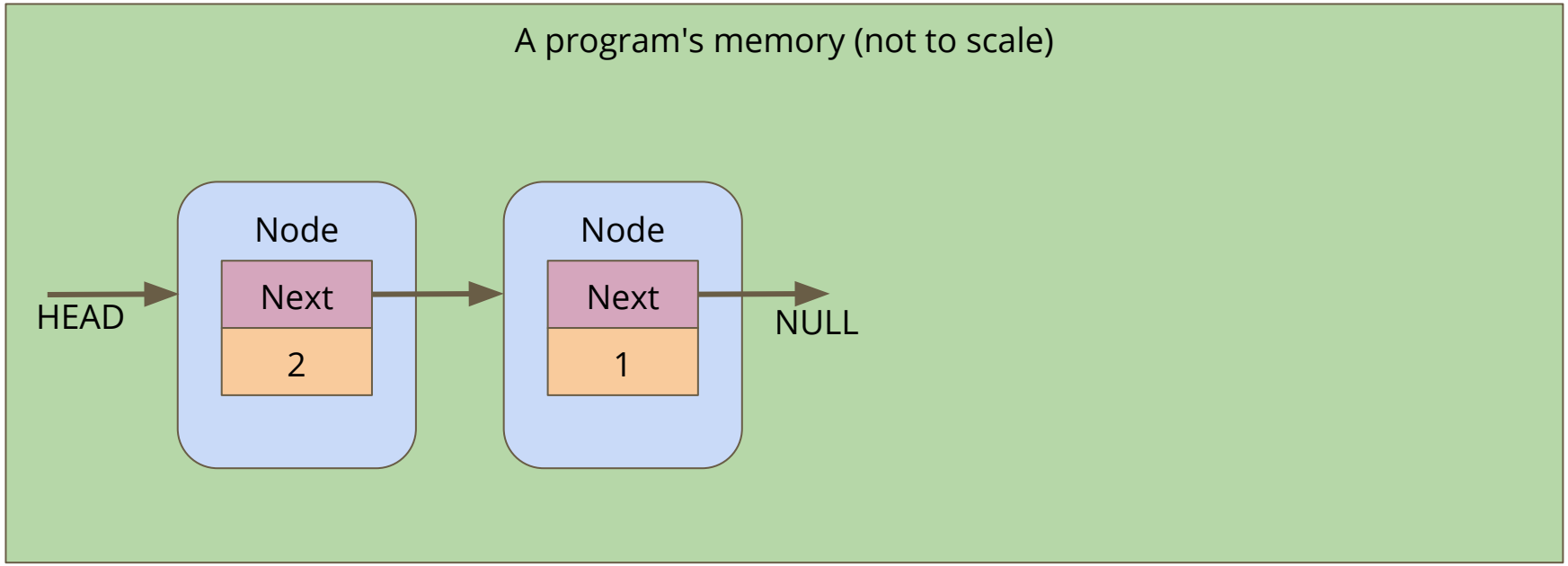
How it works 1

CreateNode makes a node with a NULL next and we point head at it



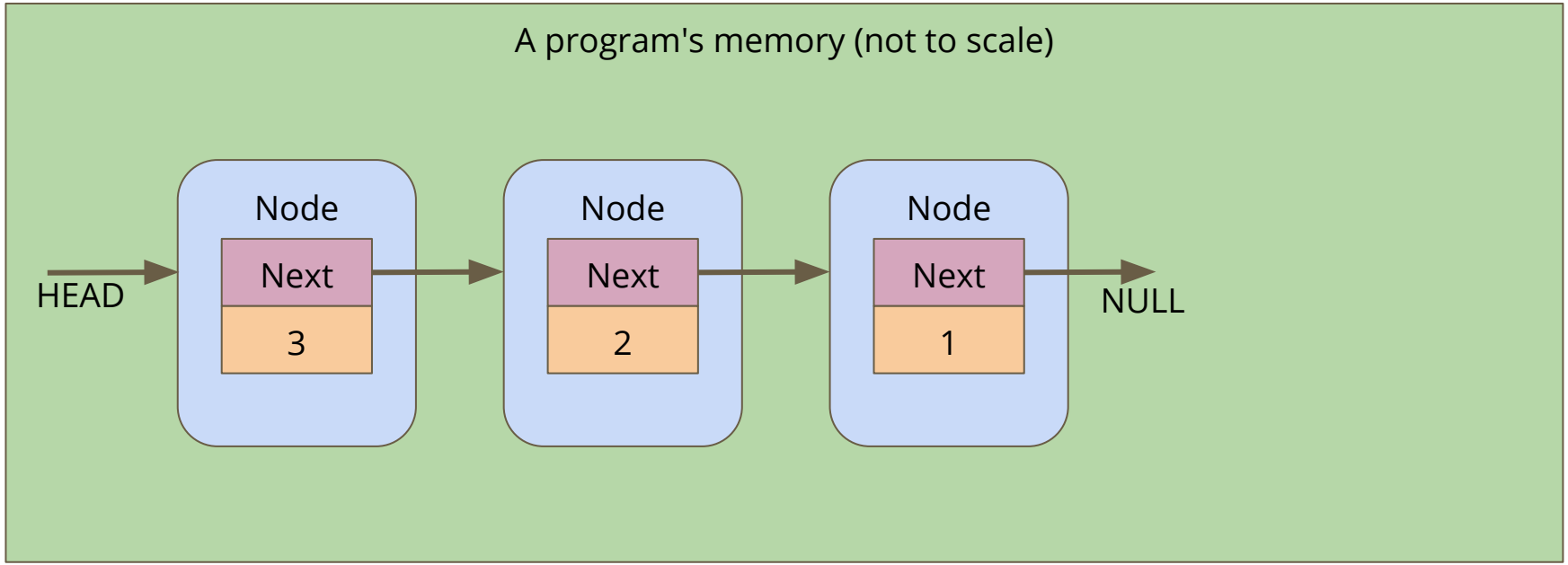
How it works 2

The 2nd node points its "next" at the old head, then it replaces head with its own address



How it works 3

The process continues . . .



We're not finished here . . .

To be continued on Thursday!

- We haven't actually used our list yet
- We'll want to be able to traverse the list
- We also want to add and remove objects

What did we learn today?

Functions and Memory

- How functions have their own piece of memory
- How we lose access to anything in a function once it returns
- How we can specifically allocate memory

Linked Lists

- We've seen a node that can point at another node
- This forms a chain of nodes known as a Linked List