COMP1511 - Programming Fundamentals

Week 7 - Lecture 12

What did we learn last lecture?

Memory

- Using memory beyond what's in our functions
- Allocating memory so that it lasts beyond the lifetime of the curly brackets

Multiple File Projects

- Using Header (*.h) and Implementation (*.c) files
- Protecting our data by hiding it
- Providing a nice interface with header functions

What are we covering today?

Command Line Arguments

- Adding information to our program when it runs

Linked Lists

- Like an array, contains multiple of the same type of variable
- More flexible in that it can change length
- Is also able to add and remove elements from partway through the list
- Tying together structs, pointers and memory allocation

Characters and Strings Recap

Our new variable type: `char`

- Represents a letter
- Is also a number, an ASCII code, and we'll often use `int`s to represent a character
- When used in arrays, they're referred to as strings
- Strings often end before the end of the array they're stored in
- When they do, we store a null terminator `'\0'` after the last character

Strings in Code

Strings are arrays of type char, but they have a convenient shorthand

```
// a string is an array of characters
char word1[] = {'h','e','l','l','o','\0'};
// but we also have a convenient shorthand
// that feels more like words
char word2[] = "hello";
```

Both of these strings will be created with 6 elements. The letters `h,e,l,l,o` and the null terminator `\0`

h	e	l	l	o	\0

Command Line Arguments

Sometimes we want to give information to our program at the moment when we run it

- The **"Command Line"** is where we type in commands into the terminal
- **Arguments** are another word for input parameters

```
$ ./program extra information 1 2 3
```

- This extra text we type after the name of our program can be passed into our program as strings

Main functions that accept arguments

`int main` doesn't have to have `void` input parameters!

```
int main(int argc, char *argv[]) {
}
```

- **argc** will be an "argument count"
- This will be an integer of the number of words that were typed in (including the program name)
- **argv** will be "argument values"
- This will be an array of strings where each string is one of the words

An example of use of arguments

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int i = 1;
    printf("Well actually %s says there's no such thing as ", argv[0]);
    while (i < argc) {
        fputs(argv[i], stdout);
        printf(" ");
        i++;
    }
    printf("\n");
}
```

Arguments in argv are always strings

But what if we want to use things like numbers?

- We can read the strings in, but we might want to process them

```
$ ./program extra information 1 2 3
```

- In this example, how do we read 1 2 3 as numbers?
- We can use a library function to convert the strings to integers!
- **`strtol()`** - "string to long integer" is from the stdlib.h

Code for transforming strings to ints

Adding together the command line arguments

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    int total = 0;

    int i = 1;
    while (i < argc) {
        total += strtol(argv[i], NULL, 10);
        i++;
    }
    printf("Total is %d.\n", total);

}
```

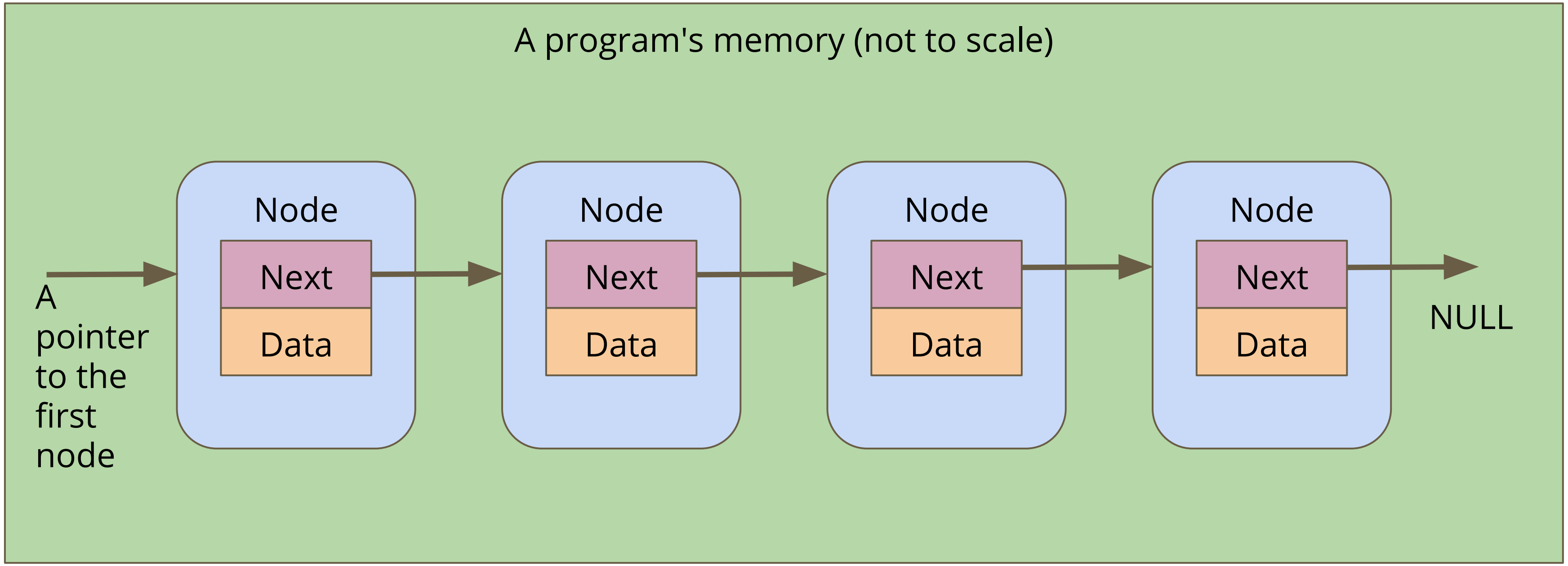
A new kind of struct

Let's make an interesting struct

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

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

A Chain of Nodes - a Linked List


A program's memory (not to scale)
A pointer to the first node
Node
Next
Data
Node
Next
Data
Node
Next
Data
Node
Next
Data
NULL

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 anywhere in the list

Linked Lists in code

What do we need for the simplest possible list?

- 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

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

A function to add a node

```c
// 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;
    n = malloc(sizeof(struct node));
    if (n == NULL) {
        // malloc returns NULL if there isn't enough memory
        // terminate the program
        printf("Cannot allocate node. Program will exit.\n");
        exit(1);
    }
    n->data = data;
    n->next = next;
    return n;
}
```

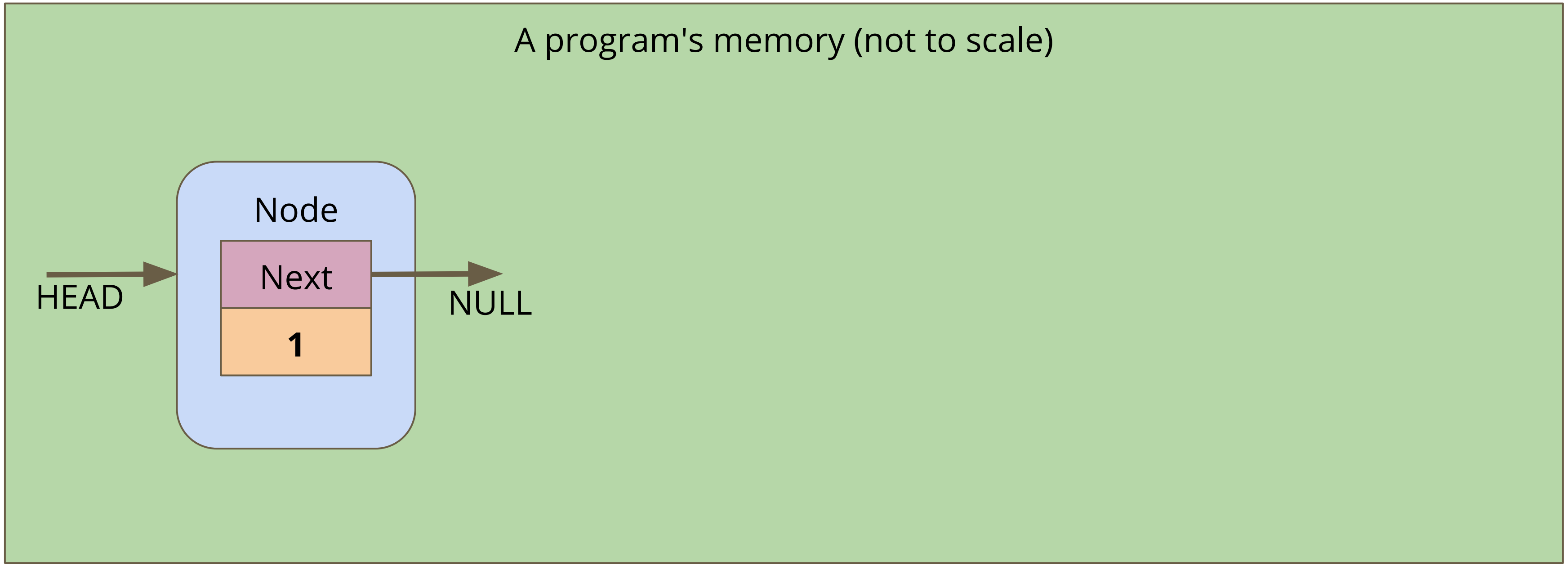
Building a list from createNode()

```
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;
}
```

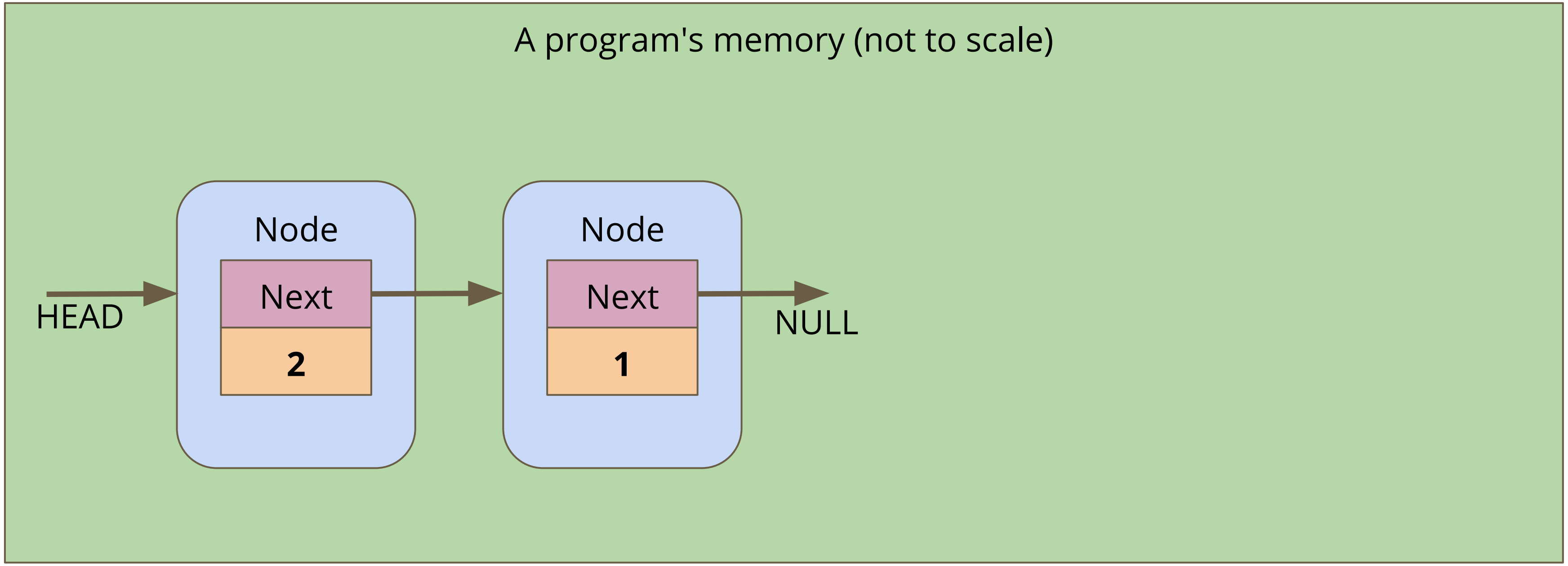
How it works 1

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



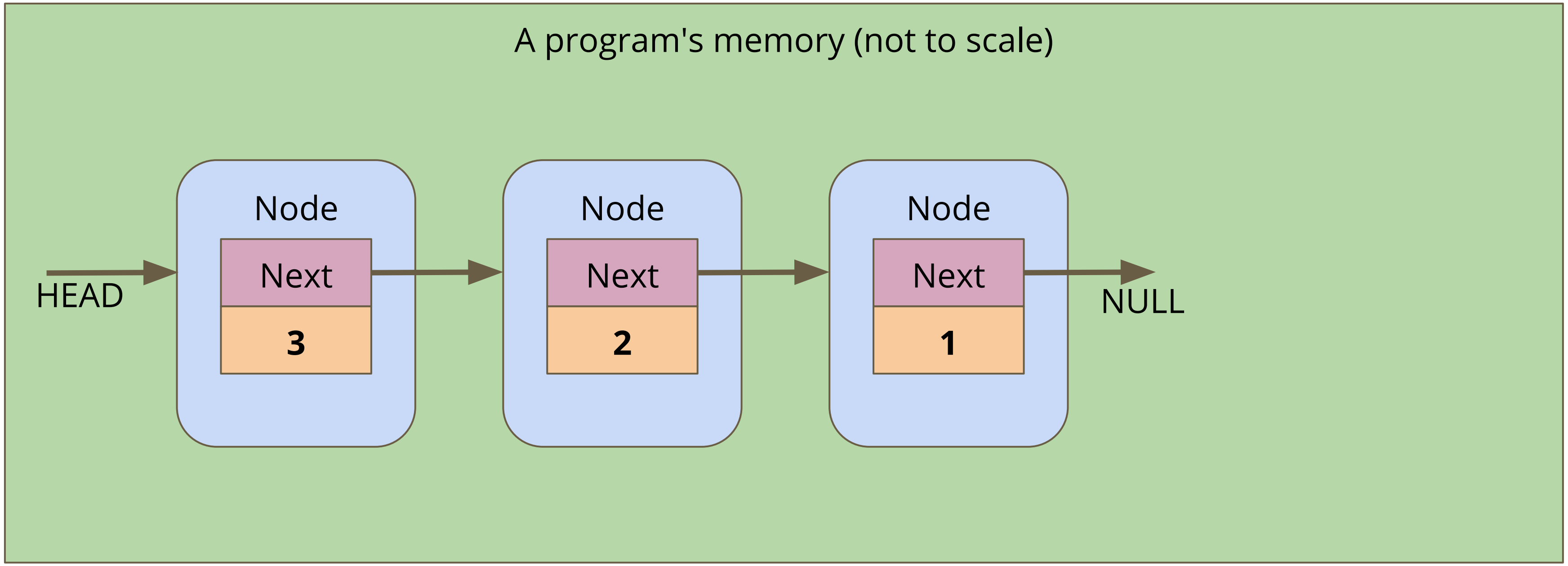
How it works 2

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



How it works 3

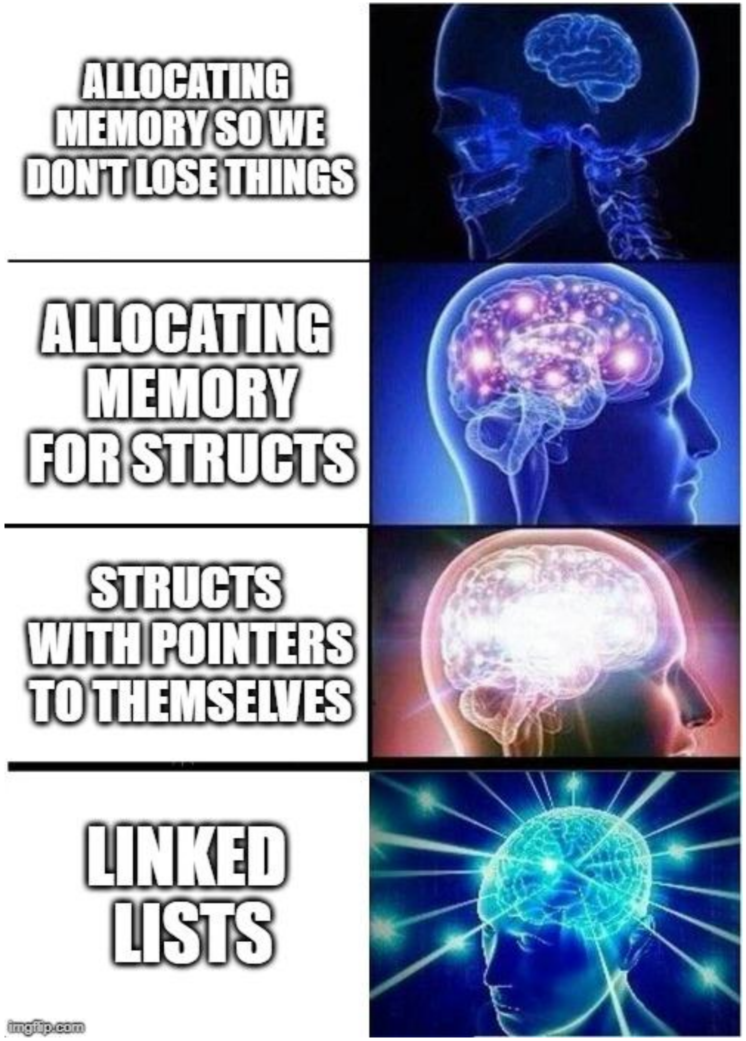
The process continues . . .



Break Time

Linked Lists

- Pointers, structs and memory allocation
- Structs with pointers to their own type
- Linked Lists combine a lot of our newer code techniques


ALLOCATING MEMORY SO WE DON'T LOSE THINGS
ALLOCATING MEMORY FOR STRUCTS
STRUCTS WITH POINTERS TO THEMSELVES
LINKED LISTS
imgflip.com

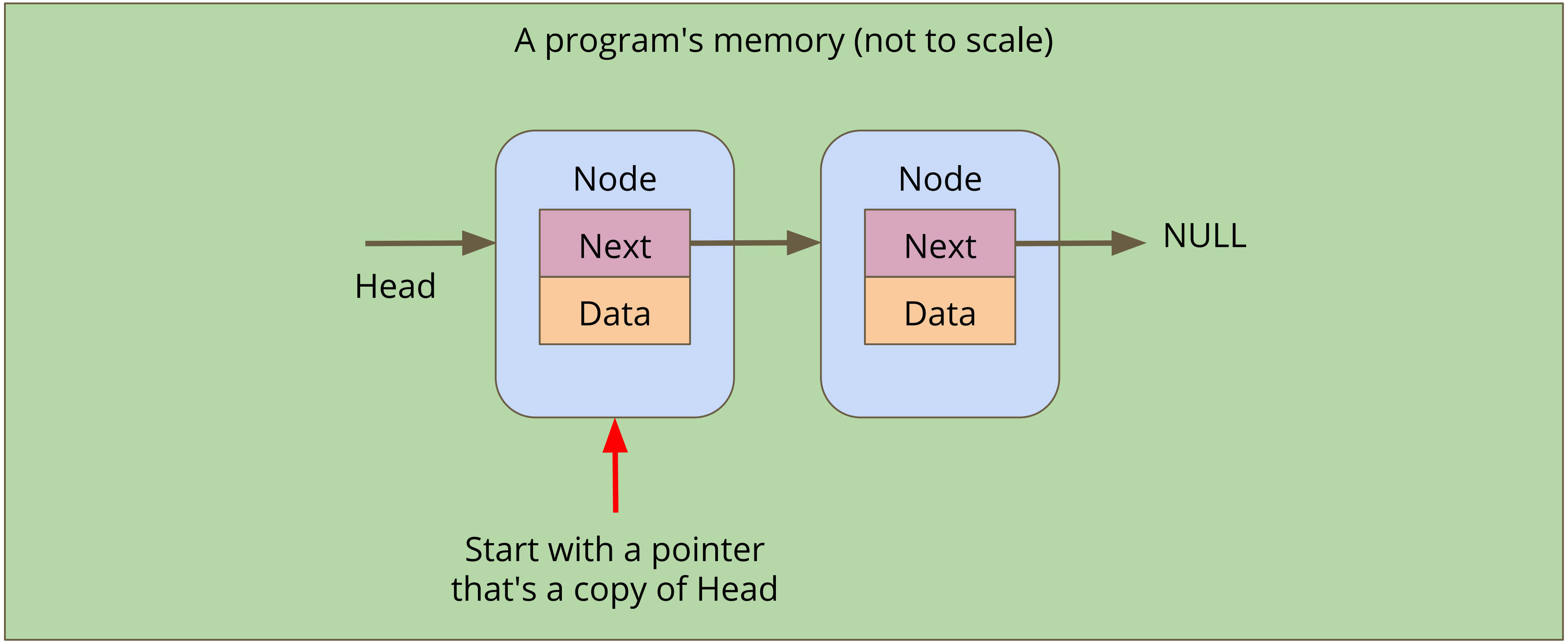
Looping through a Linked List

Linked lists don't have indexes . . .

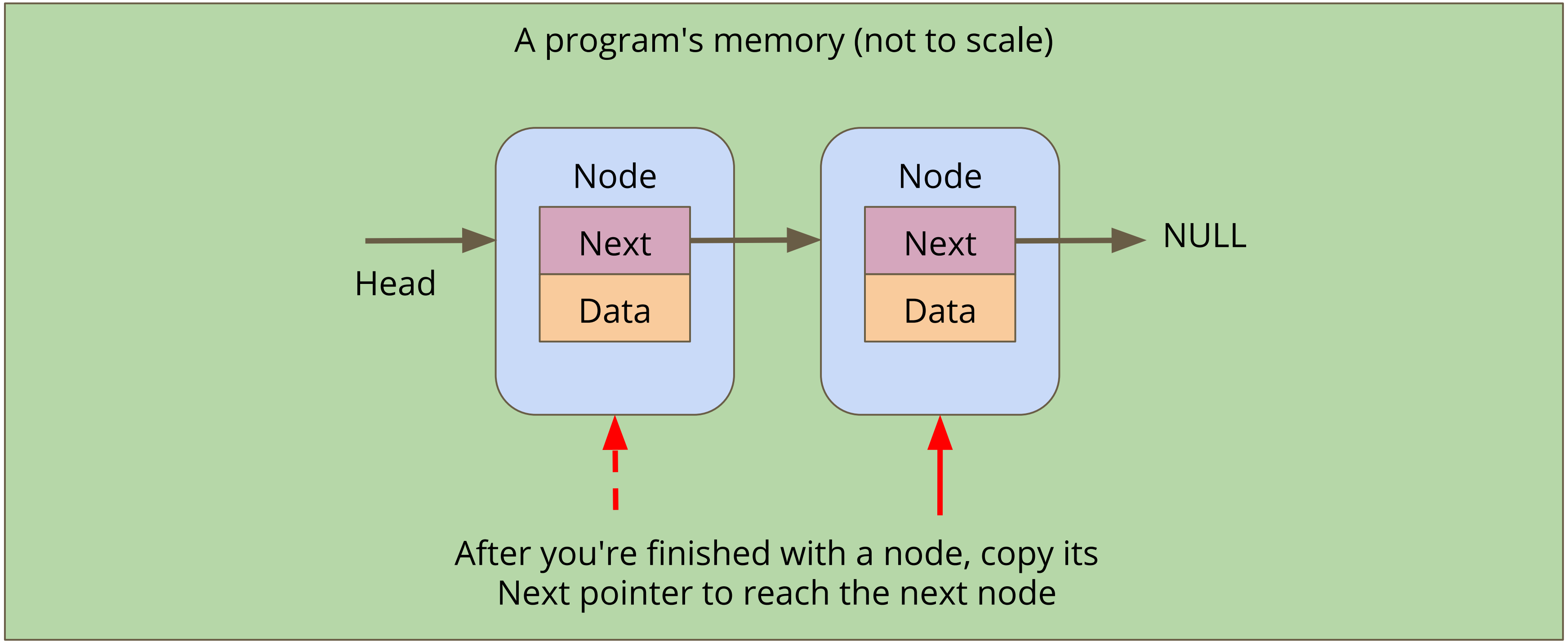
- We can't loop through them in the same way as arrays
- We have to follow the links from node to node
- If we reach a NULL node pointer, it means we're at the end of the list

```c
// Loop through a list of nodes, printing out their data
void printData(struct node *n) {
    while (n != NULL) {
        printf("%d\n", n->data);
        n = n->next;
    }
}
```

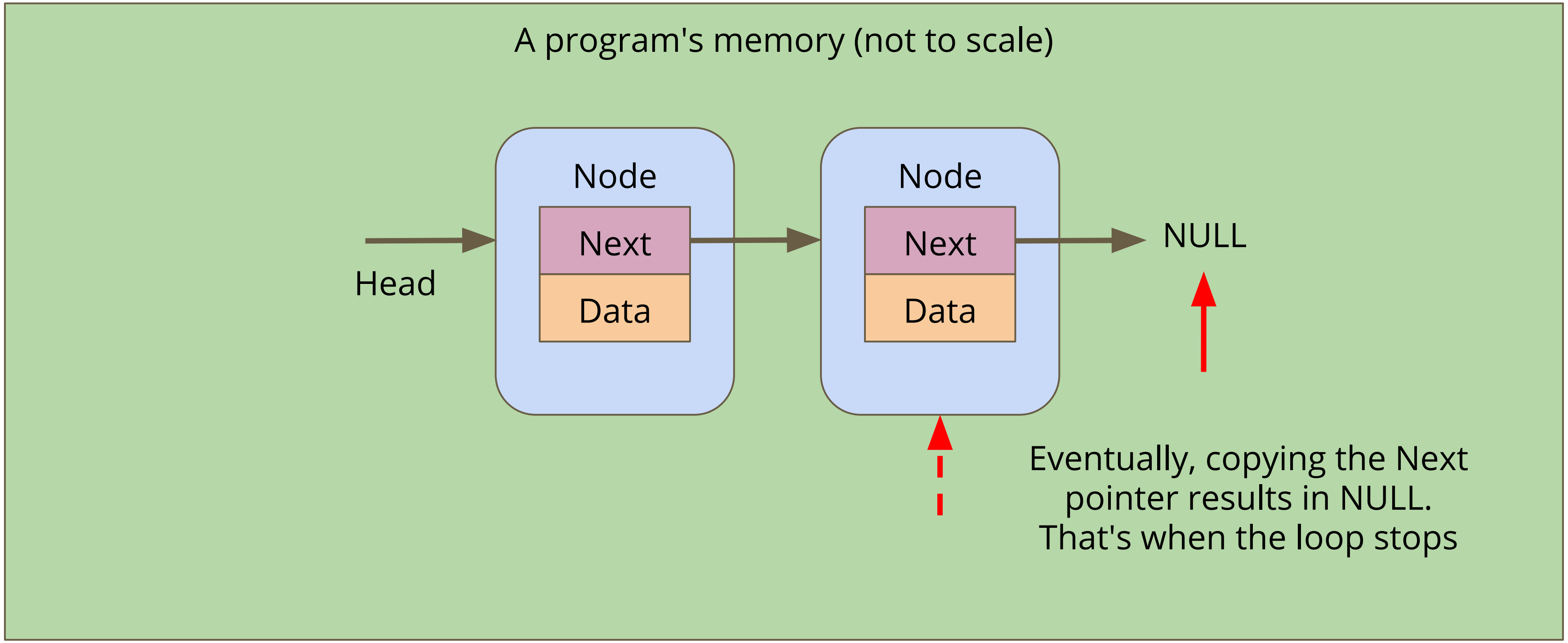
Looping through a Linked List


A program's memory (not to scale)
Node
Next
Data
Node
Next
Data
Head
NULL
Start with a pointer that's a copy of Head

Looping through a Linked List


A program's memory (not to scale)
Node
Next
Data
Node
Next
Data
Head
NULL
After you're finished with a node, copy its
Next pointer to reach the next node

Looping through a Linked List



Eventually, copying the Next pointer results in NULL. That's when the loop stops

Battle Royale

Let's use a Linked List to track the players in a game

- We're going to start by adding players to the game
- We want to be able to print all the players that are currently in the game (the list of players can change as the game goes on)
- We might want to control the order of the list, so we need to be able to insert at a particular position
- We also want to be able to find and remove players from the list if they're knocked out of the round

What will our nodes look like?

We're definitely going to want a basic node struct

- Let's start with a name
- And a pointer to the next node

```
struct node {
    char name[MAX_NAME_LENGTH];
    struct node *next;
};
```

Creating nodes

We'll want a function that creates a node

```c
// Create a node using the name and next pointer provided
// Return a pointer to this node
struct node *createNode(char newName[], struct node *newNext) {
    struct node *n;
    n = malloc(sizeof (struct node));
    if (n == NULL) {
        printf("Malloc failed, out of memory\n");
        exit(1);
    }
    strcpy(n->name, newName);
    n->next = newNext;
    return n;
}
```

Creating the list itself

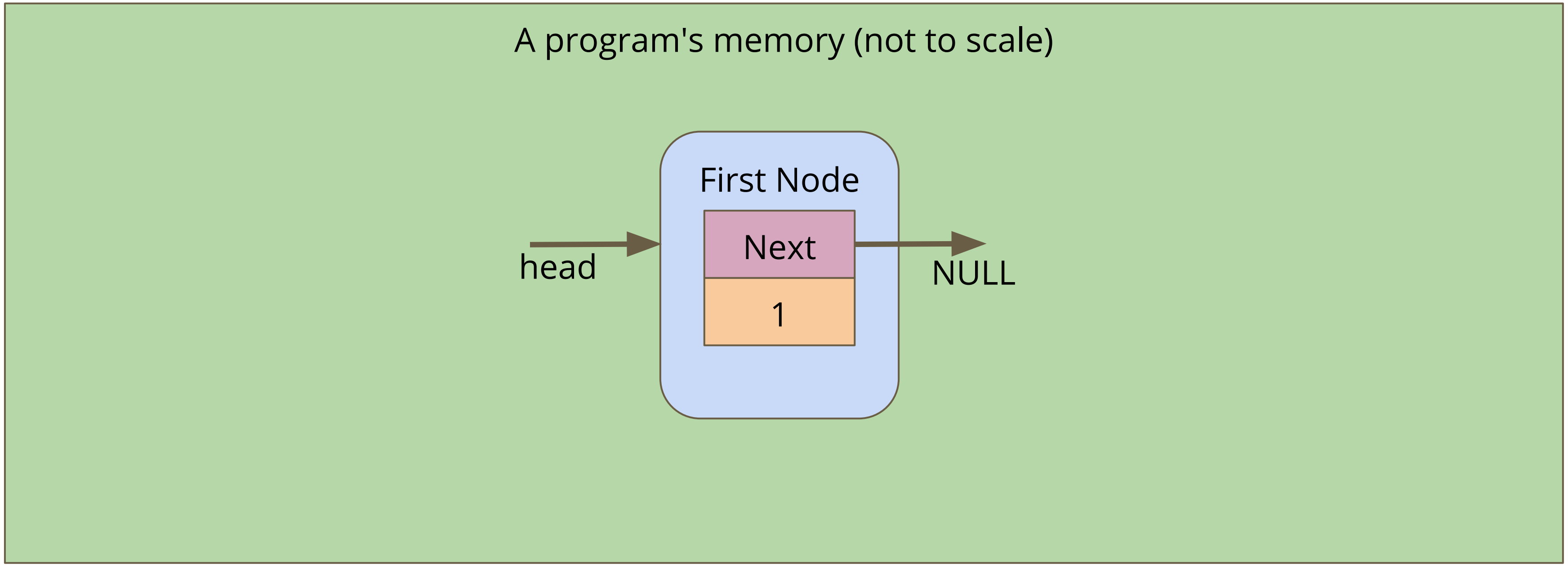
Note that we don't need to specify the length of the list!

```
int main(void) {
    // create the list of players
    struct node *head = createNode("Marc", NULL);
    head = createNode("AndrewB", head);
    head = createNode("Tom", head);
    head = createNode("Aang", head);
    head = createNode("Sokka", head);

    return 0;
}
```

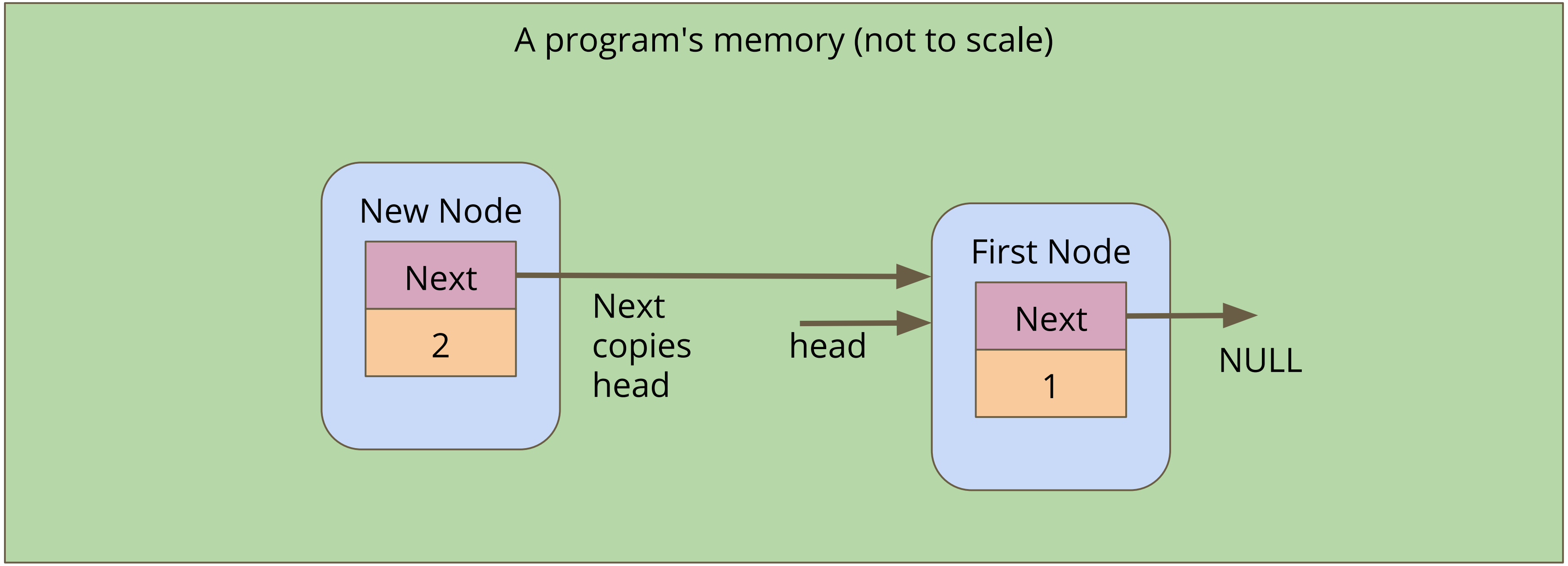
Using createNode

Head points at the First Node, its next is NULL



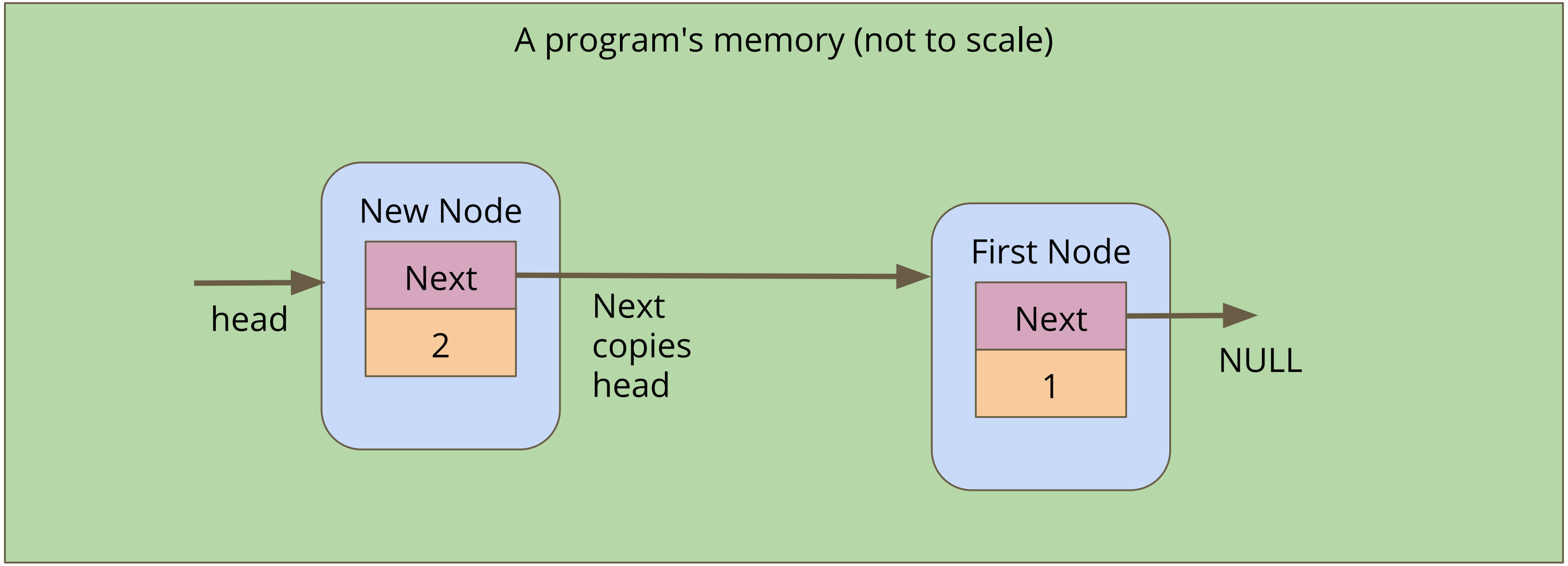
Using createNode

The New Node is created and copies the head pointer for its next



Using createNode

createNode returns a pointer to New Node, which is assigned to head



Printing out the list of players

How do we traverse a list to see all the elements in it?

- Loop through, starting with the pointer to the head of the list
- Use whatever data is inside the node
- Then move onto the next pointer from that node
- If the pointer is NULL, then we've reached the end of the list

```c
// Loop through the list and print out the player names
void printPlayers(struct node* listNode) {
    while (listNode != NULL) {
        printf("%s\n", listNode->name);
        listNode = listNode->next;
    }
}
```

To be continued

It's a big project . . . we'll continue it later!

- We might want to insert at a different place in the list
- We still want to insert for a reason (thinking about keeping lists sorted)
- We haven't yet looked at removal from a list
- Once we have all the functionality we need, we'll actually run the game

What did we learn today?

Command Line Arguments

- Taking information as the program is run

Linked Lists

- A new struct that can point at its own type
- Chaining nodes together forms a list
- Nodes can have a variety of information in them
- Code for creation of nodes and lists
- Looping through the lists