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

Structs

- Our custom variables
- Made up of other variables

What are we covering today?

Multi-File Projects

- Spreading a program over multiple files

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

C Projects with Multiple Files

For readability and also to separate code by subject

- We've already seen **`#include`**
- We can also **`#include`** our own files!
- This allows us to join projects together

Reusable sub-projects

- We'll often make some code that we can use again
- If we make it in its own file, with its own interface, we can **`#include`** it in our other projects

Header Files and C (Implementation) Files

Two different files for different purposes

- Header and C files usually go together in pairs

Header *.h file

- Shows the capabilities of a code file
- Enough to use it without needing to understand what's in it

C Implementation *.c file

- Contains the underlying implementation of the H file

File.h

Header Files show you what the code's functions are

- This file shows a programmer all they need to know to use our code
- **`typedef`** (Type Define) is a way of allowing us to create our own C Type out of another Type
- This protects our struct from access and keeps our data safe!
- Function Declarations with no definitions
- Comments that describe how the functions can be used
- No running code!

File.c

Implementation Files show you how the code runs in detail

- We can hide the complicated running code in this file
- Has includes, especially **`#include "File.h"`** (joins the two files together)
- Implements the **`struct`** mentioned in the **`typedef`** from the header
- Implements all the functions declared in the header

Main.c and other Files

Our Entry Point into our code

- The **`main()`** function is always what runs first
- For any code file (*.c) to use the functionality provided by another file, it must **`#include`** that file
- In our example, main.c needs to include person.h to be able to access the functionality provided by the "person" code files

Compiling a Project with Multiple Files

How do we compile a multi-file project?

- We need to compile all *.c files that we will use
- The *.c files will **`#include`** the necessary *.h files
- Amongst the *.c files there should be exactly one **`main()`** function
- The compiled program will run from the start of the **`main()`** function

Let's look at a multi-file project

I'm Batman!

- A set of files that allow us to define a "person"
- Each person has a name and some super powers
- `person.h` shows how we can use a person
- `person.c` has the underlying details
- `main.c` shows how we can include and use this code

person.h

What's in the Header file?

- A Typedef saying we can use `Person` to mean a pointer to a `struct person`
- No mention of what `struct person` is! We don't have direct access
- Functions to let us create and free a person
- A function to let us give powers to a person
- A function to display a person (by printing to the terminal)

person.c

Our implementation file

- The actual and hidden implementation of `struct person`
- This means that the code in the C file can use `struct person` but the `main.c` can only use `Person`
- Implementations of all the functions listed in `person.h`

main.c

The main file

- Contains the main function. There is always exactly one main function in any project. It will be where the program starts running
- **`#include`**s the **`person.h`** file (always include headers, but not C files)
- Uses things like **`Person`** and the functions provided in the header

Using the multi-file project

Compiling

- We'll compile all the C files (but no H files) into a single program
- We rely on `#include`s to get the information we need from H files
- In this case: `dcc main.c person.c -o person_demo`

Using Multi-file projects in COMP1511

- We will be keeping these reasonably simple in COMP1511
- Assignment 2 will have a multi-file project, but you will not need to create a multi-file project to pass this course

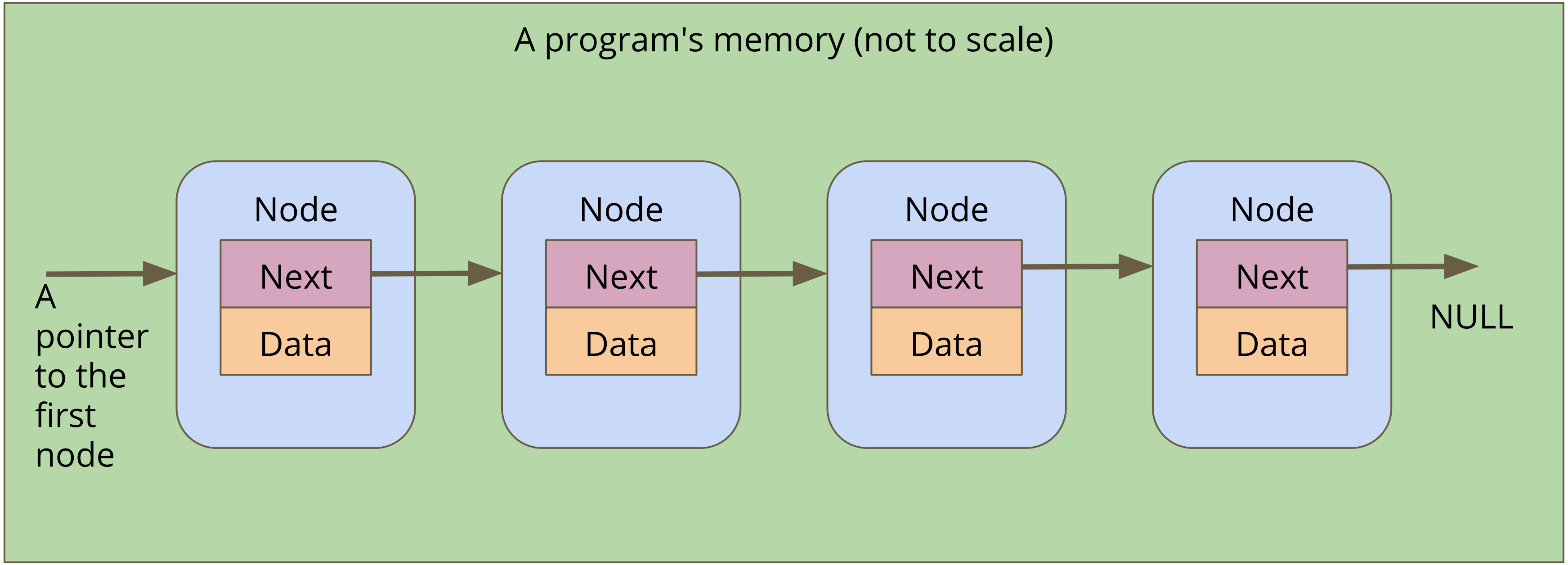
A new kind of struct

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

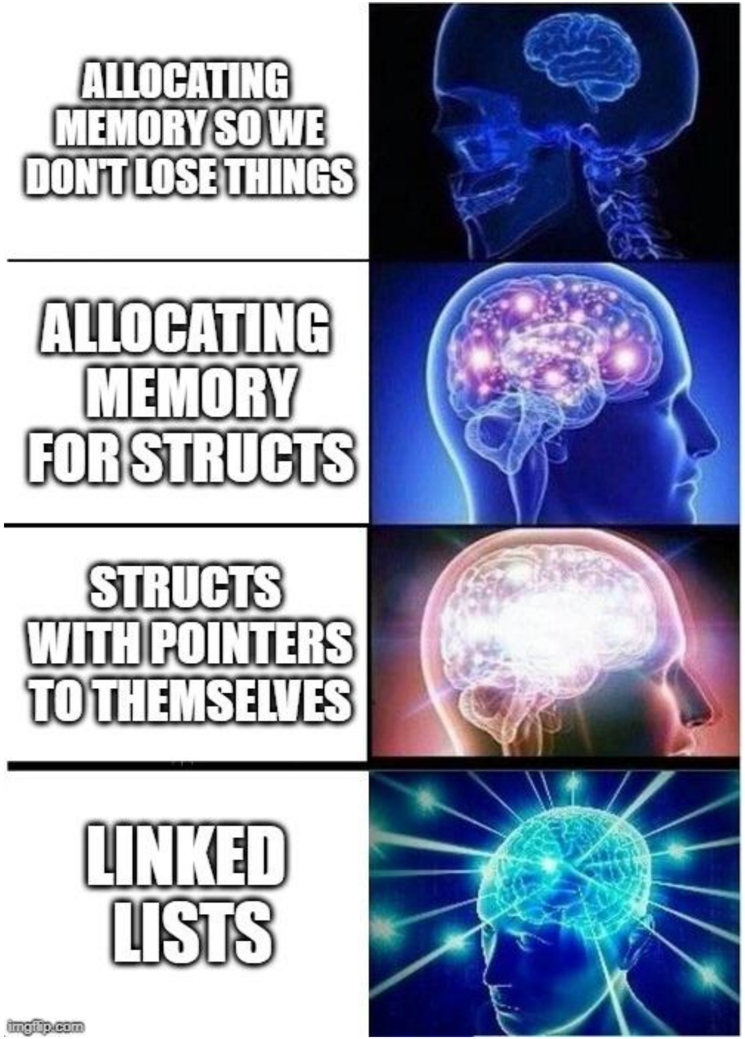
As opposed to Arrays . . .

- 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

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

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

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

A function to add a node

We've seen a similar function for creating a struct

```
// Create a node using the data and next pointer provided
// Return a pointer to this node
struct node *create_node(int data, struct node *next) {
    struct node *n = malloc(sizeof (struct node));
    n->data = data;
    n->next = next;
    return n;
}
```

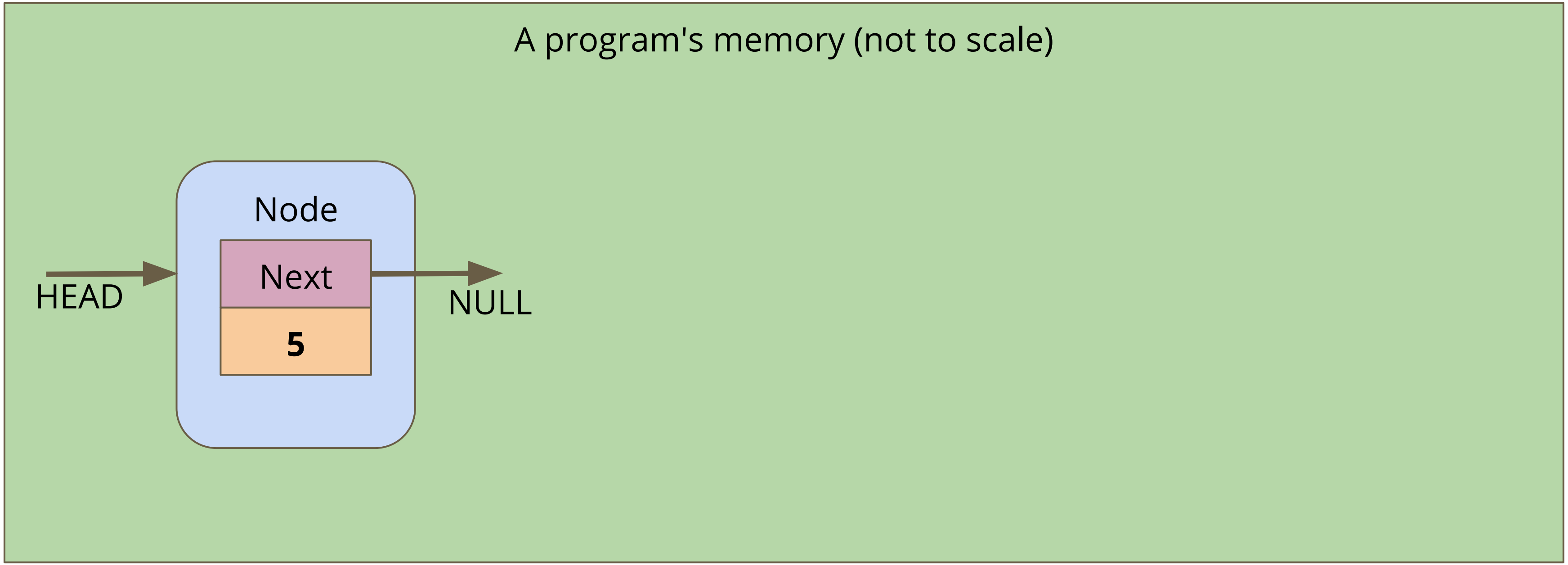
Building a list using only create_node()

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

    return 0;
}
```

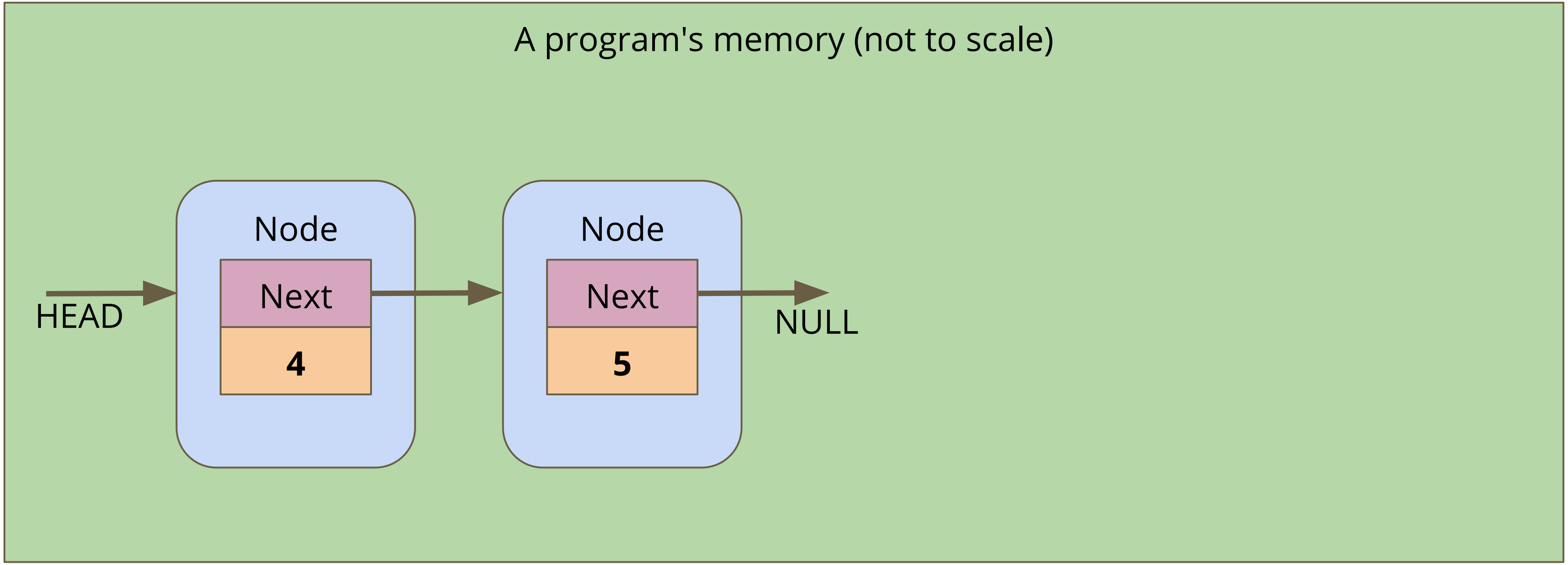
How it works 1

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



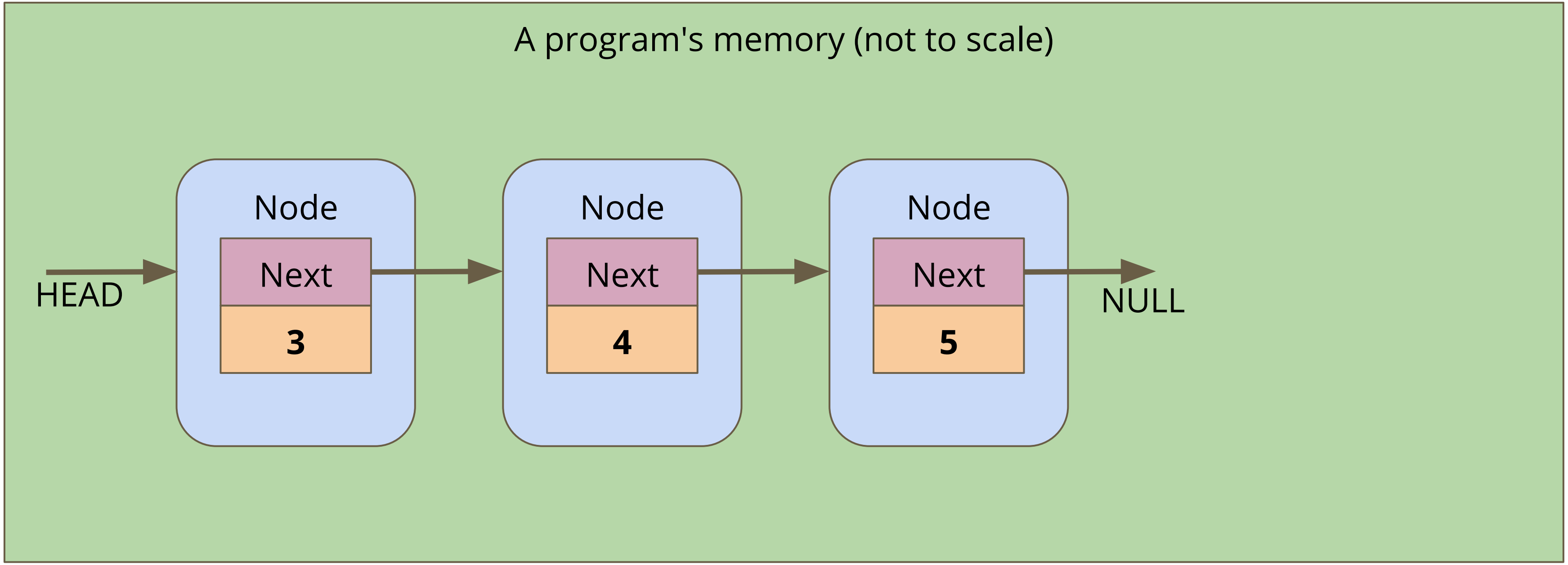
How it works 2

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 . . .



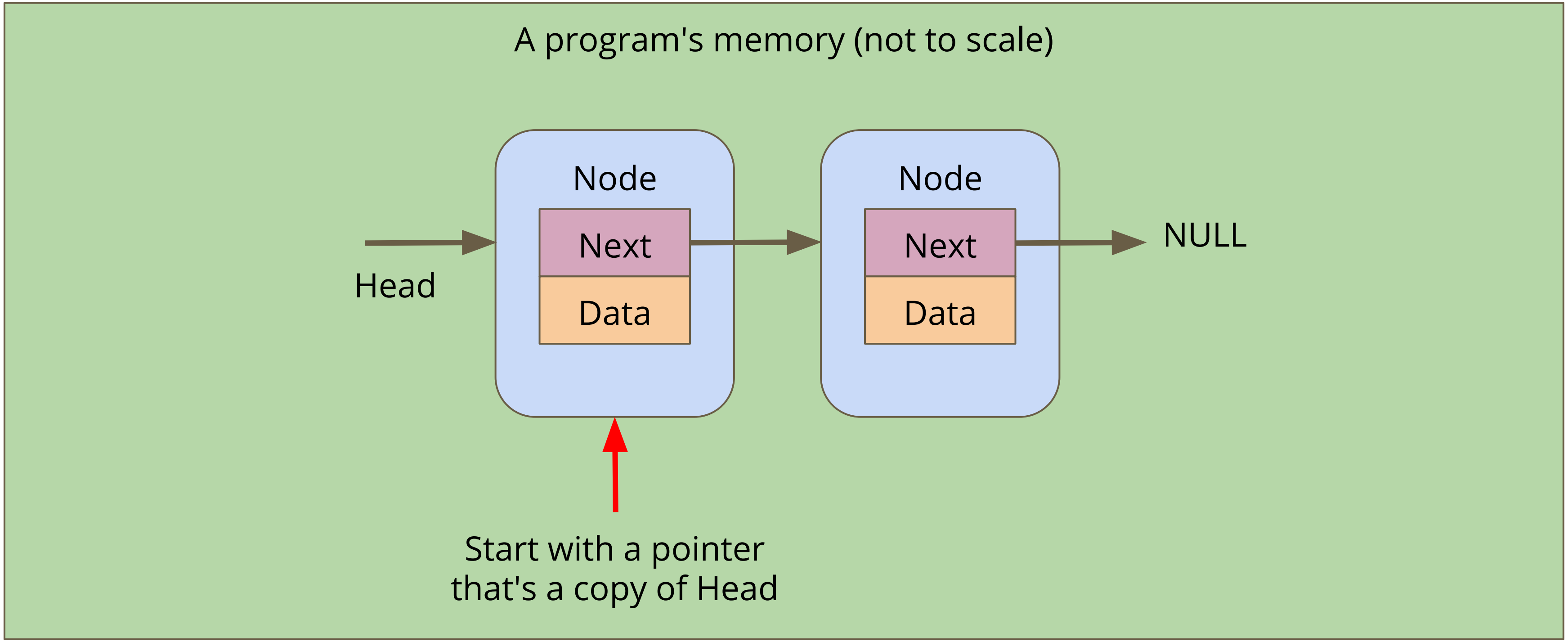
Looping through a Linked List

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

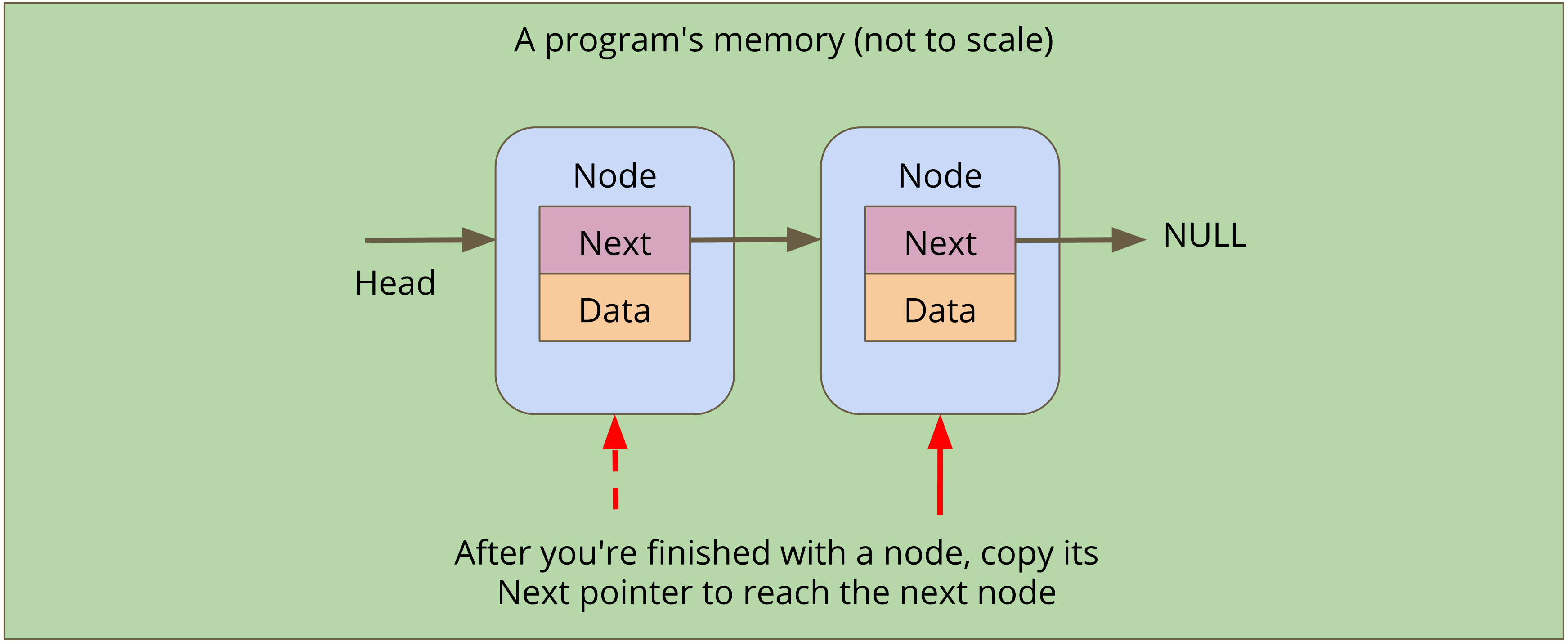
```
// Loop through a list of nodes, printing out their data
void printData(struct node *n) {
    while (n != NULL) {
        printf("%d\n", n->data);
        n = n->next;
    }
}
```

Looping through a Linked List



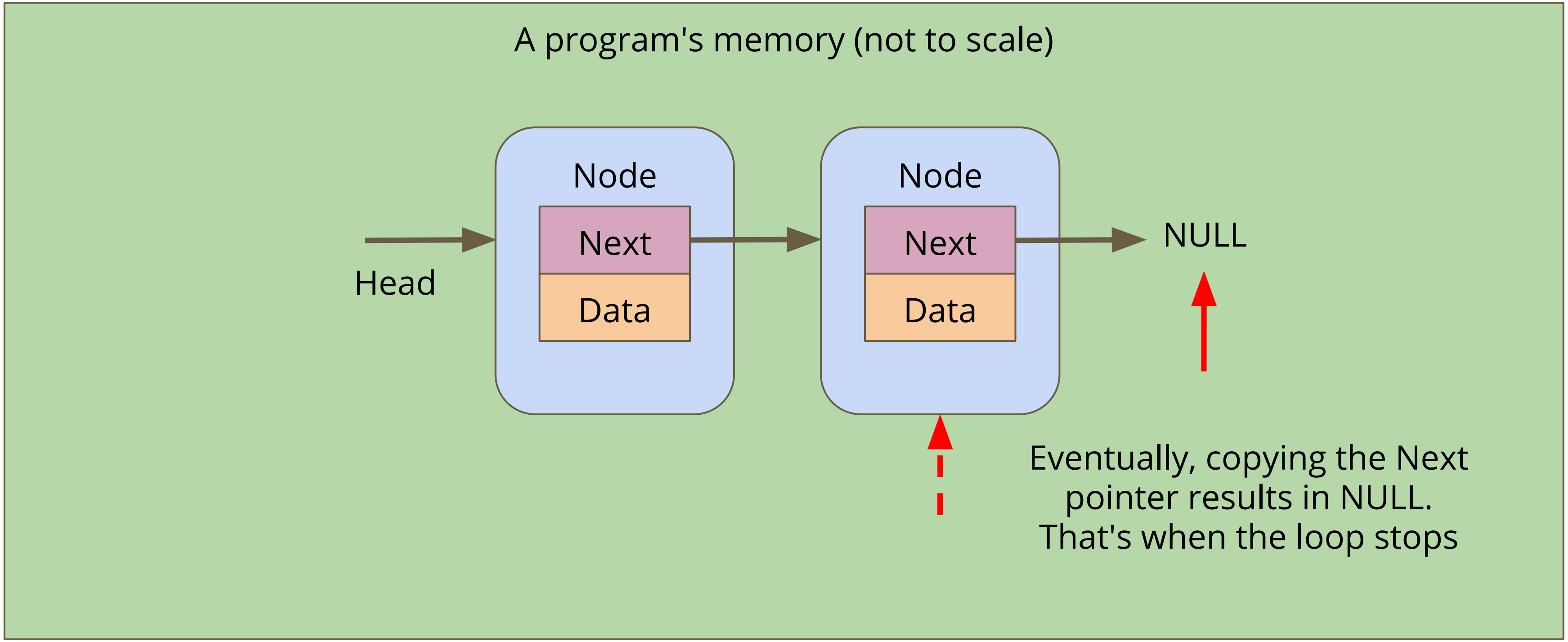
Start with a pointer
that's a copy of Head

Looping through a Linked List



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

Creating nodes

We'll want a function that creates a node

```
// Create a player using the name and next pointer provided
// Return a pointer to this player
struct player *create_player(char new_name[], struct player *new_next) {
    struct player *p = malloc(sizeof (struct player));
    strcpy(p->name, new_name);
    p->next = new_next;
    return p;
}
```

Creating the list itself

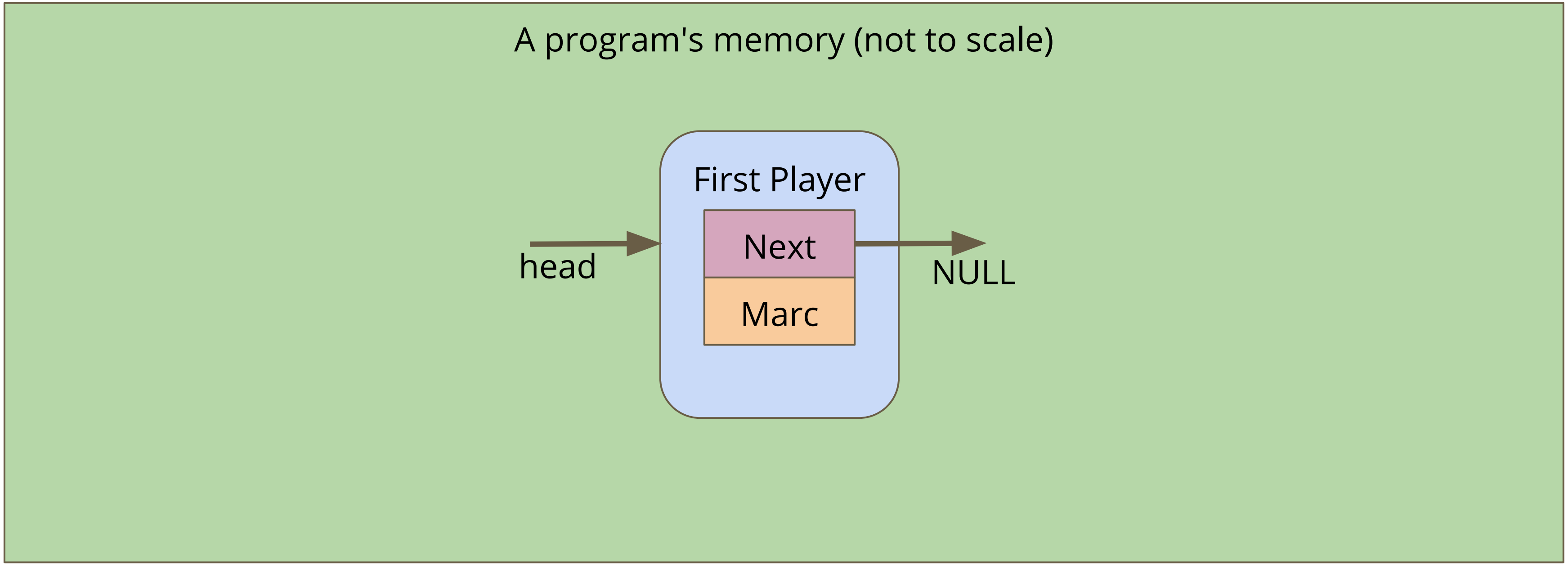
Note that we don't need to specify the length of the list!

```
int main(void) {
    // create the list of players
    struct player *head = create_player("Marc", NULL);
    head = create_player("Chicken", head);
    head = create_player("Aang", head);
    head = create_player("Goku", head);

    return 0;
}
```

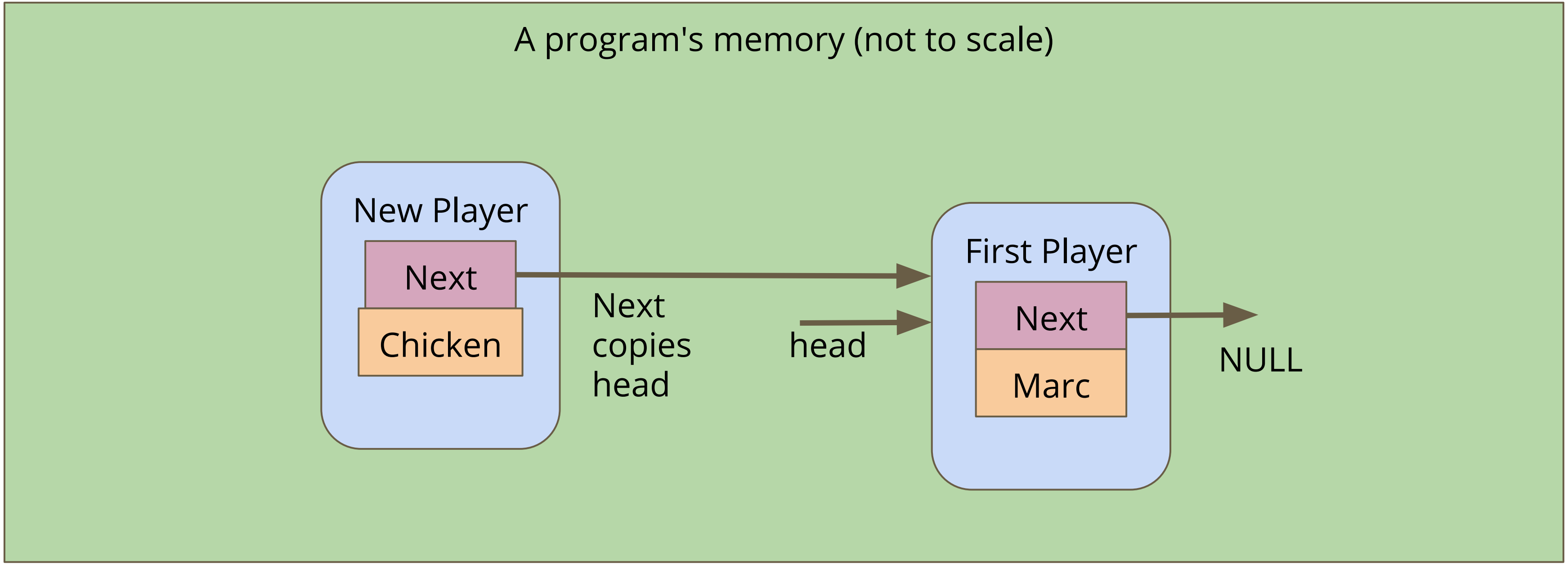
Using create_player

Head points at the First Player, its next is NULL



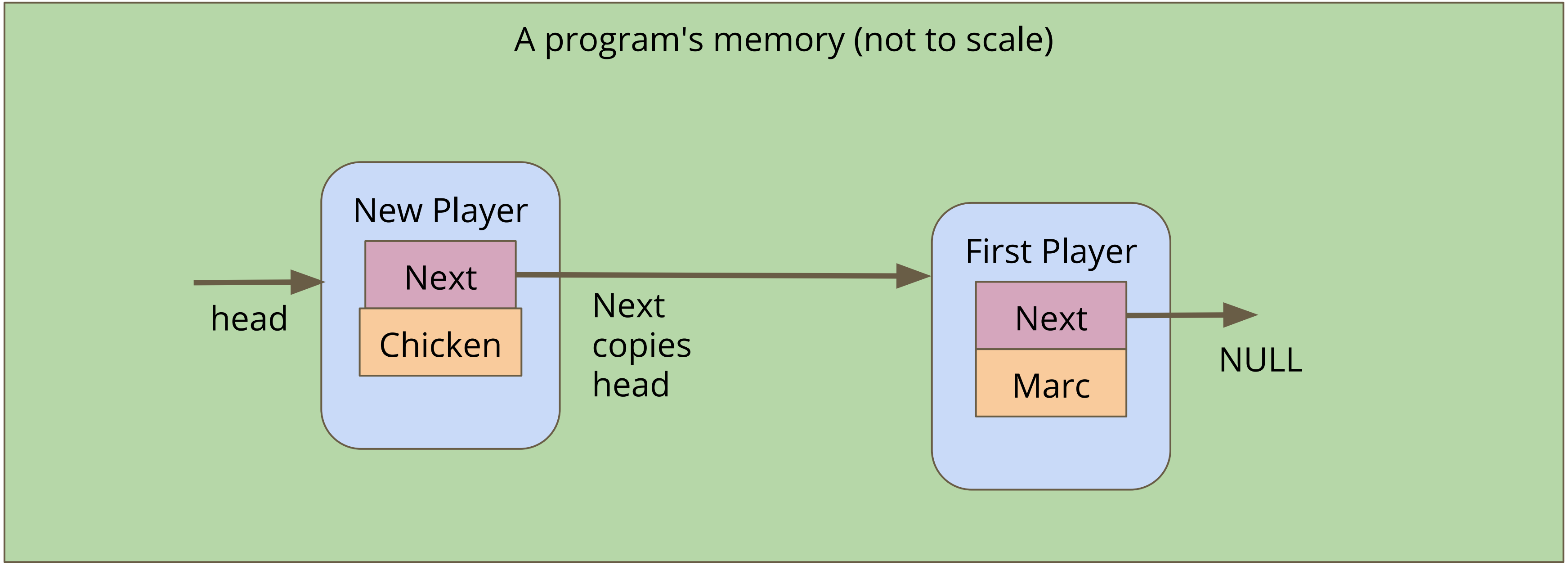
Using create_player

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



Using create_player

create_node returns a pointer to New Player, 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 player node
- Then move onto the next pointer from that player 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 print_players(struct player *current) {
    while (current != NULL) {
        printf("%s\n", current->name);
        current = current->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?

Multi-File Projects

- Spreading out our code functionality into more than one file

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