COMP1511 - Programming Fundamentals

Term 2, 2019 - Lecture 14

What did we learn yesterday?

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 learning today?

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

Recap - Memory

Memory Allocation

- Variables created inside any curly brackets will disappear when they end
- Sometimes we want to create something that persists outside of this
- `malloc()` allows us to allocate memory within our program
- `malloc()` returns a pointer to let us know where our allocated memory is
- Then we can use that pointer as if it's pointing at a normal variable
- When we're finished with our allocated memory we must `free()` it
- We can use `dcc --leak-check` to tell us if we're leaking memory

Recap - Multi-file Projects

Splitting a project into multiple files

- Header files (*.h) contain functions declarations and no running code
- They're used to show what the capabilities of the code are
- Implementation files (*.c) contain full implementations
- This is where the complicated code is hidden
- Other files (like main.c etc) will #include the header files in order to use the functionality
- All C files from the project will be compiled together

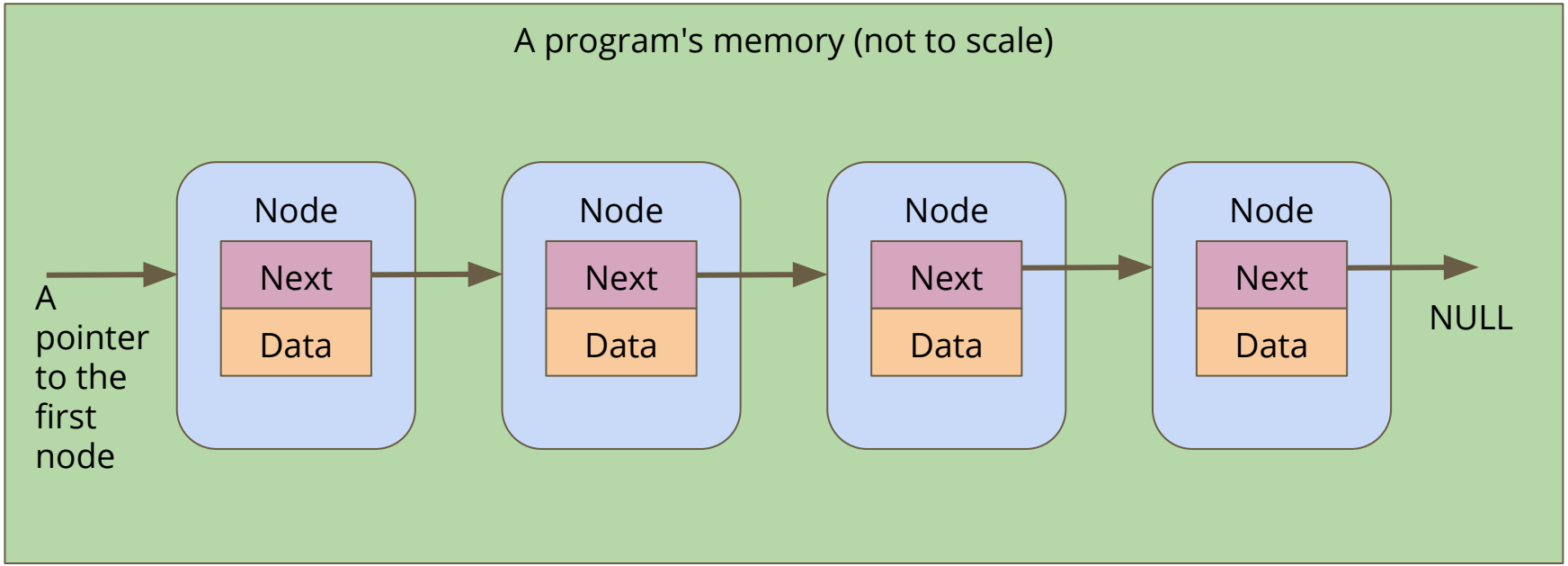
A new kind of struct

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

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

```
// 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;
}
```

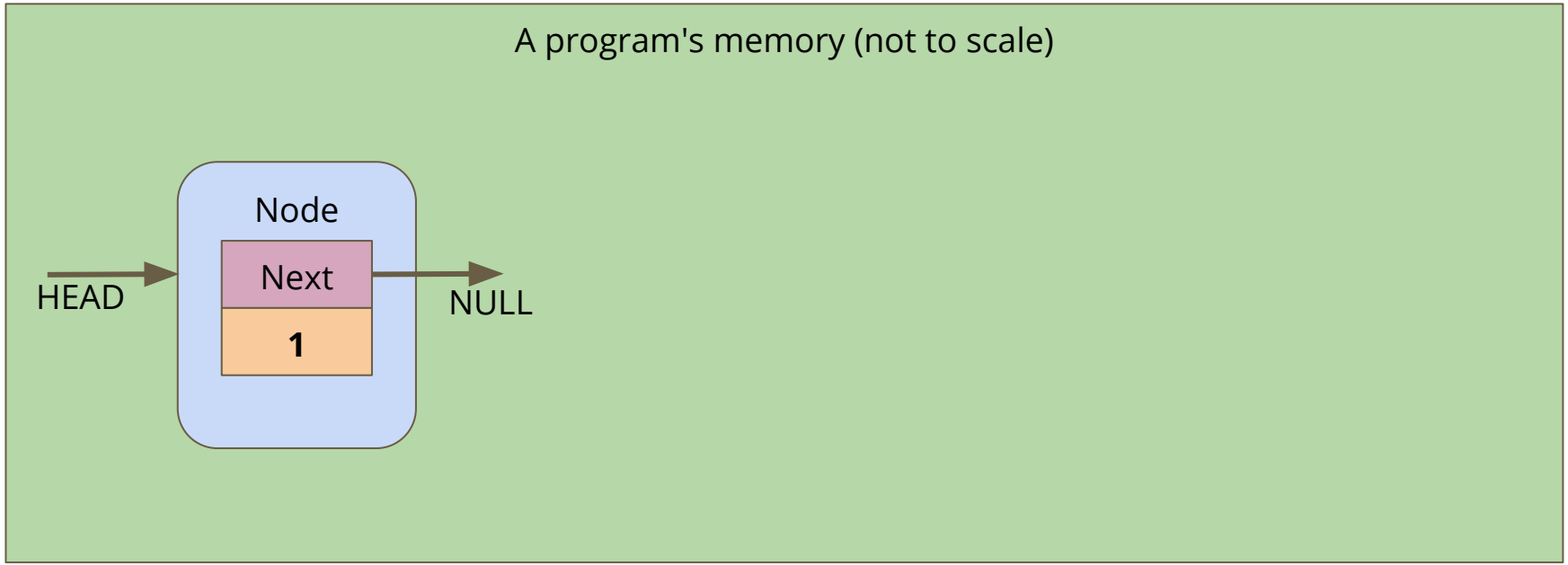
Building a list from createNode()

```
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;
}
```

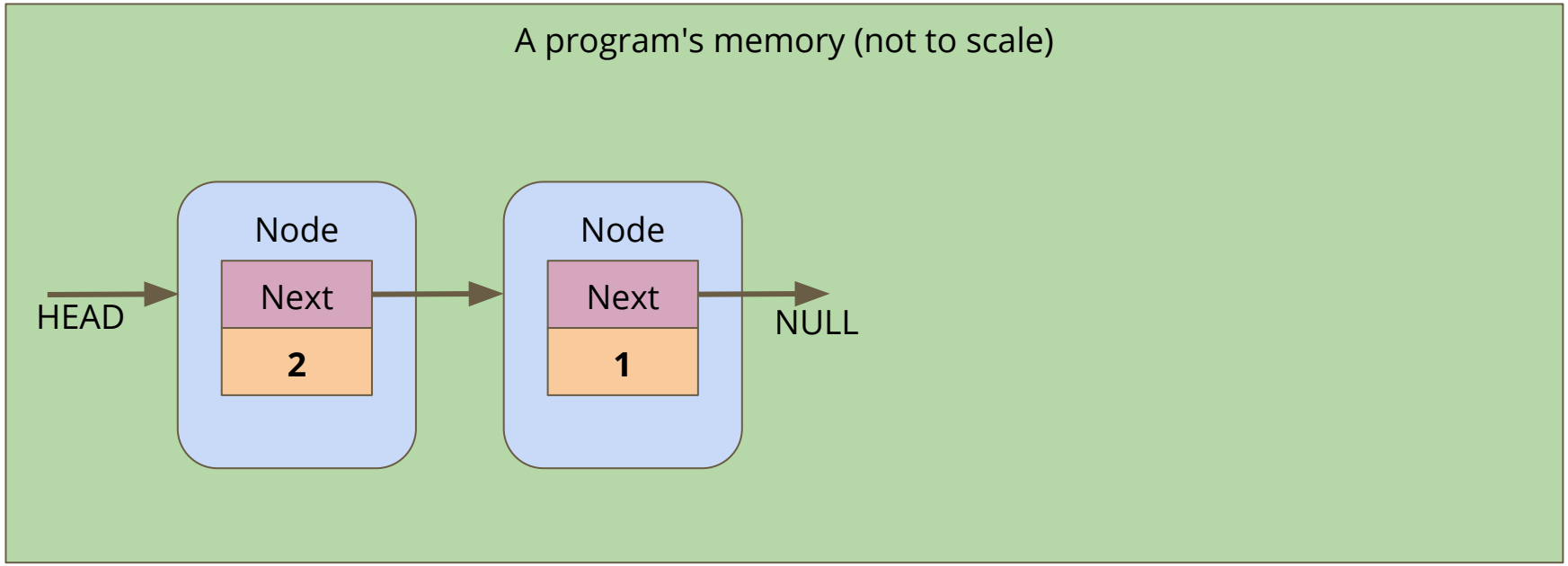
How it works 1

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



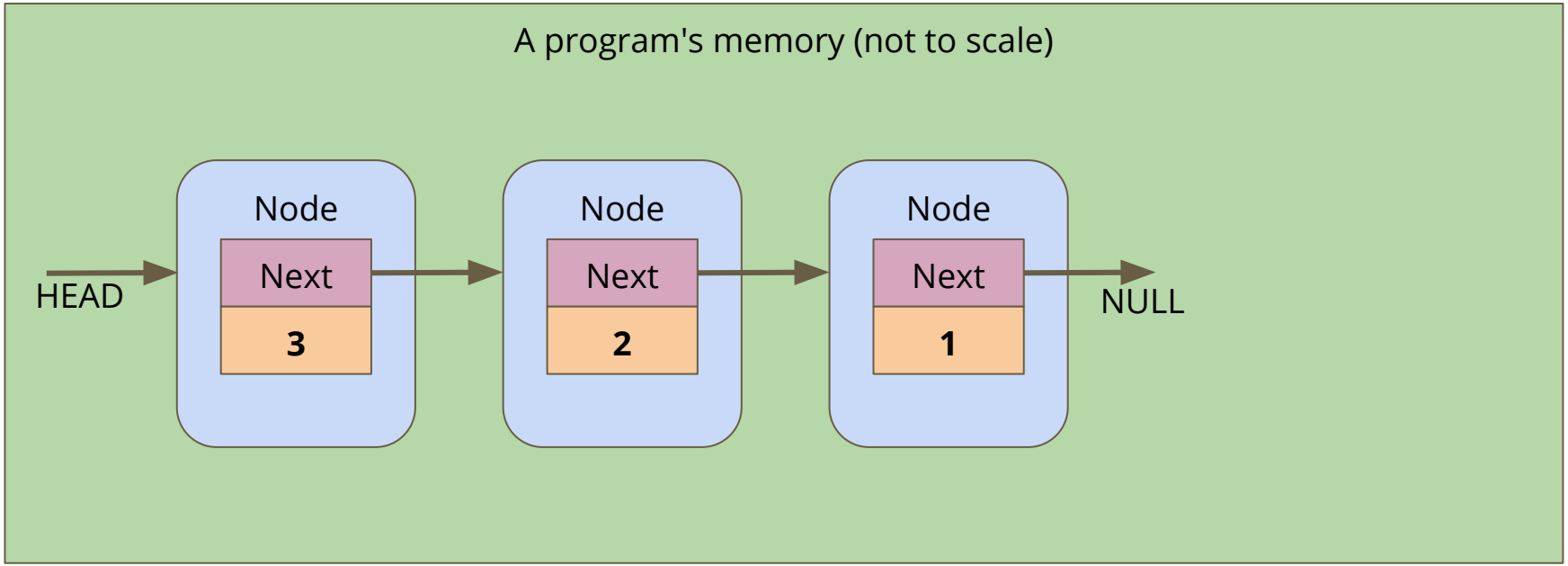
How it works 2

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 . . .



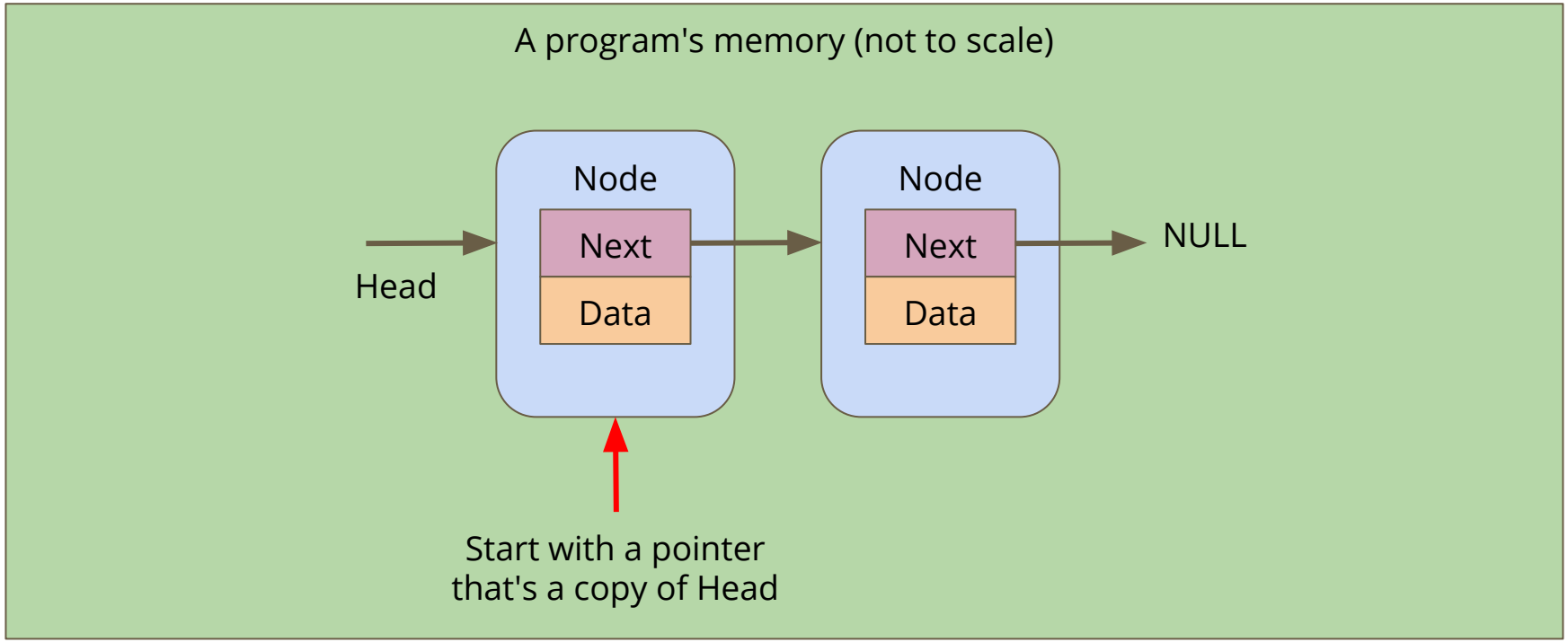
Looping through a Linked List

Linked lists don't have indexes . . .

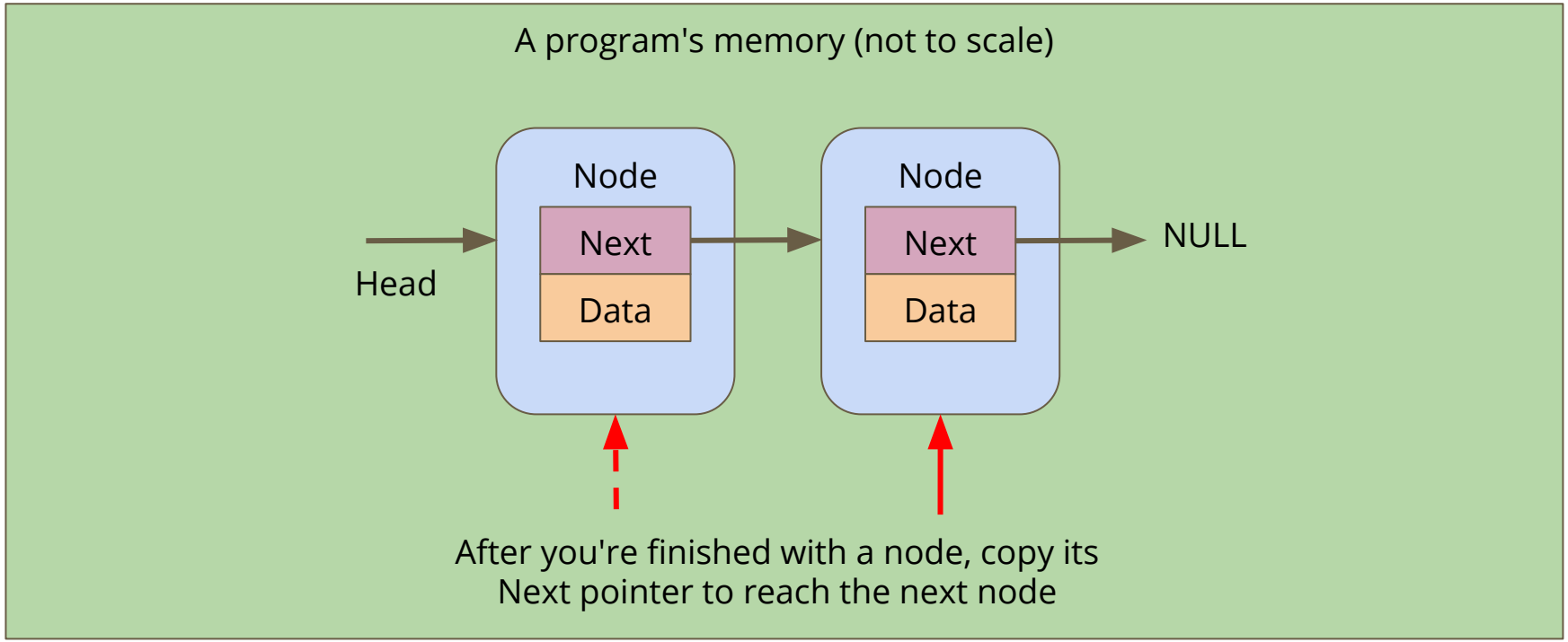
- We can't loop through them in the same way as arrays
- We have to follow the links from node to node
- If we reach a NULL node pointer, it means we're at the end of the list

```
// 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;
    }
}
```

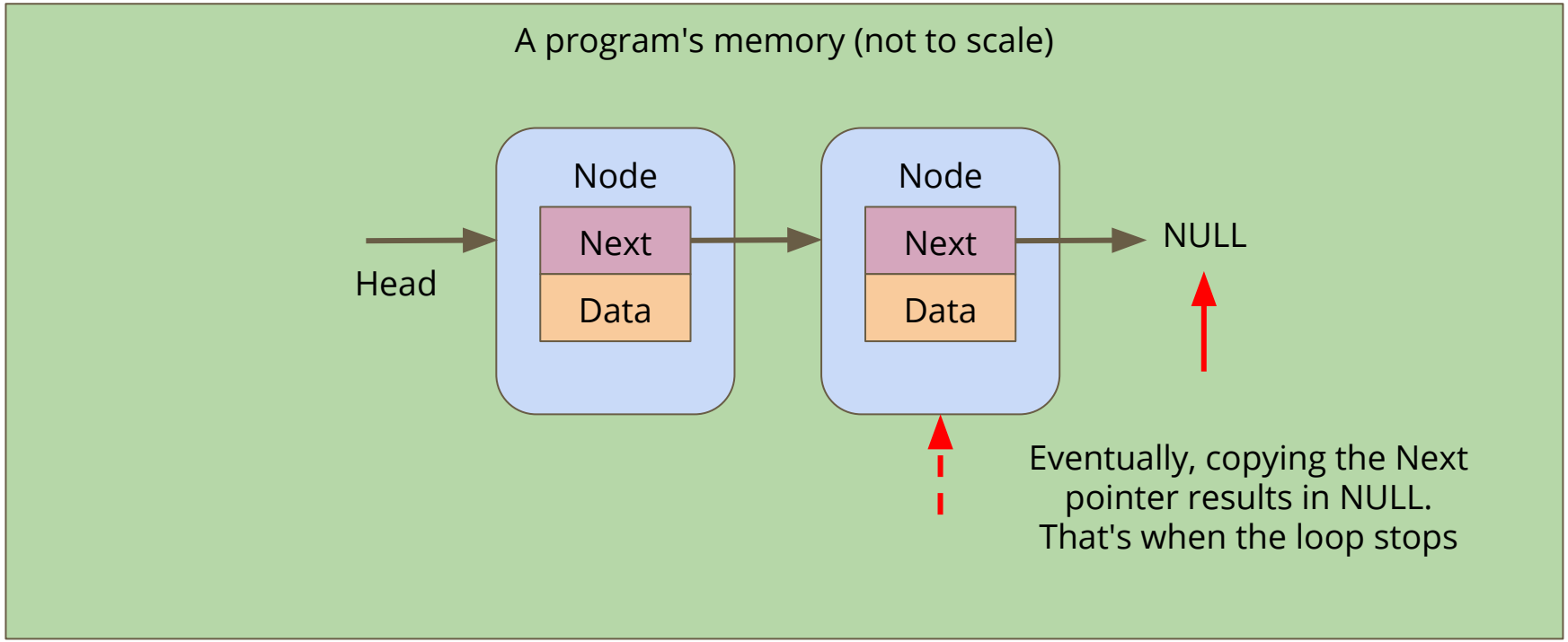
Looping through a Linked List



Looping through a Linked List



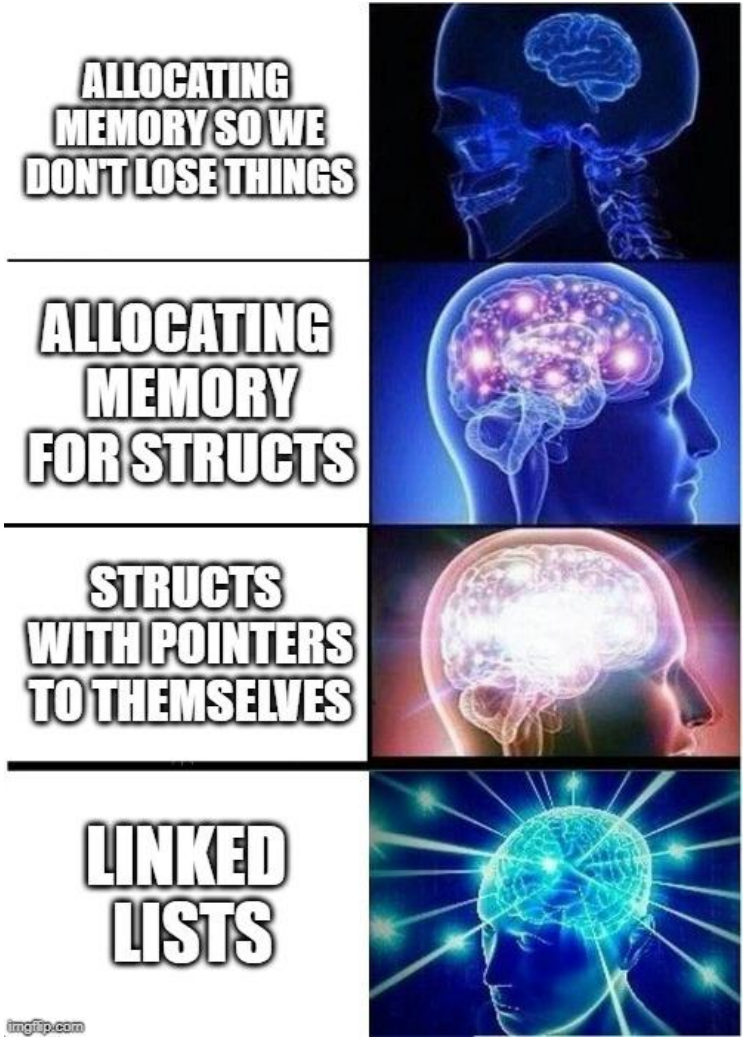
Looping through a Linked List



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



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

```
// 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

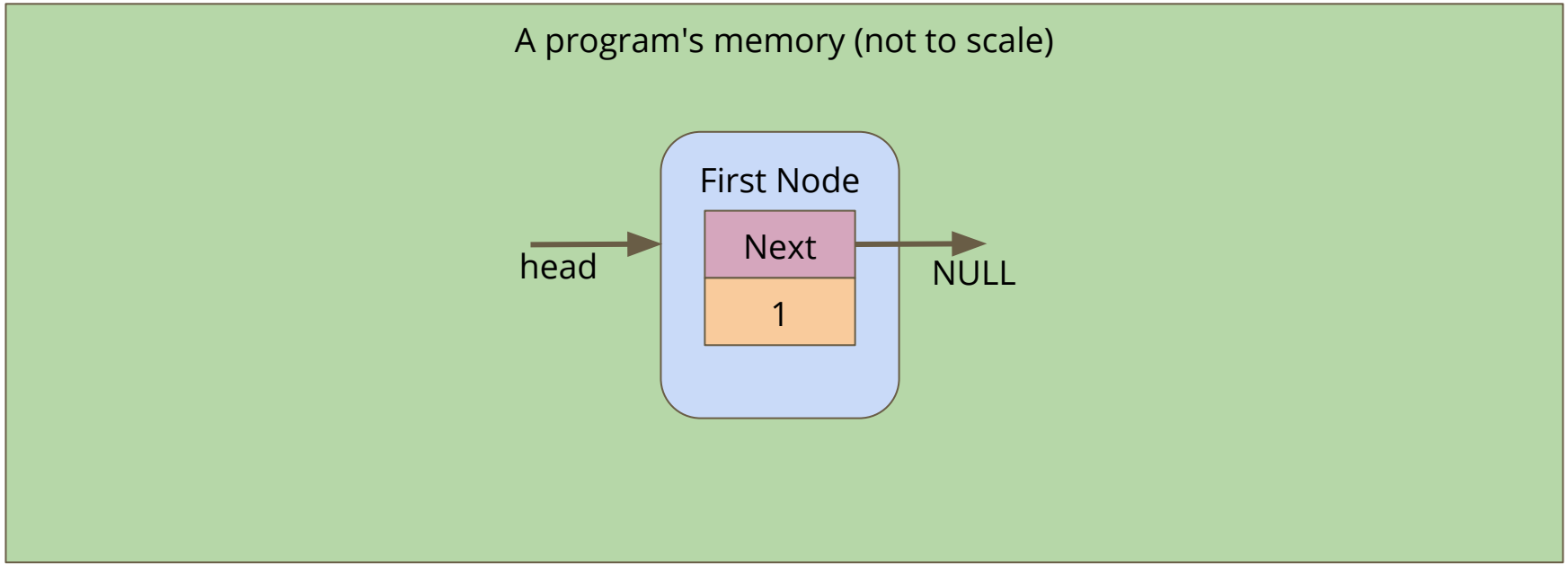
Note that we don't need to specify the length of the list!

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

    return 0;
}
```

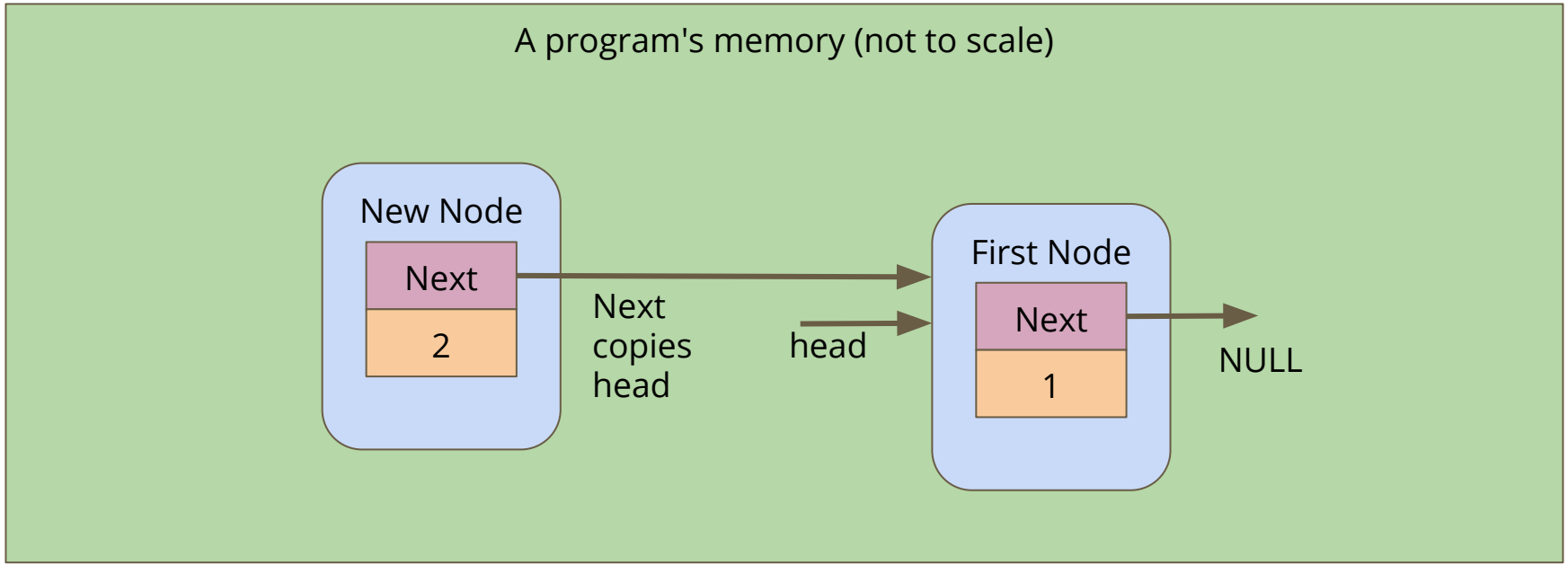
Using createNode

Head points at the First Node, its next is NULL



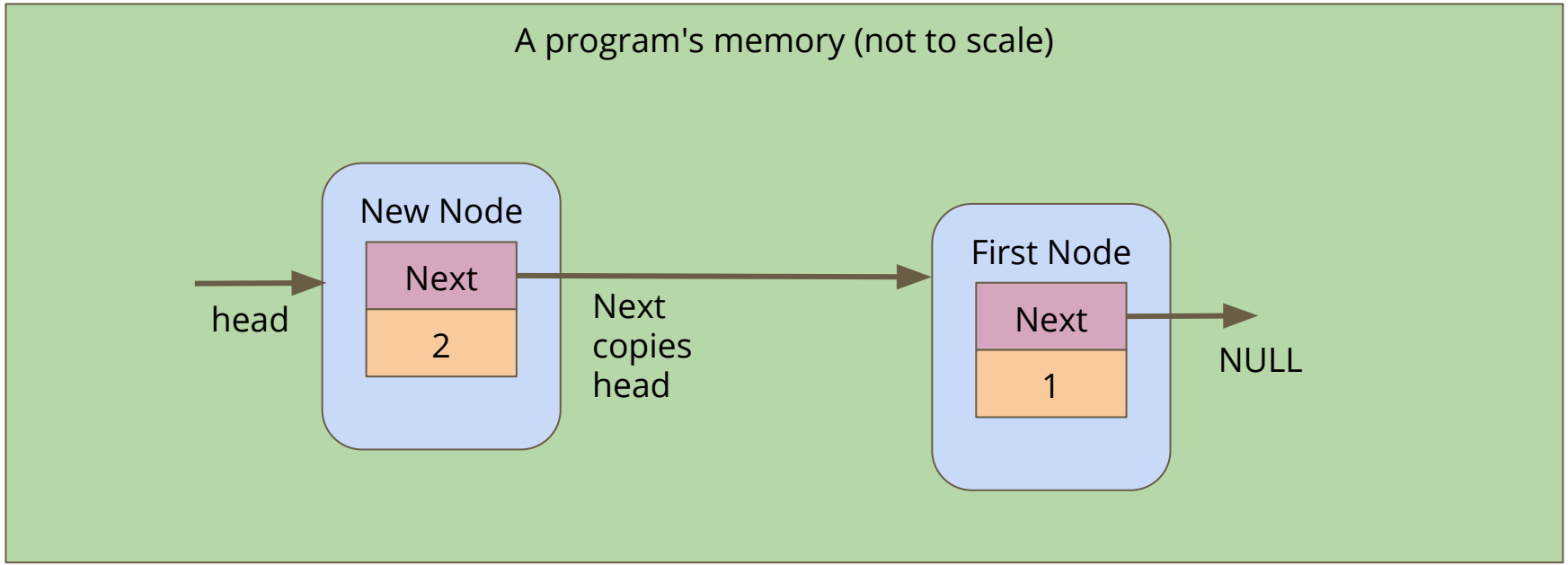
Using createNode

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

```
// 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;
    }
}
```

Inserting Nodes into a Linked List

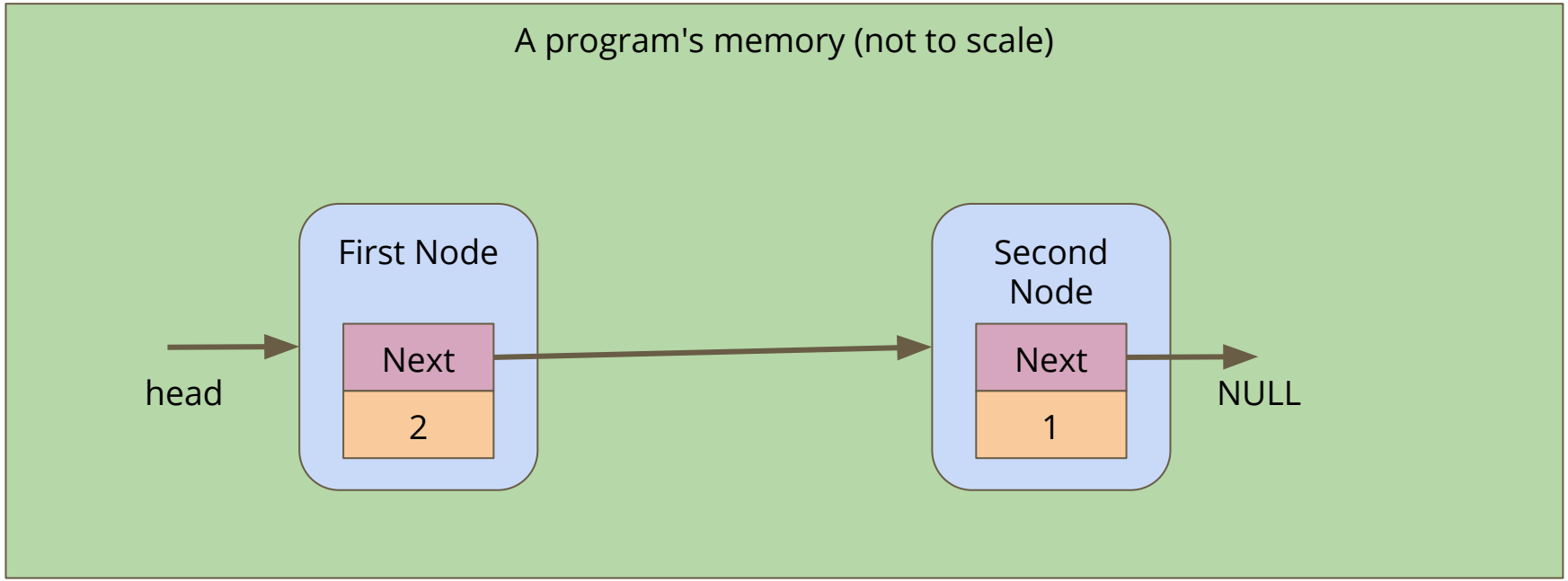
Linked Lists allow you to insert nodes in between other nodes

- We can do this by simply aiming next pointers to the right places
- We find two linked nodes that we want to put a node between
- We take the **next** of the first node and point it at our new node
- We take the **next** of the new node and point it at the second node

This is much less complicated with diagrams . . .

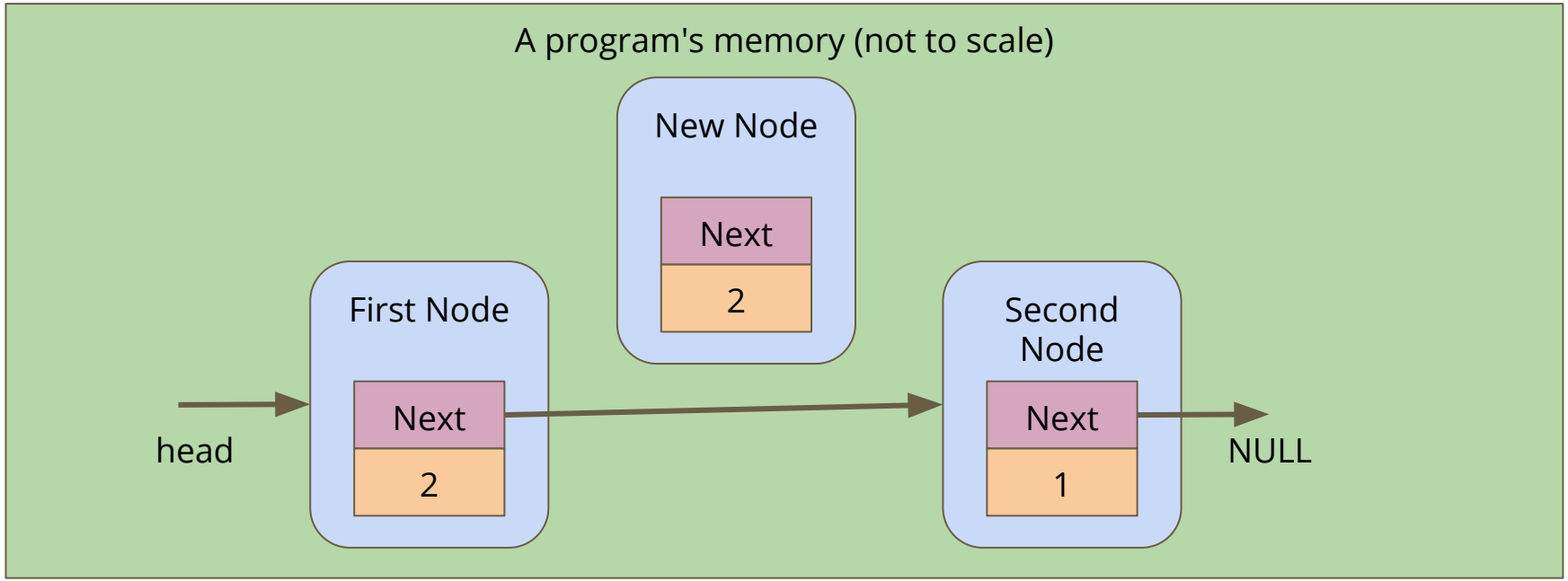
Our Linked List

Before we've tried to insert anything



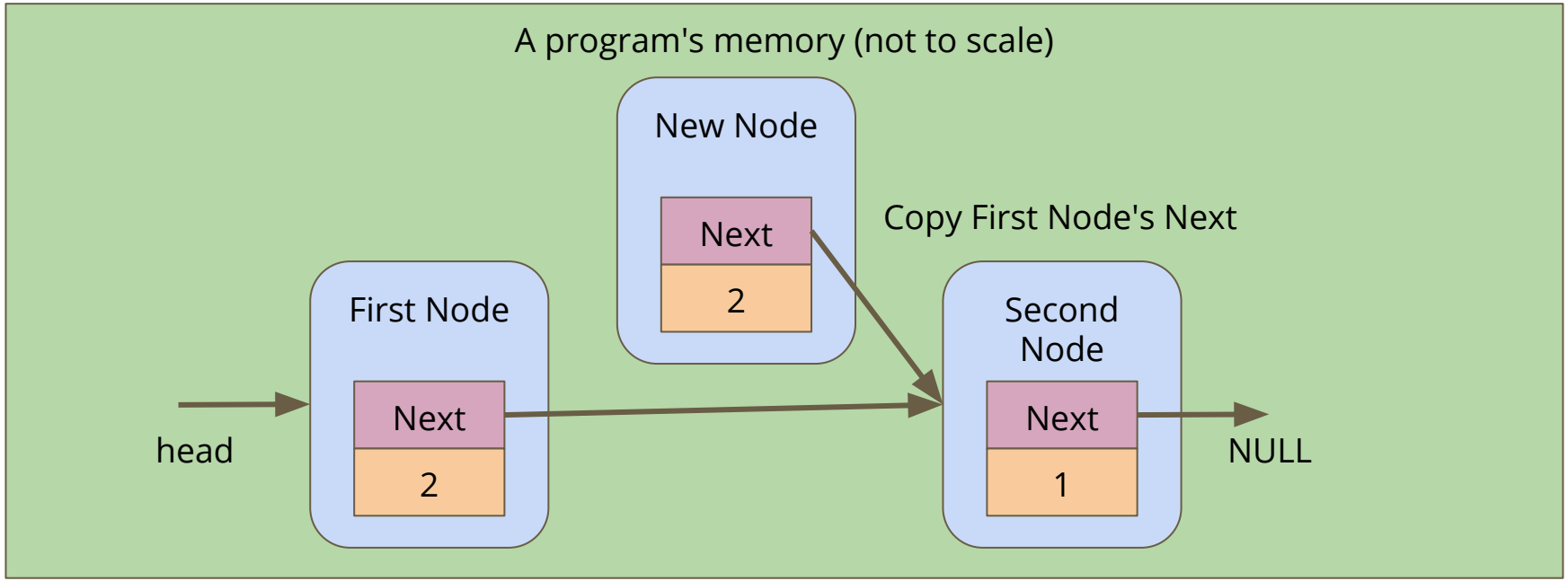
Create a node

A new node is made, it's not connected to anything yet



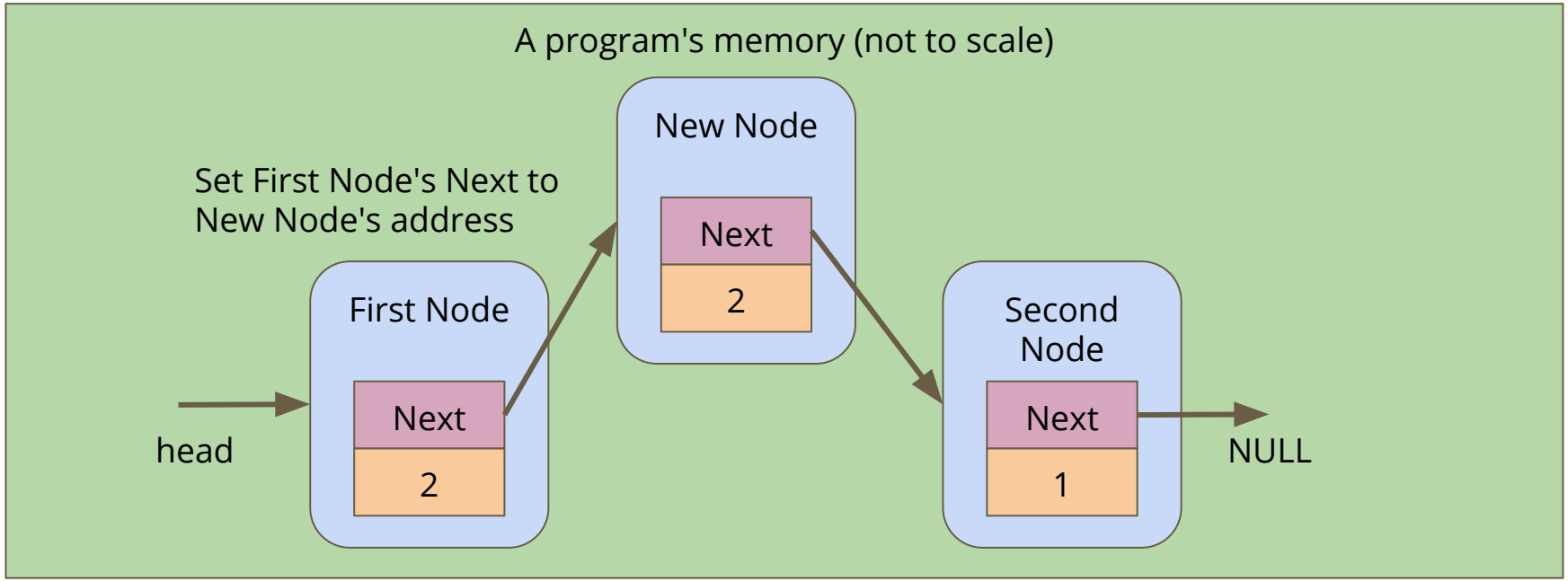
Connect the new node to the second node

Alter the **next** pointer on the New Node



Connect the first node to the new node

Alter the **next** pointer on the First Node



Code for insertion

```
// Create and insert a new node into a list after a given listNode
struct node *insert(struct node* listNode, char newName[]) {
    struct node *n = createNode(newName, NULL);
        if (listNode == NULL) {
        // List is empty, n becomes the only element in the list
        listNode = n;
        n->next = NULL;
    } else {
        n->next = listNode->next;
        listNode->next = n;
    }
    return listNode;
}
```

Inserting Nodes

We can use insertion to have greater control of where nodes are put in a list

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

    printPlayers(head);

    return 0;
}
```

To be continued

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

- 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?

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
- Inserting nodes after specific nodes