COMP1511 - Programming Fundamentals

Term 2, 2019 - Lecture 16

What did we learn yesterday?

Linked Lists

- Insertion and Removal
- Not quite finished with removal, we'll look at that more today!

What are covering today?

Battle Royale Game

- Linked List Insertion is complete
- We'll continue Linked List Removal
- Seeing the game being played
- Freeing memory (a whole list)

Assignment 2, Castle Defense

- Specification overview
- Information about assessment

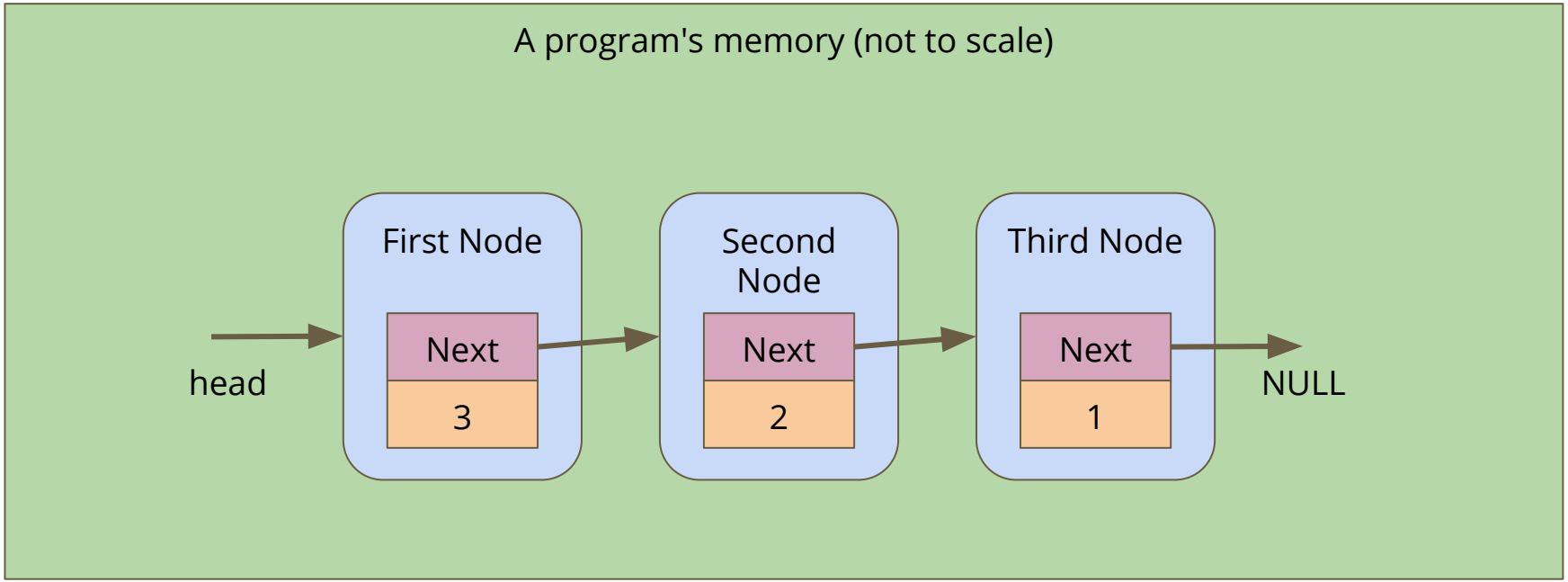
Removing a node

If we want to remove a specific node

- We need to look through the list and see if a node matches the one we want to remove
- To remove, we'll use **next** pointers to connect the list around the node
- Then, we'll free the node itself that we don't need anymore

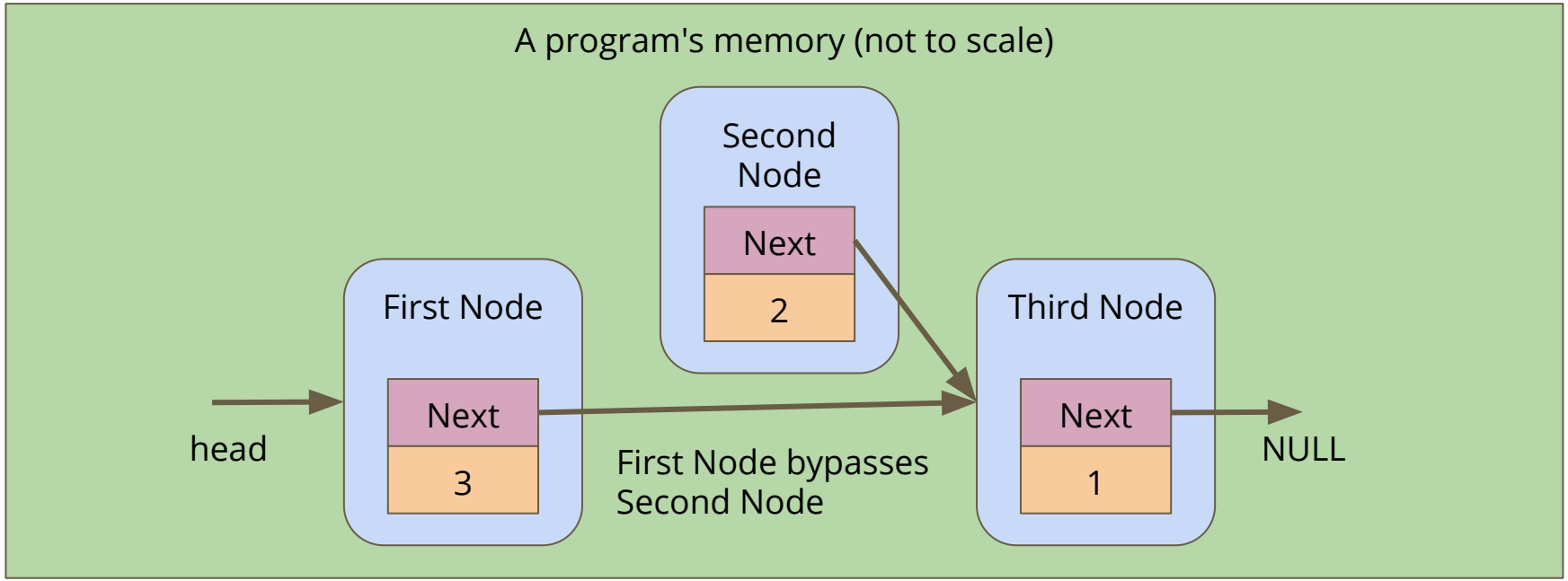
Removing a node

If we want to remove the Second Node



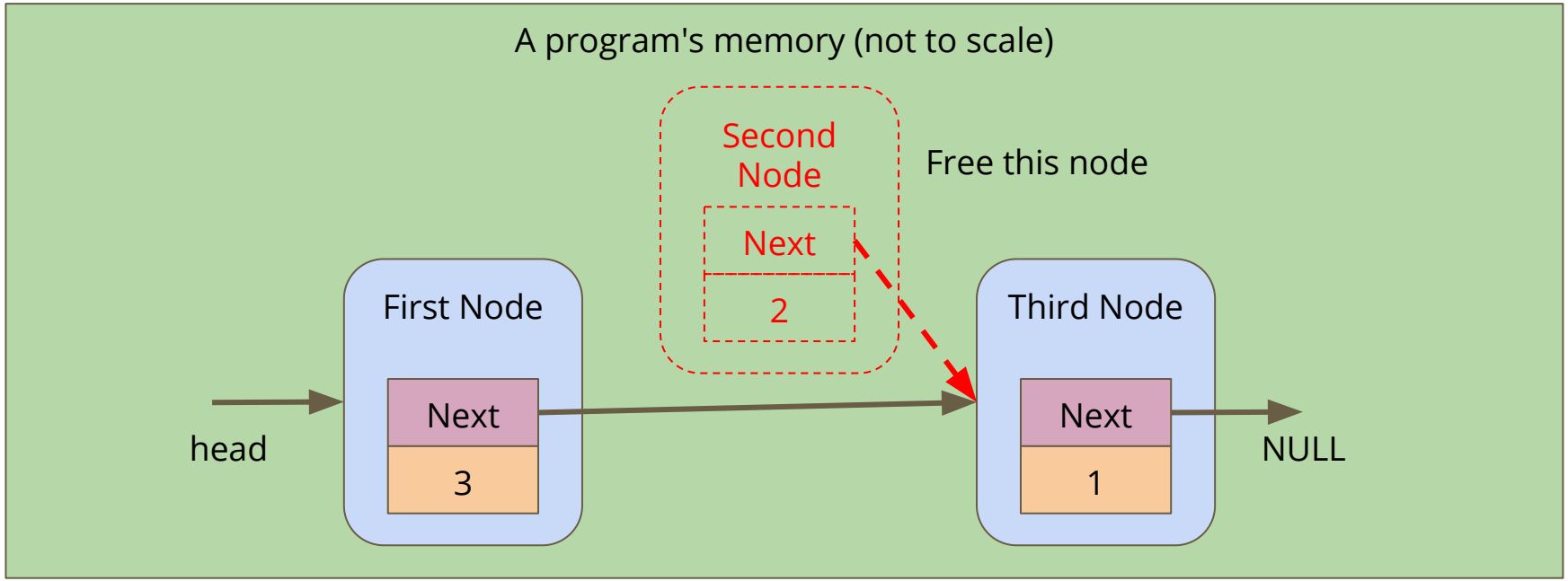
Skipping the node

Alter the First Node's **next** to bypass the node we're removing



Freeing the node

Free the memory from the now bypassed node



Finding the node

Loop until you find the right match

```
struct node *removeNode(char name[], struct node* head) {
    struct node *previous = NULL;
    struct node *n = head;
    // Keep looping until we find the matching name
    while (n != NULL && strcmp(name, n->name) != 0) {
        previous = n;
        n = n->next;
    }
    if (n != NULL) {
        // if n isn't NULL, we found the right node
```

Removing the node

Having found the node, remove it from the list

```
if (n != NULL) {
        // if n isn't NULL, we found the right node
        if (previous == NULL) {
            // it's the first node
            head = n->next;
        } else {
            previous->next = n->next;
        }
        free(n);
    }
    return head;
}
```

The Battle Royale Game

In a Battle Royale, people are removed from the game one at a time until only one person is left. They are the winner

- We can create a list of players
- We can make sure it's in a nice alphabetical order
- We can remove a single player from the list
- Now we need to remove players one at a time
- When there's only one left, they are the winner!

Game code

Once our list is created, we can loop through the game

- We print out the player list (we might want to modify that function!)
- Our user will tell us who was knocked out

```
// A game loop that runs until only one player is left
while (printPlayers(head) > 1) {
    printf("Who just got knocked out?\n");
    char koName[MAX_NAME_LENGTH];
    fgets(koName, MAX_NAME_LENGTH, stdin);
    koName[strlen(koName) - 1] = '\0';
    head = removeNode(koName, head);
    printf("----------\n");
}
printf("The winner is: %s\n", head->name);
```

Cleaning Up

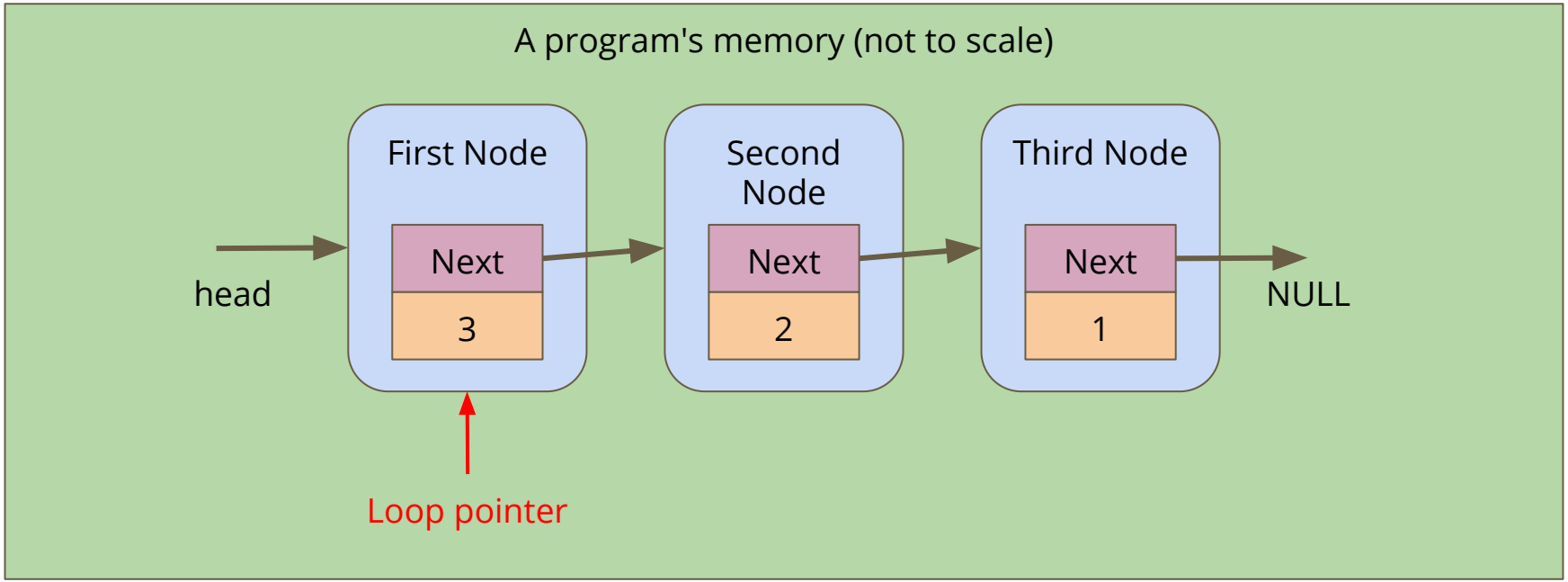
Remember, All memory allocated (malloc) needs to be freed

- We can run dcc `--leak-check` to see whether there's leaking memory
- What do we find?
- There are pieces of memory we've allocated that we're not freeing!

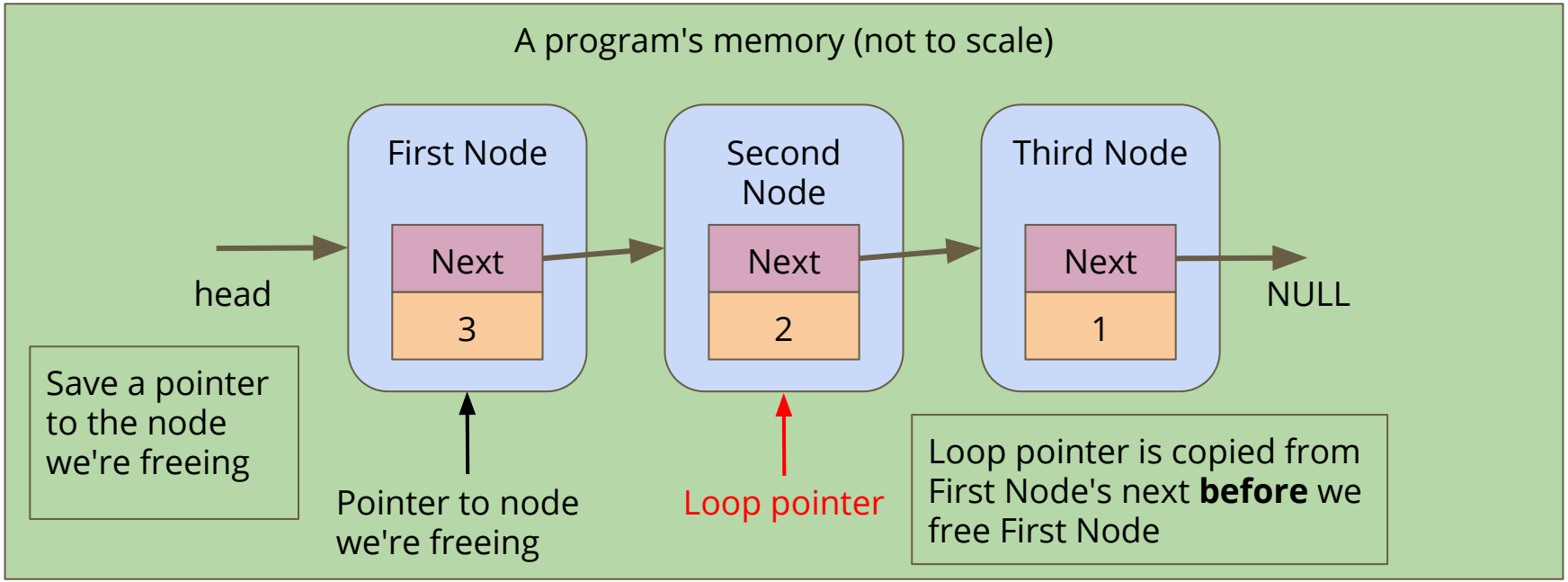
Let's write a function that frees a whole linked list

- Loop through the list, freeing the nodes
- Just be careful not to free one that we still need the pointer from!

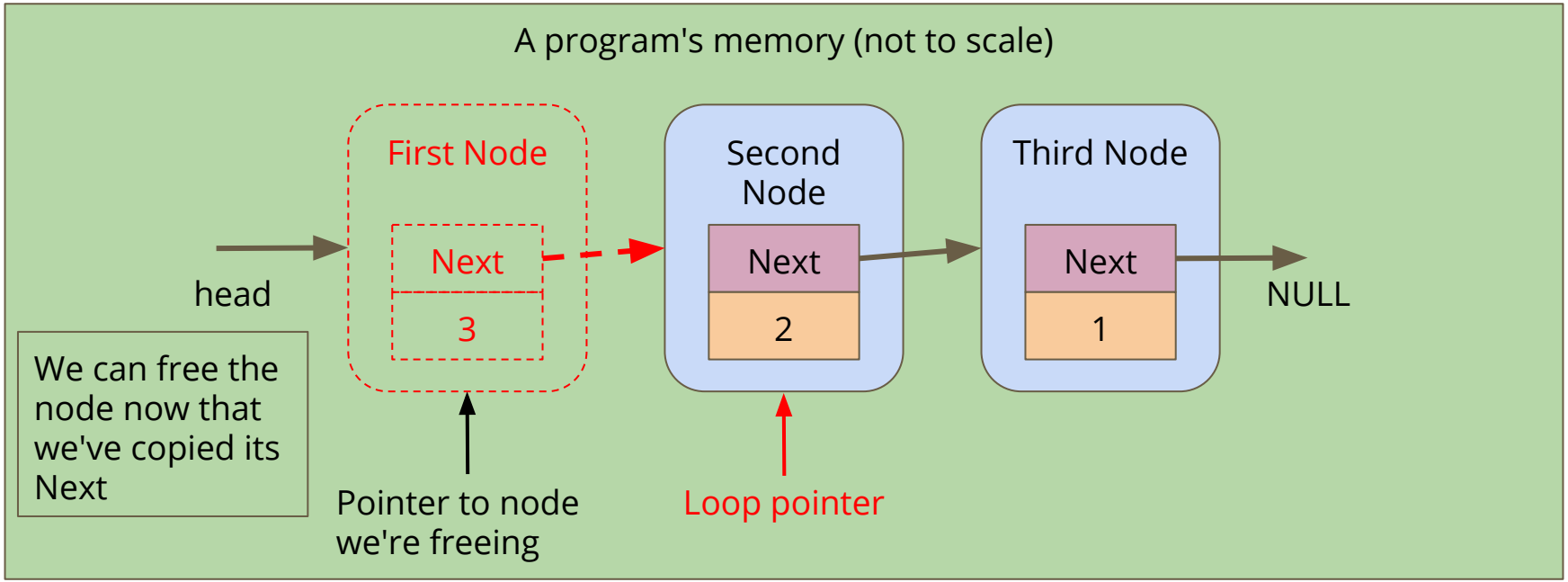
Looping to free nodes



Looping to free nodes



Looping to free nodes



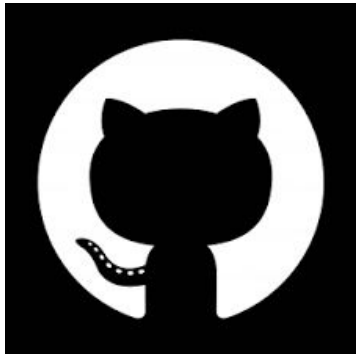
Code to free a linked list

```
// Loop through a list and free all the allocated memory
void freeList(struct node *n) {
    while(n != NULL) {
        // keep track of the current node
        struct node *remNode = head;

        // move the looping pointer to the next node
        n = n->next;

        // free the current node
        free(remNode);
    }
}
```

Break Time

Keeping track of your own code projects

- Using **git** is a really handy way to keep backups of your work
- GitHub and BitBucket are two providers that will give you free online repositories to store your code
- Graphical Interfaces are available for git (GitHub Desktop and Sourcetree respectively)
- It takes some time to get familiar with how these work . . . but you can start practicing now!



Assignment 2 - Castle Defense

The Tower Defense Genre

- A famous genre of computer games that rose to prominence around 2007-8
- Notable examples are "Desktop Tower Defense" and "Plants vs Zombies"

Tower Defense Games

Notable Features

- A land or path that enemies automatically walk on
- Players build defenses (usually towers) that automatically attack the enemies
- The aim is to destroy all the enemies before they complete their path
- This usually involves strategic placement of towers and upgrades
- The enemies scale in power as the game goes on

COMP1511's Castle Defense

We will build the "engine" behind a Tower Defense game

- A simple version of the "land" with locations and towers
- Simple enemies that move along the locations
- A very simple "step by step" time system instead of real-time movement
- The ability to affect enemies with towers
- All the details are in the Assignment Specification on the class website

How does Castle Defense work?

We have a reference solution that you can use

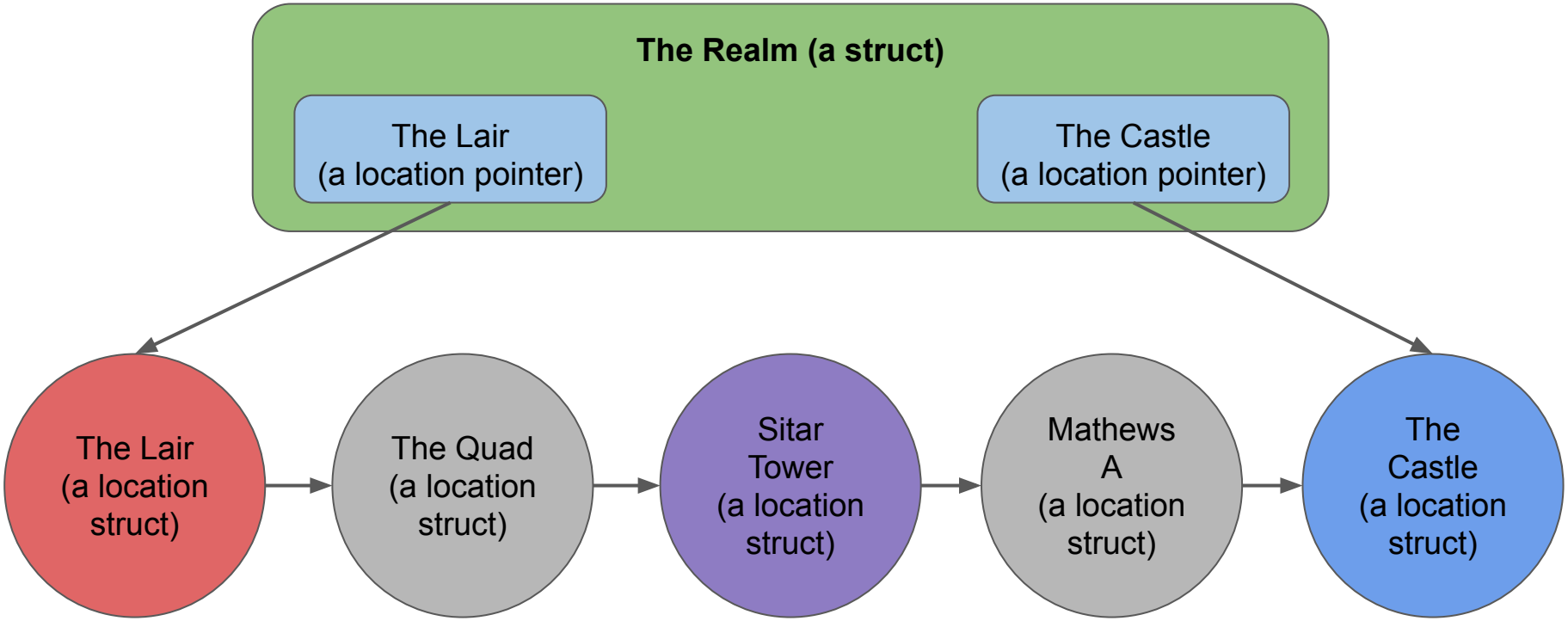
- `1511 castle_defense` runs the reference solution
- We start off by creating some lands
- Use `?` to list the commands
- We can print out the current state of the realm
- We can add towers
- We can add enemies
- We can move the enemies to their next location
- We can calculate damage done

Structures in Castle Defense

Castle Defense starts partially implemented

- The **realm** is a struct that contains and manages a linked list of **locations**
- The **locations** are already partially implemented as linked list nodes
- The **enemies** are also structs that are linked list nodes
- Each **location** has a linked list of enemies
- There are handy diagrams that show how this is organised . . .

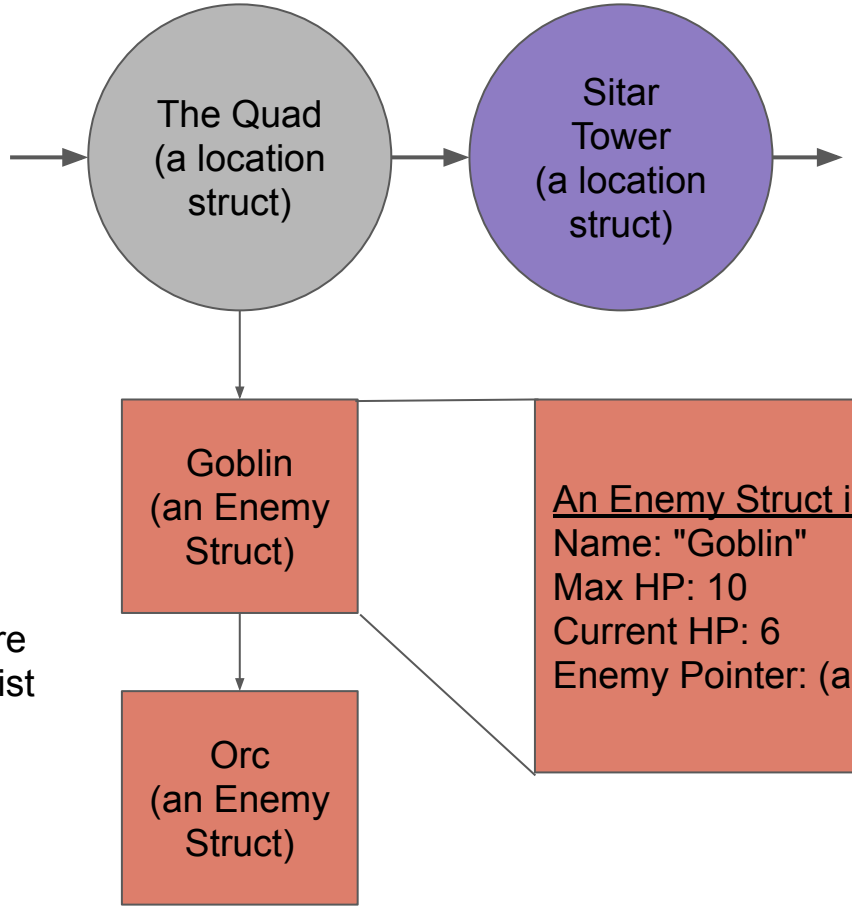
The Realm



Between The Lair and The Castle is a linked list of locations

Enemies

Part of the linked list of locations



Enemies at a location are organised into a linked list

How to get started

Setting up the Assignment

- We've provided a setup script that you can use
- First, create a directory for the Assignment (on VLAB or a CSE computer)
- Then, in that directory, run: **`1511 setup-castle-defense`**
- You will receive: **`main.c`**, **`realm.c`**, **`realm.h`** and **`test_realm.c`**
- Note that **`main.c`** and **`realm.h`** will be links and not actual files
- This is because these files cannot be edited for the assignment!
- It also means if we need to update them, you'll automatically receive the newest versions

What's in the files?

Each file has its own purpose

- **`main.c`** is the interactive program that handles input and output
- It will be calling functions in **`realm.h`**
- **`realm.h`** has the function declarations for all the functions that you will be implementing
- **`realm.c`** has the functional code for the assignment
- A lot of the **`realm.c`** functions are empty. That's where you'll be working
- **`test_realm.c`** contains a different main function to use for automated testing
- You will also be working in **`test_realm.c`** to write tests

Editing and Compiling

You will be implementing The Realm

- You'll be working mainly in **`realm.c`**, part of a multi-file project
- You'll also be creating tests in **`test_realm.c`**
- There are two ways to compile:
- For the interactive program: **`dcc main.c realm.c -o castle_defense`**
- For the automated testing: **`dcc test_realm.c realm.c -o testing`**

Working in Stages

The assignment is separated into stages based on difficulty

- Start from the beginning
- Later stages will need the earlier stages working
- A great deal of Stage 1 and 2 can be completed using techniques shown in lectures, tutorials and labs
- Feel free to use any code and algorithm design we have created in class and modify it to your needs

Working with your Linked List(s)

The location Linked List starts partially implemented

- The location struct is already set up as a linked list node struct
- The realm struct will have pointers to the start and end of the list
- You will be expected to make some functions that use and modify the linked list
- As you progress, you will find you need to make the struct more complicated
- Add fields and complexity only by necessity!

Testing

test_realm.c has some tests in it already

- You can run this to test some of the early stages functionality
- **`test_realm.c`** is not complete!
- Functions beyond Stage 1 will need your own testing functions
- **`test_realm.c`** shows you a nice way of setting up automated testing of individual functions
- This is often called **"Unit Testing"**

Assert

A valuable tool in testing

```
#include <assert.h>
// Asserts will test a code expression
int main (void) {
    int number = 2;
    int result = number * 3;

    // if this assert is false, the program will end here
    assert(result == 6);
}
```

We can use asserts to force our code to exit if one of our assumptions turns out to be false. If our program is running correctly, our asserts have no effect

Marks breakdown and Submission

More emphasis on Testing than Assignment 1

- 70% Performance Marks
- 10% Testing (mark will be given based on **`test_realm.c`**)
- 20% Code Style and Readability

Marked Files

- Only **`realm.c`** and **`test_realm.c`** can be submitted
- No other files will be accepted or marked
- Remember not to make any changes to the other files!
- Every submission via **`give`** is saved . . . use it as often as saving your files

Marking and Assessment

Pass Mark - Stage 1, reasonably readable code

- *Adding Locations to the Realm*
  - Inserting nodes into a linked list
- *Printing the Realm*
  - Traversing a linked list and calling functions
- Reasonable attempt at readable code

Marking and Assessment

Credit - Stages 1 and 2, readable code and testing

- *Adding Towers*
  - Insertion into a linked list at an arbitrary Location
- *Adding Enemies*
  - Creating a new linked list at every Location and adding to it
- *Advancing Enemies*
  - Changing the pointers that aim at linked lists
- *Testing*
  - Some testing in **`test_realm.c`** for written functions

Marking and Assessment

Distinction - Stages 1 to 3, very readable code and comprehensive testing

- *Applying Damage*
  - Ability to loop through a linked list, check for certain status and apply changes
- *Freeing Memory*
  - A program free of memory leaks that cleans up memory whenever it doesn't need it
- *Testing*
  - Testing of all new functions
- *Code Style*
  - Very good code style! Helper functions, useful variable names, easy to understand code

Marking and Assessment

High Distinction - Stage 1 to 4, reusable code and complete testing

- *Searching*
  - The ability to test strings and find matches
  - More advanced levels can find partial strings as well as use wildcards
- *Buffs*
  - Use searching to find particular list nodes to make changes
- *Testing*
  - Test all new functions
  - Test some functions for uncommon inputs and interesting cases
- *Style*
  - Easily understandable code
  - Functions that are easy to reuse and sometimes help in multiple situations

Marking and Assessment

Full Marks - All Stages, beautiful code and comprehensive testing

- *Effects*
  - Special conditions on Towers
  - Removing some elements of a linked list and merging them alphabetically into another list
- *Testing*
  - Full testing of all functions
  - Testing on different inputs that are likely to appear and cause issues
- *Style*
  - Clean solutions to problems that hardly need programming ability to understand

What did we cover today?

Linked Lists

- Removal
- Memory cleaning

Assignment 2 - Castle Defense

- Theme
- Structure
- Testing
- How to approach the assignment
- Marking Scheme