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

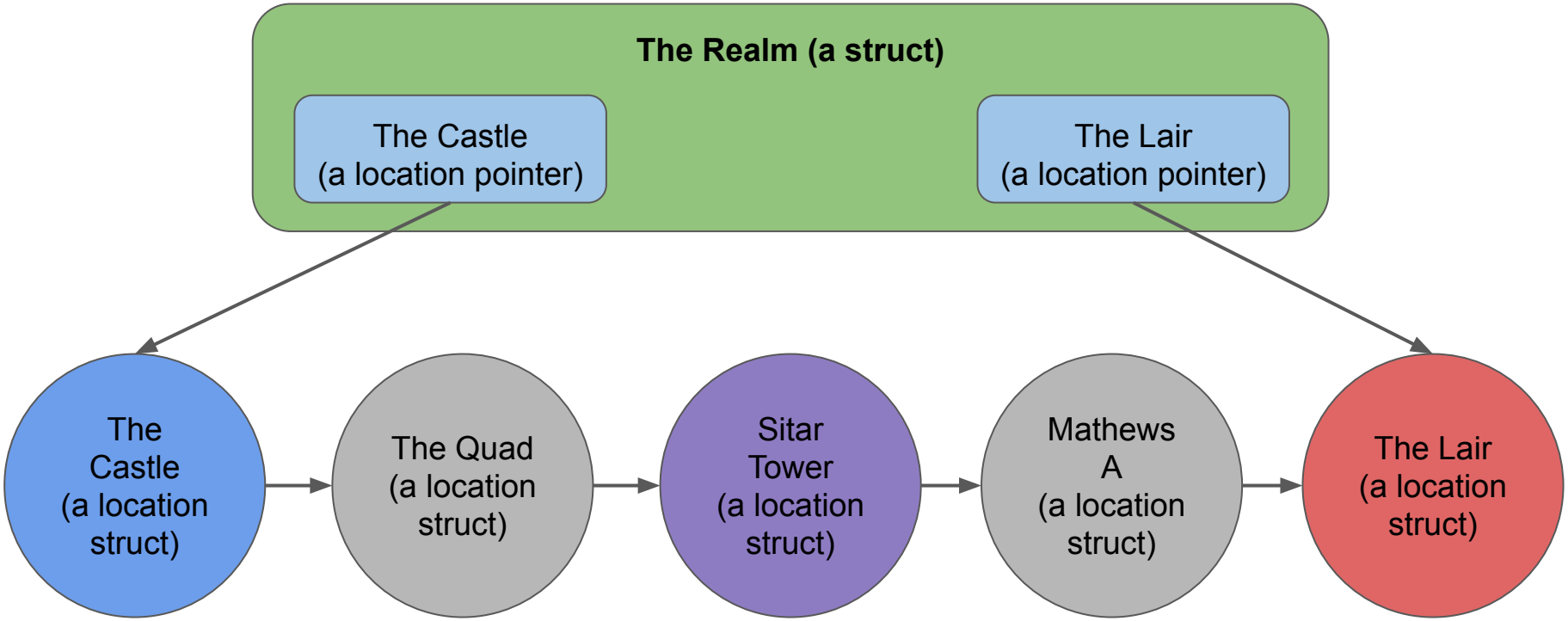
- `1511 castle_defense` runs the reference solution
- We start off by creating some lands
- Use `?` to list the commands
- We can print out the current state of the realm
- We can add towers
- We can add enemies
- We can move the enemies to their next location
- We can calculate damage done

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 . . .

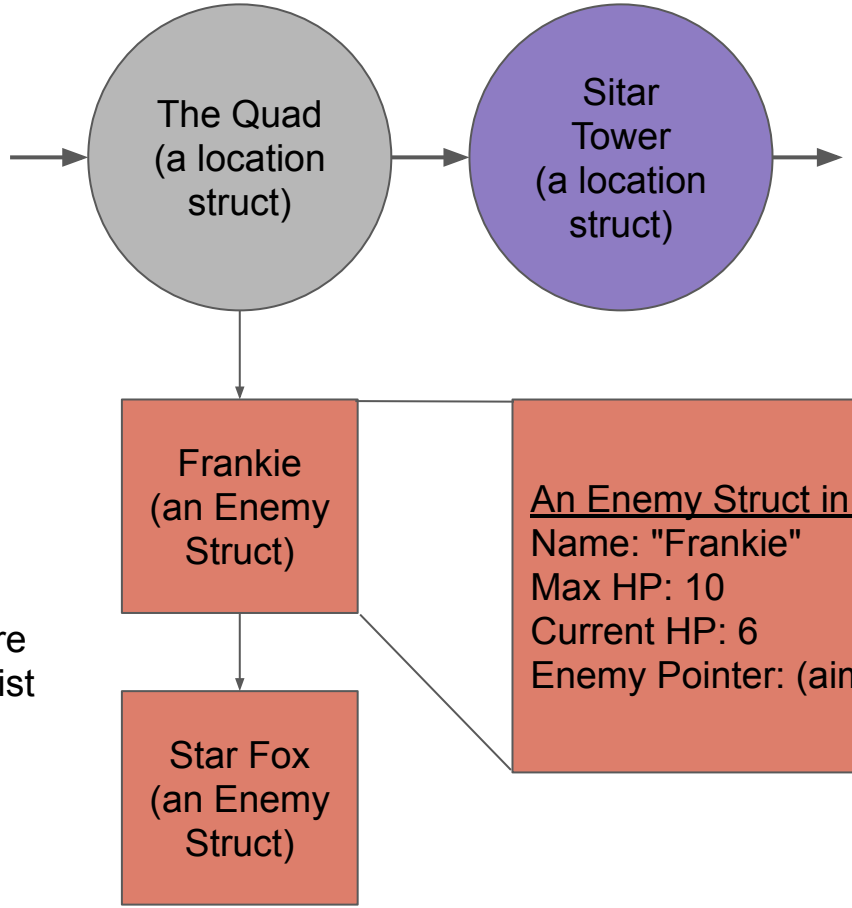
The Realm



Between The Castle and The Lair 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 a package of files
- Note that only **`realm.c`** and **`test_realm.c`** can be edited
- The rest are just links to our files and can't be edited for this 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`** and **`test_realm.h`** contain 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`** shows some examples of basic tests
- Comments show which behaviour you should test
- **`test_realm.c`** shows you a nice way of setting up automated testing of individual functions
- This is often called **"Unit Testing"**

Test and Autotests

We're going to run some automatic tests of your tests

- We'll compile your `test_realm.c` for some of the autotests
- We'll then run your tests with our own sometimes incorrect implementation of realm
- We'll let you know whether your tests are able to find the correct and incorrect behaviour

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 (50/100) - 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
- Some tests written

Marking and Assessment

Credit (65/100) - 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
- Testing
  - Some testing in **`test_realm.c`** for stages 1 and 2
- Readable code
  - Not necessarily perfect style, but a good attempt

Marking and Assessment

80/100 - 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
- *Advancing Enemies*
  - Changing the pointers that aim at linked lists
- *Freeing Memory*
  - A program free of memory leaks that cleans up memory whenever it doesn't need it
- *Testing*
  - Testing stages 1-3
- *Great Code Style*
  - Very good code style! Helper functions, useful variable names, easy to understand code

Marking and Assessment

90/100 - 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