COMP1511 - Programming Fundamentals

Week 9 - Lecture 15

What are we covering today?

Abstract Data Types

- A recap of Multiple File Projects
- More detail on things like typedef
- The ability to present capabilities of a type to us . . .
- . . . without exposing any of the inner workings

Recap - Multiple File Projects

Separating Code into Multiple files

- Header file (`*.h`) - Function Declarations
- Implementation file (`*.c`) - Majority of the running code
- Other files - can include a Header to use its capabilities

Separation protects data and makes functionality easier to read

- We don't have access to internal information we don't need
- We can't accidentally change something important
- We have a simple list of functions we can call

Using Multiple Files

Linking the Files

- A file that `#include`s the Header (`*.h`) file will have access to its functions
- It's own implementation (`*.c`) file will always `#include` it
- Implementation files are never included!

Compilation

- All Implementation files are compiled
- Header files are never compiled, they're included

An Example - A Realm

Assignment 2 - Castle Defense is a nice example

`realm.h`

- Contains only defines, typedefs and function declarations
- Is commented heavily so that it's easy to know how to use it

大概

`realm.c`

具体

- Contains actual structs
- Contains implementation of `realm.h`'s functions (once we've written them)

An Example - A Realm

How some of the other files interact . . .

`main.c`

- `#includes realm.h`
- Uses the functions in `realm.h`

`test_realm.c`

- `#includes realm.h`
- Is mutually exclusive with `main.c` because they both have main functions

抽象

Abstract Data Types

Types we can declare for a specific purpose

- We can name them
- We can fix particular ways of interacting with them
- This can protect data from being accessed the wrong way

We can hide the implementation

- Whoever uses our code doesn't need to see how it was made
- They only need to know how to use it

don't know how it make

Typedef

Type Definition

- We declare a new Type that we're going to use
- **`typedef <original Type> <new Type Name>`**
- Allows us to use a simple name for a possibly complex structure
- More importantly, hides the structure details from other parts of the code

```
typedef struct realm *Realm;
```

- We can use **`Realm`** as a Type without knowing anything about the struct underlying it

Typedef in a Header file

only use function

接口

The Header file provides an interface to the functionality

- We can put this in a **header** (*.h) file along with functions that use it
- This allows someone to see a Type without knowing exactly what it is
- The details go in the *.c file which is not included directly
- We can also see the functions without knowing how they work

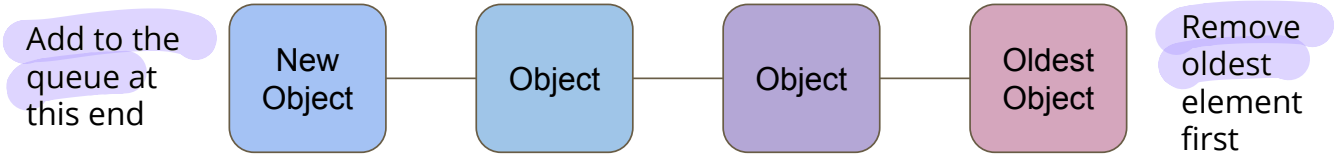
- We are able to see the **header** and use the information
- We hide the **implementation** that we don't need to know about

An Example of an Abstract Data Type - A Queue

What's a queue?

- You should be reasonably familiar with the concept
- In the human world, we sometimes line up for things
- New things join the back of the queue
- Whatever's been there the longest will be the first thing to leave the queue

first in first out



What makes it Abstract?

A Queue is an idea

- An Array or a Linked List is a very specific implementation
- A Queue is just an idea of how things should be organised
- There's a structure, but there's no implementation!

Abstract Data Type for a Queue

- We can have a header saying how the Queue is used
- The Implementation could use an Array or a Linked List to store the objects in the Queue, but we wouldn't know!

Let's build a Queue ADT

We're only concerned with how we'll use it, not what it's made of

- Our user will see a "Queue" rather than an Array or Linked List
- We will start with a Queue of integers
- We will provide access to certain functions:
  - Create a Queue
  - Destroy a Queue
  - Add to the Queue
  - Remove from the Queue
  - Count how many things are in the queue

A Header File for Queue

```
// queue type hides the struct that is is
// implemented as
typedef struct queueInternals *Queue;

// functions to create and destroy queues
Queue queueCreate(void);
void queueFree(Queue q);

// Add and remove items from queues
// Removing the item returns the item for use
void queueAdd(Queue q, int item);
int queueRemove(Queue q);

// Check on the size of the queue
int queueSize(Queue q);
```

What does our Header (not) Provide?

Standard Queue functions are available

- We can join the end or take the element from the front of the Queue
- We are not given access to anything else inside the Queue!
- We cannot take more than one element
- We aren't able to loop through the Queue

The power of Abstract Data Types

- They stop us from accessing the data incorrectly!

Queue.c

#include "queue.h"

Our *.c file is the implementation of the functionality

- The C file is like the detail under the "headings" in the header
- Each declaration in the header is like a title of what is implemented

- Let's start with a Linked List as the underlying data structure
- A Linked List makes sense because we can add to one end and remove from the other
- It also works because it can change length with no issues

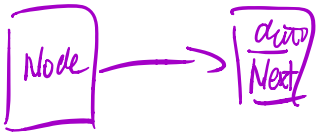
The implementation behind a type definition

We can create a pair of structs

- queueInternals represents the whole Queue
- queueNode is a single element of the list

```c
// Queue internals holds a pointer to the start of a linked list
struct queueInternals {
    struct queueNode *head;
};

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



Creation of a Queue

If we want our struct to be persistent, we'll allocate memory for it

We create our Queue empty, so the pointer to the head is NULL

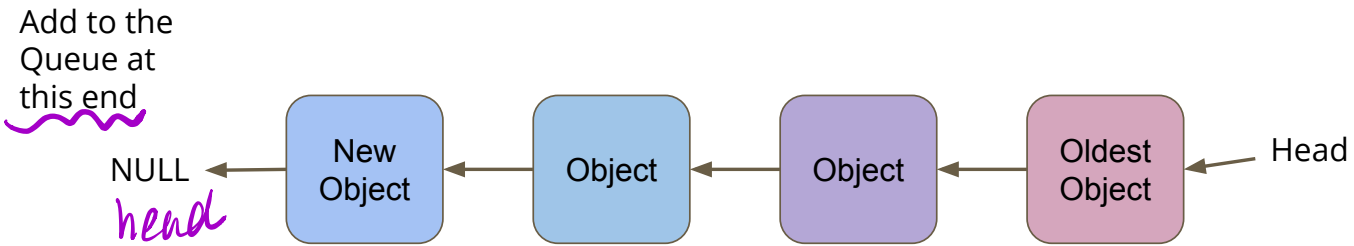
```
// Create an empty queue
Queue queueCreate(void) {
    Queue newQueue = malloc(sizeof(struct queueInternals));
    newQueue->head = NULL;
    return newQueue;
}
```

= struct queueInternals *newQueue

Adding items to the Queue

We add items to the end of the Queue

- We need to find the tail end of the Queue
- Then add an element at the end



Add Element at the end

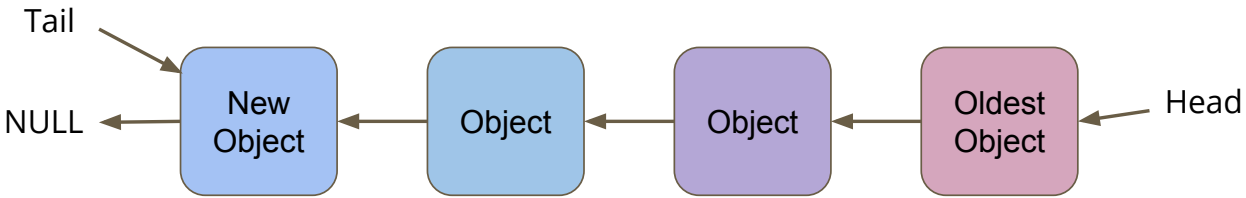
First option for adding an element at the tail end

- Loop through all the elements until the next pointer is NULL
- Add something to the end, pointing the NULL pointer at the new node

- Looping to find the end every time seems like a lot of extra work
- What if we keep track of the last element in the list using our `queueInternals` struct?

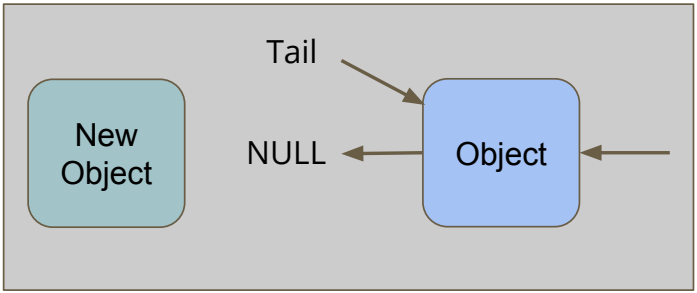
Keeping track of both ends

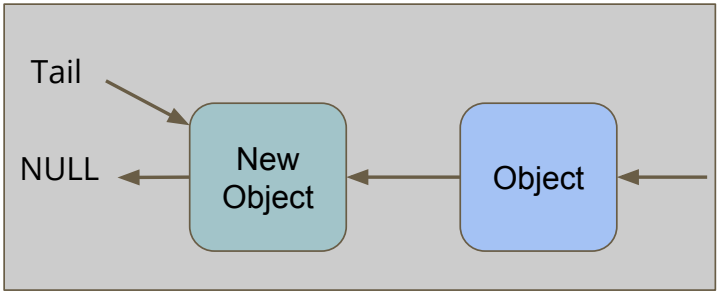
```
// Queue internals holds a pointer to the
// start and end of the linked list
struct queueInternals {
    struct queueNode *head;
    struct queueNode *tail;
};
```



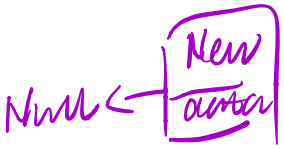
Adding to the tail

- Connect the new object to the current tail
- Move the tail pointer to the new last object
- We no longer need to loop through the whole queue to find the tail





Code for Adding




```
void queueAdd(Queue q, int item) {
    struct queueNode *newNode = malloc(sizeof(struct queueNode));
    newNode->data = item;
    newNode->next = NULL;

    if (q->tail == NULL) {
        // Queue is empty
        q->head = newNode;
        q->tail = newNode;
    } else {
        q->tail->next = newNode;
        q->tail = newNode;
    }
}
```