COMP1511 - Programming Fundamentals

Term 1, 2019 - Lecture 18
Stream B

What did we cover on Tuesday?

Linked Lists

- A complete working implementation of Linked Lists
- Removal of nodes
- Cleaning our memory

Pokédex Assignment

- Some info about the structure and approach

What are we covering today?

Working with Multiple Files

- Using Header files and including them in a project
- Compiling a project with multiple files

Abstract Data Types

- The ability to present capabilities of a type to us . . .
- Without exposing any of the inner workings

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 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

Pokemon.h

In our Header File

- Typedef (Type Define) is a way of allowing us to create our own C Type out of another Type
- Function Declarations with no definitions
- Comments that describe how the functions can be used
- No running code!

Pokemon.c

Implementation File

- Has #includes, especially #include "pokemon.h" (joins the two files together)
- Implements the struct mentioned in the typedef from the header
- Implements all the functions declared in the header
- Implements some functions for use only inside this file

- **static** marks functions as not being accessible outside the file itself
- static functions are only used as helper functions for the code in this file

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, it must #include that file
- In the assignment, main.c uses #include "pokedex.h" to be able to access your Pokedéx functionality

Compiling a Project with Multiple Files

How do we compile 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

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

Typedef

Type Definition

- We declare 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 pokemon *Pokemon;
```

- We can use "Pokemon" as a type without knowing anything about the struct underlying it

Typedef in a Header file

The Header file provides an interface to the functionality

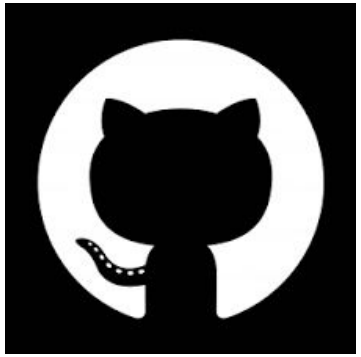
- We can put this in a header (*.h) file along with functions that use it
- This allows someone to see a Type without knowing exactly what it is
- The details go in the *.c file which is **not** included directly
- We can also see the functions without knowing how they work

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

Break Time

Keeping track of your own code projects

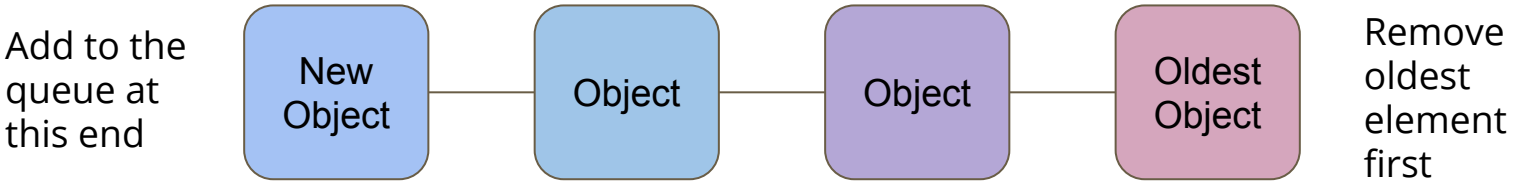
- Using **git** is a really handy way to keep backups of your work
- GitHub and BitBucket are two providers that will give you free online repositories to store your code
- Graphical Interfaces are available for git (GitHub Desktop and Sourcetree respectively)
- It takes some time to get familiar with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



Let's build 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



Functionality

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

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

```
// 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

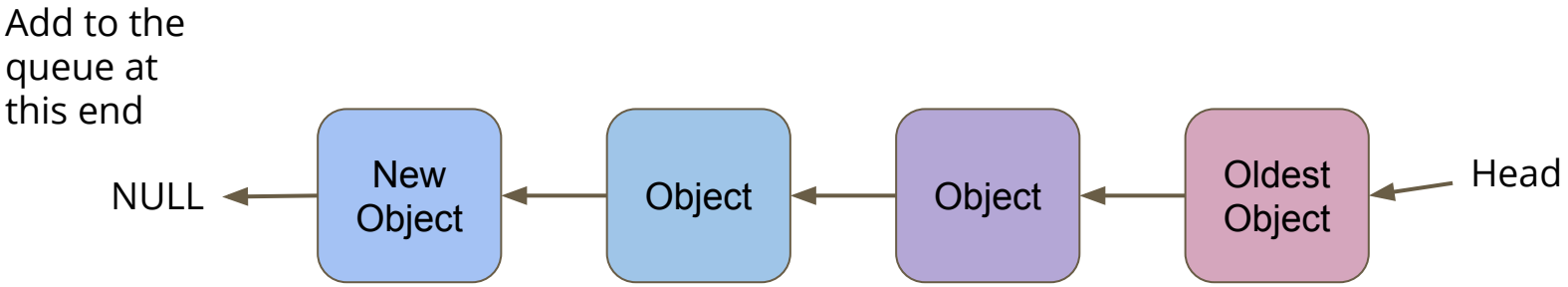
We create our queue empty, so the pointer to the head is NULL

```
// Create an empty queue
queue queueCreate(void) {
    queue newQueue = malloc(sizeof(struct queueInternals));
    if (newQueue == NULL) {
        printf("Could not allocate memory for a queue.\n");
        exit(1);
    }
    newQueue->head = NULL;
    return 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

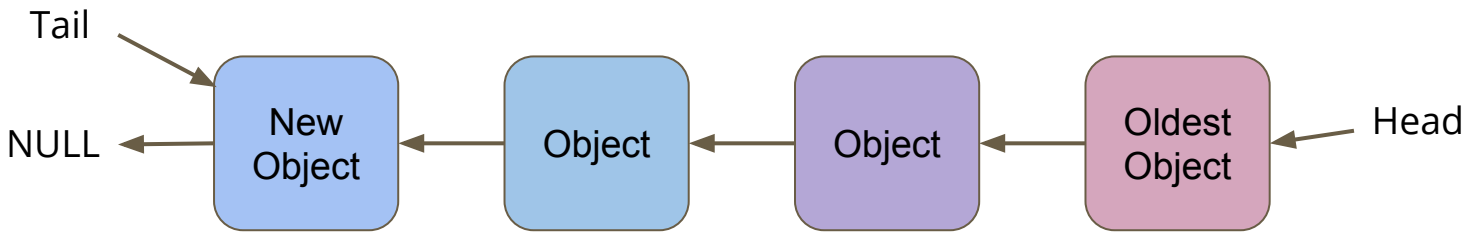
First option for adding an element at the tail end

- 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 queue_internals struct?

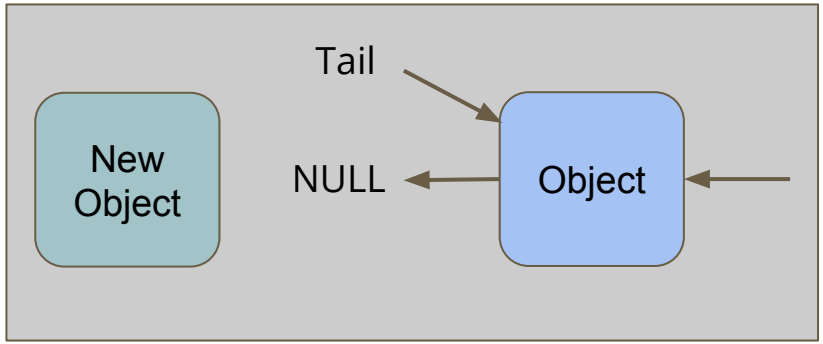
Keeping track of both ends

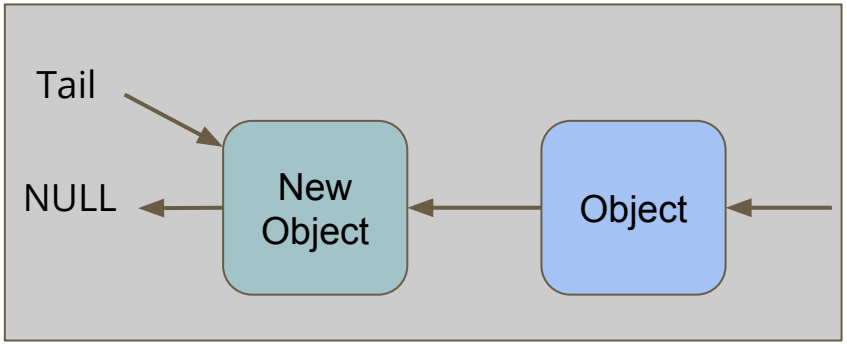
```
// Queue internals holds a pointer to the
// start and end of the linked list
struct queueInternals {
    struct queueNode *head;
    struct queueNode *tail;
};
```



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





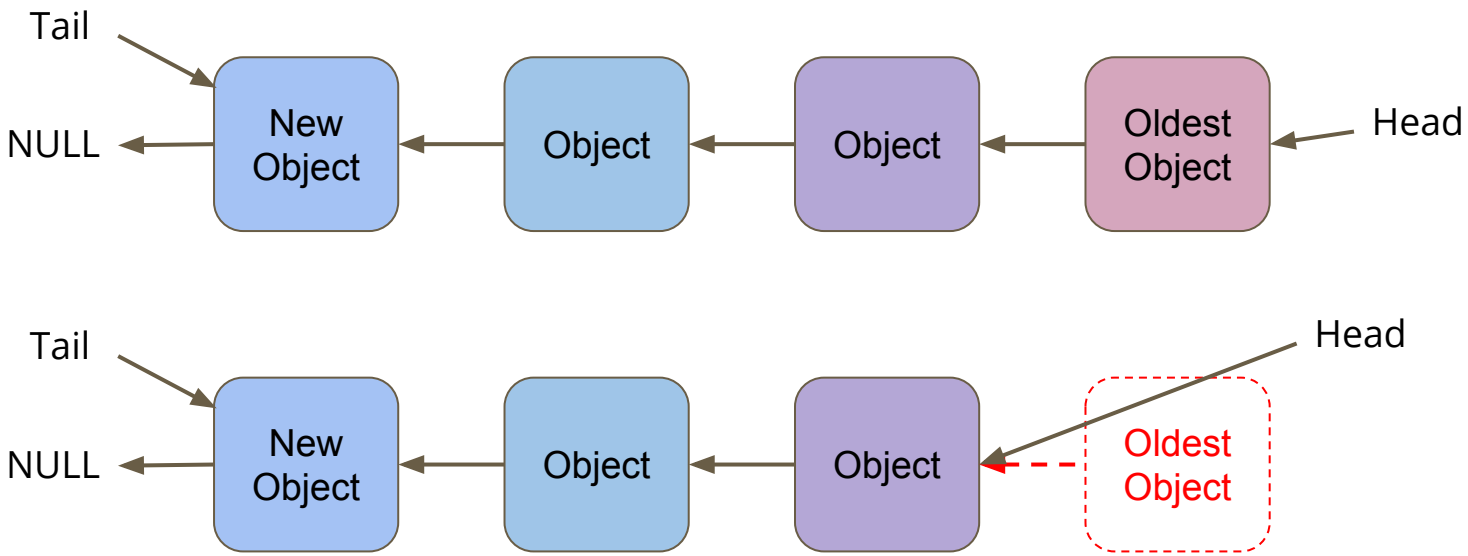
Code for Adding

```
void queueAdd(queue q, int item) {
    struct queueNode *newNode = malloc(sizeof(struct queueNode));
    if (newNode == NULL) {
        printf("Could not allocate memory for a node.\n");
        exit(1);
    }
    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;
    }
}
```

Removing a Node

The only node that can be removed is the head (the oldest node)



Code for Removing

```
// Remove the head from the list and free the memory used
int queueRemove(queue q) {
    if (q->head == NULL) {
        printf("Attempt to remove an element from an empty queue.\n");
        exit(1);
    }
    // Keep track of the old head
    int returnData = q->head->data;
    struct queueNode *remNode = q->head;

    // move the queue to the new head and free the old
    q->head = q->head->next;
    free(remNode);

    return returnData;
}
```

Testing Code in our Main.c

```
int main(void) {
    printf("Creating the Queue of Pokemon.\n");
    queue pokeQueue = queueCreate();
    int id = 1;
    printf("Pokemon ID %d joins the parade!\n", id);
    queueAdd(pokeQueue, id);
    id = 2;
    printf("Pokemon ID %d joins the parade!\n", id);
    queueAdd(pokeQueue, id);
    id = 3;
    printf("Pokemon ID %d joins the parade!\n", id);
    queueAdd(pokeQueue, id);

    printf("Pokemon ID %d just walked past!\n", queueRemove(pokeQueue));
    printf("Pokemon ID %d just walked past!\n", queueRemove(pokeQueue));
    printf("Pokemon ID %d just walked past!\n", queueRemove(pokeQueue));
    return 0;
}
```

Other Functionality

There are some functions in the header we haven't implemented

- **Destroying and freeing the Queue**
- We're still at risk of leaking memory because we're only freeing on removal
- **Display the Number of Elements**
- This would be very handy because it would allow us to tell how many elements we can remove before we risk errors

We'll finish these and look at more next week!

What did we cover today?

Multiple Files in a Project

- Organisation and Compilation

Abstract Data Types

- Using multiple files to control how a type is used
- Hiding the implementation
- Providing a fixed interface
- Our demo is a partly implemented Queue