COMP1511 - Programming Fundamentals

Term 1, 2019 - Lecture 19
Stream B

What did we cover last week?

Linked Lists

Pokédex Assignment

Projects with Multiple Files

- Using and Compiling Multiple Files

Abstract Data Types - Queues

- Providing functionality and hiding the implementation

What are we covering today?

Finishing our Queue Implementation

- Destroying and Freeing
- Returning the number of items

Another Abstract Data Type

- Stacks
- Implementing with other data structures

Recap - Abstract Data Types

Making our own types with specific uses

- Declare our functionality in a Header (*.h) file
- Hide our implementation in a *.c file

- The Header declares the type and the functions
- All the implementation is left out of the header

- The C file defines the underlying implementation

Finishing our Queue

A few leftovers

- We're not cleaning our memory properly yet
- So we need a function that frees an entire queue

- Also, a function that returns how many items are in the queue
- This makes it easier for someone to use without risking errors

Our queue.h Header File

```
// queue type hides the struct that it 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);
```

queueFree()

Free all the memory in the linked list that we're using

- Loop through the list
- free() each node as we go

```
// Destroy and Free the entire queue
void queueFree(queue q) {
    while (q->head != NULL) {
        struct queueNode *current = q->head;
        q->head = q->head->next;
        free(current);
    }
}
```

Testing for memory leaks

Let's use dcc --leakcheck

```
int main(void) {
    queue pokeParade = queueCreate();
    queueAdd(pokeParade, 1);
    queueAdd(pokeParade, 2);
    queueAdd(pokeParade, 3);

    queueFree(pokeParade);
}
```

- What happens when we run with memory leak checking?
- Remember that all memory allocated with malloc() must be freed!

queueFree() Improved

Remember to free all the memory allocations!

```
// Destroy and Free the entire queue
void queueFree(queue q) {
    while (q->head != NULL) {
        struct queueNode *current = q->head;
        q->head = q->head->next;
        free(current);
    }
    free(q);
}
```

Number of items in the Queue

Our last function is queueSize()

- Loop through the list until the end
- Count how many elements are in it

```
// Return the number of items in the queue
int queueSize(queue q) {
    struct queueNode *iterator = q->head;
    int counter = 0;
    while(iterator != NULL) {
        counter++;
        iterator = iterator->next;
    }
    return counter;
}
```

Can we be trickier?

Maybe we don't want to loop through the whole list every time?

- We have a queueInternals struct that can store information
- How about we store the size there?

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

- Then, whenever we add or remove a node, we add or subtract 1 from this variable

Completing our Queue

To go along with our size variable . . .

- queueCreate will set the size to 0
- queueAdd will add 1
- queueRemove will subtract 1

In our testing main, we can now show this working with a loop:

```
    printf("There are %d Pokemon in the parade.\n", queueSize(pokeParade));
    while(queueSize(pokeParade) > 0) {
        printf("Pokemon ID: %d just walked past.\n", queueRemove(pokeParade));
    }
```

More thoughts on the Queue

Whatever includes the queue only sees the header

- When we're using ADTs we don't know (or need to know) the implementation
- What if this queue had been implemented using an array?
- It's entirely possible . . . and worth thinking about how you might make it work

Break Time

Programming Languages

- C++, Java, C# and many others are based directly on C
- There are too many programming languages to count or learn!

- Remember the fundamentals!
- C syntax is not as important as your plans and thinking
- You will encounter many programming languages, some will feel very different from C in their approach
- But if you learn how you want to communicate with computers, the actual language you use will never be a barrier for you

Course News

Pokédex Videos

- Having trouble getting started?
- AndrewB has recorded some videos which will show you the basics of how to use the Pokédex and its files (link is on the course website in the Notices and also: http://andrewb.wiki/pokedex)

Exam revision exercises

- Available on the course website under Course Work
- Useful for going back over older content while you're studying

Course News

Public holiday timetable

- No lecture or tute/labs this Thursday
- The final Lecture will be on Tuesday 30th April at 11am
- Thursday tute/labs will also be on Tuesday 30th April

- Yesterday (Monday) tute/labs will be on Monday 29th April
- Friday tute/labs will have a tutorial on Wednesday 1st May

Stacks - another Abstract Data Type

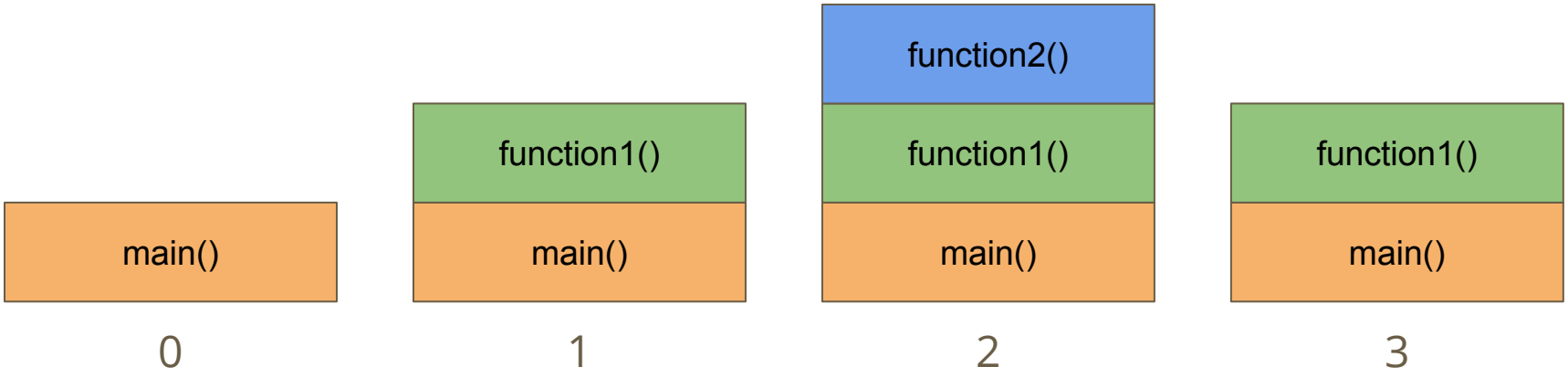
A stack is a very common data structure in programming

- It is a "Last in first out" structure
- You can put something on top of a stack
- You can take something off the top of a stack
- You can't access anything underneath

This is actually how functions work!

The currently running code is on the top of the stack

1. main calls function1
2. function1 calls function2
3. control returns to function1 when function2 returns



What kind of functions does a stack need?

Functionality to put in a header file

- create
- free
- push (add to the top of the stack)
- pop (remove from the top of the stack)
- top (show the top without removing it)
- size

We'll only be showing some of these today

A Stack Header

Looks eerily familiar . . .

```
// stack type hides the struct that it is implemented as
typedef struct stackInternals *stack;

// functions to create and destroy stacks
stack stackCreate(void);
void stackFree(stack s);

// Push and Pop items from stacks
// Removing the item returns the item for use
void stackPush(stack s, int item);
int stackPop(stack s);

// Check on the size of the queue
int stackSize(stack s);
```

Implementation

What is our internal data structure going to be?

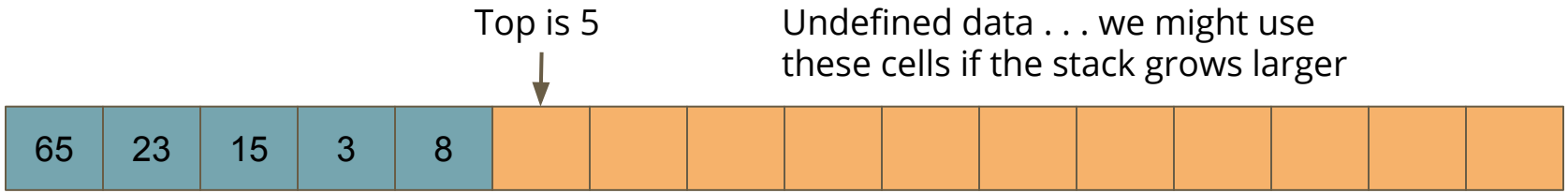
- We could use a linked list again
- We could use an array
- Whichever it is, it should be invisible to whoever includes the stack.h file

- For this example, let's use an array (just for a change)
- Our data will be stored in an array with a large maximum size
- We'll keep track of where the top is with an int

Array Implementation of a stack

A large array where only some of it is used

- Top is a particular index
- Top signifies where our data ends
- It also happens to tell us how big our stack is!



stack.c

```
// Struct representing the stack using an array
struct stackInternals {
    int stack[MAX_STACK_SIZE];
    int top;
};

// create a new stack
stack stackCreate() {
    stack s = malloc(sizeof(struct stackInternals));
    if (s == NULL) {
        printf("Cannot allocate memory for a stack.\n");
        exit(1);
    }
    s->top = 0;
    return s;
}
```

Push and Pop

These should only interact with the top of the stack

- **Push** should add an element after the end of the stack
- It should then move the top index to that new element

- **Pop** should return the element on the top of the stack
- It should then move the top index down one

Push code

```
// Add an element to the top of the stack
void stackPush(stack s, int item) {
    // check to see if we've used up all our memory
    if(s->top == MAX_STACK_SIZE) {
        printf("Maximum stack size reached, cannot push.\n");
        exit(1);
    }
    s->stackData[s->top] = item;
    s->top++;
}
```

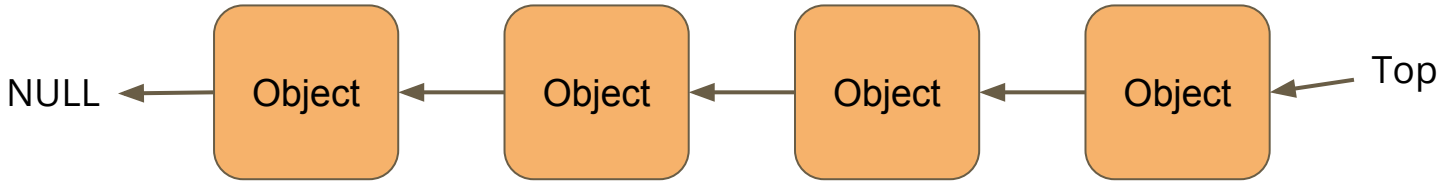
Pop code

```
// Remove an element from the top of the stack
int stackPop(stack s) {
    // check to see if the stack is empty
    if(s->top <= 0) {
        printf("Stack is empty, cannot pop.\n");
        exit(1);
    }
    s->top--;
    return s->stackData[s->top];
}
```

What if this were a linked list?

Implementation should be invisible to the including code

- Let's try to implement the same functions with a linked list
- We'll add elements to the end
- We'll also remove elements from the same end



Linked List Implementation

```
struct stackInternals {
    struct node *top;
};
struct node {
    struct node *next;
    int data;
};

stack stackCreate() {
    stack s = malloc(sizeof(struct stackInternals));
    if (s == NULL) {
        printf("Cannot allocate memory for a stack.\n");
        exit(1);
    }
    s->top = NULL;
    return s;
}
```

Push and Pop with a Linked List

All of our changes will apply to the top of the list

- **Push** adds an element to the top of the list
- Top will then point at that element

- **Pop** removes the top element of the list and returns it
- Top will then point at the next element

Push Code

```
// Add an element on top of the stack
void stackPush(stack s, int item) {
    struct node *n = malloc(sizeof (struct node));
    if (n == NULL) {
        printf("Cannot allocate memory for a node.\n");
        exit(1);
    }
    n->data = item;
    n->next = s->top;
    s->top = n;
}
```

Pop code

```
// Remove the top element from the stack
int stackPop(stack s) {
    if(s->top == NULL) {
        printf("Stack is empty, cannot pop.\n");
        exit(1);
    }
    // keep a pointer to the node so we can free it
    struct node *n = s->top;
    int item = n->data;
    s->top = s->top->next;
    free(n);
    return item;
}
```

Hidden Implementations

Neither Implementation needs to change the Header

- The main function doesn't know the difference!
- The structures and implementations are hidden from the header file and the rest of the code that uses it
- If we want or need to, we can change the underlying implementation without affecting the main code

What did we learn today?

Abstract Data Types

- Complete implementation of the Queue using a Linked List
- Partial implementation of a Stack using an Array
- Showing that we can also implement the Stack using a Linked List
- Hidden implementations mean they can change if we want!

We're finished for new content for COMP1511

- Next week's lecture will be a recap and strategies for the exam