COMP1511 PROGRAMMING FUNDAMENTALS

Lecture 15

Abstract Data Types: Stacks

LAST WEEK...

- Inserting into a linked list anywhere
- Searching through the linked list for specific conditions
- Deleting from a linked list

TODAY...

- Some more linked lists - seeing the linked list within the linked list structure, and looking at more boundary cases
- Abstract Data Types: Stacks

WHERE IS THE CODE?

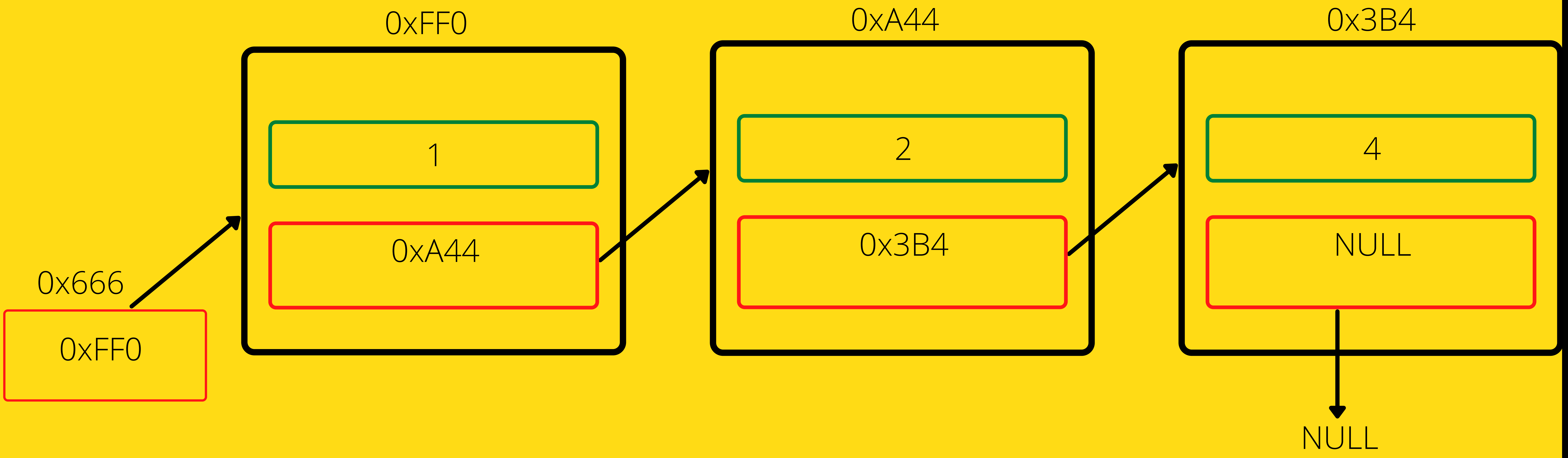
LIVE LECTURE CODE CAN BE FOUND HERE:

https://cgi.cse.unsw.edu.au/~cs1511/21T3/live/Week09/

LINKED LISTS

WHAT IS A LINKED LIST WITHIN A STRUCT?

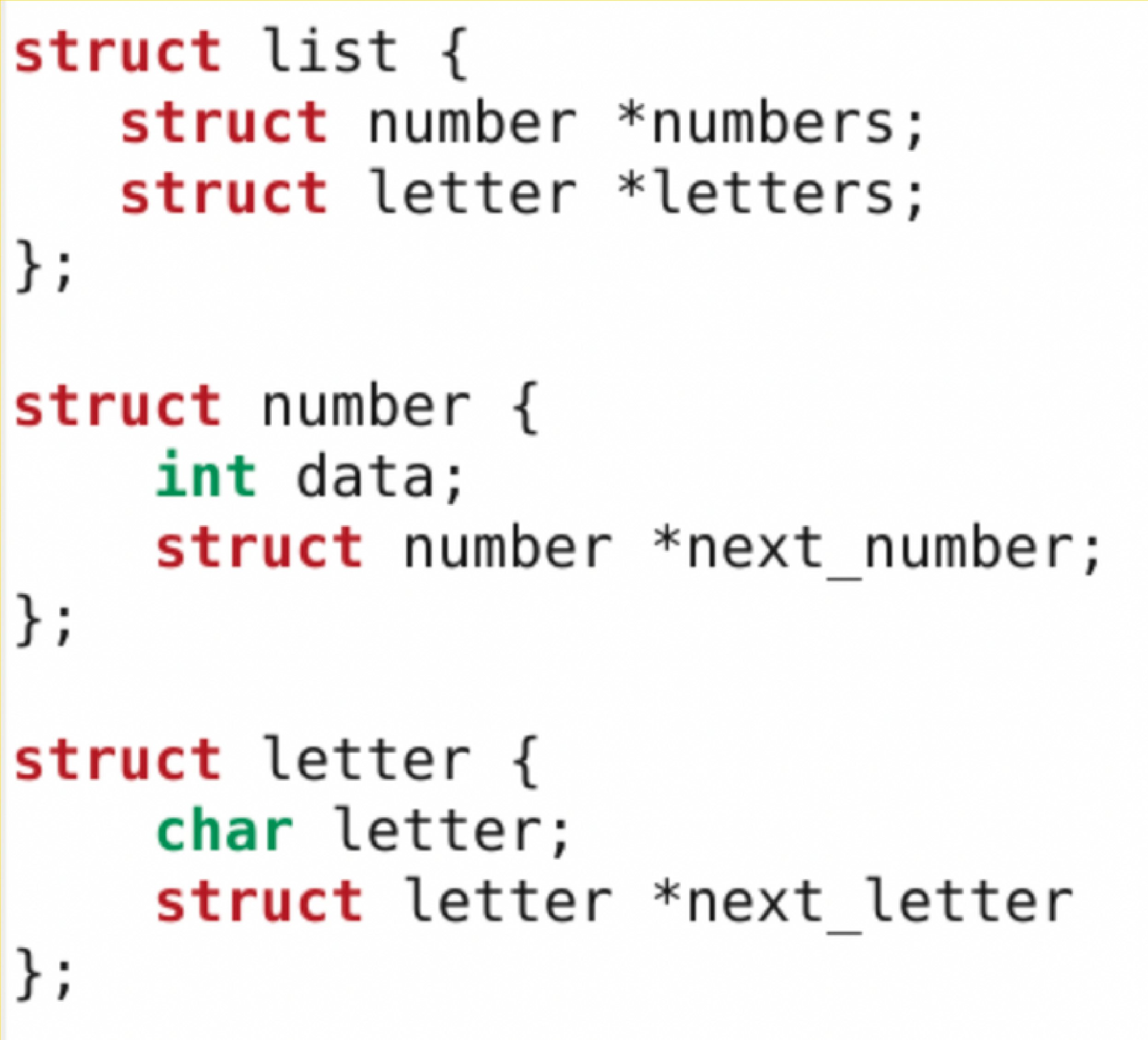
- You have seen this type of structure in your assignment.
- Usually our linked list looks like this, where the head is the first element of the list, and the head pointer stores the address of that first node:



LINKED LISTS

WHAT IS A LINKED LIST WITHIN A STRUCT?

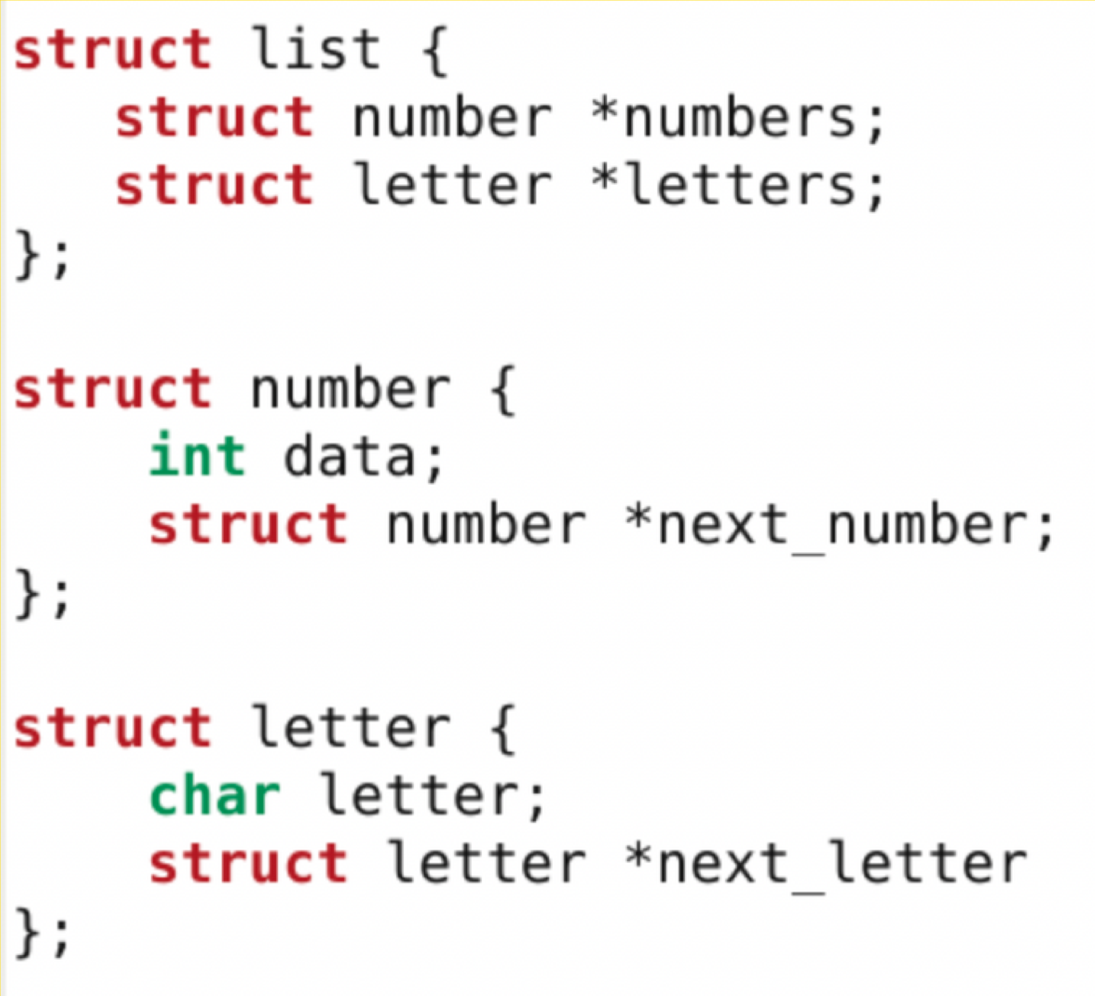
- What happens when you now have a linked list within a linked list?
- Let's say we have a list structure that contains a structure for numbers (which is a linked list) and a structure for letters (which is a linked list).
- For example consider:



```
struct list {
   struct number *numbers;
   struct letter *letters;
};

struct number {
    int data;
    struct number *next_number;
};

struct letter {
    char letter;
    struct letter *next_letter
};
```

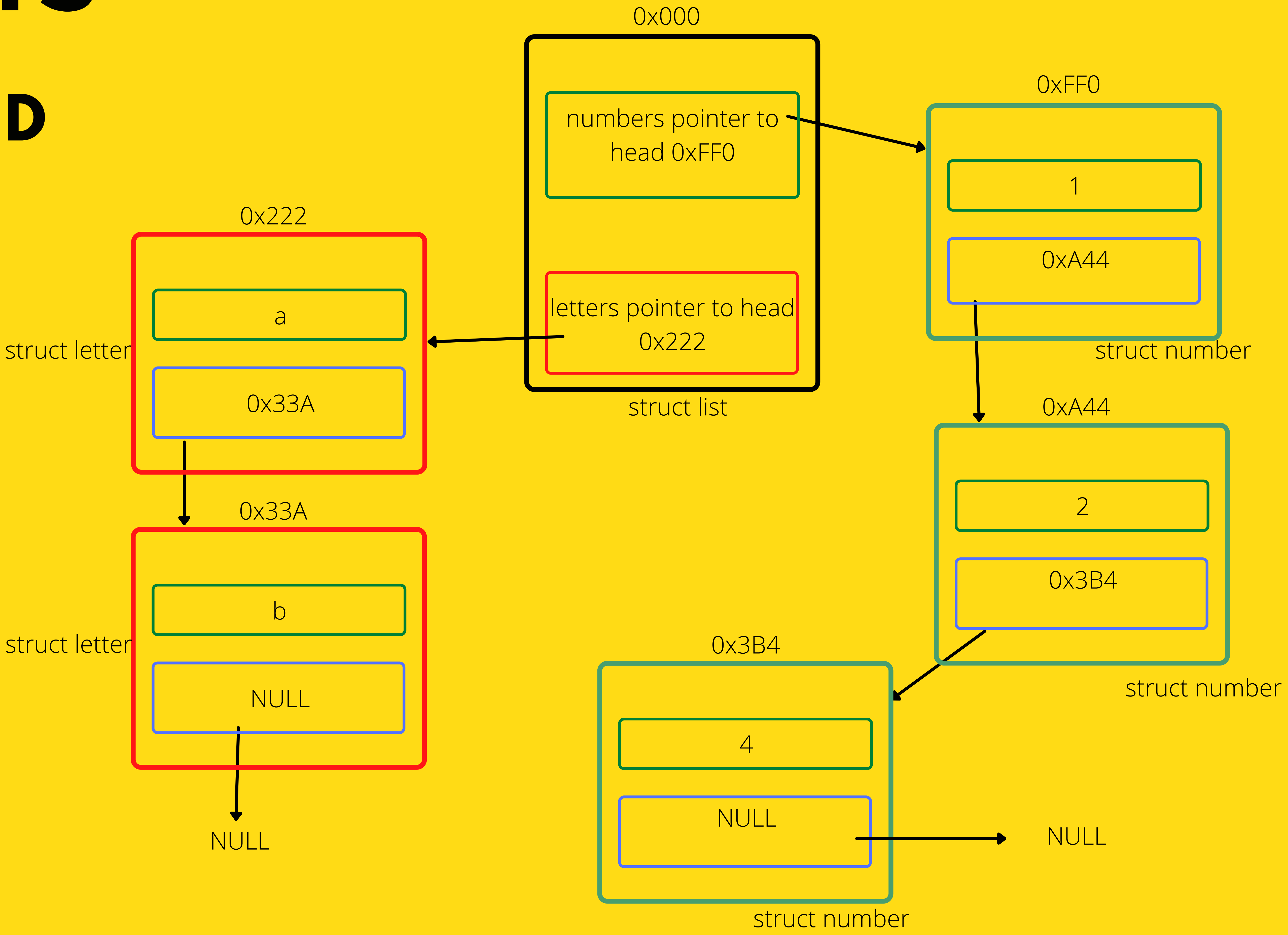
LINKED LISTS

WHAT IS A LINKED LIST WITHIN A STRUCT?



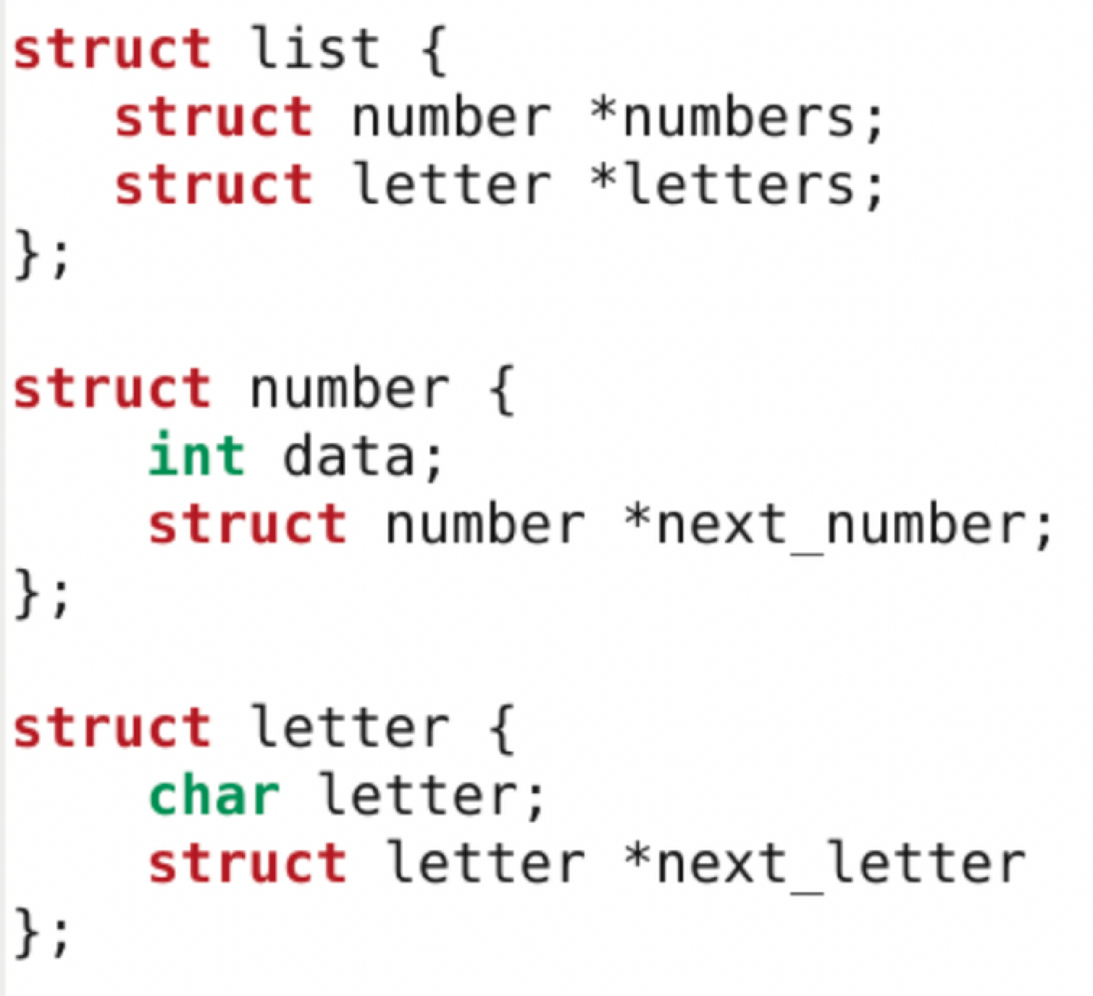
```
struct list {
   struct number *numbers;
   struct letter *letters;
};

struct number {
    int data;
    struct number *next_number;
};

struct letter {
    char letter;
    struct letter *next_letter
};
```

- Visually it looks like this:



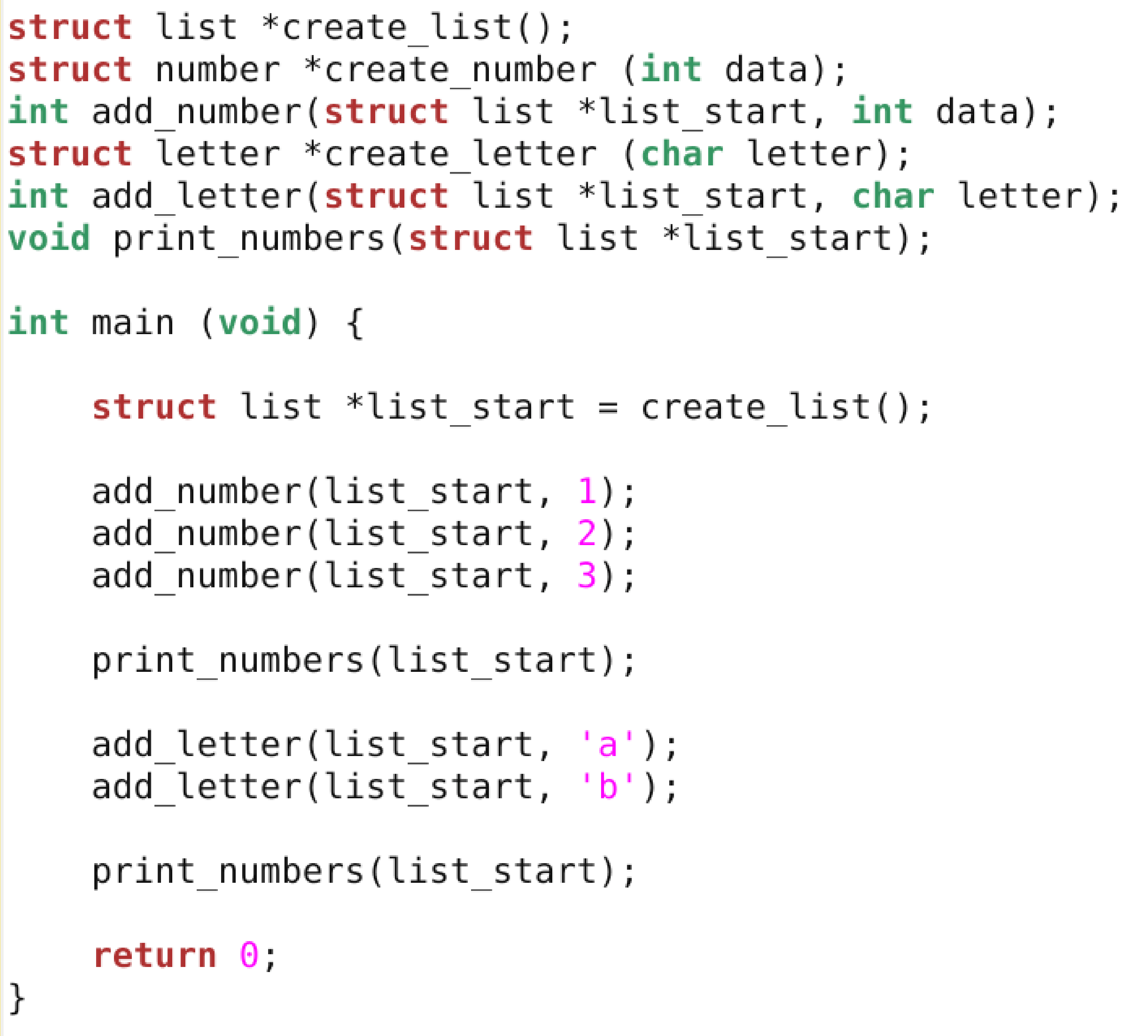
LINKED LISTS

WHAT IS A LINKED LIST WITHIN A LINKED LIST?



```c
struct list {
   struct number *numbers;
   struct letter *letters;
};

struct number {
    int data;
    struct number *next_number;
};

struct letter {
    char letter;
    struct letter *next_letter
};
```

- Let's write some code for this list...



```c
struct list *create_list();
struct number *create_number (int data);
int add_number(struct list *list_start, int data);
struct letter *create_letter (char letter);
int add_letter(struct list *list_start, char letter);
void print_numbers(struct list *list_start);

int main (void) {

    struct list *list_start = create_list();

    add_number(list_start, 1);
    add_number(list_start, 2);
    add_number(list_start, 3);

    print_numbers(list_start);

    add_letter(list_start, 'a');
    add_letter(list_start, 'b');

    print_numbers(list_start);

    return 0;
}
```

ABSTRACT DATA TYPES

WHAT ARE THEY?

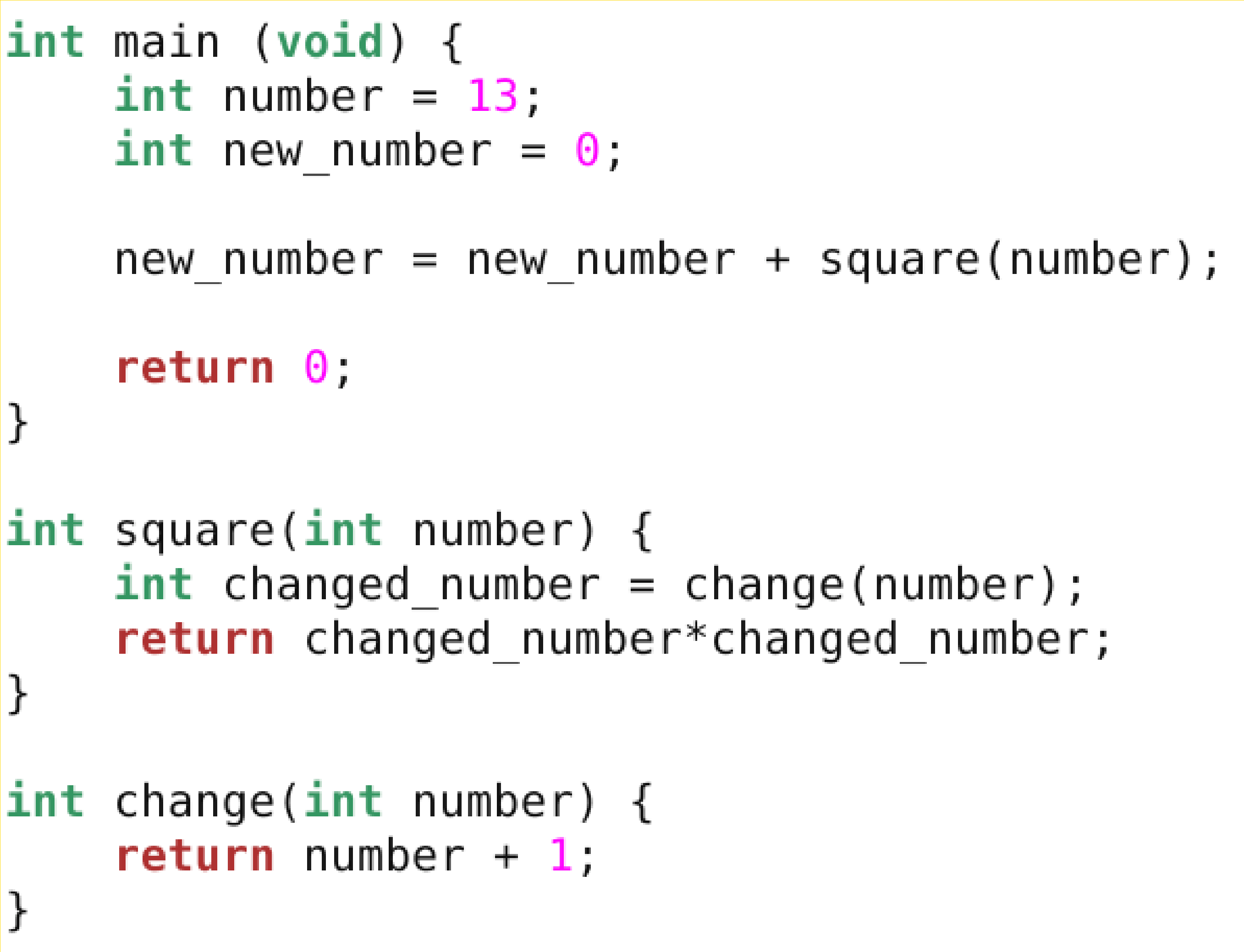
- Abstract Data Types (ADT's) are data types whose implementation details are hidden from the user
  - What does this mean?
- A common example of an ADT is something called a Stack - it has set ways in which it works but it can implemented using a number of different ways (for example, using linked lists or using arrays)
- Whoever uses our code doesn't need to see how it was made
  - They only really want to know how to use it

SO WHAT IS A STACK?

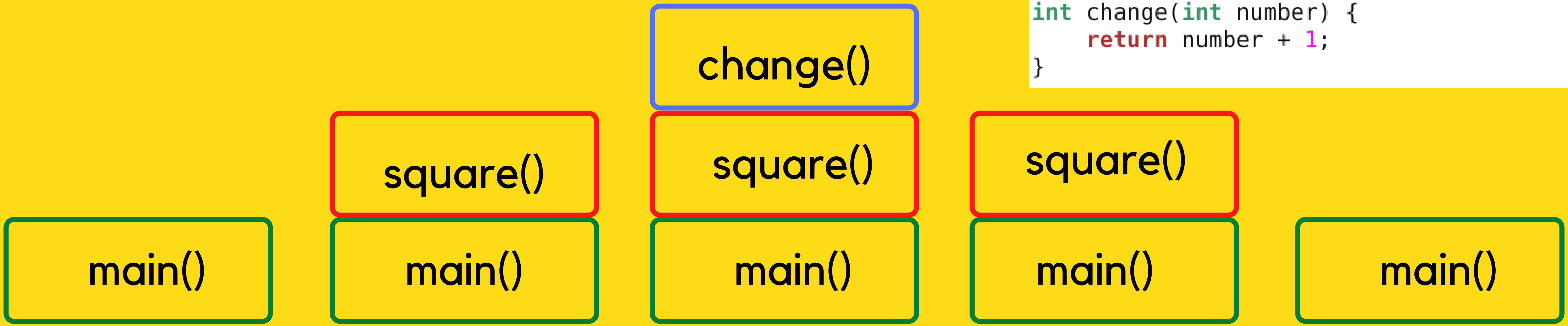
THINK DIRTY DISHES (OR EVEN CLEAN ONES!)

- A Stack is a Last In, First Out structure (LIFO)
- So you can put something on top of a stack and you can take something off the top of the stack, you cannot remove things from underneath (think of your dish stack toppling down!)

THIS IS HOW OUR MEMORY STACK WORKS FOR FUNCTIONS



```
int main (void) {
    int number = 13;
    int new_number = 0;

    new_number = new_number + square(number);

    return 0;
}

int square(int number) {
    int changed_number = change(number);
    return changed_number*changed_number;
}

int change(int number) {
    return number + 1;
}
```



the main() starts running

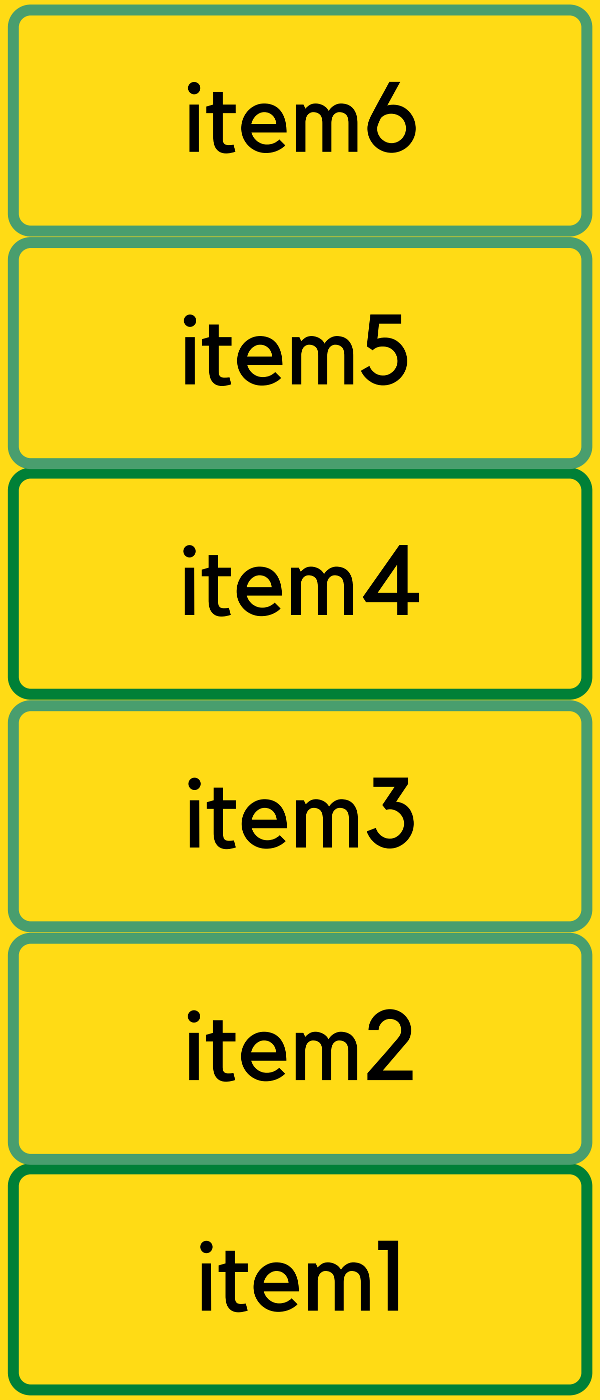
main() calls square() - only square() now accessible

square() calls change() - only change() now accessible

return from change() to square() - only square() now accessible

return from square() to main()

WHERE IS THE ABSTRACT PART?



item6

item5

item4

item3

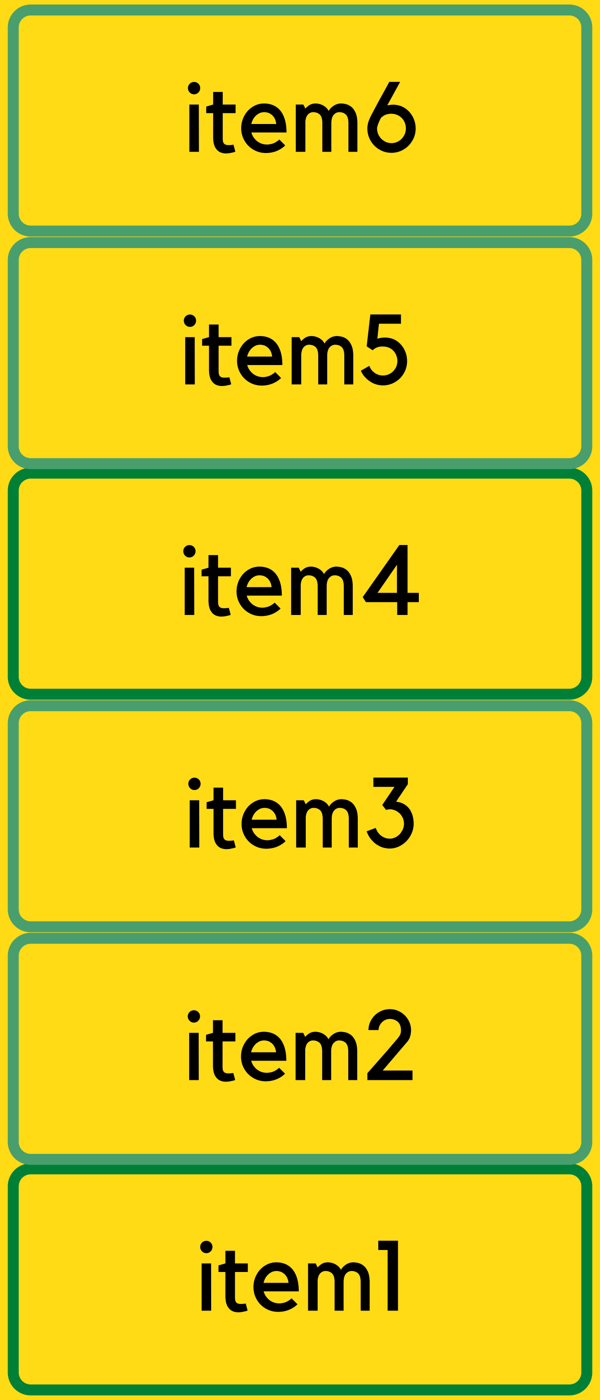
item2

item1

- The idea of a stack is just that - an idea!
- Can you think of anywhere a Stack is applied in our everyday interactions with computers?

- A stack behaves in a certain way defined by a set of rules
- I am not given an implementation for this stack
    - I can do it using arrays
    - I can do it using linked lists
- So we could have a header file that just defines how the stack is used, but it could be implemented using arrays or linked lists and we would be none the wiser - doesn't matter as long as it follows the rules of a Stack!

SO WHAT ARE THE RULES OF A STACK?


item6

item5

item4

item3

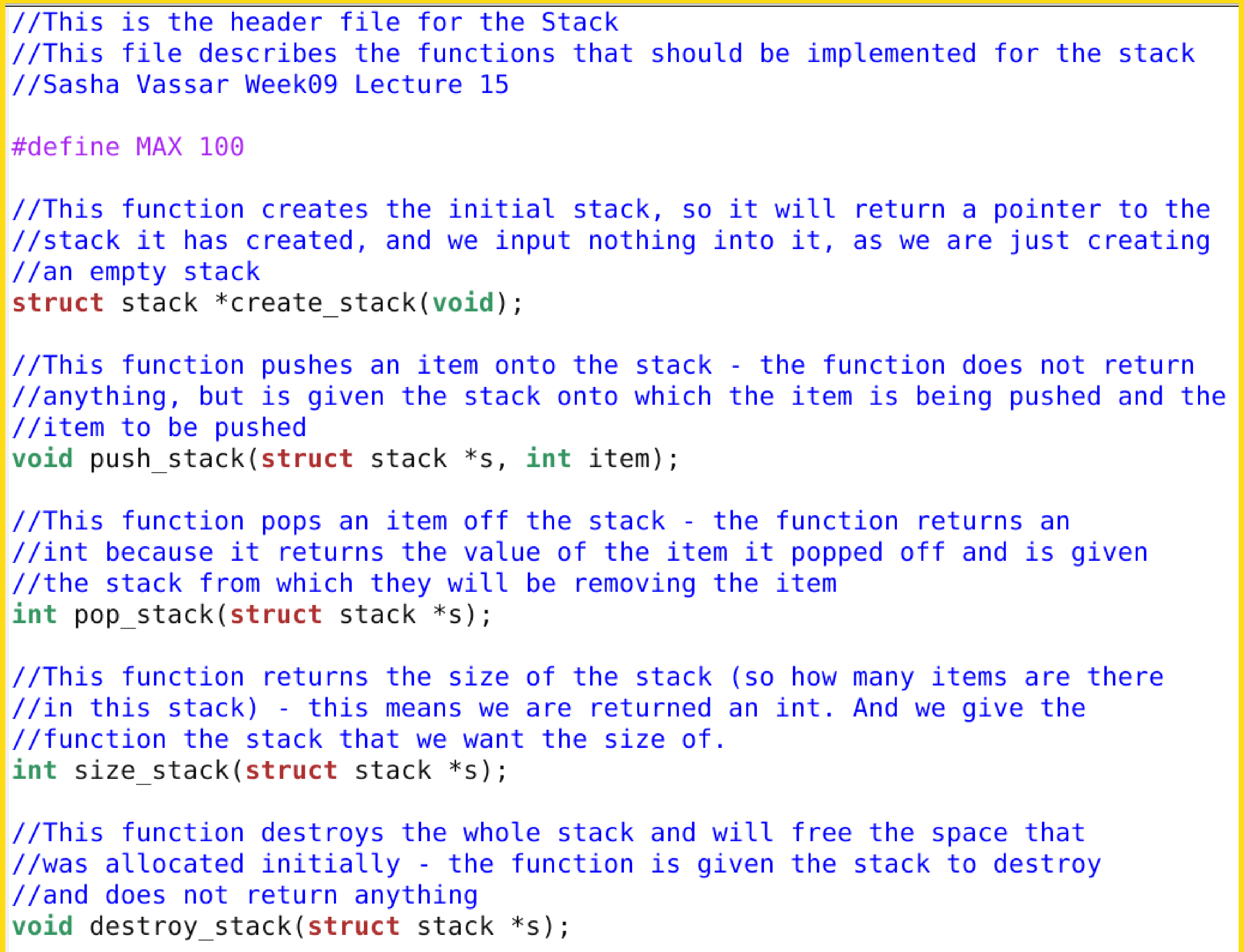
item2

item1

- The Stack has two special terms:
  - push (onto the stack, so add the element to the top of the Stack)
  - pop (off the stack, take the top element off the Stack)
- Let's look at a few functions:
  - Create a Stack
  - Add to the Stack (push)
  - Take from the Stack (pop)
  - Count how many things are in the Stack
  - Destroy the Stack
- One header file, and we will try two different implementations:
  - stack.h
  - stack_list.c
  - stack_array.c

HOW WILL THE HEADER FILE DEFINE THINGS FOR US?

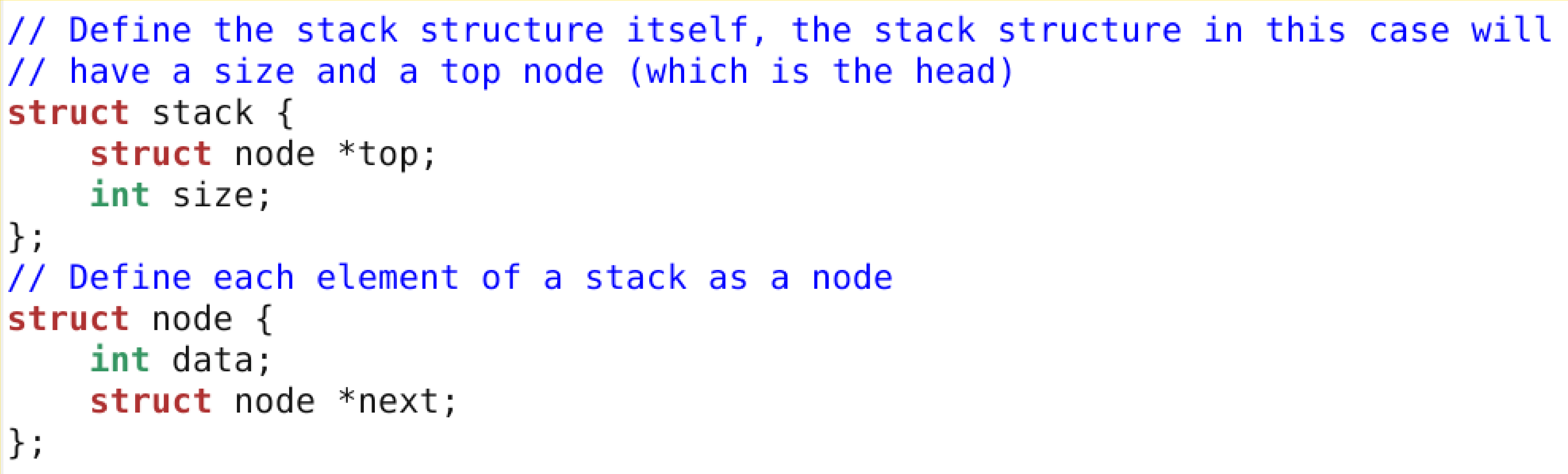
- A stack is a structure, which we will not define in the header file, as our array and linked list files may use slightly different definitions of the same structure
- We will then define our functions in the header file:



```
//This is the header file for the Stack
//This file describes the functions that should be implemented for the stack
//Sasha Vassar Week09 Lecture 15

#define MAX 100

//This function creates the initial stack, so it will return a pointer to the
//stack it has created, and we input nothing into it, as we are just creating
//an empty stack
struct stack *create_stack(void);

//This function pushes an item onto the stack - the function does not return
//anything, but is given the stack onto which the item is being pushed and the
//item to be pushed
void push_stack(struct stack *s, int item);

//This function pops an item off the stack - the function returns an
//int because it returns the value of the item it popped off and is given
//the stack from which they will be removing the item
int pop_stack(struct stack *s);

//This function returns the size of the stack (so how many items are there
//in this stack) - this means we are returned an int. And we give the
//function the stack that we want the size of.
int size_stack(struct stack *s);

//This function destroys the whole stack and will free the space that
//was allocated initially - the function is given the stack to destroy
//and does not return anything
void destroy_stack(struct stack *s);
```

LET'S DO A LINKED LIST IMPLEMENTATION FIRST

STACK: DEFINING A LINKED LIST STACK

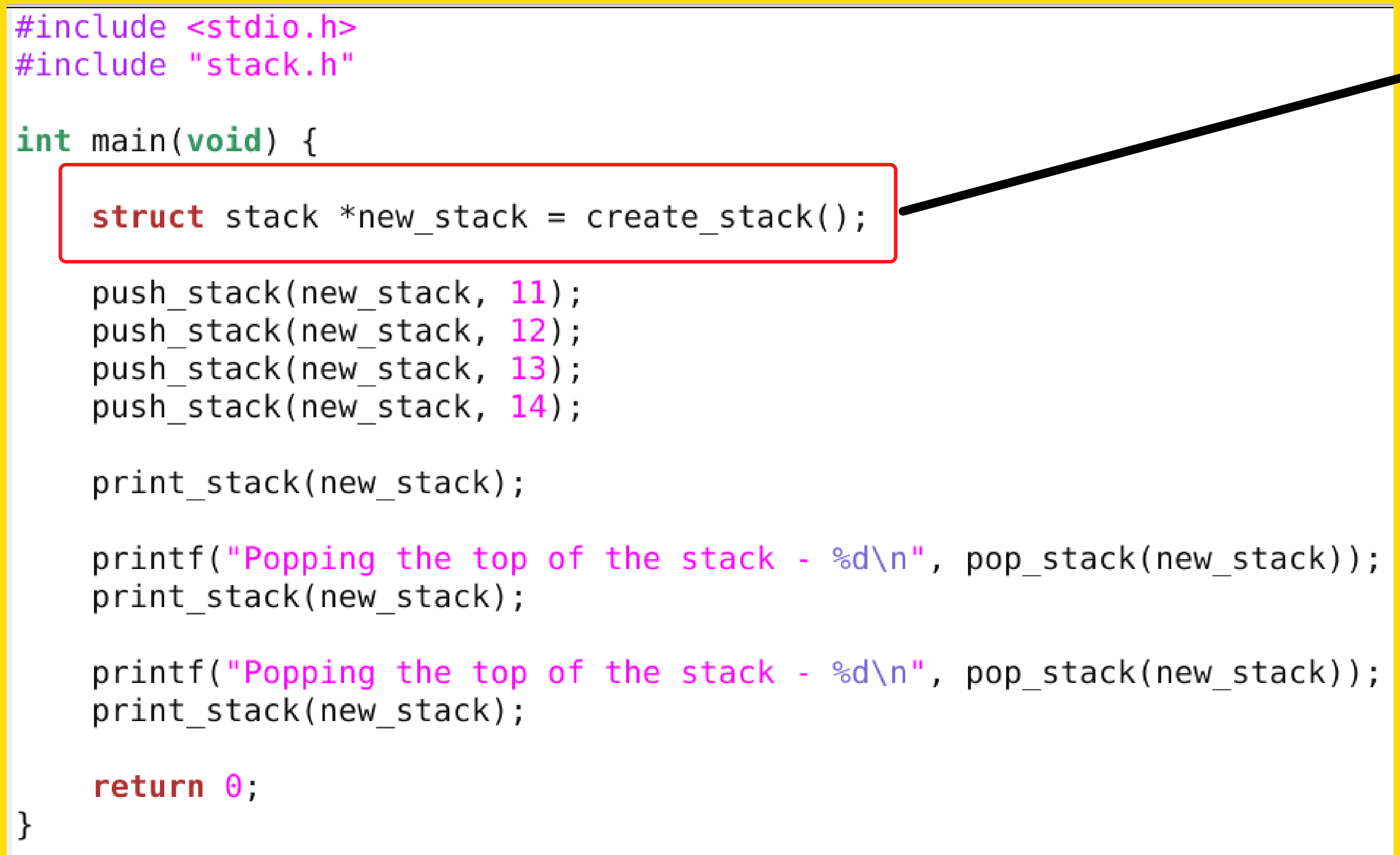


```c
// Define the stack structure itself, the stack structure in this case will
// have a size and a top node (which is the head)
struct stack {
    struct node *top;
    int size;
};
// Define each element of a stack as a node
struct node {
    int data;
    struct node *next;
};
```

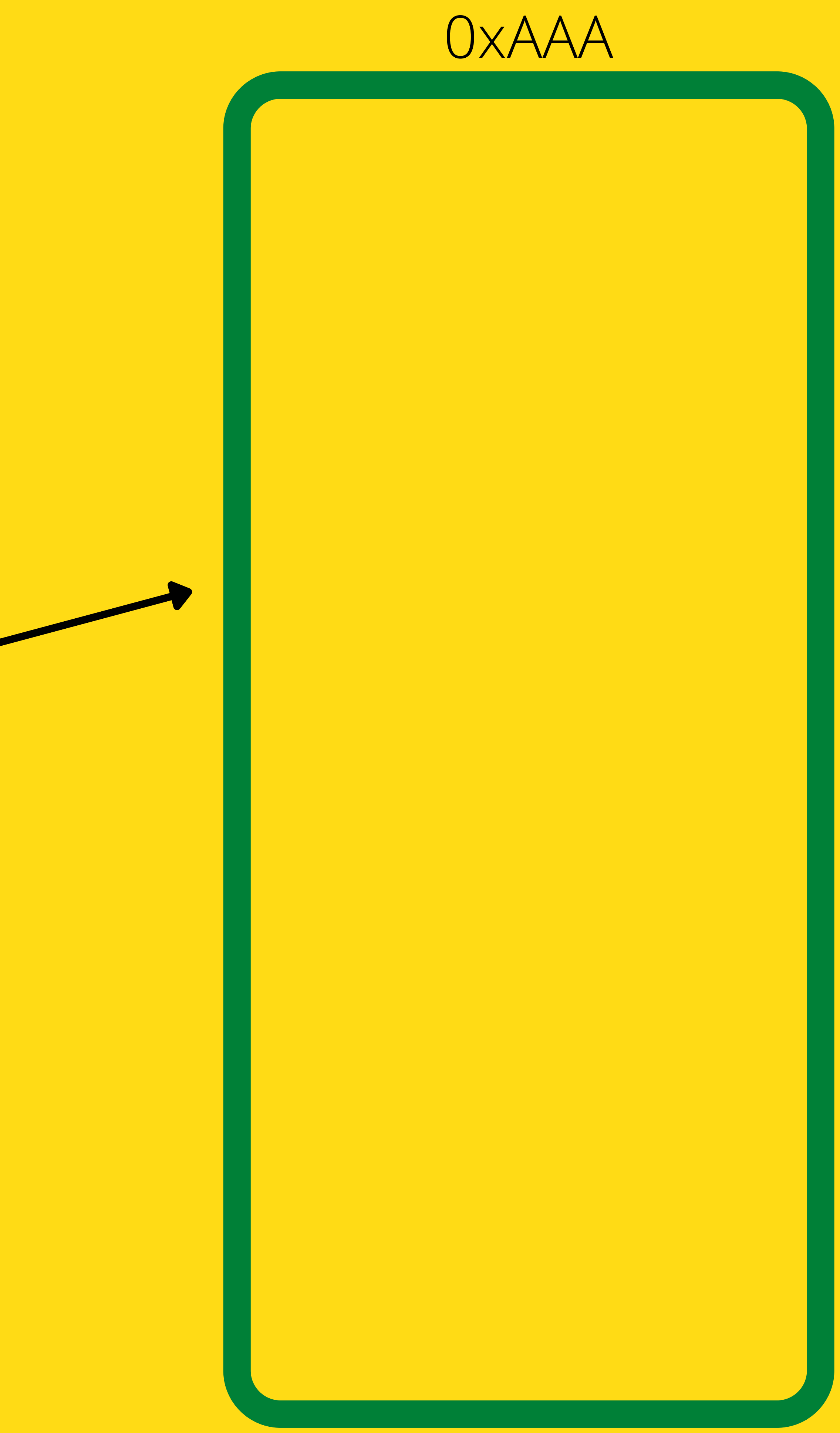
stack_list.c

LET'S DO A LINKED LIST IMPLEMENTATION FIRST

STACK



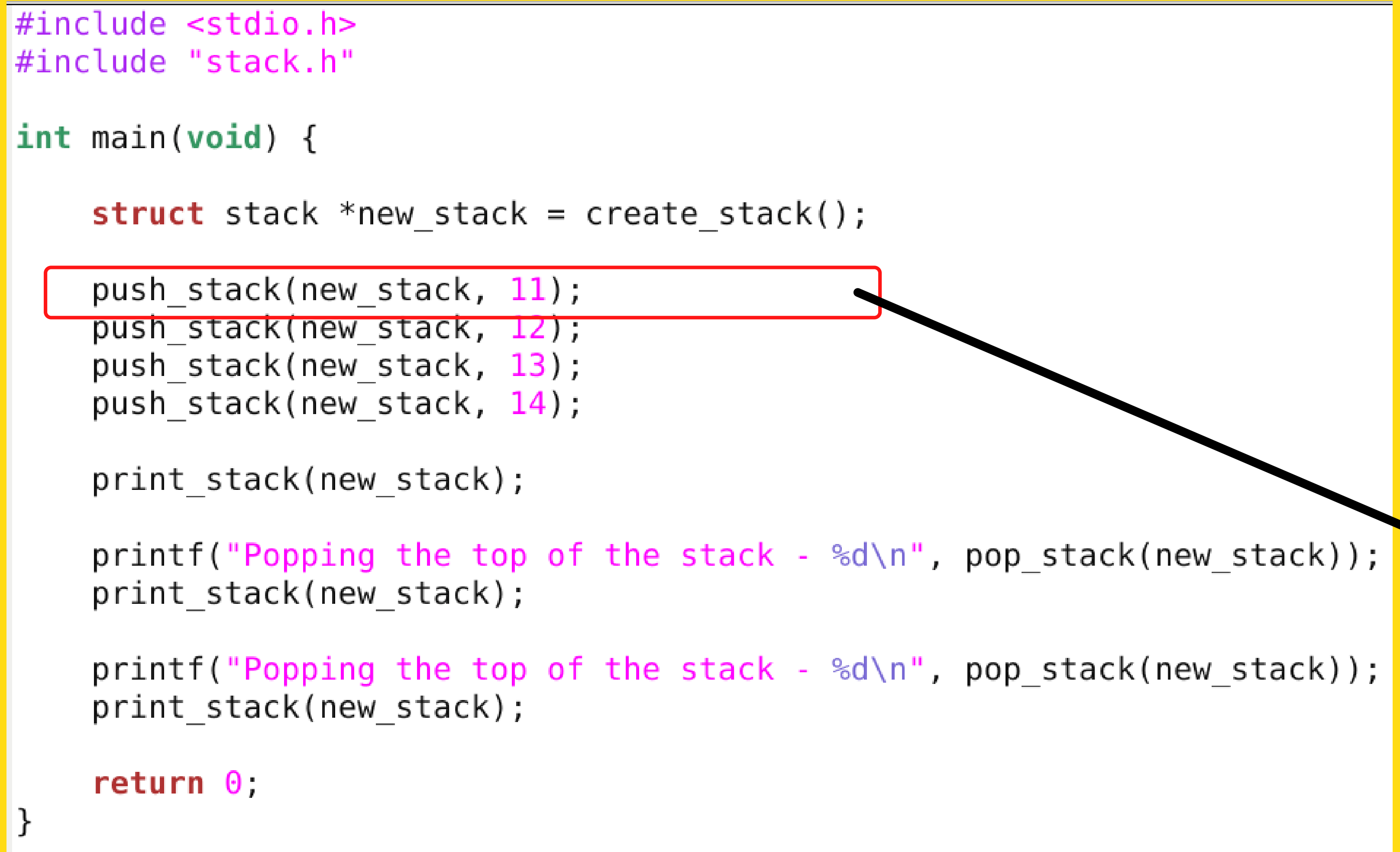
```
#include <stdio.h>
#include "stack.h"

int main(void) {

    struct stack *new_stack = create_stack();

    push_stack(new_stack, 11);
    push_stack(new_stack, 12);
    push_stack(new_stack, 13);
    push_stack(new_stack, 14);

    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    return 0;
}
```



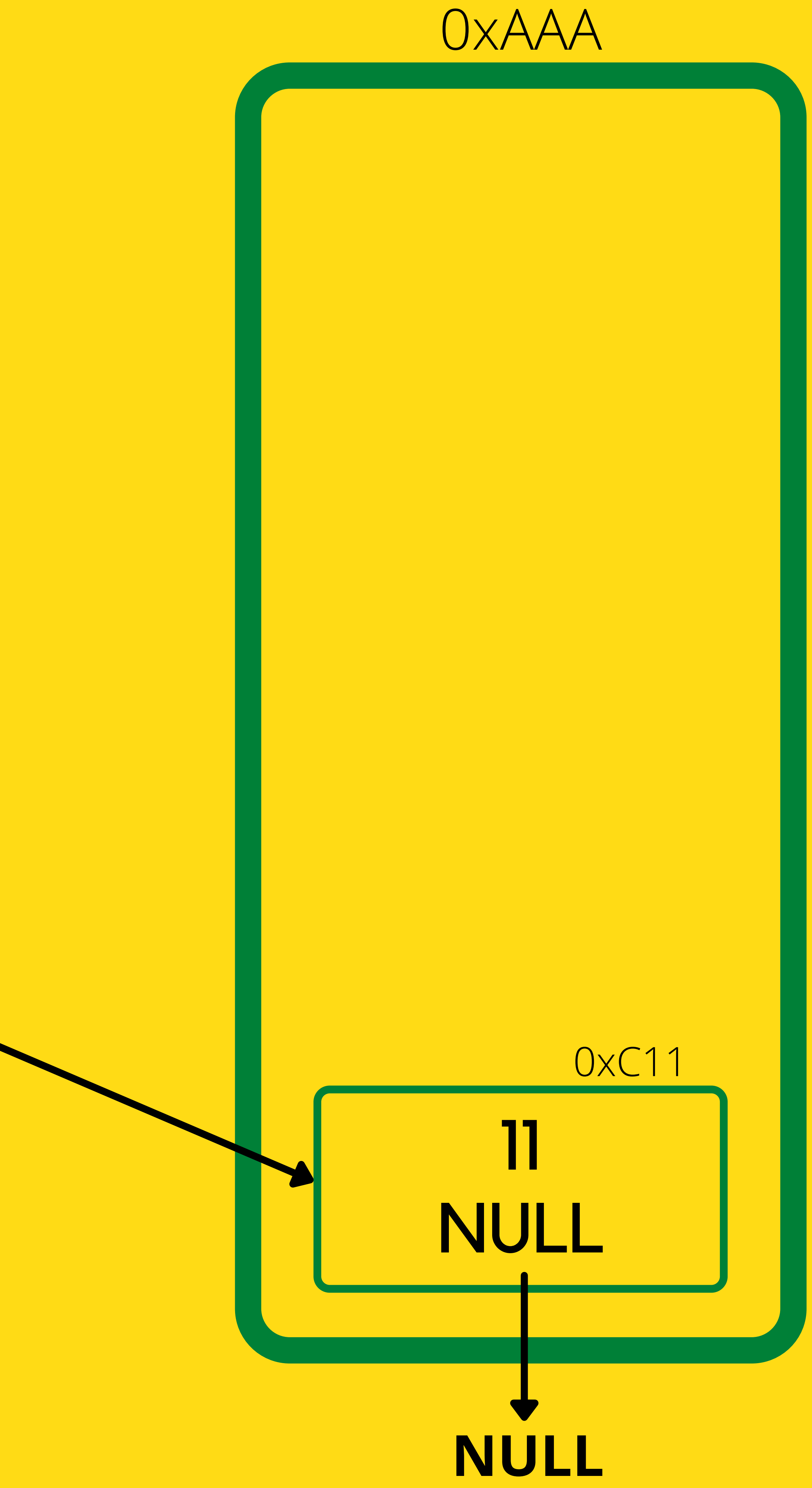
stack_list.c

LET'S DO A LINKED LIST IMPLEMENTATION FIRST

STACK



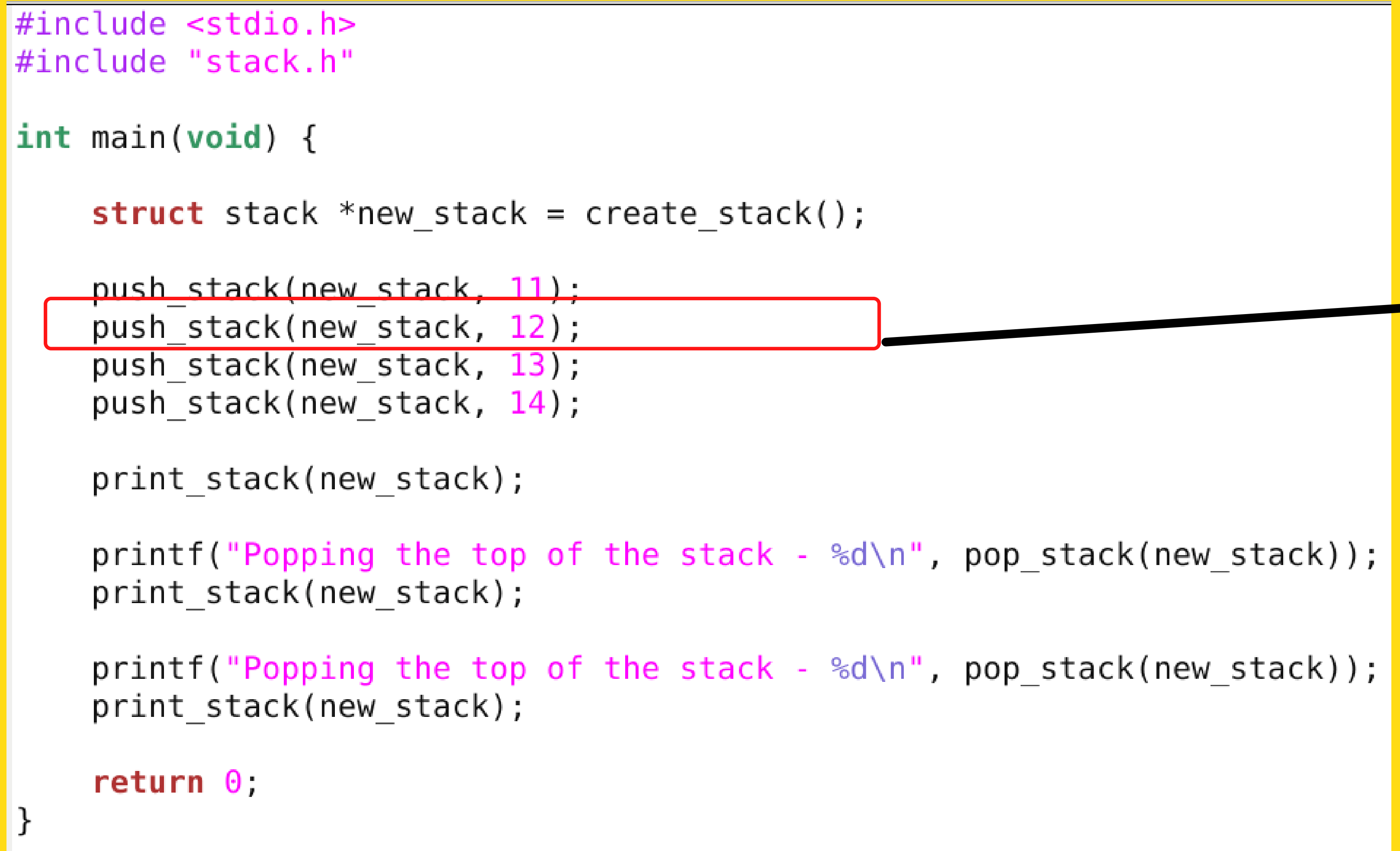
```
#include <stdio.h>
#include "stack.h"

int main(void) {

    struct stack *new_stack = create_stack();

    push_stack(new_stack, 11);
    push_stack(new_stack, 12);
    push_stack(new_stack, 13);
    push_stack(new_stack, 14);

    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    return 0;
}
```



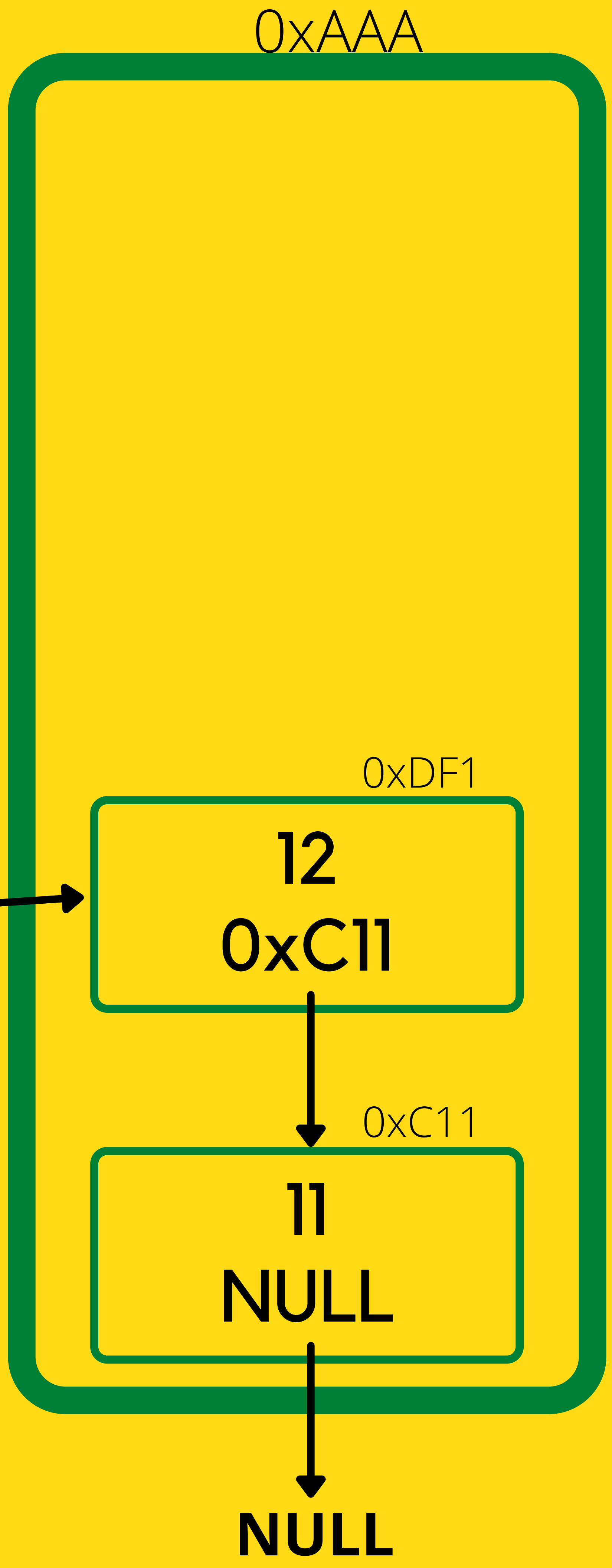
stack_list.c

LET'S DO A LINKED LIST IMPLEMENTATION FIRST

STACK



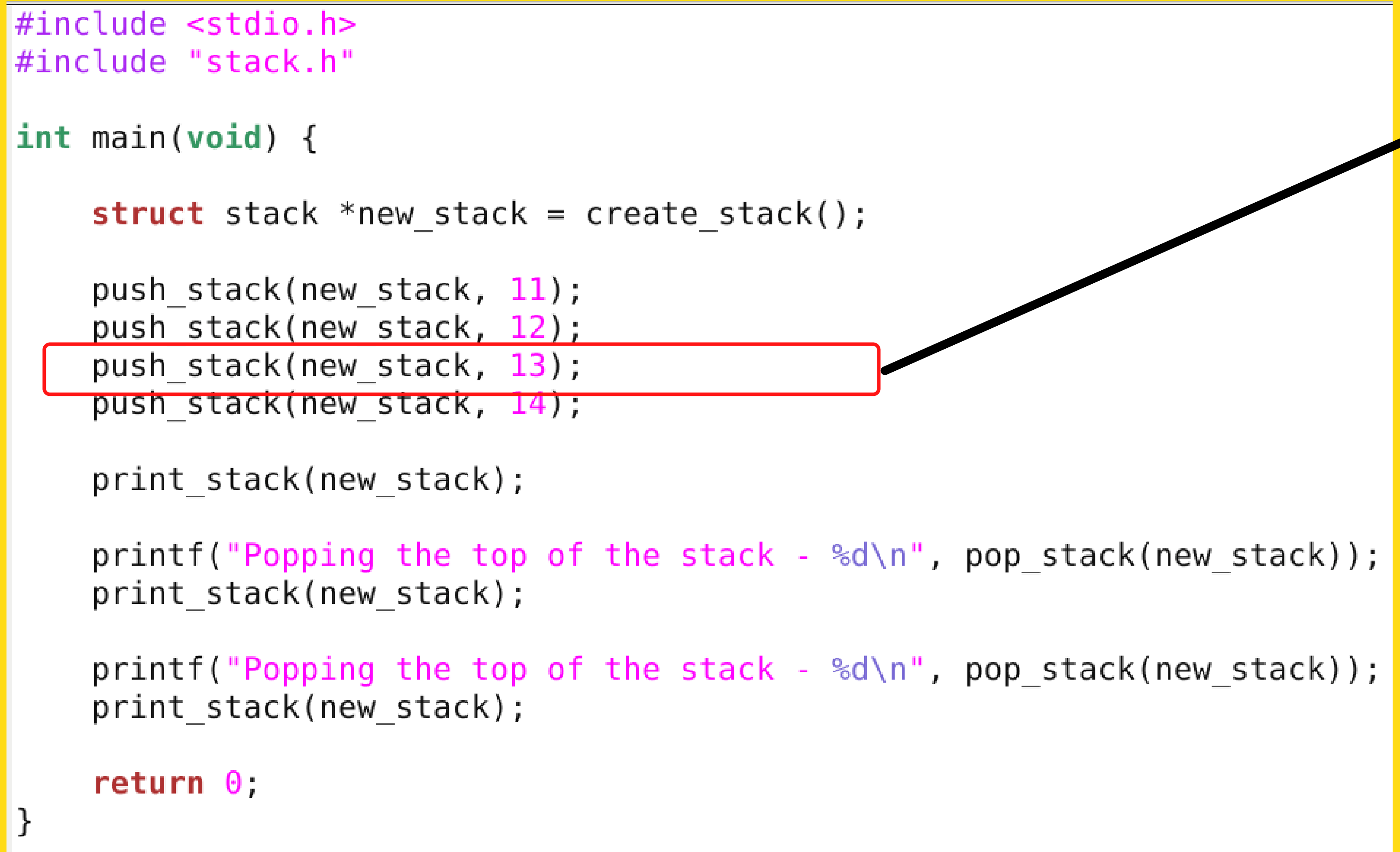
```
#include <stdio.h>
#include "stack.h"

int main(void) {

    struct stack *new_stack = create_stack();

    push_stack(new_stack, 11);
    push_stack(new_stack, 12);
    push_stack(new_stack, 13);
    push_stack(new_stack, 14);

    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    return 0;
}
```



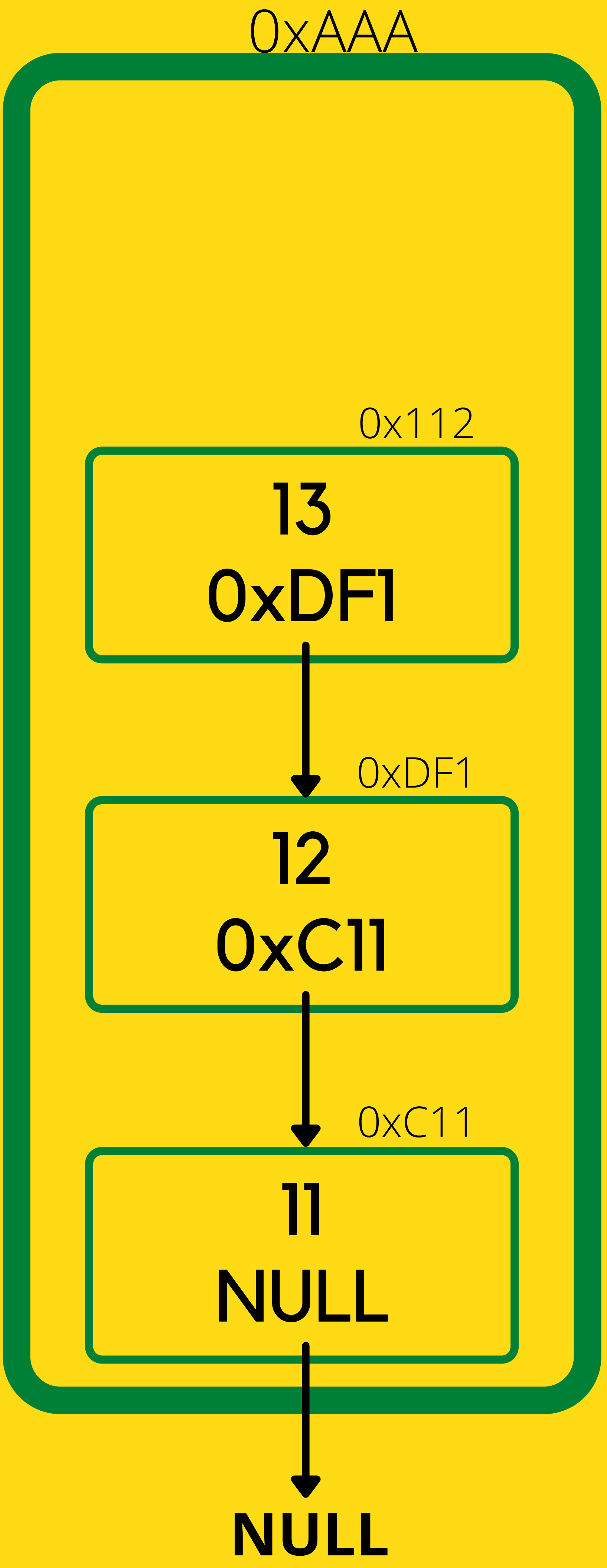
stack_list.c

LET'S DO A LINKED LIST IMPLEMENTATION FIRST

STACK



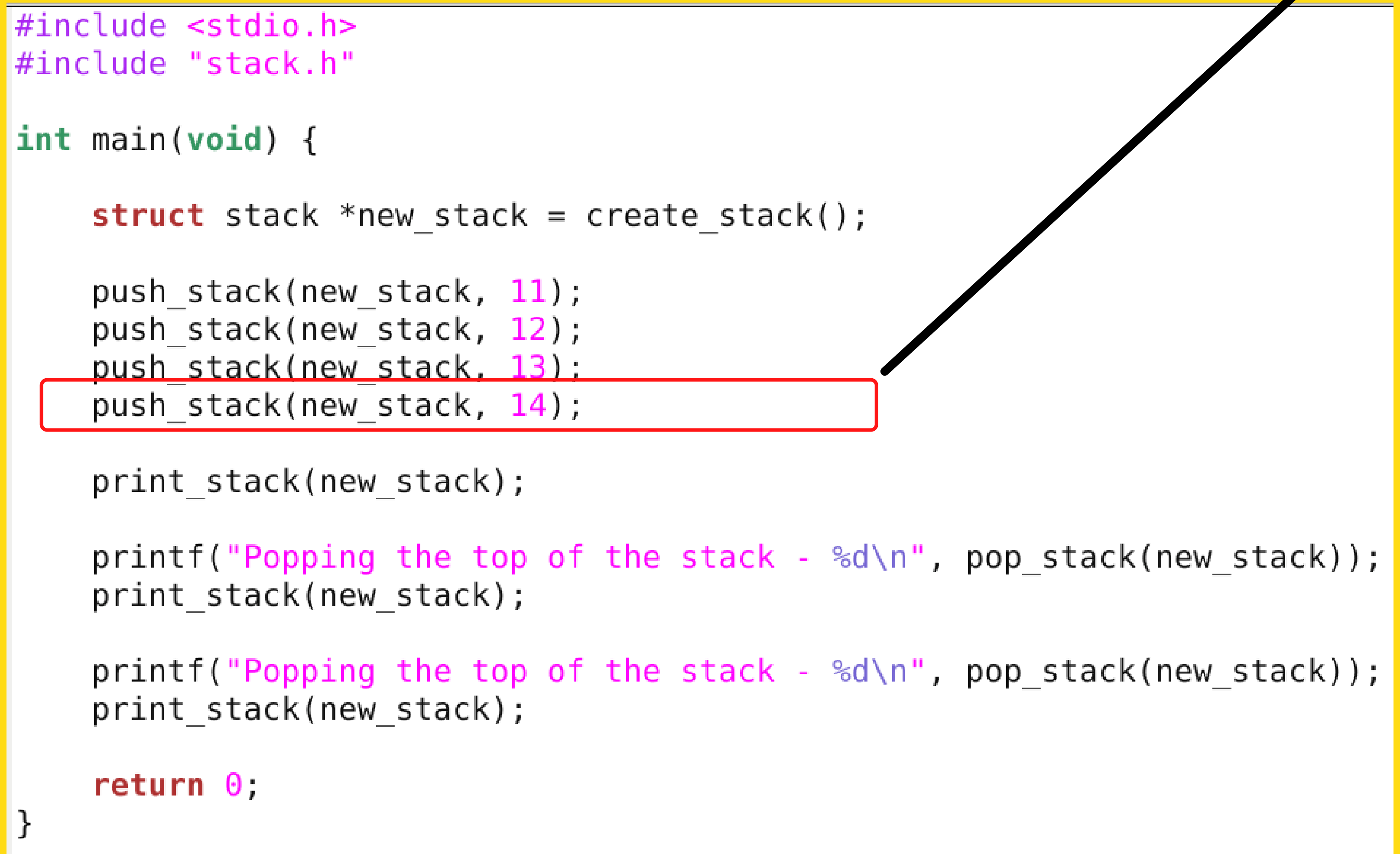
```c
#include <stdio.h>
#include "stack.h"

int main(void) {

    struct stack *new_stack = create_stack();

    push_stack(new_stack, 11);
    push_stack(new_stack, 12);
    push_stack(new_stack, 13);
    push_stack(new_stack, 14);

    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    return 0;
}
```



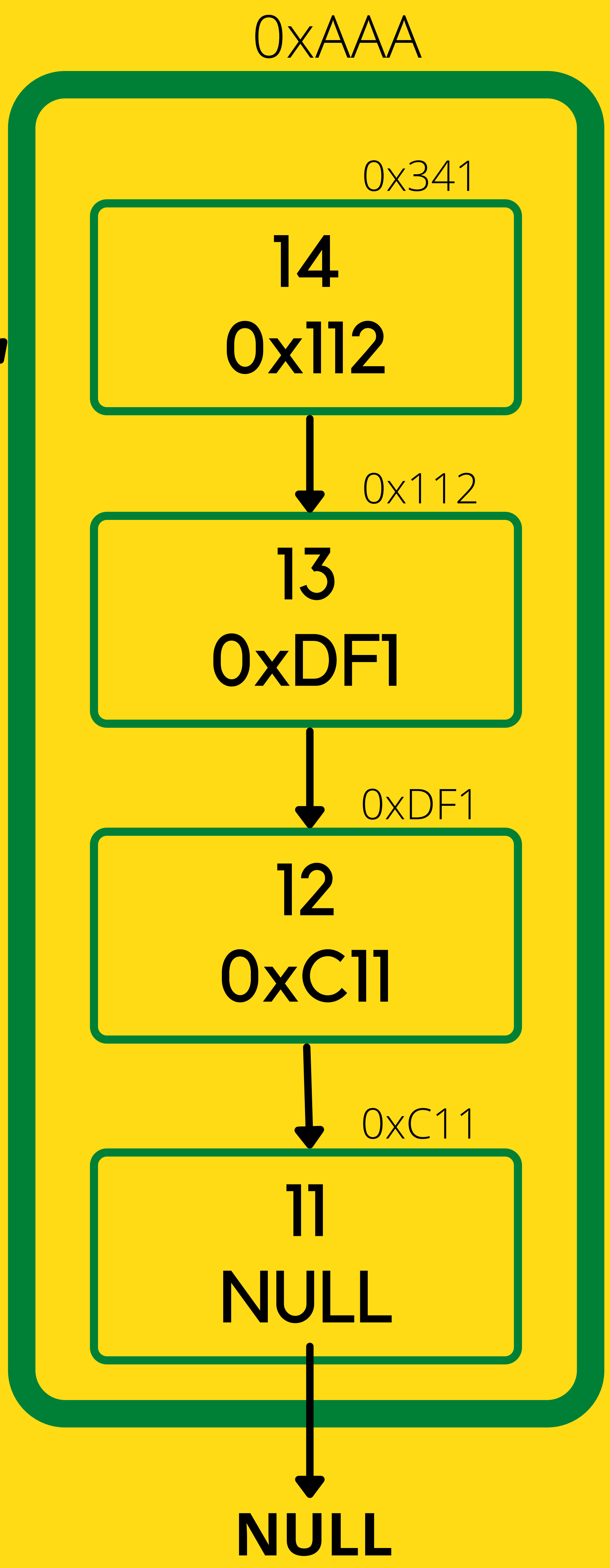
stack_list.c

LET'S DO A LINKED LIST IMPLEMENTATION FIRST

STACK



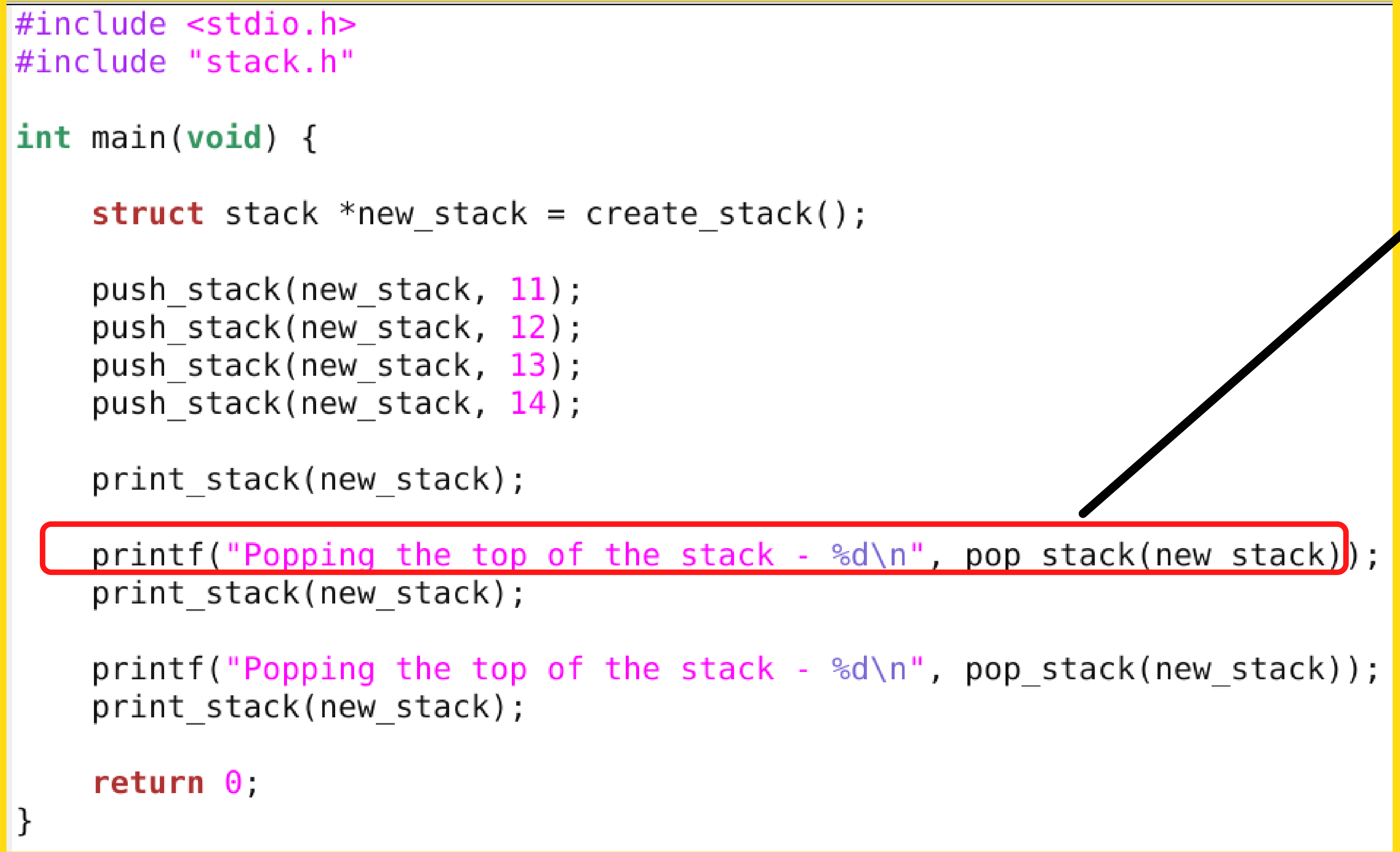
```
#include <stdio.h>
#include "stack.h"

int main(void) {

    struct stack *new_stack = create_stack();

    push_stack(new_stack, 11);
    push_stack(new_stack, 12);
    push_stack(new_stack, 13);
    push_stack(new_stack, 14);

    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    return 0;
}
```



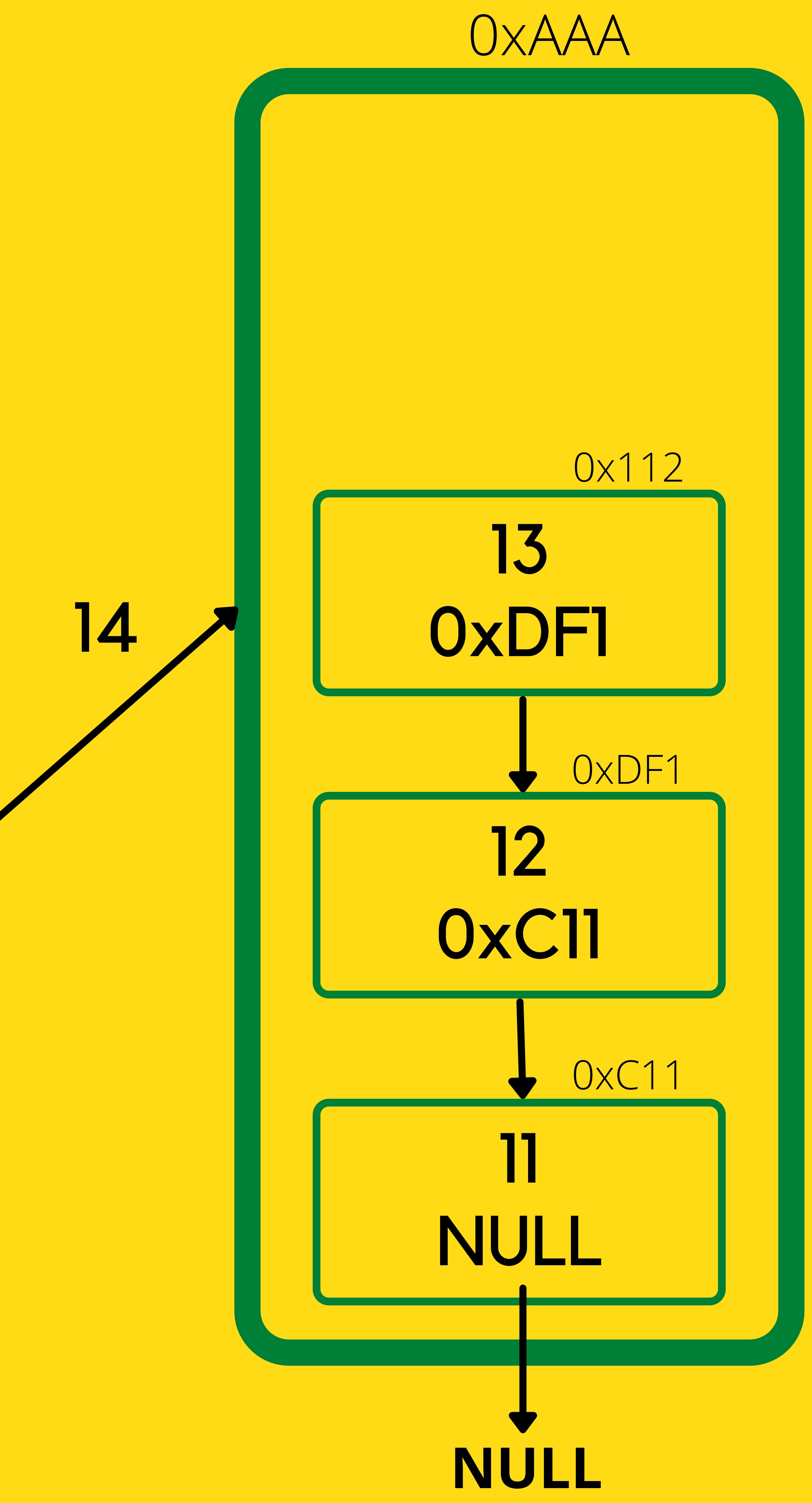
stack_list.c

LET'S DO A LINKED LIST IMPLEMENTATION FIRST

STACK



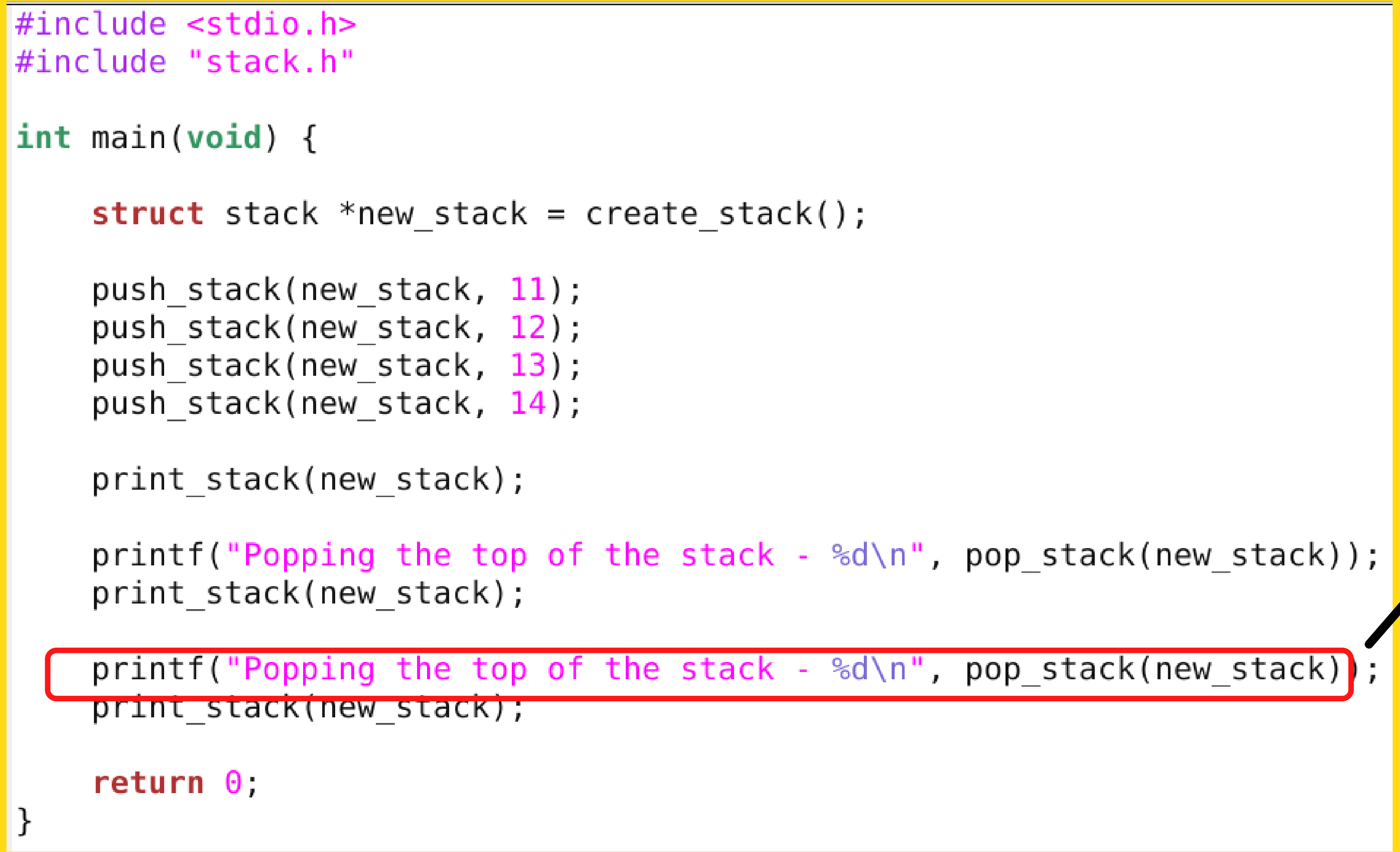
```c
#include <stdio.h>
#include "stack.h"

int main(void) {

    struct stack *new_stack = create_stack();

    push_stack(new_stack, 11);
    push_stack(new_stack, 12);
    push_stack(new_stack, 13);
    push_stack(new_stack, 14);

    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    return 0;
}
```



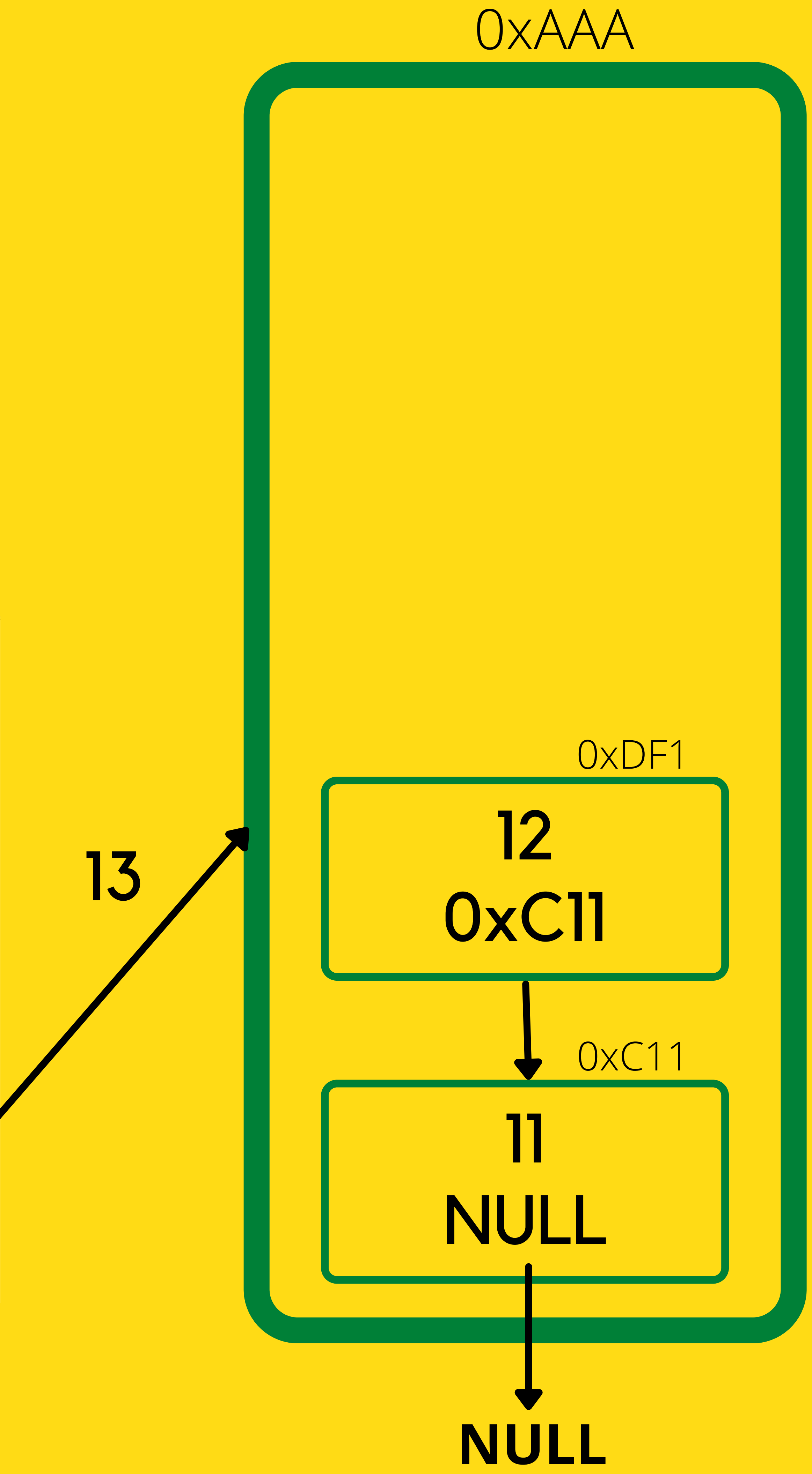
stack_list.c

LET'S DO A LINKED LIST IMPLEMENTATION FIRST

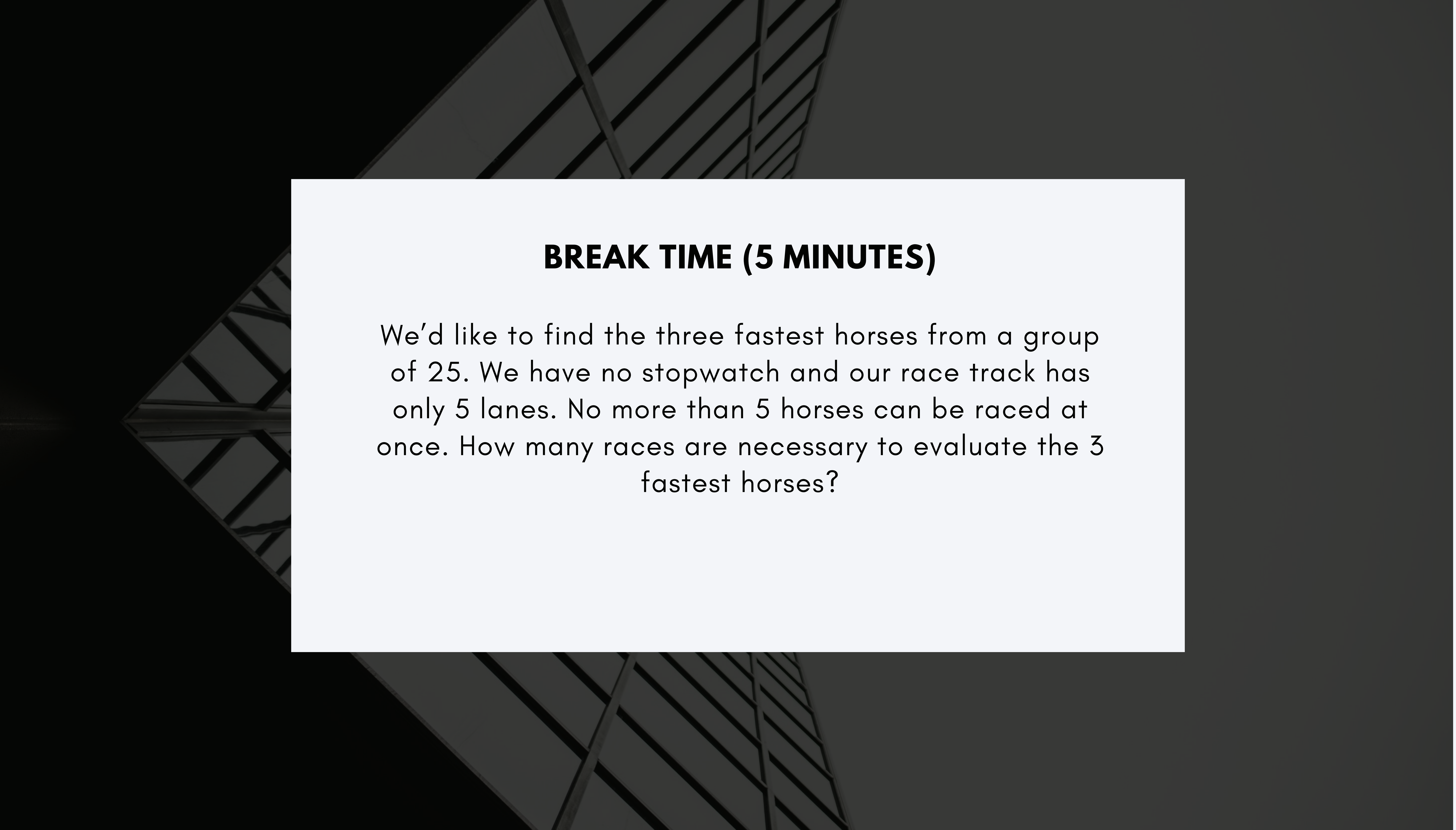
STACK



```
#include <stdio.h>
#include "stack.h"

int main(void) {

    struct stack *new_stack = create_stack();

    push_stack(new_stack, 11);
    push_stack(new_stack, 12);
    push_stack(new_stack, 13);
    push_stack(new_stack, 14);

    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    return 0;
}
```



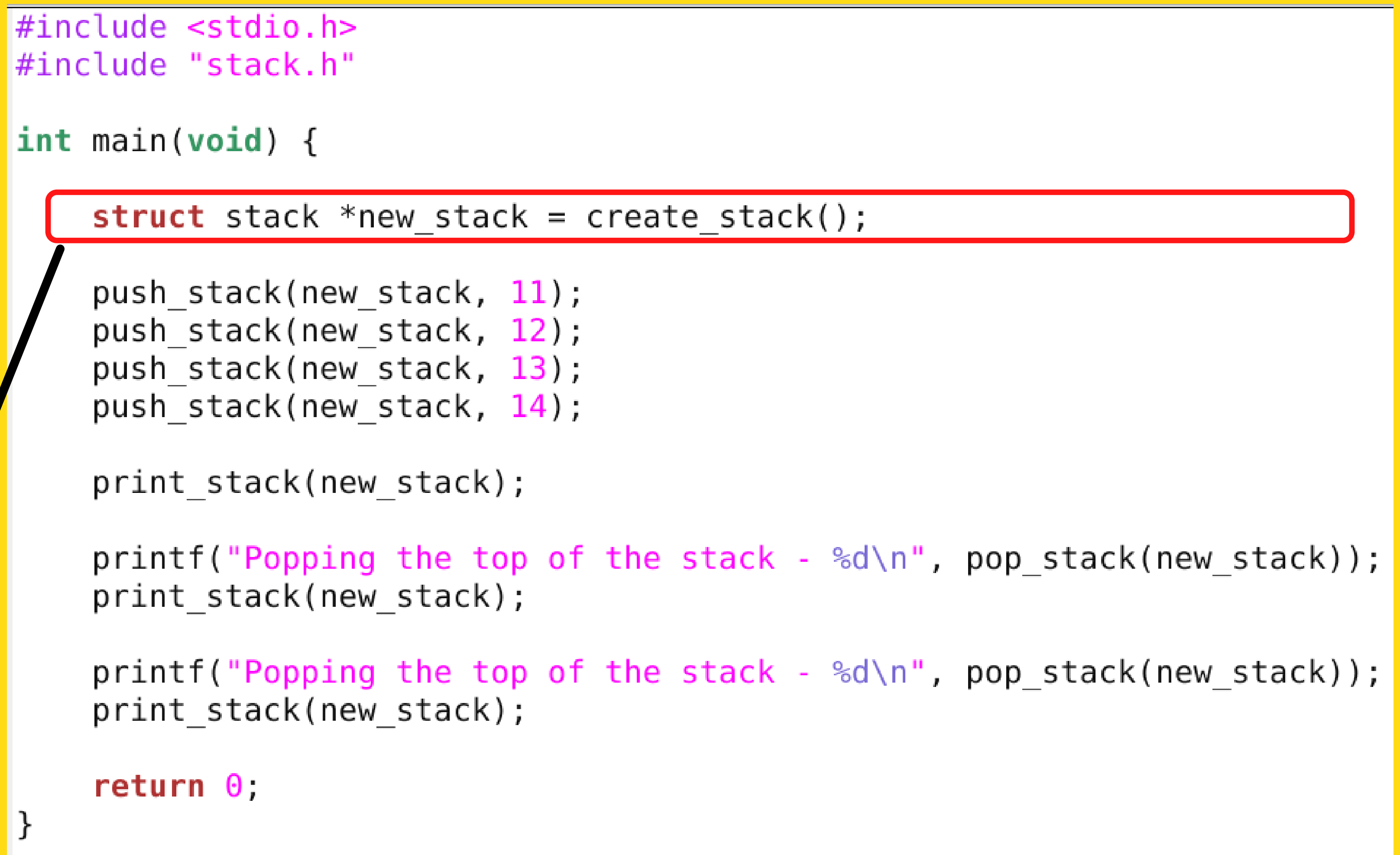
stack_list.c



BREAK TIME (5 MINUTES)

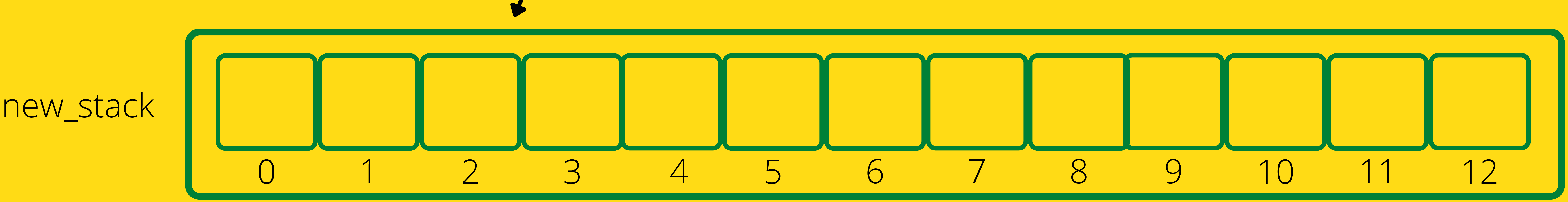
We'd like to find the three fastest horses from a group of 25. We have no stopwatch and our race track has only 5 lanes. No more than 5 horses can be raced at once. How many races are necessary to evaluate the 3 fastest horses?

WHAT ABOUT AN ARRAY IMPLEMENTATION?



```
#include <stdio.h>
#include "stack.h"

int main(void) {

    struct stack *new_stack = create_stack();

    push_stack(new_stack, 11);
    push_stack(new_stack, 12);
    push_stack(new_stack, 13);
    push_stack(new_stack, 14);

    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    return 0;
}
```

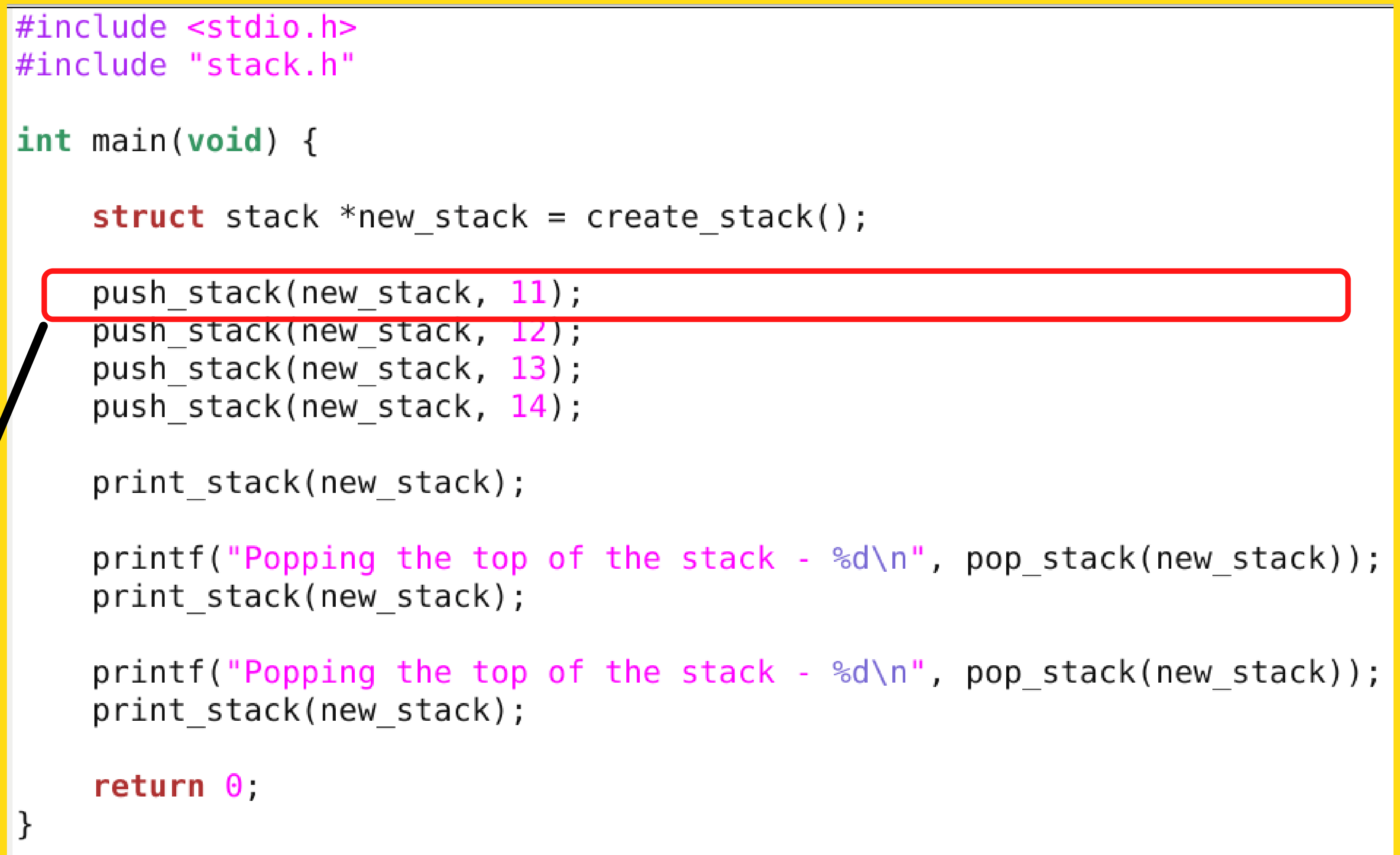
STACK



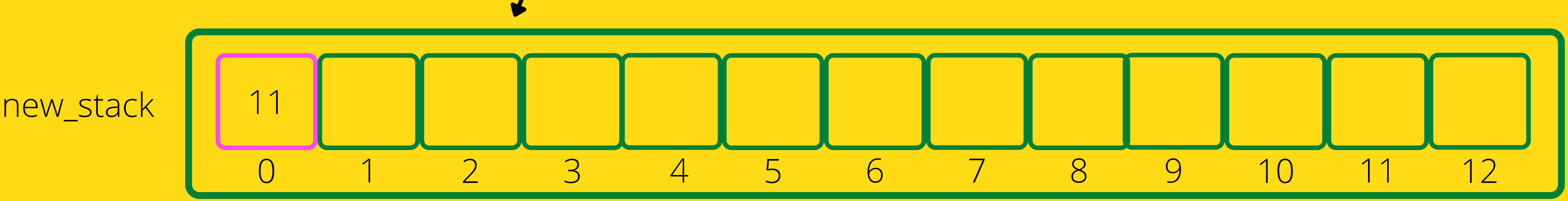
stack_array.c

WHAT ABOUT AN ARRAY IMPLEMENTATION?

STACK



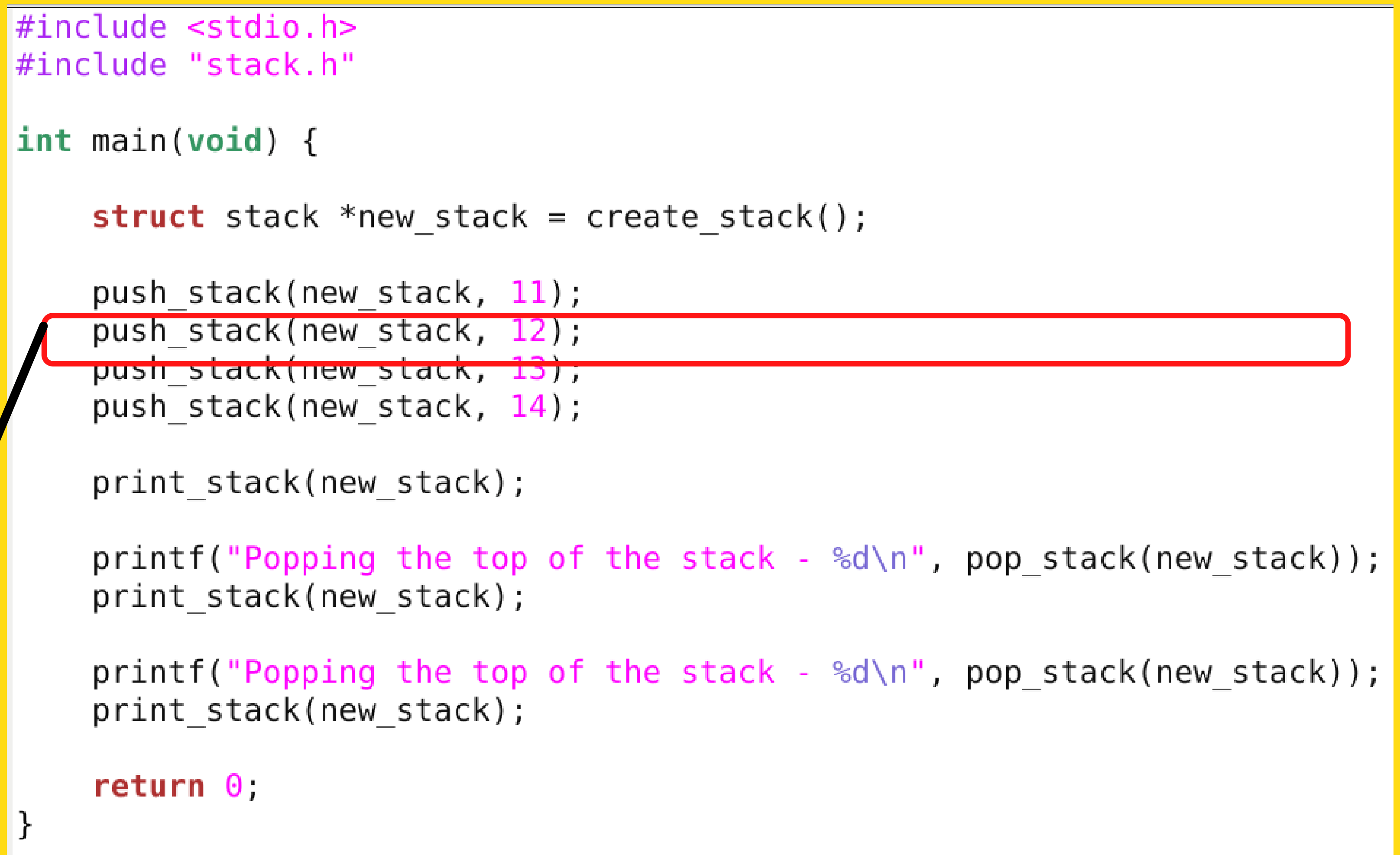
```c
#include <stdio.h>
#include "stack.h"

int main(void) {

    struct stack *new_stack = create_stack();

    push_stack(new_stack, 11);
    push_stack(new_stack, 12);
    push_stack(new_stack, 13);
    push_stack(new_stack, 14);

    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    return 0;
}
```



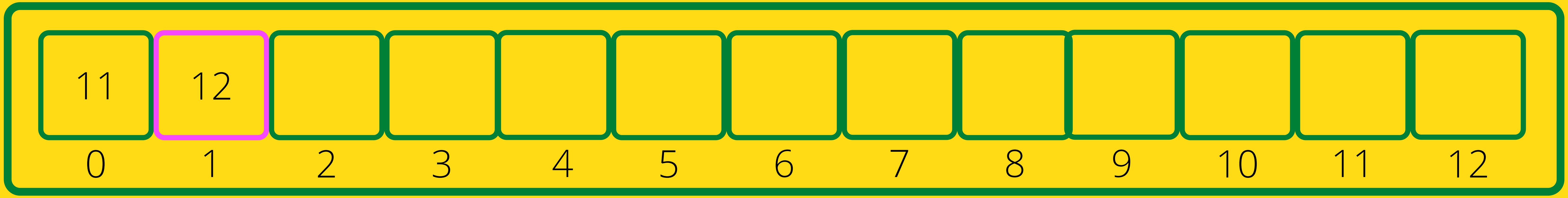
stack_array.c

WHAT ABOUT AN ARRAY IMPLEMENTATION?

STACK



```
#include <stdio.h>
#include "stack.h"

int main(void) {

    struct stack *new_stack = create_stack();

    push_stack(new_stack, 11);
    push_stack(new_stack, 12);
    push_stack(new_stack, 13);
    push_stack(new_stack, 14);

    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    return 0;
}
```

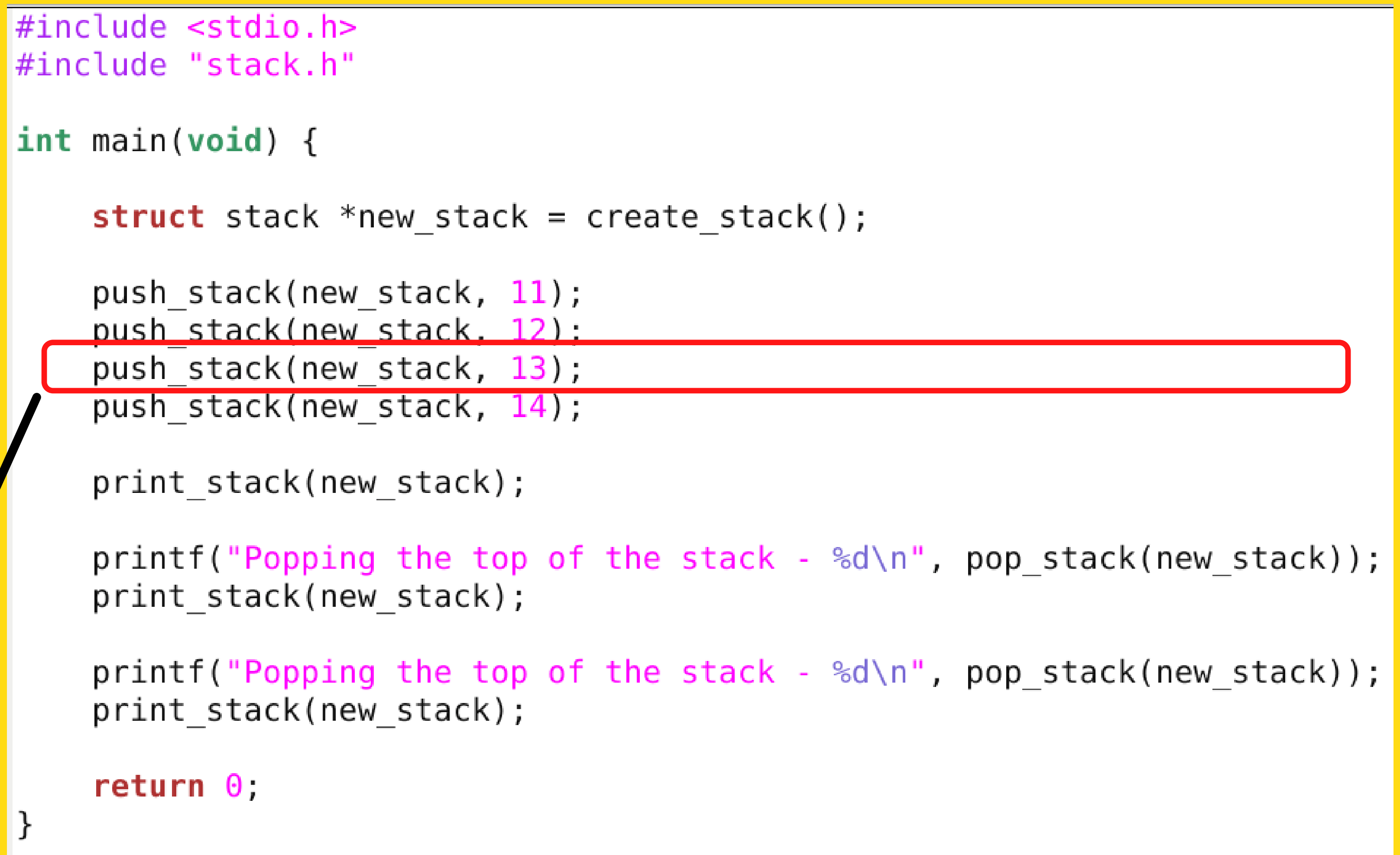


new_stack	11	12											
	0	1	2	3	4	5	6	7	8	9	10	11	12

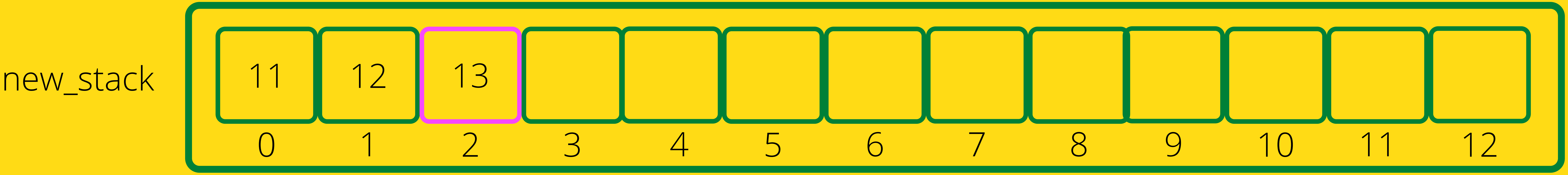
stack_array.c

WHAT ABOUT AN ARRAY IMPLEMENTATION?

STACK



```
#include <stdio.h>
#include "stack.h"

int main(void) {

    struct stack *new_stack = create_stack();

    push_stack(new_stack, 11);
    push_stack(new_stack, 12);
    push_stack(new_stack, 13);
    push_stack(new_stack, 14);

    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    return 0;
}
```

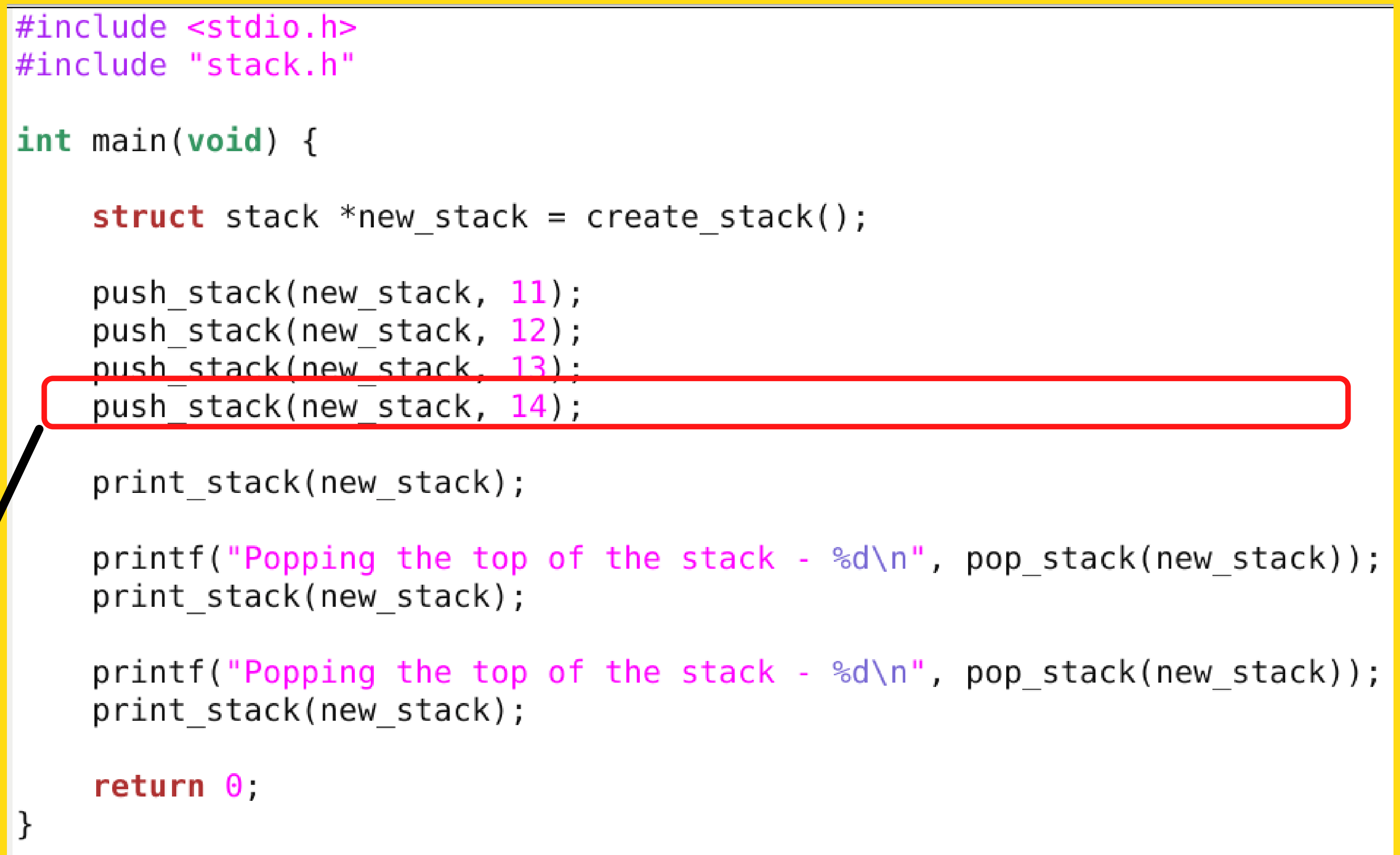


new_stack	11	12	13										
	0	1	2	3	4	5	6	7	8	9	10	11	12

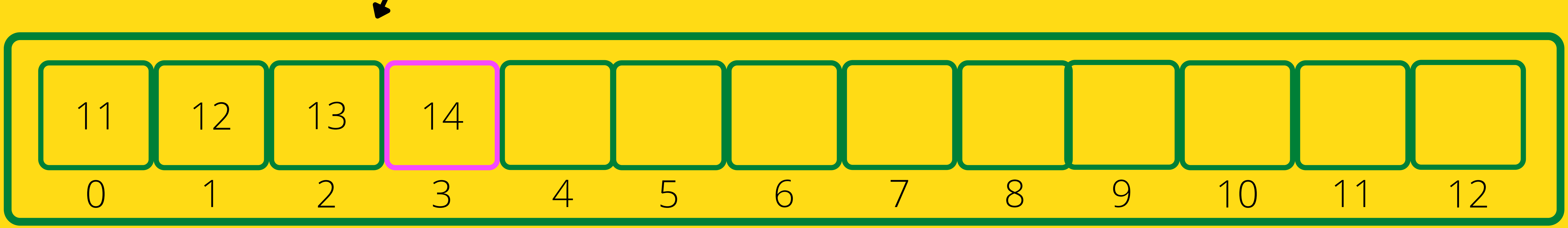
stack_array.c

WHAT ABOUT AN ARRAY IMPLEMENTATION?

STACK



```c
#include <stdio.h>
#include "stack.h"

int main(void) {

    struct stack *new_stack = create_stack();

    push_stack(new_stack, 11);
    push_stack(new_stack, 12);
    push_stack(new_stack, 13);
    push_stack(new_stack, 14);

    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    return 0;
}
```

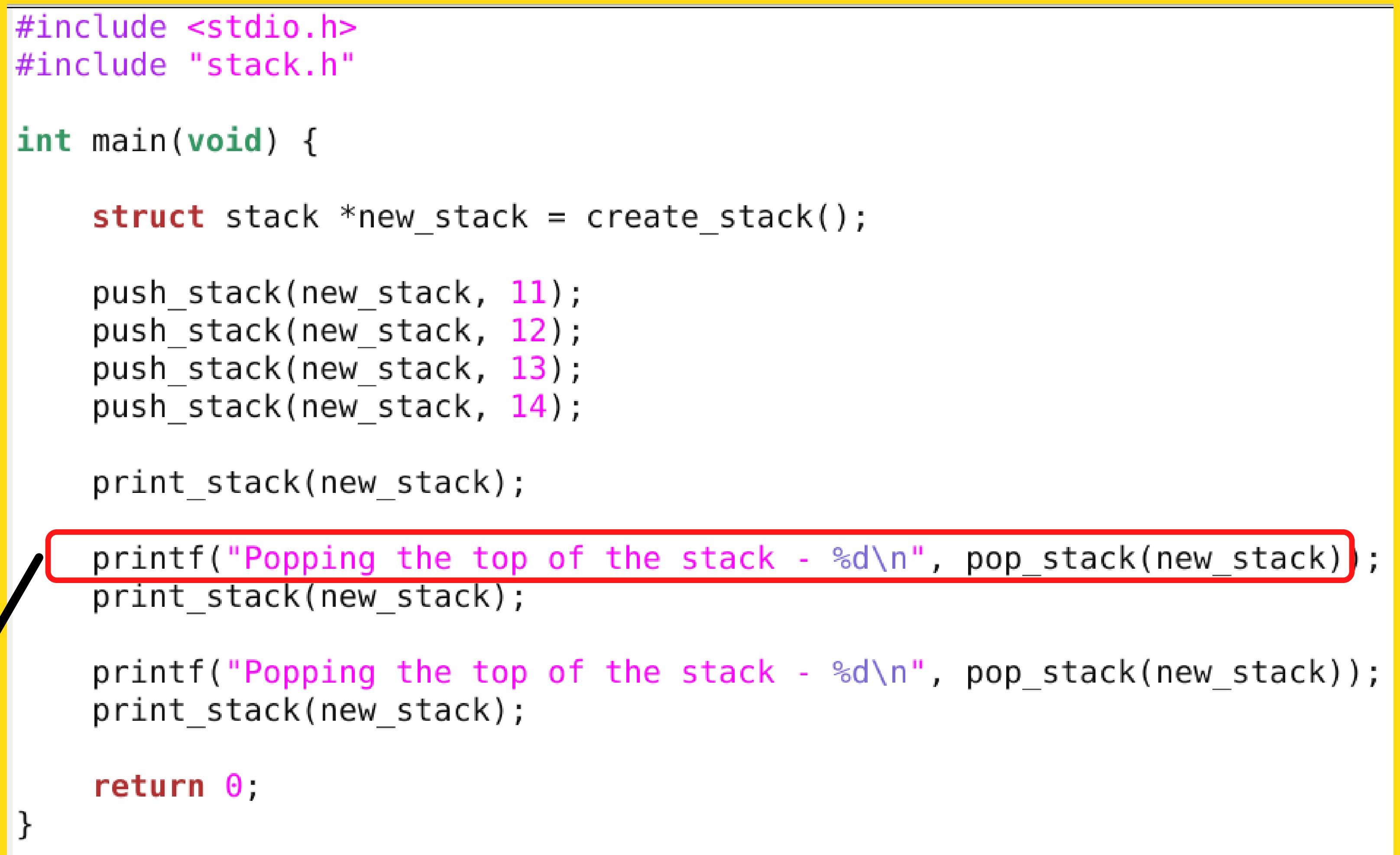


new_stack

11	12	13	14									
0	1	2	3	4	5	6	7	8	9	10	11	12

stack_array.c

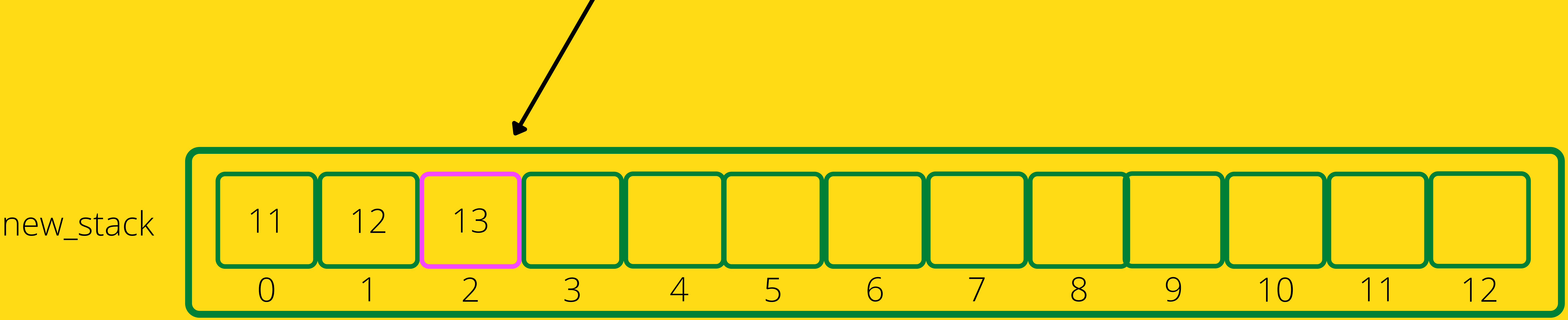
WHAT ABOUT AN ARRAY IMPLEMENTATION?



```
#include <stdio.h>
#include "stack.h"

int main(void) {

    struct stack *new_stack = create_stack();

    push_stack(new_stack, 11);
    push_stack(new_stack, 12);
    push_stack(new_stack, 13);
    push_stack(new_stack, 14);

    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    return 0;
}
```

STACK

14

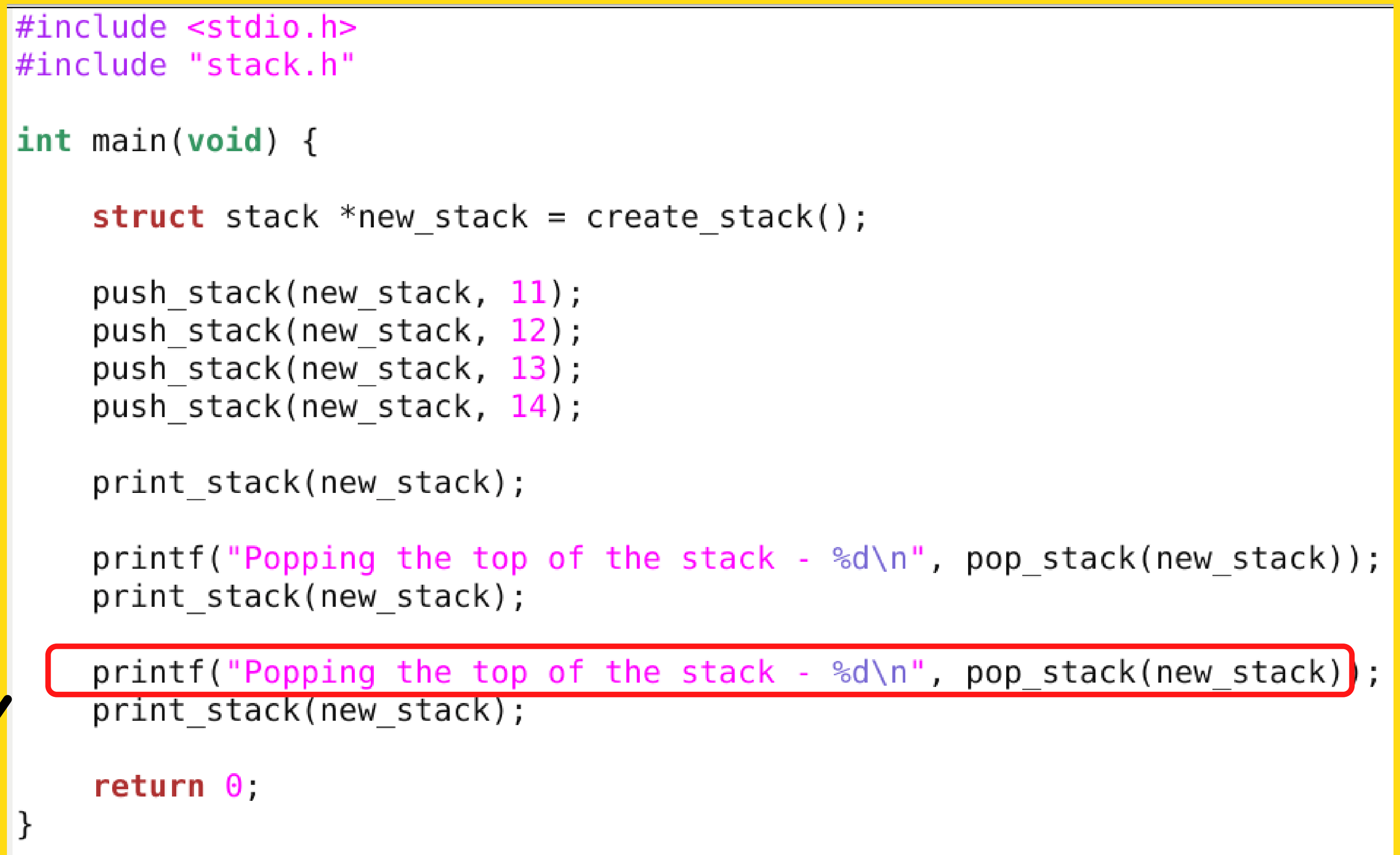


new_stack

0	1	2	3	4	5	6	7	8	9	10	11	12
11	12	13										

stack_array.c

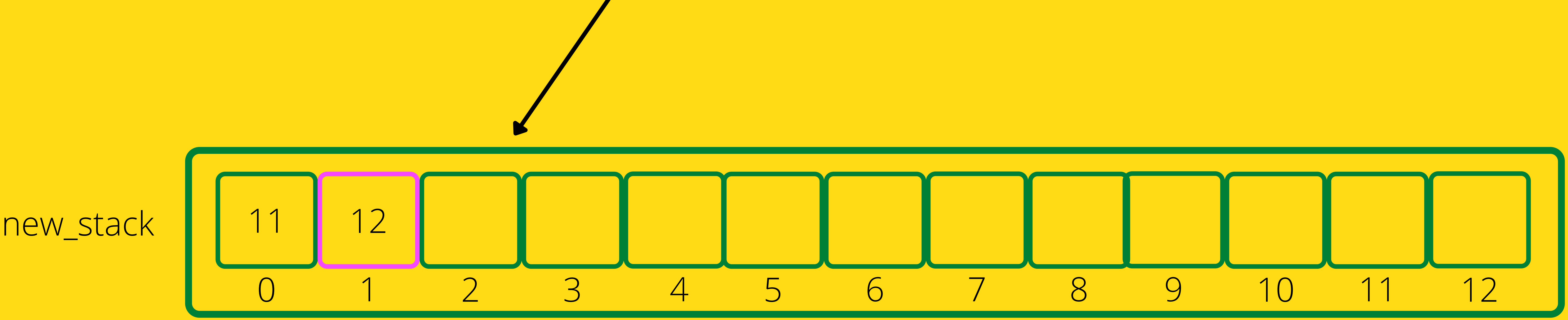
WHAT ABOUT AN ARRAY IMPLEMENTATION?



```
#include <stdio.h>
#include "stack.h"

int main(void) {

    struct stack *new_stack = create_stack();

    push_stack(new_stack, 11);
    push_stack(new_stack, 12);
    push_stack(new_stack, 13);
    push_stack(new_stack, 14);

    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    return 0;
}
```

STACK

13



stack_array.c

OTHER ABSTRACT DATA TYPES

QUEUES

- There other abstract data types,
  - one that works in the opposite way to a Stack is a Queue
- A queue works just like a physical queue at the shops (or when you line up to get some great tickets for a music festival)
- So a Queue operates on First In, First Out principle – if you get in a queue first, you will be served first...
- To get into the queue, you enqueue, and to get out of the queue, dequeue.
- There are of course other possibilities for abstract data types!

FEEDBACK?

PLEASE LET ME KNOW ANY FEEDBACK FROM TODAY'S LECTURE!

www.menti.com

Code: 6391 0195

WHAT DID WE LEARN TODAY?

LINKED LIST: LINKED LIST IN A STRUCT

letter_number.c

ABSTRACT DATA TYPES: STACK

stack.h

stack_list.c

stack_array.c

ANY QUESTIONS?

DON'T FORGET YOU CAN ALWAYS EMAIL US ON CS1511@CSE.UNSW.EDU.AU FOR ANY ADMIN QUESTIONS

PLEASE ASK IN THE FORUM FOR CONTENT RELATED QUESTIONS