COMP1511 Week 8

multifile projects, struct pointers and linked lists

Joanna Lin

What we'll cover

- **multifile projects**
  - header files (.h)
  - implementation files (.c)
- **linked lists**
  - memory allocation and pointers
  - storing struct pointers inside structs
  - linked list data structure
  - representing linked lists with struct pointers

Multifile Projects

Multifile Projects

Structure

- main.c
  - contains the **`main`** function
- .h files (header files)
  - contains function prototype and function comments.
  - **`#include`**'d in the file that needs it (e.g. **`#include "my_file.h"`**)
  - Note double quotes are used for your own header files, whereas angular brackets are used for C standard library header files. (e.g. **`#include <stdio.h>`** vs **`#include "my_file.h"`**)
- .c files
  - contains function definition (the implementation of the functions)
  - compiled alongside main.c (e.g. **`dcc my_file.c main.c -o program`**)

Multifile Projects

What it looks like

letters.h

```c
#ifndef _LETTERS_H_
#define _LETTERS_H_
// Checks if the character passed in is a letter.
//
// `check_letter` will be passed in:
// - `ch` -- the character to check
//
// `check_letter` will return 1 if `ch` is a letter and 0 otherwise.
int check_letter(char ch);
#endif
```

notice how precisely the function is described

main.c

```c
#include <stdio.h>

int check_letter(char ch);

int main(void) {

    char input;
    printf("Enter letter: ");
    while (scanf(" %c", &input) == 1) {
        if (check_letter(input) == 1) {
            printf("It's a letter!\n");
        } else {
            printf("It's not a letter!\n");
        }

        printf("Enter letter: ");
    }

    return 0;
}

int check_letter(char ch) {
    return ('a' <= ch && ch <= 'z') || ('A' <= ch && ch <= 'Z');
}
```

```c
#include <stdio.h>

#include "letters.h"

int main(void) {

    char input;
    printf("Enter letter: ");
    while (scanf(" %c", &input) == 1) {
        if (check_letter(input) == 1) {
            printf("It's a letter!\n");
        } else {
            printf("It's not a letter!\n");
        }

        printf("Enter letter: ");
    }

    return 0;
}
```

We now require a **`#include`**

letters.c

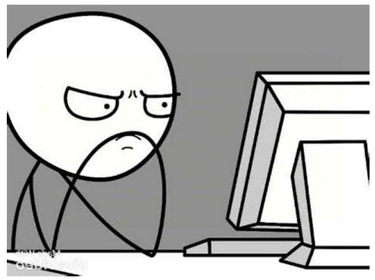
```c
#include "letters.h"
// Function that checks whether input is an alphabetic letter
int check_letter(char ch) {
    return ('a' <= ch && ch <= 'z') || ('A' <= ch && ch <= 'Z');
}
```

Multifile Projects

Benefits

- spread code across multiple files to make navigating our code easier
- make collaborating on projects easier.

Programmer X



Programmer X works on:

.c file

Programmer X can implement functions knowing how it is meant to behave.

Source of truth

.h file

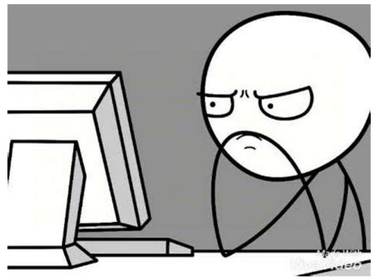
Programmer X and Y don't need to know how each other's code works to be able to write their own code!

Programmer Y works on:

main.c

Programmer Y can write the logic of the program using the functions declared in the .h file, based on their described behaviour.

Programmer Y



Struct Pointers

Struct Pointers

Recap

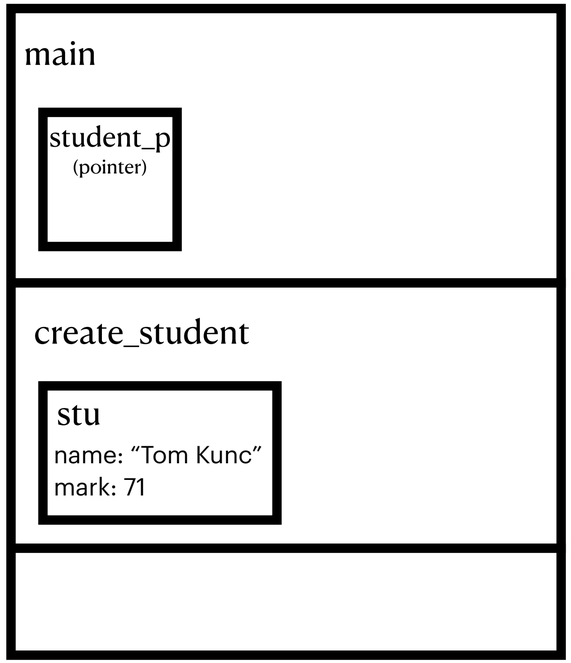
- Theoretically, they work like any other pointer.
  - we declare it by adding a `*` at the end of the struct type `struct student *student_p;`
  - struct pointers store memory addresses of structs.
  - we can dereference the pointer (change the struct at the memory address).
- C has syntactic sugar for accessing fields of the struct at the memory address.
  - Instead of `(*student_p).field_name`, we can write `student_p->field_name`.
- **Practically**, we create them quite differently from other pointers.
  - Initialising the fields of a struct is generally quite long and repetitive, so we often use functions as 'factories' to create a struct, initialise its fields and return a pointer to it.

Returning an Address: Bad

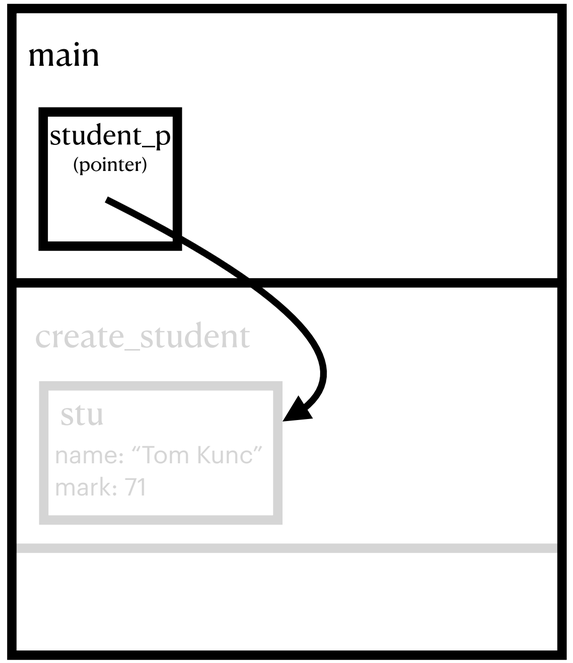
```c
#include <stdio.h>
#include <string.h>
#define MAX_LENGTH 50

struct student {
    char name[MAX_LENGTH];
    int mark;
};

struct student *create_student(char *name, int mark);

int main(void) {
    struct student *student_p = create_student("Tom Kunc", 71);
    printf("Name: %s, Mark: %d\n", student_p->name, student_p->mark);
    return 0;
}

struct student *create_student(char *name, int mark) {
    struct student stu;
    strcpy(stu.name, name);
    stu.mark = mark;
    return &stu;
}
```

Returning an Address

Problem



`create_student` creates a `struct student` called `stu` and initialises its fields.



when we exit `create_student`, its memory is deallocated and the address of `stu` is stored in `student_p`. now `student_p` is left pointing at memory that is unsafe to access.

The Heap

Self-Memory Management

- So far, all memory we've work with have been allocated on a part of computer memory called the **stack**.
  - As soon as a block of memory goes 'out of scope', it gets deallocated (destroyed) by the program.
  - This is good because we don't have to manage memory ourselves — the program does it for us and so memory usage stays efficient. However, it's not very helpful when we want to return an address!
- Instead, we take advantage of a separate block of memory called the **heap**.
  - Anything allocated on the heap stays allocated *forever* — the program will not manage that part of memory for us.
  - This means we must free (deallocate) memory on the heap when we no longer need that block of memory, otherwise we'll create **memory leaks** (more on this next week). This means that the block of memory can't be used by other processes even though we'll never use what is stored there ever again.
  - Luckily, most operating systems will automatically deallocate memory on the heap when your program terminates, but you should *not* rely on this. For every memory allocation on the heap, we must make sure we deallocate it!
  - We use the function `malloc` to allocate memory, and `free` to deallocate memory.

Returning an Address: Good

`free` and `malloc` come from `stdlib.h`

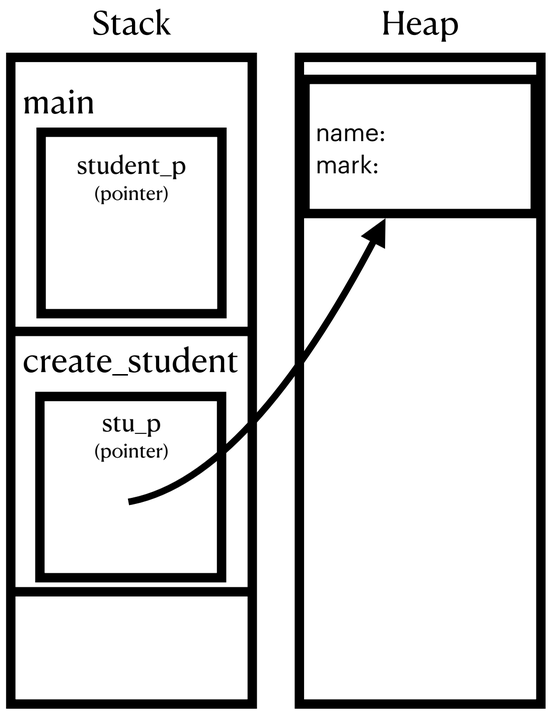
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_LENGTH 50

struct student {
    char name[MAX_LENGTH];
    int mark;
};

struct student *create_student(char *name, int mark);

int main(void) {
    struct student *student_p = create_student("Tom Kunc", 71);
    printf("Name: %s, Mark: %d\n", student_p->name, student_p->mark);
    free(student_p)
    return 0;
}

struct student *create_student(char *name, int mark) {
    struct student *stu_p = malloc(sizeof(struct student));
    strcpy(stu_p->name, name);
    stu_p->mark = mark;
    return stu_p;
}
```

Frees (deallocates) memory from the heap once we no longer need so the memory can be used again

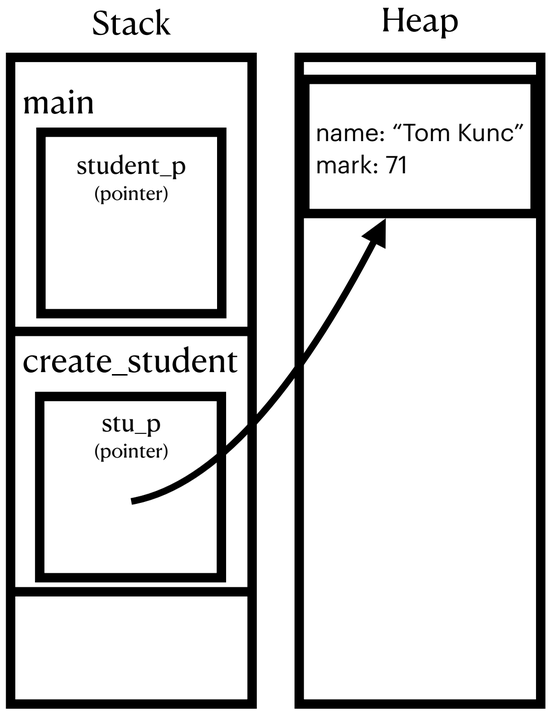
`sizeof` is an operator (not a function) evaluating the number of bytes of the type it is given

memory `alloc`ate:
Allocates memory sufficient for a `struct student` on the **heap**, and returns a pointer to the block of memory it allocated

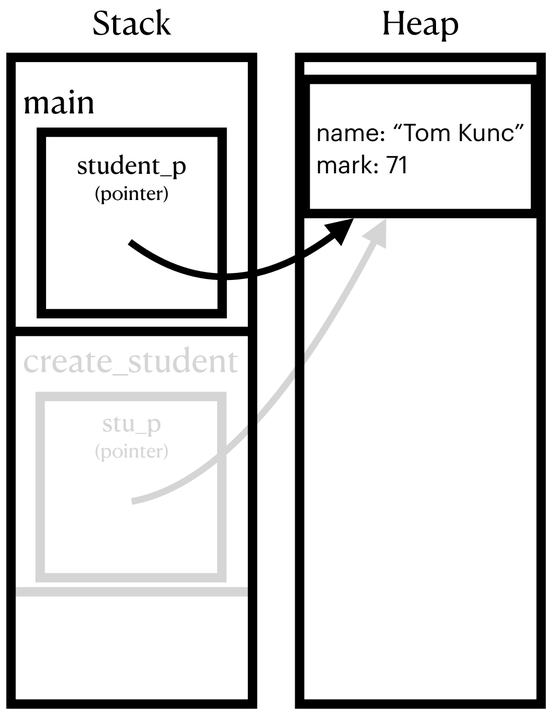
Memory Model



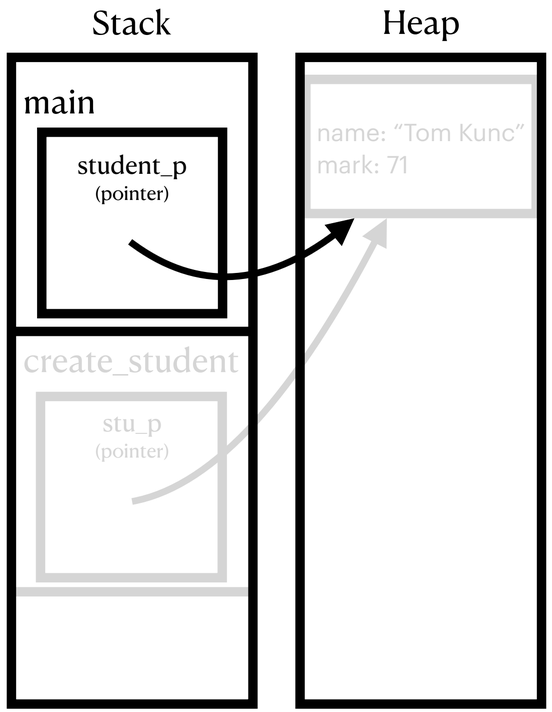
`malloc` allocates a block of memory on the heap for a `struct student`



The struct fields are initialised by dereferencing the `stu_p` pointer



`create_student` returns the address to the heap-allocated struct, which is copied into the `student_p` variable. `student_p` points to the heap-allocated struct.



When **no longer needed**, we call `free` to deallocate memory on the heap so that other programs can use it.

<u>We should never access freed memory!</u>

- We can no longer guarantee that what is at the memory address is what we expect
- + security reasons etc...

Linked Lists

An Interesting Struct

Representing a 'Node'

- Recall how we can use structs in C, to represent things with characteristics (such as a student). We can also use it to represent something more abstract...
- Consider the following definition of a struct called a '**`struct node`**'

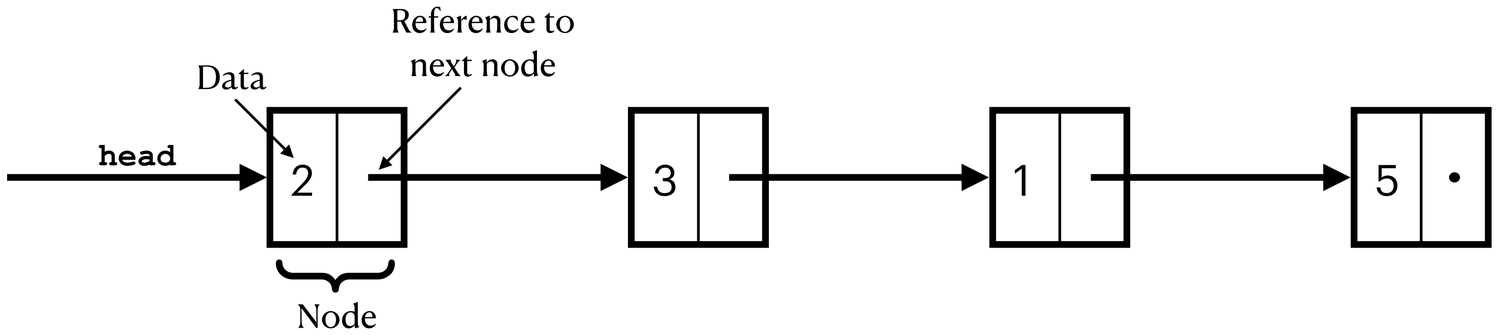
```
struct node {
    int data;
    struct node *next;
};
```

- It stores a pointer to a **`struct node`**!
  - We can make this **`struct node`** point to another **`struct node`**, which can then point to another **`struct node`**.
  - We can make something quite interesting with such a construction.

What is a Linked List

A Data Structure

- A linked list is an ordered collection of nodes. Every node contains a piece(s) of data, and (with the exception of the last node) a reference to the next element in the list.
- We can visualise it like so



- Here, we have a *list* of 4 nodes *linked* together by references.
- We are usually given only where the first node of a list is through the **`head`** pointer. This is our means of accessing elements in the list.
  - To get to the *k*th node in the list, we will have to pass through all the nodes before it.
  - This is unlike an array where we are able to access any element immediately with an index.
  - Linked lists, however, offer other benefits like easy resizing and insertion at the start of a list.

Why Struct Pointers?

Can't we store a struct instead?

- Suppose we rewrote the node like this:

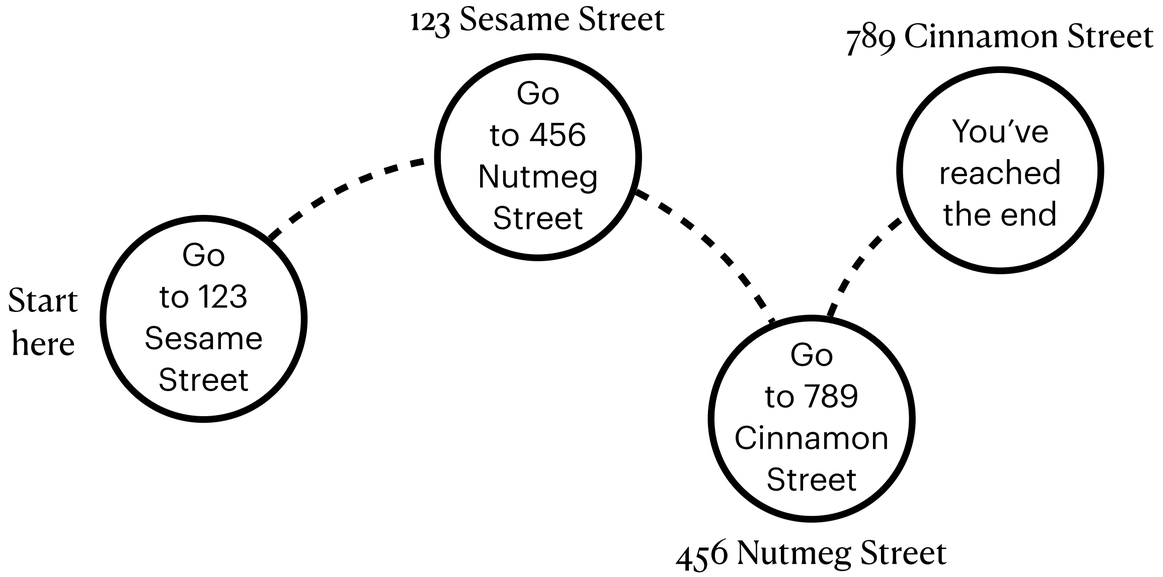
```
struct node {
    int data;
    struct node next;
};
```

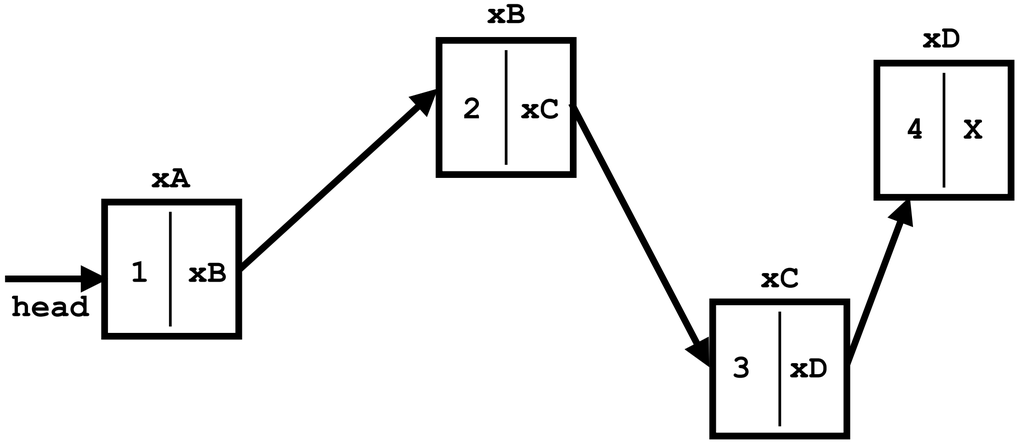
- For a program to run, the size of all variable types needs to be known so that the appropriate amount of memory can be allocated to variables of those types. How do we figure out the size of this **`struct node`**? (we can't)
  - To calculate the size of **`struct node`**, we need to know the size of a **`struct node`**
- A pointer has a fixed size (4 bytes), so the compiler knows how much memory a **`struct node`** takes up!

Linked Lists

Analogy

- To get to a certain element in the list, why do we need to traverse the list starting from the beginning? when we are given a linked list diagram, we can just 'see' all of the elements in a linked list.
- Elements of a linked lists aren't stored in a contiguous block of memory unlike an array, so the computer can't compute exactly where the *k*th element is— we can't find a node in the list without the previous one
- We can think of linked lists as a treasure hunt: we are given a starting location, which contains a hint only to the next location, which contains a hint to the next location etc. until we reach the end. At any point, we could have only gotten to the current location having gone past the previous one.



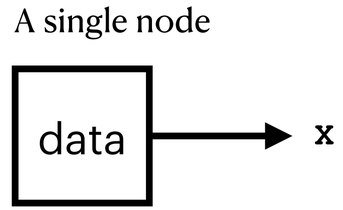


Linked List

Node Creation

- We mentioned at the beginning that we often use functions as a ‘factory’ for generating structs. We do the same for nodes in a linked lists. For example:

```c
// Creates a node initialised to the given data value
struct node *create_node(int data) {
    struct node *new_node = malloc(sizeof(struct node));
    new_node->data = data;
    new_node->next = NULL;
    return new_node;
}
```



- We often just call the node pointer the **`node`** and not **`node_ptr`**

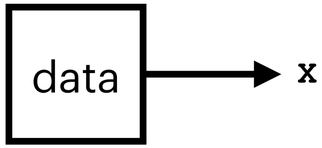
Problem Solving

- Whenever we are tasked with a problem to solve, we should always start by drawing a diagram.
- Suppose we wanted to write a function that inserts a node at the beginning of the list. We are given the list and a value to prepend
    1. We will need to create a node to prepend to the list.
    2. We will need to make that node point to the first element in the linked list.
    3. Readjust the head pointer.
- When we get to coding, we may need to work around some technical difficulties. Like
    - is there anything we've drawn that isn't practical in code? do we need to reorder some steps or perhaps figure out a workaround?
    - how do we tell the calling function that the head of the list has changed?
    - are there any 'edge' cases? That is, does our diagram assume something that doesn't work when the list is, say, empty or contains only one node etc.

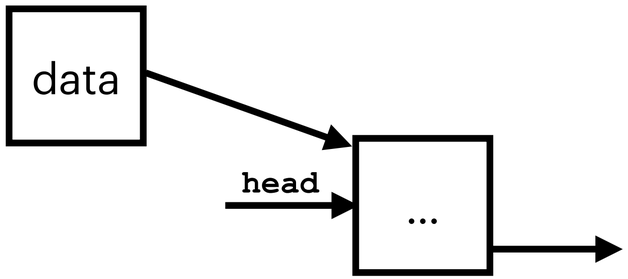
We have...

We should...

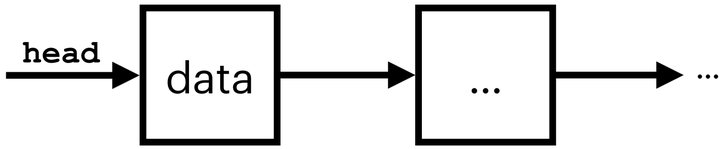
Step 1: Create the node to prepend



Step 2: Make the node point to the first element in the list



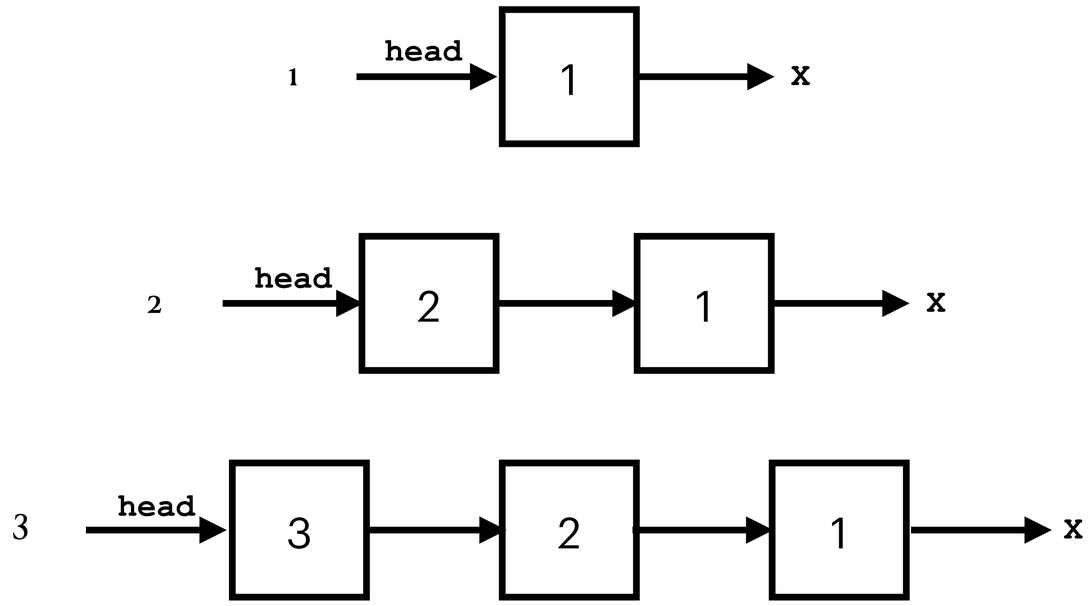
Step 3: Readjust the head pointer



Linked List Exercise

Prepending a Node

- Functions dealing with linked lists often return the head of the list because the start of the linked list might have changed, and the program that called the function wouldn't know otherwise!




```c
struct node *prepend(int data, struct node *head);

int main(void) {
    struct node *head = prepend(1, NULL);
    head = prepend(2, head);
    head = prepend(3, head);
    return 0;
}

struct node *prepend(int data, struct node *head) {
    struct node *new_head = create_node(data);
    new_head->next = head;
    return new_head;
}
```

Linked Lists

Traversal

- We often want to traverse a linked list (go through all or some of its elements)
- We use a while loop

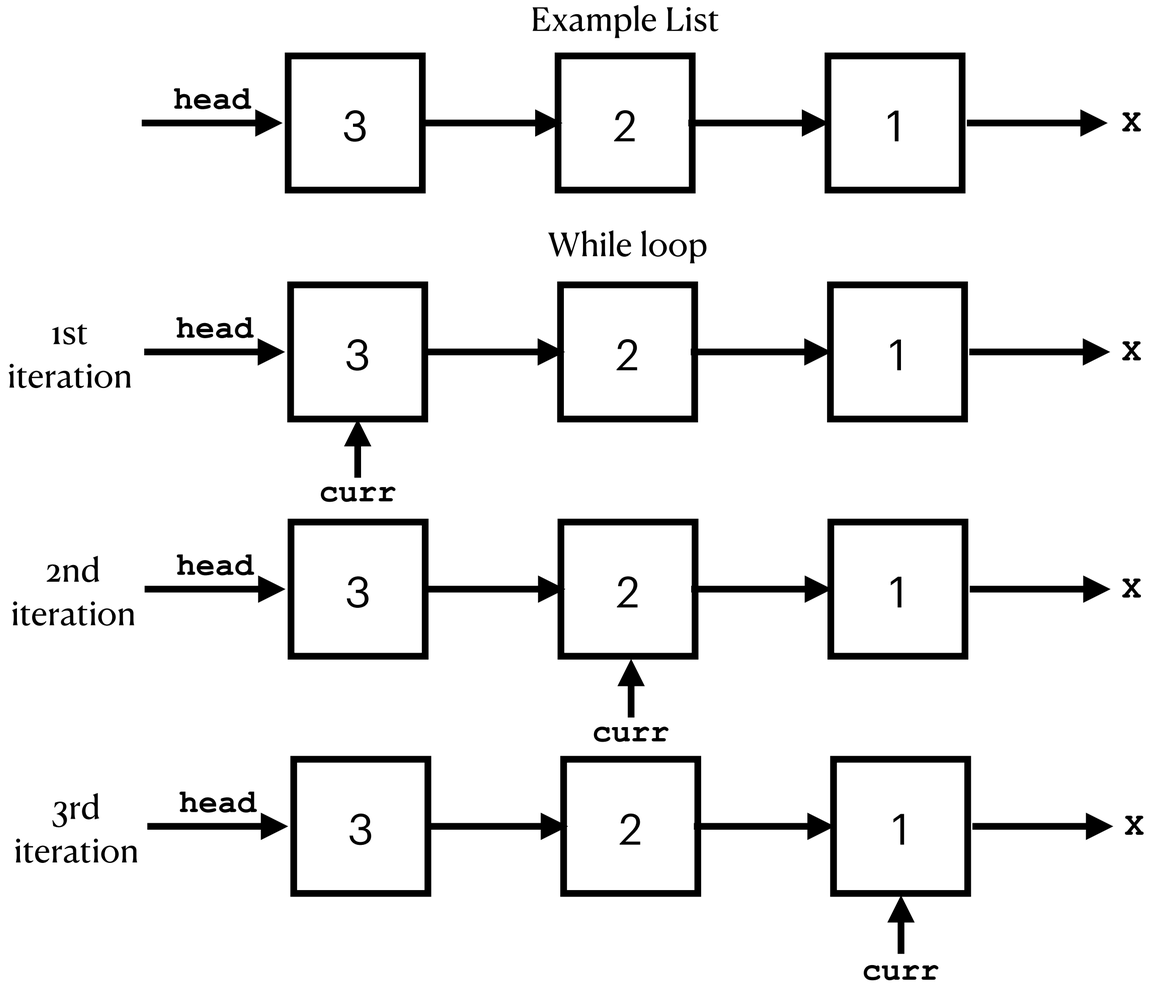
Initialise the variable we use to iterate over the list. Analogous to `int i = 0`

```
struct node *current = head;
while (current != NULL) {
    printf("%d -> ", current->data);
    current = current->next;
}
printf("X\n");
```

Determine the loop condition. Analogous to `i < length`

Use loop variable to access node fields. Analogous to `array[i]`

Determine step to take to get to the next iteration of the while loop. Analagous to `i++`.



Linked Lists vs Arrays

Is one better than the other? It depends.

- We want to justify the need for inventing something new.
  - What's wrong with what we have?
  - Are there any limitations with linked lists that arrays don't have?

Array	Both	Linked Lists
• We are able to access any piece of data immediately using an index. • Memory efficient — we only need to store one thing at each index.	• We can store many variables that are associated with each other under one variable name. Through just one entity in our code, we can access multiple pieces of data.	• We don't need to know the size of the list upfront and can change the size of the list with ease. • Inserting at the start of the list is fast — we don't need to shift elements.

Tips

How to Practice Linked Lists

- There are common programming patterns that arise when performing common operations on linked lists, such as creation, insertion, traversal of linked lists and deletion of nodes.
  - For example, one pattern we've seen is using a function to create nodes, instead of writing out the initialisation every time.
  - **Observe and learn** these patterns — don't try to reinvent the wheel and introduce unnecessary complexity (but do experiment to see why these operations cannot be simplified further!) This will allow you to offload these basic operations to muscle memory, and focus your attention on solving the problem at hand.
- Do and **repeat** lab exercises on linked lists after this week.
  - When you come back to the lab exercises, you'll likely find that you are able to simplify your solution and solve the problem more effectively!