COMP1511/1911 Programming Fundamentals

Week 7 Lecture 2

Linked Lists

Assignment 2

Out today 1pm

Live stream online Monday 4:30pm

Assignment 2: CS Dungeon

- It is an individual assignment
- Aims of the assignment
    - Work with a larger problem and codebase
    - Work with multiple C files
    - Problem solve with linked lists
        - You MUST use linked linked lists. You can’t change the linked lists into arrays and just do it with arrays!!!!!!!! You will get 0 performance.
    - Practice using strings
    - Being a responsible heap user (free your malloced memory)
- You will be assessed on style! 20% of your mark
- COMP1911 just need to complete stages 1 and 2

Assignment 2 Live Stream

Time: Monday 4:30

YouTube Link

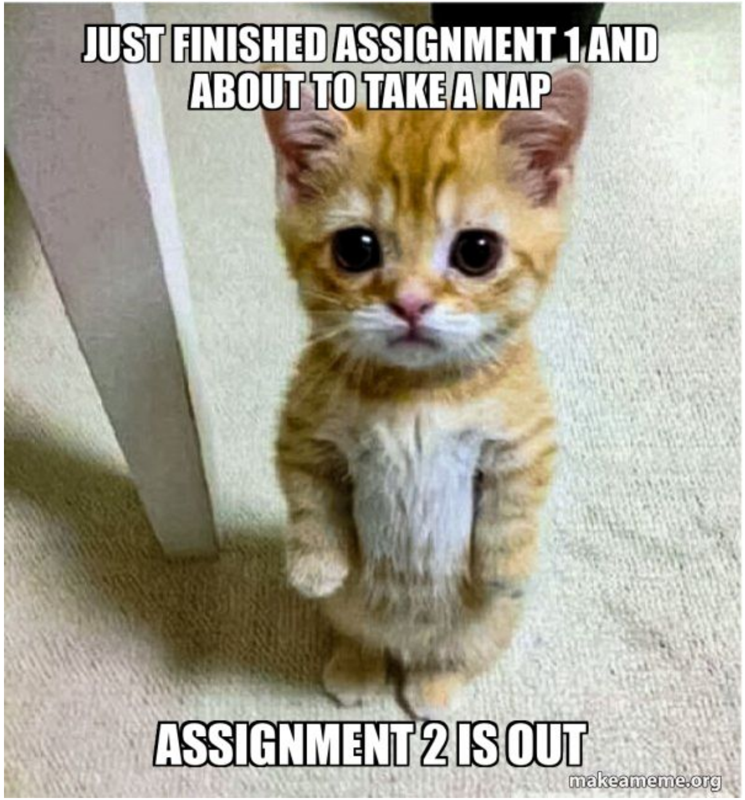
Recording will also be available

Assignment Due Date:

Friday Week 10 5pm

Don't leave it until the last minute!

Help sessions will be very busy the week before the deadline!!!!!!!!



Last Lecture

- Pointers basics recap
- Pointers and arrays
- Memory and the stack
- Dynamic Memory, malloc, realloc and the heap

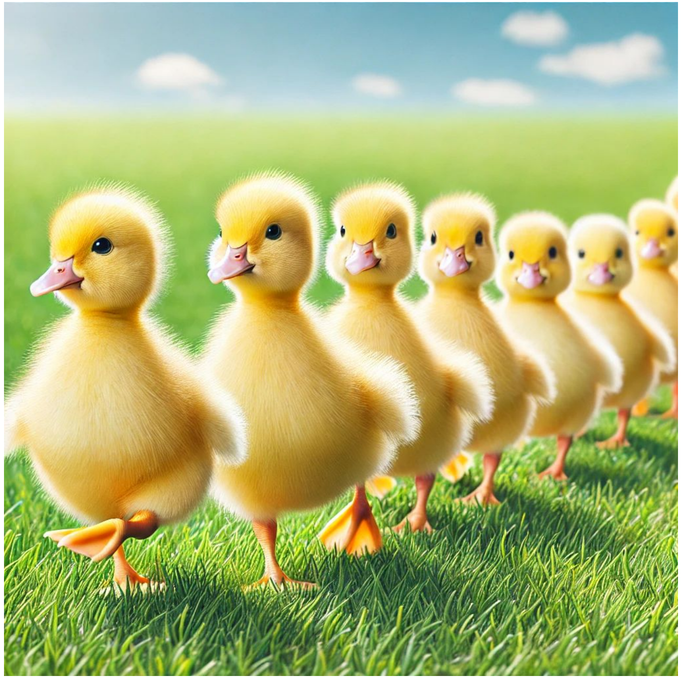
Today's Lecture

The moment you have all been waiting for

Linked Lists - Your first introduction

- Why are we learning linked lists?
- What is a linked list?
- Inserting at the head
- Traversing a linked list
- Inserting at the tail

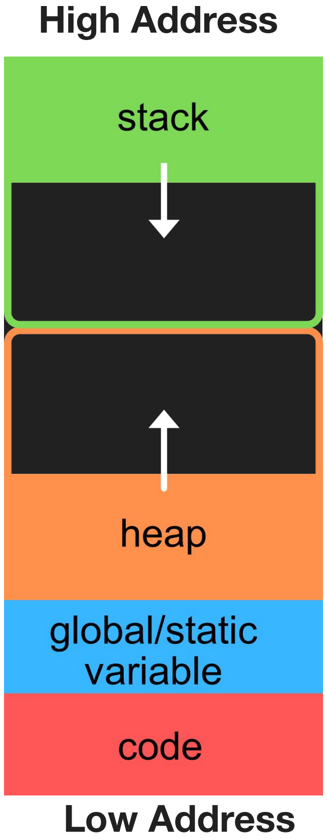
But first a recap of malloc!



Link to Week 7 Live Lecture Code

https://cgi.cse.unsw.edu.au/~cs1511/24T3/live/week_7/

The Heap

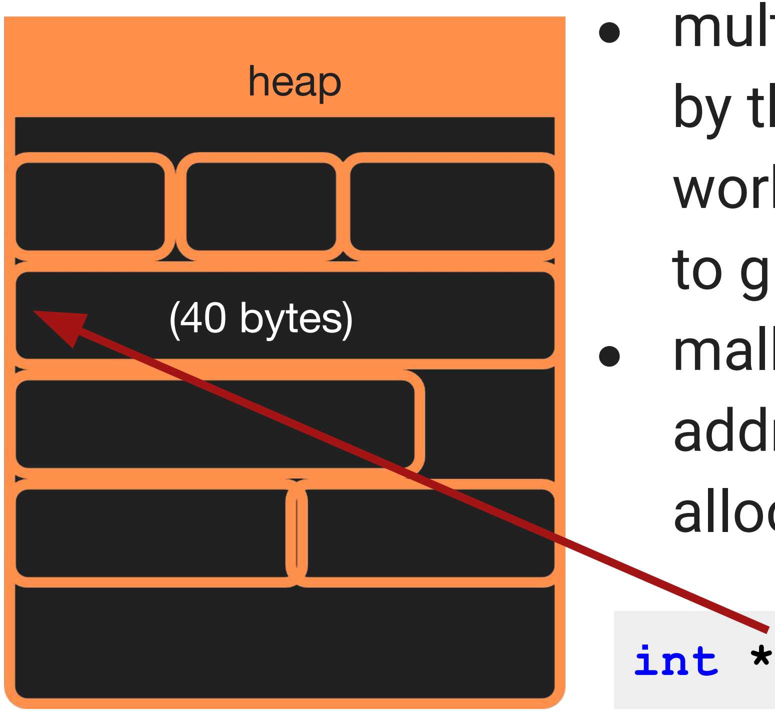


- Unlike stack memory, heap memory is allocated by the programmer
- It won't be deallocated until it is explicitly freed by the programmer
- You now have the power to control memory on the heap!
- With power comes heaps of responsibility

The Heap: malloc

- malloc is short for memory allocate
- malloc lets us ask for a number of bytes of memory on the heap
- malloc returns
  - a pointer to the chunk of memory or
  - NULL if there is not enough memory left to give us
  - You should always check for NULL in case.
- This allows us to dynamically create memory when we need it that will last beyond the end of functions and until we say we don’t want it anymore.
- You need to #include <stdlib.h> to use malloc

Using malloc



- multiply the number of elements you need by the sizeof the type of the element to work out how many bytes you want malloc to give you
- malloc will return a pointer to the starting address of the chunk of memory it allocated

```
int *numbers = malloc(10 * sizeof(int));
```

Putting it all together

```
// create array
int *data = malloc(num_elements *sizeof(int));
// check malloc was successful

// Use the array somehow
// etc etc

// Free array when finished with array
free(data);
```

Note: You can check for memory leaks using dcc with the flag

`dcc --leak-check`

Exercise: return pointer to struct

```
struct coordinate {
    int x;
    int y;
};
// return a pointer to a coordinate struct with given x and y
struct coordinate *create_coordinate(int x, int y);
// print coordinate in the format (x, y)
void print_coordinate(struct coordinate *p);
```

Write the functions and write a main function to

1. Call the first function with x and y 10, -1 and
2. Call the function to print the point.

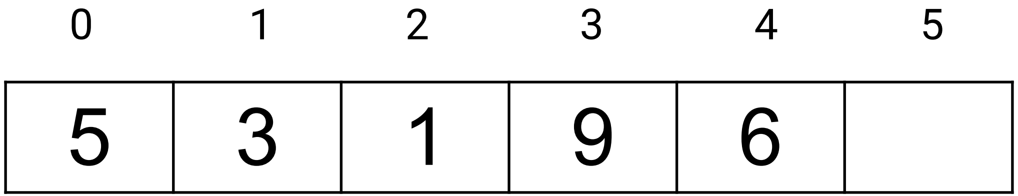
Linked Lists

Linked Lists

- An alternative to using an array to store collections of data
    - Arrays are amazing and we won’t be forgetting about them
    - This is just another option!
- Linked Lists are suitable for sequential data:
    - playlists of songs
    - image galleries
    - web browser history
- Why would we want to use a linked list instead of an array?

Array Advantages

- Store collections of data in contiguous blocks of memory
- Great for sequential access or random access
- It is easy to insert or delete items at the end



0	1	2	3	4	5
5	3	1	9	6	

Array Disadvantages

- Messy and inefficient for inserting or deleting in the middle
- E.g. How can we insert an item at or delete from index 1 in the array below?

0	1	2	3	4	5
5	3	1	9	6	

Array Disadvantages

We would need to move all the subsequent data along to

- make room to insert an item at index 1
- remove the gap if we deleted an item at index 1

0	1	2	3	4	5
5	3	1	9	6	

Array Disadvantages

How can we insert an item into the array below?

- With a static array we can’t!
- With a dynamic array we can use realloc
  - How much bigger do we make it? Just 1 bigger? double the size?

0	1	2	3	4	5
5	3	1	9	6	7

Linked List Advantages

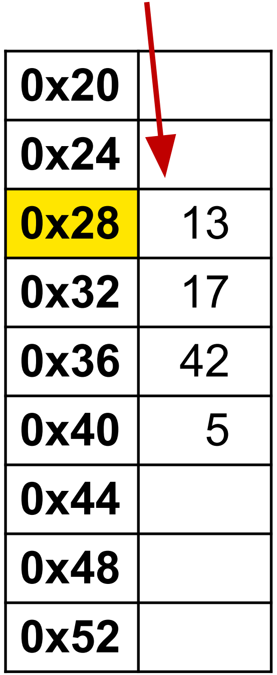
- They are dynamic structures
  - They grow and shrink as needed
- They don’t need contiguous memory like an array
- Insert or delete items anywhere in the list
  - by modifying one or two pointers
  - without moving existing data

Linked List Disadvantages

- Not good for random access 🙁
    - You have to traverse from the beginning of the list
- Extra overhead of storing a pointer for each data item

Arrays in Memory

```
int array[] = {13, 17, 42, 5};
```

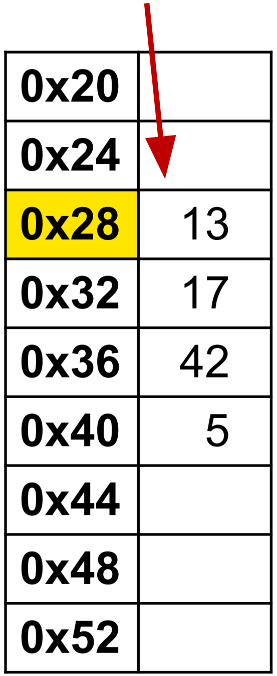


0x20	
0x24	
0x28	13
0x32	17
0x36	42
0x40	5
0x44	
0x48	
0x52	

- The array name gives us the address of the beginning of the chunk of memory
- Arrays are stored contiguously which allows us to use indexes and make random access quick and easy

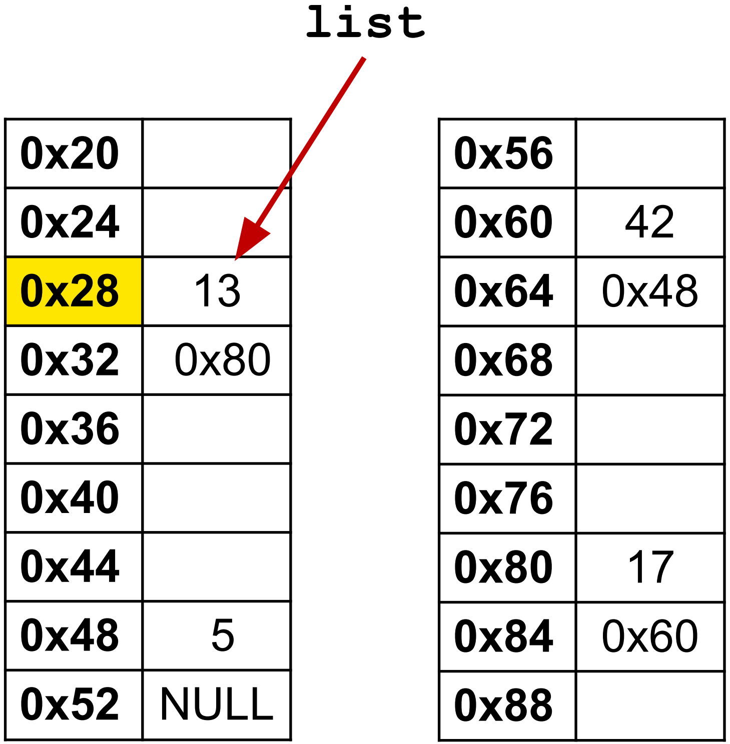
Arrays vs Linked Lists in Memory

```
int array[] = {13, 17, 42, 5};
```



0x20	
0x24	
0x28	13
0x32	17
0x36	42
0x40	5
0x44	
0x48	
0x52	

- Linked list data is not contiguous
- It is scattered throughout memory.



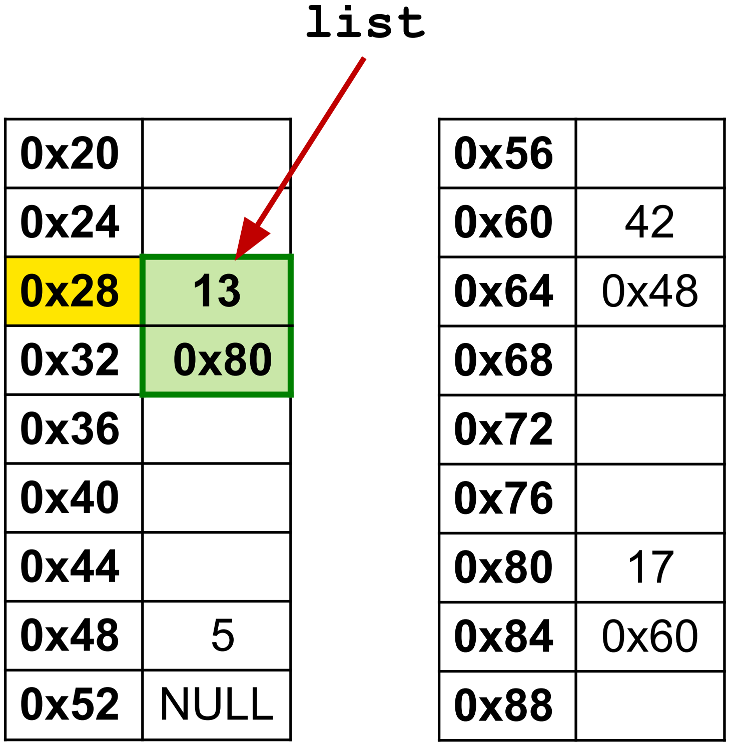
`list`

0x20	
0x24	
0x28	13
0x32	0x80
0x36	
0x40	
0x44	
0x48	5
0x52	NULL

0x56	
0x60	42
0x64	0x48
0x68	
0x72	
0x76	
0x80	17
0x84	0x60
0x88	

Linked Lists in Memory

- You need a pointer to the first piece of data in the list
- And for every piece of data you store in the list you need to store a link (pointer containing the address) to the next item in the list.
- Like a scavenger hunt.

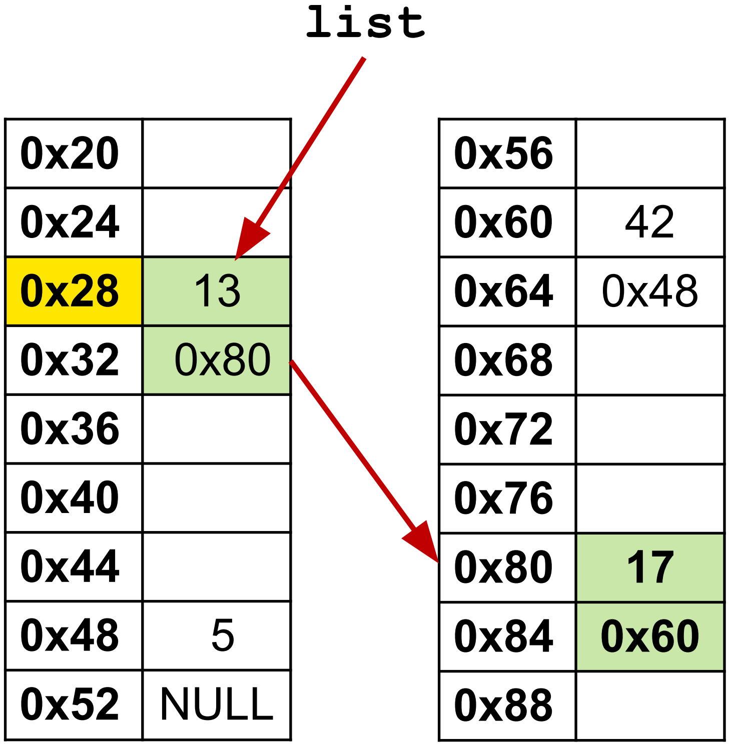


0x20	
0x24	
0x28	13
0x32	0x80
0x36	
0x40	
0x44	
0x48	5
0x52	NULL

0x56	
0x60	42
0x64	0x48
0x68	
0x72	
0x76	
0x80	17
0x84	0x60
0x88	

Linked Lists in Memory

- You need a pointer to the first piece of data in the list
- And for every piece of data you store in the list you need to store a link (pointer containing the address) to the next item in the list.
- Like a scavenger hunt.



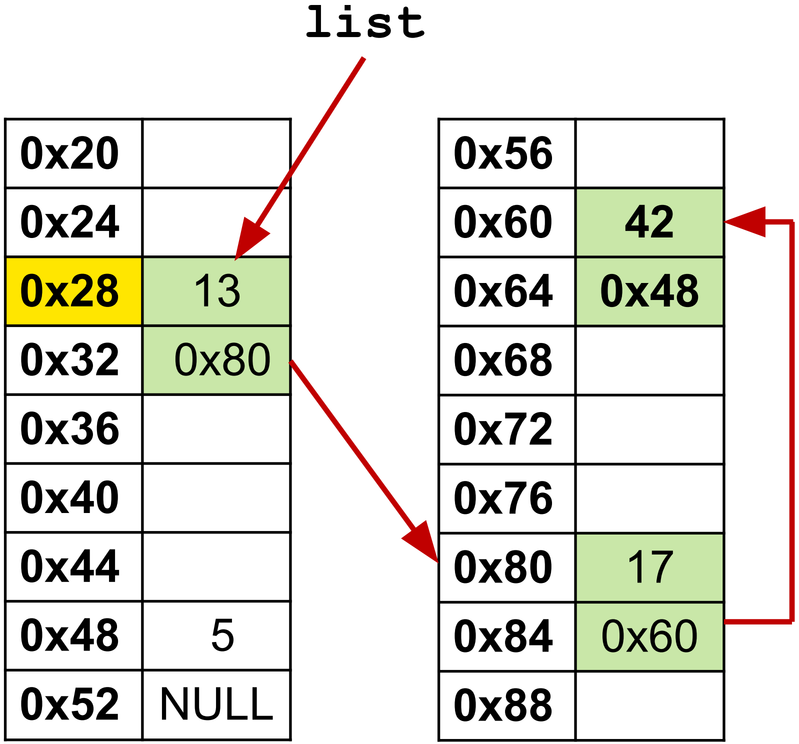
list

0x20	
0x24	
0x28	13
0x32	0x80
0x36	
0x40	
0x44	
0x48	5
0x52	NULL

0x56	
0x60	42
0x64	0x48
0x68	
0x72	
0x76	
0x80	17
0x84	0x60
0x88	

Linked Lists in Memory

- You need a pointer to the first piece of data in the list
- And for every piece of data you store in the list you need to store a link (pointer containing the address) to the next item in the list.
- Like a scavenger hunt.



list

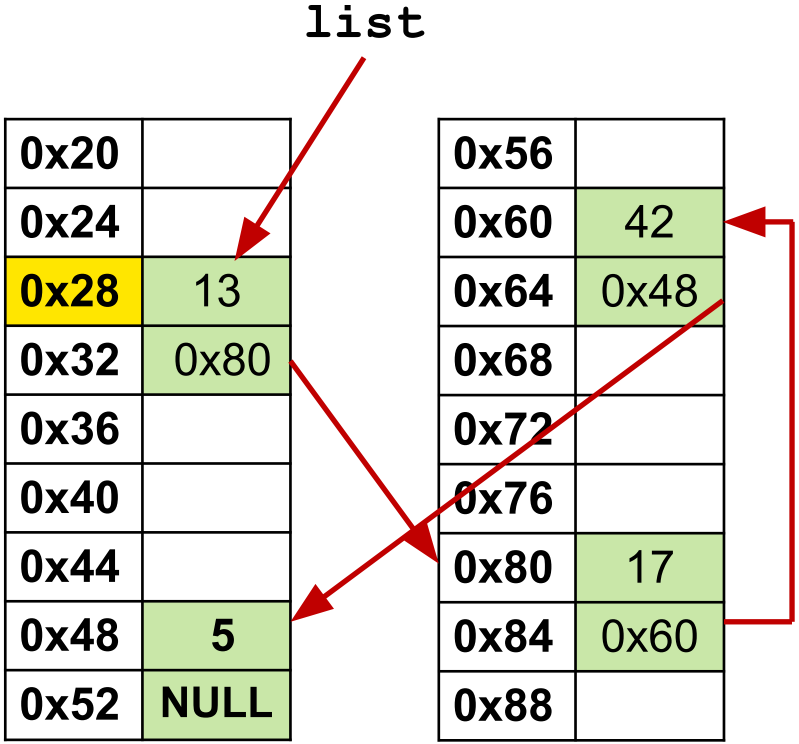
0x20	
0x24	
0x28	13
0x32	0x80
0x36	
0x40	
0x44	
0x48	5
0x52	NULL

0x56	
0x60	42
0x64	0x48
0x68	
0x72	
0x76	
0x80	17
0x84	0x60
0x88	

Linked Lists in Memory

- When the value of the pointer to the next piece of data is NULL you have reached the end of the list.

- **Congratulations!**
  You have just traversed your first linked list.

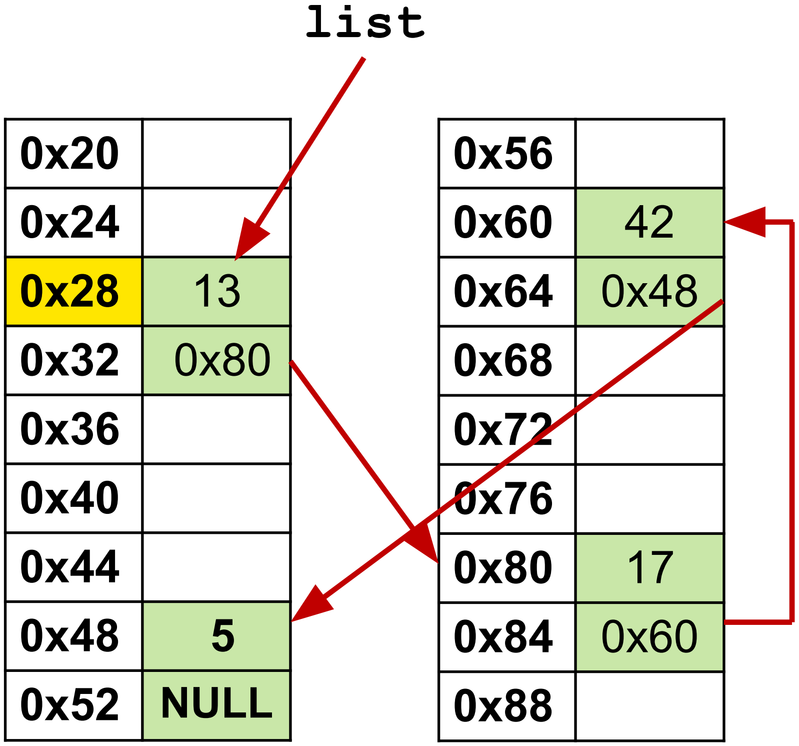


0x20	
0x24	
0x28	13
0x32	0x80
0x36	
0x40	
0x44	
0x48	5
0x52	NULL

0x56	
0x60	42
0x64	0x48
0x68	
0x72	
0x76	
0x80	17
0x84	0x60
0x88	

Linked Lists in Memory

- We say it is sequential as we have to start at the beginning of the list and traverse to access items
- We can't jump to a particular item like we can with array indexes



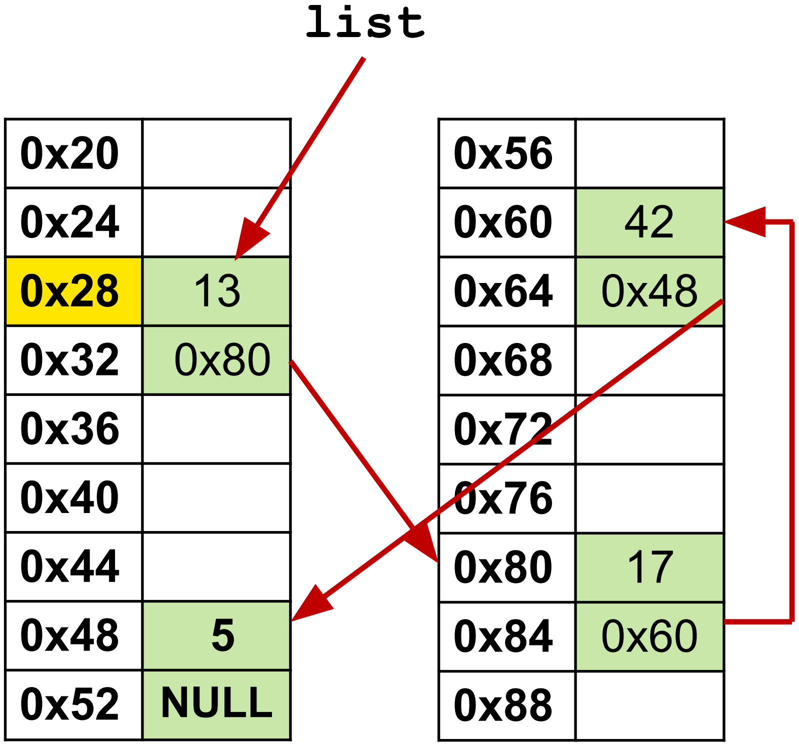
0x20	
0x24	
0x28	13
0x32	0x80
0x36	
0x40	
0x44	
0x48	5
0x52	NULL

0x56	
0x60	42
0x64	0x48
0x68	
0x72	
0x76	
0x80	17
0x84	0x60
0x88	

Linked List Nodes

What type in C would allow us to store both the

- **int** data and also the
- **address** of the next item in the list?



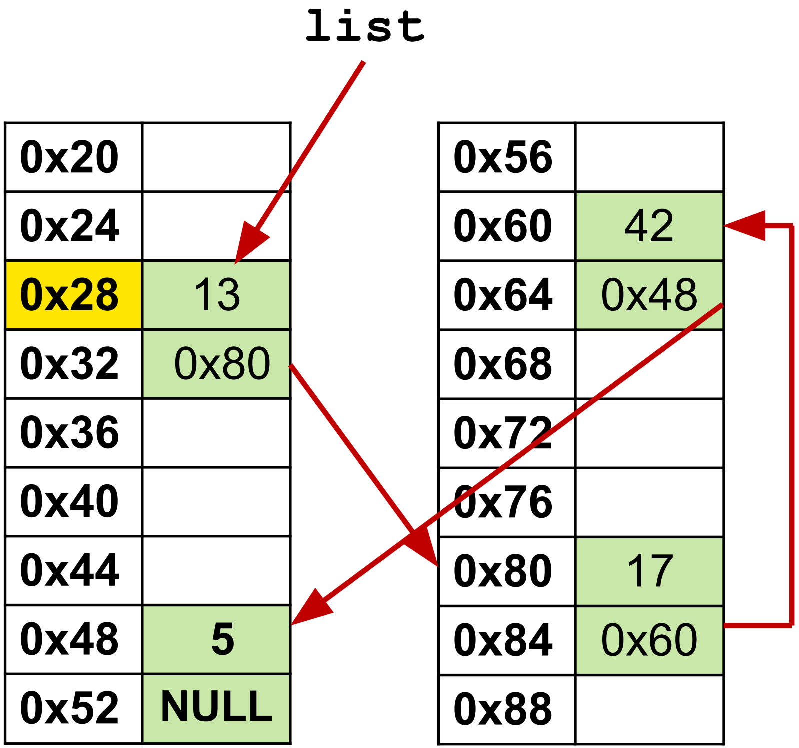
0x20	
0x24	
0x28	13
0x32	0x80
0x36	
0x40	
0x44	
0x48	5
0x52	NULL

0x56	
0x60	42
0x64	0x48
0x68	
0x72	
0x76	
0x80	17
0x84	0x60
0x88	

Linked List Nodes

- We can store our data and a pointer together in a struct.
- We often call these nodes when working with linked lists

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



0x20	
0x24	
0x28	13
0x32	0x80
0x36	
0x40	
0x44	
0x48	5
0x52	NULL

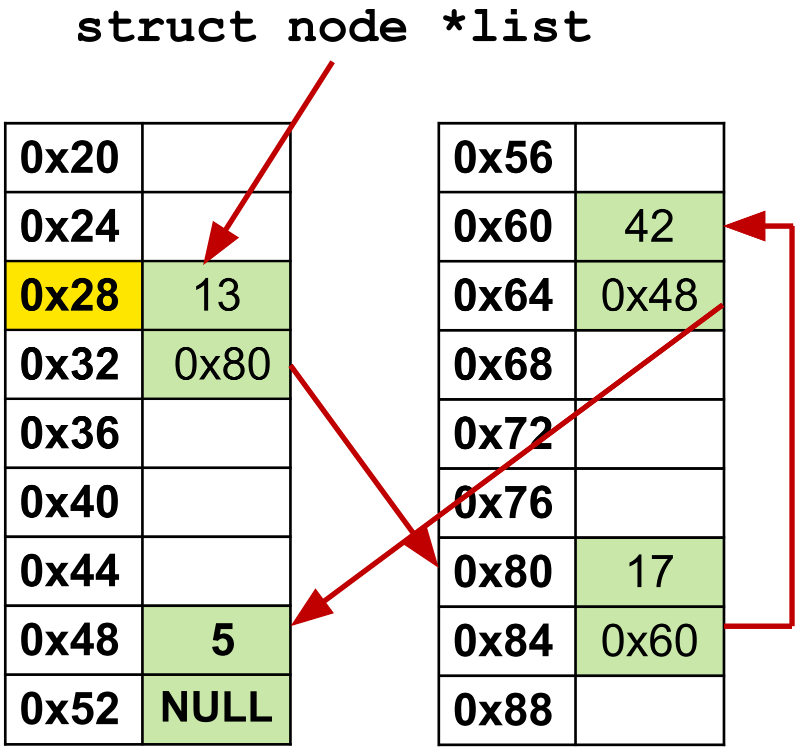
0x56	
0x60	42
0x64	0x48
0x68	
0x72	
0x76	
0x80	17
0x84	0x60
0x88	

Linked List Nodes

The list variable is a pointer to the first node in the list

```
struct node *list;
```

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



struct node *list

0x20	
0x24	
0x28	13
0x32	0x80
0x36	
0x40	
0x44	
0x48	5
0x52	NULL

0x56	
0x60	42
0x64	0x48
0x68	
0x72	
0x76	
0x80	17
0x84	0x60
0x88	

Linked List Nodes

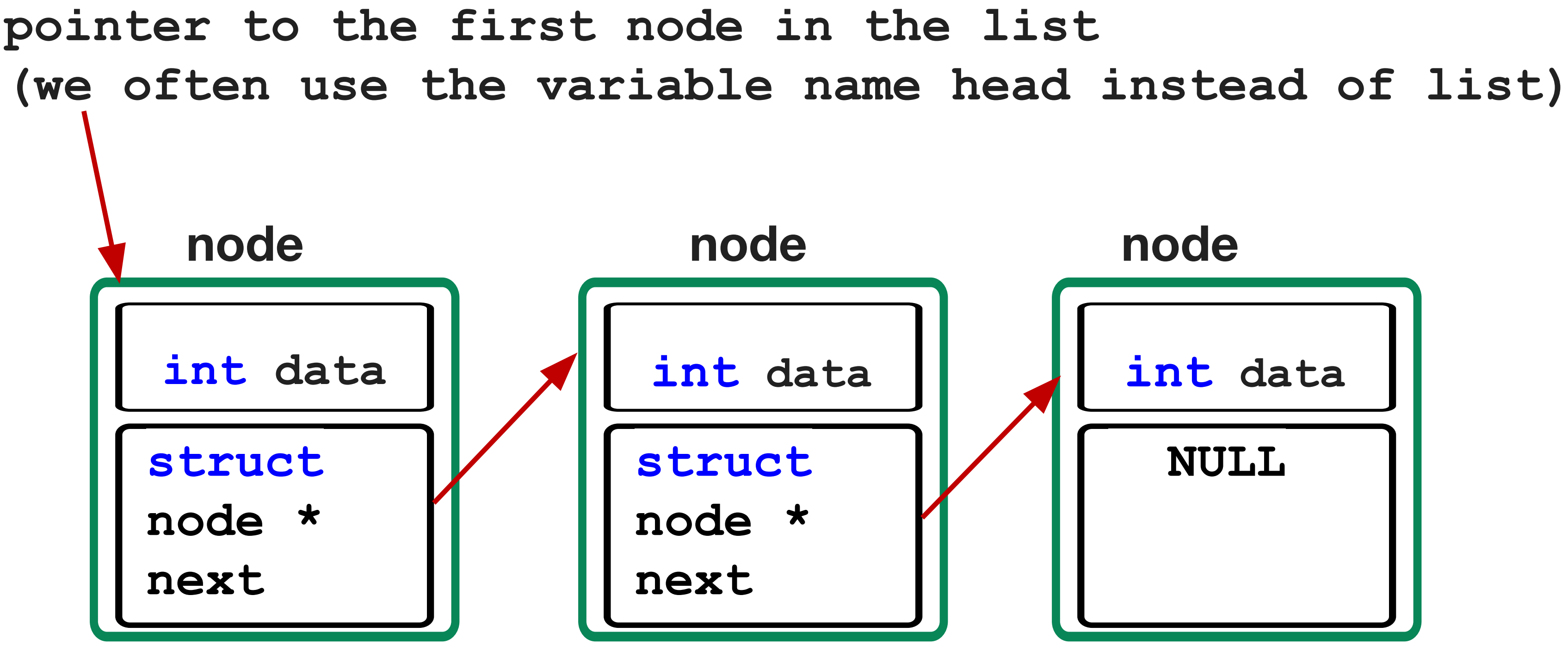
The list variable is a pointer to the first node in the list

```
struct node *list;
```

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

- Each node has some **data**
    - In this case it is one int but it could be whatever type of data you need
    - Later we will see different types of data in our linked lists
- Each node has a pointer to the **next** node (of the same data type)

Visualising Linked Lists



pointer to the first node in the list
(we often use the variable name head instead of list)

node

int data

struct node * next

node

int data

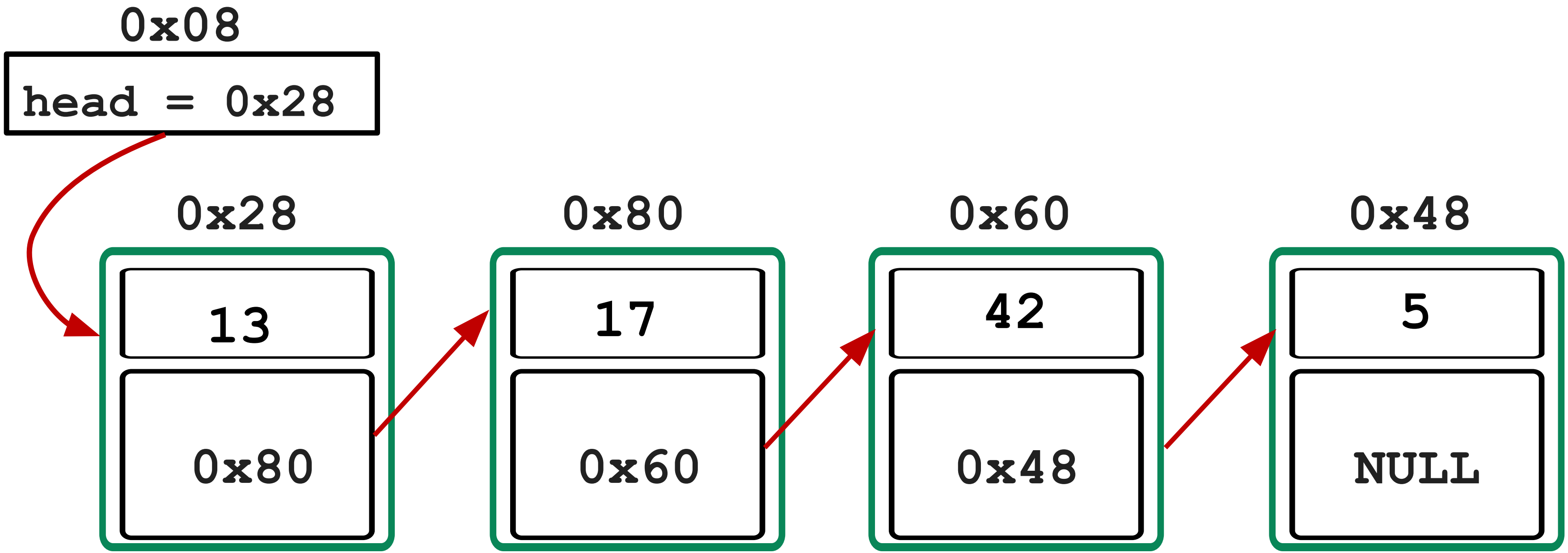
struct node * next

node

int data

NULL

Visualising Linked Lists

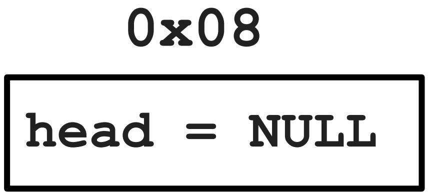


Creating a linked list

Let’s write the code to create a linked list with nothing in it.

```
struct node *head = NULL;
```

We can visualise it as follows



0x08

head = NULL

Hooray! Who said linked lists were difficult?

Creating a Node

We will be using **malloc** to create nodes on the **heap**.

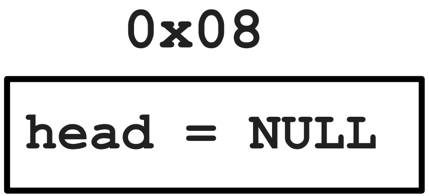
- we want full control to be able to
    - create new nodes whenever we need to
    - free them whenever we are finished with them

Steps needed are:

1. malloc a struct node
2. set the data member in the node
3. set the pointer to the next node

Creating a List with 1 Node in C

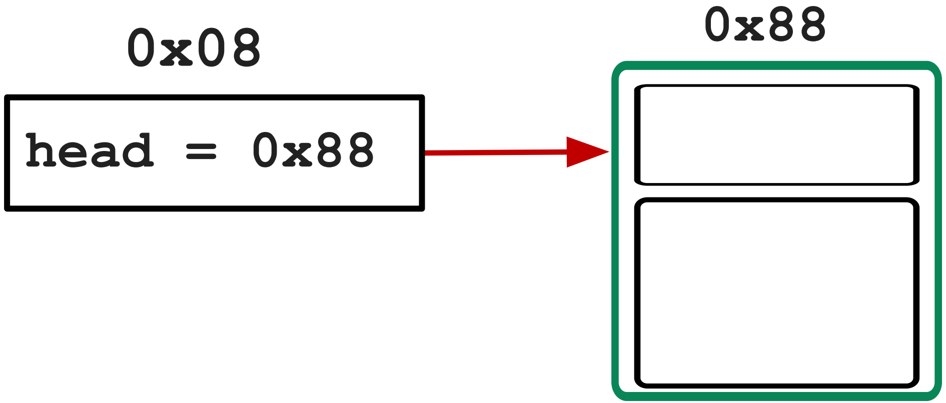
```
struct node {
    int data;
    struct node *next;
};
```



```
struct node *head = NULL;
```

Creating a List with 1 Node in C

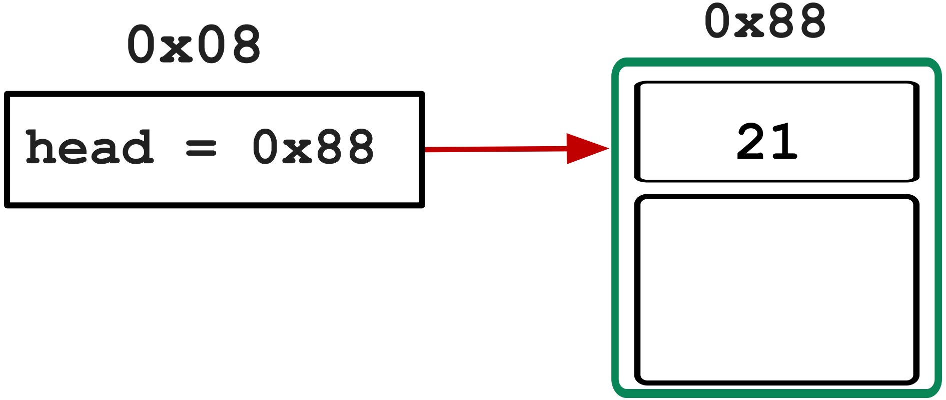
```
struct node {
    int data;
    struct node *next;
};
```



```
struct node *head = NULL;
head = malloc(sizeof(struct node));
```

Creating a List with 1 Node in C

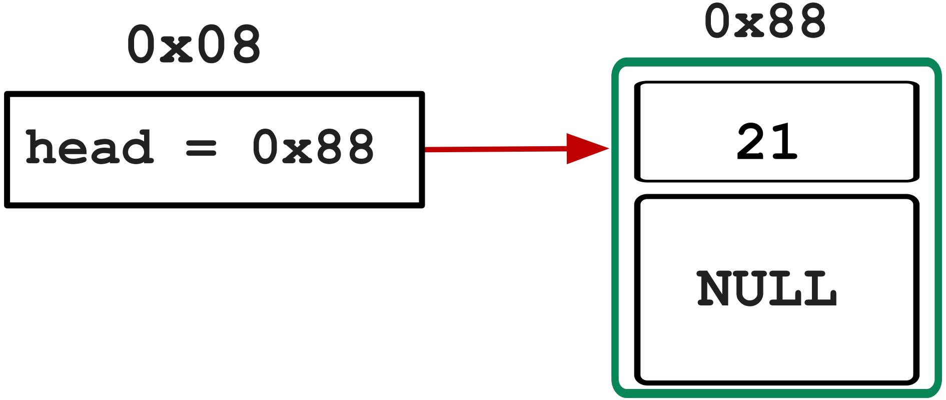
```
struct node {
    int data;
    struct node *next;
};
```



```
struct node *head = NULL;
head = malloc(sizeof(struct node));
head->data = 21;
```

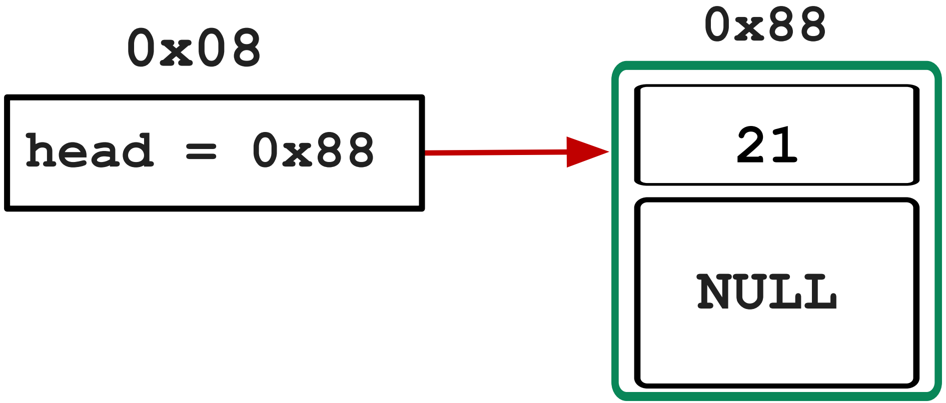
Creating a List with 1 Node in C

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



```c
struct node *head = NULL;
head = malloc(sizeof(struct node));
head->data = 21;
head->next = NULL;
```

Creating a List with 1 Node in C

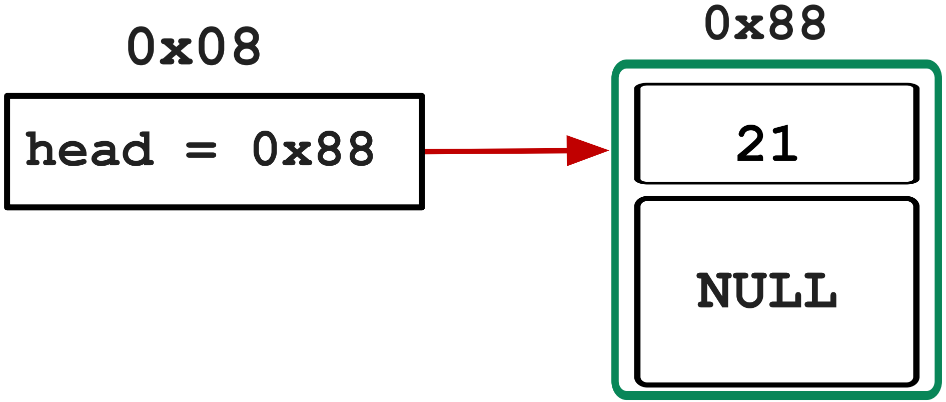


Now we have a linked list of size 1.

Let’s create another node.

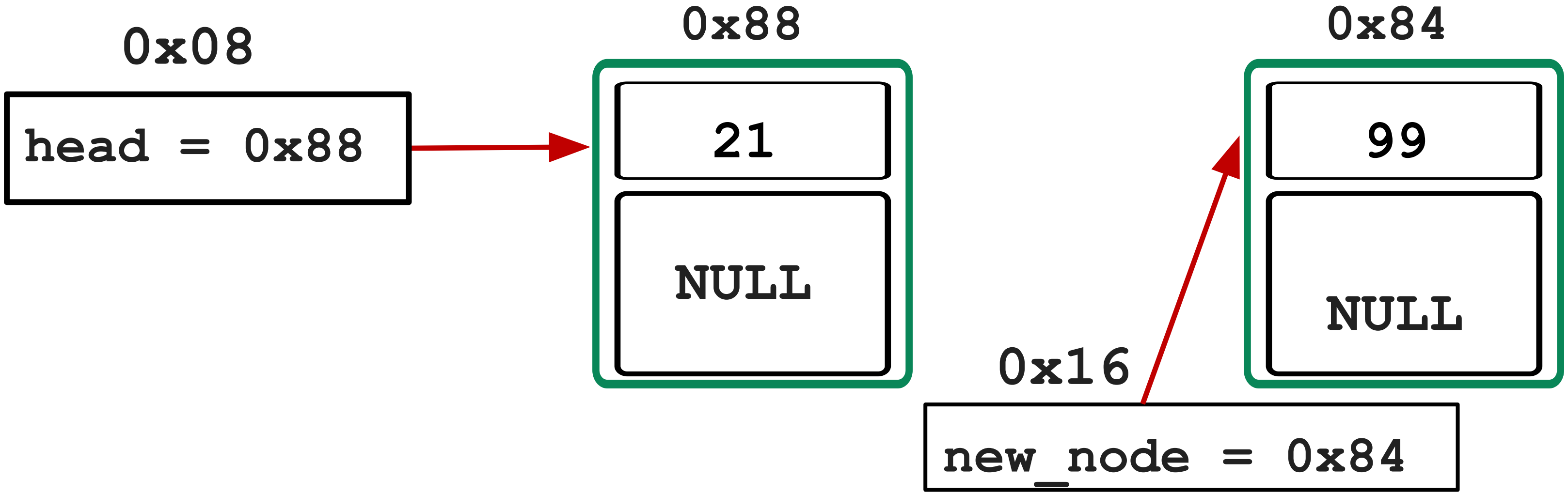
Then we can connect it to the end or the beginning of this list!

Connect 2 nodes: Add new node to the end



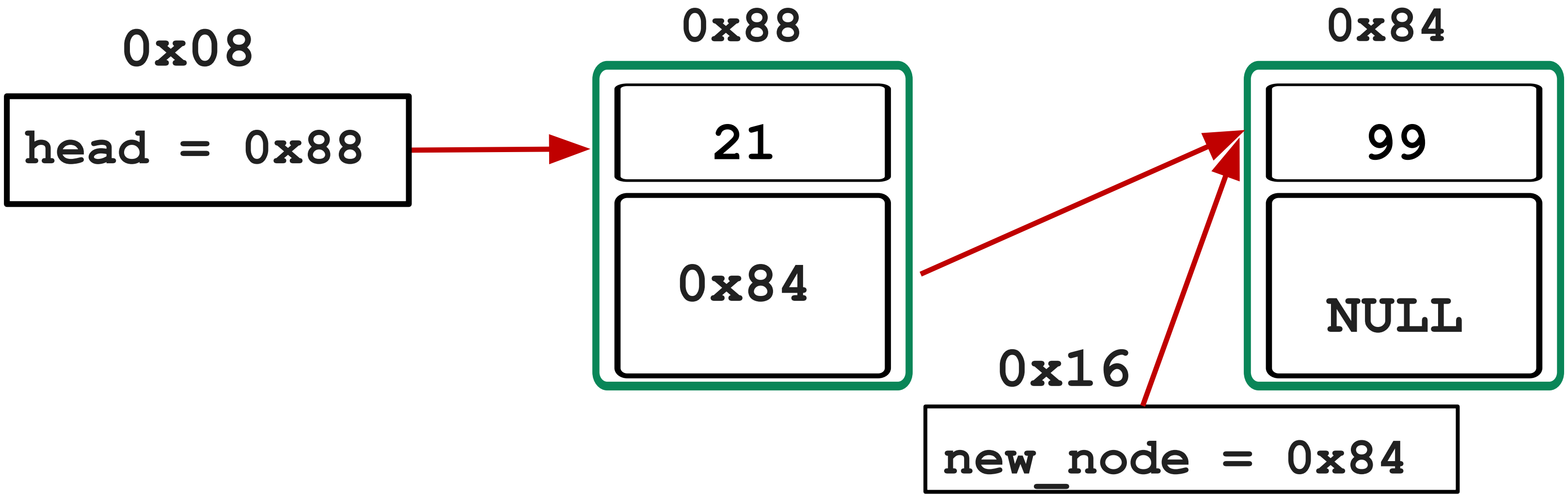
We will create a new node and link it to the end of this list
The end of the list is often called the tail.

Connect 2 nodes: Add new node to the end



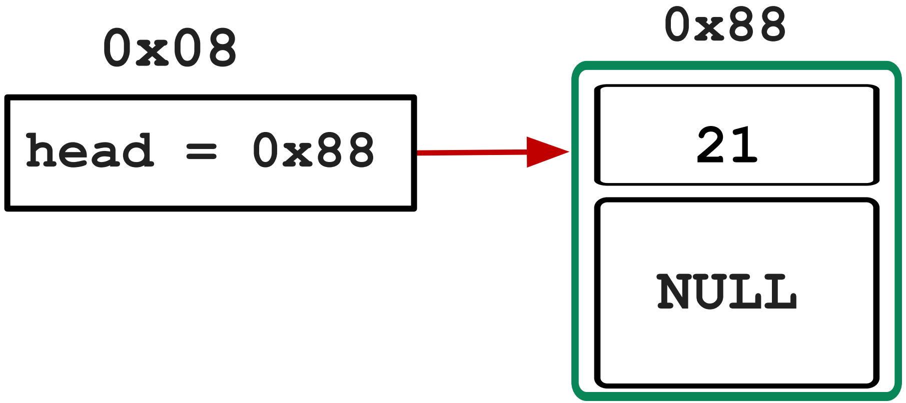
```
struct node *new_node = malloc(sizeof(struct node));
new_node->data = 99;
new_node->next = NULL;
```

Connect 2 nodes: Add new node to the end



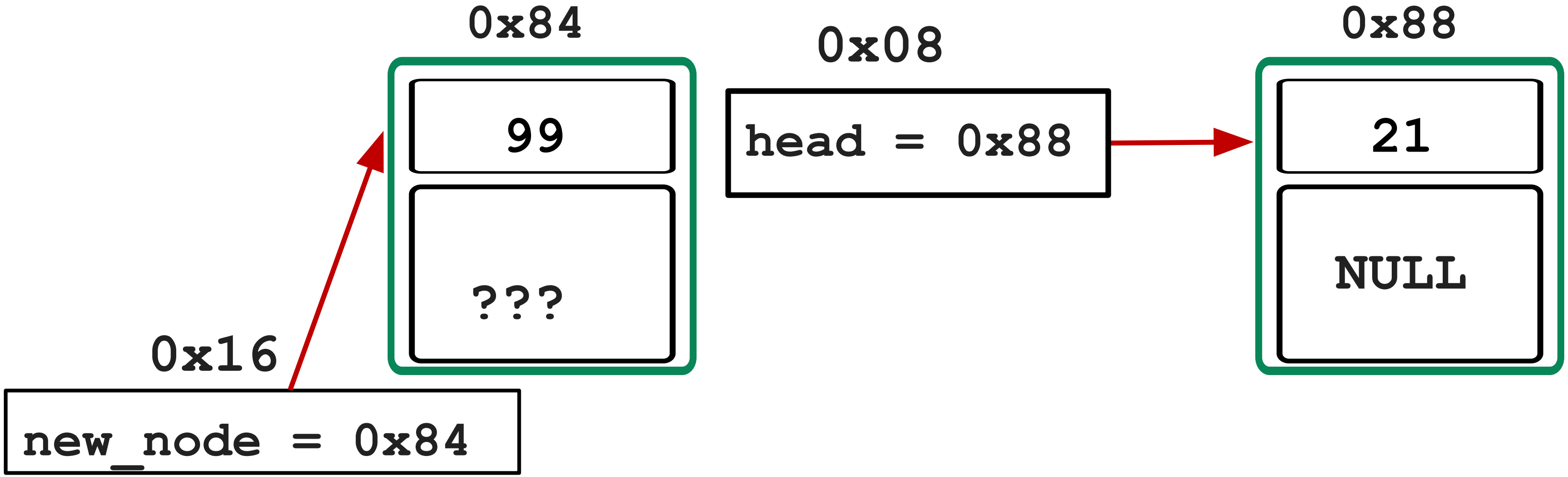
```
// Connect(link) the head of the list to the new_node
head->next = new_node;
```

Connect 2 nodes: Add new node to the start



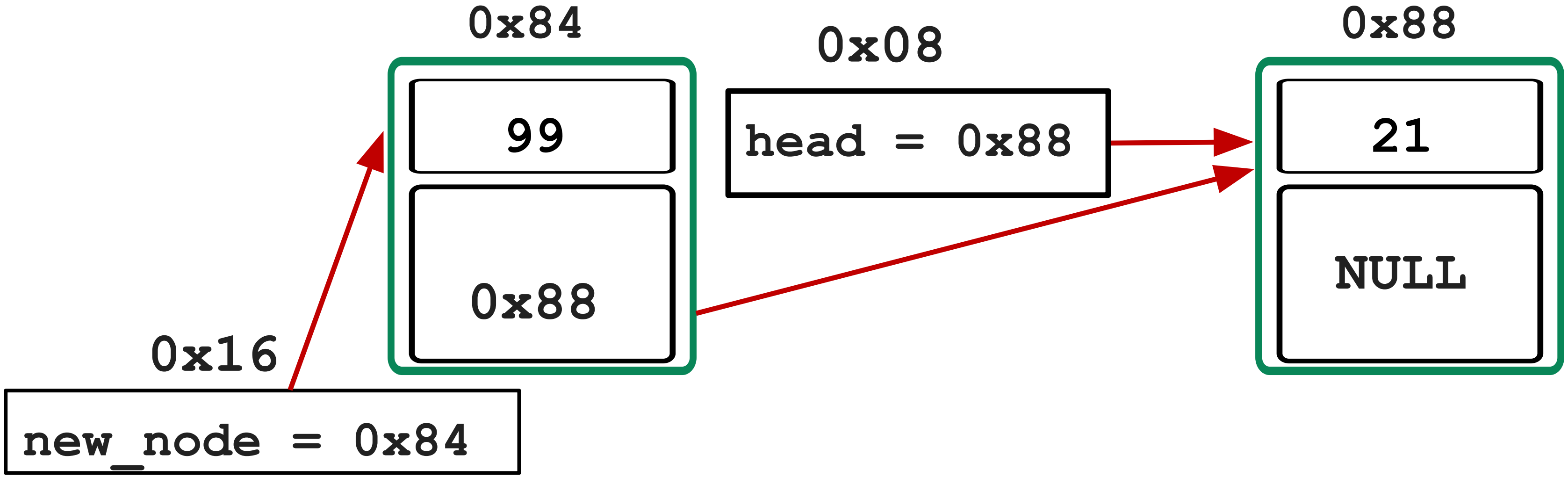
We will create a new node and link it to the start of this list
The start of the list is often called the head.

Connect 2 nodes: Add new node to the start



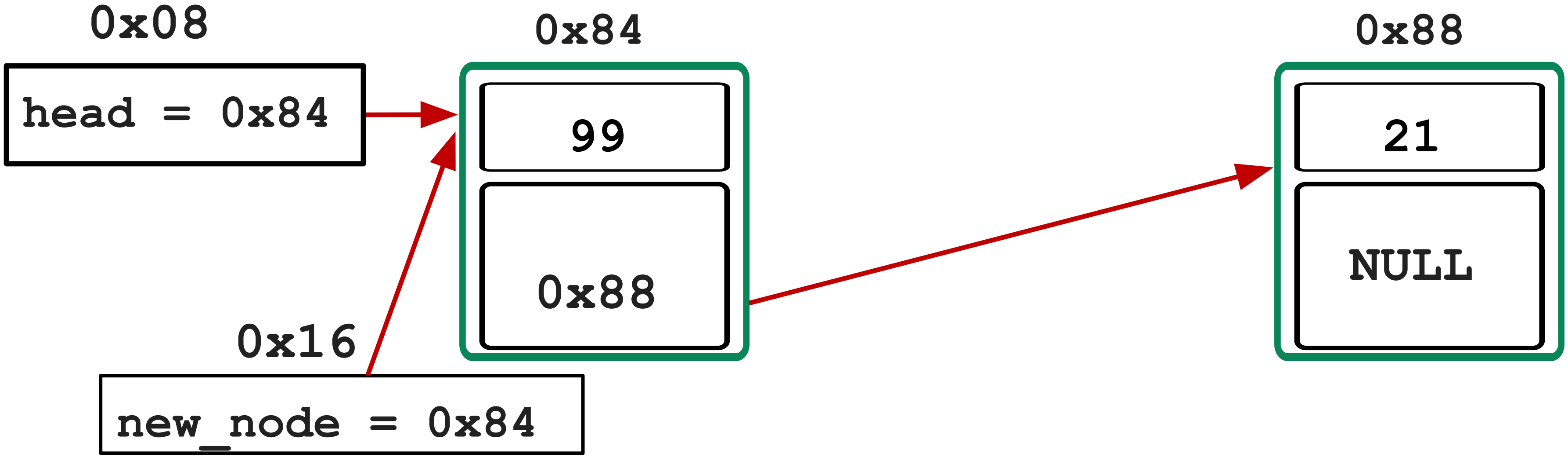
```
struct node *new_node = malloc(sizeof(struct node));
new_node->data = 99;
new_node->next = ???;
```

Connect 2 nodes: Add new node to the start



```
struct node *new_node = malloc(sizeof(struct node));
new_node->data = 21;
new_node->next = head;
```

Connect 2 nodes: Add new node to the start



```
head = new_node;
```

Coding Time

`linked_list_intro.c`

Create a list with 3 nodes

Print the contents of the first 3 nodes in the list

Coding Time

`list_list_functions.c`

- How can we put our code to create a new node into a function?
- How could we use that to create a list by adding each node to head using a loop?
- How would we print the whole list? Even if it had 1000s of nodes?
- How could we add nodes to the end of the list? Even if it had 1000s of nodes?
- We want a function to free all nodes too. But let’s leave that until another lecture…

Create Node Function

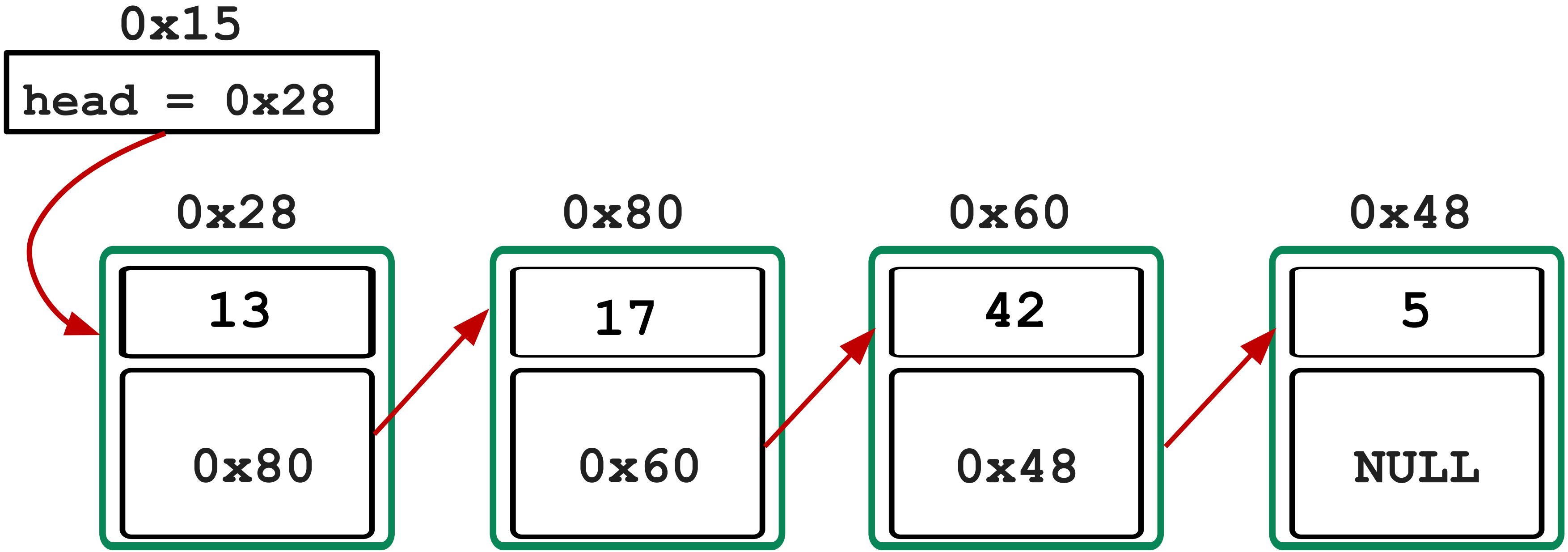
```c
// Creates and returns a new node with given data and
// next pointer. returns NULL if memory allocation fails.
struct node *create_node(int data, struct node *next){
    struct node *new_node = malloc(sizeof(struct node));
    if (new_node == NULL) {
        return NULL;
    }
    new_node->data = data;
    new_node->next = next;
    return new_node;
}
```

Creating a Linked List Inserting at Head

```c
// What would the contents of our list be?
int main(void) {
    struct node *head = NULL;
    for(int i = 0; i < 10; i++) {
        struct node *new_node = create_node(i, head);
        head = new_node;
    }
    return 0;
}
```

Printing a Node

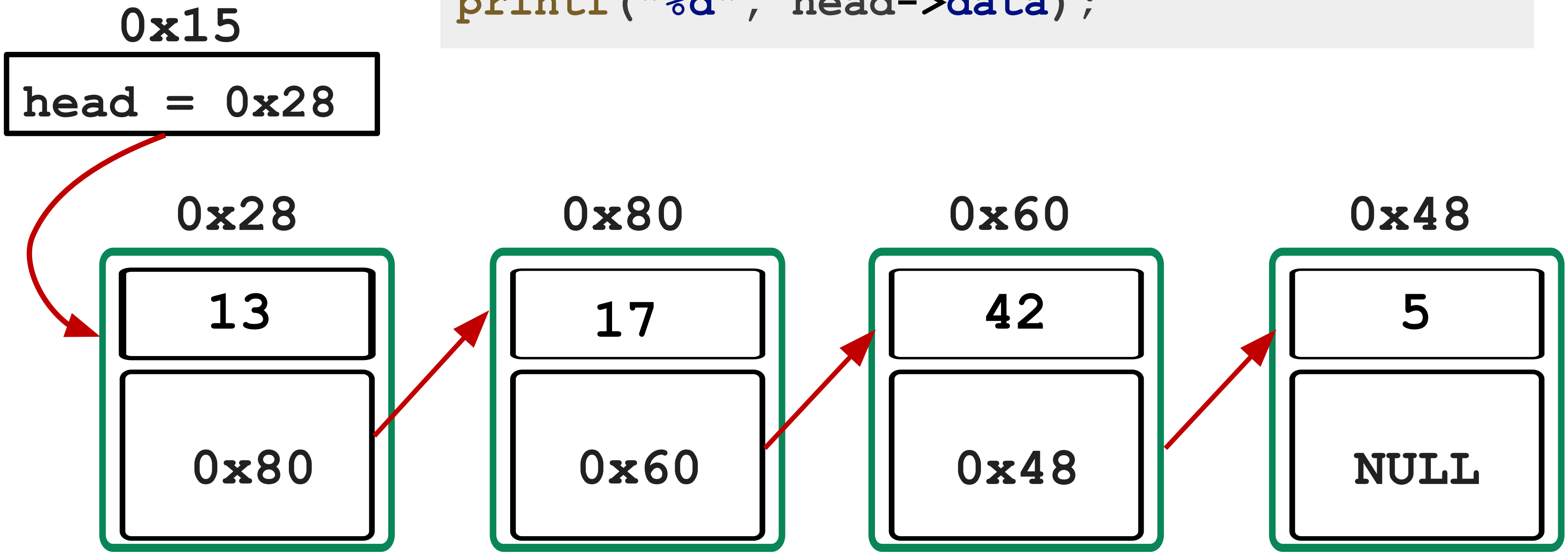
How could I print the data from the first node in this linked list?



Printing a Node

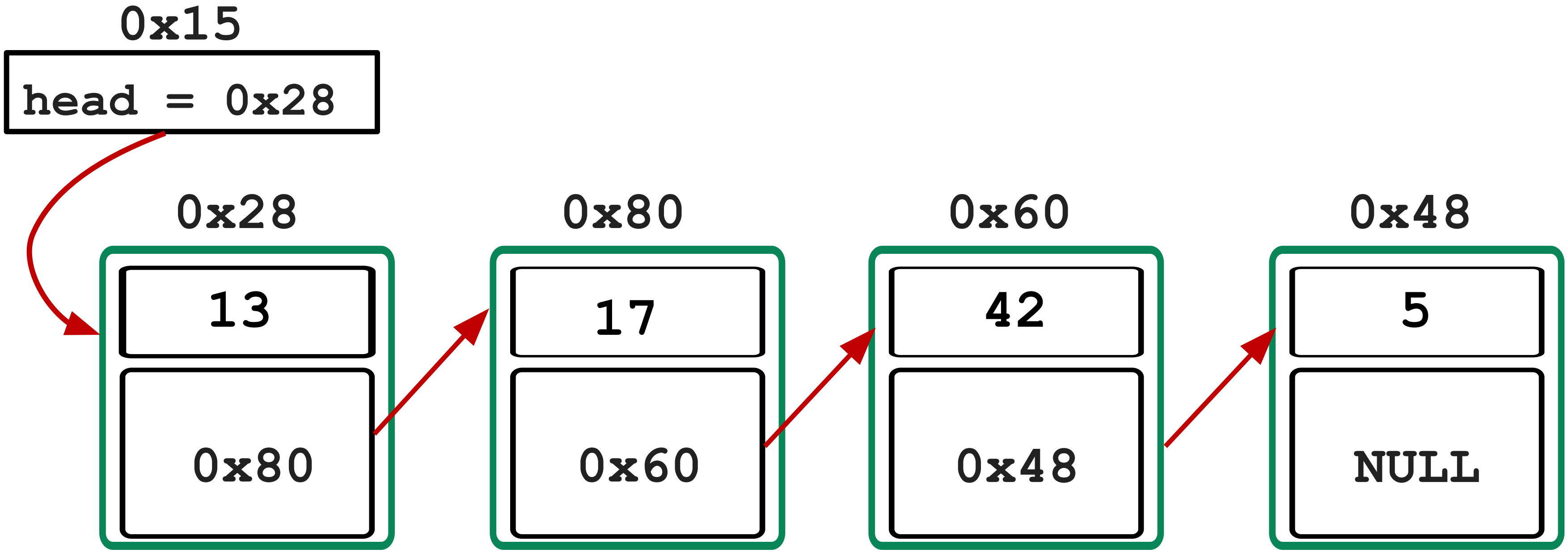
How could I print the data from the first node in this linked list?

```
printf("%d", head->data);
```



Printing a Linked Lists

How could I print data from **each** node in this linked list?



Traversing a list

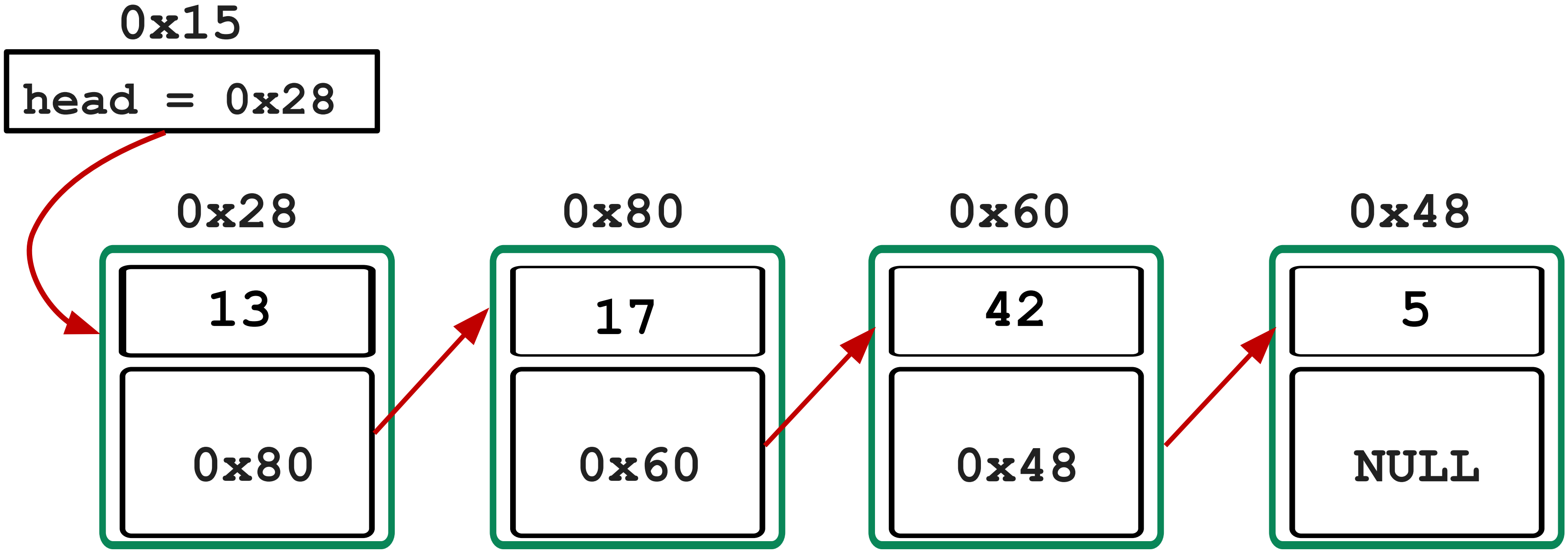
Traversing a list means

- starting at the head of the list
- moving node by node until we get to the end of the list.

We often want to traverse a list, node by node to do things like

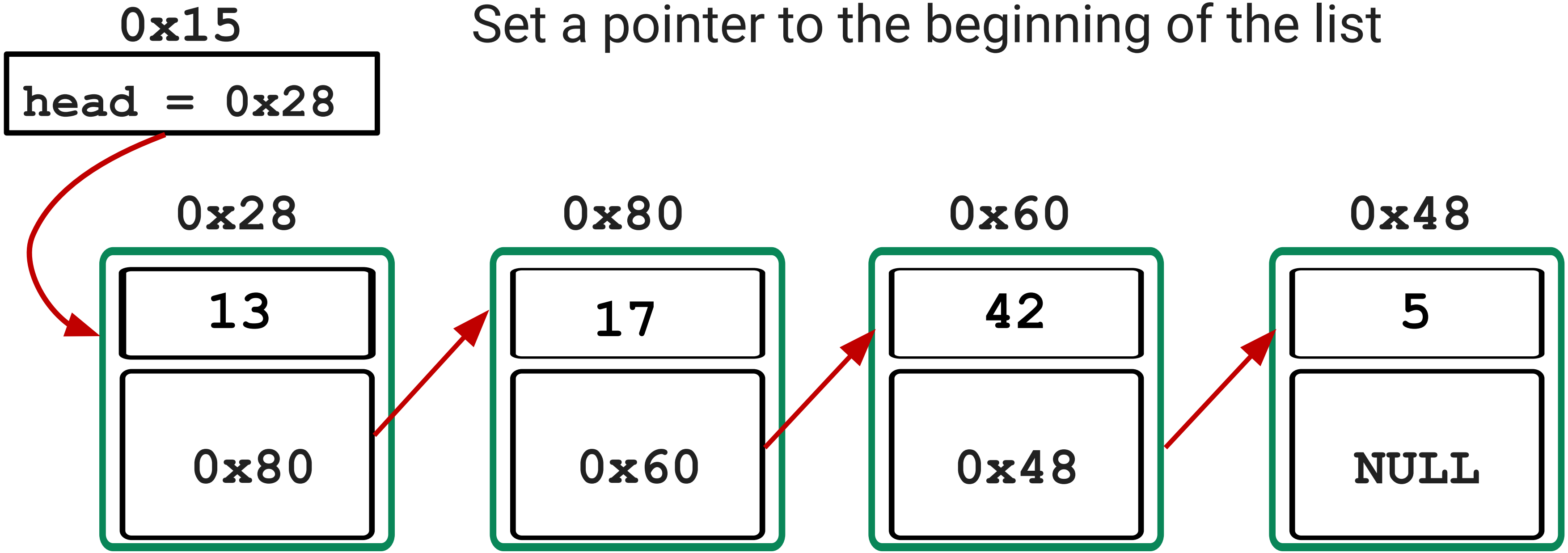
- print the data in each node in the list
- count the number of nodes in the list
- search for data in the list

Traversing a List



Traversing a List

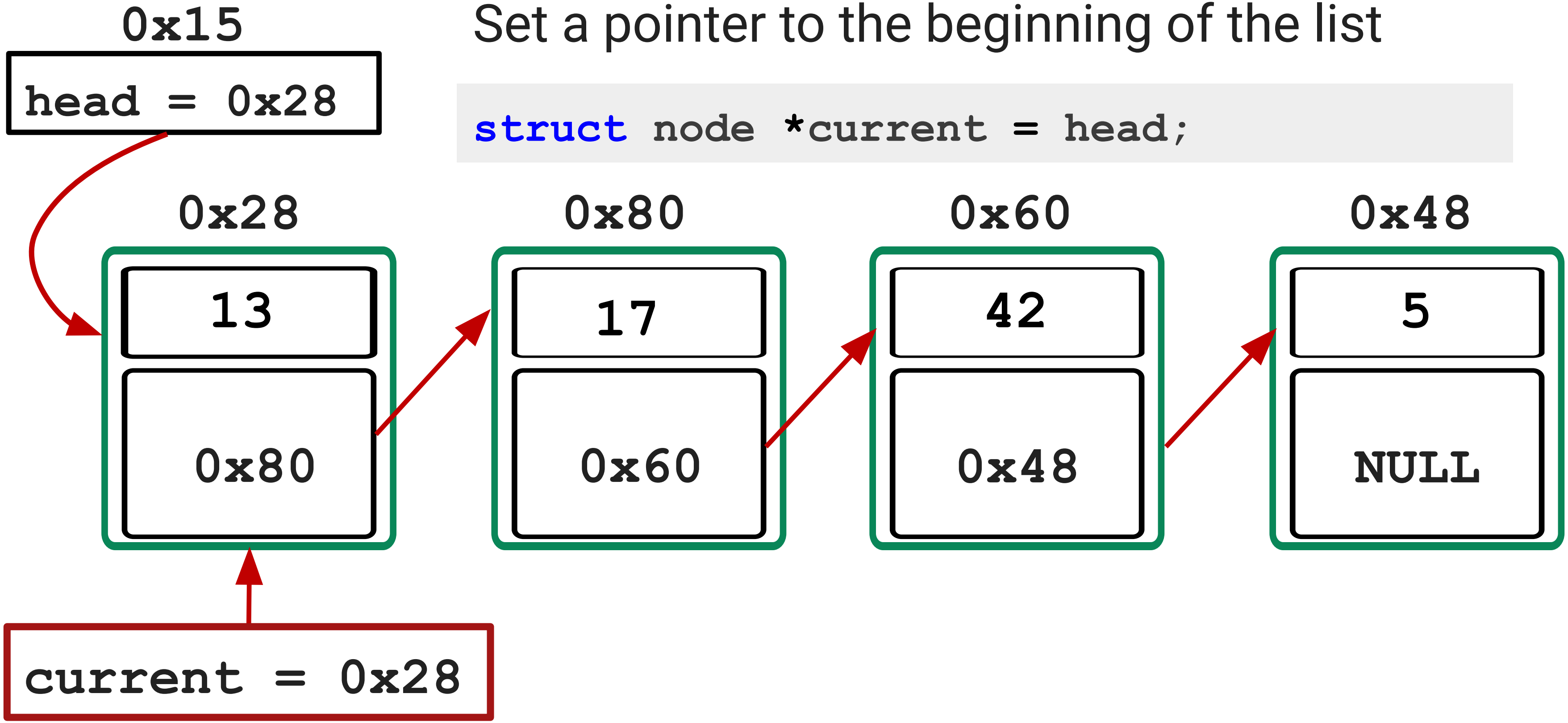
Set a pointer to the beginning of the list



Traversing a List

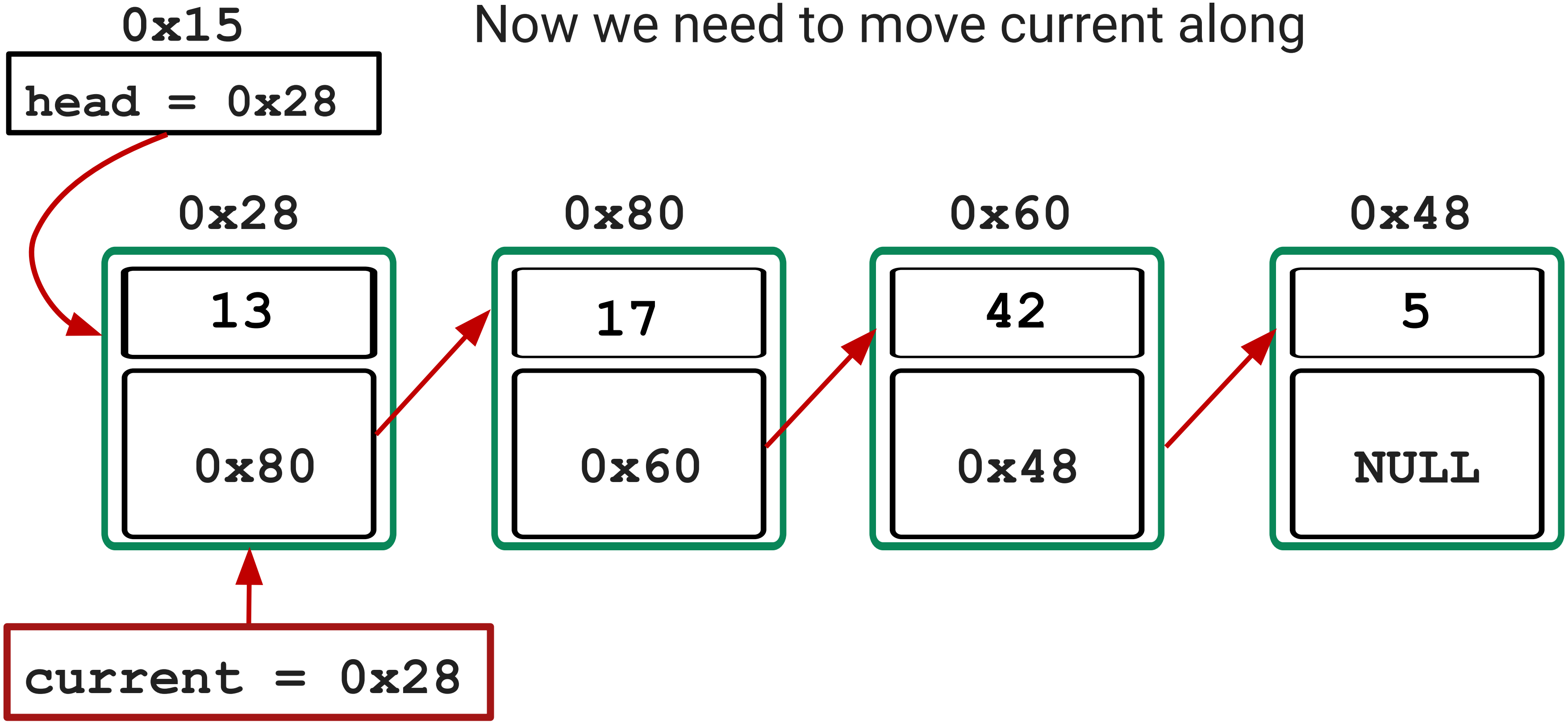
Set a pointer to the beginning of the list

```
struct node *current = head;
```



Traversing a List

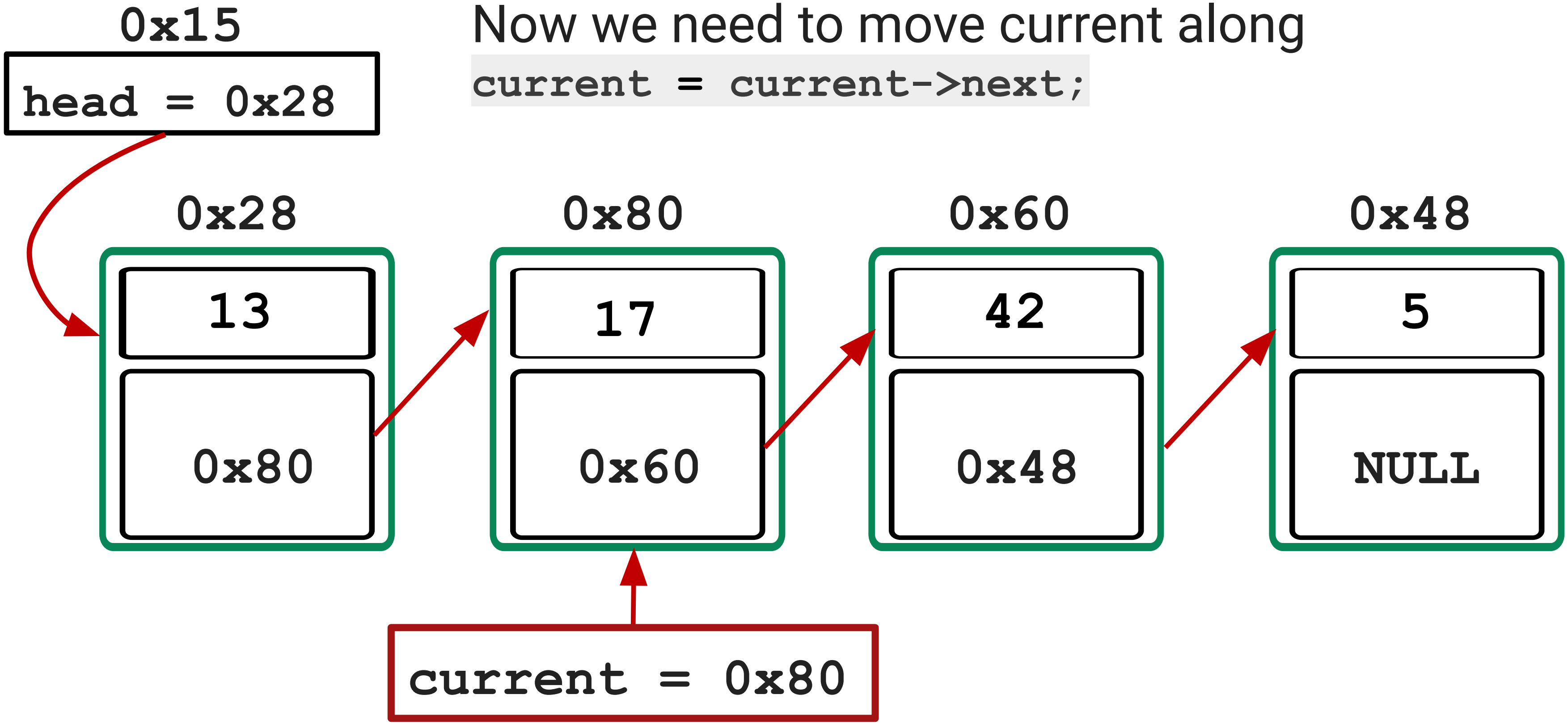
Now we need to move current along



Traversing a List

Now we need to move current along

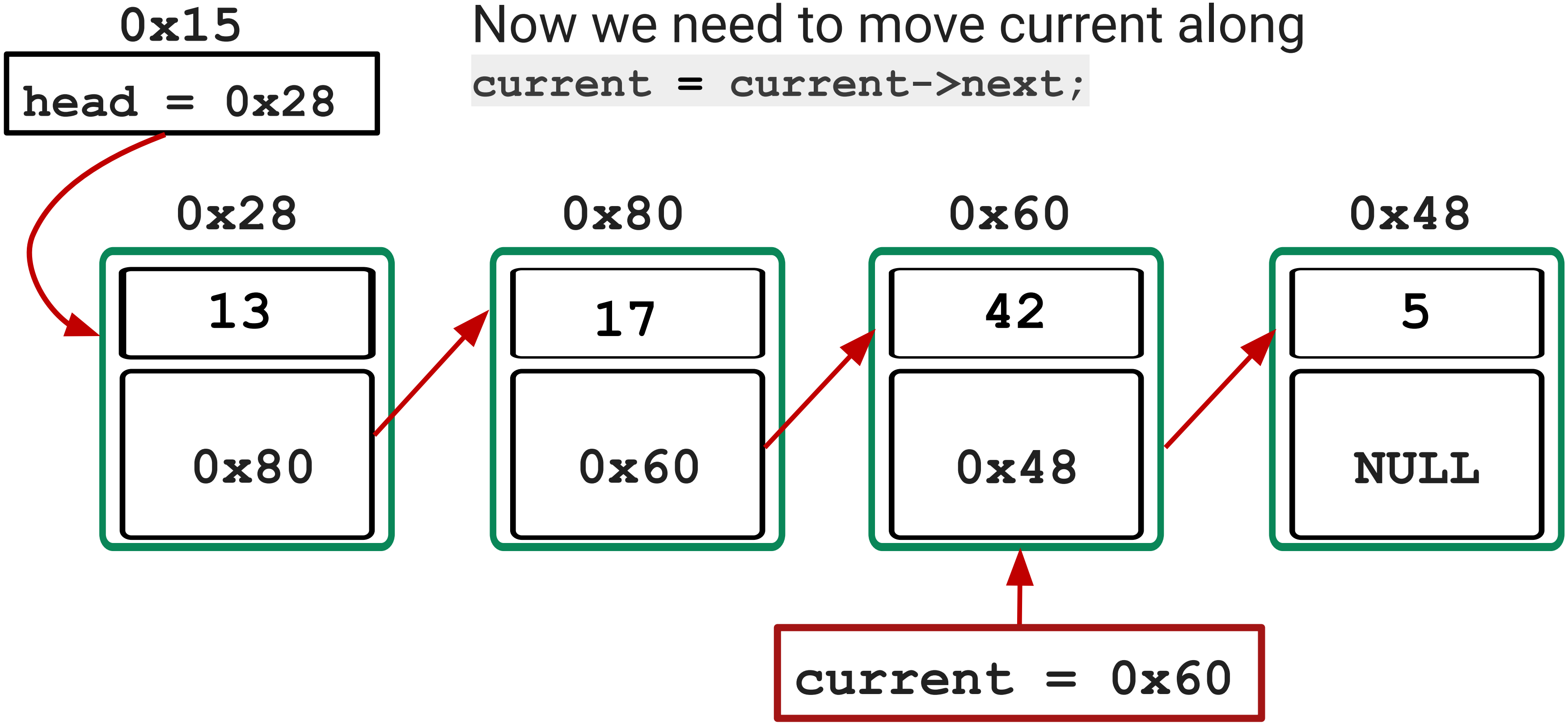
```
current = current->next;
```



Traversing a List

Now we need to move current along

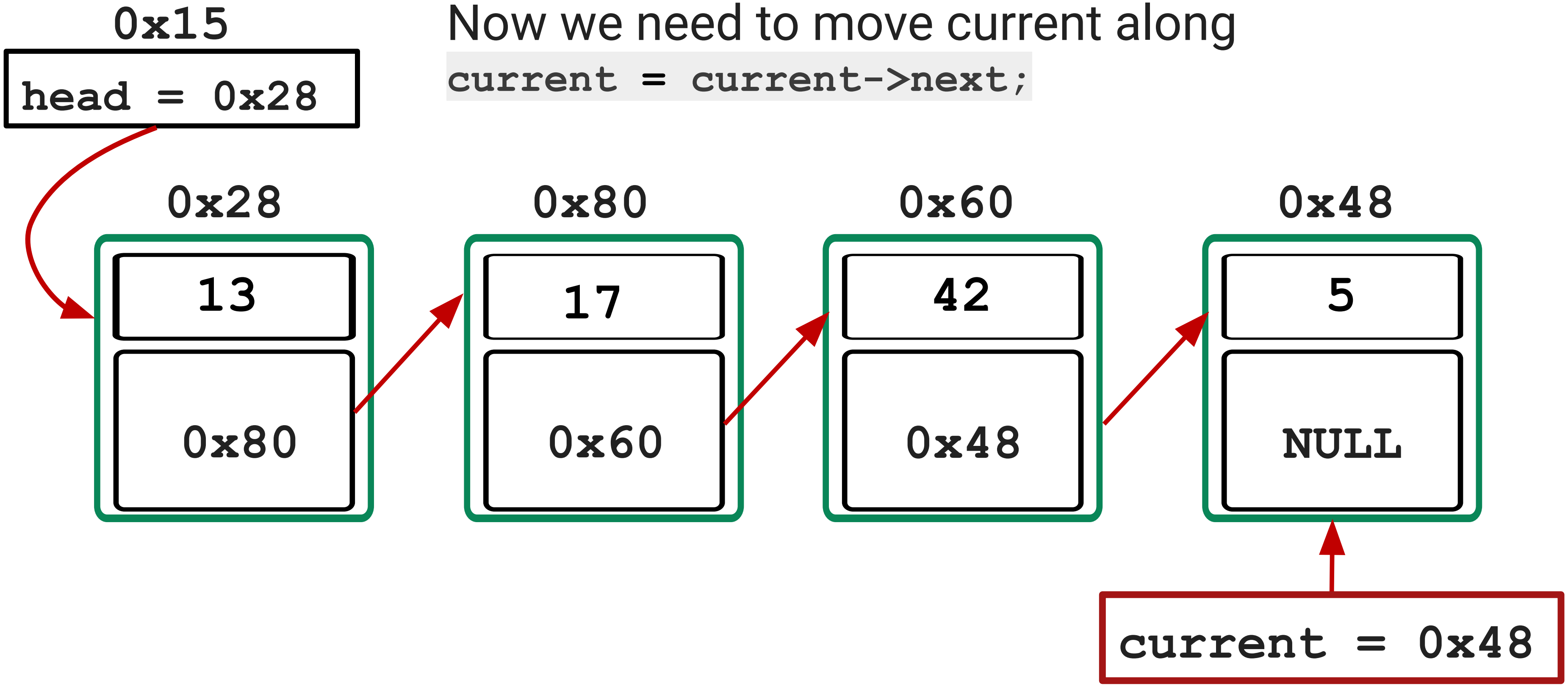
```
current = current->next;
```



Traversing a List

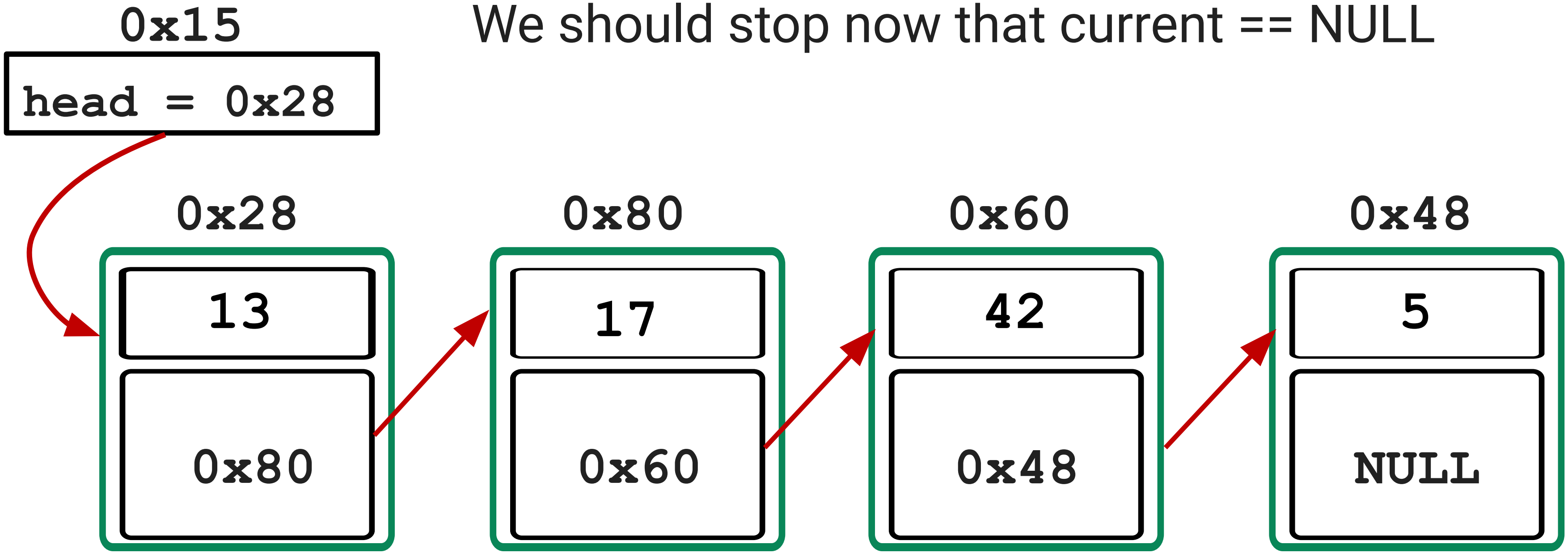
Now we need to move current along

```
current = current->next;
```



Traversing a List

We should stop now that current == NULL



current = NULL

Printing a list

```c
// Traversing the list and printing the contents (data)
// from each node
void print_list(struct node *head) {
    struct node *current = head;
    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }
    printf("\n");
}
```

Inserting at the tail (end) of a list

Where can I insert in a linked list?

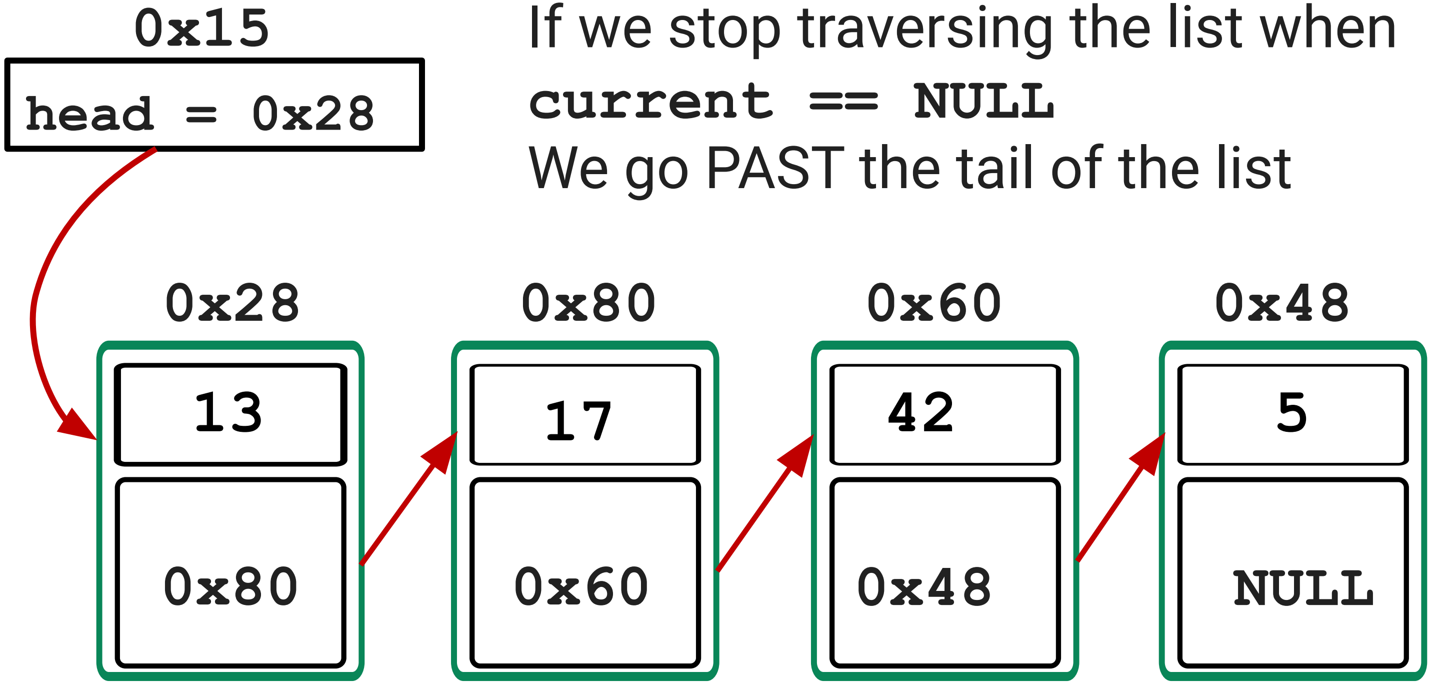
- At the head (what we just did!)
- Between any two nodes that exist (next lecture!)
- After the tail as the last node (now!)

To insert a node at the end of the list we need to

- Find the last node in the list
- Connect the last node in the list to the new node

Finding the Tail of the list

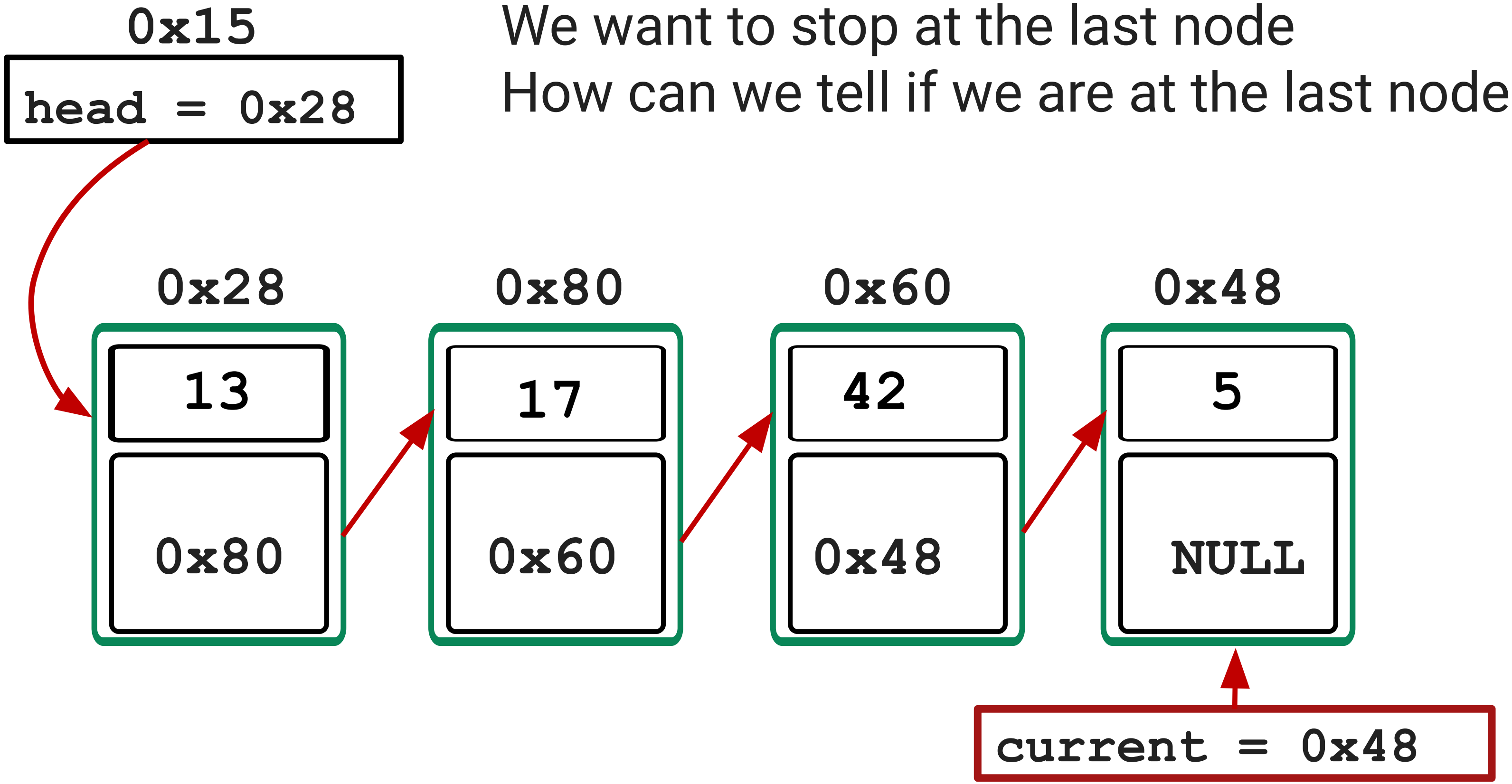
If we stop traversing the list when `current == NULL` We go PAST the tail of the list



current = NULL

Finding the Tail of the list

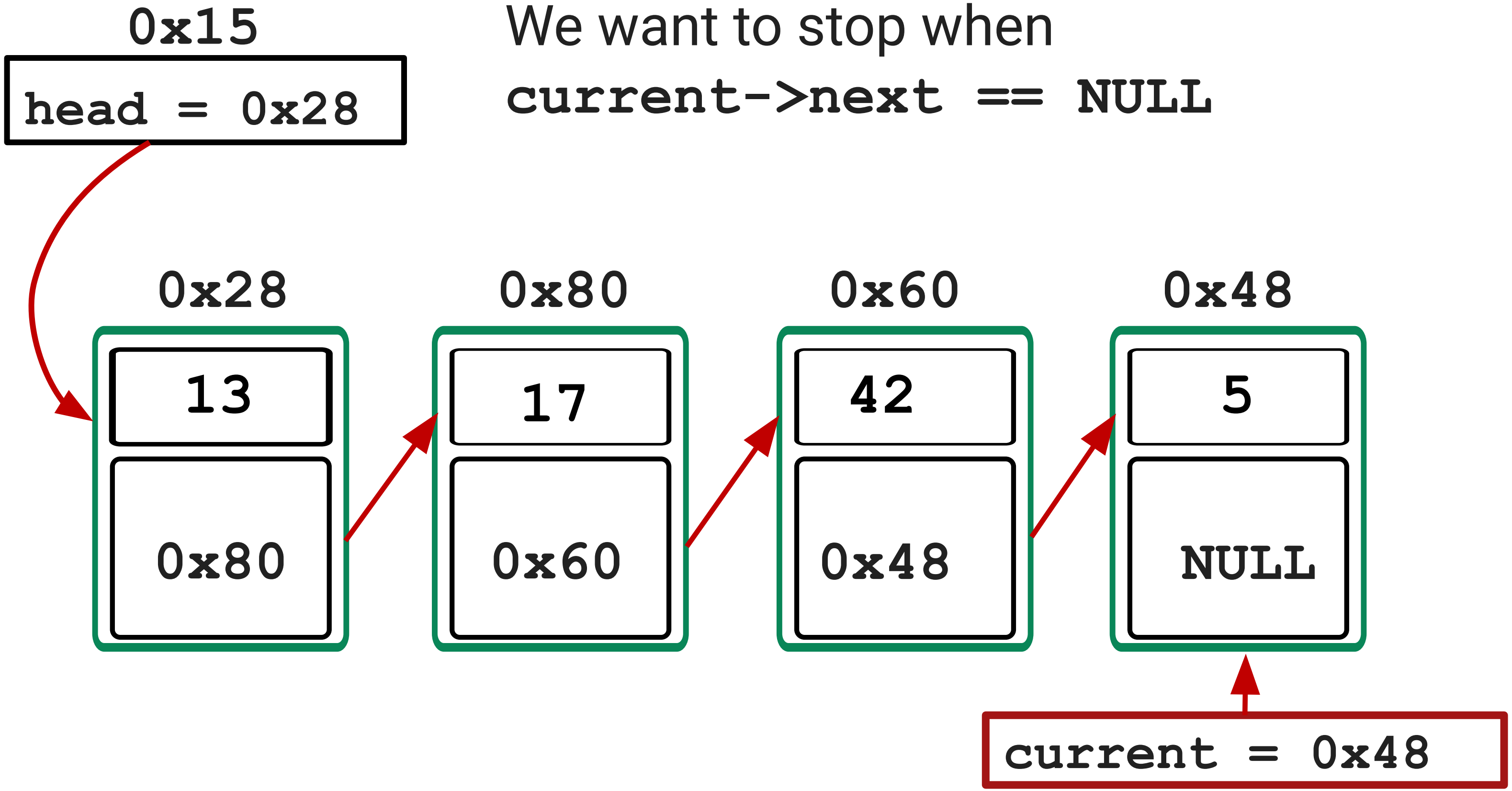
We want to stop at the last node
How can we tell if we are at the last node?



Finding the Tail of the list

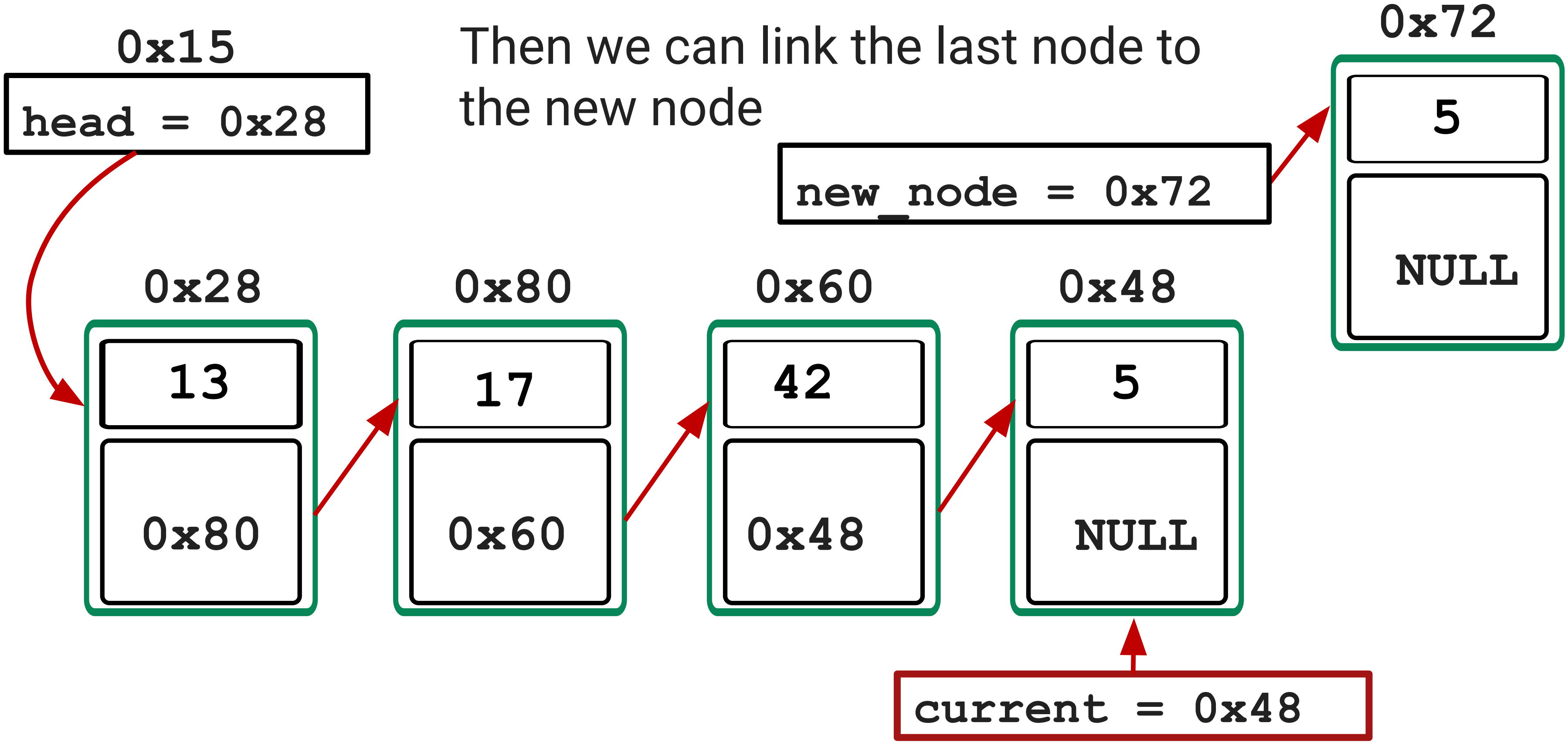
We want to stop when

`current->next == NULL`



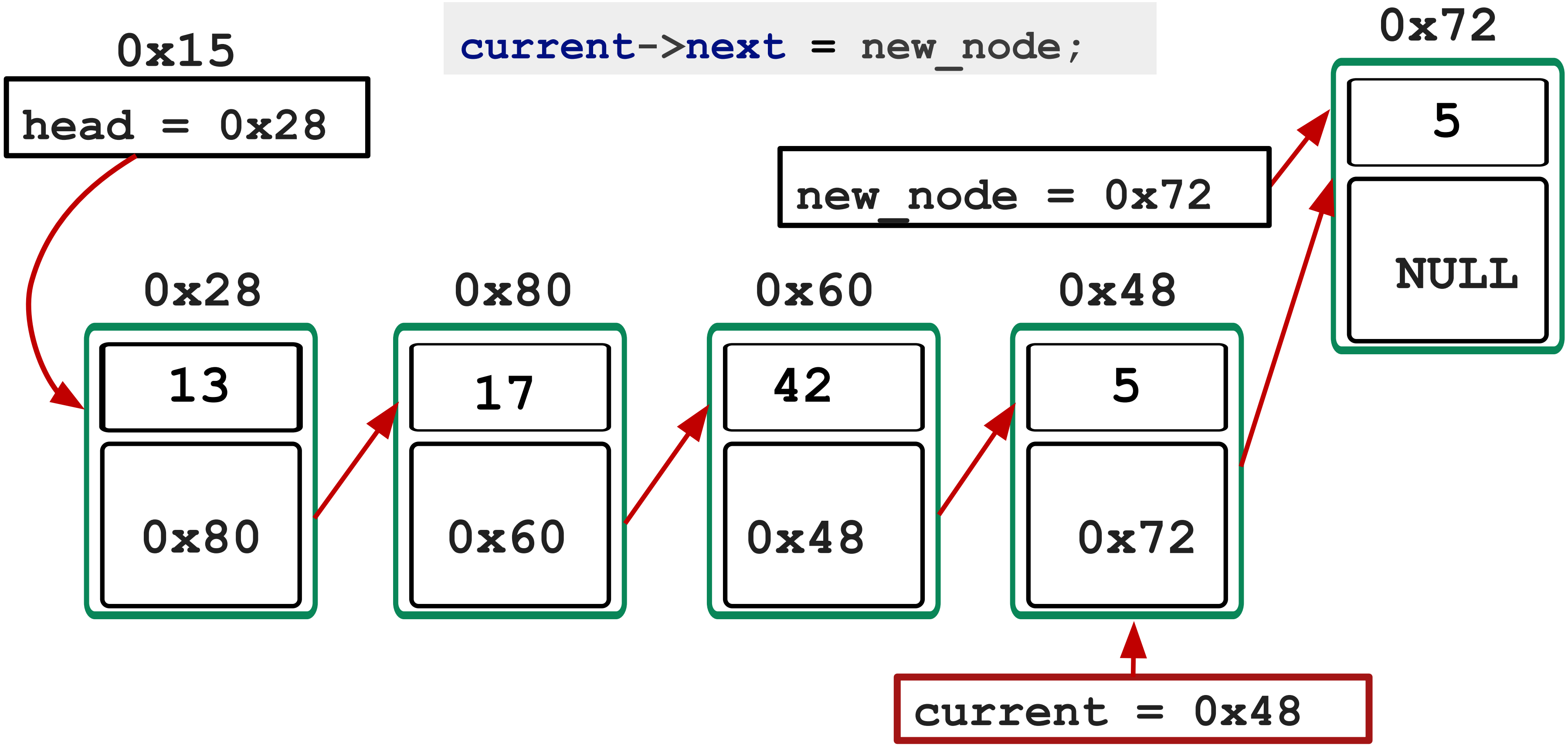
Insert at the Tail of the list

Then we can link the last node to the new node



Insert at the Tail of the list

```
current->next = new_node;
```



Inserting at Tail (with a big bug)

```
// What valid input could cause this function to break?
void insert_at_tail(struct node *head, int data){
    struct node *current = head;
    // Find the tail of the list
    while (current->next != NULL) {
        current = current->next;
    }
    // Connect new node to the tail of the list
    struct node *new_node = create_node(data, NULL);
    current->next = new_node;
 }
```

Linked List Test Cases

It is always important to test your linked list functions with:

- An empty list
- A list with one node
- A list with more than one node

Our function only inserts at the end of the list. If we were writing a function to insert anywhere into a list we would want to test

- Inserting at the beginning
- Inserting in the middle
- Inserting at the end

Inserting At Tail Code Bug

If we have an empty list

- head == NULL;
- so then current == NULL;
- so `current->next`
  will be dereferencing a NULL pointer and result in a run time error

```
void insert_at_tail(struct node *head, int data){
    struct node *current = head;
    // Find the tail of the list
    while (current->next != NULL) {
```

Inserting at Tail (still with a bug)

```
void insert_at_tail(struct node *head, int data){
    struct node *new_node = create_node(data, NULL);
    if (head == NULL) { // Special case for empty list
        head = new_node;
    } else {
        struct node *current = head;
        // Find the tail of the list
        while (current->next != NULL) {
            current = current->next;
        }
        // Connect new node to the tail of the list
        current->next = new_node;
    }
 }
```

Inserting At Tail Code Bug

The code no longer crashes!!!

But we still end up with an empty list when we use the function.

Why?

```
int main(void) {
    struct node *head = NULL;
    insert_at_tail(head, 9);
    // local variable head is in main is still NULL
    return 0;
}
```

Fixing Inserting at Tail Code

We need to modify the prototype so it can return the head of the list and we need to assign that return value to our local variable.

```c
struct node *insert_at_tail(struct node *head, int data);
int main(void) {
    struct node *head = NULL;
   // local variable head has been updated :)
    head = insert_at_tail(head, 9);
    return 0;
 }
```

Inserting at Tail

```
struct node *insert_at_tail(struct node *head, int data){
    struct node *new_node = create_node(data, NULL);
    if (head == NULL) { // Special case for empty list
        head = new_node;
    } else {
        struct node *current = head;
        // Find the tail of the list
        while (current->next != NULL) {
            current = current->next;
        }
        // Connect new node to the tail of the list
        current->next = new_node;
    }
    return head;
 }
```

What did we learn today?

- Recap dynamic memory, malloc (malloc_struct.c)
- Linked Lists Intro (linked_list_intro.c)
- Inserting nodes at the start of the list (linked_list_functions.c)
- Traversing a List
- Inserting an item at the tail of a list

Next lecture:

- Inserting an element anywhere in the list!
- Deleting an element
- Lists containing other types of data

Feedback Please!

Your feedback is valuable!

If you have any feedback from today's lecture, please follow the link below or use the QR Code.

Please remember to keep your feedback constructive, so I can action it and improve your learning experience.

https://forms.office.com/r/nTz8Wkd0vB

Reach Out

Content Related Questions:
Forum

Admin related Questions email:
cs1511@unsw.edu.au

Don’t forget to attend Help Sessions

And Revision sessions if needed