Linked Lists...

An introduction to creating dynamic data structures

Linked Lists definition

- Example of the common use of "list":
  - TODO:
    - Read task description
    - Design test data
    - Create makefile
    - Collect other files needed
    - Begin working on code
    - Type in first version
    - Test it
    - …

Linked Lists definition

- More formal definition:

  <Llist> ::== Nothing | element <Llist>

- Examples
  - [nothing]
  - element
  - element element
  - element element element
- Example where “element” is an integer
  - [nothing]
  - 16
  - 12, 15, 19, -22, 0, 54

Linked Lists and pointers

- The word "list" suggests an ordered collection
- The word "linked" hints at *pointers* to where the next element in the list is located
  - This is different from arrays, which use contiguous memory locations
  - Linked lists may use multiple links to link elements according to different interpretations of "order".
  - We will only consider "singly linked lists" for now.

Linked Lists memory diagrams

- A list of the numbers [0, 12, -4, 22]
- An array of the numbers [0, 12, -4, 22]

- Where are arrays stored?
  - Global/static/extern/const?
  - Stack?
  - Heap?
- Where are linked lists stored?

Linked Lists: Internals

- The dynamic nature of the linked list data structure means we must allocate and free memory on demand

- Some languages do this for you.

- Not so for C
  - you have to do it explicitly.

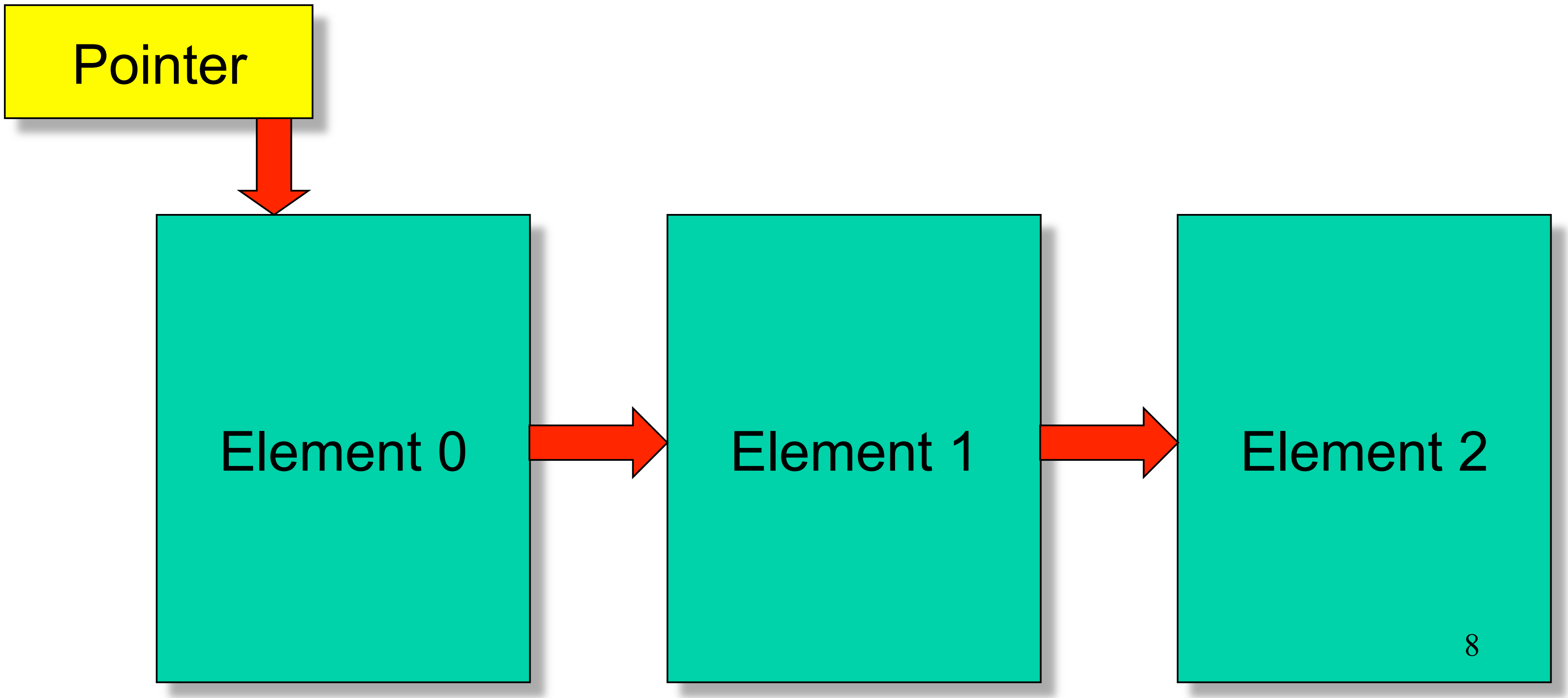
Linked Lists: Internals

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

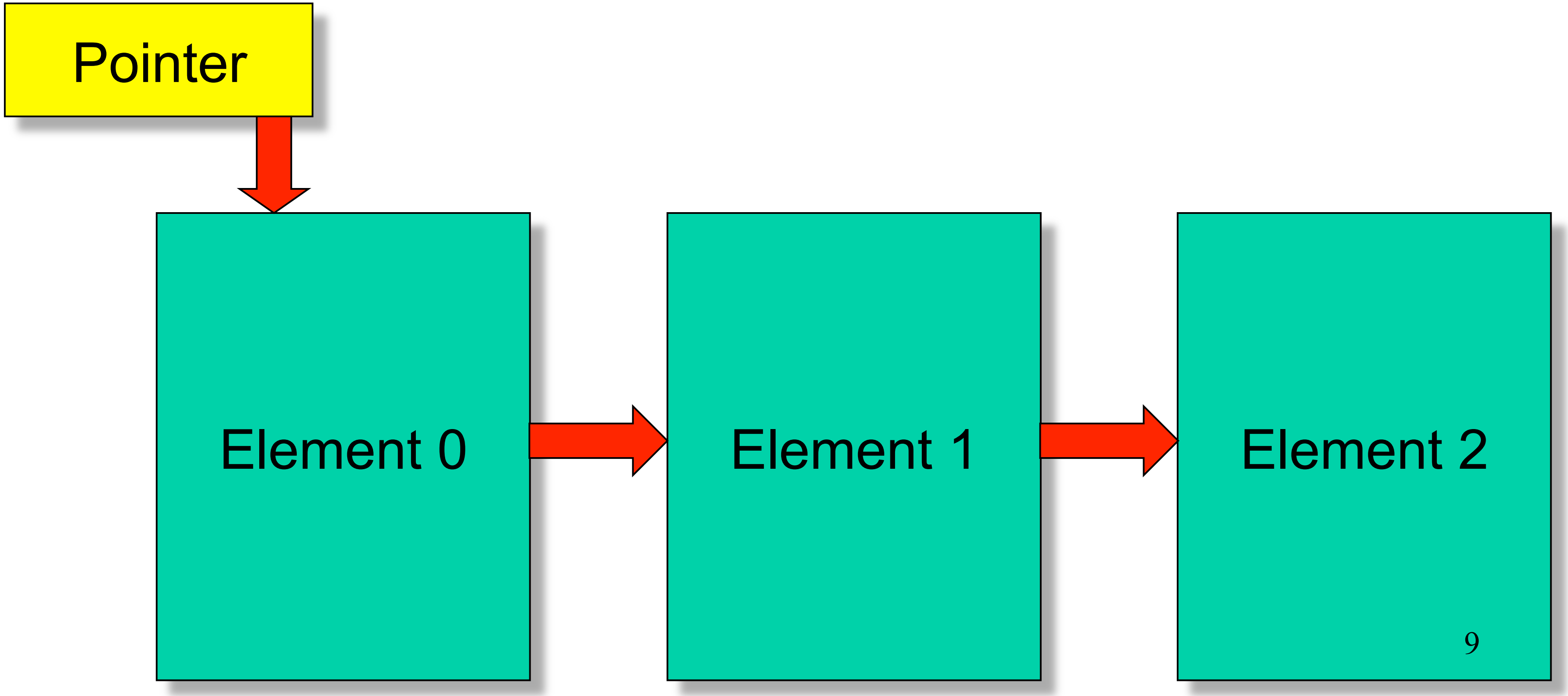
Linked Lists: Internals

Here is a high-level schematic of a linked list.

A pointer enables us to reference its 1st element.



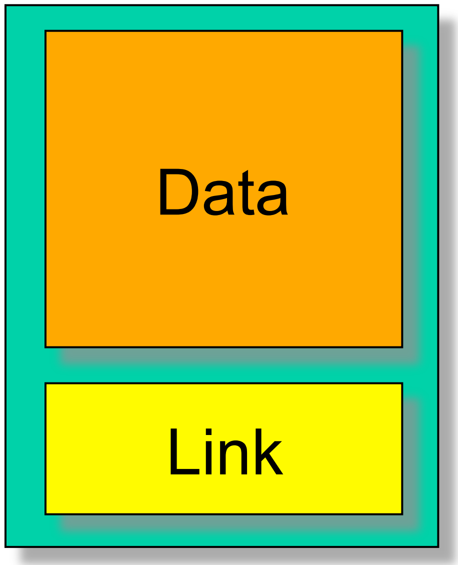
Three distinctive parts of the list - what are they?



Linked Lists: Internals

Each ***element*** in our singly-linked list has two components:

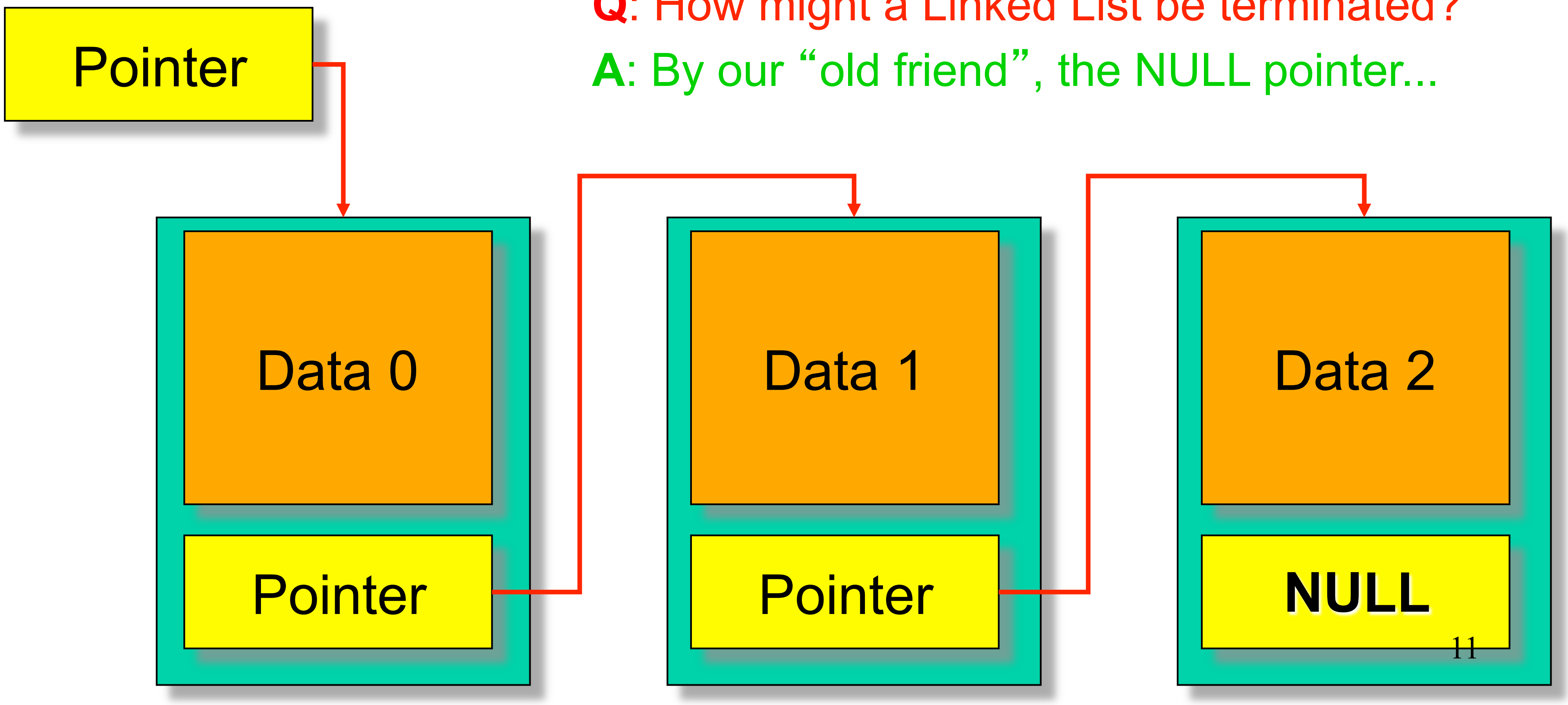
- the data component
  - anything you want...
- the link component
  - a pointer, of course!



Linked Lists: Internals

Q: How might a Linked List be terminated?

A: By our "old friend", the NULL pointer...



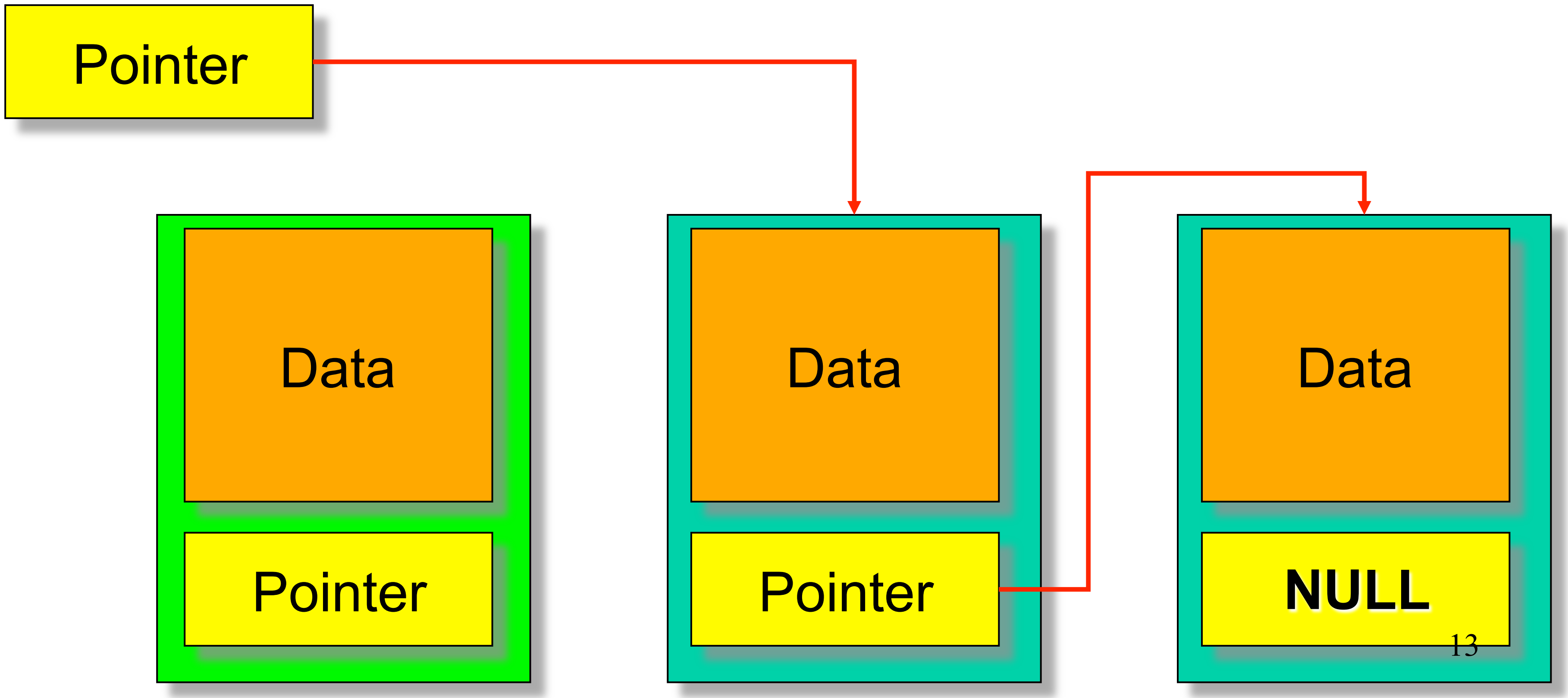
Linked Lists: Internals

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

struct node n;
n.data = 0;
n.next = NULL;
```

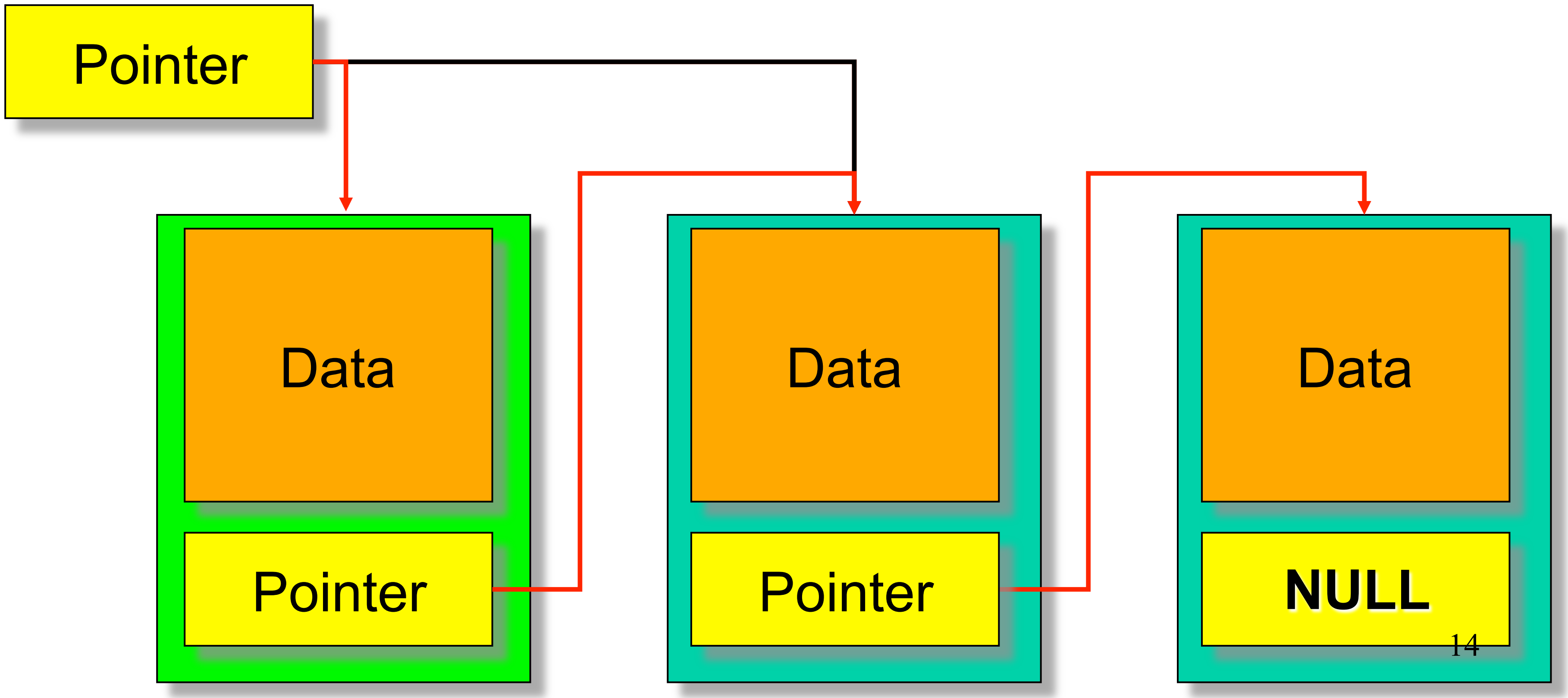
Linked Lists: Internals

Adding an element to the front of the list.



Linked Lists: Internals

Adding an element to the front of the list.

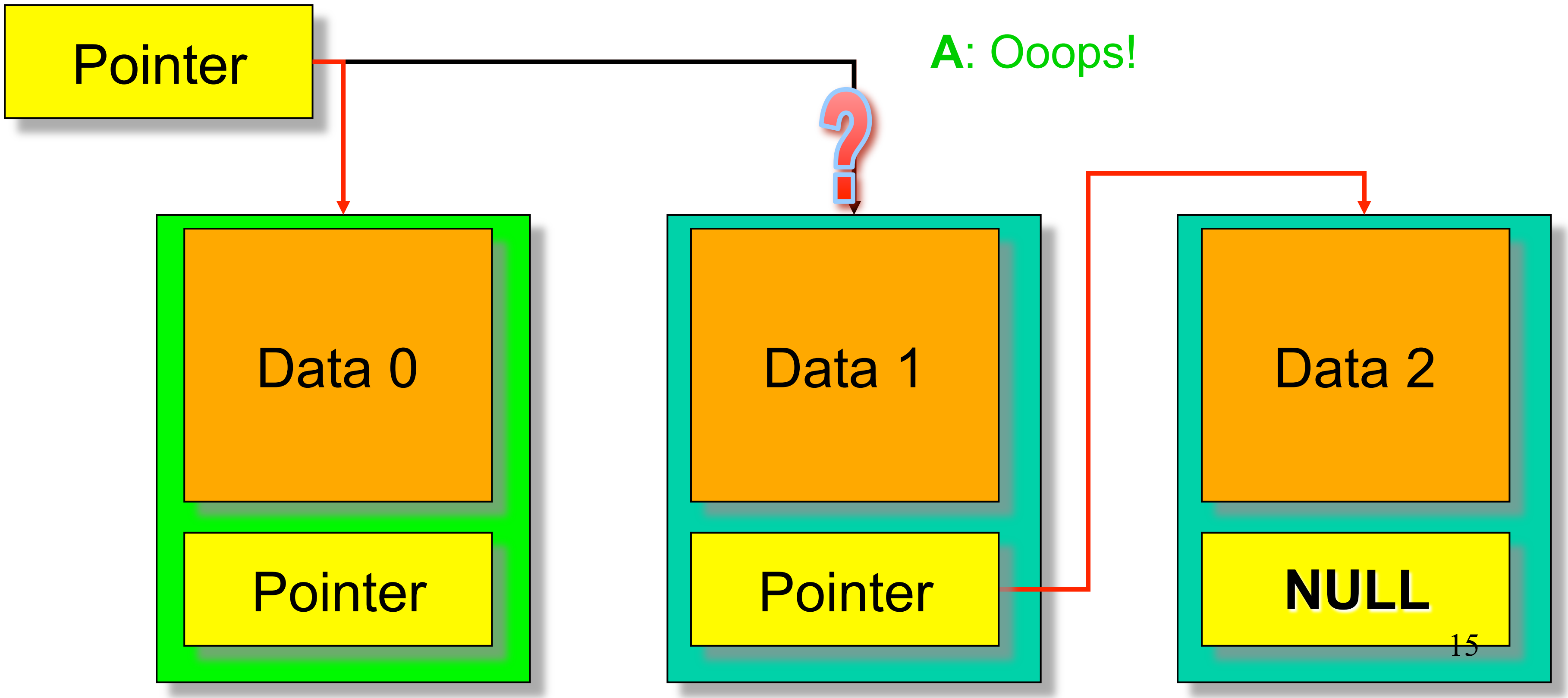


Linked Lists: Internals

New links are forged by simple assignment operations.

Q: What happens if we change the order?

A: Ooops!



Linked Lists: Internals

Using “boxes and arrows” to show what we are doing with pointers is very helpful…

But it is easy to lose sight of one simple thing:

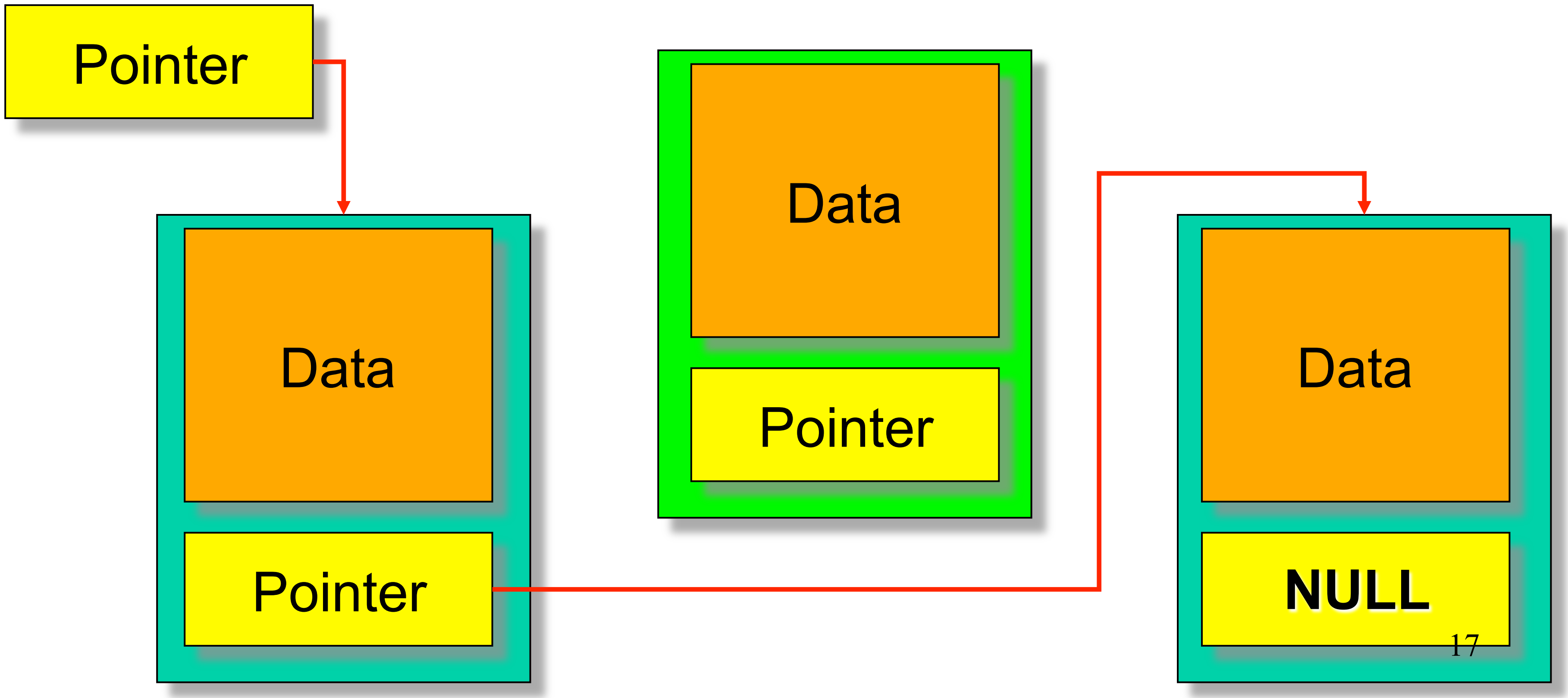
One pointer can only point to one thing!

Every time we “draw an arrow” we effectively erase any existing arrow coming from that box.

In the previous example we “lost” the linked list because we neglected this!

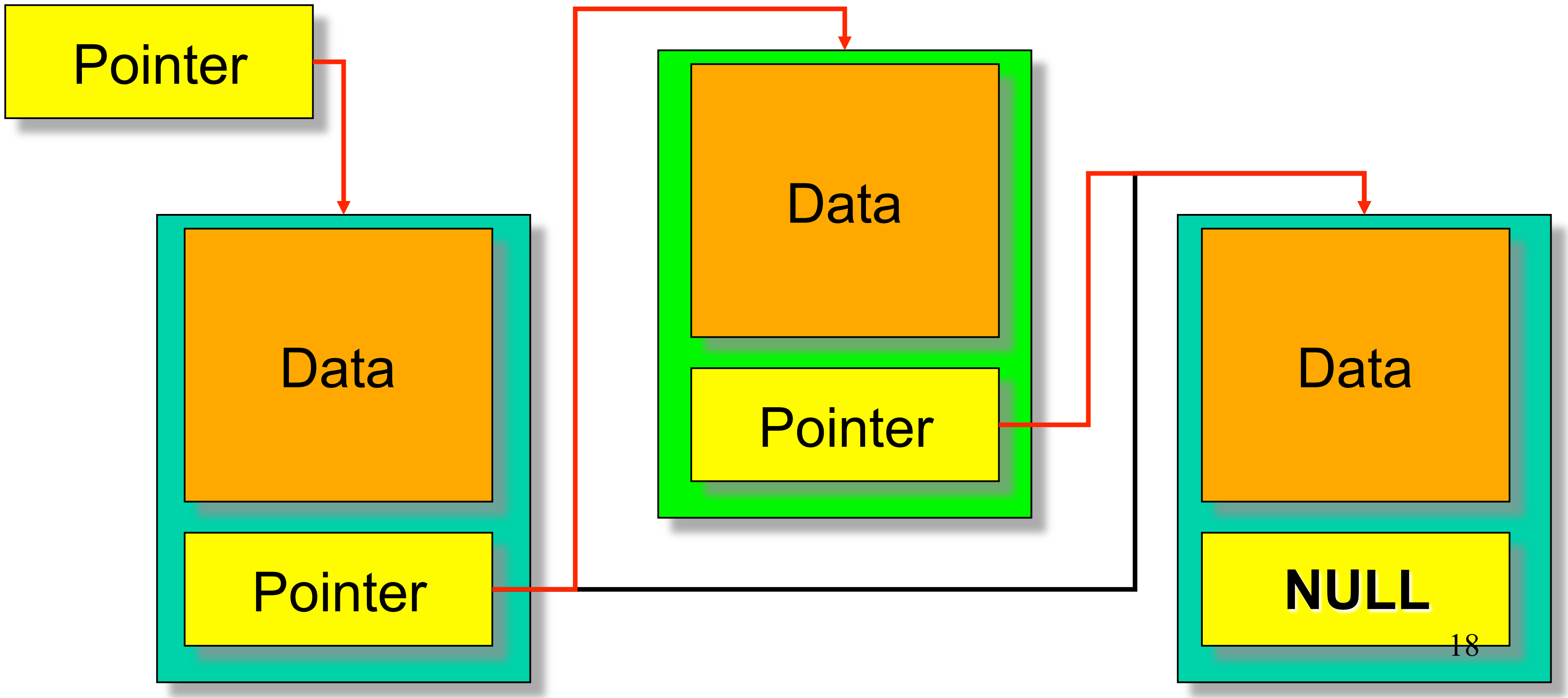
Linked Lists: Internals

Adding an element elsewhere.

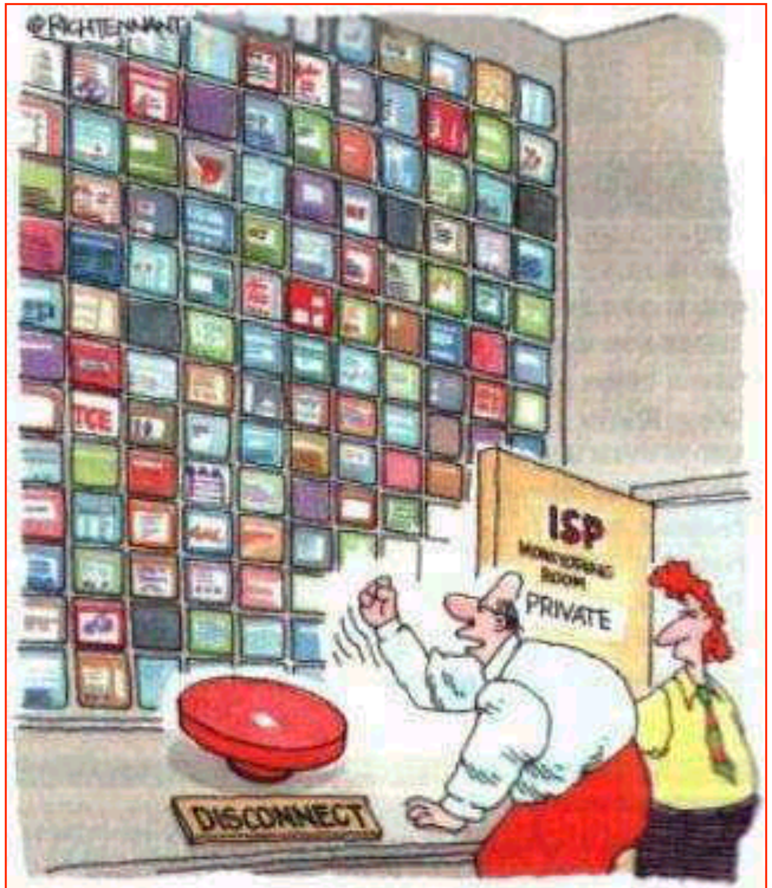


Linked Lists: Internals

Adding an element elsewhere.



Characteristics of Linked Lists



"I'll be right there. Let me just take care of this user. He's about halfway through a 3-hour download."

Characteristics of Linked Lists

What is the worst-case situation for altering the n-th element of:

– an array?

Altering an element anywhere in the array has the same cost. Arrays elements are referred to by their address offset and have O(1) cost.

– a linked list?

Altering an element to the end of the linked list is more costly as the list has to be traversed. The cost is therefore O(n).

Characteristics of Linked Lists

Advantages of linked lists:

- Dynamic nature of the data structure
- Flexible structure
  - singly linked lists
  - doubly linked
  - circular lists
  - etc

Characteristics of Linked Lists

Disadvantages of linked lists:

- Linear worst-case cost of access

- The absence of a Linked List implementation in standard *C* libraries

Characteristics of Linked Lists

Disadvantages of linked lists:

- Linear worst-case cost of access
  - Skip lists
- The absence of a Linked List implementation in standard *C* libraries
  - Build your own

Programming Linked Lists

The dynamic nature of the data structure means we must allocate and free memory

`malloc()`

- to allocate memory when we add an element

`free()`

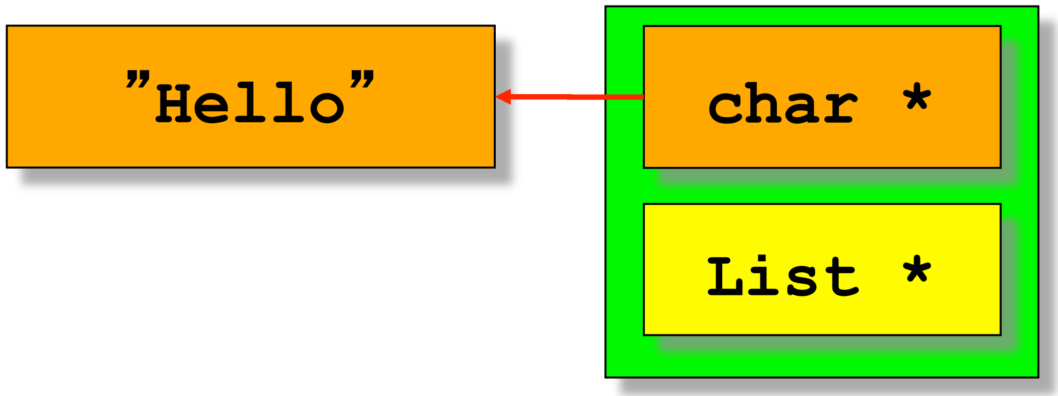
- to de-allocate memory when we remove an element

Programming Linked Lists

The element we will use for the example:

```c
typedef struct List List;

struct List {
  char *content;
  List *next;
}
```



Could be **anything**, of course!!

Note the self-referential nature of the pointer; it points to one of these structures.

Programming Linked Lists

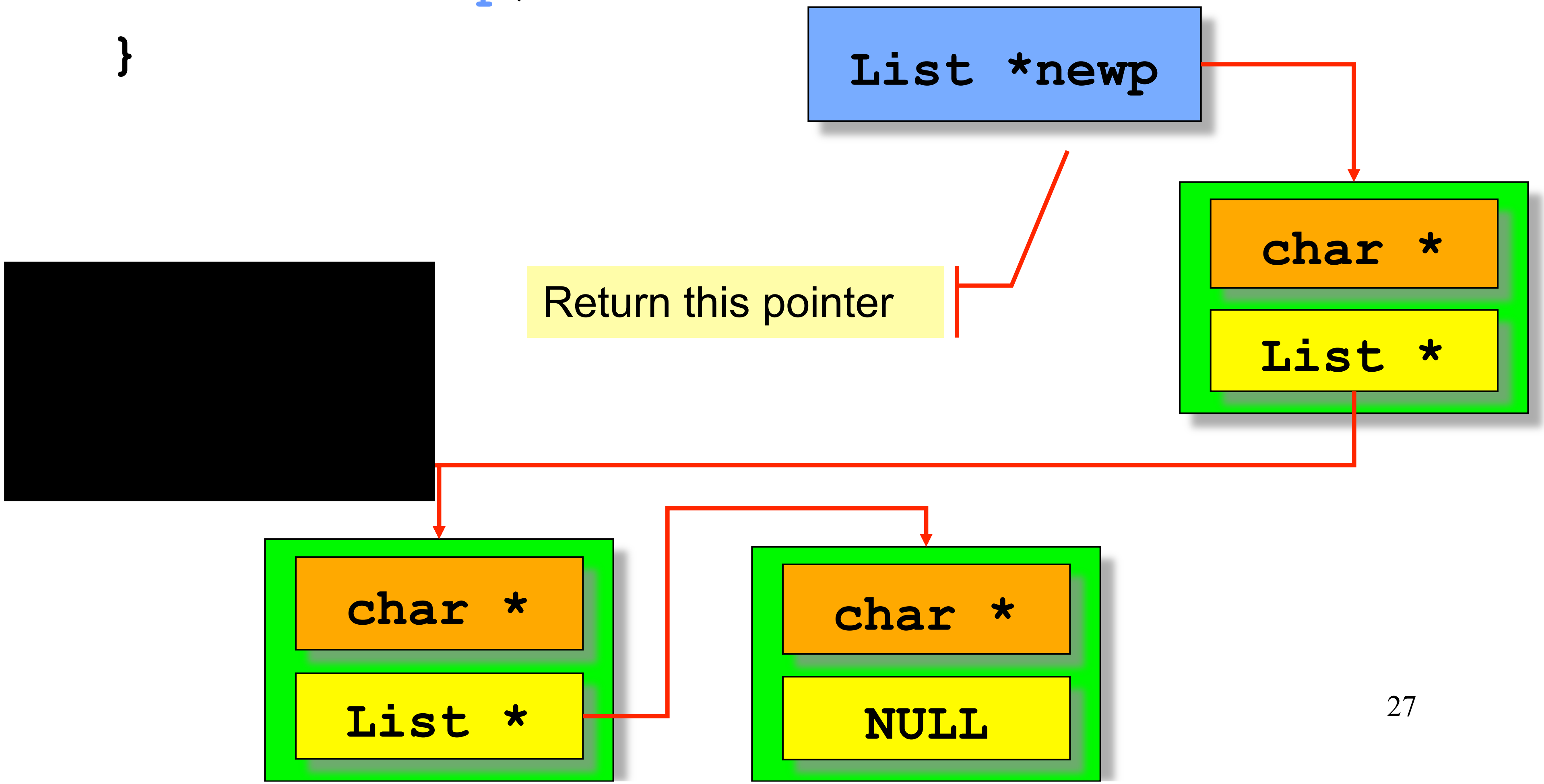
Adding an element to the front of the list.

```
List *addToFront(List *listp, List *newp) {
  newp->next = listp;
  return newp;
}
```

Draw memory diagrams for:

empty list; list with one element; list with three elements.

```
List *addToFront(List *listp, List *newp) {
  newp->next = listp;
  return newp;
}
```



Programming Linked Lists

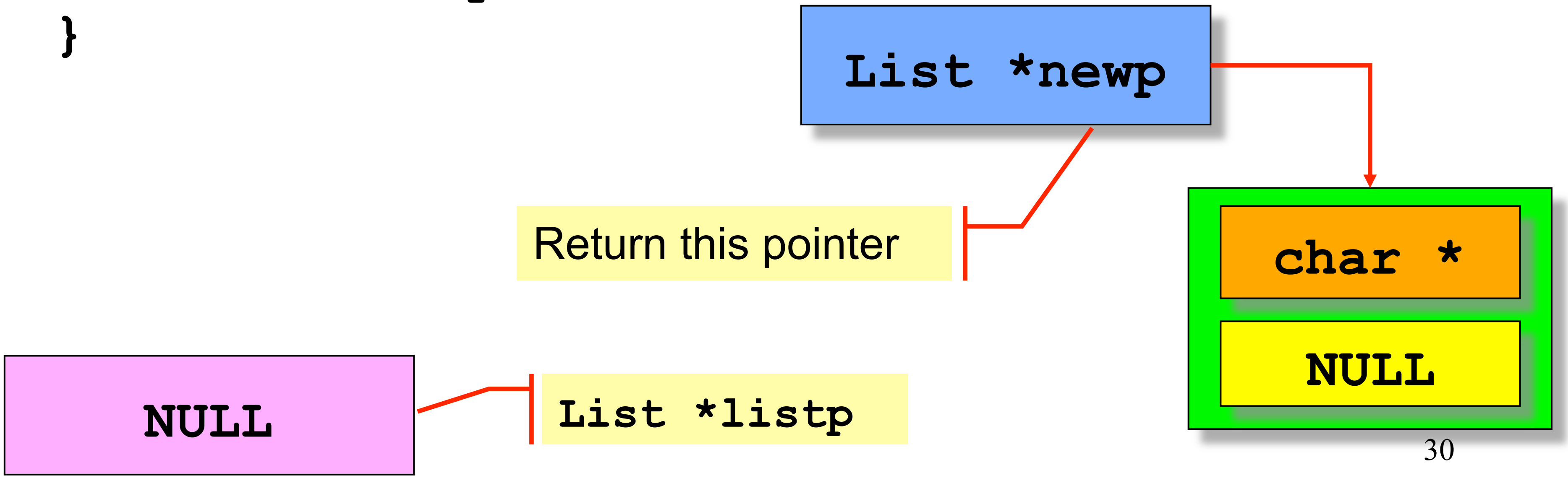
Adding an element to the end of the list.

```
List *addToEnd(List *listp, List *newp) {
  List *p;
  if (listp == NULL)
     return newp;
  for (p = listp; p->next != NULL; p = p->next)
     ;           /* null statement */
  p->next = newp;
  return listp;
}
```

Programming Linked Lists

Draw pictures of case of empty list *listp

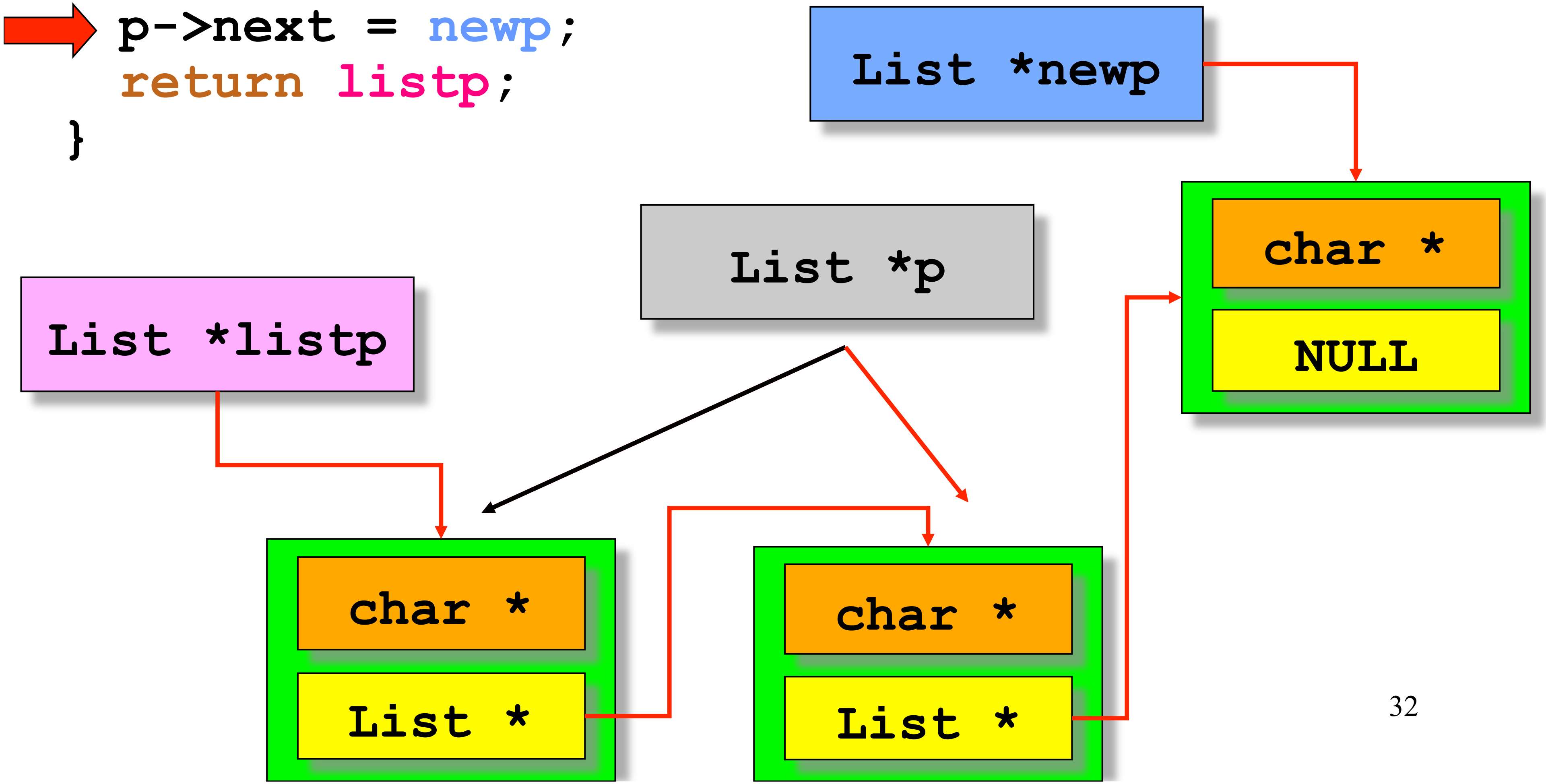
```
List *addToEnd(List *listp, List *newp) {
  List *p;
  if (listp == NULL)
     return newp;
  for (p = listp; p->next != NULL; p = p->next)
     ;
  p->next = newp;
  return listp;
}
```



Programming Linked Lists

Draw pictures of case of list *listp containing 2 elements

```
List *addToEnd(List *listp, List *newp) {
  List *p;
  if (listp == NULL)
     return newp;
  for (p = listp; p->next != NULL; p = p->next)
     ;
  p->next = newp;
  return listp;
}
```



Programming Linked Lists

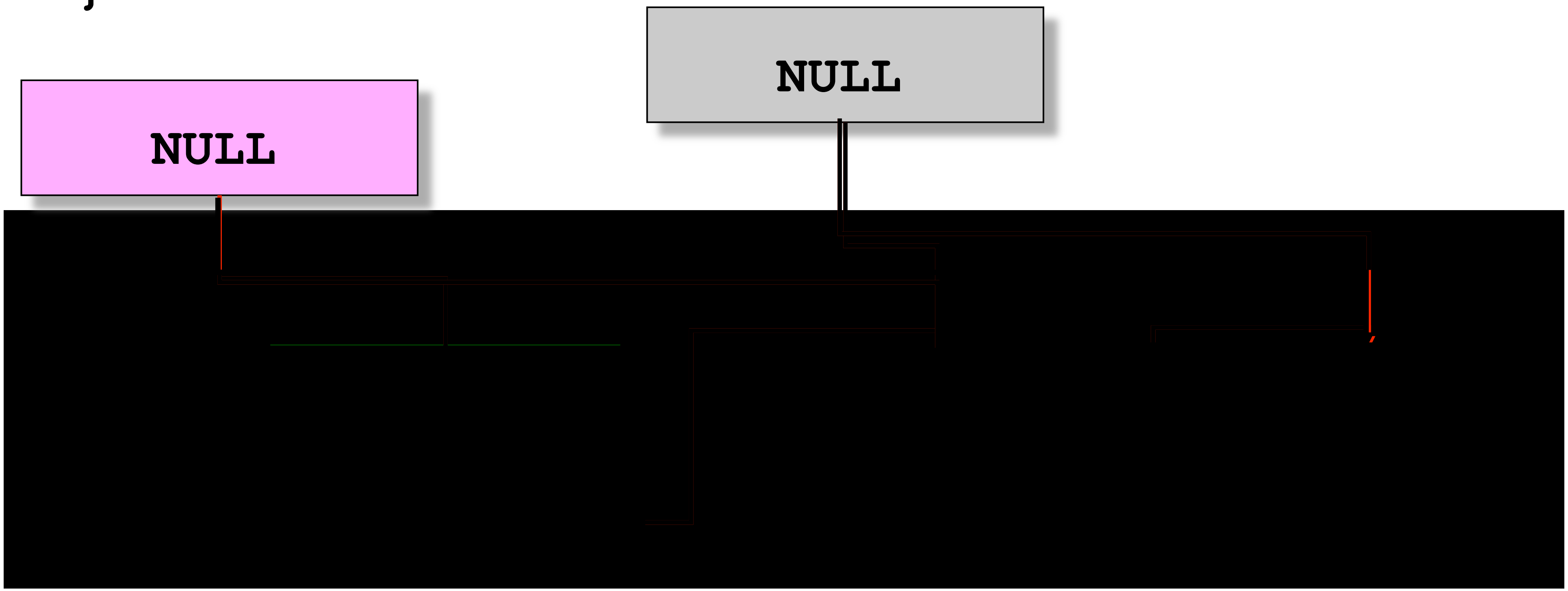
De-allocating a complete list

```
void freeAll(List *listp) {
  List *p;
  for ( ; listp != NULL; listp=p) {
     p = listp->next;
     free(listp->content);     /* the string */
     free(listp);
  }
}
```

Programming Linked Lists

Draw the memory diagram for a multi element list.

```
void freeAll(List *listp) {
  List *p;
  for ( ; listp != NULL; listp = p ) {
     p = listp->next;
     free(listp->content);
     free(listp);
  }
}
```



Programming Linked Lists

Write a function that deletes the first element in a list.

```
List * deleteFirst(List *listp)
{
  List *p;

  if (listp != NULL)
  {
     p = listp;
     listp = listp->next;
     free(p);
  }
  return listp;
}
```

Programming Linked Lists

Write a function that counts the number of elements in a list.

```
int count(List *listp)
{
  int cnt = 0;

  for ( ; listp != NULL; cnt++)
      listp = listp->next;

  return cnt;
}
```

Summary

- ✓ To extend understanding of pointers by using them to create dynamic data structures
- ✓ Understand when to use a Linked List
- ✓ Be able to create a Linked List in C
  - ✓ understand internals
  - ✓ be able to program them!

Sources

- Images
  - http://www.club101.org/graphics/imagepic.gif

End of Segment