Array Lists (cont)
Linked Lists

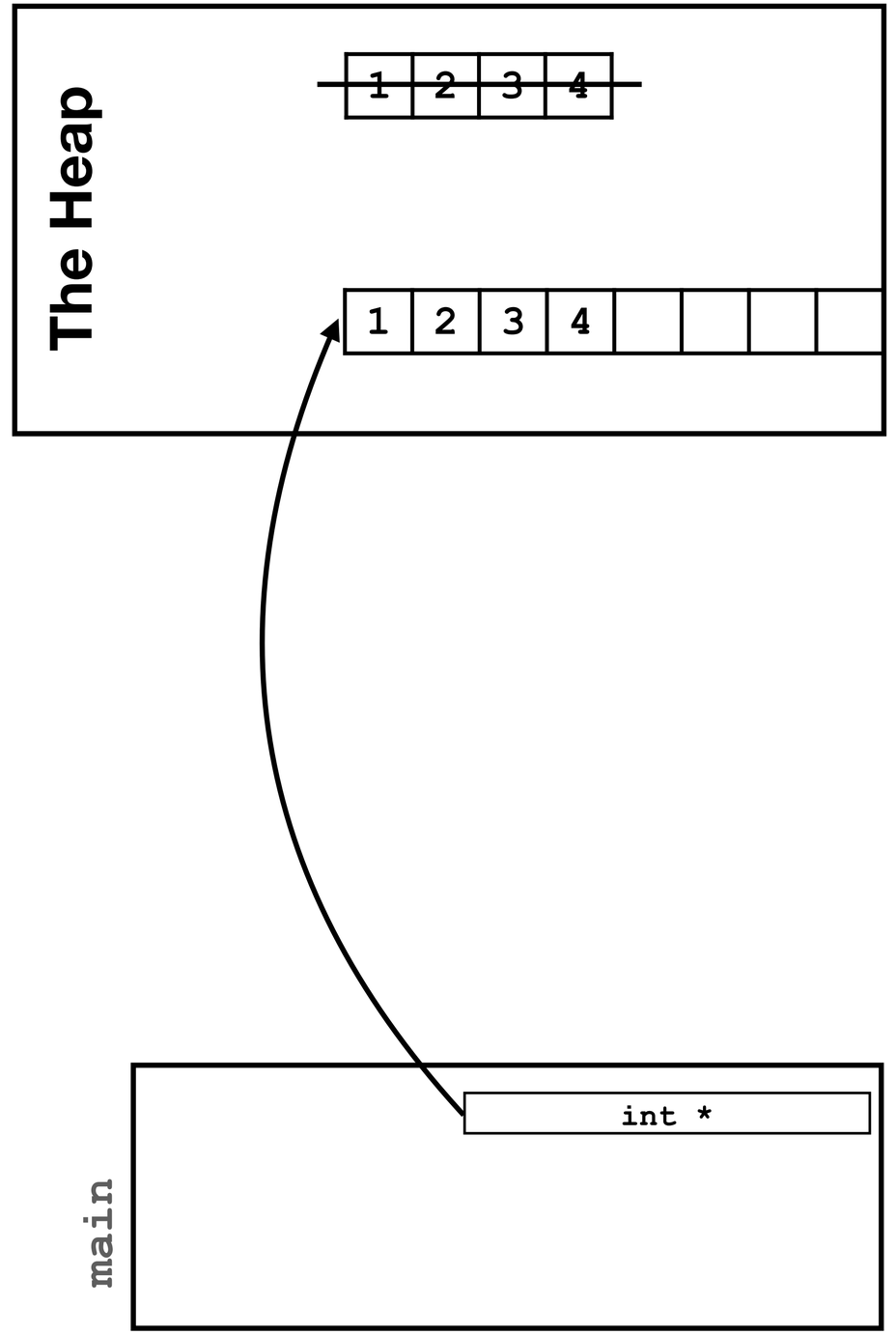
CS143: lecture 10

Konstantinos Ameranis, July 9

Array

Demo

- Let's write this together! (cont)



Array

ArrayList.c - at

Array

ArrayList.c - at

```
int at(const ArrayList array, const size_t i) {
    // Check if valid position
    if (i >= array.length) {    // Check ArrayList bounds
        fprintf(stderr, "Index %lu is out of bounds for ArrayList of length %lu\n",
i, array.length);
        abort();
    }
    return array.array[i];
}
```

Array

ArrayList.c - remove_at

Array

ArrayList.c - remove_at

```
int remove_at(ArrayList *array, size_t i) {
    int element = at(*array, i);
    for (size_t j = i; j < array->length - 1; j++) {
        array->array[j] = array->array[j + 1];
    }
    array->length--;

    return element;
}
```

Array

ArrayList.c - is_empty

Array

ArrayList.c - is_empty

```
int is_empty(const ArrayList array) {
    return array.length == 0;
}
```

Array

ArrayList.c - print_list

Array

ArrayList.c - print_list

```
void print_list(const ArrayList array) {
    printf("[");
    for (size_t i = 0; i < array.length; i++) {
        printf("%d", array.array[i]);
        if (i < array.length - 1) {
            printf(", ");
        }
    }
    printf("]\n");
}
```

Array

ArrayList_example.c

Array

ArrayList_example.c

```
#include <stdlib.h>

#include "ArrayList.h"

int main(void) {
    ArrayList array = create();
    for (size_t i = 0; i < 100; i++) {
        append(&array, i);
    }
    print_list(array);
    for (size_t i = 0; i < 20; i++) {
        remove_at(&array, i);
    }
    print_list(array);

    return EXIT_SUCCESS;
}
```

Array

ArrayList_example.c

Array

ArrayList_example.c

```
./ArrayList_example
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43,
44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64,
65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85,
86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99]
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 40, 41,
42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62,
63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83,
84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99]
```

Understanding `void *`

`void *`

The dark side of C

- How do we usually declare a pointer?
  - `char *`
  - `int *`
  - `struct student *`
- Why do we care what the pointer is pointing to?
  - To figure out how many bytes to read/write
- What if we never read/write to the memory location?
  - If we just want to hold onto a pointer

`void *`

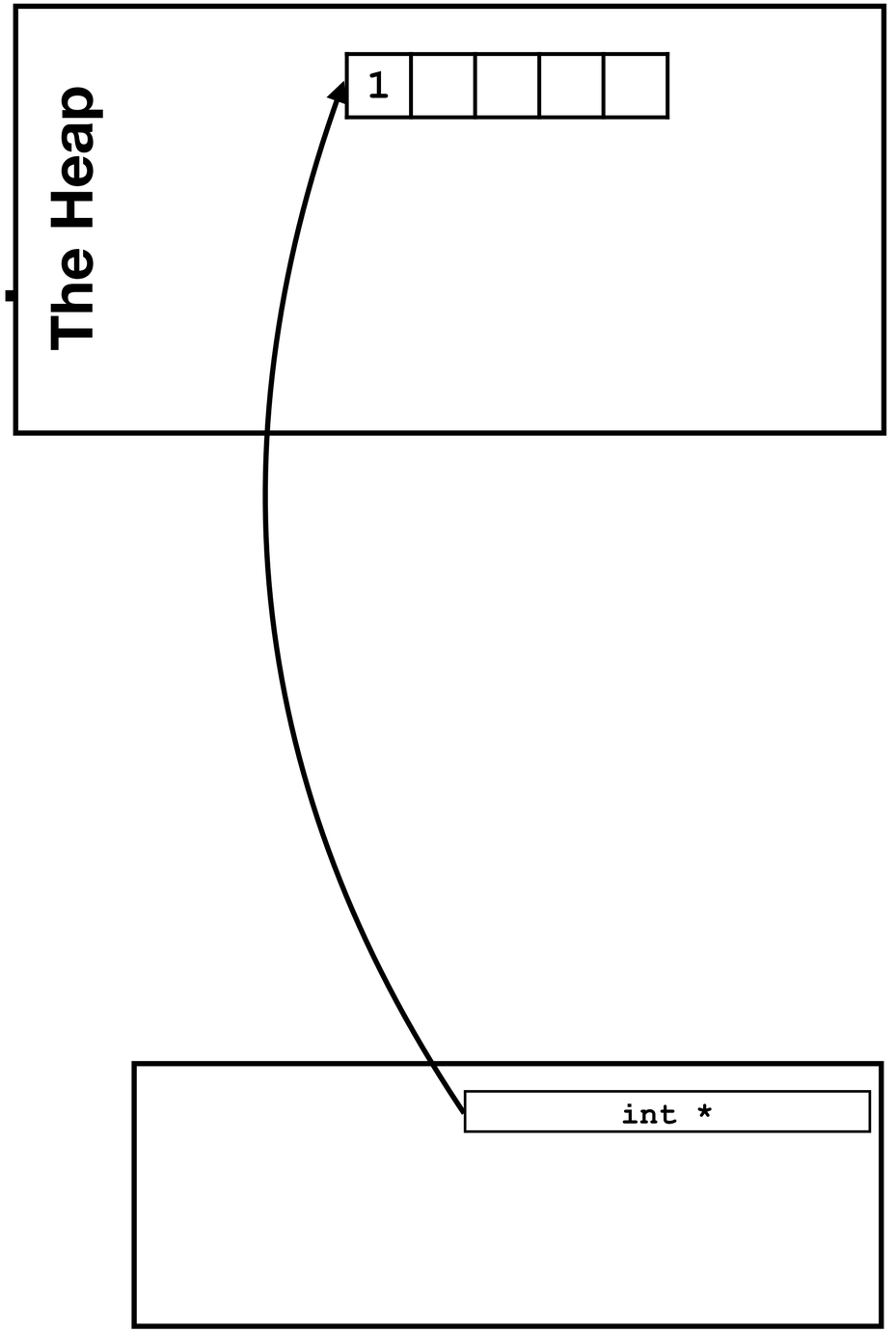
The dark side of C

- void * is a *generic* pointer. Just an address; there is no information about what it points to.
- Can't dereference void * without *casting* it first.
- Can we make our array implementation generic?

Linked List

Array

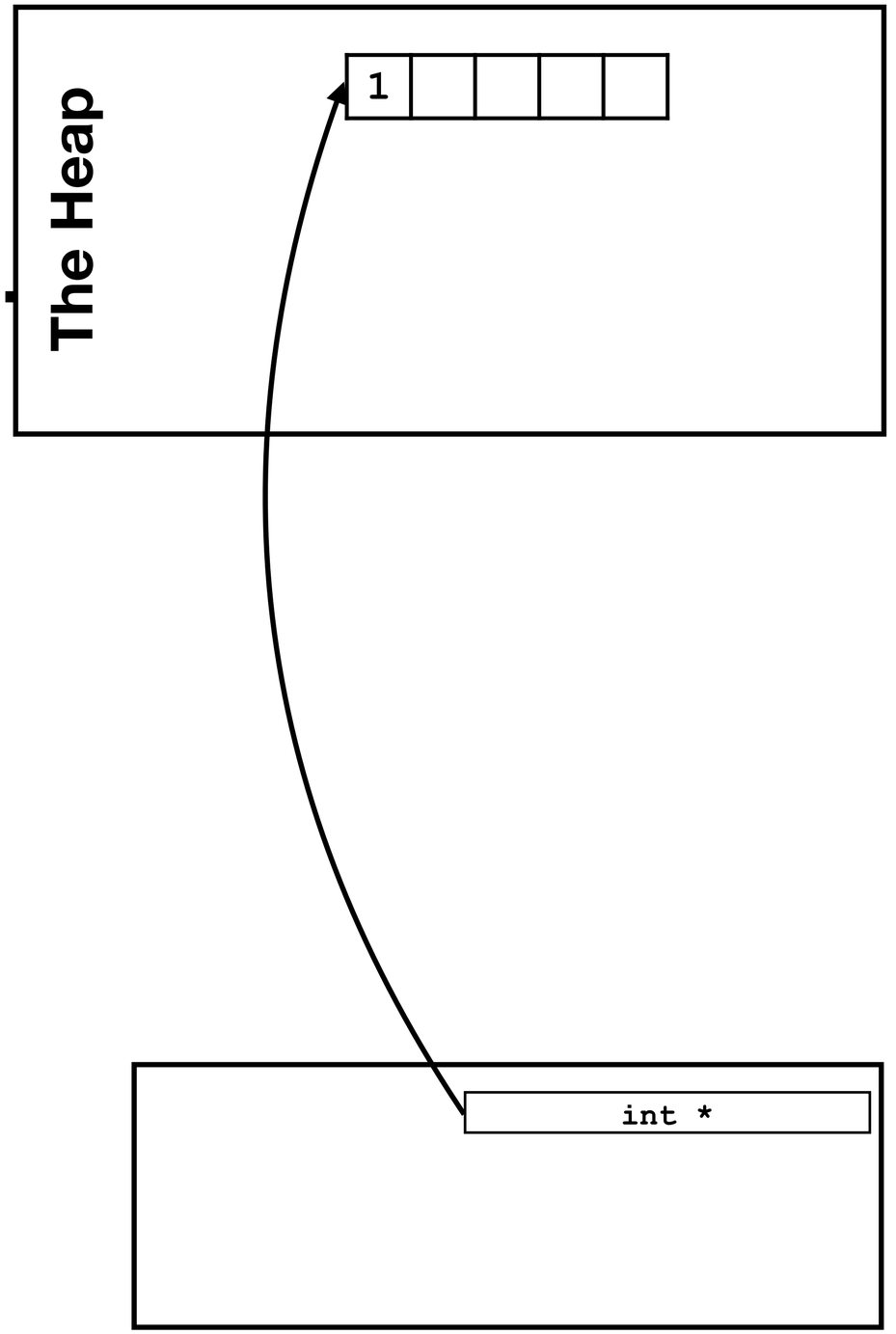
Another way to grow the size of the array...



Array

Another way to grow the size of the array...

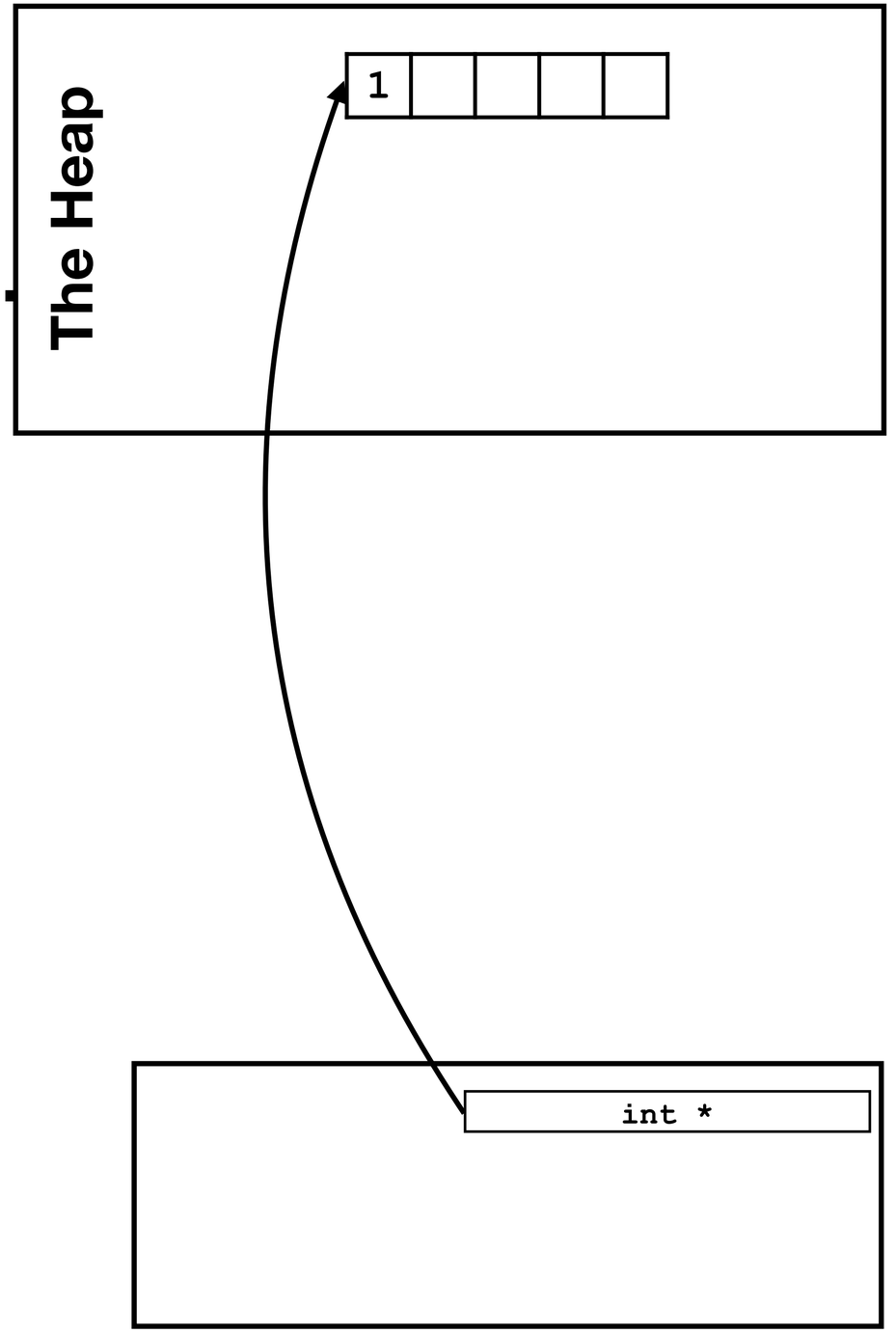
- What if we store a pointer at the end of the array



Array

Another way to grow the size of the array...

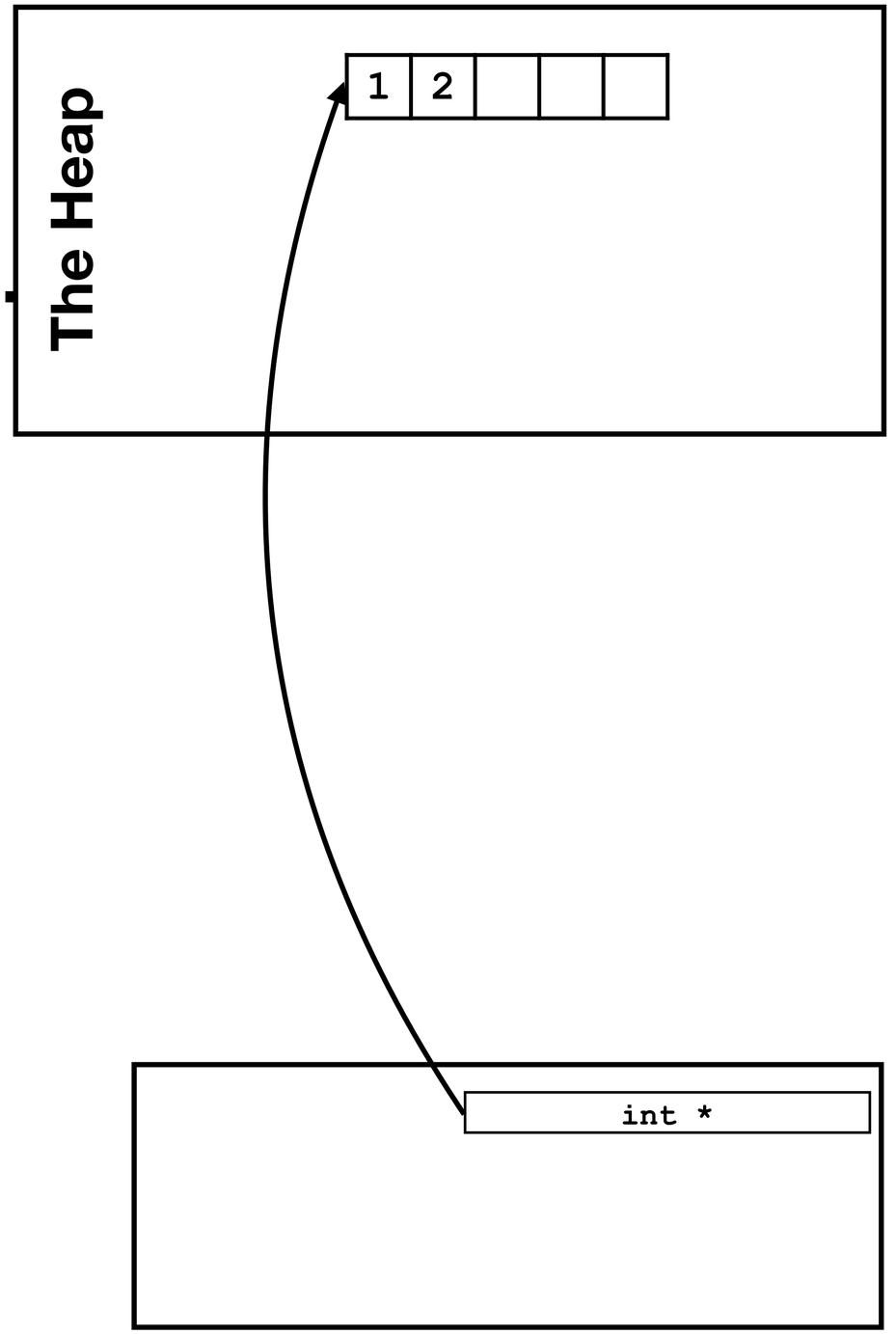
- What if we store a pointer at the end of the array
- ...when the array is filled up, we direct it to somewhere else.



Array

Another way to grow the size of the array...

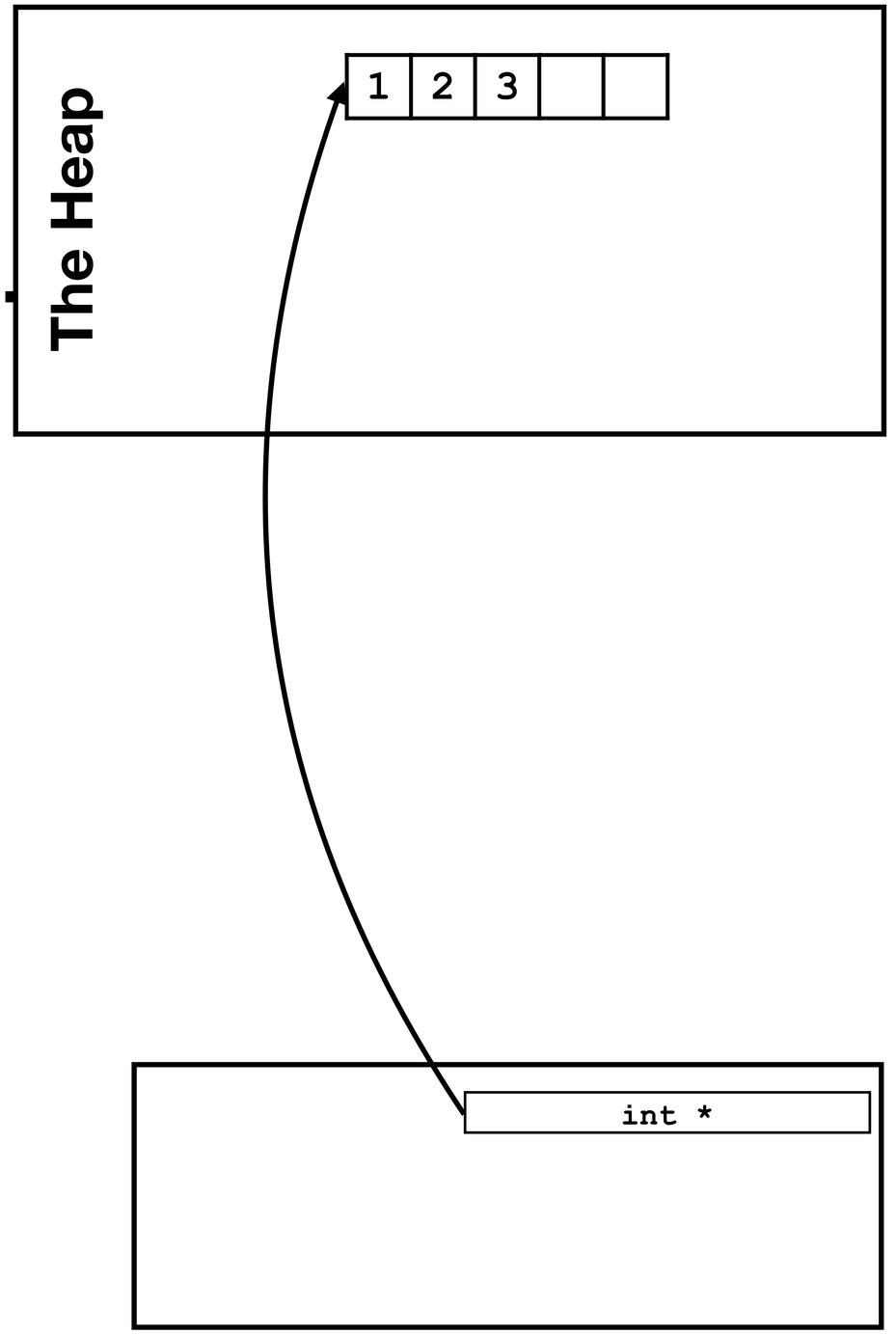
- What if we store a pointer at the end of the array
- ...when the array is filled up, we direct it to somewhere else.



Array

Another way to grow the size of the array...

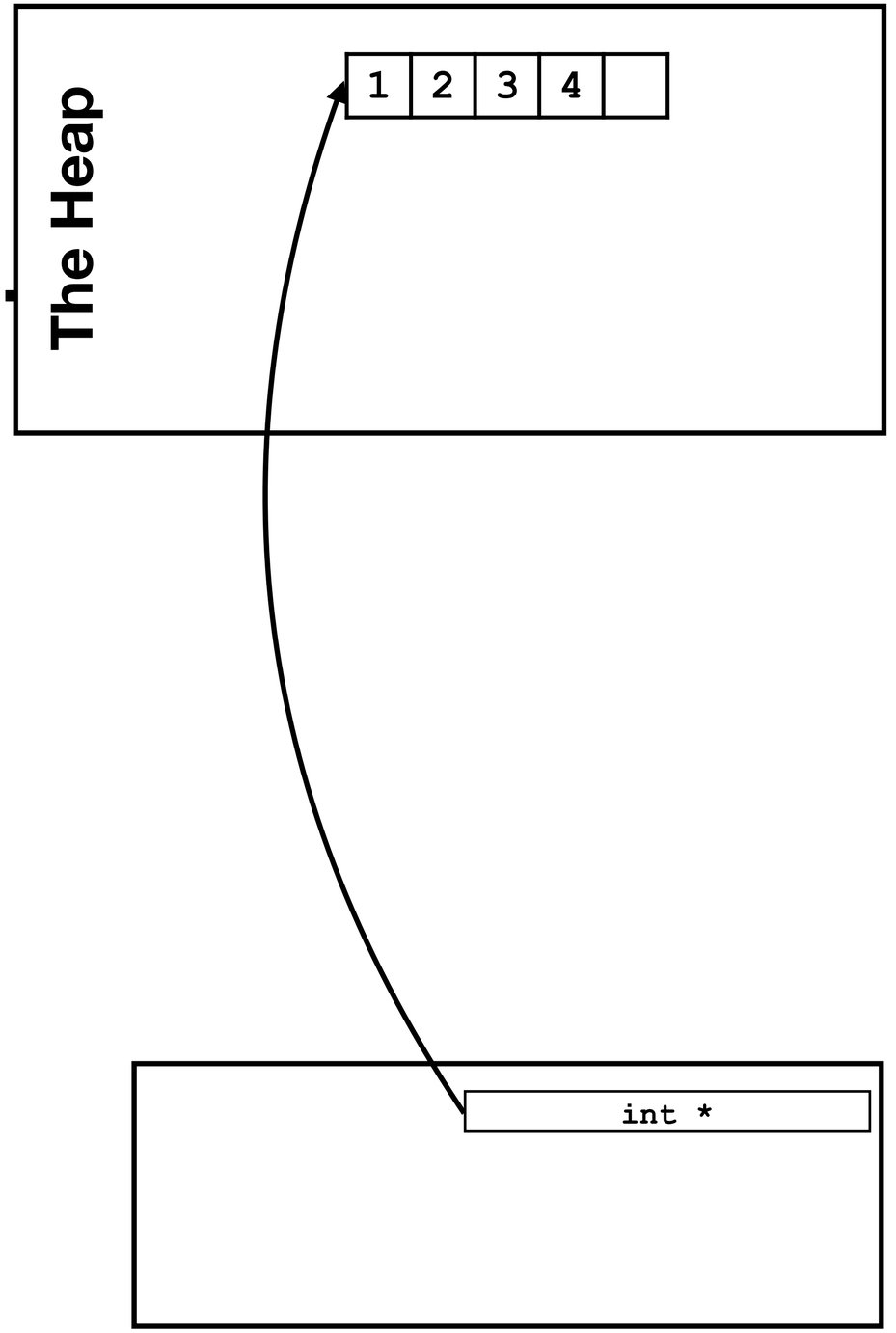
- What if we store a pointer at the end of the array
- ...when the array is filled up, we direct it to somewhere else.



Array

Another way to grow the size of the array...

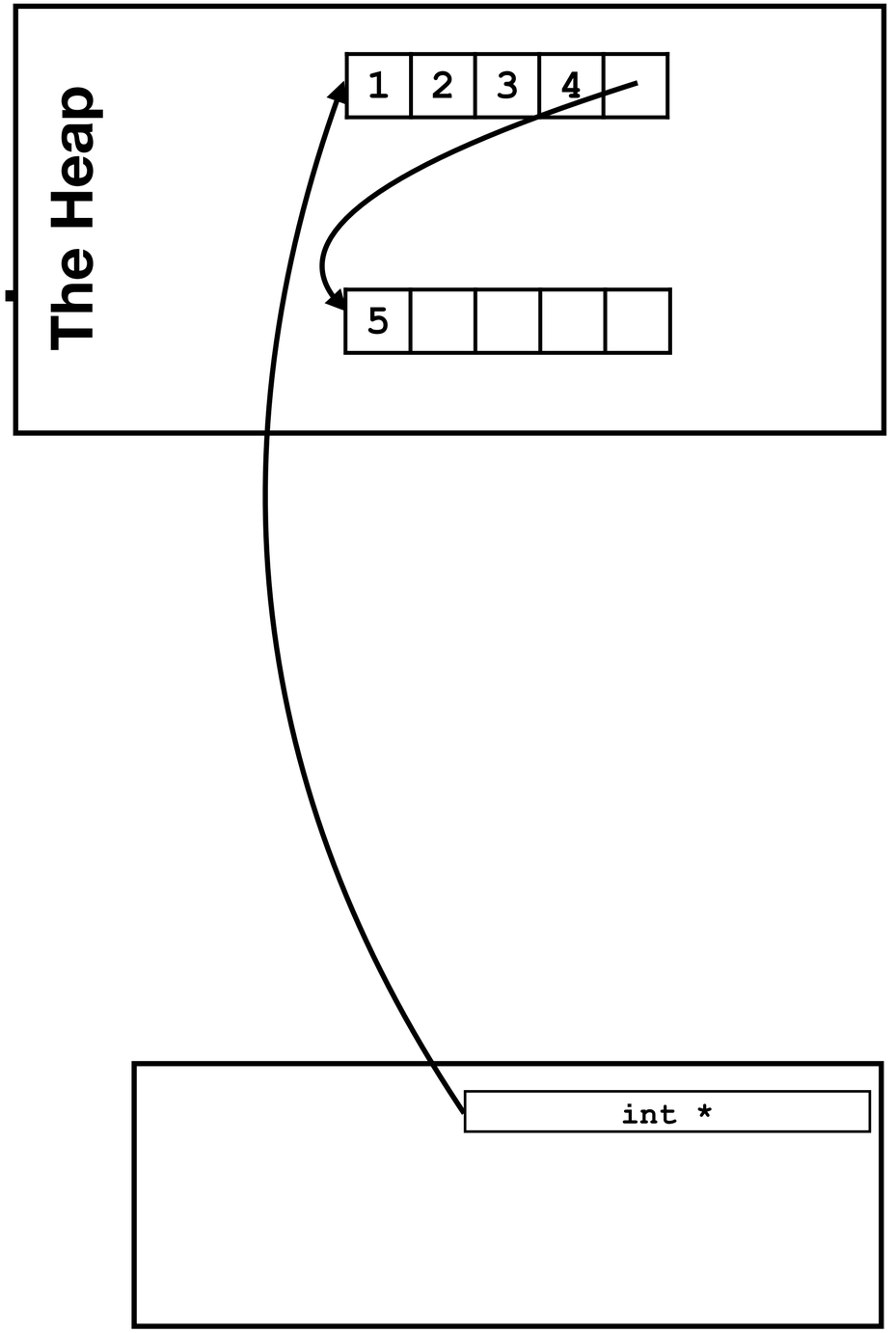
- What if we store a pointer at the end of the array
- ...when the array is filled up, we direct it to somewhere else.



Array

Another way to grow the size of the array...

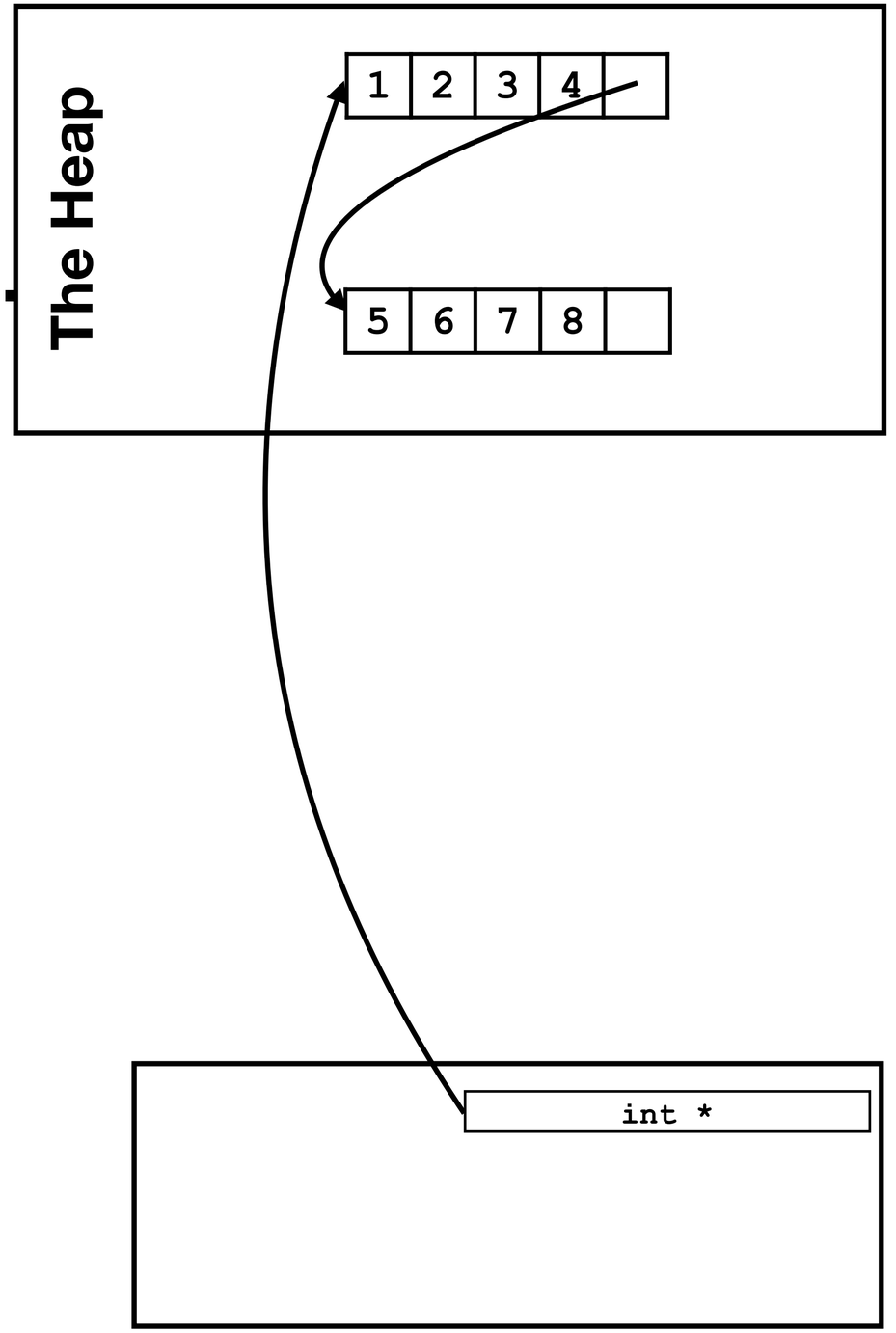
- What if we store a pointer at the end of the array
- ...when the array is filled up, we direct it to somewhere else.



Array

Another way to grow the size of the array...

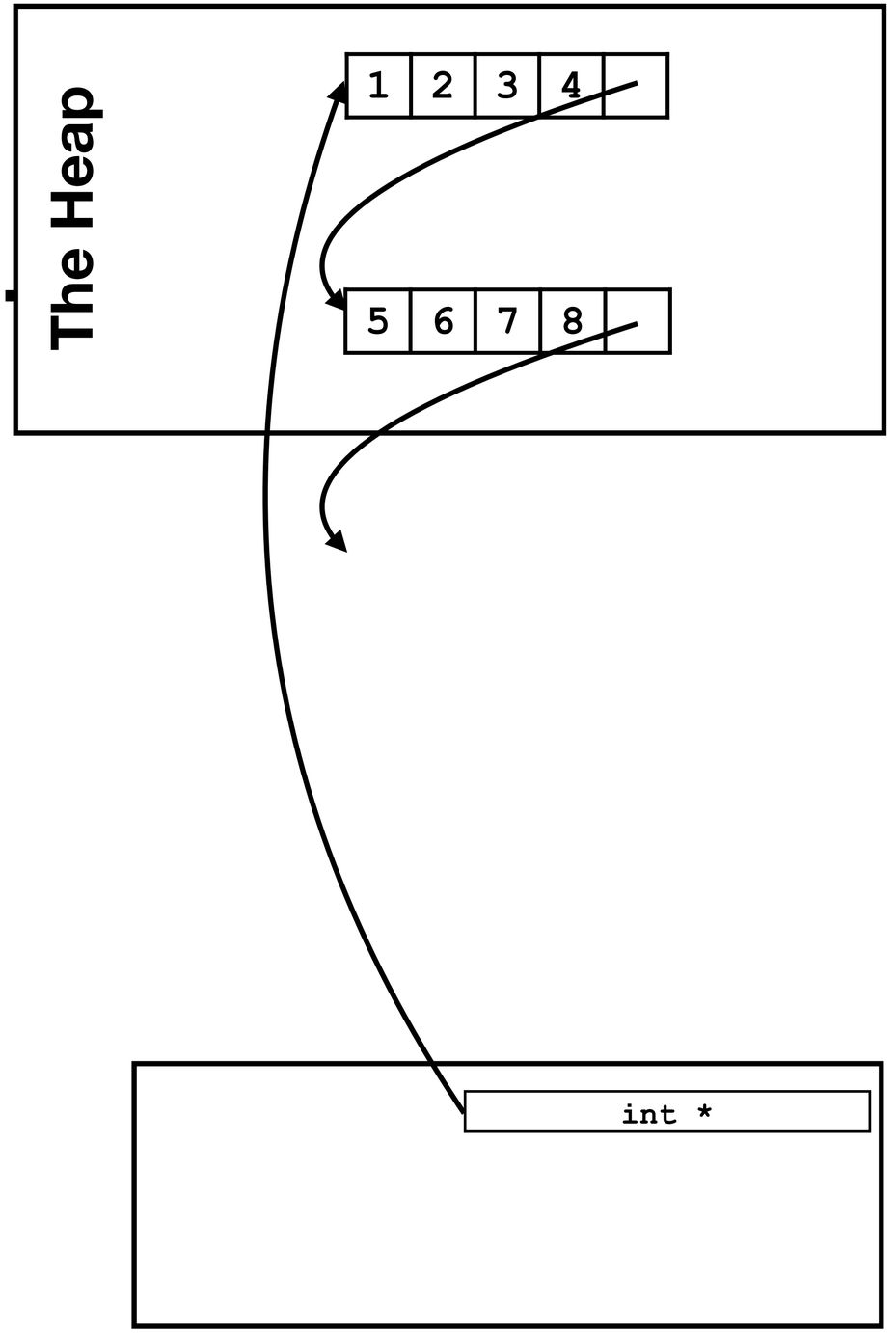
- What if we store a pointer at the end of the array
- ...when the array is filled up, we direct it to somewhere else.



Array

Another way to grow the size of the array...

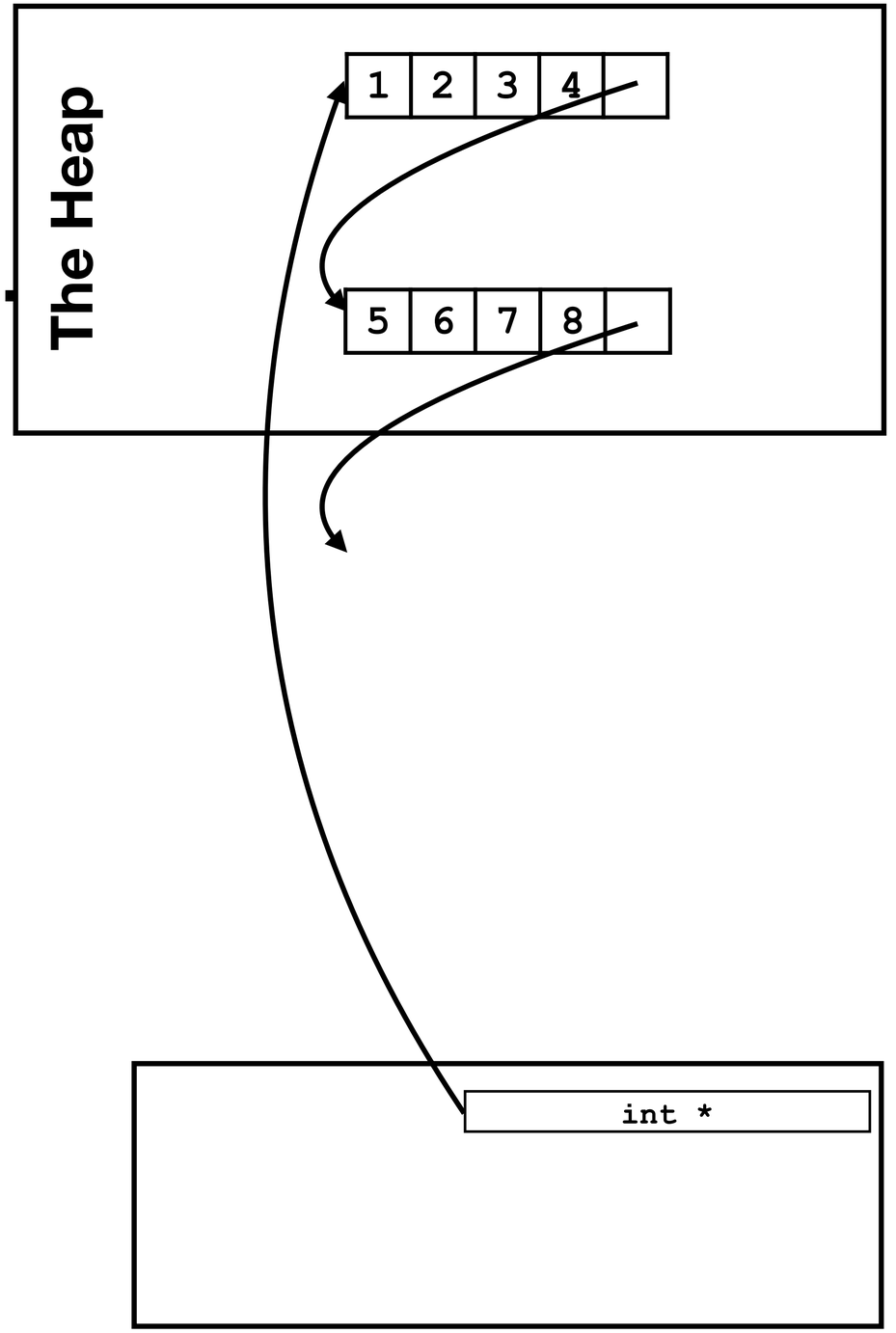
- What if we store a pointer at the end of the array
- ...when the array is filled up, we direct it to somewhere else.



Array

Another way to grow the size of the array...

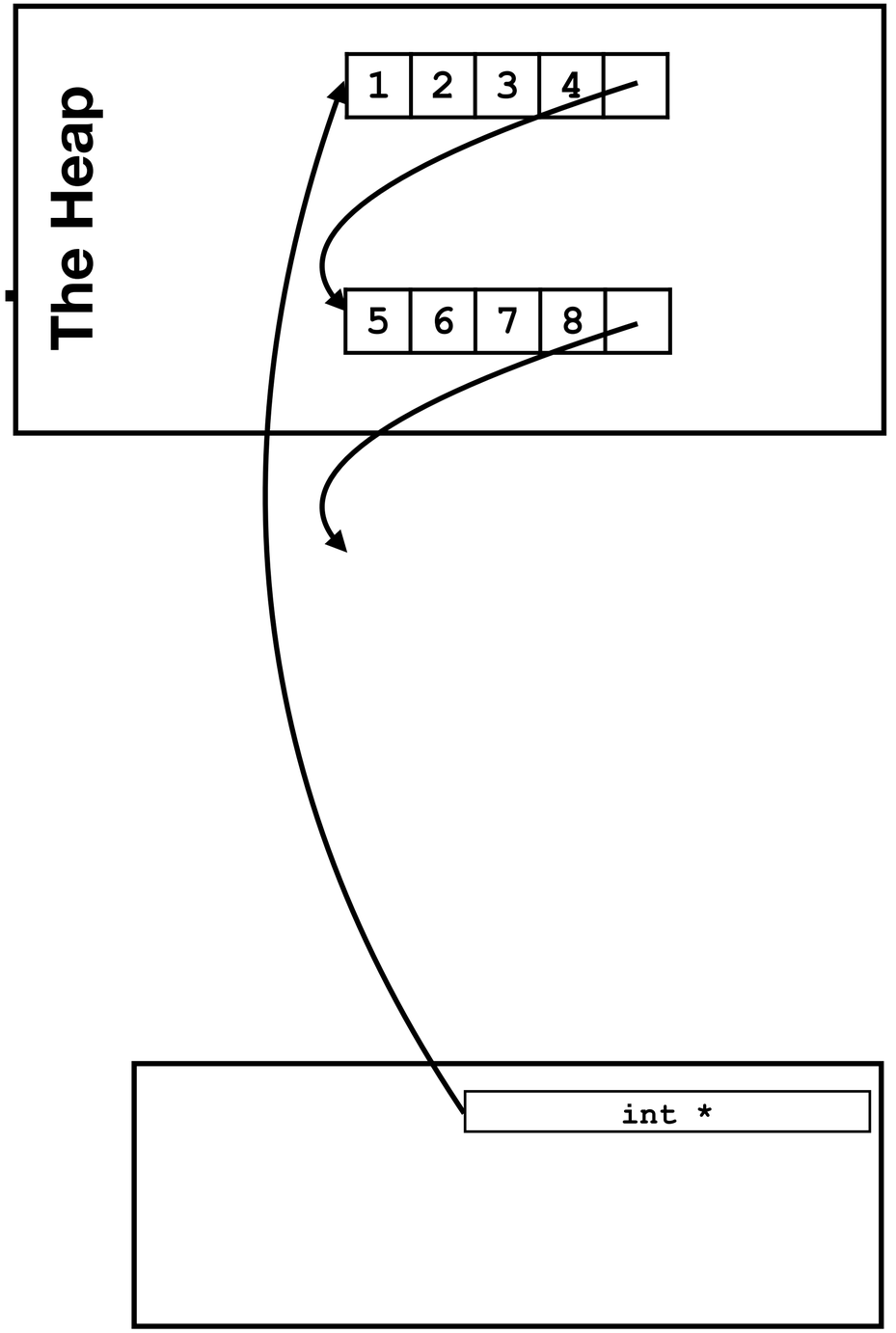
- What if we store a pointer at the end of the array
- ...when the array is filled up, we direct it to somewhere else.



Array

Another way to grow the size of the array...

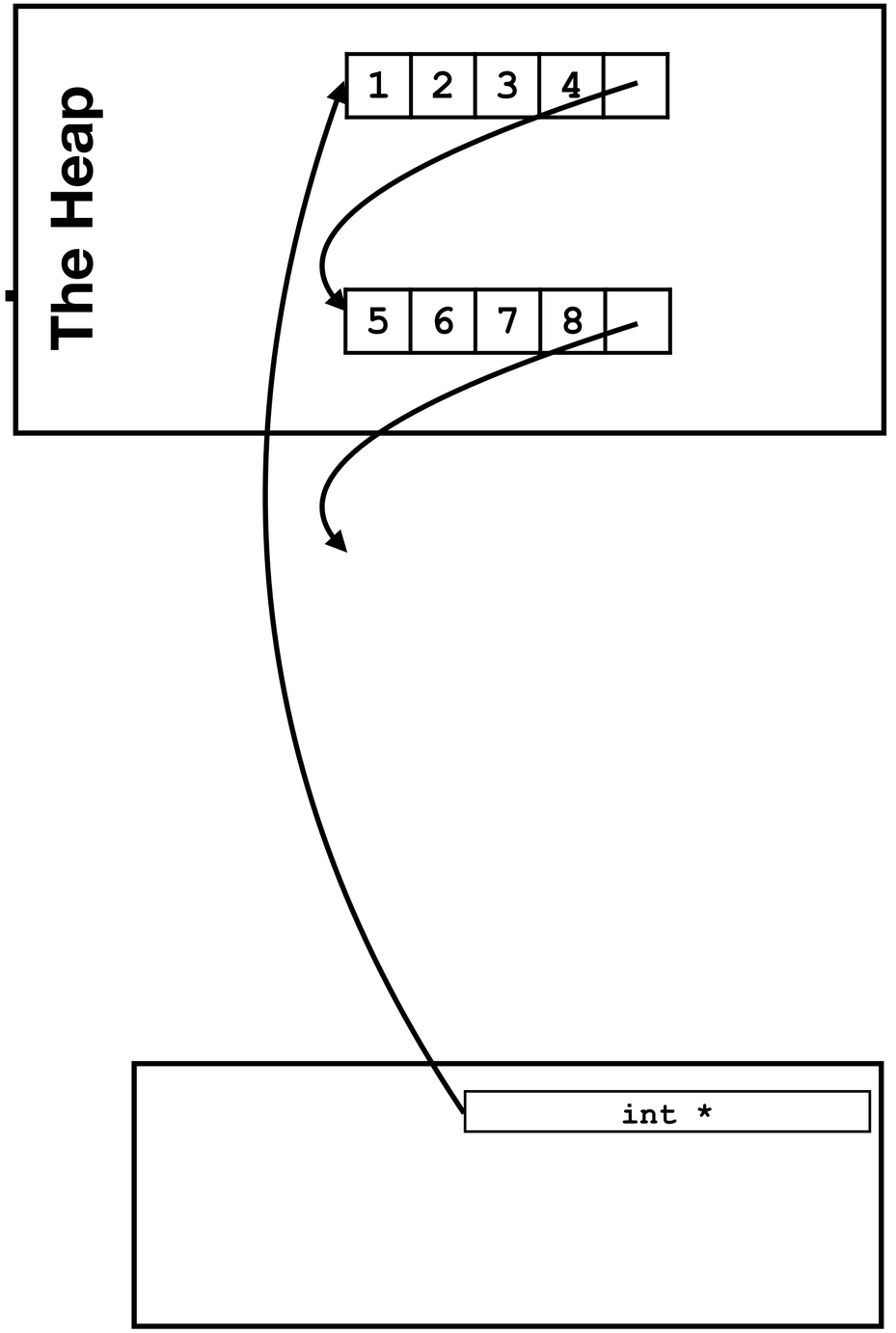
- What if we store a pointer at the end of the array
- ...when the array is filled up, we direct it to somewhere else.
- Good idea?



Array

Another way to grow the size of the array...

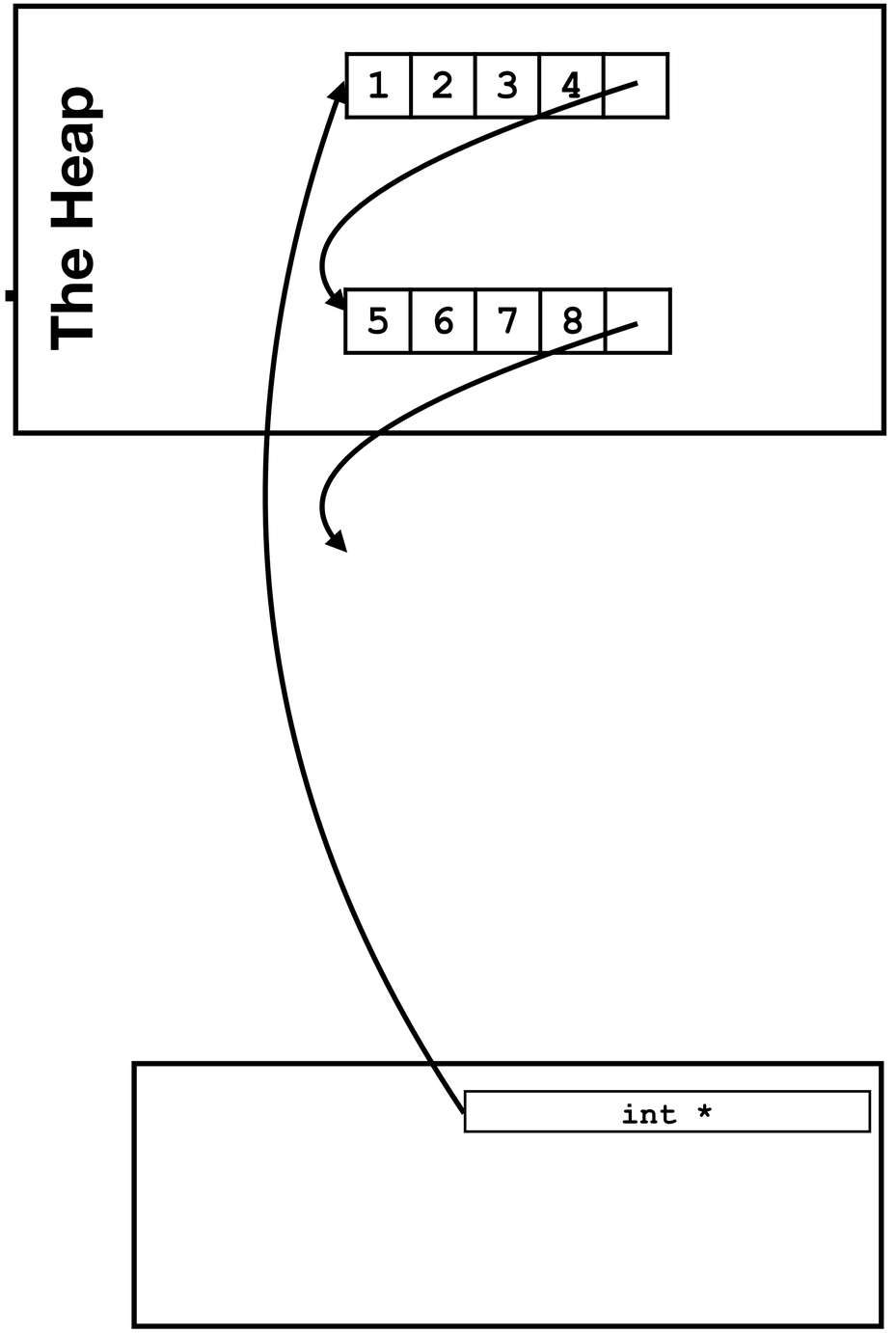
- What if we store a pointer at the end of the array
- ...when the array is filled up, we direct it to somewhere else.
- Good idea?
  - No copying needed when we grow



Array

Another way to grow the size of the array...

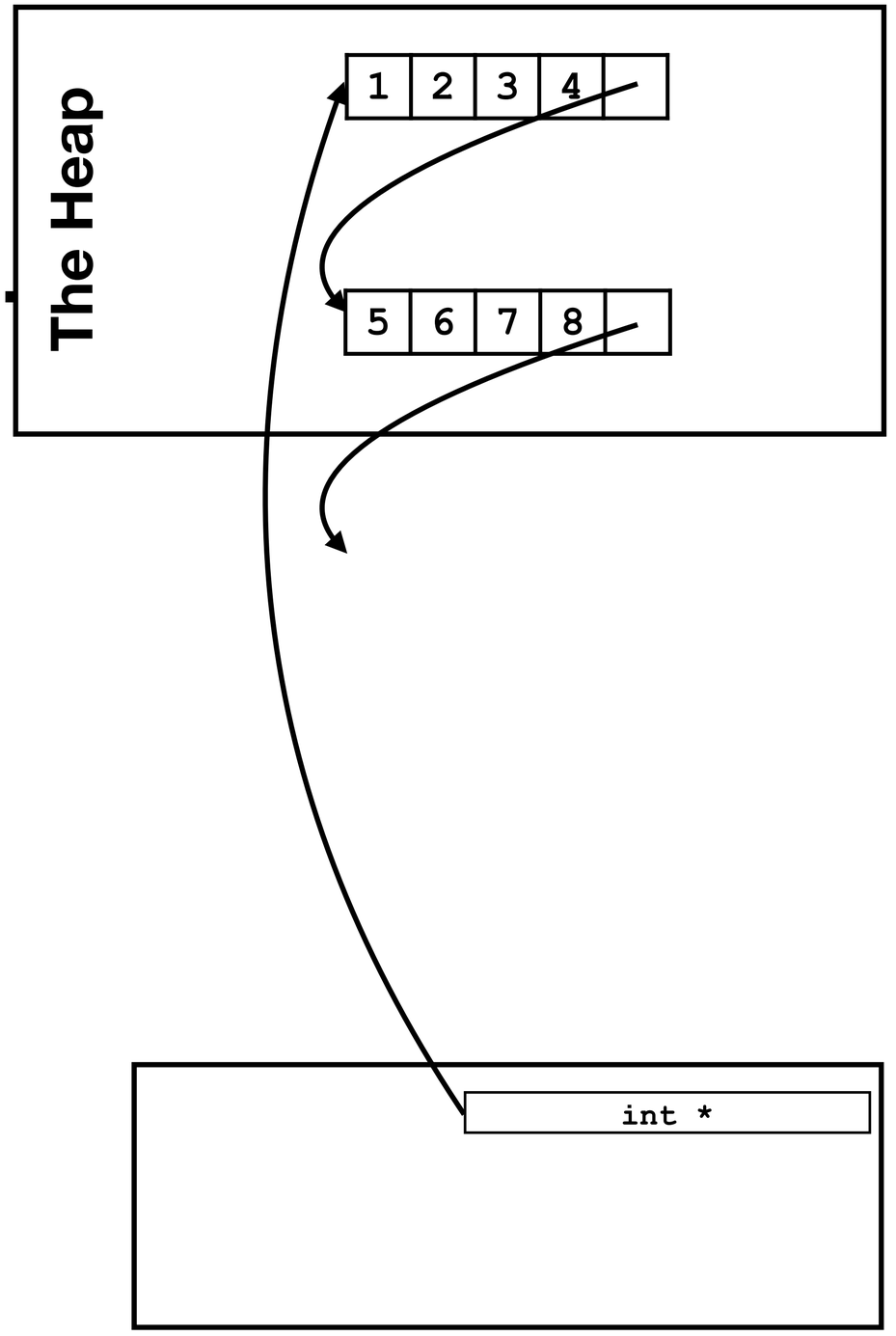
- What if we store a pointer at the end of the array
- ...when the array is filled up, we direct it to somewhere else.
- Good idea?
  - No copying needed when we grow
  - Removing elements at the front is also less costly



Array

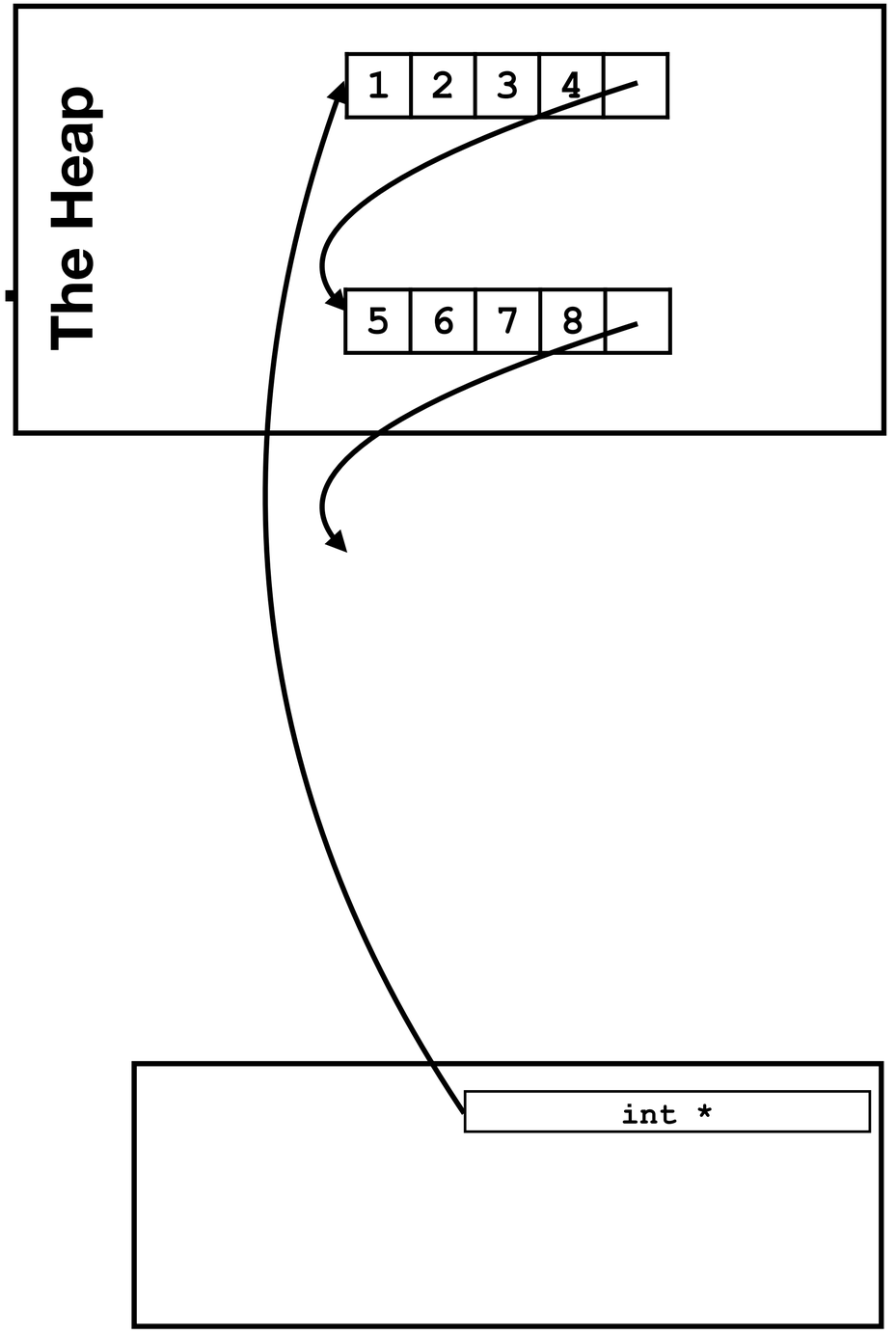
Another way to grow the size of the array...

- What if we store a pointer at the end of the array
- ...when the array is filled up, we direct it to somewhere else.
- Good idea?
  - No copying needed when we grow
  - Removing elements at the front is also less costly
  - But how do you index? Say I want 10-th element



Array

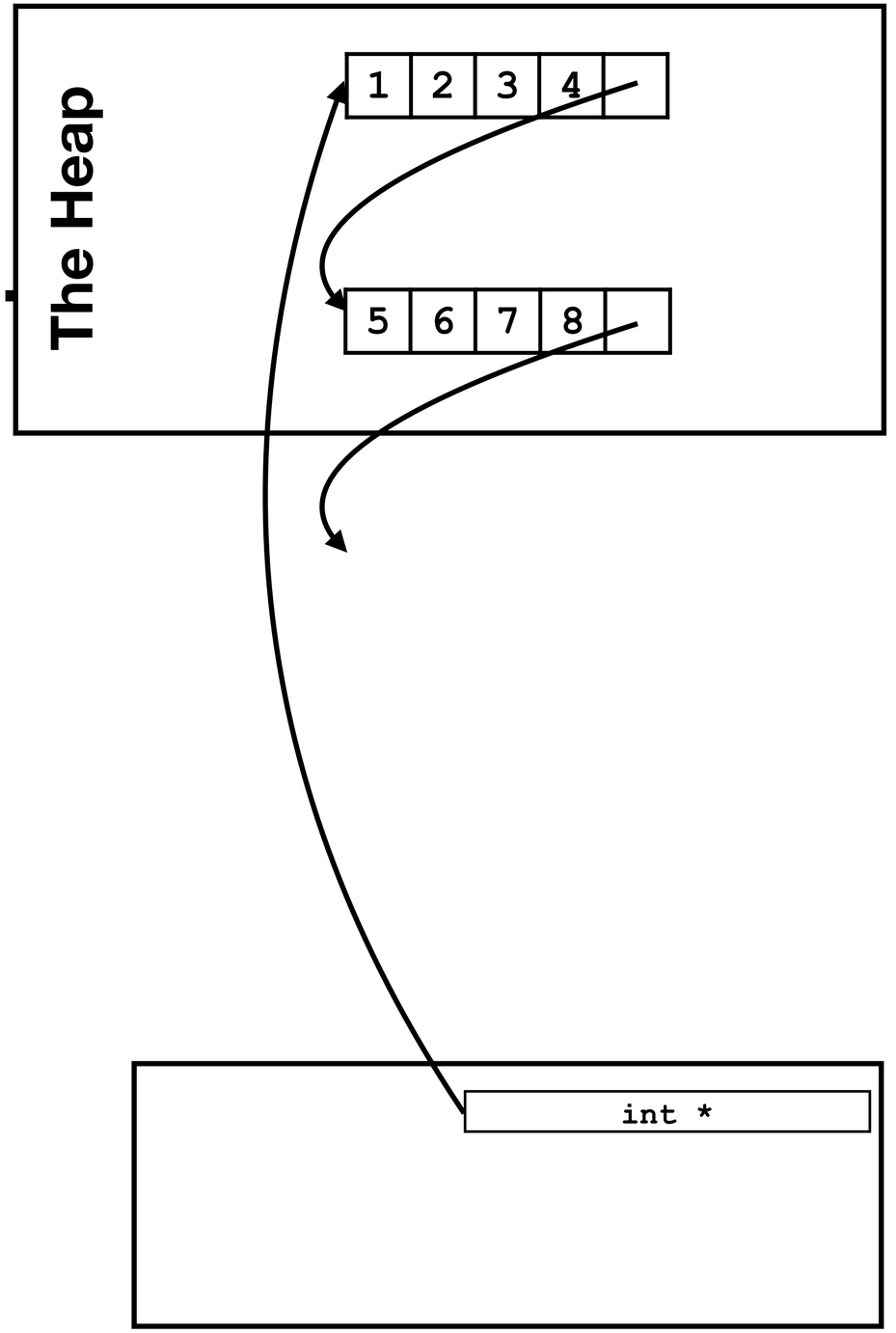
Another way to grow the size of the array...



Array

Another way to grow the size of the array...

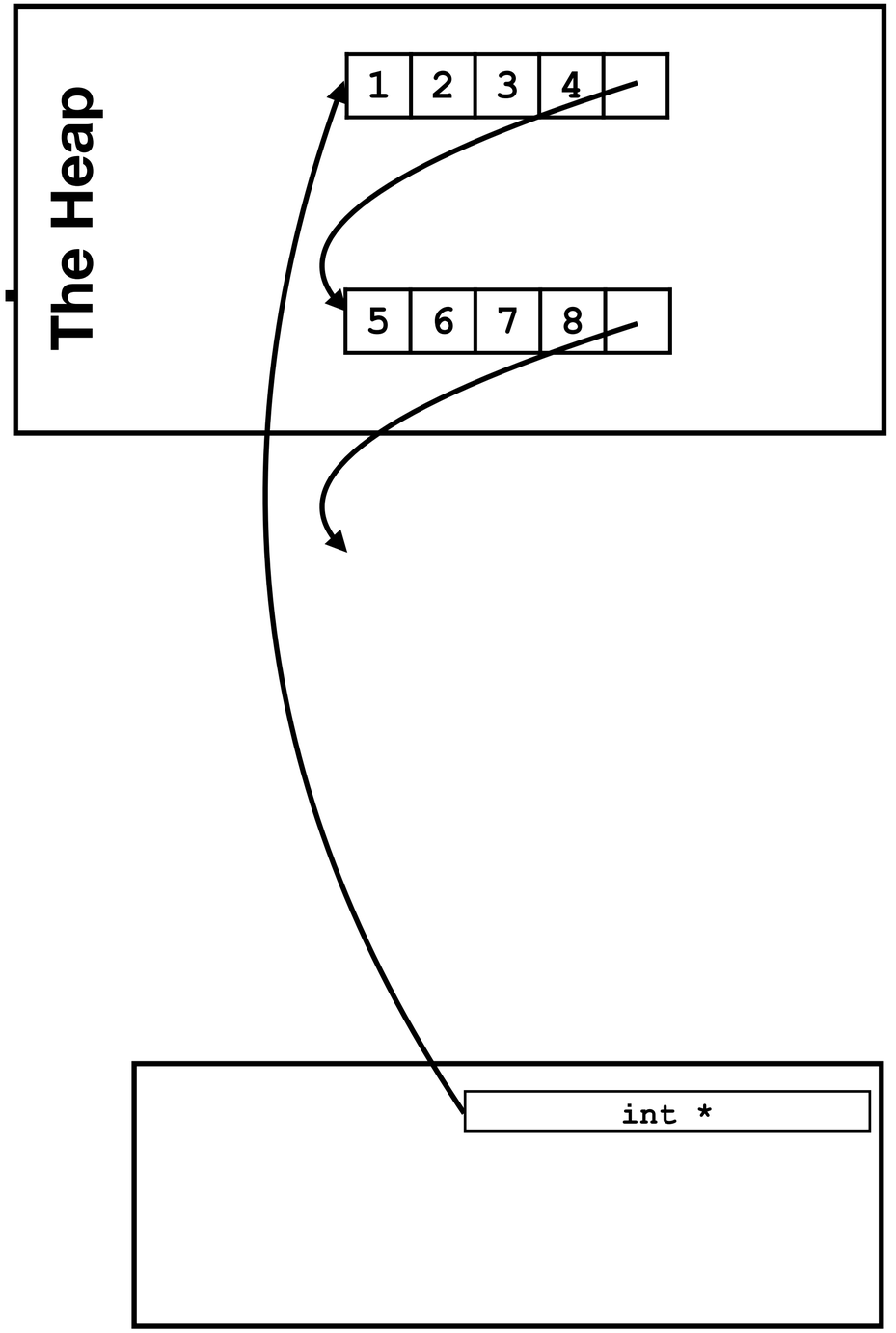
- Also, how many elements should we allocate per small array?



Array

Another way to grow the size of the array...

- Also, how many elements should we allocate per small array?
- How about just one?



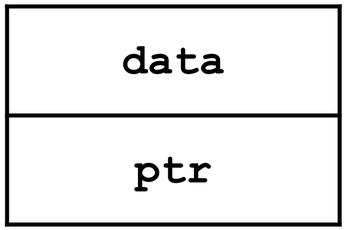
Linked Lists

Linked Lists

- A linked list consists of *nodes*

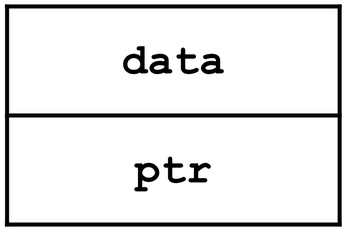
Linked Lists

- A linked list consists of *nodes*
- A node consists of



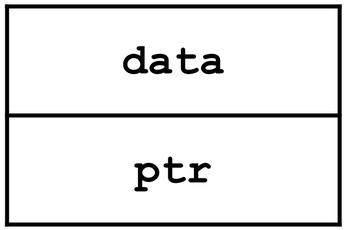
Linked Lists

- A linked list consists of *nodes*
- A node consists of
  - A piece of data



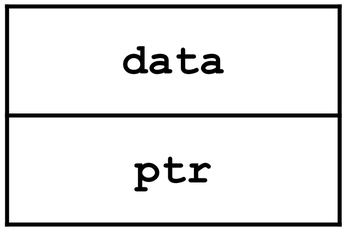
Linked Lists

- A linked list consists of *nodes*
- A node consists of
  - A piece of data
  - A pointer to the next node



Linked Lists

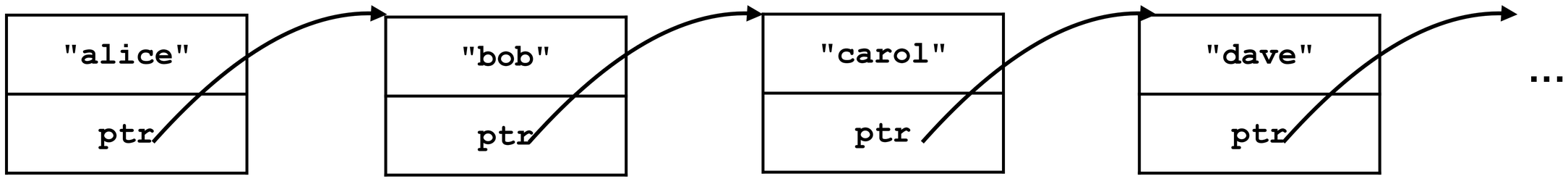
- A linked list consists of *nodes*
- A node consists of
  - A piece of data
  - A pointer to the next node




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

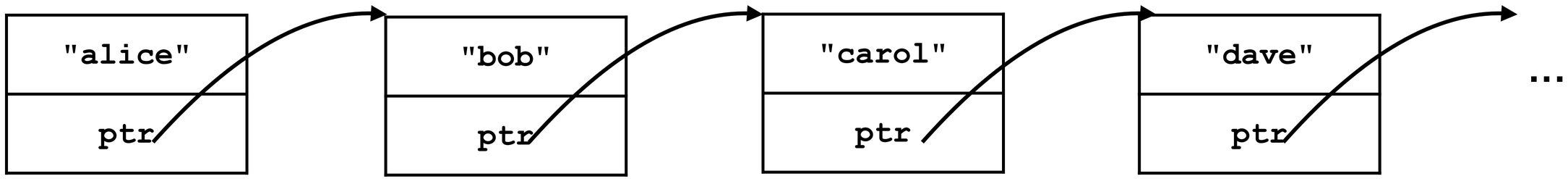
Linked Lists

- A linked list consists of *nodes*
- A node consists of
  - A piece of data
  - A pointer to the next node
- A linked list is a chain (linear series) of nodes



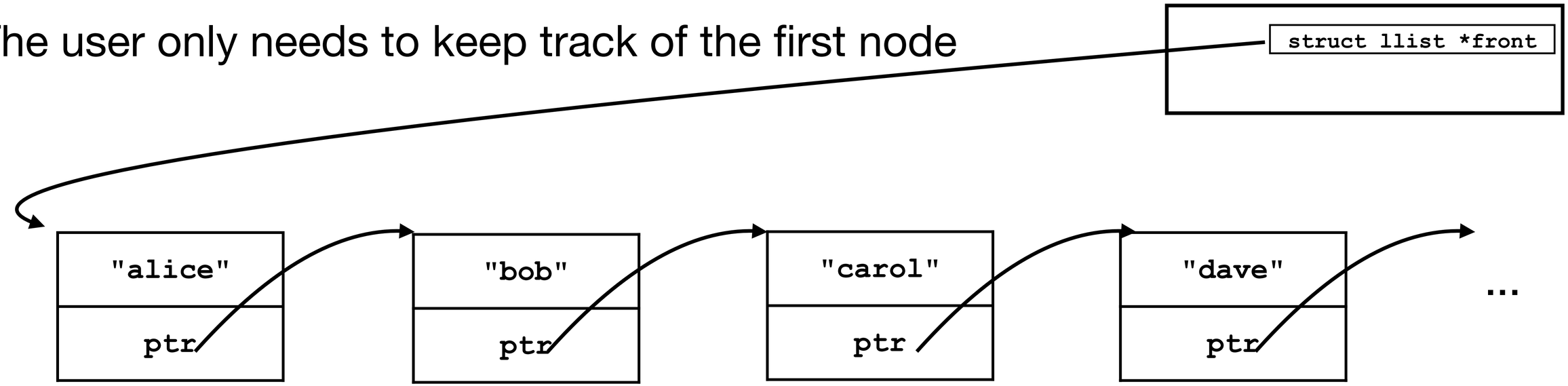
Linked Lists

- Each node could be stored anywhere in memory
  - Unlike arrays, data is contiguous
  - Each node keeps track of the next node



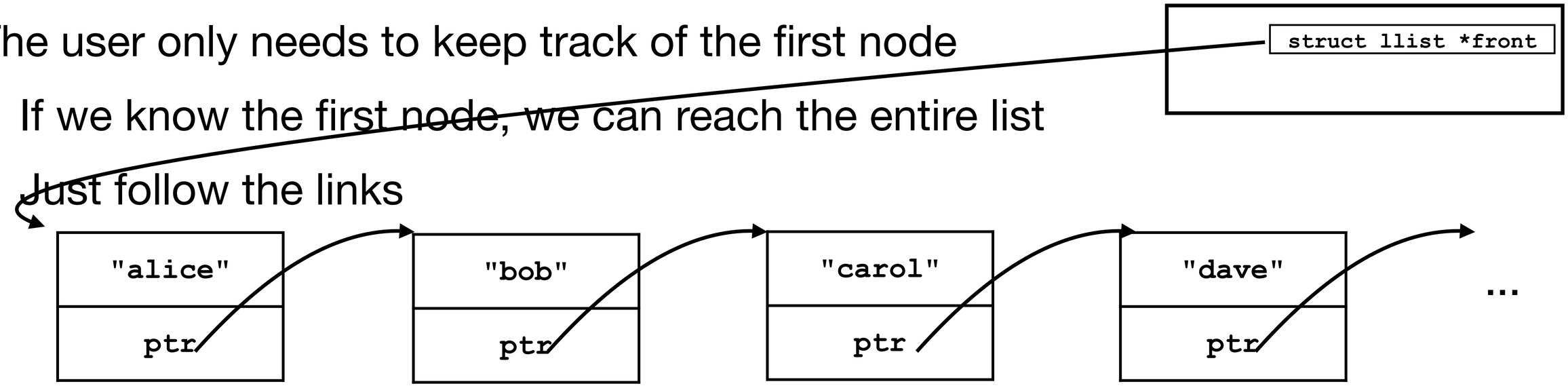
Linked Lists

- Each node could be stored anywhere in memory
  - Unlike arrays, data is contiguous
  - Each node keeps track of the next node
- The user only needs to keep track of the first node



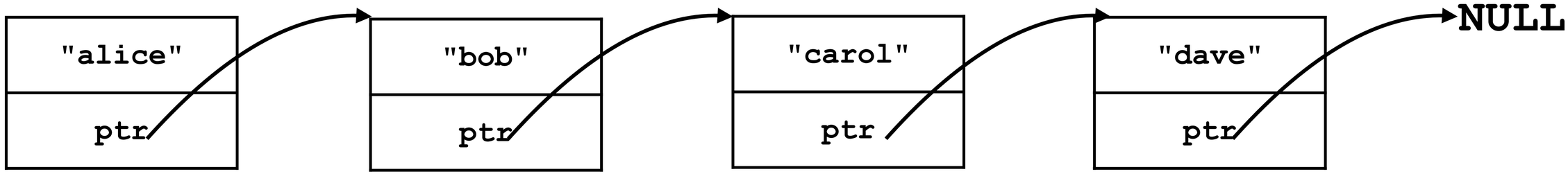
Linked Lists

- Each node could be stored anywhere in memory
  - Unlike arrays, data is contiguous
  - Each node keeps track of the next node
- The user only needs to keep track of the first node
  - If we know the first node, we can reach the entire list
  - Just follow the links

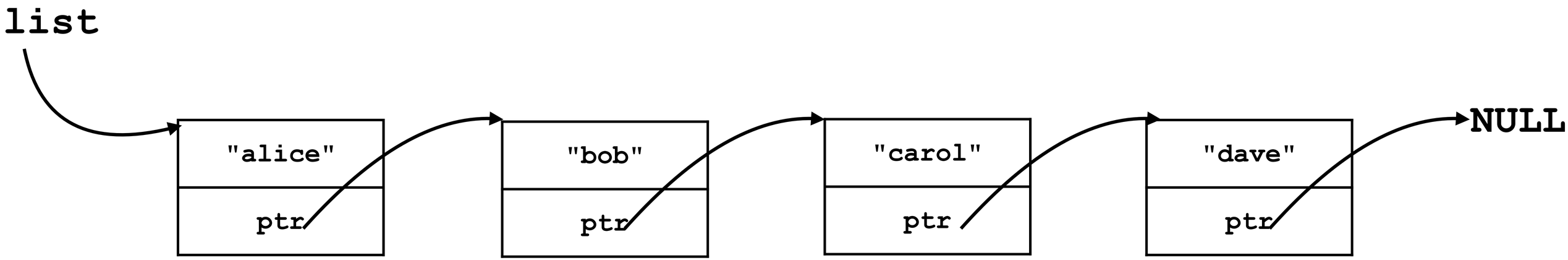


Linked Lists

- The end of the list is signaled by the special pointer **`NULL`**
- **`NULL`** is a pointer that points nowhere

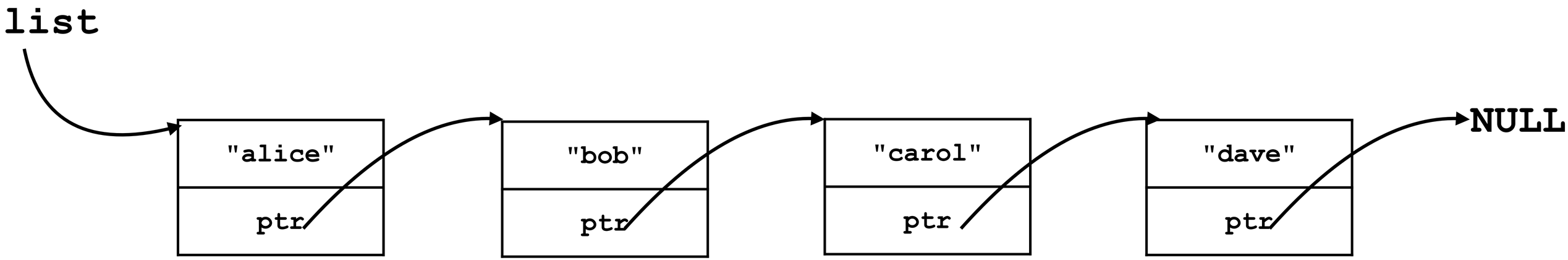


Linked Lists



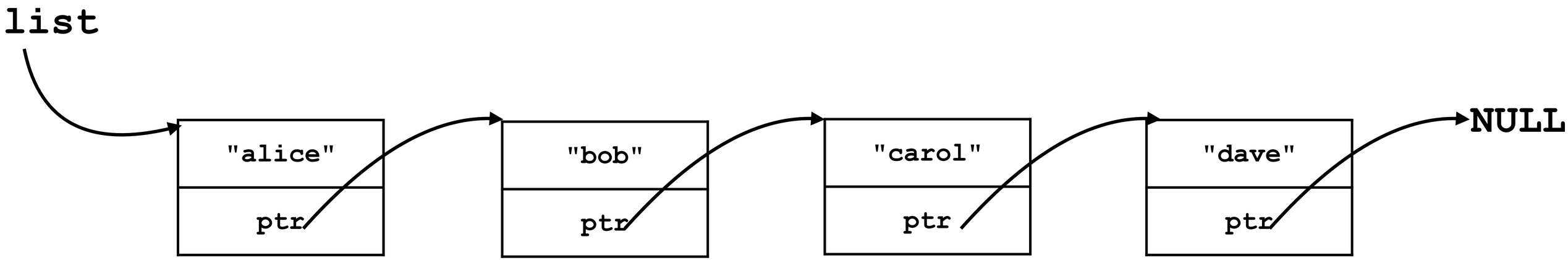
Linked Lists

- A *linked list* can be either:



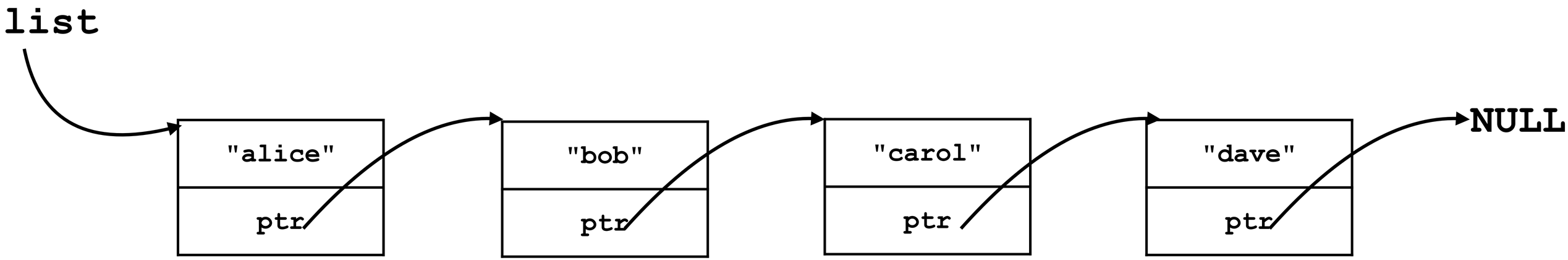
Linked Lists

- A *linked list* can be either:
  - NULL (empty)



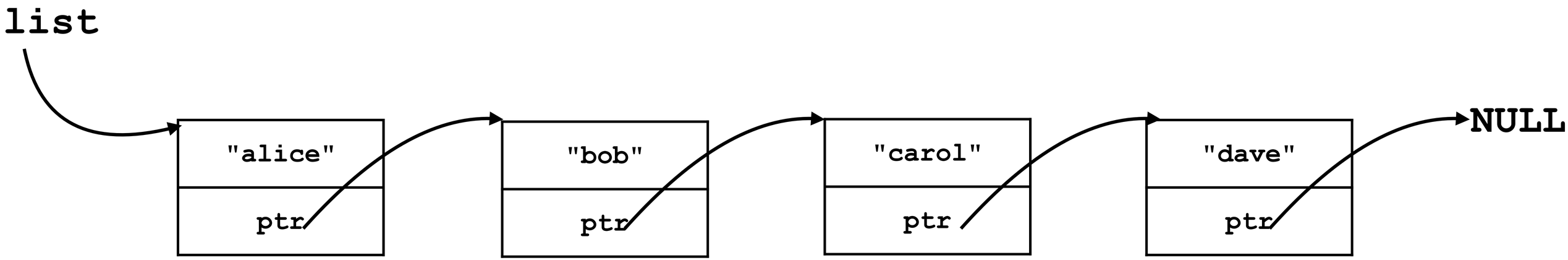
Linked Lists

- A *linked list* can be either:
  - NULL (empty)
  - A node with data and a pointer to a *linked list*



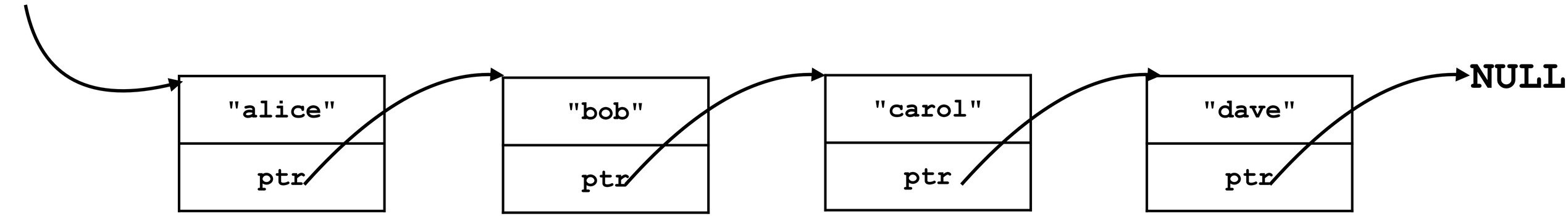
Linked Lists

- A *linked list* can be either:
  - NULL (empty)
  - A node with data and a pointer to a *linked list*
- Recursion



Linked Lists

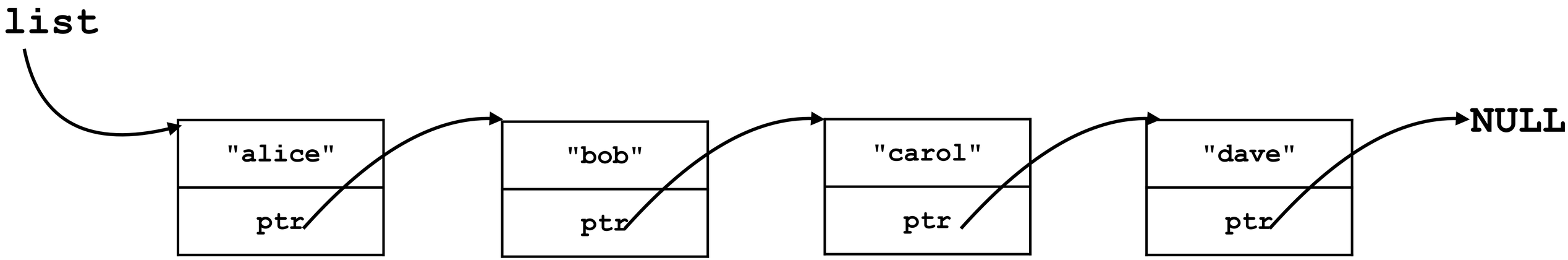
- A *linked list* can be either:
  - NULL (empty)
  - A node with data and a pointer to a *linked list*
- Recursion
  - although we wouldn't actually implement the operators recursively



Linked Lists

`prepend`

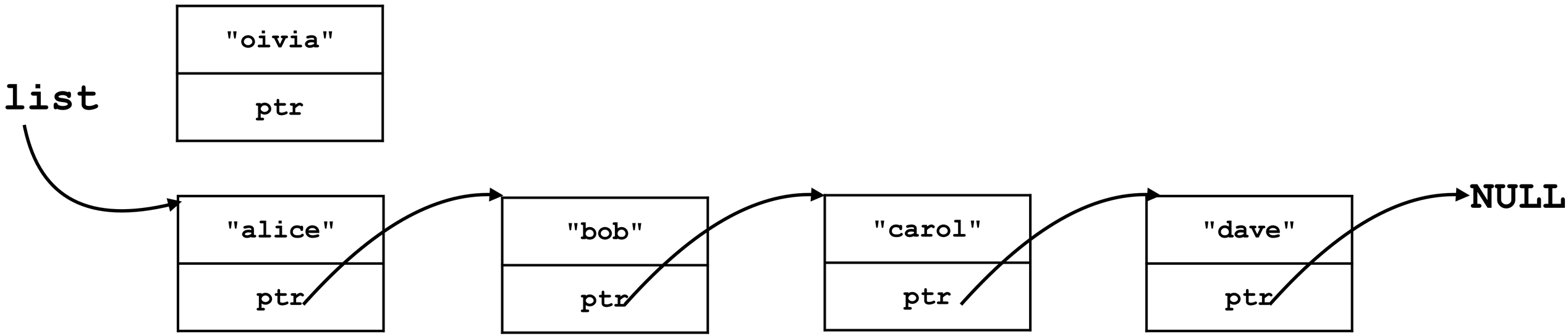
- We want to add "Olivia" to be front.



Linked Lists

`prepend`

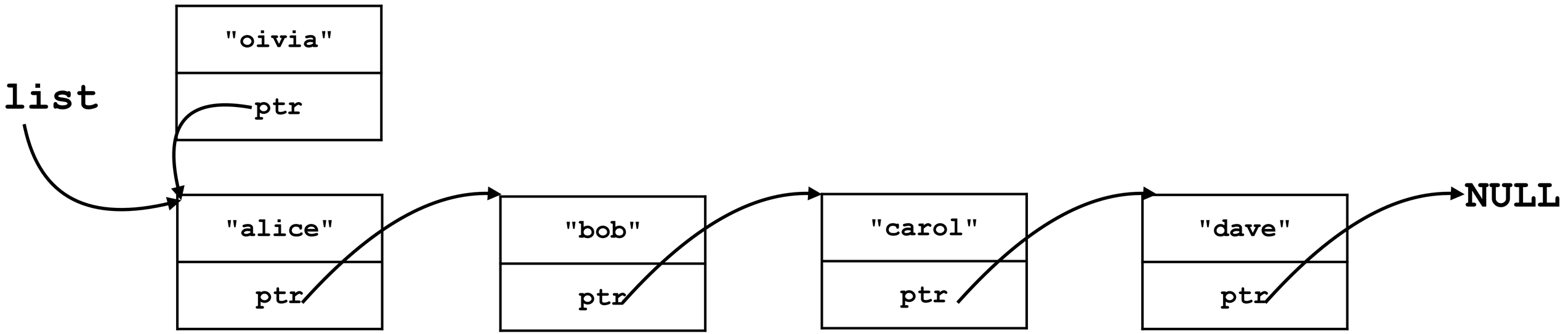
- We want to add "Olivia" to be front.
  1. malloc a node with data "Olivia"



Linked Lists

`prepend`

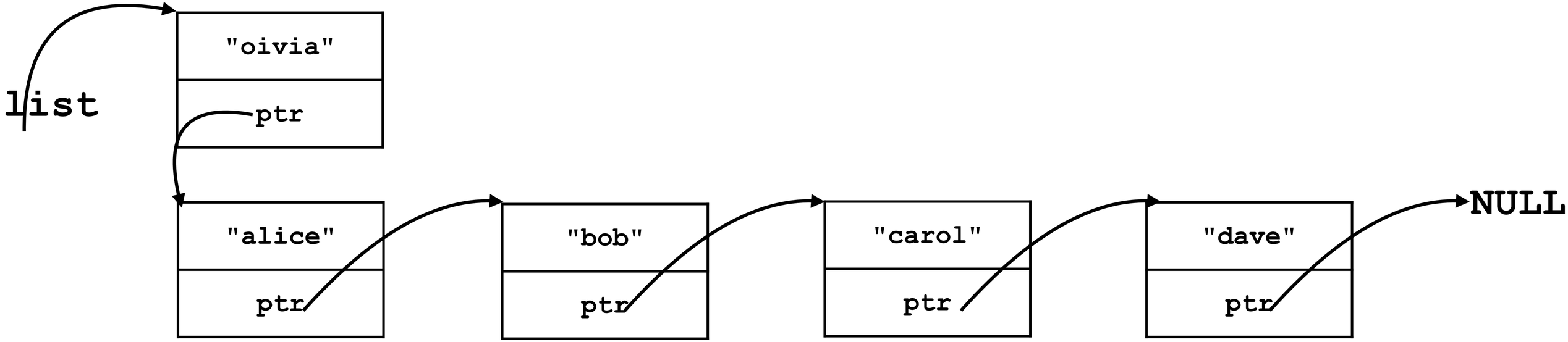
- We want to add "Olivia" to be front.
  2. Set the Olivia's pointer to the front



Linked Lists

`prepend`

- We want to add "Olivia" to be front.
    3. Update the front pointer to point to "Olivia"



Linked Lists

`prepend`

Linked Lists

`prepend`

- Prepending in a linked list is very efficient

Linked Lists

`prepend`

- Prepending in a linked list is very efficient
  - Remember array needs to shift everything by one

Linked Lists

`prepend`

- Prepending in a linked list is very efficient
  - Remember array needs to shift everything by one
  - The longer the list, the slower prepending is. $O(n)$

Linked Lists

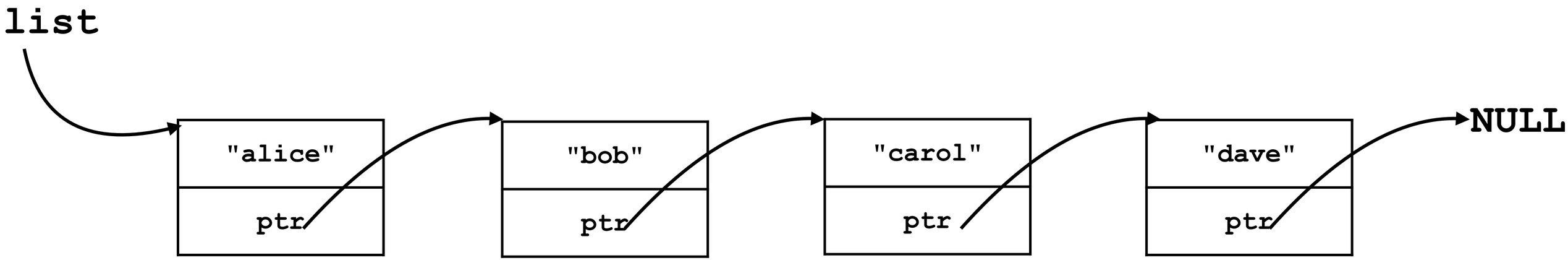
`prepend`

- Prepending in a linked list is very efficient
  - Remember array needs to shift everything by one
  - The longer the list, the slower prepending is. $O(n)$
  - Linked list does the same amount of work regardless of how long the list is. $O(1)$

Linked Lists

`len()`

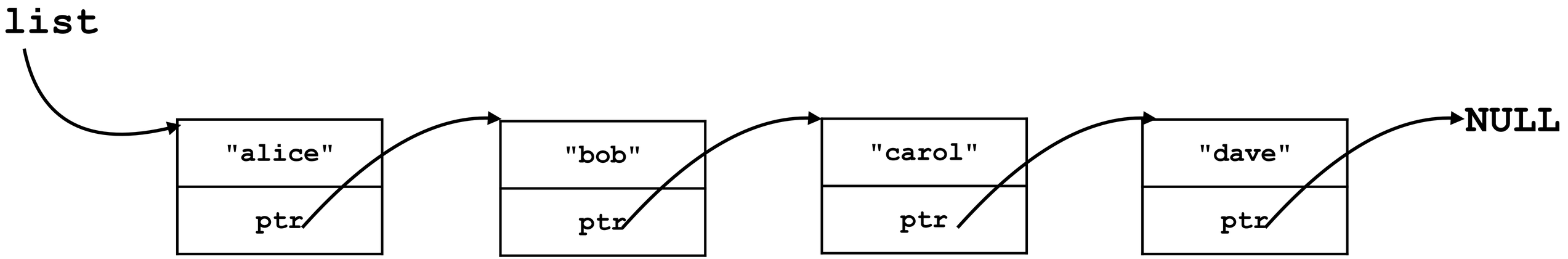
- What if we want to find the length of the list?



Linked Lists

`len()`

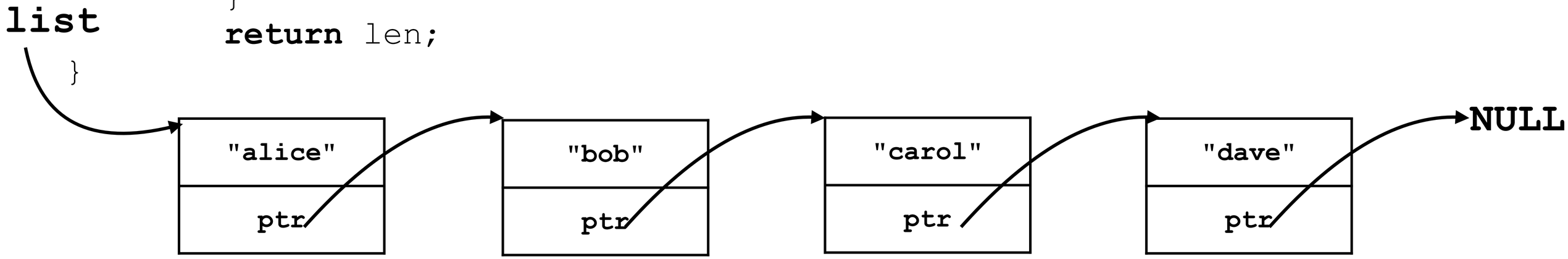
- What if we want to find the length of the list?
- Just following the links until we reach **`NULL`**, and count along the way



Linked Lists

len()

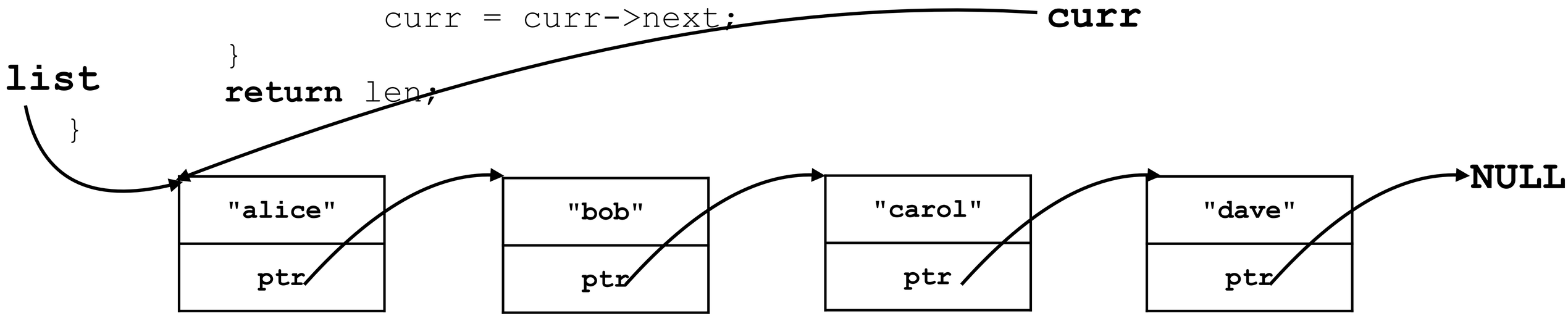
```
int llist_len(struct llist *list)
{
        struct llist *curr = list;
        int len = 0;
        while (curr != NULL) {
                len += 1;
                curr = curr->next;
        }
        return len;
}
```



Linked Lists

len()

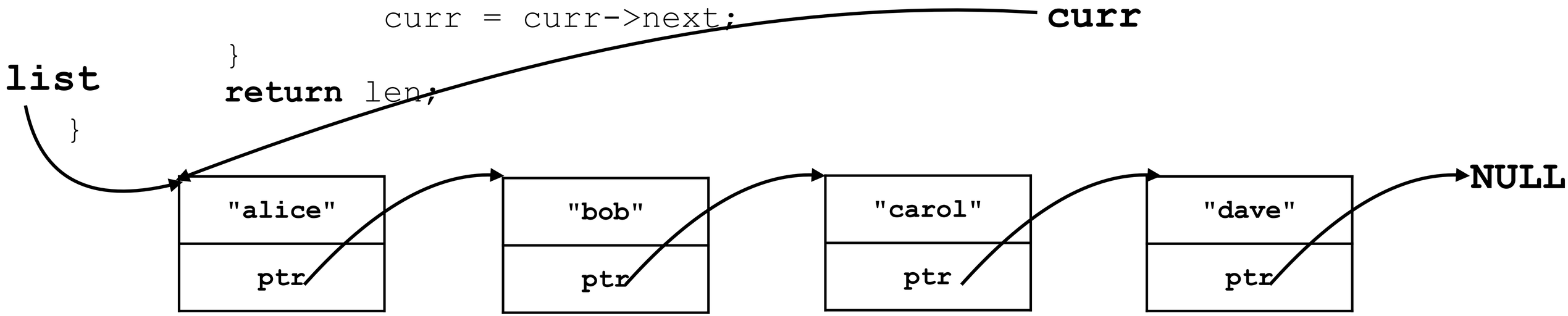
```
int llist_len(struct llist *list)
{
        struct llist *curr = list;
        int len = 0;
        while (curr != NULL) {
                len += 1;
                curr = curr->next;
        }
        return len;
}
```



Linked Lists

len()

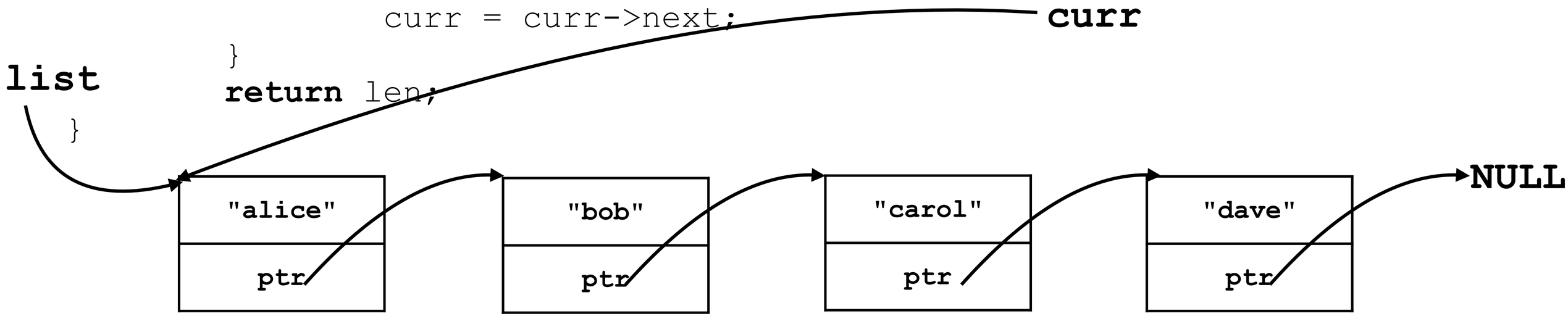
```
int llist_len(struct llist *list)
{
        struct llist *curr = list;
        int len = 0;
        while (curr != NULL) {
                len += 1;
                curr = curr->next;
        }
        return len;
}
```



Linked Lists

len()

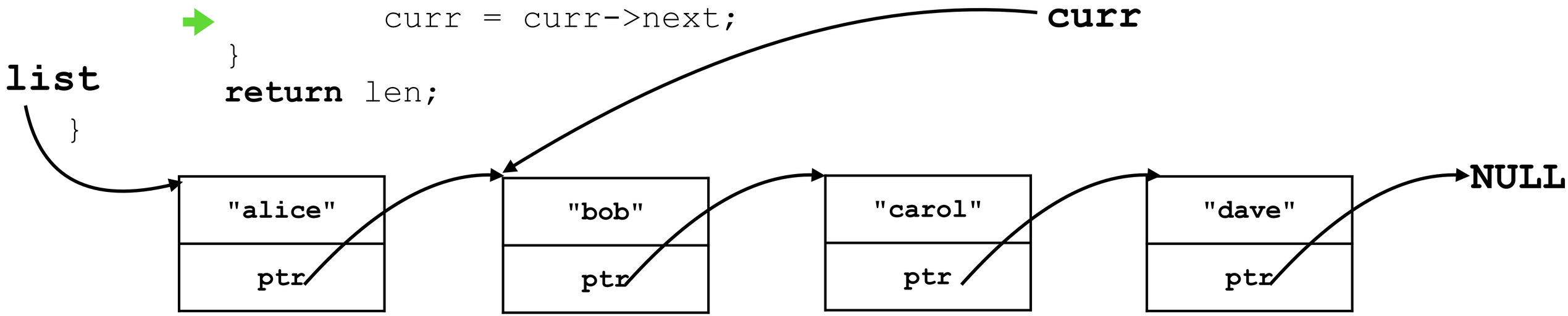
```
int llist_len(struct llist *list)
{
        struct llist *curr = list;
        int len = 0;
        while (curr != NULL) {
                len += 1;
                curr = curr->next;
        }
        return len;
}
```



Linked Lists

len()

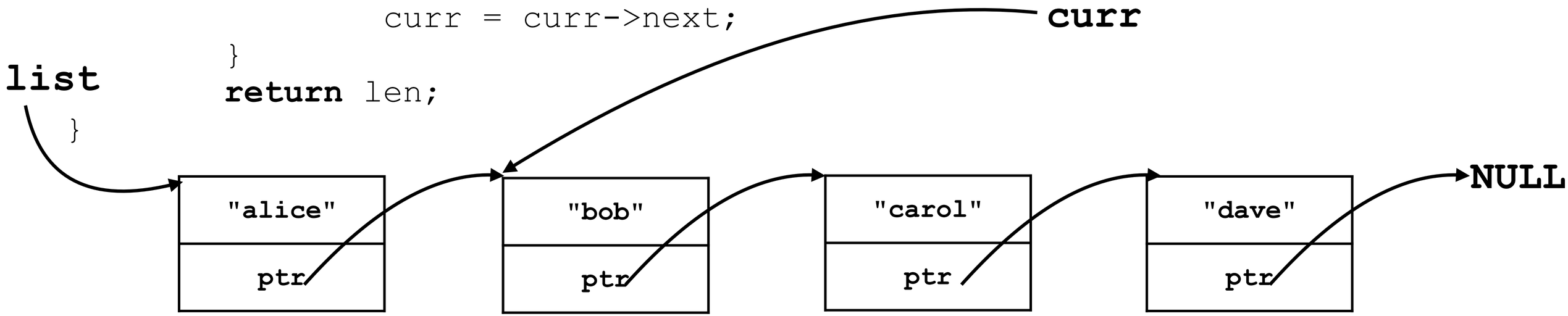
```
int llist_len(struct llist *list)
{
        struct llist *curr = list;
        int len = 0;
        while (curr != NULL) {
                len += 1;
                curr = curr->next;
        }
        return len;
}
```



Linked Lists

len()

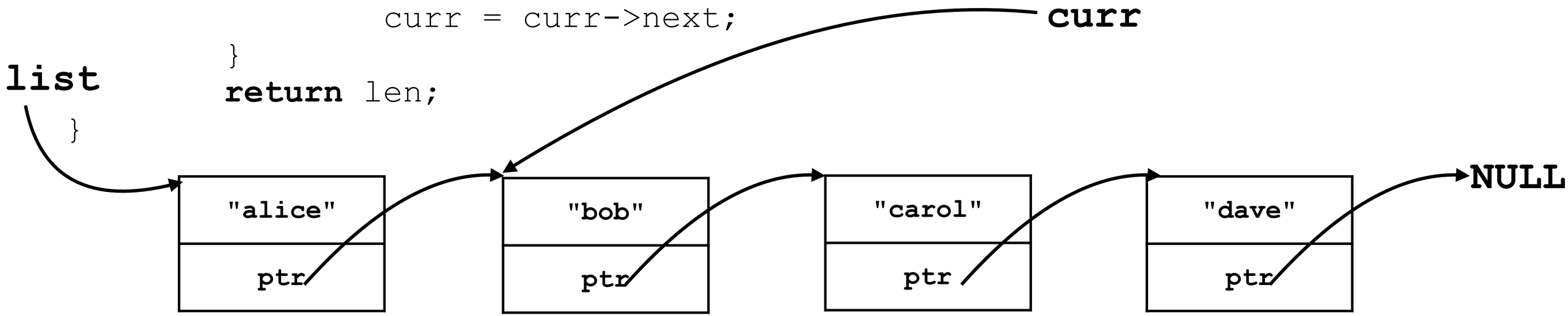
```
int llist_len(struct llist *list)
{
        struct llist *curr = list;
        int len = 0;
        while (curr != NULL) {
                len += 1;
                curr = curr->next;
        }
        return len;
}
```



Linked Lists

len()

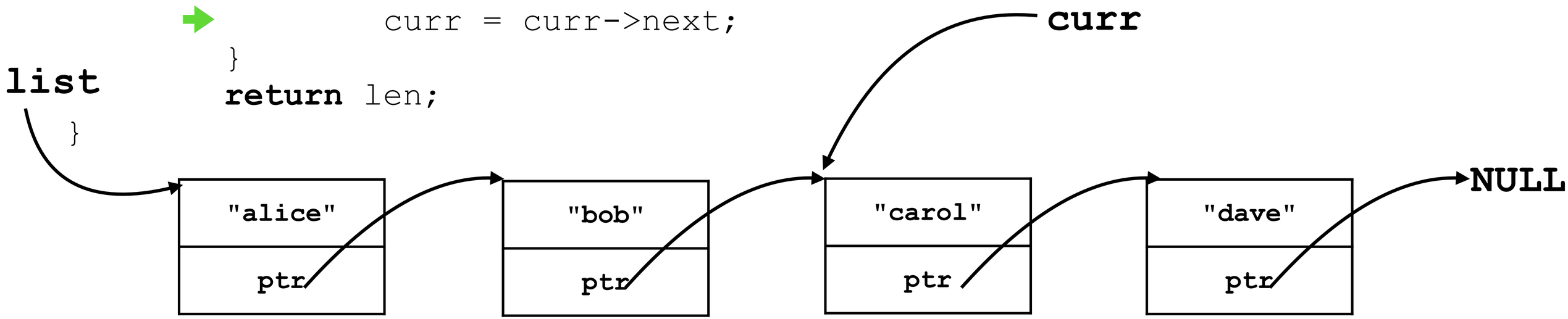
```
int llist_len(struct llist *list)
{
        struct llist *curr = list;
        int len = 0;
        while (curr != NULL) {
                len += 1;
                curr = curr->next;
        }
        return len;
}
```



Linked Lists

len()

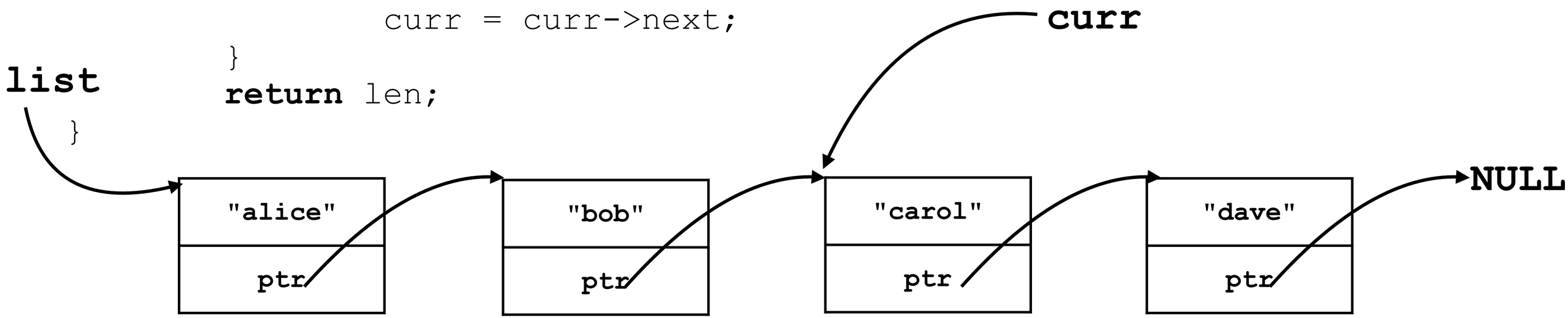
```
int llist_len(struct llist *list)
{
        struct llist *curr = list;
        int len = 0;
        while (curr != NULL) {
                len += 1;
                curr = curr->next;
        }
        return len;
}
```



Linked Lists

len()

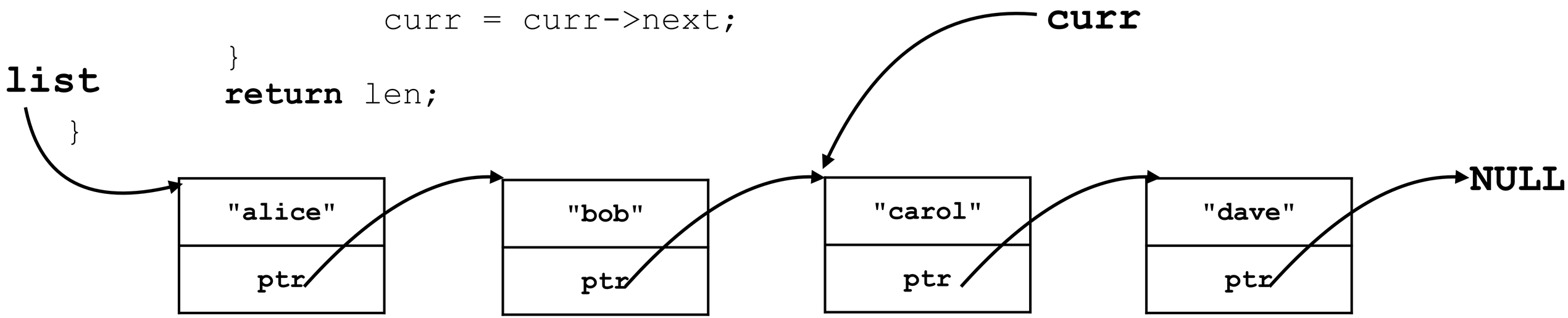
```
int llist_len(struct llist *list)
{
        struct llist *curr = list;
        int len = 0;
        while (curr != NULL) {
                len += 1;
                curr = curr->next;
        }
        return len;
}
```



Linked Lists

len()

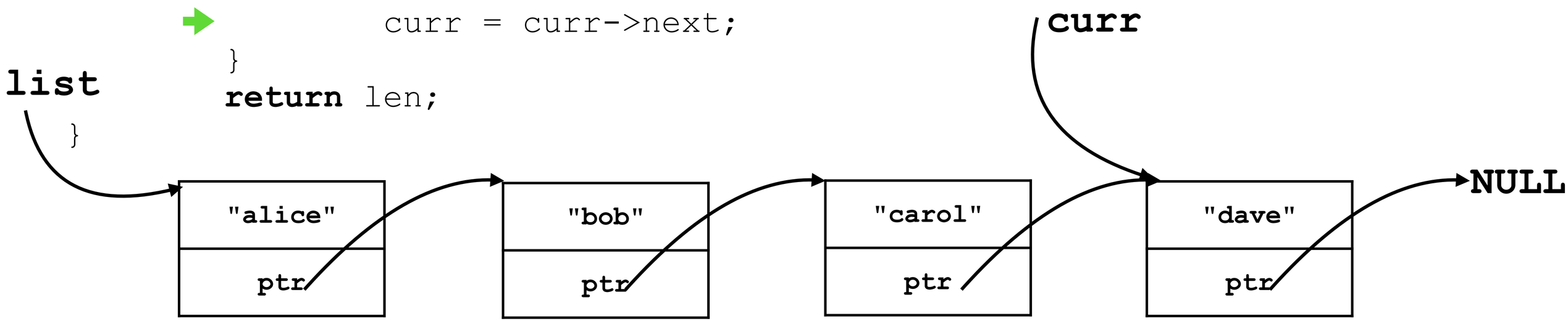
```
int llist_len(struct llist *list)
{
        struct llist *curr = list;
        int len = 0;
        while (curr != NULL) {
                len += 1;
                curr = curr->next;
        }
        return len;
}
```



Linked Lists

len()

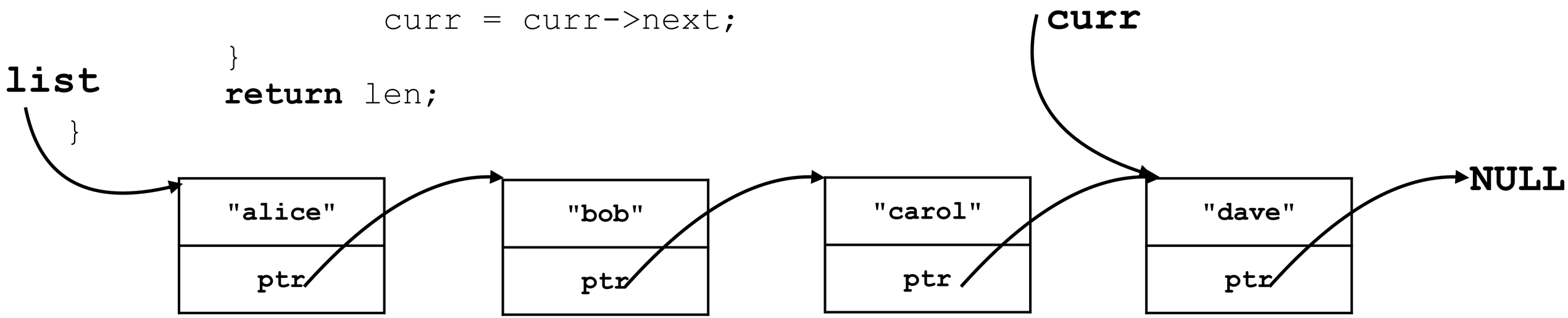
```
int llist_len(struct llist *list)
{
        struct llist *curr = list;
        int len = 0;
        while (curr != NULL) {
                len += 1;
                curr = curr->next;
        }
        return len;
}
```



Linked Lists

len()

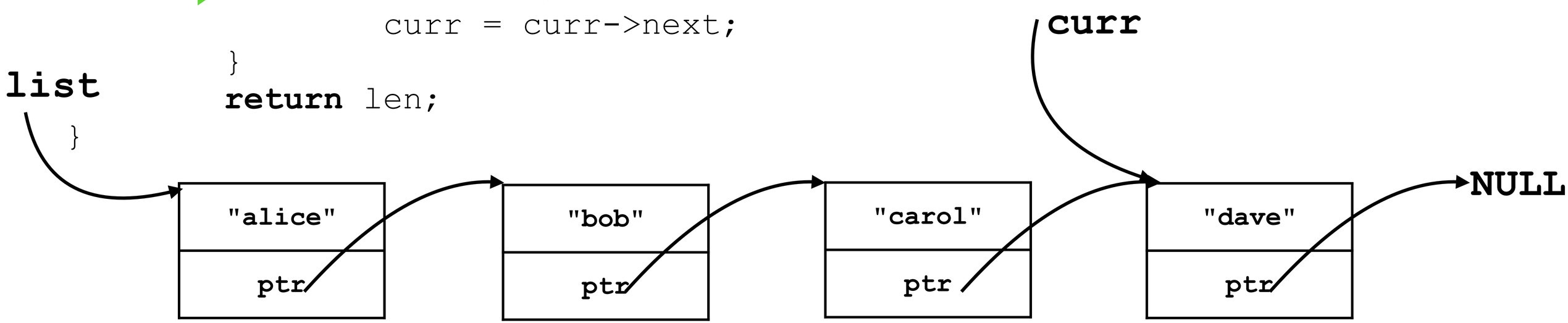
```
int llist_len(struct llist *list)
{
        struct llist *curr = list;
        int len = 0;
        while (curr != NULL) {
                len += 1;
                curr = curr->next;
        }
        return len;
}
```



Linked Lists

len()

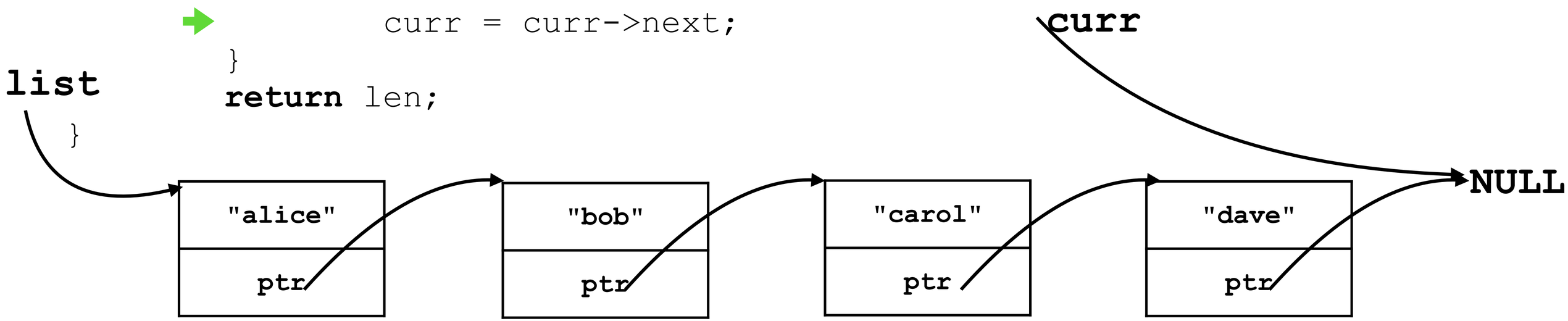
```
int llist_len(struct llist *list)
{
        struct llist *curr = list;
        int len = 0;
        while (curr != NULL) {
                len += 1;
                curr = curr->next;
        }
        return len;
}
```



Linked Lists

len()

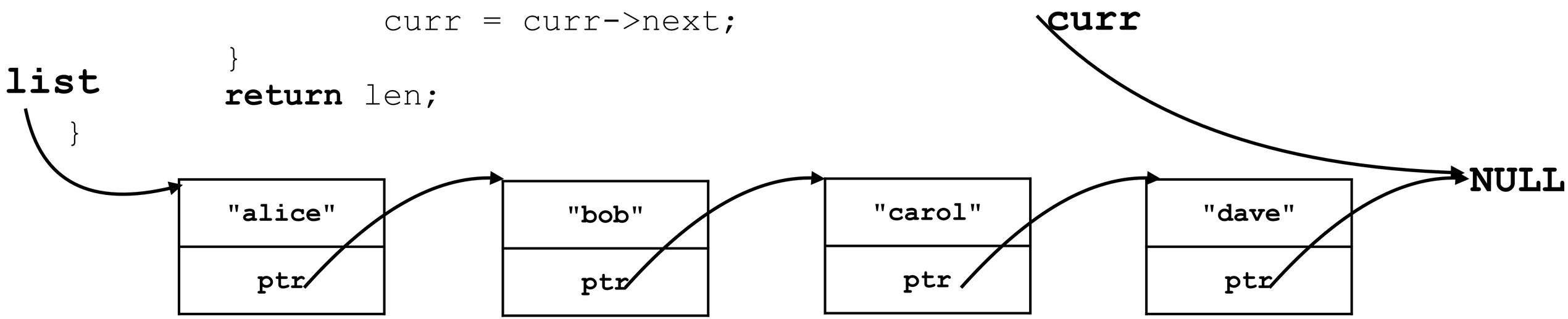
```
int llist_len(struct llist *list)
{
        struct llist *curr = list;
        int len = 0;
        while (curr != NULL) {
                len += 1;
                curr = curr->next;
        }
        return len;
}
```



Linked Lists

len()

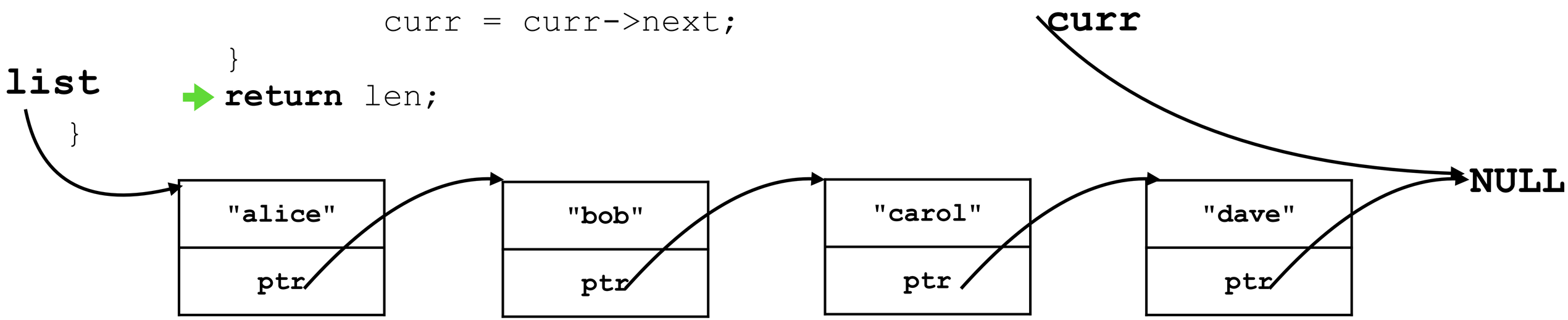
```
int llist_len(struct llist *list)
{
        struct llist *curr = list;
        int len = 0;
        while (curr != NULL) {
                len += 1;
                curr = curr->next;
        }
        return len;
}
```



Linked Lists

len()

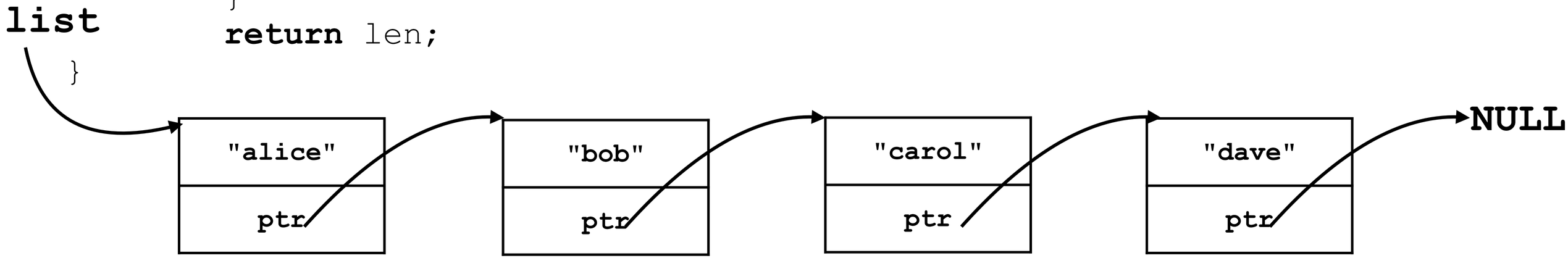
```
int llist_len(struct llist *list)
{
        struct llist *curr = list;
        int len = 0;
        while (curr != NULL) {
                len += 1;
                curr = curr->next;
        }
        return len;
}
```



Linked Lists

len()

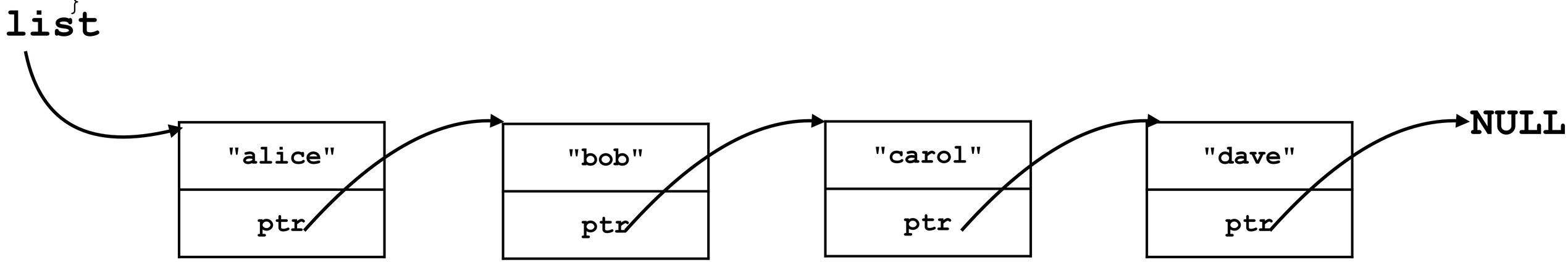
```
int llist_len(struct llist *list)
{
        int len = 0;
        for (struct llist *curr = list;
             curr != NULL;
             curr = curr->next) {
            len += 1;
        }
        return len;
}
```



Linked Lists

len()

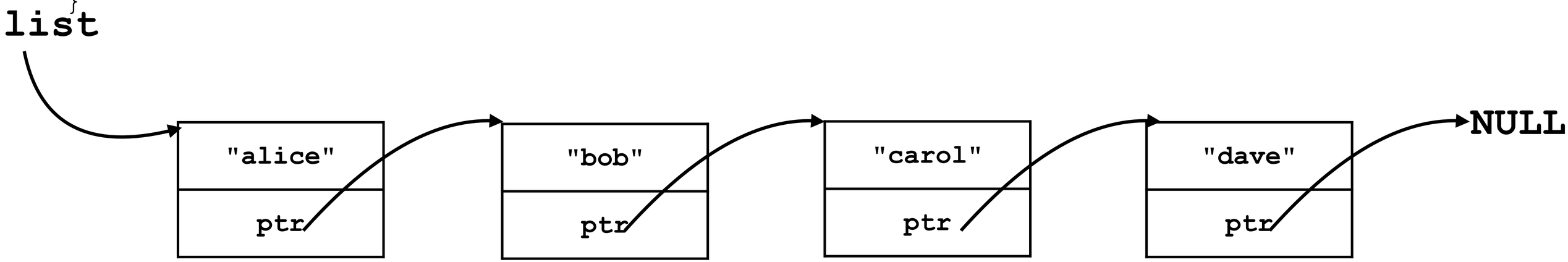
```
int llist_len(struct llist *list)
{
        int len = 0;
        for (struct llist *curr = list;
             curr != NULL;
             curr = curr->next, len += 1);
        return len;
}
```



Linked Lists

`len()` recursively

```
int llist_len(struct llist *list)
{
        if (list == NULL) {
                return 0;
        } else {
                return 1 + llist_len(list->next);
        }
}
```



Linked Lists

`len`

Linked Lists

`len`

- Arrays have to remember the length; cannot calculate the length $O(1)$

Linked Lists

`len`

- Arrays have to remember the length; cannot calculate the length $O(1)$
- Linked lists can still keep track of the length

Linked Lists

`len`

- Arrays have to remember the length; cannot calculate the length $O(1)$
- Linked lists can still keep track of the length
  - Keep a counter for insert/delete

Linked Lists

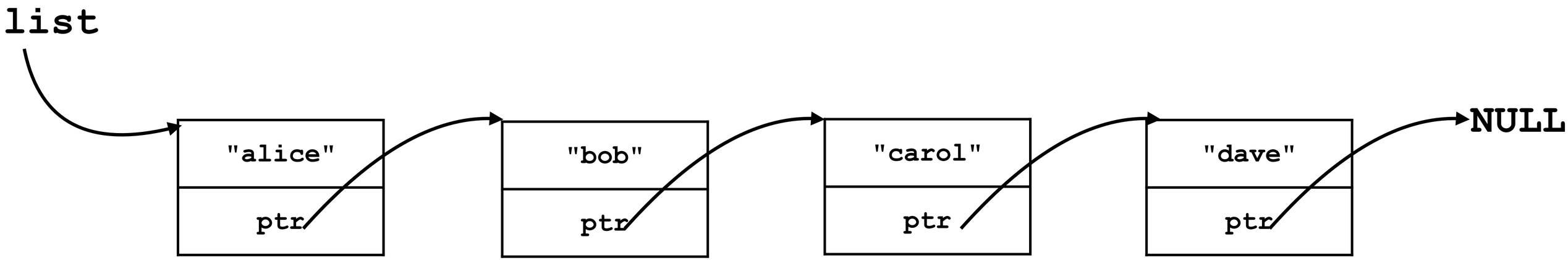
`len`

- Arrays have to remember the length; cannot calculate the length $O(1)$
- Linked lists can still keep track of the length
  - Keep a counter for insert/delete
  - But calculating takes $O(n)$

Linked Lists

`append()`

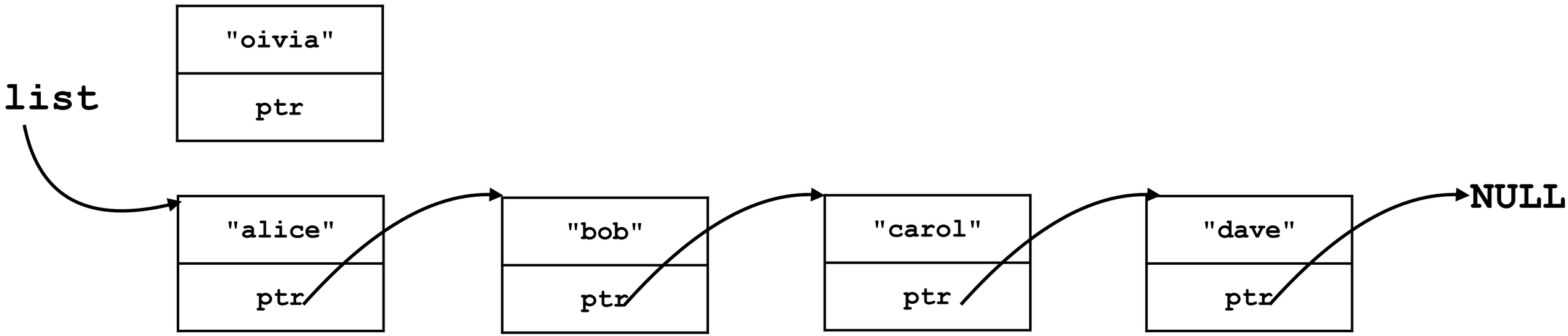
- We want to add "Olivia" to the back of the list



Linked Lists

`append()`

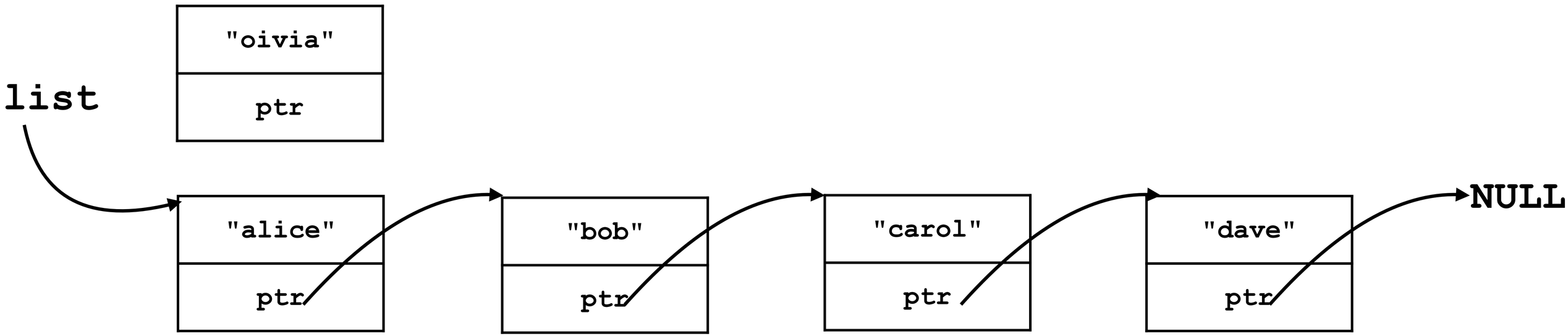
- We want to add "Olivia" to the back of the list
    1. malloc



Linked Lists

`append()`

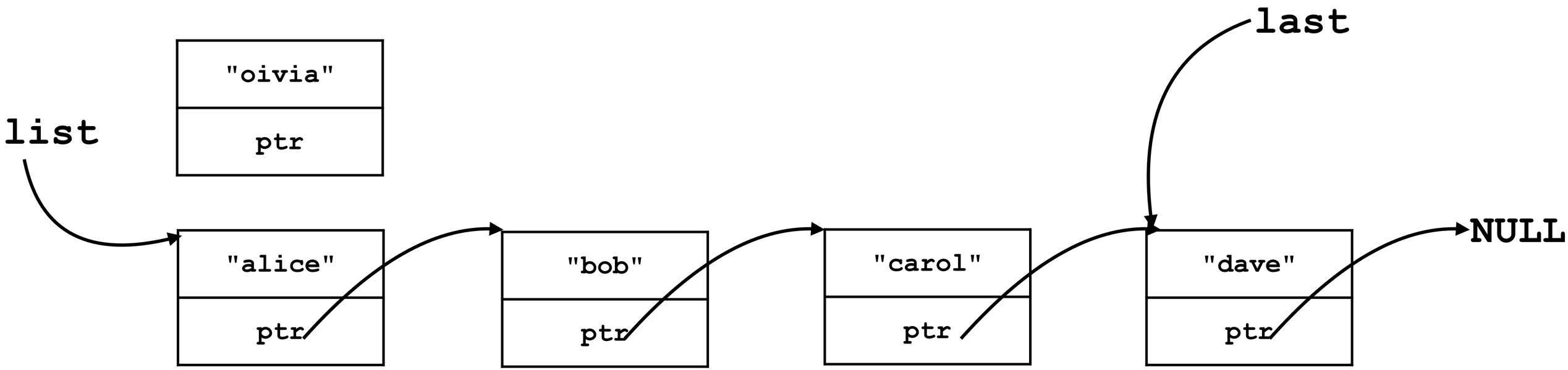
- We want to add "Olivia" to the back of the list
  2. find the last node in this list



Linked Lists

`append()`

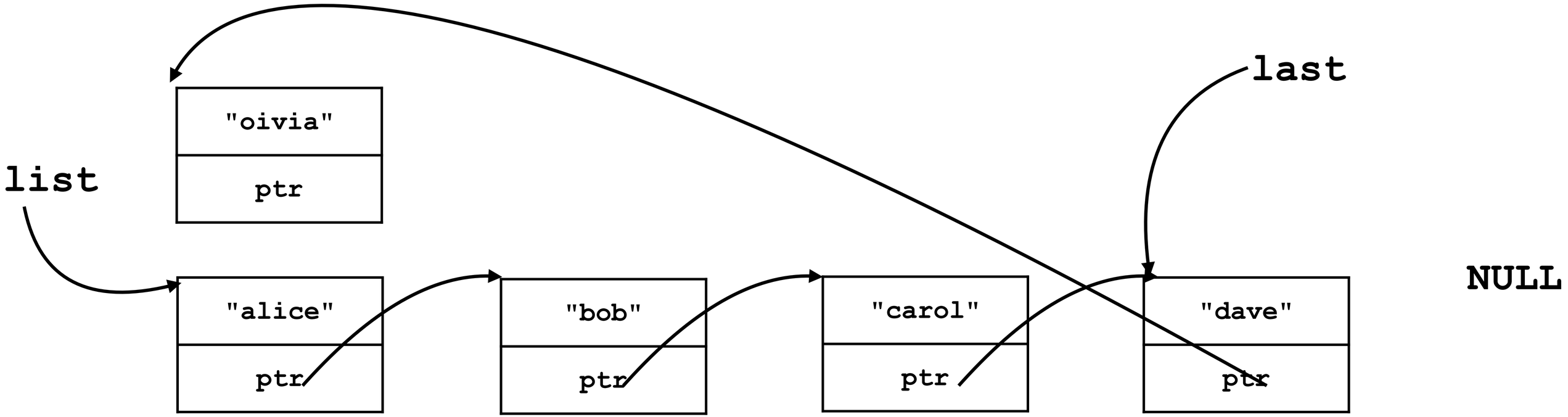
- We want to add "Olivia" to the back of the list
  2. find the last node in this list



Linked Lists

`append()`

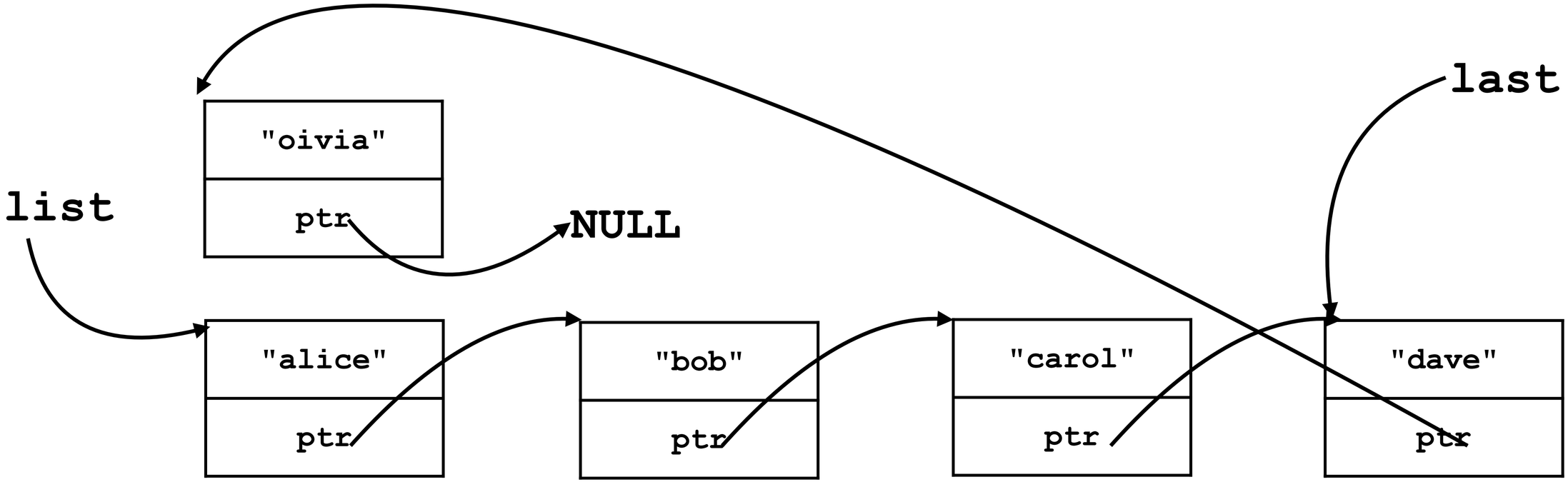
- We want to add "Olivia" to the back of the list
  3. set `last->next = olivia`



Linked Lists

`append()`

- We want to add "Olivia" to the back of the list
  4. set olivia's pointer to `NULL`



Linked Lists

`append`

Linked Lists

`append`

- Appending to list involves finding the last node

Linked Lists

`append`

- Appending to list involves finding the last node
  - If we know the last node, $O(1)$

Linked Lists

`append`

- Appending to list involves finding the last node
  - If we know the last node, $O(1)$
  - Finding the last node, $O(n)$

Linked Lists

`append`

- Appending to list involves finding the last node
  - If we know the last node, $O(1)$
  - Finding the last node, $O(n)$
- How about array list?

Linked Lists

`append`

- Appending to list involves finding the last node
  - If we know the last node, $O(1)$
  - Finding the last node, $O(n)$
- How about array list?
  - Finding the last node: $O(1)$, `end = start + length`

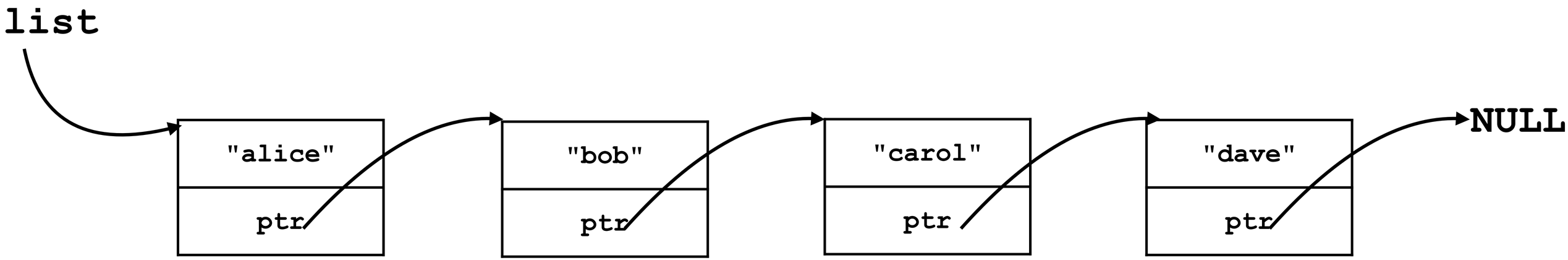
Linked Lists

`append`

- Appending to list involves finding the last node
  - If we know the last node, $O(1)$
  - Finding the last node, $O(n)$
- How about array list?
  - Finding the last node: $O(1)$, `end = start + length`
  - Inserting: $O(1)$ most of the time, $O(n)$ if unlucky

Linked Lists

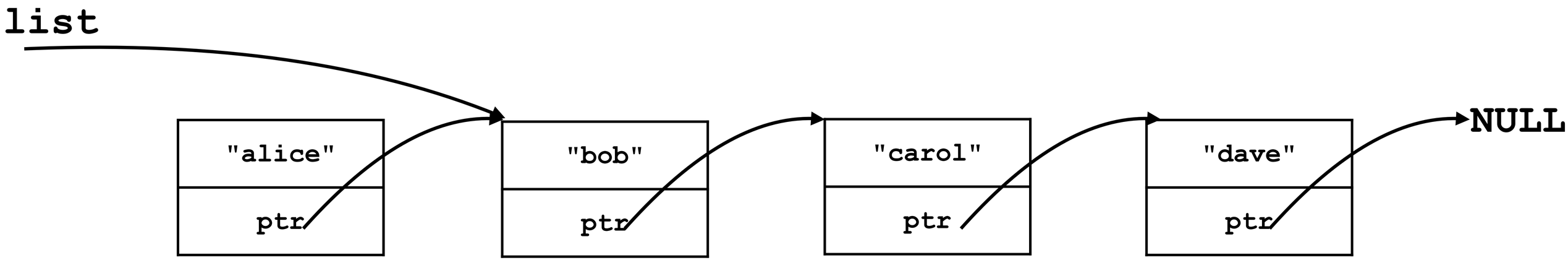
`remove front`



Linked Lists

`remove front`

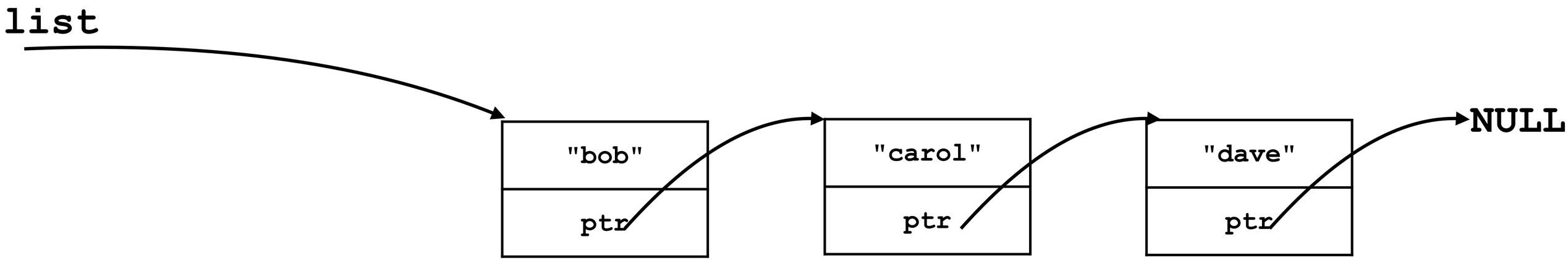
1. Move the front to front->next



Linked Lists

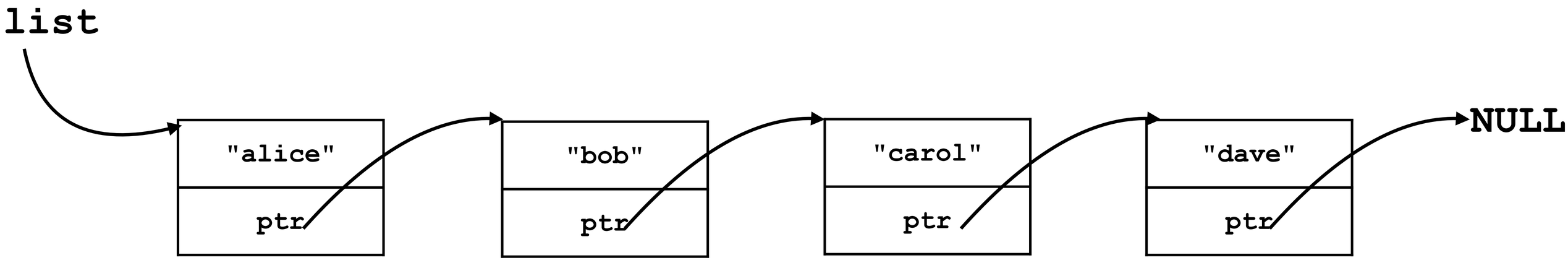
`remove front`

2. Free the first node



Linked Lists

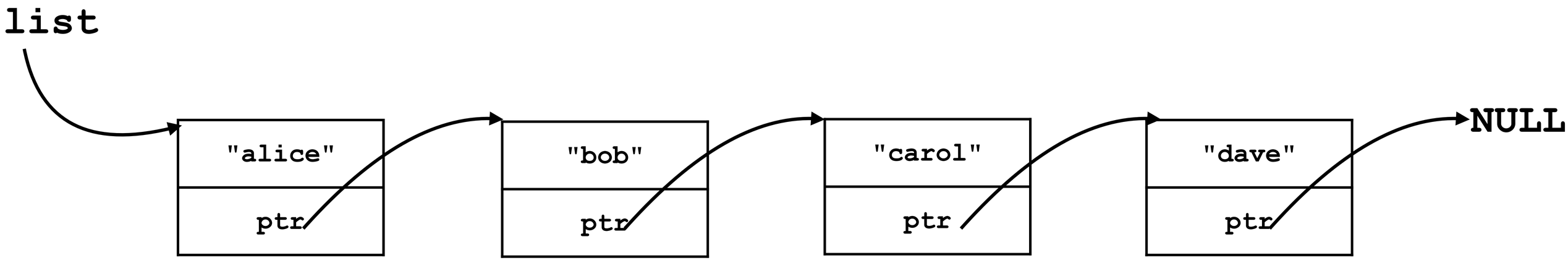
`remove middle`



Linked Lists

`remove middle`

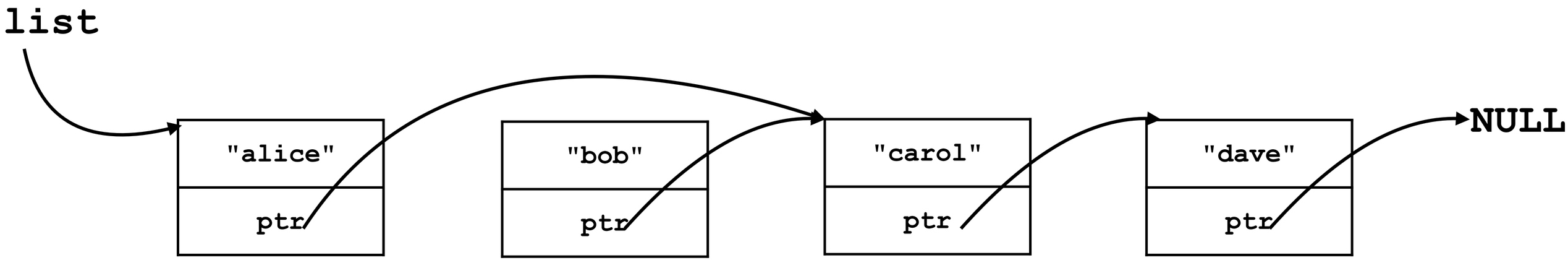
1. Move the previous pointer's next to current pointer's next



Linked Lists

`remove middle`

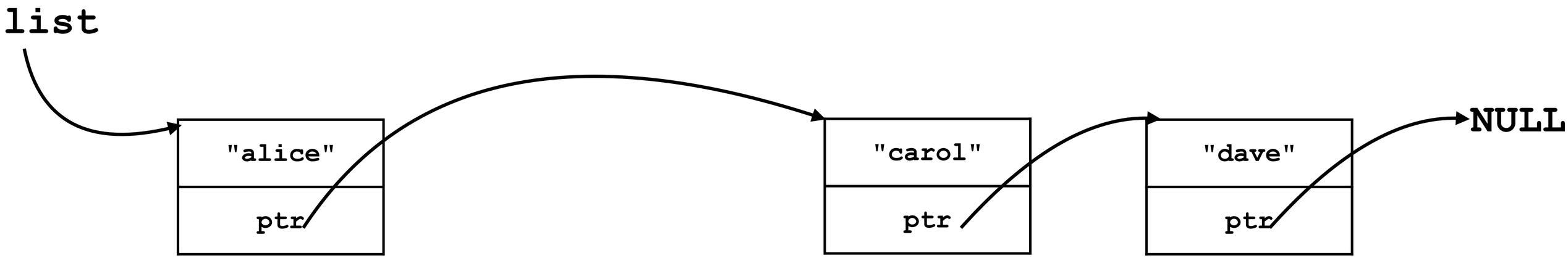
1. Move the previous pointer's next to current pointer's next



Linked Lists

`remove middle`

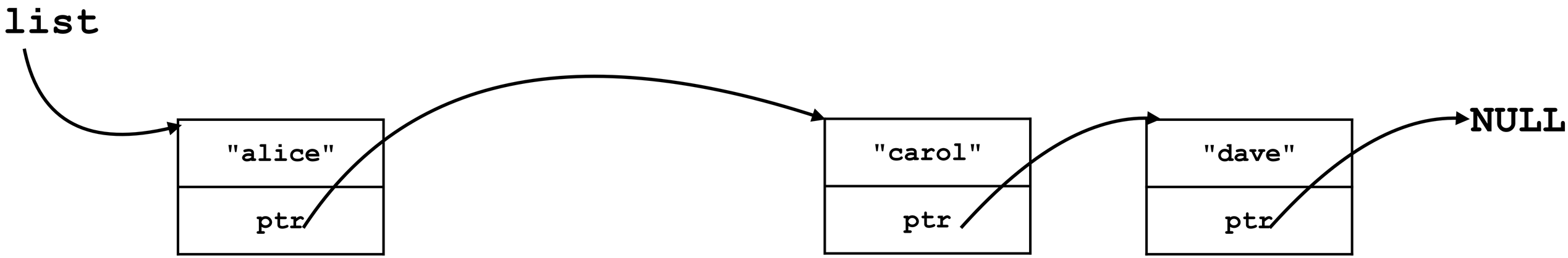
2. Free the node



Linked Lists

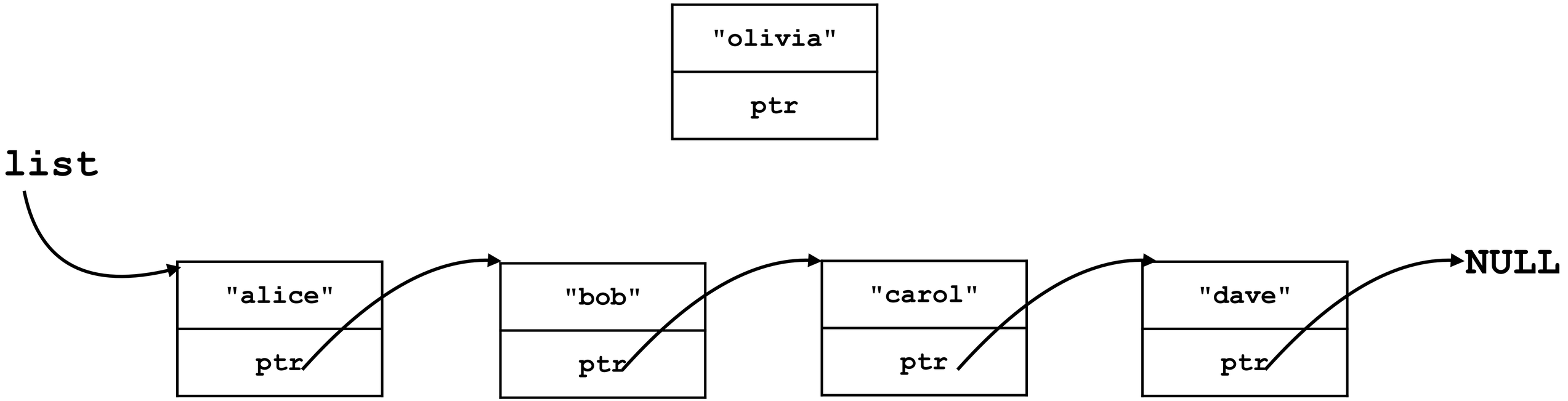
`remove last`

1. Set the second to last's `next` pointer to **`NULL`**



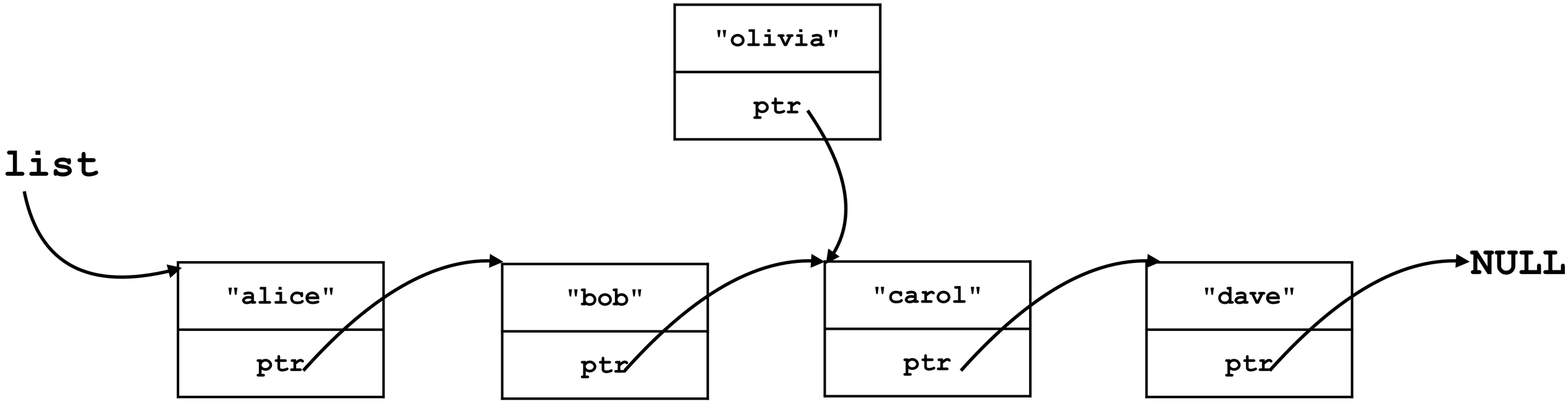
Linked Lists

`insert` works in similar way



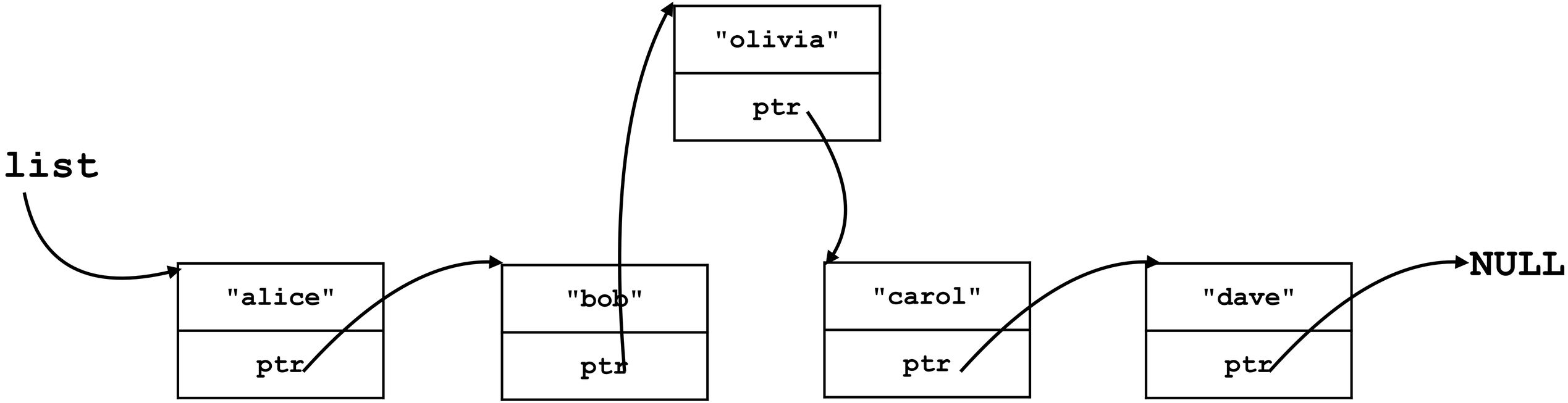
Linked Lists

`insert` works in similar way



Linked Lists

`insert` works in similar way



Linked Lists

`remove/insert`

Linked Lists

`remove/insert`

- Finding the index/item takes $O(n)$

Linked Lists

`remove/insert`

- Finding the index/item takes $O(n)$
- But if the node is found, removing does constant amount of work.

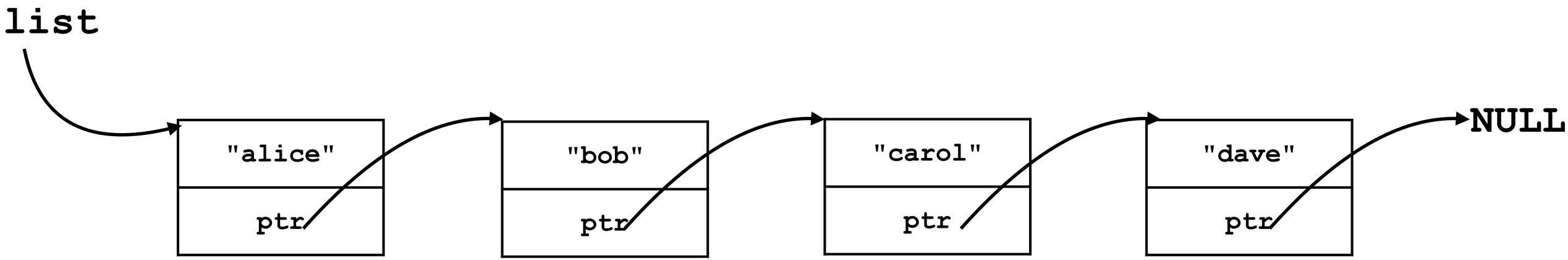
Linked Lists

`remove/insert`

- Finding the index/item takes $O(n)$
- But if the node is found, removing does constant amount of work.
- Array always needs to shift

Linked Lists

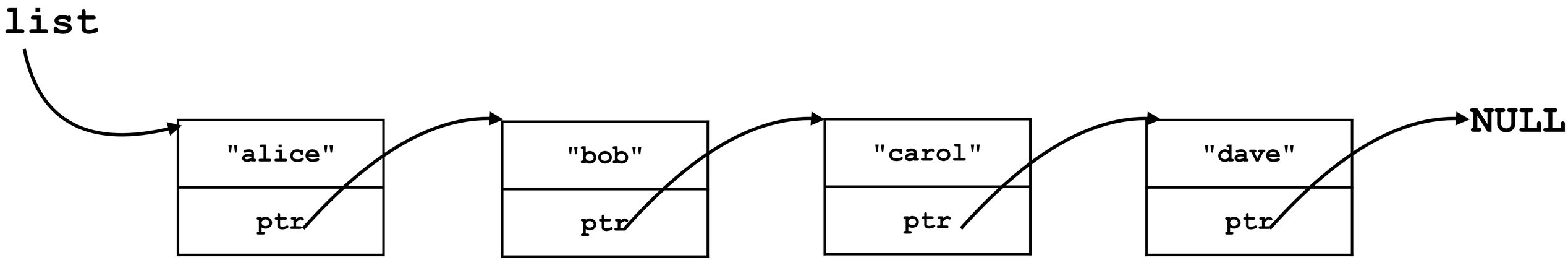
`data`'s type



Linked Lists

`data`'s type

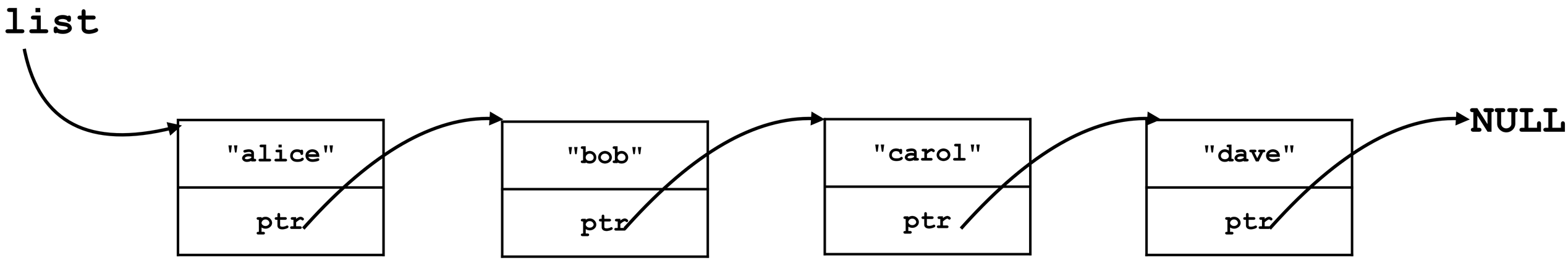
- What's the type of the data each node carries?



Linked Lists

`data`'s type

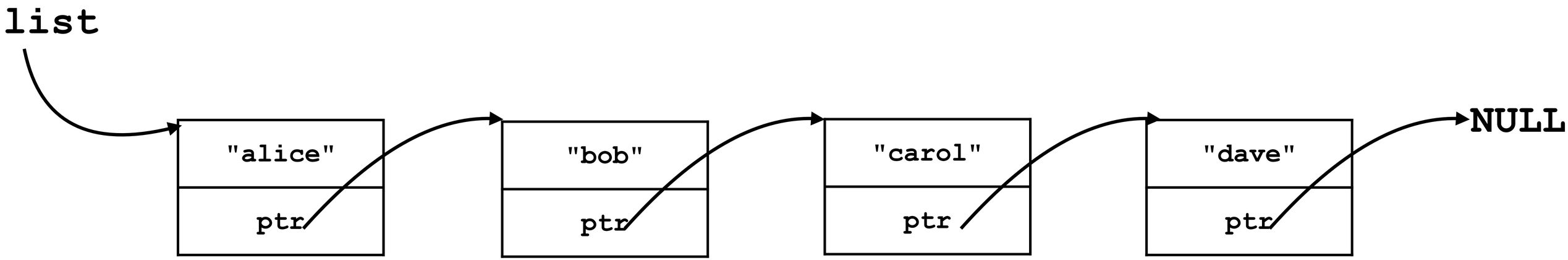
- What's the type of the data each node carries?
  - int



Linked Lists

`data`'s type

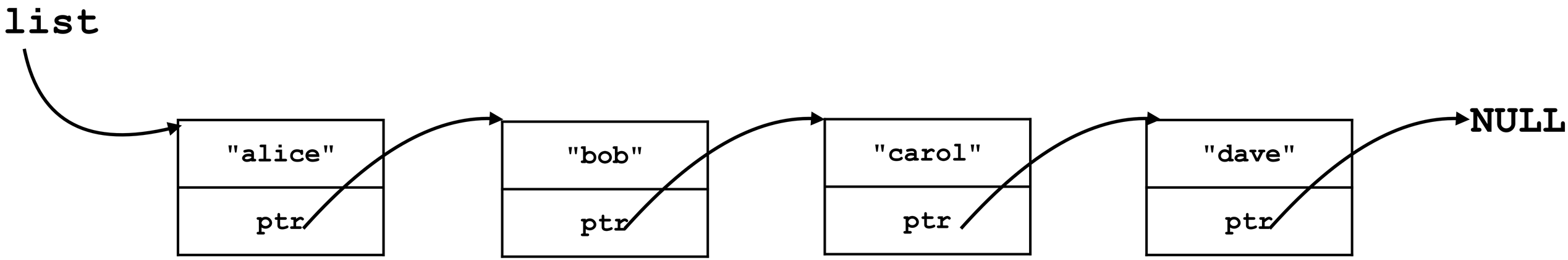
- What's the type of the data each node carries?
  - int
  - char *



Linked Lists

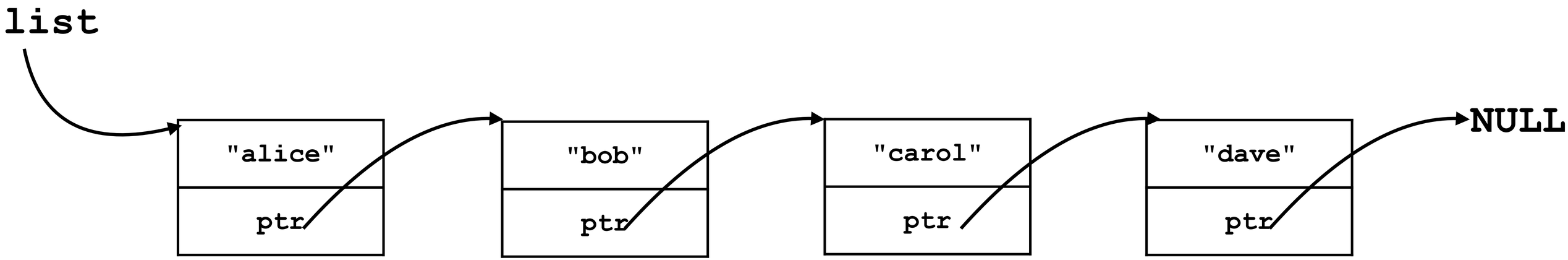
`data`'s type

- What's the type of the data each node carries?
  - int
  - char *
  - ...



Linked Lists

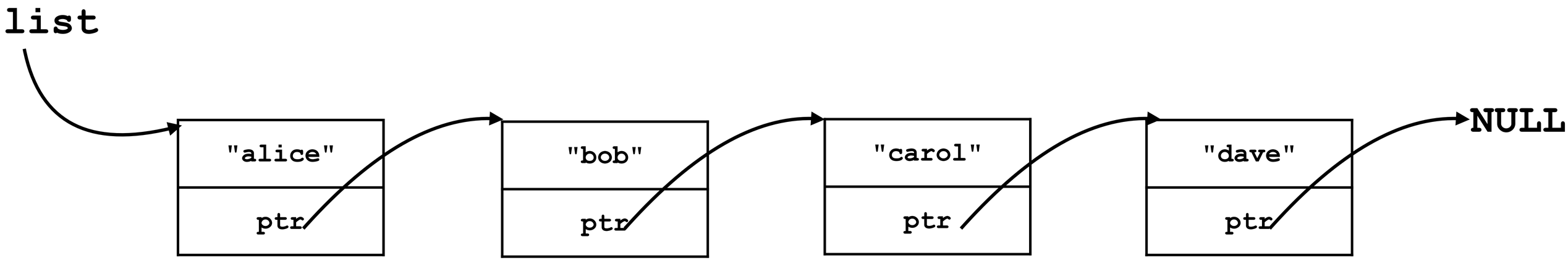
`data`'s type



Linked Lists

`data`'s type

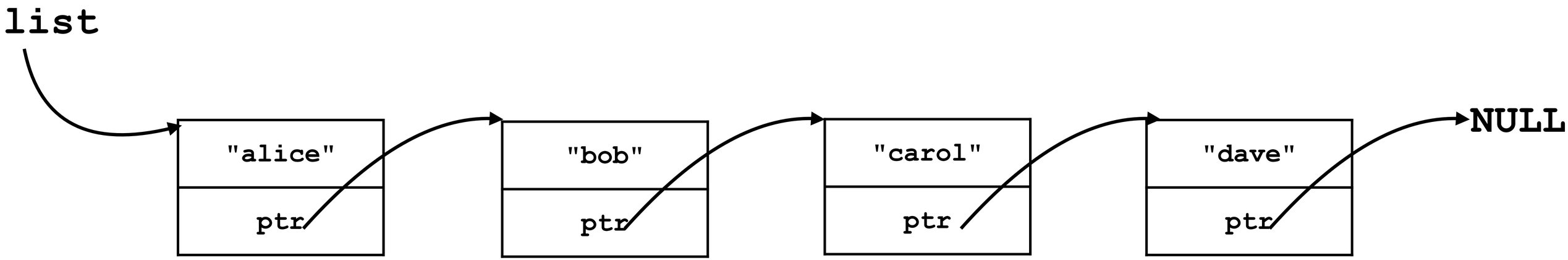
- What's the type of the data each node carries?



Linked Lists

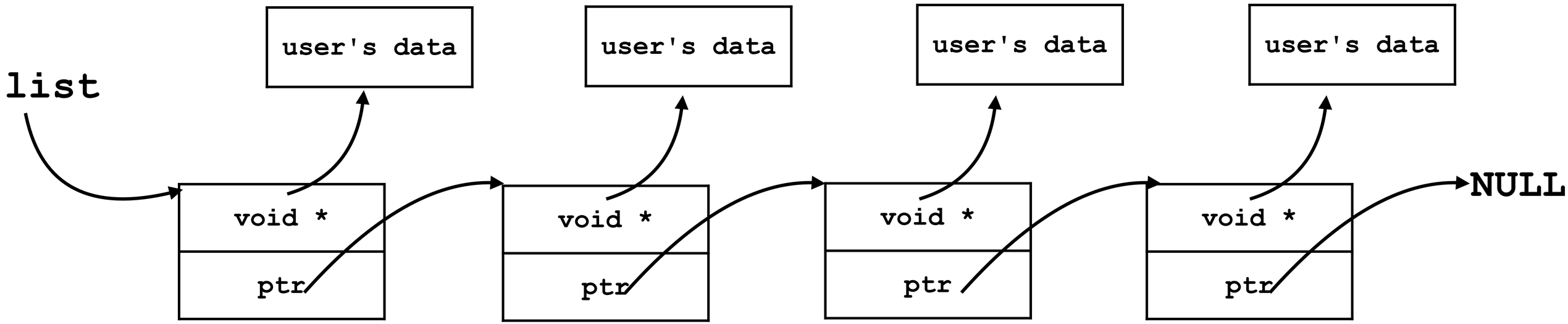
`data`'s type

- What's the type of the data each node carries?
  - Any pointer!



Linked Lists

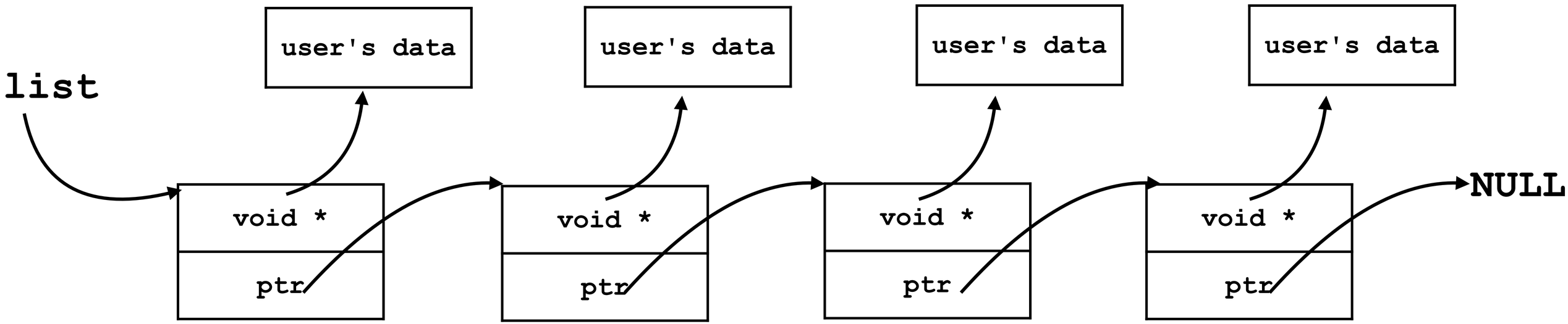
`data`'s type



Linked Lists

`data`'s type

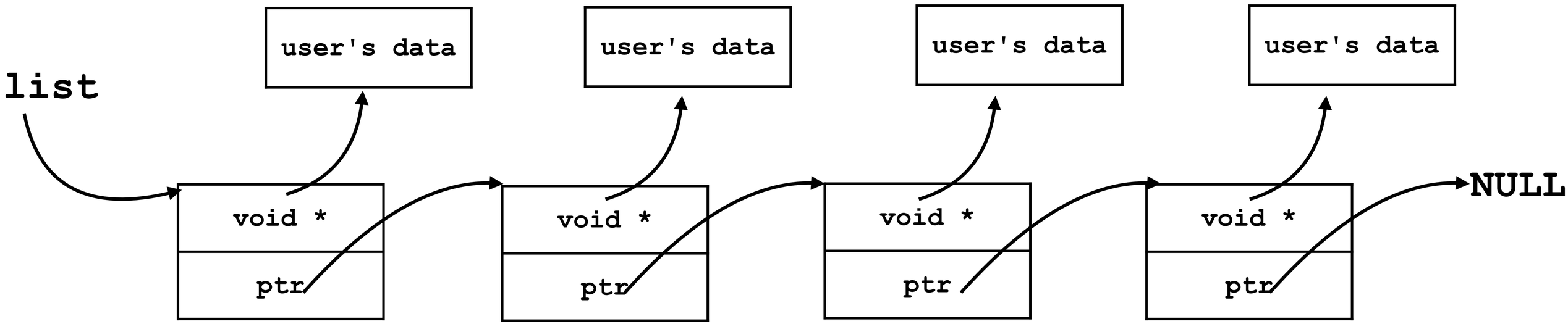
- What's the type of the data each node carries?



Linked Lists

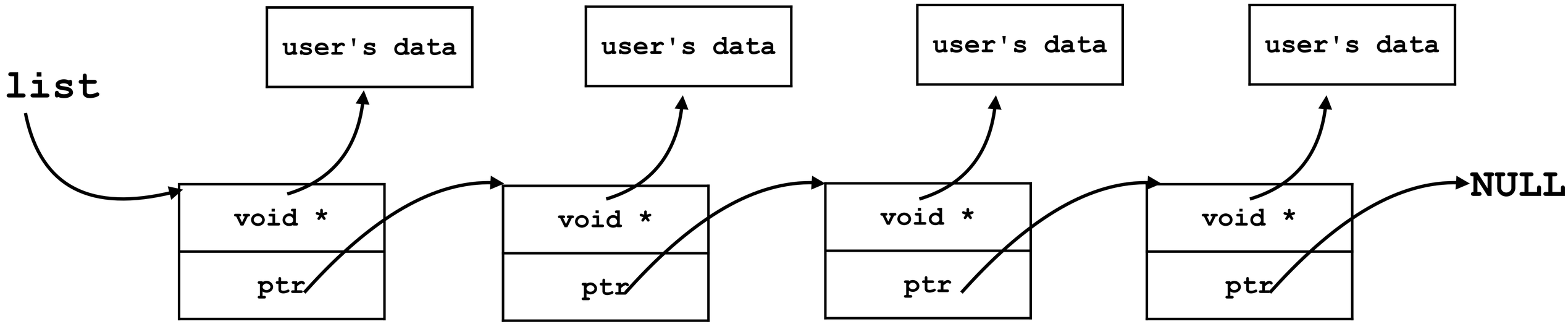
`data`'s type

- What's the type of the data each node carries?
  - Any pointer!



Linked Lists

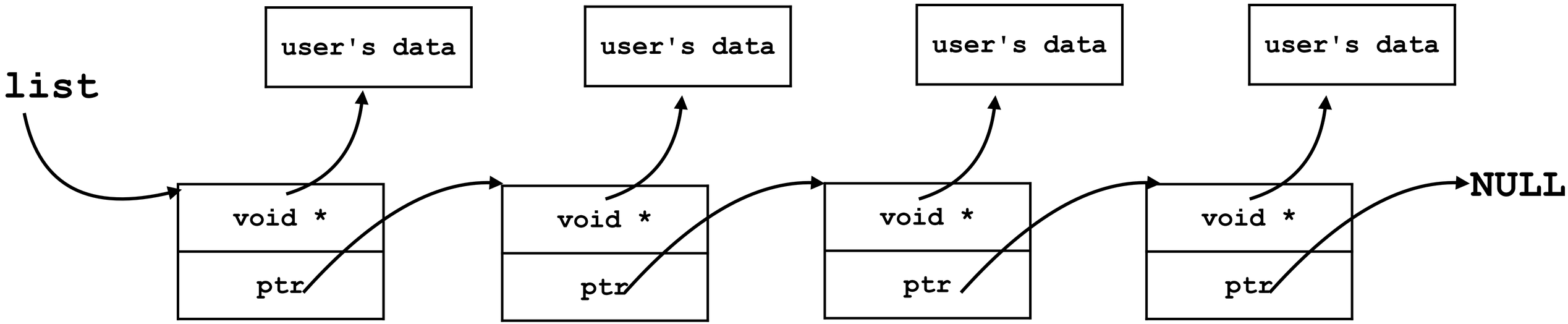
`data`'s type



Linked Lists

`data`'s type

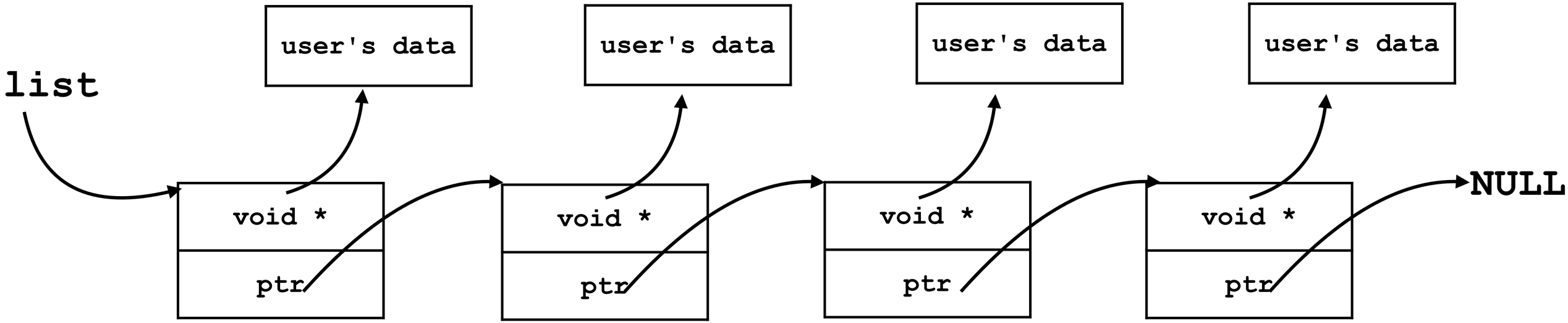
- What's the type of the data each node carries? Any pointer!



Linked Lists

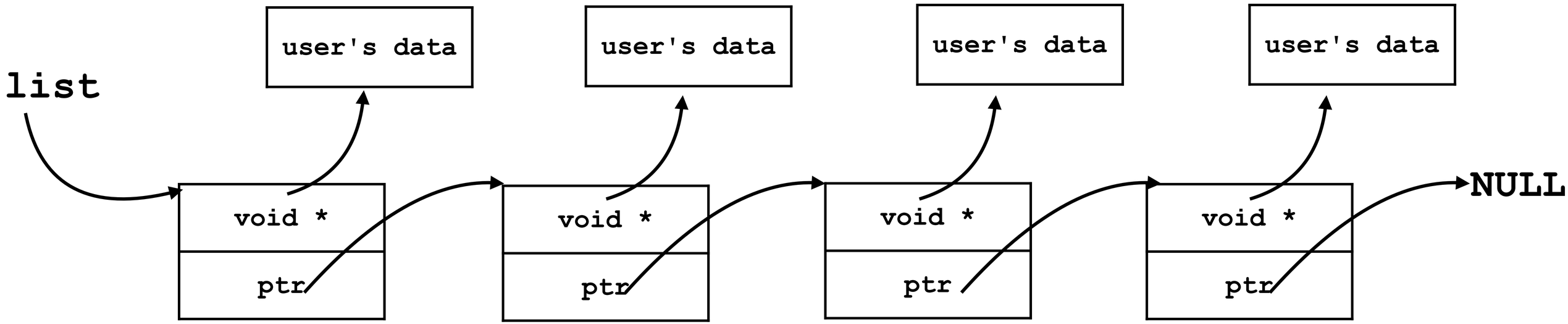
`data`'s type

- What's the type of the data each node carries? Any pointer!
- The data structure does not manage the memory that user's data use.



Linked Lists

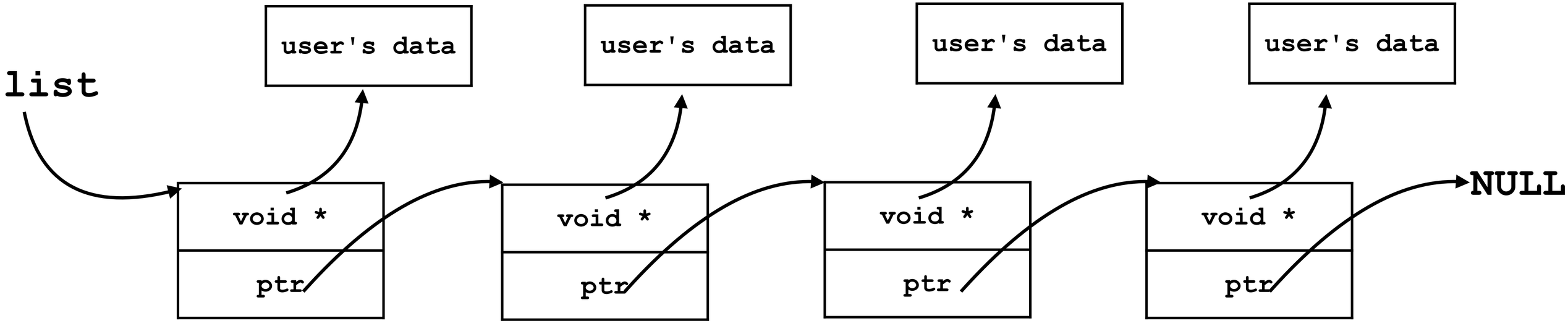
`data`'s type



Linked Lists

`data`'s type

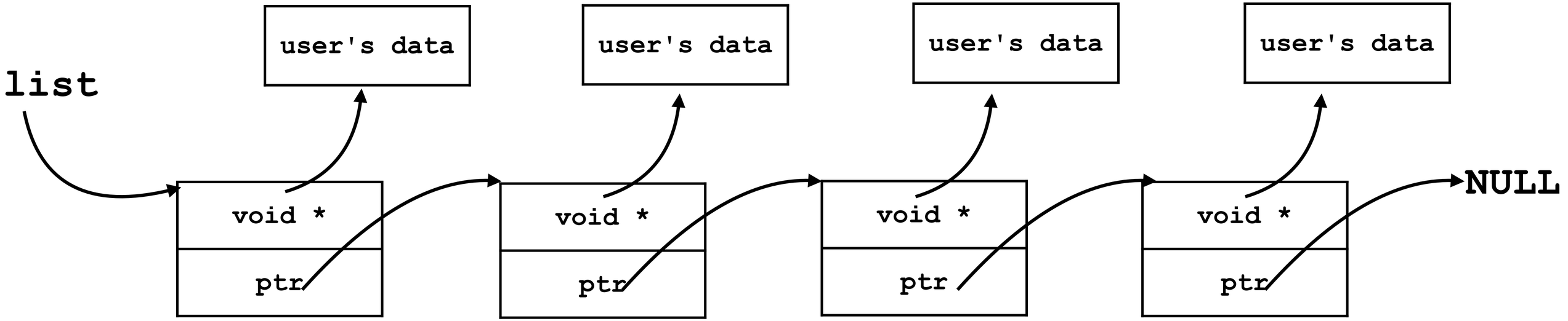
- What's the type of the data each node carries? Any pointer!



Linked Lists

`data`'s type

- What's the type of the data each node carries? Any pointer!
- The data structure does not manage the memory that user's data use. The user is responsible for freeing the `void *` pointers.



Linked Lists

`data`'s type

- What's the type of the data each node carries? Any pointer!
- The data structure does not manage the memory that user's data use. The user is responsible for freeing the `void *` pointers.
- We call this a *boxed data structure.*