Linked Lists

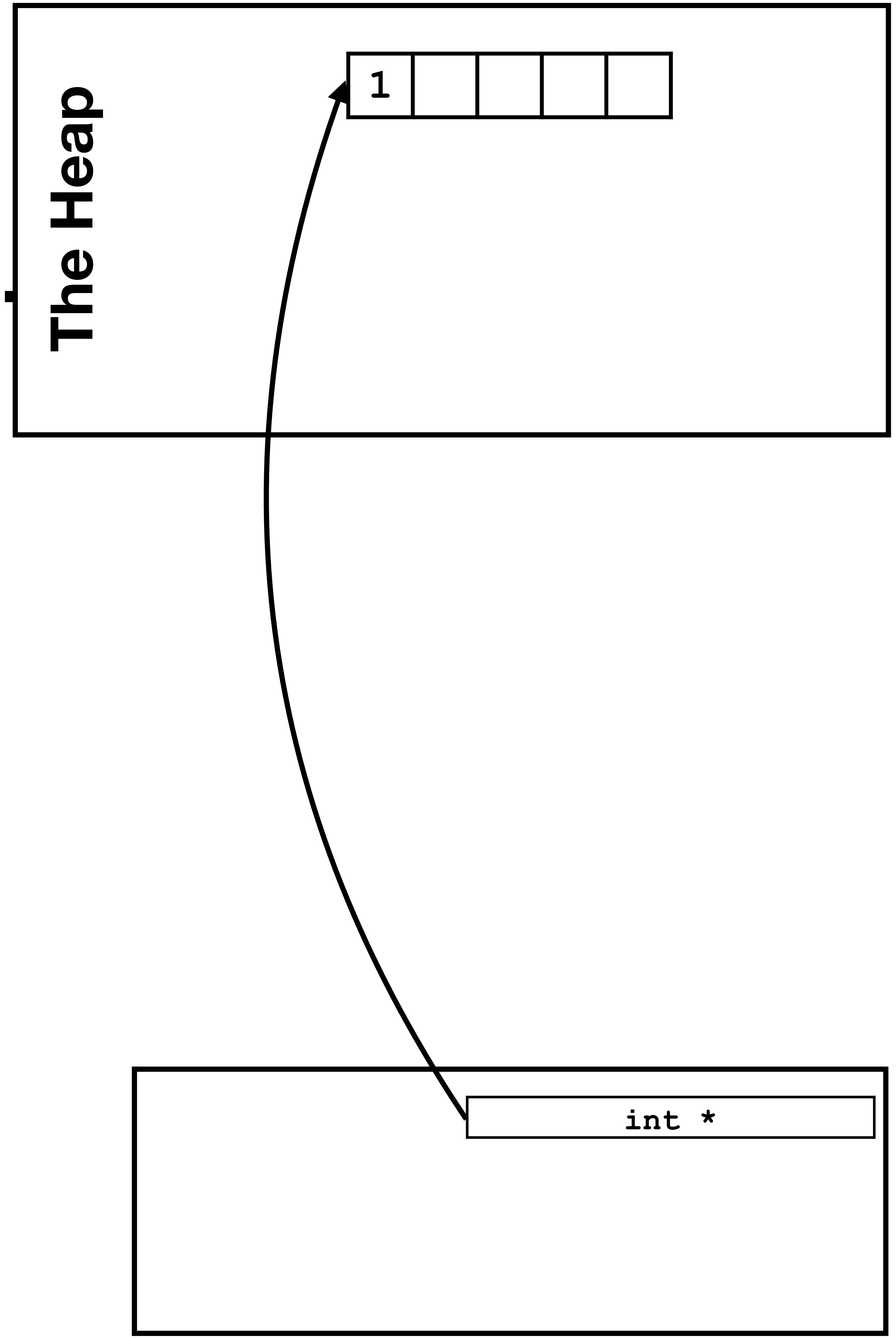
CS143: lecture 11

Byron Zhong, July 2

Array

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

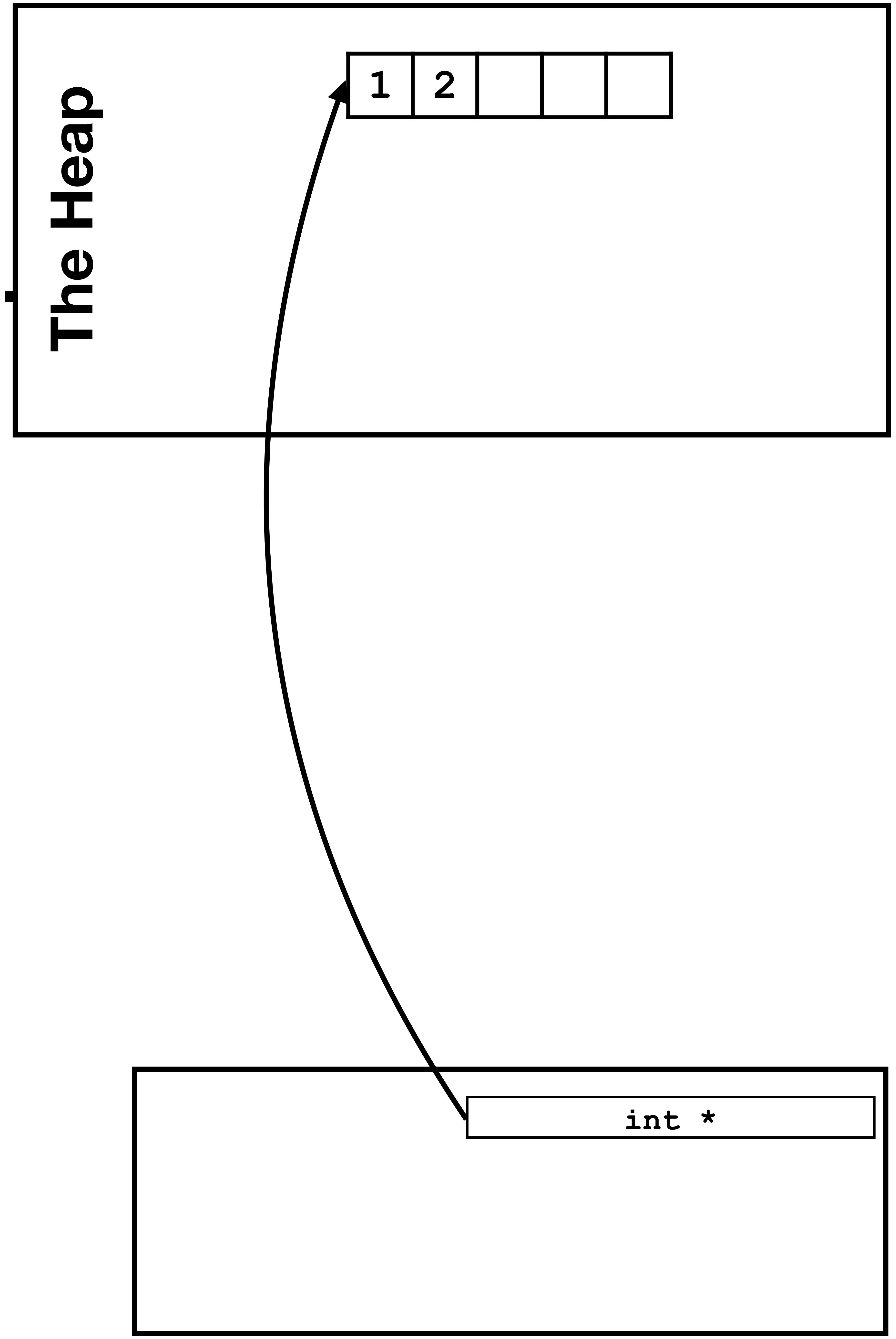
- What if we store a pointer at the end of the array
- ...when the array is filled up, we direct it to somewhere else.



Array

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

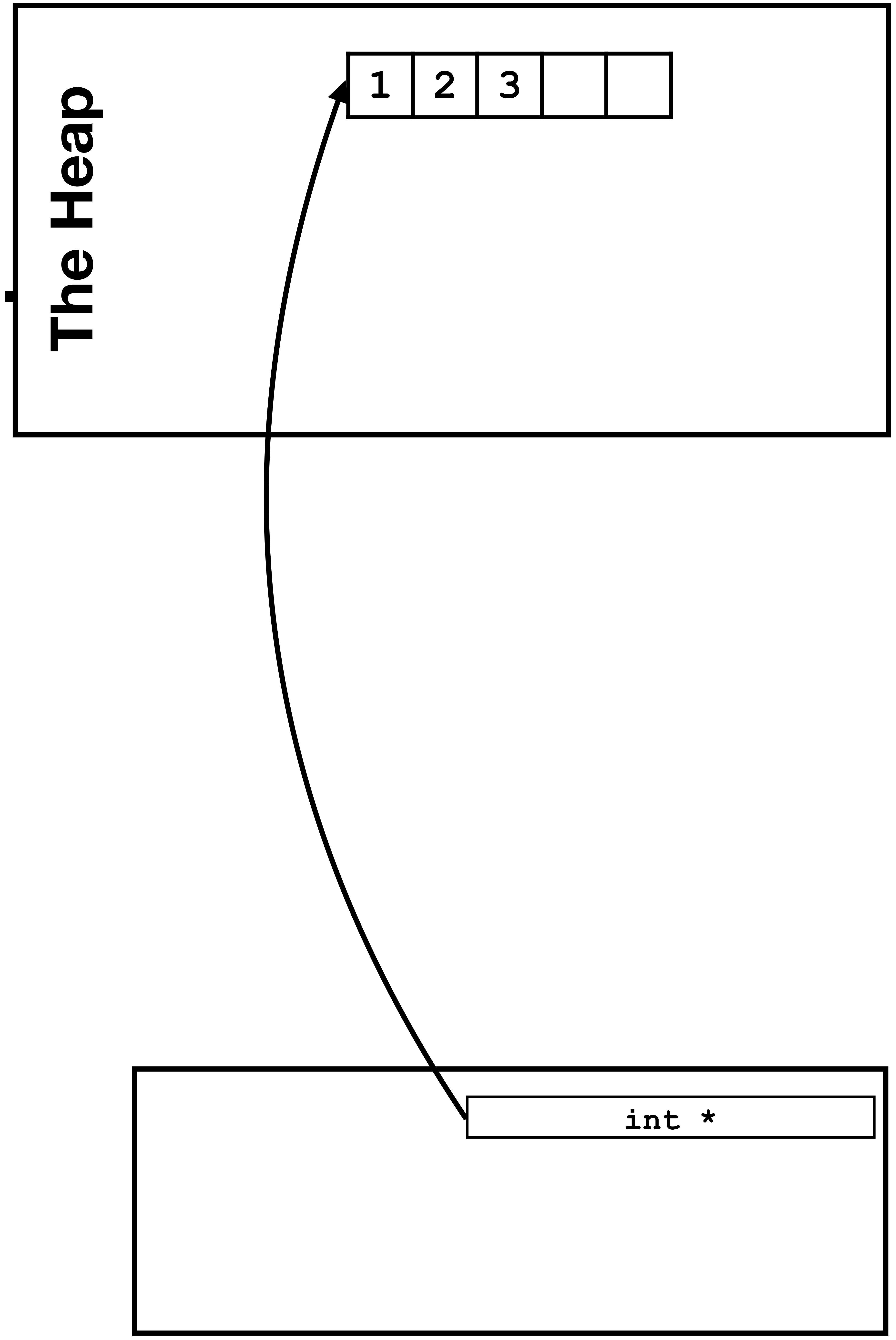
- What if we store a pointer at the end of the array
- ...when the array is filled up, we direct it to somewhere else.



Array

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

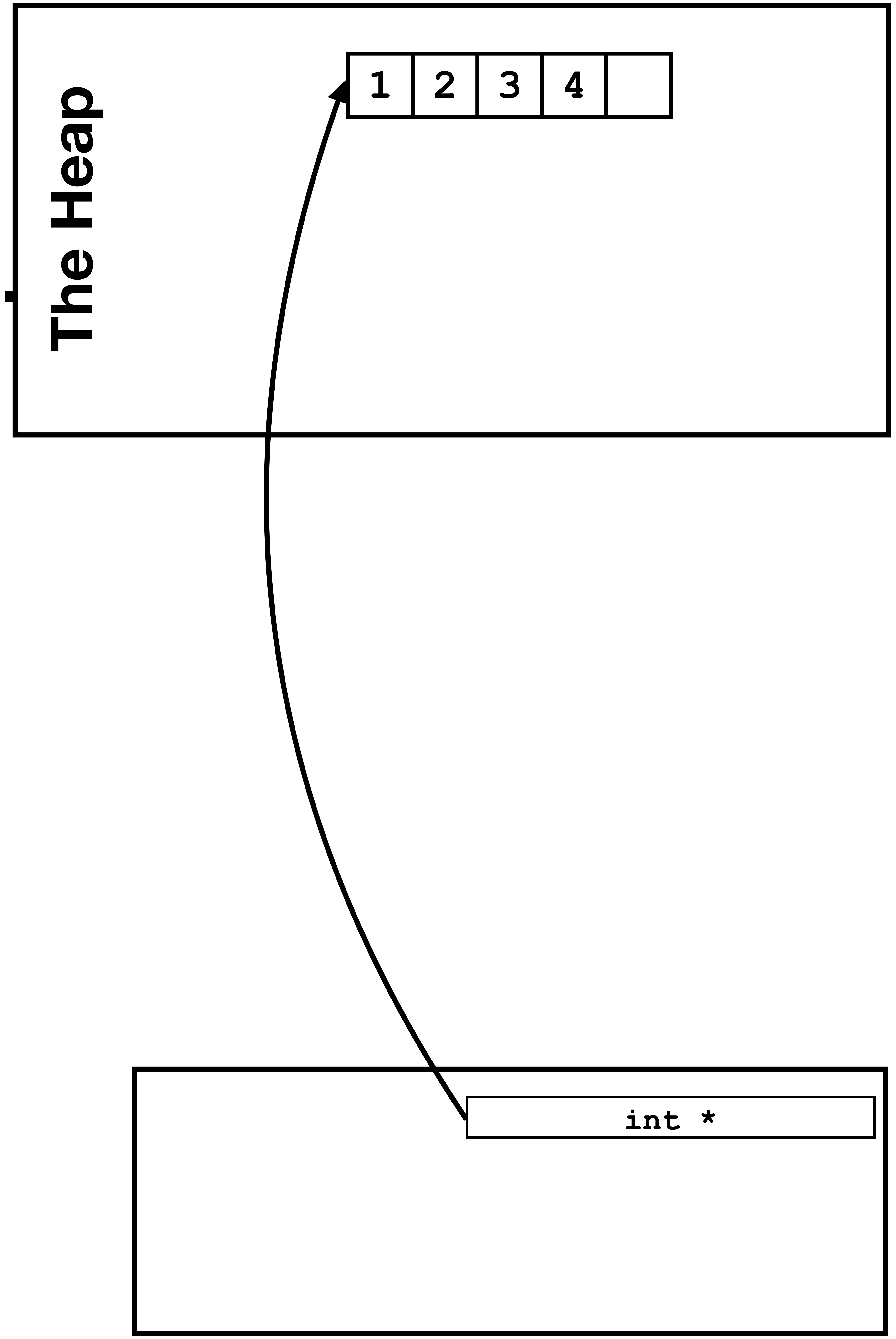
- What if we store a pointer at the end of the array
- ...when the array is filled up, we direct it to somewhere else.



Array

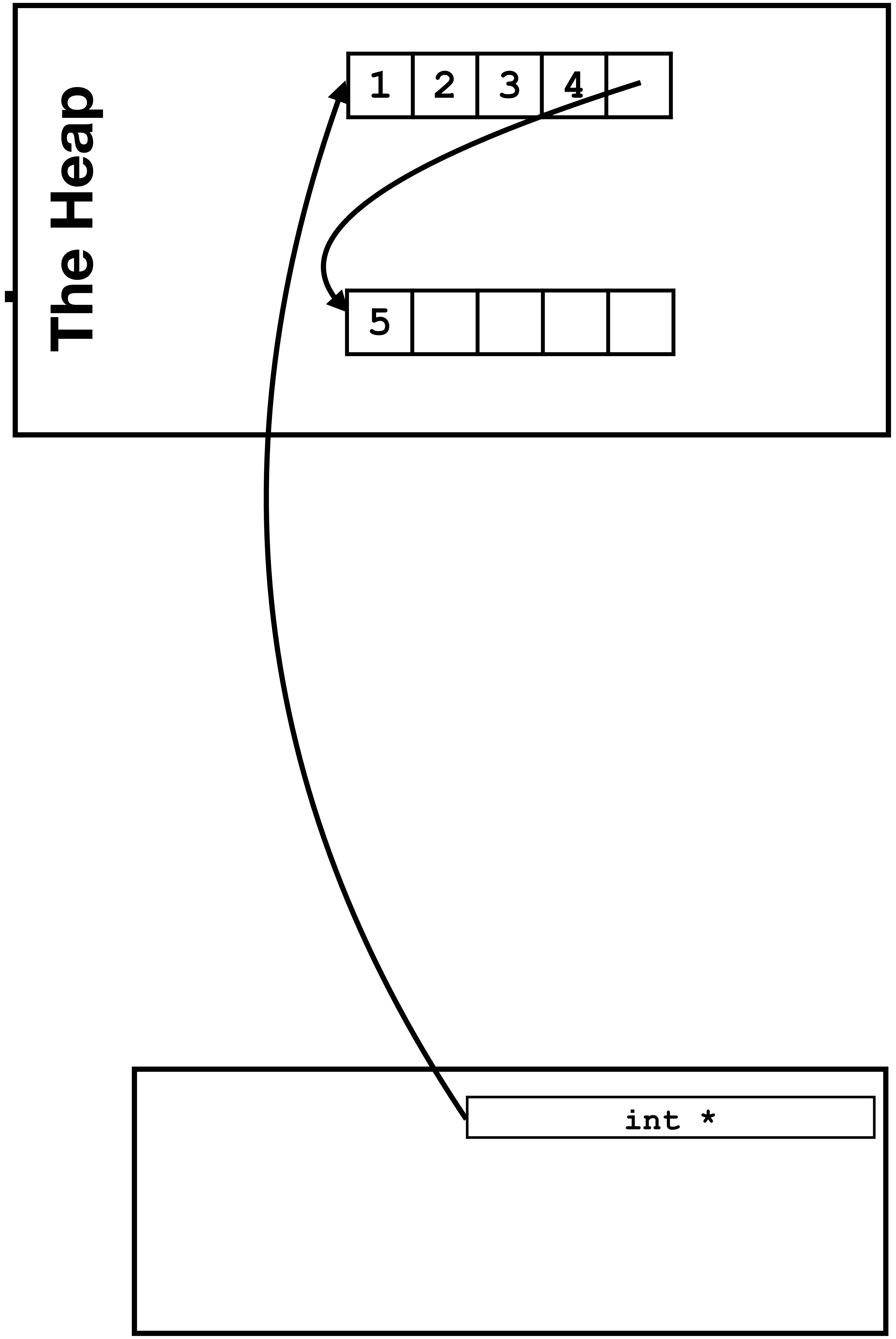
Another way to grow the size of the array...

- 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...

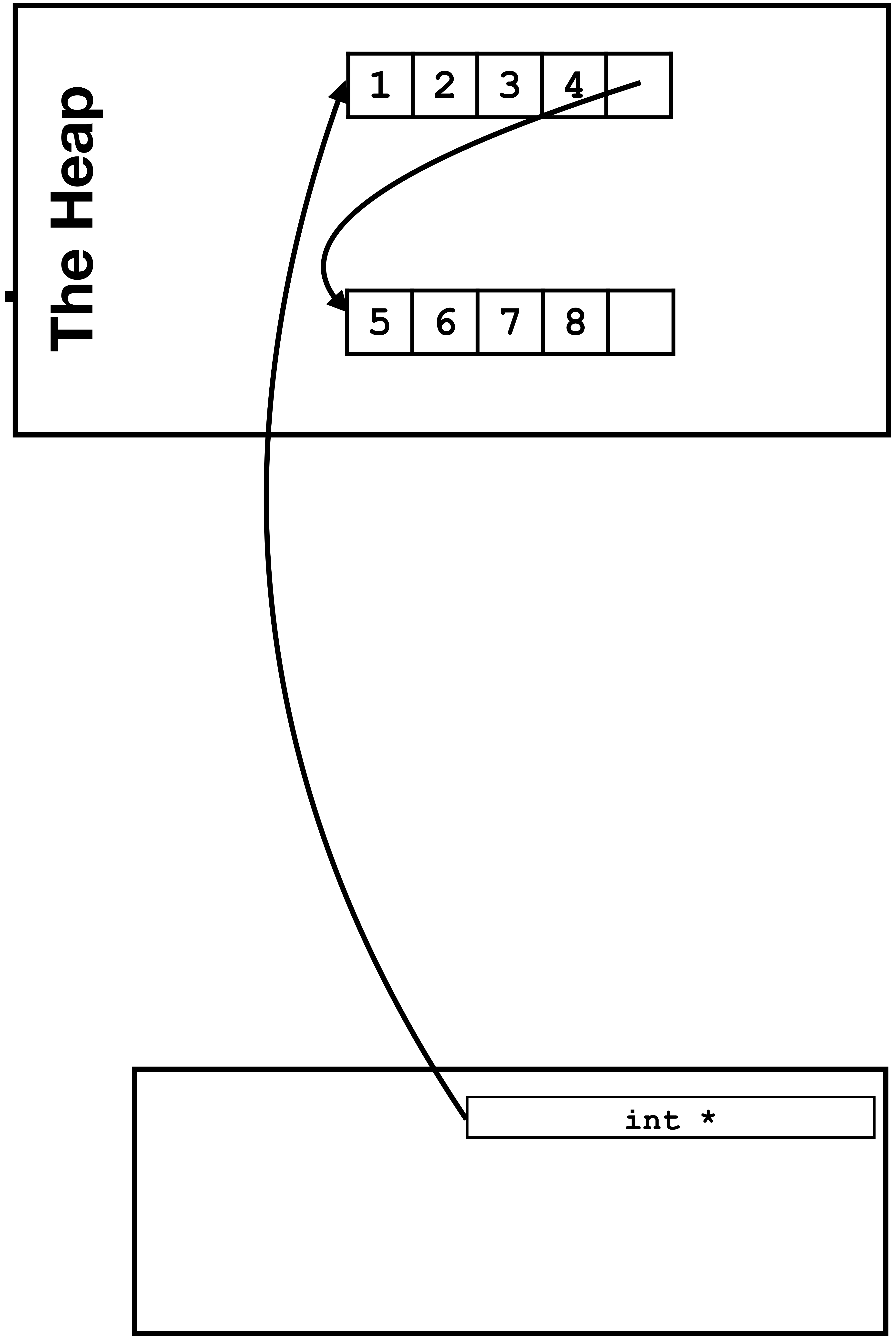


- What if we store a pointer at the end of the array
- ...when the array is filled up, we direct it to somewhere else.

Array

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

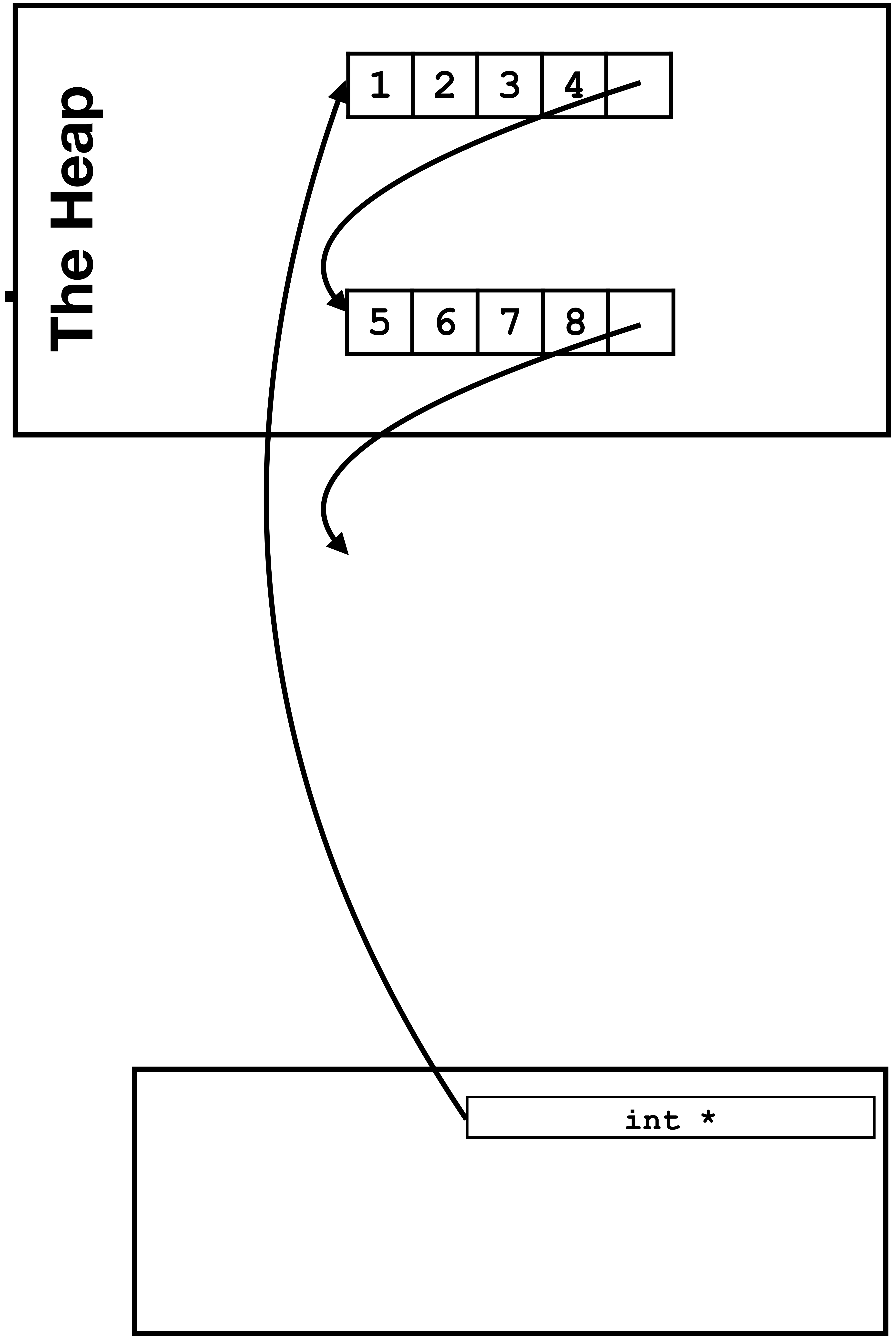
- What if we store a pointer at the end of the array
- ...when the array is filled up, we direct it to somewhere else.



Array

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

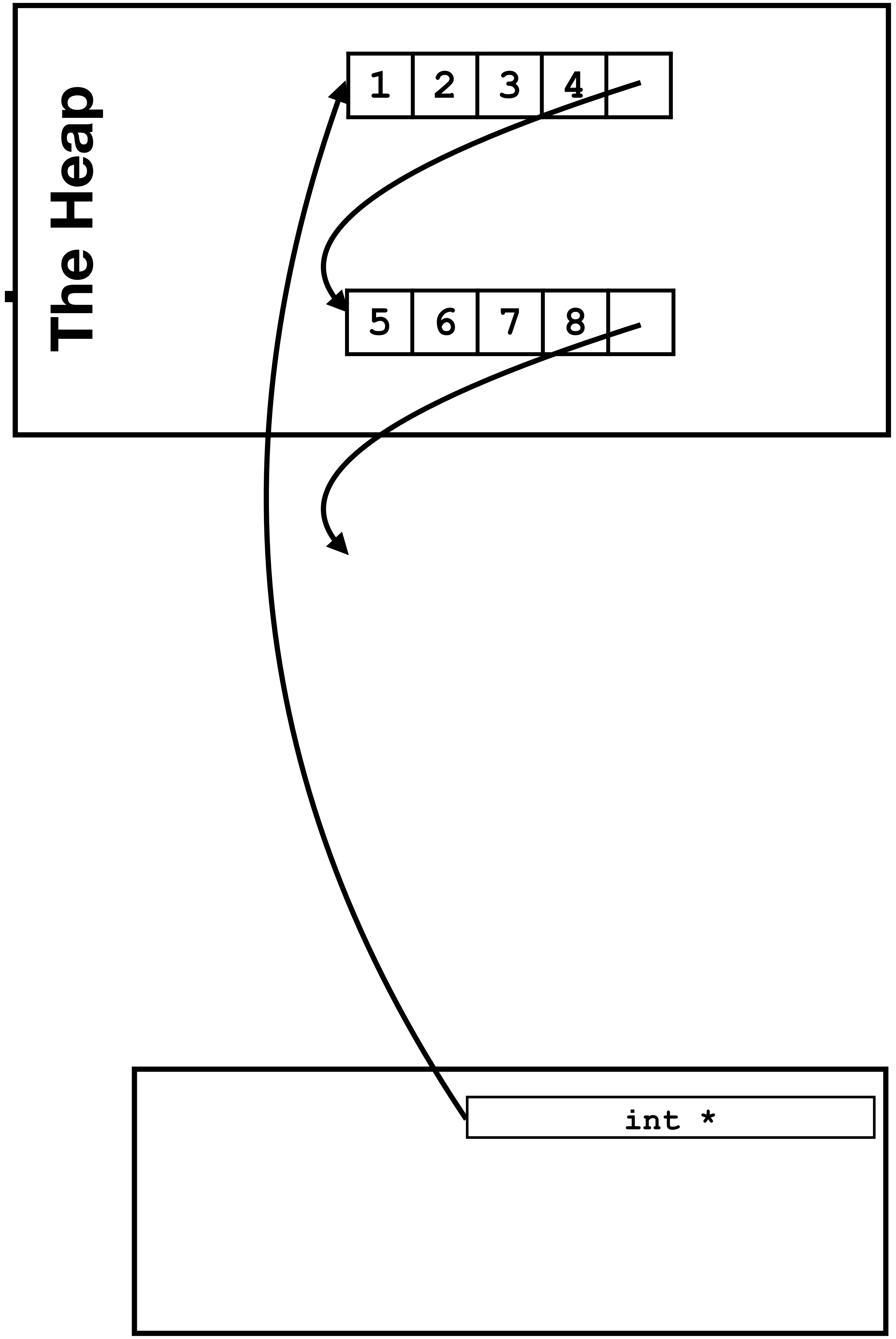
- What if we store a pointer at the end of the array
- ...when the array is filled up, we direct it to somewhere else.



Array

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

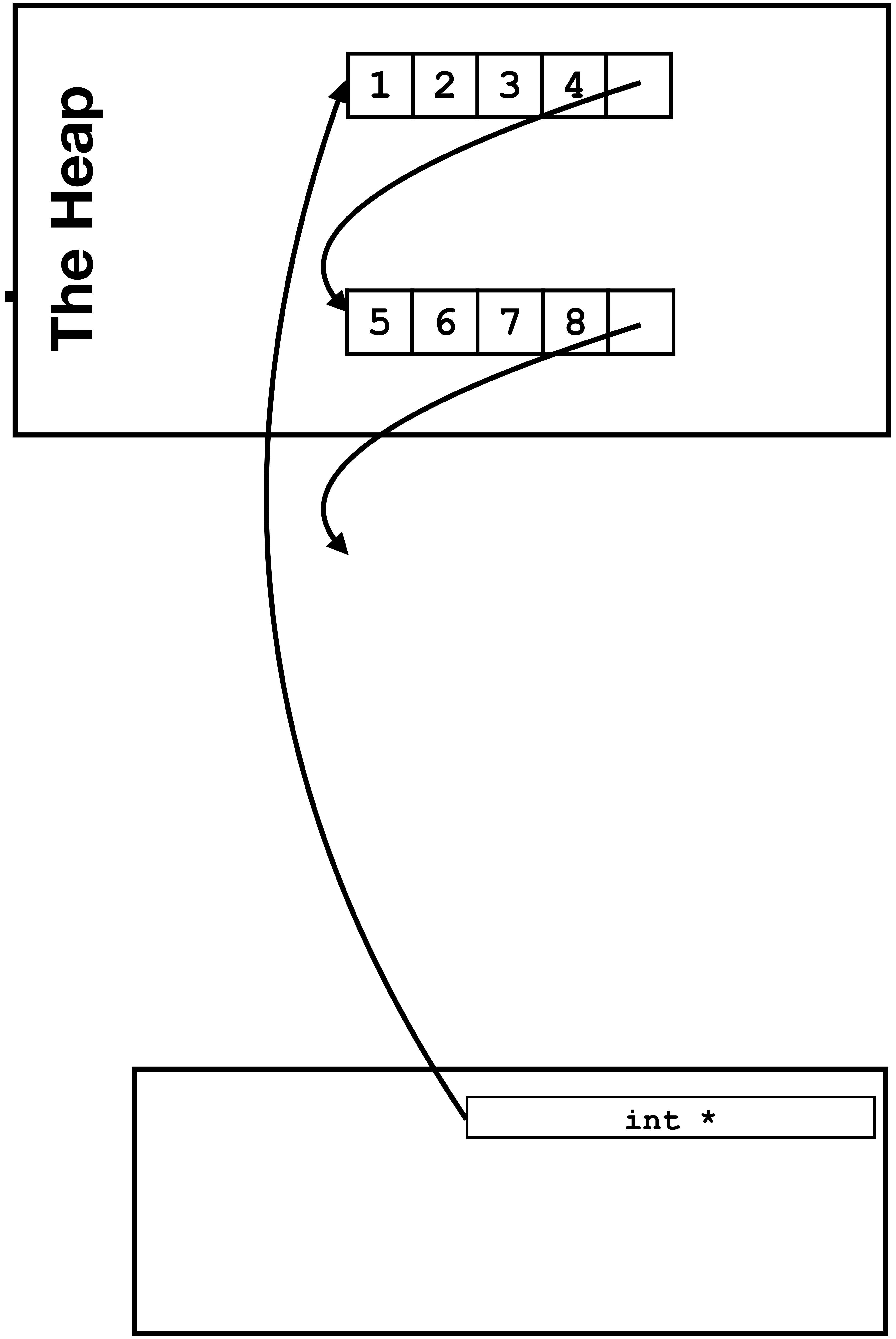
- What if we store a pointer at the end of the array
- ...when the array is filled up, we direct it to somewhere else.
- Good idea?
    - No copying needed when we grow
    - Removing elements at the front is also less costly
    - But how do you index? Say I want 10-th element



Array

Another way to grow the size of the array…

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



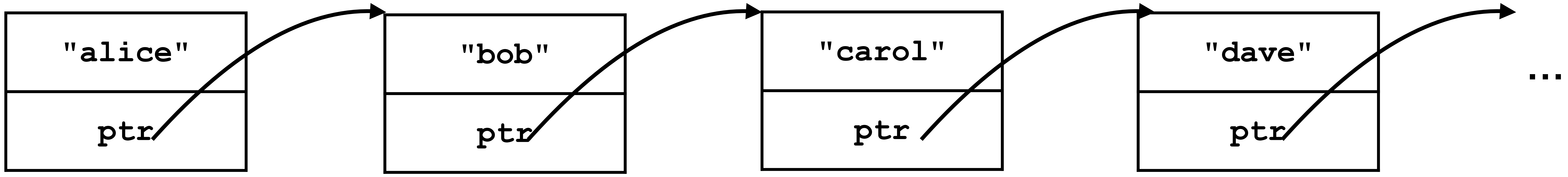
Linked Lists

- 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;
};
```

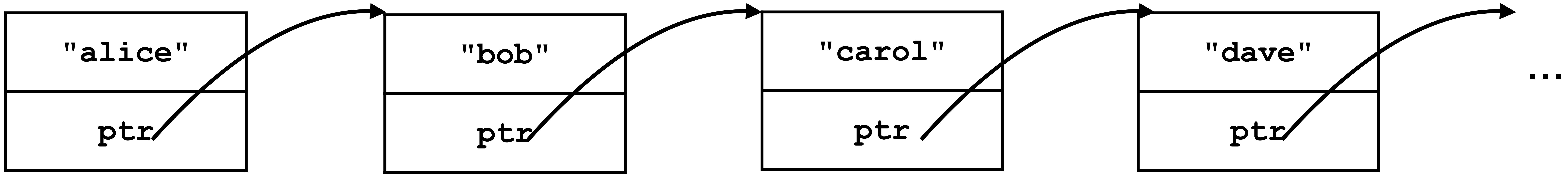
Linked Lists

- 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



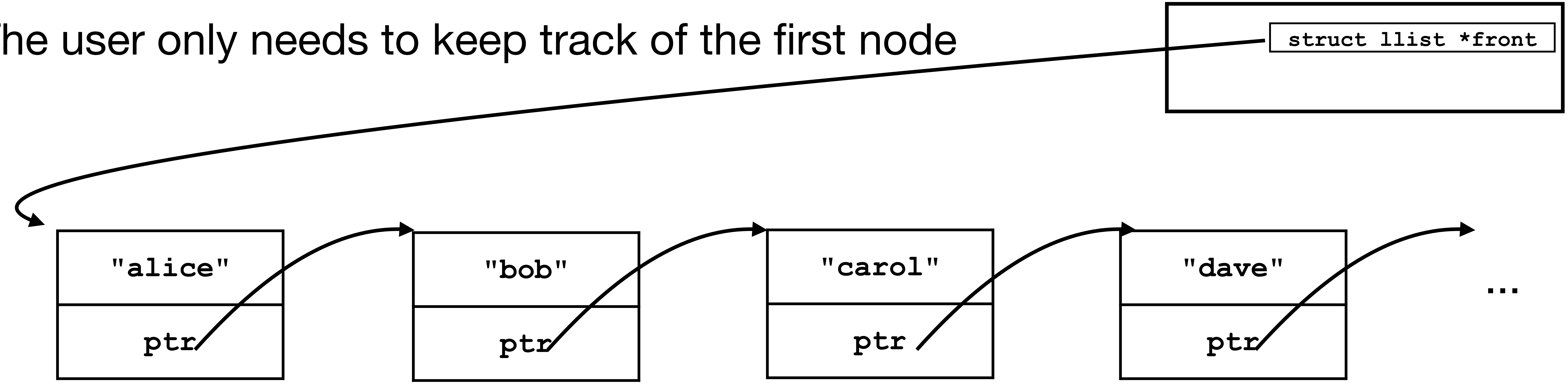
Linked Lists

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



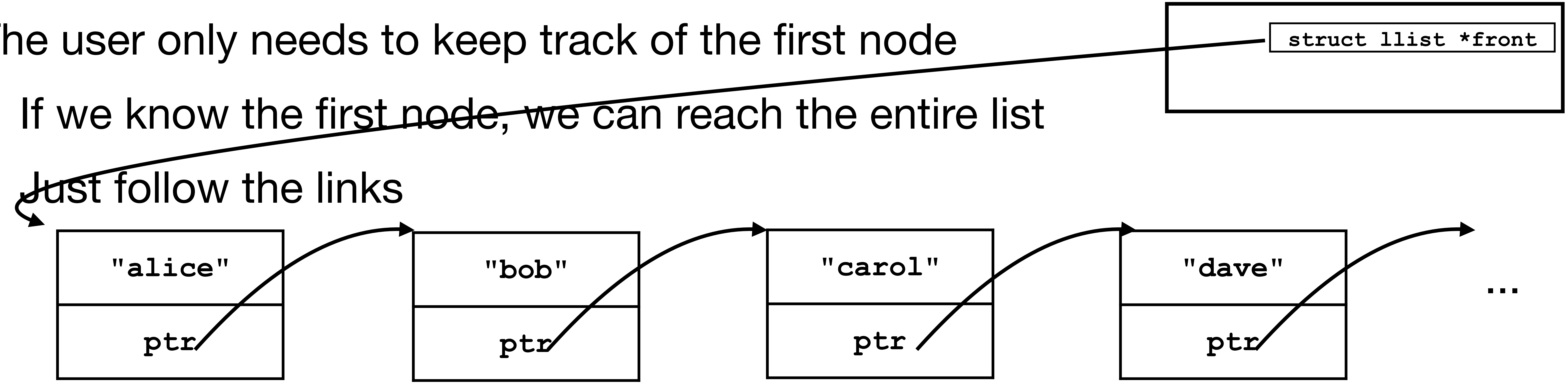
Linked Lists

- 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



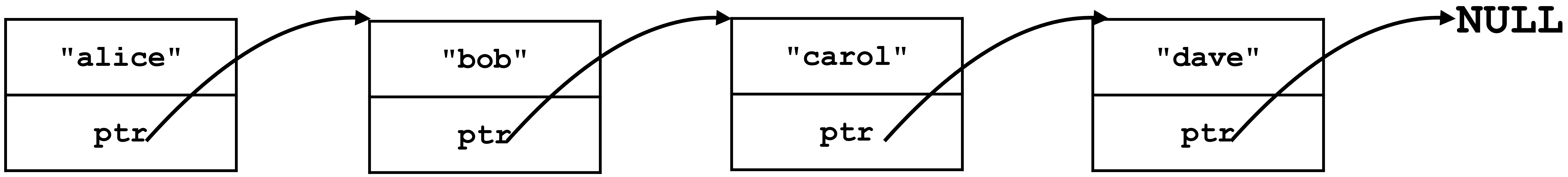
Linked Lists

- 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

- 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`

- 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()`

```c
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

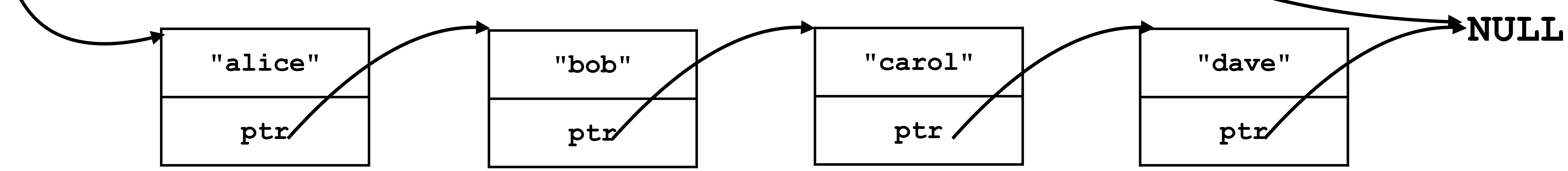
len()

```
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()`

```c
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()

```c
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()`

```c
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`

- 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()`

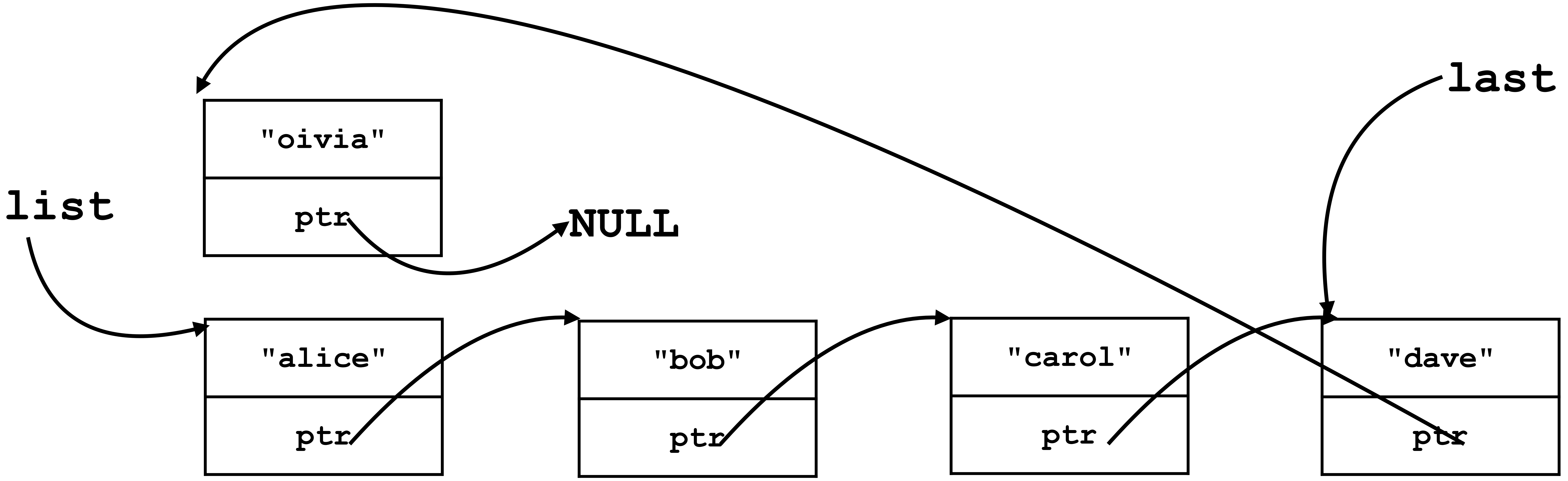
- We want to add "Olivia" to the back of the list

  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`

- 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`

- 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

- 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?
  - Any pointer!

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

- 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.*