Lecture 08
Linked Lists

CS211 – Fundamentals of Computer Programming II

Branden Ghena – Winter 2022

Slides adapted from:
Jesse Tov

Northwestern

Administrivia

- Homework 3 due tonight
  - Remember that slip days exist

  - Beware: office hours are overloaded
    - Prepare for long delays until you can get help, and only high-level help
    - Feel free to ask questions on Campuswire too
      - I’ll be checking it all evening

- No lab due this weekend!
  - Get started on Homework 4 early instead (which will be out tonight)

Homework 3 hint: comparing strings

```
char* a = “abc”
char b[4] = {‘a’, ‘b’, ‘c’, ‘\0’}
if (a == b) {
  print(“They match!\n”);
} else {
  print(“They do not match\n”);
}
```

This code prints: “They do not match\n”. Why?

What does `a == b` compare?

Two pointers!

Strings must be compared with strcmp()

- https://www.cplusplus.com/reference/cstring/strcmp/

- int strcmp(const char* str1, const char* str2)
  - Compares two strings character-by-character until reaching a '\0'
  - Returns an integer value of the following:
    - <0 – str1 comes before str2 alphabetically
    - 0 – str1 is equal to str2
    - >0 – str1 comes after str2 alphabetically

SEGV is a null pointer dereference

```
Check failed (test/test_vc.c:36):
  assertion: cp
test/test_vc.c:37:9: runtime error: load of null pointer of type 'size_t'
AddressSanitizer:DEADLYSIGNAL
=================================================================
==1167490==ERROR: AddressSanitizer: SEGV on unknown address 0x000000000000 (pc 0x000000400cbe bp 0x000000000000 sp 0x7ff
e84837c00 T0)
==1167490==The signal is caused by a READ memory access.
==1167490==Hint: address points to the zero page.
SCARINESS: 10 (null-deref)
    #0 0x400cbd in test_2_candidates test/test_vc.c:37
    #1 0x400cbd in main test/test_vc.c:66
    #2 0x7f544b789492 in __libc_start_main (/lib64/libc.so.6+0x23492)
    #3 0x400a8d in _start (/home/slc8828/cs211/hw03/.bin/test_vc-16+0x400a8d)

AddressSanitizer can not provide additional info.
SUMMARY: AddressSanitizer: SEGV test/test_vc.c:37 in test_2_candidates
==1167490==ABORTING
```

- This AddressSanitizer error is due to dereferencing a NULL pointer
  - Often in Homework 3, it’s because you tried to read a NULL candidate name
  - Possibly with `strcmp()`

Today’s Goals

- Practice dynamic memory allocation with arrays
    - How do we make an array the dynamically changes size?

- Introduce and explore concept of linked lists
    - What are they and what are their advantages?
    - How do we write code that uses them?

- Discuss concept of pointers to pointers

Getting the code for today

`cd ~/cs211/lec/` (or wherever you put stuff)

```
tar -xkvf ~cs211/lec/08_linked_lists.tgz
cd 08_linked_lists/
```

Outline

- **Dynamic Arrays**
- Linked Lists
- Linked List Details
- Pointers to Pointers

Dealing with dynamic input

- What if you want to read in data, but you don't know how much data there might be?

- Arrays in C are a fixed size
- But you can `malloc()` as many times as needed
  - Request some memory
  - Use until you run out
  - Request more memory and copy existing values over

  - `realloc()` makes this simple

Example of dynamic memory: read_line()

```
char* read_line(void)
```

- Reads an entire line at a time from stdin
  - Can’t know in advance how many bytes there will be to read
  - Keeps reading in bytes until ‘\n’ character or end-of-file
  - Needs to request more memory until it holds the entire line
- Note: part of the 211 library, not standard C

Live coding: implement read_line()

readline.c

```
char* read_line(void)
```

- Requirements
  - Read from stdin until ‘\n’ or end-of-file (EOF)
  - Allocate an array to hold the read characters
    - Make sure to end it with a ‘\0’
  - Returns
    - NULL pointer if EOF was reached immediately
    - Pointer to string otherwise (not including the newline character)

Realloc versus malloc

- We could just `malloc()` and copy ourselves, what does `realloc()` add?

- `realloc()` can be far more efficient
  - Doesn't have to copy data at all if there is room in the heap to expand

- Also simpler for programmers
  - Can't forget to free the old memory if `realloc()` does it for you

Default string size will change efficiency

- Memory efficiency
  - Pointer returned could have way more memory than characters
  - User might hold on to memory for a while before freeing
  - The less wasted memory, the less memory the program needs

- Runtime speed
  - `malloc()` and `realloc()` are slow
  - The fewer times we call them, the faster the program will run

- Need to pick a sweet spot to balance the two of these
  - Real program: starts at 80 characters, doubles size when reallocating

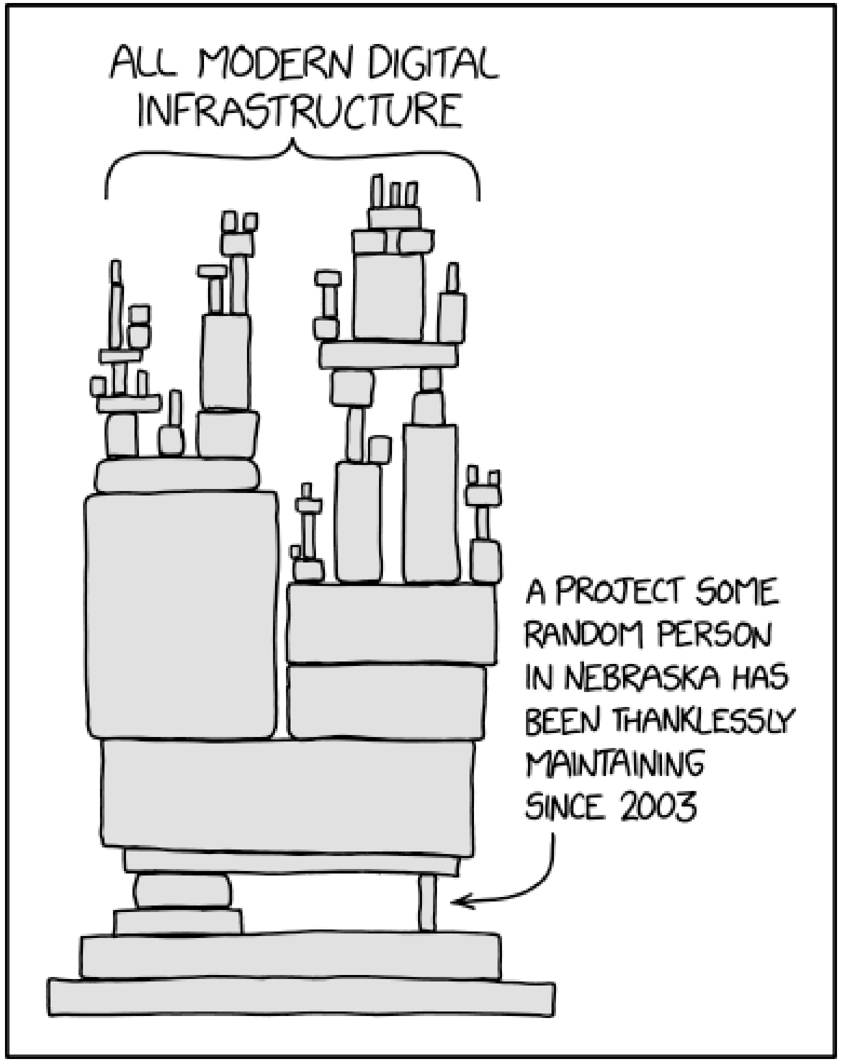
Does efficiency really matter though?

- If you're writing a CS211 homework: **no**

- If you're writing a Javascript interpreter for Firefox,
  - Which has millions of users
  - times hundreds of websites per day for each user
  - times hundreds of lines of code per website
  - and each line of code is read with `read_line()`

  - **YES**

Break + relevant xkcd



https://xkcd.com/2347/

Outline

- Dynamic Arrays
- **Linked Lists**
- Linked List Details
- Pointers to Pointers

Problems with arrays

- They make a lot of sense when you have fixed data
- But they’re not very flexible for dynamic data

- Not smooth or simple to grow/shrink arrays
  - Lots of thought for how to dynamically change memory

- Writing to an array only overwrites existing slots, doesn’t append
  - How would we add data to front of an array?

Live coding example

array_add_front.c

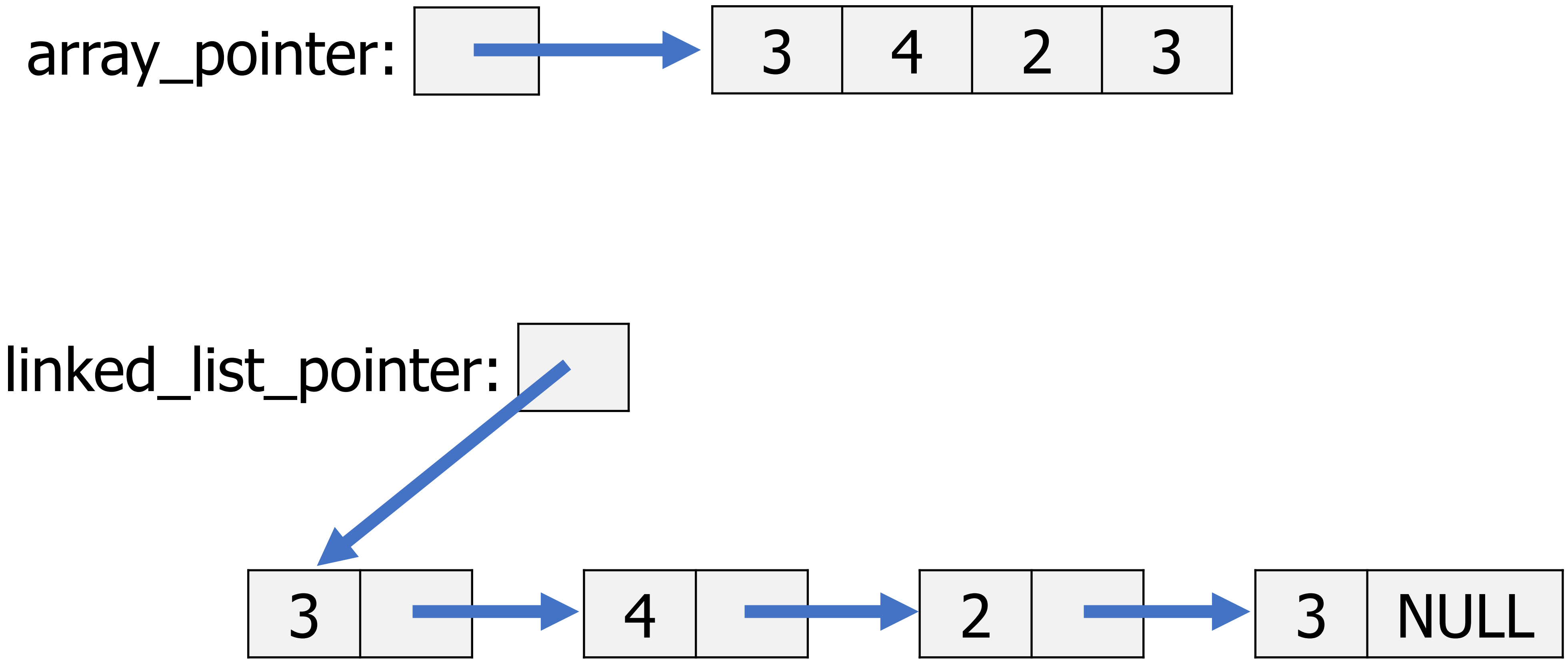
- Let’s write a function that adds to the front of an existing array

- `print_array()`
  - Prints contents of array so we can see what the program is doing

- `add_to_front()`
  - Appends a value to the front of an existing array

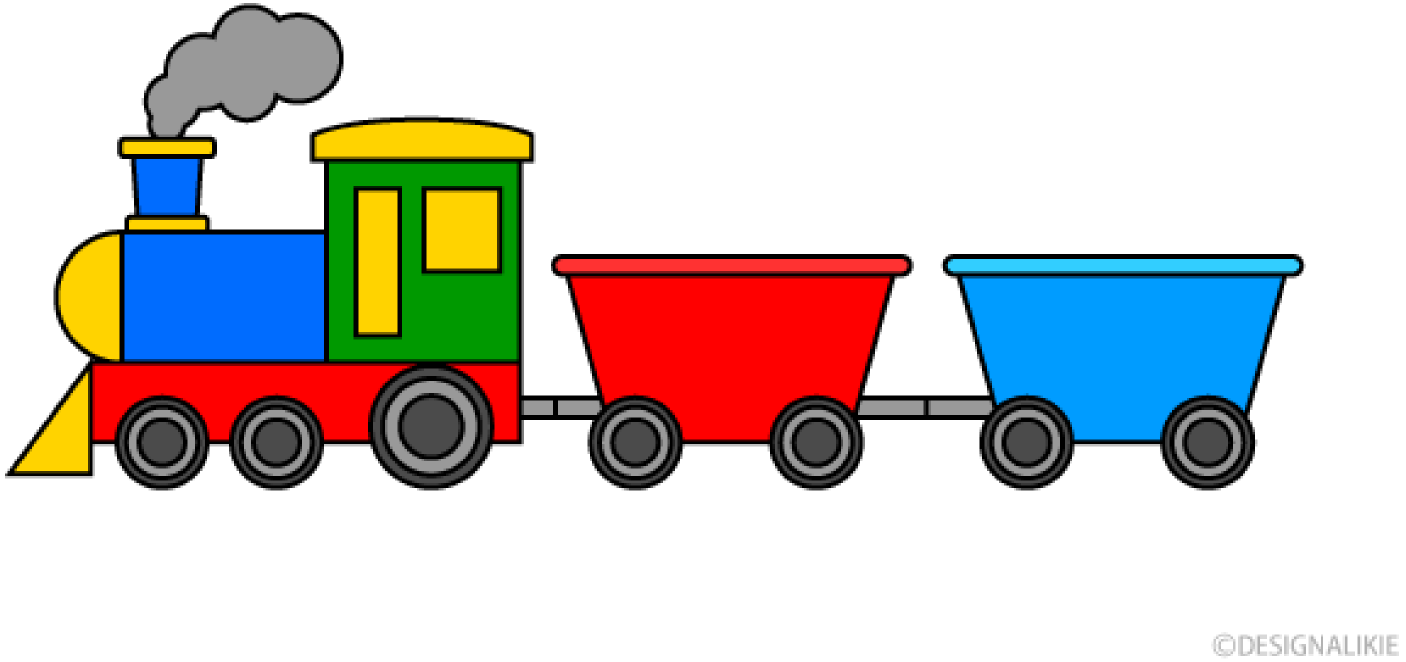
  - It’s annoying to try to append to an array
    - It’s also very inefficient. Needs to move *every* element

An alternative: linked allocations



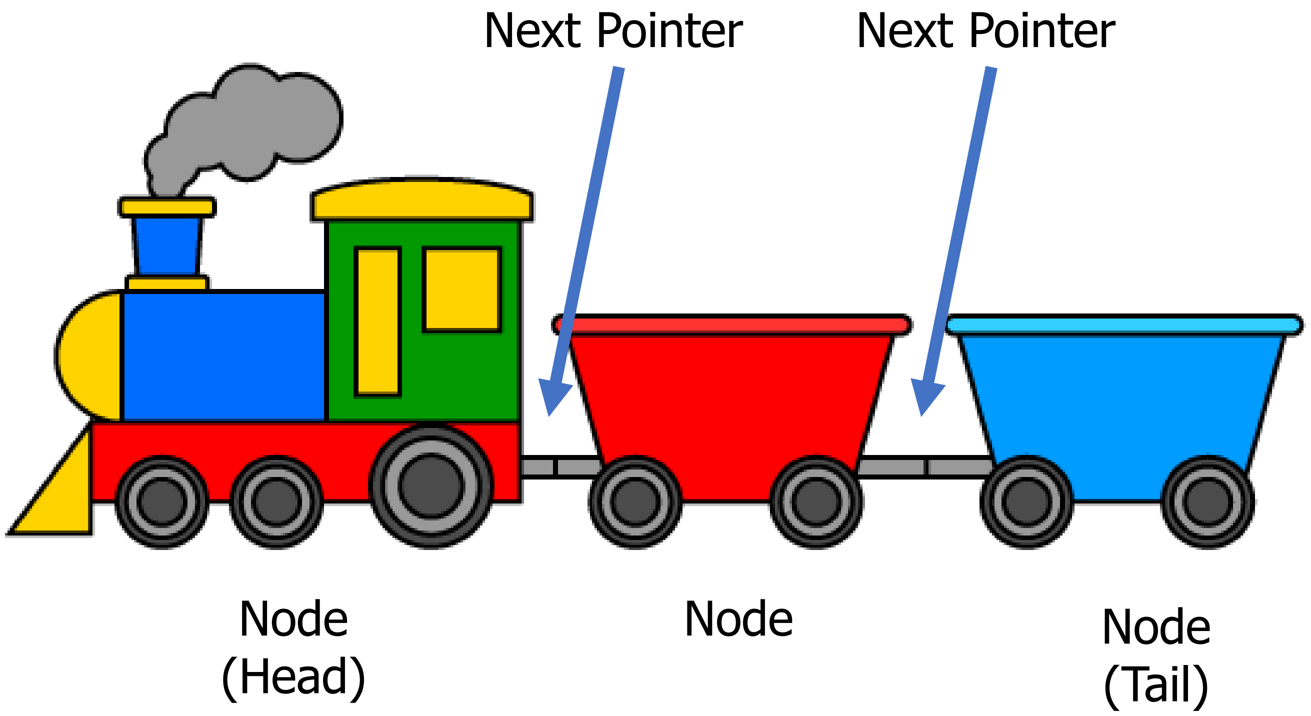
Linked list analogy as a train

- Think of a linked list as a train



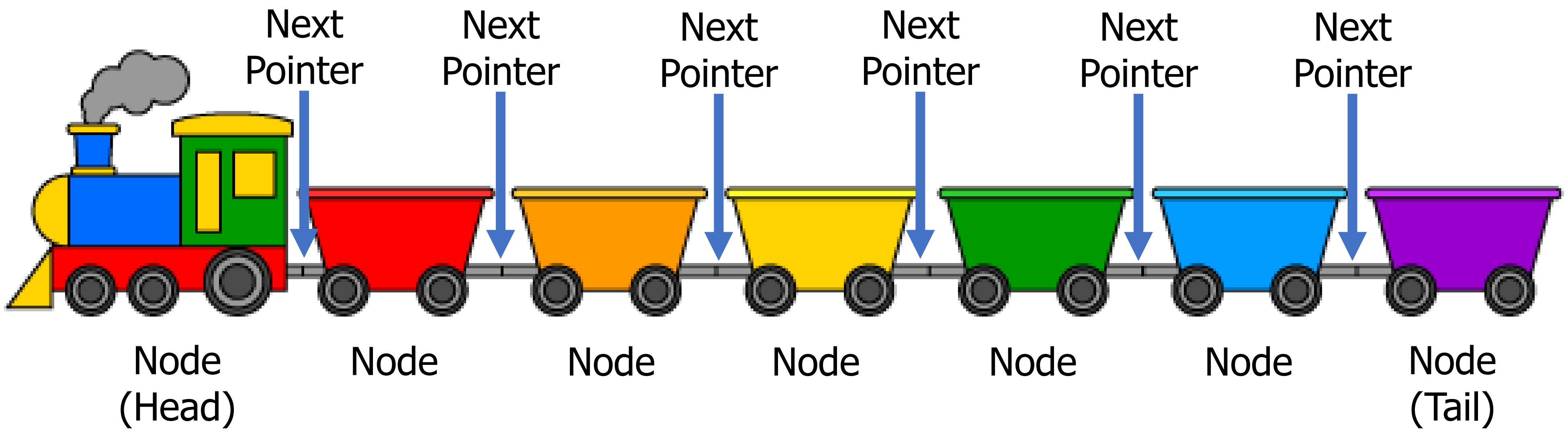
Linked list analogy

- Think of a linked list as a train
  - Multiple Nodes are linked together into a Linked List



Linked list analogy

- Think of a linked list as a train
  - Multiple Nodes are linked together into a Linked List
  - Additional Nodes can be added anywhere in the Linked List
    - Just disconnect, add the Node in between, and reconnect



C code for a linked list structure

- Array version:

```
int myarray[];
```

- Linked List version:

```
struct node {
  int value;
  struct node* next;
};
typedef struct node node_t;

node_t* head;
```

Rules for linked lists

- The variable holding the "list" is actually a pointer to the first node of the list
  - Just like an array is a pointer to the first element in the array
  - First node: the "Head" of the list

- Each node must have a pointer to the next node in the list

- The last node in the list has a NULL pointer
  - Last node: the "Tail" of the list

Live coding example

linked_list.c

- Working with a linked list
  - Create an empty list
  - Add elements to the list
  - Determine length
  - Print entire list

Break + Question: Which is better, and why?

```
void free_list(list_t list)
{
  while (list != NULL) {
    free(list);
    list = list->next;
  }
}
```

```
void free_list(list_t list)
{
  if (list != NULL) {
    free_list(list->next);
    free(list);
  }
}
```

Break + Question: Which is better, and why?

```
void free_list(list_t list)
{
  while (list != NULL) {
    free(list);
    list = list->next;
  }
}
```

```
void free_list(list_t list)
{
  if (list != NULL) {
    free_list(list->next);
    free(list);
  }
}
```

Error: use after free!!

Best would be a *working* while loop

Outline

- Dynamic Arrays
- Linked Lists
- **Linked List Details**
- Pointers to Pointers

The memory for each list node must be managed

linked_list.c

- Lists are composed of many small memory allocations rather than one large memory allocation (like arrays)

- So every individual node needs to be separately freed in order to destroy the entire list

- Let’s write `list_destroy()`

Lists have no random access

linked_list.c

- You can ask an array for any item, and you get it immediately
  - `array[6]`

- All access for linked lists is sequential
  - You must start at the head and "walk" the list until you get to the item
  - `list->next->next->next->next->next->next->value`

- Let's write `get_at_index()`

Items can be added at any point in the list

linked_list.c

- We can add/remove the middle item of the list
  - Just make sure you get the next pointer right

- Arrays can't support that kind of thing
  - You would have to copy over all the later elements in the array

- Let's write `list_append_front()` and `list_remove_front()` functions

Break + Open Question

- Which uses more memory, an array or a linked list?
  - Assume each contains the same values

- How much more?

Linked lists take more memory than arrays

- Each node must include data and a “next” pointer

- This increases overall memory use
  - As a cost for the ease of use that linked lists provide

- Compare an array of `int` versus linked list of `int`
  - Linked list will be 3x the amount of memory

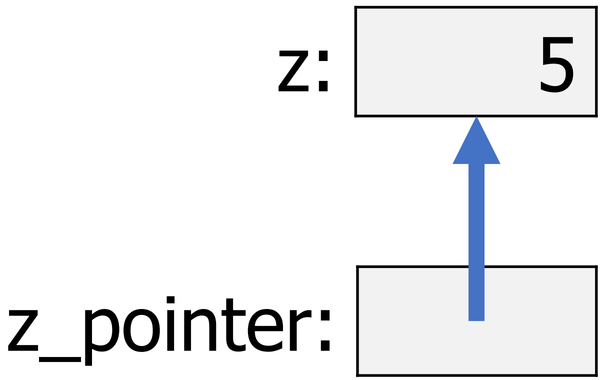
  - The large in size your data is, the less the overhead will be

Outline

- Dynamic Arrays
- Linked Lists
- Linked List Details
- **Pointers to Pointers**

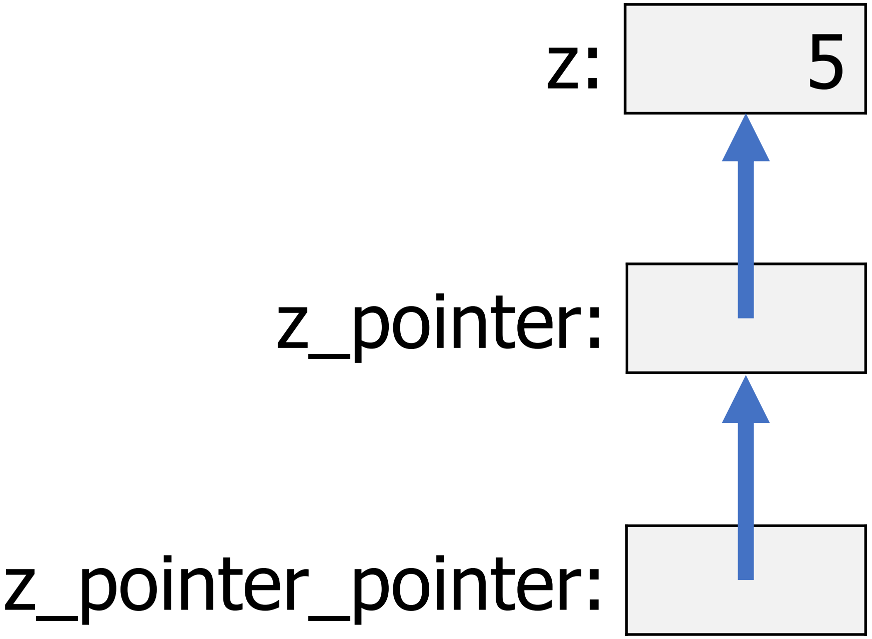
Pointers are another type of value

- Values could be a number, like 5 or 6.27
- Or they could be a "pointer" to an **object**
  - Points at the object, not the variable or value
  - It points at the "chunk of memory"
    - Technically, in C it holds the address of that memory



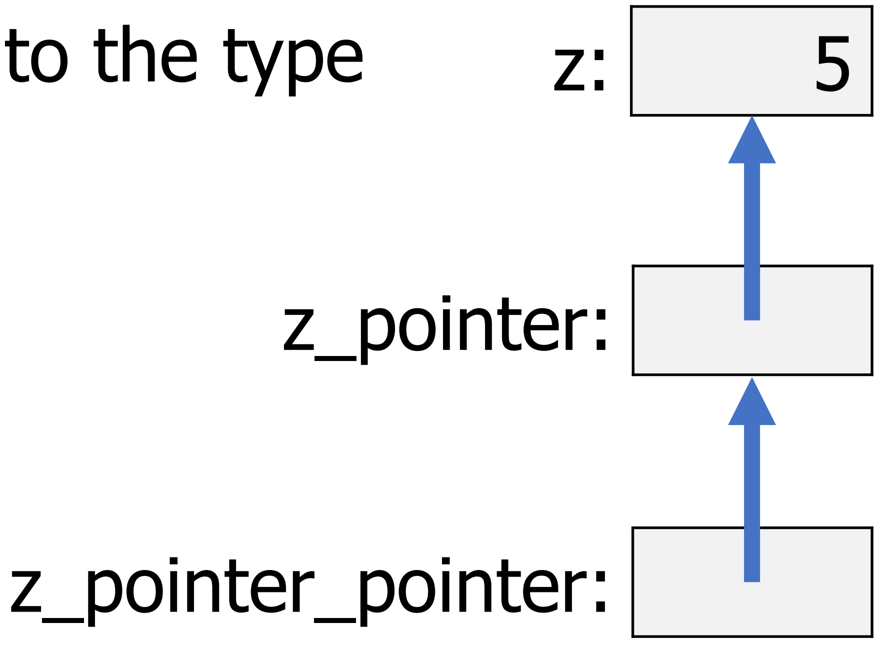
We can make a pointer to another pointer

- Pointers are values stored in an object
  - That object has a memory address
  - We could make a pointer to a pointer



Double pointers in C

- To make a pointer to something, add a * to the type




```
int z = 5;

int* z_pointer = &z;

int** z_pointer_pointer = &z_pointer;
```

When is this useful?

linked_list.c

- Various functions in the linked list code need to return the new head of the linked list
  - Instead, they could update the linked list variable

```
struct node* list_append_front(struct node* list, int value);
```

could become

```
void list_append_front(struct node** list, int value);
```

Also occurs in arguments to main

- argv is an array of strings
  - Strings are `char*`
  - So argv is `char**`

- `char* argv[]` is equivalent to `char** argv`

Outline

- Dynamic Arrays
- Linked Lists
- Linked List Details
- Pointers to Pointers