CS 521: Systems Programming

Arrays

Lecture 7

Arrays

- In C, arrays let us store a collection of values of the same type
  - `int list[10];`
  - `double dlist[15];`
- Internally, they are represented as a chunk of memory large enough to fit all the required elements
- Note that the arrays must be dimensioned when they're declared
  - In older versions of C the dimension had to be a constant

Accessing Array Elements

- Setting/retrieving the values of an array is the same as it is in Java:
  - `list[2] = 7;`
  - `list[1] = list[2] + 3;`
- However, one interesting note about C is there is **no boundary checking**, so:
  - `list[500] = 7;`
  - …may work just fine.
  - 

Experiment: When will it Break?

- We can try modifying out-of-bounds array elements
  - see: `array_break.c`
- We can even do it in a loop to test the limits
  - Different operating systems / architectures may react differently
  - Let's try it now. Open your editor, create an array, and write a loop that iterates beyond its boundaries.
    - When does it segfault? How big was your initial array?
- At this point, you might be wondering:
  - What is wrong with C?!
  - What is the meaning of life?

Out-of-bounds Access

- So we can do things like this in C:
  - `int list[5];`
  - `list[10] = 7;`
- Your program may work fine… or crash!
- It's never a good idea to do this
- So why does C let us do it anyway?

Safety vs. Performance

- C favors performance over safety
  - Compare: C program vs Python equivalent
  - Helpful: time command
- Especially in the glory days of C, adding lots of extra checks meant poor performance
  - Additional instructions for those checks
  - If you don't want/need them, then the language shouldn't force it on you!
- This can lead to dangerous bugs

Initializing an Array [1/2]

- Let's create our list of integers:
  - `int list[10];`
- When we do this, C sets aside a place in memory for the array
  - It doesn't clear the memory **unless we ask it to**
    - Another common cause of subtle bugs
- Creating a list of integers initialized to zero:
  - `int list[10] = { 0 };`

Initializing an Array [2/2]

Thus far we've always specified the array size. There is a shorthand for doing this if you already know the contents of the array:

```
// Will auto-size to 5:
int nums[] = { 1, 82, 9, -3, 26 };
```

Here, the compiler will fill in the size for you.

Memory Access

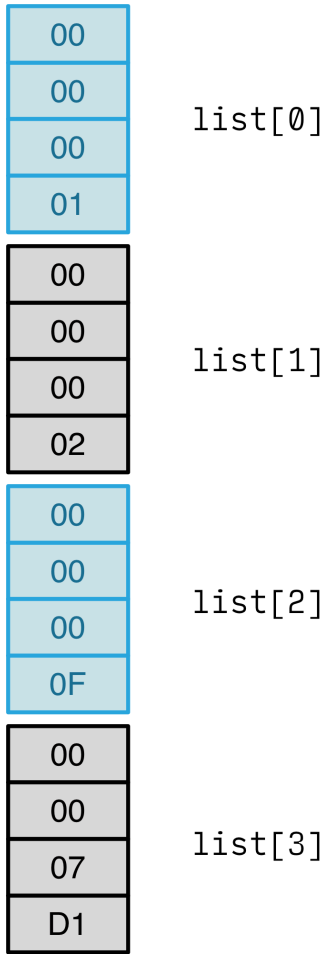
- What happens when we retrieve the value of `list[5]`?
- Find the location of `list` in memory
- Move to the proper offset: `5 * 4 = byte 20`
  - Assuming `sizeof(int) = 4`
- Access the value
- Accessing, say, `list[500]` is just moving to a position in memory and retrieving whatever is there

Visualizing Arrays in Memory

```
/* Note: calculating the array
 * dimensions automatically! */

int list[] = {
    1,
    2,
    15,
    2001
};

sizeof(int) = 4
```



- Note how the visualization represents the integers in **hexadecimal**

The sizeof Operator

- We can use the sizeof operator in C to determine how big things are
  - Somewhat like:
    - len() in python
    - .length in Java, or
    - .size() in Java
- Much more low-level
  - `size_t sz = sizeof(int);`
  - `printf("%zd\n", sz); // Prints 4 (on my machine)`

Array Size [1/2]

- Let's try this out:
  - `int list[10];`
  - `size_t list_sz = sizeof(list);`
- Any guesses on the output?
  - (pause for everyone to yell out guesses)
- On my machine, it's 40:
  - 40 bytes (10 integers at 4 bytes each)
  - This can be different depending on architecture
- In C, `sizeof(char)` is guaranteed to be 1.

Array Size [2/2]

- Knowing the number of bytes in the array can be useful, but not that useful
- Usually we want to know how many elements there are in an array
- To do this, we'll divide by the array **type** (int - 4 bytes):
  - `int list[10];`
  - `size_t list_sz = sizeof(list) / sizeof(list[0]);`
  - `printf("%zd\n", list_sz); /* 10 (for me) */`

Behind the Scenes

- Arrays in C are actually (constant) pointers
  - `int list[5];`
  - `list` is the same as `&list[0];`
- You can't change what they point at, but otherwise they work the same
- So accessing `list[2]` is really just dereferencing a pointer that points two memory addresses from the start of the array
  - …one reason we have 0-based arrays

We can make this more "fun…"

- Since arrays are just constant pointers, we have another way to access them:
  - `list[5]` is the same thing as: `*(list + 5)`
- Workflow:
  1. Locate the start of the array
  2. Move up 5 memory locations (4 bytes each*)
  3. Dereference the pointer to get our value

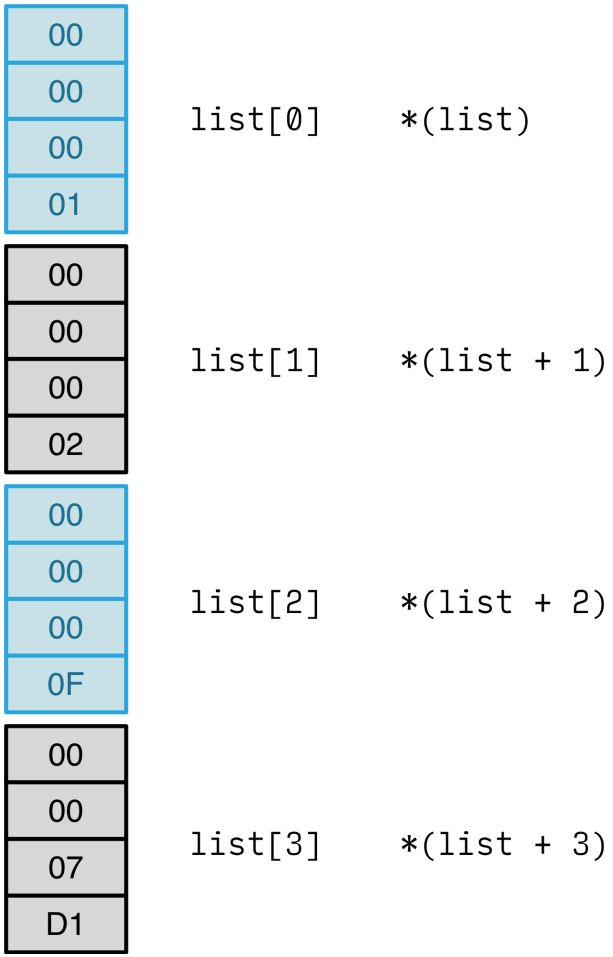
Pointer Arithmetic

- Manipulating pointers in this way is called **pointer arithmetic**
- `arr[i];` is the same thing as: `*(arr + i);`
- `arr[6] = 42;` is the same as `*(arr + 6) = 42;`

Visualizing Arrays with Pointer Arithmetic

```
int list[] = {
    1,
    2,
    15,
    2001
};

sizeof(int) = 4
```



A Note on Pointer Arithmetic

- In general, stick with using regular array syntax
- You may see pointer arithmetic in production code, but it should only be used in situations that make the code **more** understandable
- Haphazardly showing off your knowledge of pointer arithmetic is a recipe for confusing code 

Arrays as Function Arguments

- When we pass an array to a function, its pointer-based underpinnings begin to show
- If we modify an array element inside a function, will the change be reflected in the calling function?
  - ...
  - ...why?
- In fact, when an array is passed to a function it **decays** to a pointer
  - The function just receives a pointer to the first element in the array. That's it!

Array Decay

- When an array decays to a pointer, we lose its dimension information
- Let's imagine someone just gives us a pointer
  - Do we know if it points to a single value?
  - Is it the start of an array?
- Functions are in the same situation: they don't know where this pointer came from or where it's been
  - `sizeof()` doesn't work as expected

Dealing with Decay

- Array dimensions are often very useful information!
  - If we don't know how many elements are in the array, then we could read/write beyond the end of it
- There are two viable strategies to deal with this:
  1. Pass the size of the array into the function as an argument
  2. Put some kind of identifier at the end of the array so we know where it ends as we iterate through
     - (this is the way strings work!)