CS 521: Systems Programming

Pointers and Arrays

Lecture 3

Today's Schedule

- Pointers
- Argument Passing Conventions
- Arrays

Today's Schedule

- **Pointers**
- Argument Passing Conventions
- Arrays

Passing by Value

- In C, function arguments are passed by **value**
  - **NOT** pass by reference
- This means that changes to the argument *inside* the function are not reflected *outside* the function
  - When you call a function, like: `location(2, 4);`
  - Copies will be made of `2` and `4` and passed to `location()`
- Sometimes we actually do want to change the value of a variable when it's passed into a function, though…

Passing by Reference [1/2]

Here’s what a ***swap*** function should produce, but it doesn’t seem possible if `a` and `b` are just copies:

```
int a = 3;
int b = 8;
printf("%d, %d\n", a, b);
swap(a, b);
printf("%d, %d\n", a, b);
```

Output:

```
3, 8
8, 3
```

Passing by Reference [2/2]

- If you want to make outside changes to a variable passed to a function, then you must use **pointers**
- Pointers are a special type of integer that hold a memory address
    - They are still passed by value; the value is the memory address
    - However, we can use the memory address to access a variable defined *outside* a function

Pointer Syntax

- `int *x;` – defines a *pointer*. Note that this doesn't create an integer, it creates a **pointer to an integer**.
  - To make life a little easier, focus on the fact that it's a pointer. Don't worry about its data type for now.
- `&` – 'address of' operator. `&a` returns a pointer to `a`.
  - When a function takes a pointer as an argument, you need to give it an address
- After passing the value of the pointer (memory address), we can **dereference** it (`*` operator) to retrieve/change the data it points to:
  - `*x = 45;`

Demo: Writing swap()

Today’s Schedule

- Pointers
- **Argument Passing Conventions**
- Arrays

Defining a Function

Functions are defined in C like this:

```
<return type> <function name>(<argument list>)
{
    ...
}
```

- If the function does not return a value, the return type should be void
- If there are no arguments, then the argument list is void (not required)
- Let’s dig a bit deeper into this…

Argument Conventions [1/2]

- Coming from the Java or Python world, we're used to passing inputs to our functions
- The result (output) of the function is usually given to us in the return value
  - In Python you can even return a tuple. Nice!
- This is **not** the case with C.
  - In many cases, both the function inputs **and** outputs are passed in as arguments
  - The return value is used for error handling

Argument Conventions [2/2]

Here's an example:

```c
/* Here's a function that increments an integer. */
void add_one(int *i)
{
    *i = *i + 1;
}

int a = 6;
add_one(&a); /* a is now 7 */
```

“In/Out” Arguments

- In C, some of the function arguments serve as **outputs**
- Or in the example we just saw, the function argument is **both** an input and an output!
- Some API designers even label these arguments as “in” or “out” args (example from the Windows API):

```
BOOL WINAPI FindNextFile(
 _In_ HANDLE hFindFile,
 _Out_ LPWIN32_FIND_DATA lpFindFileData
);
```

That’s Weird… Why?!

- **Reason 1**: C does not have exceptions
  - Problem in a Java/Python function? Throw an exception!
  - Exceptions are a bit controversial among programming language designers
- In C, the return value of functions often indicates success or failure, called a ***status code***
- Functions don’t ***have*** to be designed this way, but it’s a very common convention

Efficiency

- **Reason 2**: Speed!
- Return values have to be copied back to the calling function
  - Say my function returns a bitmap image. The entire thing is going to get copied!
- In a language that focuses on speed and efficiency, updating the values directly in memory is faster
- Imagine transferring lots of large strings, objects, etc. around your program, copying them the whole time

Arguments/Return Values: How to Know?

- The return value **might** indicate a status code… and it might not. 💩
- To be sure, use the `man` (manual) pages
  - (You could also google it, but that can occasionally lead you to the wrong documentation / advice)
- The C documentation is in section **3** of the man pages:
  - `man 3 printf`
- Each man page will explain how the arguments and return values are used

Error Messages

- Many C functions return a status code **and** set `errno`
  - Global variable that contains the last **error number**
- You can use the `perror()` function to convert this number into plain English (or your local language)
- Pass in a string prefix to help you trace your code:
  - Call `perror("open");` after `open(...)` function call
  - Result: `open: No such file or directory`
    - (assuming the file being opened didn't actually exist)

void Argument [1/3]

- In C, there’s a difference between `function()` and `function(void)`
- void arg: the function takes no arguments
- Empty arg list: the function may or may not take arguments
    - If it does, they can be of any type and there can be any number of them

void Argument [2/3]

- Why is this important?
- First, to understand older code
- From the C11 standard:
  - “The use of function declarators with empty parentheses (not prototype-format parameter type declarators) is an obsolescent feature.”
- Second, this may lead to incorrect function prototypes or passing incorrect args in your code

void Argument [3/3]

So, to sum up:

```c
/* Takes an unspecified number of args: */
void function();
```

And:

```c
/* Takes no args: */
void function(void);
```

Today's Schedule

- Pointers
- Argument Passing Conventions
- **Arrays**

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

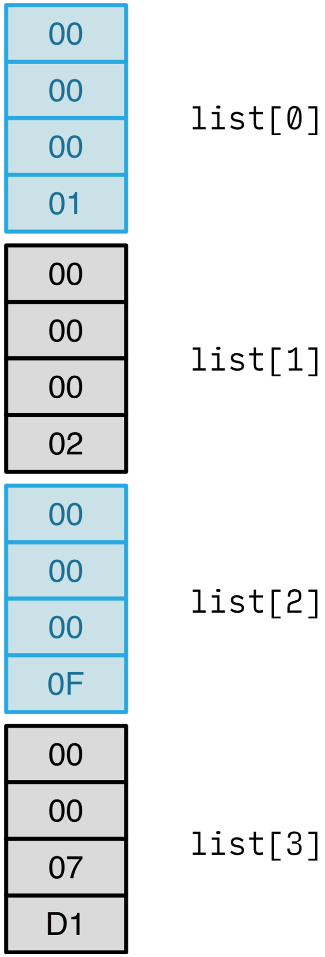
- What happens when we retrieve the value of `list[5]`?
- Find the location of `list` in memory
- Move to the proper offset: `5 * 4 = byte 20`
  - Assuming `sizeof(int) = 4`
- Access the value
- Accessing, say, `list[500]` is just moving to a position in memory and retrieving whatever is there

Visualizing Arrays in Memory

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

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

sizeof(int) = 4
```



00
00
00
01
list[0]

00
00
00
02
list[1]

00
00
00
0F
list[2]

00
00
07
D1
list[3]

- 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

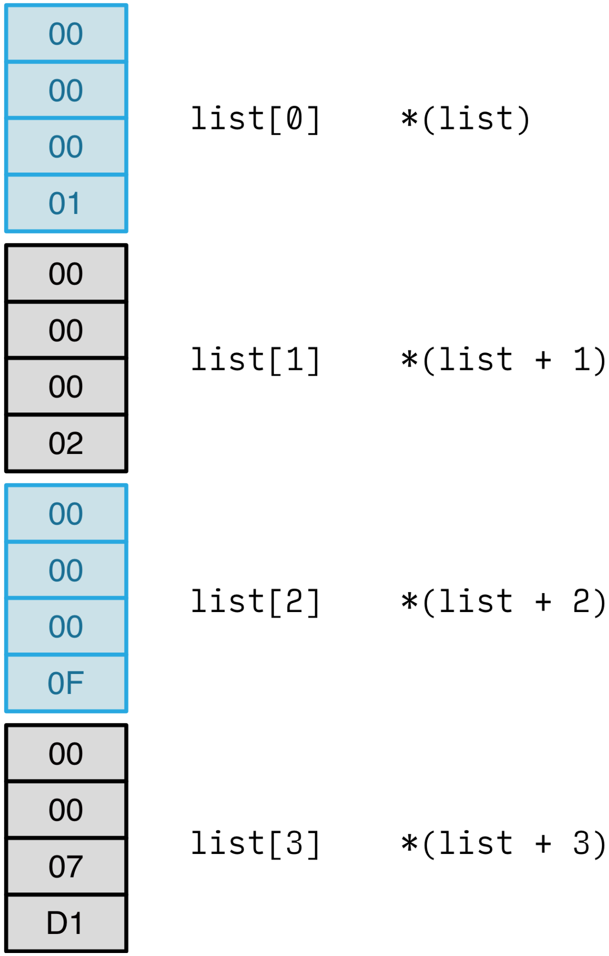
Pointer Arithmetic

- 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
```



00	list[0]	*(list)
00		
00		
01		
00	list[1]	*(list + 1)
00		
00		
02		
00	list[2]	*(list + 2)
00		
00		
0F		
00	list[3]	*(list + 3)
00		
07		
D1		

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!)

Lab 2

- Lab 2 will give you a chance to work with pointers
- We'll create a ***reciprocal cipher*** function that takes characters and rotates them to "encrypt" strings
- Let's set this up and then take a tour of the code
    - (clone the repo and then continue on for info about Makefiles)

Make

- All 521 projects will include a `Makefile`
  - This tells the `make` utility what to do
  - Essentially just a recipe for building your program
  - All our projects are composed of several C files
- Hints (applies to Lab 3 as well):
  - `make` – compile and produce executable
  - `make test` – run the test cases
  - `make clean` – clean up all build artifacts

How make Works

Makefiles are a recipe composed of instructions like this:

```
target: dependency
	instructions
```

(Note the tabs)

- You provide a target, like ‘array’ – the name of the file that gcc will generate
- The dependency tells us what files you need to build your program. In this case, it’s ‘array.c’

Let’s go!

Time to start working on the lab.