CS 326: Operating Systems

Remembering How to C

Lecture 3

Today's Schedule

- C Background
- Lab 1
- C Tips

Today's Schedule

- **C Background**
- Lab 1
- C Tips

Oh Say Can You C?

- All the assignments in this course will be in C
  - This is less about ***torturing you*** and more about using the **right tool** for the job 😄
- The C programming language was invented around 1970
  - It's old.
  - It has decades of baggage
  - Legend has it that Dennis Ritchie invented it while he was riding around in his horse-drawn carriage

What C is Useful For

- Nearly all operating systems are written in C
  - Linux: C!!!
  - macOS: most of the low-level functionality is C
  - Windows: C and (some) C++
- Android uses the Linux kernel, iOS is a macOS fork
- Embedded systems: elevators, refrigerators, routers, TVs are often written in C
- High-performance software is usually written in C

Why are OS written in C?

- It's fast
- It's reasonably high level (at least we don't have to program in assembly!)
  - Still easy to manipulate bits, registers, and other low-level constructs
- We have complete control over memory allocation
- Great for interfacing with hardware

TIOBE Language Rankings

Indisputable proof that C is the best:

Jan 2021	Jan 2020	Change	Programming Language	Ratings	Change
1	2	^	C	17.38%	+1.61%
2	1	v	Java	11.96%	-4.93%
3	3		Python	11.72%	+2.01%
4	4		C++	7.56%	+1.99%
5	5		C#	3.95%	-1.40%

(see https://www.tiobe.com/tiobe-index/)

What the Future Will Hold

- For the first time in a ***very*** long time, it's starting to look like C might have some challengers
  - Nim, Zig, Rust
- **Rust** has built up a fairly large following for systems programming
  - Lots of features that make it modern but still highly performant
  - Several experimental Rust OS are under development
- For now, knowing C is still a valuable skill.

Today's Schedule

- C Background
- **Lab 1**
- C Tips

Let's C What You've Got

- It's Lab 1 time… on paper!
  - Debugging buggy C programs
  - Split up into groups of 5 (maximum), and I'll pass out a paper copy of the lab
- The challenge: can you figure out what's wrong with these functions without running / compiling them?
- Once you've worked on these for a bit, let's regroup to remind ourselves about all the things that make C "fun"

Today's Schedule

- C Background
- Lab 1
- **C Tips**

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
- If you want to make outside changes to a variable passed to a function, then you **must** use pointers
  - They are still passed by value; the value is the memory address
  - Arrays are (kind of) fancy syntax for pointers so you can modify their contents in a function

Pointers

- Although C **only** supports passing by value, we can implement pass by reference with pointers
  - After passing the value of the pointer (memory address), we can **dereference** it (`*` operator) to retrieve/change the data it points to
- `&` – the 'address of' operator.
- `int * x;` – defines a *pointer*. Note that this doesn't create an integer, it creates a **pointer to an integer**.
  - It doesn't matter *what* type it points to; a pointer is always going to just be a memory address

C Argument Conventions

- One thing that sets C apart is function arguments are frequently used for both **input** ***and*** **output**
  - e.g., `strcpy(dest, src)`
    - `dest` is modified, `src` is used as the input
- Return values are often used for **status codes**
  - Whether or not the function succeeded
  - C doesn't have exceptions, so we have to check that everything worked ourselves by inspecting these!

Arrays

- In C, arrays let us store a collection of values of the **same** type
- C will set aside space for the array in memory:

```
(num_elements * sizeof(data type))
```

- To simplify, you can think of arrays as a pointer to the beginning of that memory
  - Accessing elements means doing pointer arithmetic

Accessing Array Elements

- Retrieving the values of an array is the same as it is in Java:
  - `list[2] = 7`
  - `list[1] = list[2] + 8;`
- List access: find the starting address of `list`, then add `index * sizeof(data type)` bytes
- But there is **NO** boundary checking!
  - `list[500] = x;` may work even if `list` is only made up of 100 elements

Mutability

- When you initialize a string like this:
  - `char str[] = "Hello world!";`
- …the contents will be allocated as a mutable array
- But when you do this:
  - `char *str = "Hello world!";`
- …you are only creating a pointer to a ***string literal***

Strings as Arrays

- Let's look at C strings:

- Note how our string contains 6 characters, but the array representation has 7
- The `\0` is the NUL byte, a control character
  - Just like `\n`, etc., we write it with two characters but it is just shorthand for a single character
  - Its value also happens to be 0 (decimal)
  - C string functions assume this is present; if it's not, you only have an **array of characters** and your program will crash

Array Decay

- When a C array is passed to a function, we lose its dimension information
  - It *decays* to a pointer (*array decay*)
- This is why we generally pass the size of arrays to functions
- This would be extremely inconvenient with strings, so using `\0` to denote the end of a string is an acceptable tradeoff
  - meh!

Command Line Arguments

- The `main` function receives command line arguments:
  - `int main(int argc, char *argv[])`
- We receive two parameters:
  - `argc` – the number of command line arguments
  - `argv` – the arguments themselves
- Some notes:
  - `argc` will always be at least 1
  - `argv` will always start with the name of your program

Final Tip

- Remember: C is different than Java or Python
  - Don't try to write C in Java or Python style; instead, embrace the fun 😄