CS 107
Lecture 5:
Strings, Arrays, Pointers

Monday, October 9, 2017

Computer Systems
Fall 2017
Stanford University
Computer Science Department

Lecturers: Julie Zelenski and Chris Gregg
Brainstorm: How to approach this class?
What strategies are working? Which aren’t?

Value of code reading
Intensive vs extensive

Striking right balance
Experimenting with system can be a great way to learn!
But correct answer can be meaningless without correct understanding
Today’s topics

- Quick tour of `<string.h>`
- Why pointers?
- Pointer operations
- Tracing pointer use, memory diagrams
- Pointer vs array
Let’s code!

/afs/ir/class/cs107/samples/lect5

pig.c
Why pointers?

◆ Pointers facilitate sharing, efficiency
  Linked data structures (lists, trees, graphs)
  Efficiently pass/return without making additional copies
  Avoid redundancy in data structures, link to one version of truth

◆ In C, pointers are ubiquitous
  Access specific memory location by address (e.g. system peripherals)
    Can also take the address of any L-value and then manipulate as pointer variable
  Pass by reference handled manually
    Pass pointer to function & manually dereference
  Which came first— the array or the pointer?
    In many situations, arrays/pointers interchangeable in C, C-string is just a char *
  Pointer arithmetic
    C allows some "interesting" manipulations on pointers
  Function pointers
    Pointers to code??? Tell me more....
Meet your address space

0xfffffffff000
Stack

8MB reserved

0xfffffffff000
Shared library text/data

Sized for library

0x602018
Heap

Grows on demand

0x600000
Global data

Sized for executable

0x400000
Text (machine code)

Low addresses deliberately unmapped

---

OKAY, HUMAN. HUH?
BEFORE YOU HIT COMPARE, LISTEN UP.
YOU KNOW WHEN YOU'RE FALLING ASLEEP AND YOU IMAGINE YOURSELF WALKING OR SOMETHING.
AND SUDDENLY YOU PISTED, STUMBLE, AND TOOT AHEAD?
Yeah!

YOU SEE WHAT A SEDFUL FEELS LIKE.
DOUBLE-OCKET YOUR DATIN POINTERS, OKAY?
Address: is a number that identifies location of a byte in memory

Pointer: is a variable, value is address, C type is a "pointee *"

&  address-of
  Evaluates to a memory location
  Can be applied to any expression that is "L-value" (can appear on left hand side of assign)
  Type is pointee *

*  dereference
  Evaluates to the pointee at address (i.e. read or write contents at that address)
  Can be applied to any address
  Type is pointee

Address is just an unsigned long
  Regular arithmetic/relational operators apply to numeric value of address
Array declaration

int nums[5];
Allocates space for 5 ints, contiguous memory, indexed from 0 to 4
Use of array name "decays" to address of first element

Pointer declaration

int *ptr;

What is same? What is different?

Operations that work on both
- Dereference
- Pointer arithmetic
- Array indexing

Operations that only work only on pointer
- Can reassign the pointer to hold a different address, not so with array

Does the memory layout look the same? Let's draw it!