C Introduction

CS 2130: Computer Systems and Organization 1

Xinyao Yi Ph.D.
Assistant Professor

UNIVERSITY of VIRGINIA | ENGINEERING

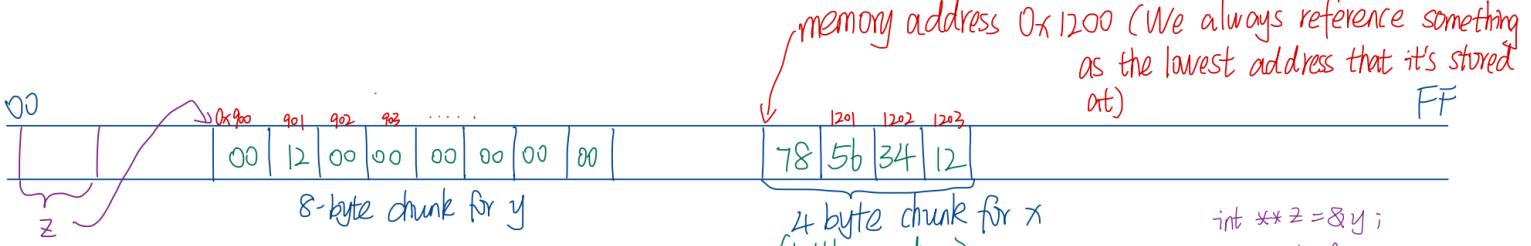
Announcements

- Homework 6 due
- Homework 7 **due Monday at 11:59pm** on Gradescope
- Quiz 7 available today on Gradescope, due Sunday
- Exam 2 next Friday

Data Types in C

Pointers - how C uses addresses!

- Hold the address of a position in memory
- Need to know the kind of information stored at that location



int x = 0x12345678

int *y;

y = &x, (I want y point to x, which means store the address of x in y).

int **z = &y;

*z : dereference z

**z: dereference *z

**z = 0x00000000;
(set x to 0x00000000)

*z = 25; (Warning: you're trying to assign an integer to a pointer)
(But later, when you use it as an address, may seg Fault).

y = 0xABCDEF01; (I'm changing the pointer) → I may don't want to do this. Sometimes seg fault?

*y = 25; (When I use an asterisk, it will say is follow the pointer on y, follow the address to the place in memory that y points to and change that value)

Using Compilers

Arrays

(We pick a spot in memory, and the next so many spots are our array)

Array: 0 or more values of same type stored contiguously in memory

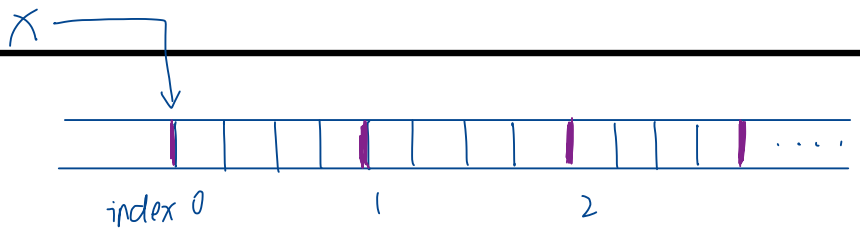
- Declare as you would use: `int myarr[100];` (100 integers in my array)
- `sizeof(myarr)` = 400 — 100 4-byte integers (How big is my array in bytes?)
- `myarr` treated as pointer to first element
- Can declare array literals:

```
int y[5] = {1, 1, 2, 3, 5}
```

(optional)

When I create an array, it actually create a pointer to the first thing in that list.

Pointers and Arrays



(dereference x will be the first element of array x)

`*x` and `x[0]` are equivalent

- Pointer to single value and pointer to first value in array
- Treat array as pointer to the first value (lowest address)
- Indexing into array: `x[n]` and `*(x+n)` → pointer arithmetic
  - If `x` is an `int *`, then `x+1` points to **next int** in memory
  - Adding 1 to pointer adds `sizeof()` the type we're pointing to

Not skip 5 bytes in memory

I know this pointer is a pointer to an integer, so I plus 5 * sizeof(int)

Pointers and Arrays

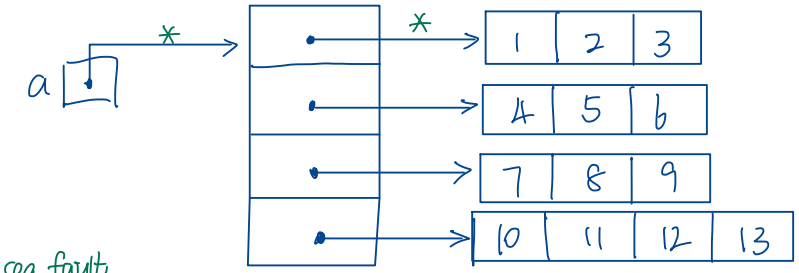
one pointer or an array of pointer?

How do I know? sizeof( )

one integer or an array of integer?

Consider: **int **a** (a pointer to a pointer to an integer)

I suppose "a" point to the first thing of an array. This array should be an array of pointers
And I suppose the pointers in this pointer array point to an array of integers.



They are just pointers to a place in memory.
It's doesn't matter how big the arrays are

```
**a ⇒ 1
a[0][0] ⇒ 1
*(a[2]) ⇒ 7
(*a)[2] ⇒ 3
a[1][3] ⇒ Something weird here. In Java, just seg fault.
a[1][0] ⇒ 4
a[0][1] ⇒ 1
*a[1] ⇒ *(a[1]) ⇒ 4
```

But in C, I can just give you the value at that place.
① Maybe seg fault if you read too far
② Maybe some number there?

I will know something wrong when I use this value later. And if I'm writing sth, I may overwrite sth I don't know.

Pointers

sizeof(char *) == sizeof(long double *)

- All pointers are the same size: address size in underlying ISA
- Two special int types (defined using typedef)
  - `size_t` - integer the size of a pointer (unsigned)
  - `ssize_t` - integer the size of a pointer (signed)
  - With our compiler and ISA, these are both variants of `long`

sizeof() returns a value of type size_t

→ These represent integers with the same size as a pointer.

Pointers

Consider the following code:

```
int x = 10;
int *y = &x;
int *z = y + 2;
long w = ((long)z) - ((long)y);
```

We suppose x lives at address 1000

y point to x, so y == 1000

pointer arithmetic, y points to an integer, so y+2 means go forward 2 ints in memory. which is 4 bytes

since y was 1000. so z will be 1000+8 = 1008

cast the address to long. So we treat them like plain integers. 1008-1000 = 8

Why is w = 8?

Other Types and Values

if I write numbers, they are implicitly cast to integers, which is 32 bits.

If I want to write something longer, I need to add some suffixes.

- Literal values - integer literals are implicitly cast
  - `unsigned long very_big = 9223372036854775808uL`
  - `u` for unsigned, `L` for long, `LL` for long long (capitalization doesn't matter, but you'll usually see "u" lowercase and "L" uppercase.
- `enum` - named integer constants (in ascending order)
  - `enum { a, b, c, d=100, e };` (when I dealing with choices the user has to make, the easiest way to do is using integers, for easier reading. I give them some names)
    0 1 2 101
    I can change where I'm at using "="
    `int foo = e;` It's a type!
- `void` - a byte with no meaning or "nothing" → I can't do equals or any operations on it.
  - Pointers: `void *p` (I don't care what's there)
  - Return values: `void myfunction();` (I don't care what is in register rax).

```
float b = 123.4;
void *p = &b;
float c = *(float*)p;
```

- Casting - changing type, converting
  - Integer: zero- or sign-extend or truncate to space
  - Int to float: convert to nearby representable value
  - Float to int: truncate remainder (no rounding)

casting pointers does some weird things. we'll talk about that later.