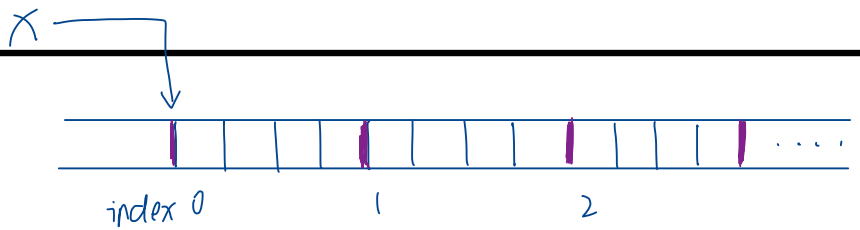
# C Introduction

## CS 2130: Computer Systems and Organization 1

**Xinyao Yi** Ph.D.
Assistant Professor

UNIVERSITY of VIRGINIA | ENGINEERING

# 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

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.

# Structures

## `struct` - Structures in C

- Act like Java classes, but no methods and all public fields
- Stores fields adjacently in memory (but may have padding)
- Compiler determines padding, use `sizeof()` to get size
- Name of the resulting type includes word `struct`

```
struct foo {
    long a;
    int b;
    short c;
    char d;
};

struct foo x;
x.b = 123;
x.c = 4;
```

the order of this list is important. It tells the compiler how to build this thing.

4+2+1=7

may be one byte of padding?
I don't know.

↳ declare a variable x of type struct foo.

I can set directly the fields.

When you create a struct in C, the compiler decides where to place each field in memory. It tries to align each field on an address that's a multiple of it's natural size, to make the CPU access faster.



A struct in a struct?

Yes, just set a struct as one field of another struct.

## Structure Literals

We can use struct literals just like we saw with the array literals.

```
struct a {
    int b;
    double c;
};

/* Both of the following initialize b to 0 and c to 1.0 */
struct a x = { 0, 1.0 };
struct a y = { .b = 0, .c = 1.0 };
```

"a" is a part of the type, not a part of the name.

→ must give values to fields in order.

→ you can use equal notation to give them values in other orders.

`struct a z;` ⟹ If I don't give any values, just allocate the memory and whatever is in memory is whats going to be there. I don't know what they are.

# typedef

`typedef` - give new names to any type!

typedef unsigned long size_t;
typedef unsigned long address_t;

- Fairly common to see several names for same data type to convey intent
- Ex: `unsigned long` may be `size_t` when used in sizes
- Examples:

```
typedef int Integer;
Integer x = 4;
typedef double ** dpp;
```

Integer becomes another name for int.

dpp ptr; (Using typedef can make complex pointer types look simpler)

- Used with *anonymous structs*:

```
typedef struct { int x; double y; } foo;
foo z = { 42, 17.4 };
```

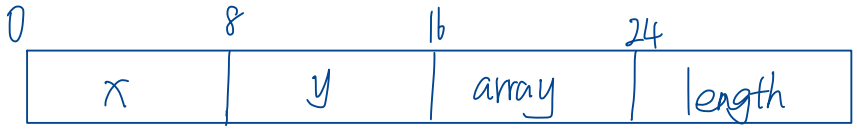
struct { ... } defines an anonymous struct (no struct name given).
The typedef immediately gives it the alias foo.
You can now create variables like foo z; instead of writing struct something z;.

# Struct Example

```
typedef struct {
    long x;
    long y;
    long *array;
    long length;
} foo;
```

remember we need this ";" at the end!

Assuming I have no padding because all of these are 64 bits.




foo a;

a.array ⇒ "a+16" to get me to the array.