

C Introduction

```
long x = 0x12345678;  
double y = (double) x;  
double z = x * (double) 2;  
// does z == y?
```

CS 2130: Computer Systems and Organization 1
October 31, 2025

Announcements

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

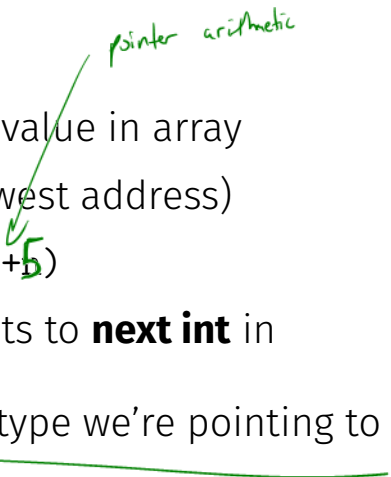
Arrays

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

- Declare as you would use: `int myarr[100];`
- `sizeof(myarr) = 400` — 100 4-byte integers
- `myarr` treated as pointer to first element
- Can declare array literals:
`int y[5] = {1, 1, 2, 3, 5}`

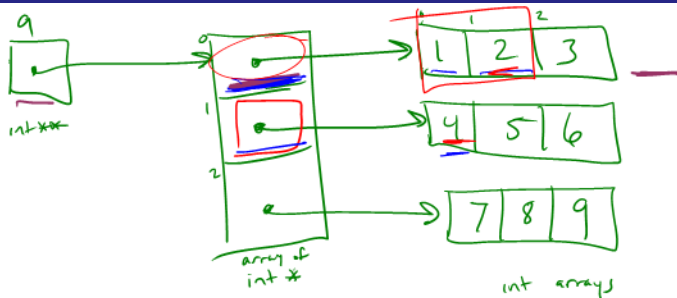
Pointers and Arrays

`*myarr` and `myarr[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: `myarr[5]` and `*(myarr+5)`
 - If `myarr` is an `int *`, then `myarr+1` points to **next int** in memory
 - Adding 1 to pointer adds `sizeof()` the type we're pointing to
- 

Pointers and Arrays

Consider: `int **a`



`**a = 1`

`a[0][0] = 1`

`(*a)[1] = 2`

`*(a[1]) = 4`

`a[0][3] = ?`

`a[0][1] = 2`

`a[1][0] = 4`

`*(long *)*a = 0x0060060206000001;`

`a[0][3] = 5;`

Pointers

- All pointers are the same size: address size in underlying ISA
- Two special integer 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`

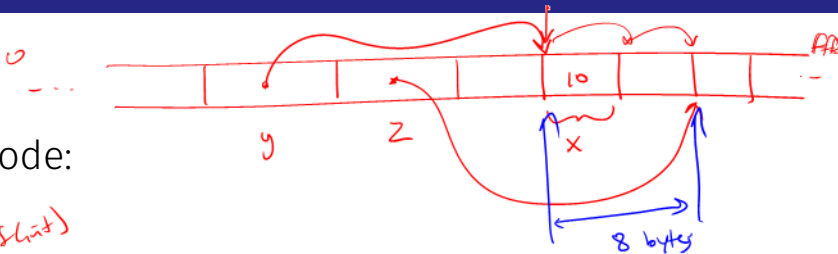
Pointers

Consider the following code:

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

Handwritten notes:
- A red arrow points from the `2` in `y + 2` to the text `2 * sizeof(int)`.
- Blue underlines are under `((long)z)` and `((long)y)` in the last line of code.

Why is `w = 8`?



Other Types and Values

- Literal values - integer literals are implicitly cast
 - unsigned long very_big = 9223372036854775808uL
 - u for unsigned, L for long
- enum - named integer constants (in ascending order)
 - enum { a, b, c, d=100, e };
int foo = e;
- void - a type with no meaning or "nothing"
 - Pointers: void *p
 - Return values: void myfunction();

* (float *) p

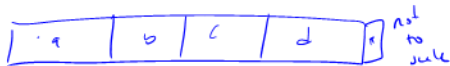
Other Types and Values

- 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)

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;
```

Structure 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 };
```

typedef

typedef - give new names to any type!

- 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;
```

- Used with *anonymous structs*:

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