CS61 Scribe Notes 9/11
Contributors: Kevin Eskici, Aman Kandola, Erin Masatsugu, Donny Yung

Administrivia: Instructions for submitting intermediary check-in to come!

Lecture 4: Data Representation

mexplore.c
-every int takes 4 bytes...even if you put characters in them :)

Difference between two pointers into the same array is the difference between the indices they point to.

- ie. &a[1] - &a[0] = 1, even when a is an int array.

Pointers in C are names for objects whose values contain addresses which are just numbers.

uintpntr_t is a special type of signed integer big enough to hold a pointer.

If you cast the pointers as uintptr_t's to treat them solely as addresses, the difference between the addresses will be 4 bytes, the exact difference between the addresses.

If you subtract two int pointers, the result is the distance of the addresses divided by the type size.

Integer arithmetic
b - a //normal math result

Pointer arithmetic
b - a //b and a type T*
result is ((uintpntr_t)b - (unintpntr_t) a) / sizeof(T)
(technically we should have used signed types since a negative result is valid, so that we could do arithmetic in the integer space)
i.e.
&a[0] - &a[1] = -1

Can only subtract pointers that have the same type.
Also cannot add pointers together, it just doesn't make sense.
If a>0 and b>0, it is not guaranteed that a+b>0, because of integer overflow

You don't need to cast pointers when assigning if the variable type and pointer type are the same (but it can be stylistically good)

<u>Computer Arithmetic</u> (what actually happens)
Can't assume that if you have
a > 0 and b > 0
that a + b > 0

what is -1? A number that when you add one to to get 0.
On our computers this number is 255 !

<u>Two's Complement Arithmetic</u>
We get cool operations as a result like:
-~X -> Flip All Bits of X
To get the negative of a number, ~ it and add 1.
-x == ~x +1
255 is -1 because if you add 1 to it you get 0

&: x & y = 1 iff x and y both =1, 0 otherwise
& represented by a truthtable

_	0	1
0	0	0
1	0	1

| : x | y = 1 if either x, y or both = 1, 0 otherwise
We have |

_	0	1
0	0	1
1	1	1

^ : x^y = 1 only if either x or y = 1 but not both, 0 otherwise
We have ^

_	0	1
0	0	1
1	1	0

If running on a 32 bit machine 2^32 - 4 = -4

```
char* allocate_with_extra(size_t x){
        return malloc(x + 1);
}
//you get x + 1 bytes back
```

if you called malloc with 2^32 -1, in which case you get (nil) since it can't give you that much space (this isn't dropbox)
malloc is allowed to return a non-null result when you call it with 0

Structs:

```
struct meta {
        int a;
};
```

declaration:
`struct meta m = {m};`

if you want to declare without writing struct you can also do

```
struct meta{
        int a;
}; typedef struct meta meta
```

or even

```
typedef struct meta {
        int a;
} meta;
```

Size of structure is only the size of the stuff in it-there is no overhead space cost at all!

```
typedef struct meta {
        int a;
        char ch;
        int b;
} meta;
```

int a, char ch, and int b are stored contiguously and in order in memory
size of the above struct is 12 bytes (not 9!). this is because of alignment.

Alignment of T- A such that every object of type T has addresses multiple of A.

alignment is always equal to or smaller than the size

(assuming a 32 bit machine)

type	size	alignment
char	1	1
int	4	4

structure size has to be a multiple of the alignment of the type in the struct with the biggest alignment. Alignment of structure is the least common multiple of the alignments in the structure. Alignments are always 1, 4, or 8.

Facts about alignment:

1. sizeof(T) is a multiple of alignof(T)
   reason: array layout
2. alignof(struct M) = max{alignof(T)} for all components T
3. Structs need to have proper alignment to provide for the case where you have an array of structs.
4. Alignment in general is important for transferring data between different computers. Data is generally sent/received in 64-bit blocks, but if the data straddles these blocks, problems arise.

Threads:

Scenario: you have 3 CPU's, each of which is in a tight loop.
first one is in a tight loop setting x =1, second is in a tight loop setting x = -1, third is in a tight loop printing x.
You'd expect 1 or -1 to be printed repeatedly
BUT if x was not aligned, it might print 0xFFFF 0001 (wtf?!)
If you want every int you read to be a value you wrote, you need to make sure the int is aligned.

Fun stuff:
Writing sum different ways ends up compiling to the same instructions on the real machine because the instructions are simply bytes.
This means you could replace the function with the string of the instructions and still run it.

You can treat data uniformly as either data or instructions.

Hello kitty is code….but is she a cat?