

6.1903

Pointers, Arrays,
Chars, Ints, Floats...
Data Representation

Fall 2026

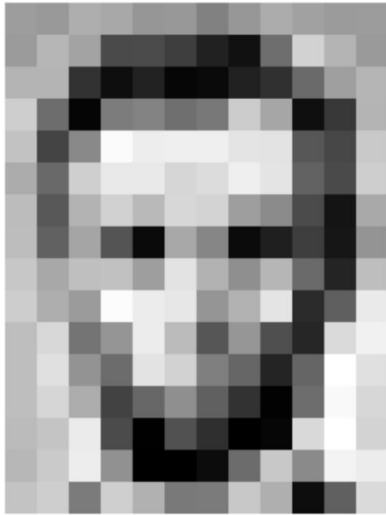
Announcement

There is a quiz instead of lecture next Monday (9/28).
Come to the
usual room: 32-123.

Lecture 3 will be the following Monday (10/5).

Remember to check the course calendar on the
website!

Review: Everything is a number to a CPU



157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	106	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	85	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	106	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	85	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

Hello world!

Character:	H	e	l	l	o	␣	w	o	r	l	d	!
Decimal:	72	101	108	108	111	32	119	111	114	108	100	33
Binary:	1001000	1100101	1101100	1101100	1101111	0100000	1110111	1101111	1110010	1101100	1100100	0100001

You can have 1's and 0's

- It's all just bits.
- How you use them is up to you: **types!**
- The CPU doesn't know types – Only operations.

The Situation:

n bits can express 2^n things

- 1 bit can represent how many different things?
- 2 bits can represent how many different things?
- 3?...
- 8?

Flip: How many bits needed?

t possibilities need $\text{ceil}(\log_2(t))$ bits

- Is a light on or off...how many bits do we need?
- There are seven days in the week...how many bits do we need to describe what day it is?
- How many states in US?
- All characters/symbols in existence?
- All numbers in existence?

Context Matters

A set of bits means something to **us**, not the CPU.
If we see 0b10100011, we could interpret it as:

unsigned int8	163
signed int8	-93
ASCII (char)	<invalid>
pointer	Address 0xA3
float	$2.28411649685 \times 10^{-43}$ (w/ 0's for upper 24 bits)

The programming language's type system keeps it straight.

Overview Today

- Addresses:



- Pointers

- Arrays

- Pointer Arithmetic

- A little bit on c-strings

- Characters:

- ASCII

- Unicode

- Numbers:

- Ints

- Signed Ints

- Decimals, etc

- Booleans

Recap: Computer Organization

- The smallest unit accessible in memory of most modern computers is a **byte** which is 8 bits.
- Our 6.190 kits use a **32-bit** system which means the memory is organized into **four-byte words**

Address:

Value:

"0"

"4"

"8"

"12"

"16"

"..."

Byte 3	Byte 2	Byte 1	Byte 0
Byte 7	Byte 6	Byte 5	Byte 4
Byte 11	Byte 10	Byte 9	Byte 8
Byte 15	Byte 14	Byte 13	Byte 12
Byte 19	Byte 18	Byte 17	Byte 16

ad nauseum...

Computer Memory

- 4-byte words with addresses.
- Each word contains 32 bits of data
- We'll express the addresses and data in hexadecimal since it is clean but we could do in decimal or binary
- Consecutive words have addresses that are 4 bytes apart.

Address:	Value:
0x00000000	0x068CEBC4
0x00000004	0x4EFBC573
0x00000008	0x1C91CB27
0x0000000C	0x67E35A09
0x00000010	0x77449A8E
0x00000014	0xDF2B917F
0x00000018	0x70DF0957
0x0000001C	0x077B5EF9
0x00000020	0x90097C09
0x00000024	0xD1EFF84C
0x00000028	0xFD68945E
0x0000002C	0x14EEF44E
0x00000030	0x61360243
0x00000034	0xFF8E624F
0x00000038	0x806080A9
0x0000003C	0x1B9A30D7

Li'l Demo

Need three (0b00000011) volunteers.

Memory and Pointers

- Addresses are 32 bits.
- When stored as values, they are called “pointers.”
- C pointers have types (“int *”, “float *”, “char *”, etc.) that tell you how big the object pointed to is.

MEMORY:

Address:	Value:
0x00000000:	0x00000000
0x00000004:	0x420000e4
...	
...	
0x3fc93f50:	0x00000000
0x3fc93f54:	0x0000001a
0x3fc93f58:	0x0000000f
0x3fc93f5c:	0x0000000b
0x3fc93f60:	0x00000000
0x3fc93f64:	0x00000000
0x3fc93f68:	0x00000000
0x3fc93f6c:	0x4201652c
...	
...	
0xffffffff8:	0x00000000
0xffffffffc:	0x0000001a

Pointers Point

- The value of a pointer is a 32 bit memory byte address
- Some data types use different amounts of bytes:
 - One byte (char, uint8_t, int8_t)
 - Two bytes: (uint16_t, int16_t)
 - Four bytes: (uint32_t, int32_t, float)
- In order to know *how many bytes* to read, a pointer needs to have a **type**!
- In order to know *how* to interpret the memory pointed to, a pointer needs a **type**!

Pointer Syntax

```
int x; // assume x was declared as integer somewhere
```

- & means one thing: Get the address of a variable

```
int *ptr_x = &x; // point int pointer ptr_x to address of x
```

- * kinda means two things!
 - On declaration it specifies that the type is a pointer:

```
int *ptr_x = &x; // point int pointer ptr_x to address of x
```

- When used as an operator it is “dereference” which means, “get the value at the address.”

```
int result = *ptr_x + *ptr_y; // add thing ptr_x points  
                             // to thing ptr_y points
```

Aside: Sloppy Syntax

- When declaring pointers, you can do (they all means the same thing):

```
int * ptr_x; // pointer ptr_x of type int
```

```
int *ptr_x; // pointer ptr_x of type int
```

```
int* ptr_x; // pointer ptr_x of type int
```

Working With Pointers

- What will this output?

```
int main(void){
    int x = -16;           // make signed int -16
    int* ptr_x = &x;       // point pointer ptr_x to address of x
    int y = -32;           // make a int of -32
    int * ptr_y = &y;       // point pointer ptr_y to address of y
    int result = *ptr_x + *ptr_y; // add thing ptr_x points
                                // to thing ptr_y points
    printf("%d + %d is %d\n", x,*ptr_y,result);
}
```

Output:

```
-16 + -32 is -48
```

Destroying the World with Pointers

- What will this output?

```
int main(void){
    int x = -16;        // make signed int -16
    int* ptr_x = &x;    // point pointer ptr_x to address of x
    float y = -32;      // make a float of -32
    int* ptr_y = &y;    // point pointer ptr_y to address of y
    int result = *ptr_x + *ptr_y; //add thing ptr_x points
                                // to thing ptr_y points
    printf("%d + %f is %d\n", x,y,result);
}
```

Output:

```
-16 + -32.000000 is -1040187408
```

Interpreting the contents at &y as an int is a problem. y is a float, but ptr_y is an int pointer. When we dereference, we're interpreting the bits as if they encode an integer

Destroying the World with Pointers

- What will this output?

```
int main(void){
    char c = 'J';
    char* ptr_c = &c; // point ptr_c to c's address
    int* ptr_d = &c;   // point ptr_d to c's address
    printf("ptr_c points to %c. ptr_d points to %d\n", *ptr_c, *ptr_d);
}
```

Output: `ptr_c points to J. ptr_d points to 525944906`

Interpreting the contents at &c as an int was a problem. A char is not an int...their encodings and sizes are different! See that in a little bit!

Destroying the World with Pointers

- What will this output?

```
int main(void){
    int8_t e = 22;           // 8 bit signed int e
    int8_t* ptr_e = &e;      // point to it with correct pointer type
    int* ptr_f;              // make a 32 bit signed pointer
    ptr_f = &e;              // point a (32 bit) signed int to e
    printf("ptr_e points to %d. ptr_f points to %d\n", *ptr_e, *ptr_f);
}
```

Output: `ptr_e points to 22. ptr_f points to 923355158`

Interpreting the contents at &e as a 32-bit int was a problem. Both pointers are of integer type but different sizes of integers! (more on that in a little bit!!!)

Destroying the World with Pointers

- What will this output?

```
int main(void){  
    int8_t e = 22;          // 8 bit signed int e  
    int8_t* ptr_e = &e;    // point to it with correct pointer type  
    printf("ptr_e points to %d.\n", ptr_e);  
}
```

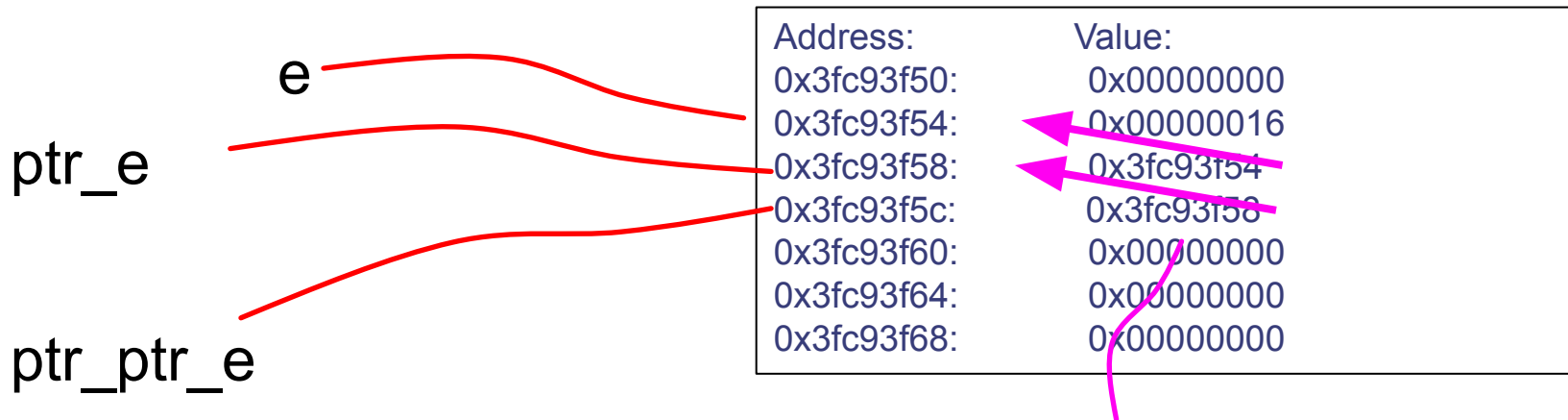
Output: `ptr_e points to 1829778399.`

Forgot to dereference!...printed the address of e rather than the value in e's memory

Pointers Themselves Live in Memory

- A pointer is itself a variable....which lives in memory...which means it can be pointed to as well...we call that a pointer pointer

```
int main(void){  
    int e = 22;           // 32 bit signed int e  
    int* ptr_e = &e;      // point to it with correct pointer type  
    int** ptr_ptr_e = &ptr_e; // make a pointer to a pointer  
}
```



The values of pointers are the addresses of other variables

Pointers Themselves Live in Memory

- Consider this:

```
int main(void){  
    int e = 22;          // 32 bit signed int e  
    int* ptr_e = &e;     // point to it with correct pointer type  
    int** ptr_ptr_e = &ptr_e; //make a pointer to a pointer  
    **ptr_ptr_e = 15;  
    printf("The value of e is %d \n", e);  
}
```

Output: The value of e is 15

ptr_ptr_e points to ptr_e, so we need to first dereference to get to ptr_e's value, which is itself a reference (address) to e, so we need to then dereference a second time!

Pointers All the Way Down...

- The whole thing can keep going
- `int i`: `i` is data of type `int`
- `int * i`: `i` is a pointer to data of type `int`
- `int ** i`: `i` is a pointer to a pointer to data of type `int`
- `int *** i`: `i` is a pointer to a pointer to data to object of type `int`
- `int **** i`: `i` is a pointer to a pointer to a pointer to a pointer to data of type `int`

Why have such abominations?

- Because it can sometimes be easier and quicker to modify a thing “in-place” than go through the whole act of returning a value.
- As we’ll see in when we run into assembly the act of “returning” a value in a function may involve additional steps, especially if we’re holding onto a lot of pieces of data at once (many variables)

The Simplified Situation

```
int f1(int x, int y){  
    return x+y;  
}  
void app_main(){  
    int a = 5;  
    int b = 9;  
    int c;  
    c = f1(a,b);  
}
```

*"Take in x and y,
give me back their
sum. I will then
copy it into c."*

```
void f2(int x, int y, int *z){  
    *z = x+y;  
}  
void app_main(){  
    int a = 5;  
    int b = 9;  
    int c;  
    f2(a,b, &c);  
}
```

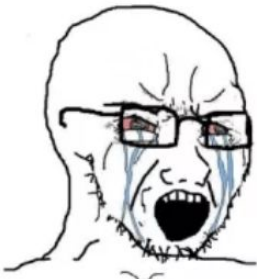
*"Take in x and y, store
their sum where z points.
Just take care of it. I trust
you."*

***These effectively do the same thing, one
modifies "in place" and one modifies externally***

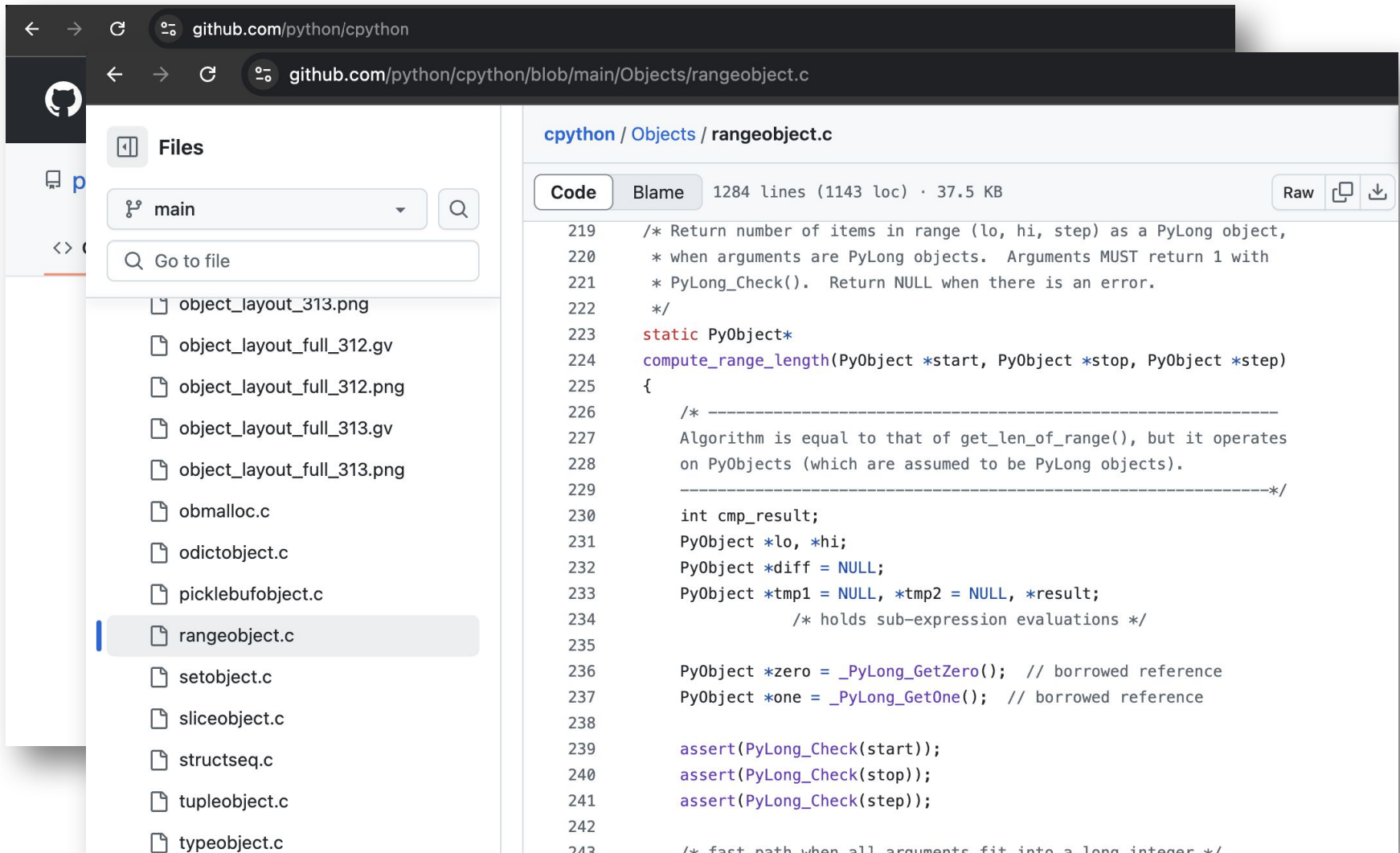
General Theme in C

- C was invented as a “close-to-the-machine” language for writing OS kernels.
- Compared with most modern (and not a few contemporary) languages, there are a lot fewer guardrails* and a lot more trust in the coder.

*guardrails cost resources

C 	Sure buddy! Do whatever you want with that pointer. I trust you!
C++/Rust 	No no no! You're doing it all wrong! You're going to ruin everything!
Java/Python 	There are no pointers in Ba Sing Se

Peel Slowly and See



github.com/python/cpython

github.com/python/cpython/blob/main/Objects/rangeobject.c

Files

main

Go to file

- object_layout_313.png
- object_layout_full_312.gv
- object_layout_full_312.png
- object_layout_full_313.gv
- object_layout_full_313.png
- obmalloc.c
- odictobject.c
- picklebufobject.c
- rangeobject.c**
- setobject.c
- sliceobject.c
- structseq.c
- tupleobject.c
- typeobject.c

cpython / Objects / rangeobject.c

Code Blame 1284 lines (1143 loc) · 37.5 KB

Raw

```
219  /* Return number of items in range (lo, hi, step) as a PyLong object,
220  * when arguments are PyLong objects. Arguments MUST return 1 with
221  * PyLong_Check(). Return NULL when there is an error.
222  */
223  static PyObject*
224  compute_range_length(PyObject *start, PyObject *stop, PyObject *step)
225  {
226      /* -----
227       Algorithm is equal to that of get_len_of_range(), but it operates
228       on PyObjects (which are assumed to be PyLong objects).
229       -----*/
230      int cmp_result;
231      PyObject *lo, *hi;
232      PyObject *diff = NULL;
233      PyObject *tmp1 = NULL, *tmp2 = NULL, *result;
234      /* holds sub-expression evaluations */
235
236      PyObject *zero = _PyLong_GetZero(); // borrowed reference
237      PyObject *one = _PyLong_GetOne(); // borrowed reference
238
239      assert(PyLong_Check(start));
240      assert(PyLong_Check(stop));
241      assert(PyLong_Check(step));
242
243      /* fast path when all arguments fit into a long integer */
```

Another Reason for Pointers...Arrays, etc...

- Pointers are valuable since we can refer to **large** memory structures....such as arrays...by just sharing a memory address (and not a copy of the whole thing!)
- In C (and most languages), arrays are continuous chunks of memory.
- At creation, both the type of its elements and its size must be known since we need to “claim” that much memory.
- For an int (32 bits or 4 bytes), a 10 long array is 40 bytes in size.
- For a char (8 bits or one byte), a 10 long array is 10 bytes in size.

Creating an Array does Two Things:

1. Allocates Memory for the array
2. Creates a pointer by which you access that spot in memory

```
int a = 11;  
int* x = &a;  
int y[10] = {1,2,3,4,5,6,7,8,9,10};  
// x and y are treated the same by the CPU! int pointers! (Sort of)
```

Arrays and Pointers

- Consider this code...

```
int y[10] = {1,2,3,4,5,6,7,8,9,10};
*(y+3) = 123;
printf("%d\n",y[3]); // what does this print?
```

- Dereferencing with an offset is identical to indexing! In fact:
 - y[3] == *(y+3)//SAME THING**
- They are equivalent...and goes both ways:

```
int* d;
int y[10] = {1,2,3,4,5,6,7,8,9,10};
*(y+3) = 123;
d = y+3;
printf("%d\n",*(y+3));
printf("%d\n",d[1]); // what does this print?
```

Indexing

- Watch your limits! Arrays exist in memory...and there are no boundary checks!

```
int* d;  
int y[10] = {1,2,3,4,5,6,7,8,9,10};  
*(y+3) = 123;  
d = y+3;  
printf("%d\n",*(y+30)); // what print?  
printf("%d\n",d[100]);  // what print?
```

- It is the coder's responsibility to keep track of the memory boundaries...sometimes you just get garbage. Other times you get crash!

Slight Difference Between Array Variable and Pointer

- That array's pointer is the only way to access the memory that makes up the array so be careful!
- What would happen below if this could run?

```
int a = 11;
int* x = &a;
int y[10] = {1,2,3,4,5,6,7,8,9,10};
// x and y are treated the same by the CPU! int pointers! (sort of)
y = &a; // not allowed will throw compile error
printf("%d\n",*y);
```

- This protects you from losing access to your memory location!
- An Array cannot be reassigned!

Arrays into Functions

- When an array gets handed into a function an issue arises!: “Arrays devolve to pointers inside functions”
- When an array is handed in as an argument to a function in C, within the scope of the function, all it sees is a pointer.
NOT THE ARRAY!
- The information about how large the array is, usually needs to get sent along with it separately!

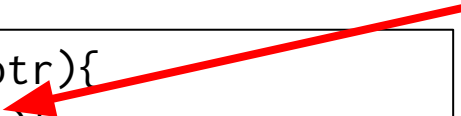
For example

- Consider this code that should double every entry in an array:

unknown

```
void double_every_entry(int* ptr){
    for (int i = 0; i<IUNNO; i++){
        *(ptr+i) = *(ptr+i)*2;
    }
}

void app_main(){
    int y[10] = {1,2,3,4,5,6,7,8,9,10};
    double_every_entry(y);
}
```



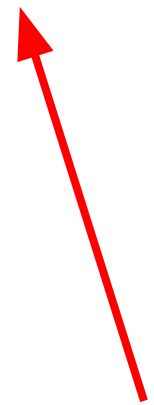
A Fix:

- Hand length/metadata about array in along with array

```
void double_every_entry_better(int* ptr, int length){
    for (int i = 0; i<length; i++){
        *(ptr+i) = *(ptr+i)*2;
    }
}

void app_main_better(){
    int LEN = 10;
    int y[LEN] = {1,2,3,4,5,6,7,8,9,10};
    double_every_entry_better(y,LEN);
}
```

metadata



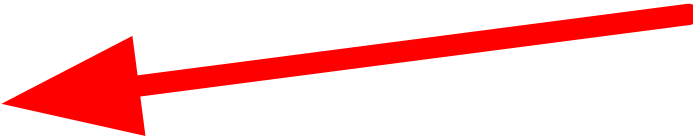
Strings: a type of array

- In C, arrays of chars are how we make basic strings!

```
char x[] = "Cats";           // easier way  
char x[] = {'C', 'a', 't', 's', '\0'}; // longer way
```

- It would be annoying to have to repeatedly hand around second arguments with the array saying how long the string is
- Solution: Use a special character to mark the end of a string (NULL character...see recitation this week)
- Much, much more next week!

Overview Today

- Addresses:
 - Pointers
- Arrays
- Pointer Arithmetic
- A little bit on c-strings
- Characters:
 - ASCII
 - Unicode
- Numbers:
 - Ints
 - Signed Ints
 - Decimals, etc
- Booleans

Characters: ASCII

- ASCII: Represent original computer characters using 8 bits:256 possibilities (though only lower 128 used)
- (what's used in **char** data type)

0b01000001 is 'A'

0b01100001 is 'a'

0b00000000 is NULL

Dec	Bin	Hex	Char	Dec	Bin	Hex	Char	Dec	Bin	Hex	Char	Dec	Bin	Hex	Char
0	0000 0000	00	[NUL]	32	0010 0000	20	space	64	0100 0000	40	@	96	0110 0000	60	`
1	0000 0001	01	[SOH]	33	0010 0001	21	!	65	0100 0001	41	A	97	0110 0001	61	a
2	0000 0010	02	[STX]	34	0010 0010	22	"	66	0100 0010	42	B	98	0110 0010	62	b
3	0000 0011	03	[ETX]	35	0010 0011	23	#	67	0100 0011	43	C	99	0110 0011	63	c
4	0000 0100	04	[EOT]	36	0010 0100	24	\$	68	0100 0100	44	D	100	0110 0100	64	d
5	0000 0101	05	[ENQ]	37	0010 0101	25	%	69	0100 0101	45	E	101	0110 0101	65	e
6	0000 0110	06	[ACK]	38	0010 0110	26	&	70	0100 0110	46	F	102	0110 0110	66	f
7	0000 0111	07	[BEL]	39	0010 0111	27	'	71	0100 0111	47	G	103	0110 0111	67	g
8	0000 1000	08	[BS]	40	0010 1000	28	(	72	0100 1000	48	H	104	0110 1000	68	h
9	0000 1001	09	[TAB]	41	0010 1001	29	)	73	0100 1001	49	I	105	0110 1001	69	i
10	0000 1010	0A	[LF]	42	0010 1010	2A	*	74	0100 1010	4A	J	106	0110 1010	6A	j
11	0000 1011	0B	[VT]	43	0010 1011	2B	+	75	0100 1011	4B	K	107	0110 1011	6B	k
12	0000 1100	0C	[FF]	44	0010 1100	2C	,	76	0100 1100	4C	L	108	0110 1100	6C	l
13	0000 1101	0D	[CR]	45	0010 1101	2D	-	77	0100 1101	4D	M	109	0110 1101	6D	m
14	0000 1110	0E	[SO]	46	0010 1110	2E	.	78	0100 1110	4E	N	110	0110 1110	6E	n
15	0000 1111	0F	[SI]	47	0010 1111	2F	/	79	0100 1111	4F	O	111	0110 1111	6F	o
16	0001 0000	10	[DLE]	48	0011 0000	30	0	80	0101 0000	50	P	112	0111 0000	70	p
17	0001 0001	11	[DC1]	49	0011 0001	31	1	81	0101 0001	51	Q	113	0111 0001	71	q
18	0001 0010	12	[DC2]	50	0011 0010	32	2	82	0101 0010	52	R	114	0111 0010	72	r
19	0001 0011	13	[DC3]	51	0011 0011	33	3	83	0101 0011	53	S	115	0111 0011	73	s
20	0001 0100	14	[DC4]	52	0011 0100	34	4	84	0101 0100	54	T	116	0111 0100	74	t
21	0001 0101	15	[NAK]	53	0011 0101	35	5	85	0101 0101	55	U	117	0111 0101	75	u
22	0001 0110	16	[SYN]	54	0011 0110	36	6	86	0101 0110	56	V	118	0111 0110	76	v
23	0001 0111	17	[ETB]	55	0011 0111	37	7	87	0101 0111	57	W	119	0111 0111	77	w
24	0001 1000	18	[CAN]	56	0011 1000	38	8	88	0101 1000	58	X	120	0111 1000	78	x
25	0001 1001	19	[EM]	57	0011 1001	39	9	89	0101 1001	59	Y	121	0111 1001	79	y
26	0001 1010	1A	[SUB]	58	0011 1010	3A	:	90	0101 1010	5A	Z	122	0111 1010	7A	z
27	0001 1011	1B	[ESC]	59	0011 1011	3B	;	91	0101 1011	5B	[	123	0111 1011	7B	{
28	0001 1100	1C	[FS]	60	0011 1100	3C	<	92	0101 1100	5C	\	124	0111 1100	7C	
29	0001 1101	1D	[GS]	61	0011 1101	3D	=	93	0101 1101	5D	]	125	0111 1101	7D	}
30	0001 1110	1E	[RS]	62	0011 1110	3E	>	94	0101 1110	5E	^	126	0111 1110	7E	~
31	0001 1111	1F	[US]	63	0011 1111	3F	?	95	0101 1111	5F	_	127	0111 1111	7F	[DEL]

Characters: ASCII

- ASCII: Represent original computer characters using 8 bits: 256 possibilities (though only lower 128 used)

ASCII (1977/1986)

	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C	_D	_E	_F
0_	NUL 0x00	SOH 0x01	STX 0x02	ETX 0x03	EOT 0x04	ENQ 0x05	ACK 0x06	BEL 0x07	BS 0x08	HT 0x09	LF 0x0A	VT 0x0B	FF 0x0C	CR 0x0D	SO 0x0E	SI 0x0F
1_	DLE 0x10	DC1 0x11	DC2 0x12	DC3 0x13	DC4 0x14	NAK 0x15	SYN 0x16	ETB 0x17	CAN 0x18	EM 0x19	SUB 0x1A	ESC 0x1B	FS 0x1C	GS 0x1D	RS 0x1E	US 0x1F
2_	space 0x20	! 0x21	" 0x22	# 0x23	\$ 0x24	% 0x25	& 0x26	' 0x27	(0x28	) 0x29	* 0x2A	+ 0x2B	, 0x2C	- 0x2D	. 0x2E	/ 0x2F
3_	0 0x30	1 0x31	2 0x32	3 0x33	4 0x34	5 0x35	6 0x36	7 0x37	8 0x38	9 0x39	: 0x3A	; 0x3B	< 0x3C	= 0x3D	> 0x3E	? 0x3F
4_	@ 0x40	A 0x41	B 0x42	C 0x43	D 0x44	E 0x45	F 0x46	G 0x47	H 0x48	I 0x49	J 0x4A	K 0x4B	L 0x4C	M 0x4D	N 0x4E	O 0x4F
5_	P 0x50	Q 0x51	R 0x52	S 0x53	T 0x54	U 0x55	V 0x56	W 0x57	X 0x58	Y 0x59	Z 0x5A	[0x5B	\ 0x5C	] 0x5D	^ 0x5E	_ 0x5F
6_	` 0x60	a 0x61	b 0x62	c 0x63	d 0x64	e 0x65	f 0x66	g 0x67	h 0x68	i 0x69	j 0x6A	k 0x6B	l 0x6C	m 0x6D	n 0x6E	o 0x6F
7_	p 0x70	q 0x71	r 0x72	s 0x73	t 0x74	u 0x75	v 0x76	w 0x77	x 0x78	y 0x79	z 0x7A	{ 0x7B	 0x7C	} 0x7D	~ 0x7E	DEL 0x7F

Letter

Number

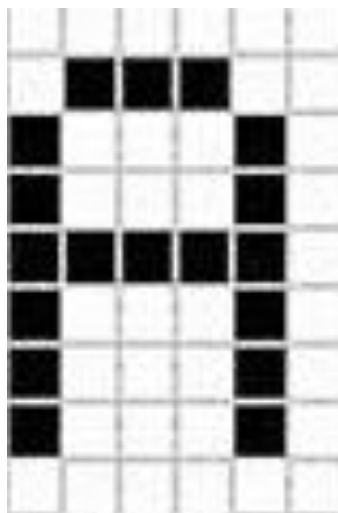
Punctuation

Symbol

Other/non-printable

The char encoding is separate from the render (see in lab 2)

- When a computer stores 'A' it only stores 8 bits. The “render” of the character is looked up using other storage.
- In this way, when an email is sent, the 8bit encoding for 'A' is all that is sent, not the high-resolution of the letter 'A' which takes more bits (data)



54 bit map for
letter A from a
font set

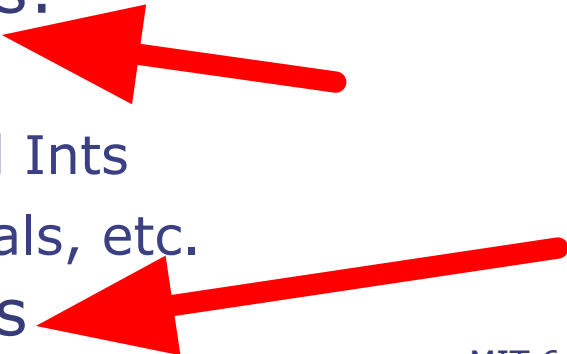
Characters: Unicode

- More modern (current) text representation
- ***Up to**** 4-bytes (32 bits) vs. 1 byte/8 bits of ASCII
- ASCII still uses only one byte...but other symbols use more than that. (clever reverse-compatible variable-length encoding scheme)
- Represent the ~1.1 million possible characters in Unicode...we've assigned 155,063 so far...got room to grow!
- Assignment of meaning is carried out by **Unicode Consortium in Mountainview, CA**

8 bits is a byte (now)

- For a multitude of reasons, including the fact that it is the lowest power of two to hold a single ASCII symbol, most computers settled on having **an 8 bit byte** being the fundamental unit of accessible memory in the 1970s.
- This means:
 - Can't read/write *less than one byte of memory*
 - The **minimal size** of a variable became 8 bits (one byte)
 - As computers got "bigger" it was easier to expand in clean multiples of prior architectures...so from 8 bits went to:
 - 16 bit computers
 - 32 bit computers (our RISC-V embedded board in 6.1900)
 - 64 bit computers (most modern PCs)

Overview Today

- Addresses:
 - Pointers
 - Arrays
 - Pointer Arithmetic
 - A little bit on c-strings
 - Characters:
 - ASCII
 - Unicode
 - Numbers:
 - Ints
 - Signed Ints
 - Decimals, etc.
 - Booleans
- 
- Two red arrows are drawn on the slide. One arrow points from the right towards the word 'Ints' in the 'Numbers' list. The other arrow points from the right towards the word 'Booleans' at the bottom of the list.

Boolean

- Original C doesn't* really have a “bool” **type**.
- Part of the reasoning was since a Boolean only needs **one bit** to express itself, and since **one byte (8 bits)** is the smallest accessible memory unit, they left it up to the programmer to decide how they wanted to do that.
- In general an unsigned int is used such that:
 - 0 is false
 - Everything else is true

*C99 did bring in a bool library <stdbool.h>

Recap: Encoding Positive Integers

It is natural to encode positive integers as a sequence of bits. Each bit is assigned a weight. Ordered from right to left, these weights are increasing powers of 2. The value of an N-bit number is given by the following formula:

$$v = \sum_{i=0}^{N-1} 2^i b_i$$

2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4				
2^3	2^2	2^1	2^0								
0	1	1	1	1	1	0	1	0	0	0	0

What value does 011111010000 encode?

$$\begin{aligned} V &= 0 * 2^{11} + 1 * 2^{10} + 1 * 2^9 + \dots \\ &= 1024 + 512 + 256 + 128 + 64 + 16 = 2000 \end{aligned}$$

Smallest number? 0

Largest number? $2^N - 1$

Most arithmetic works out well too!

▪ Add/Subtract: 0b10001001 (137)

 + 0b00000101 (5)

 0b10001110 (142)

▪ Multiply/Divide: 0b00000101 (5)

 * 0b00000110 (6)

 0b00000000 (0)

 0b000001010 (10)

+ 0b0000010100 (20)

 0b0000011110 (30)

Unsigned Ints (uints):

- **uint8_t**: 1 byte (8 bits): 2^8 values: 256 numbers to rep



- **uint16_t**: 2 bytes (16 bits): 2^{16} values: 65,536 numbers



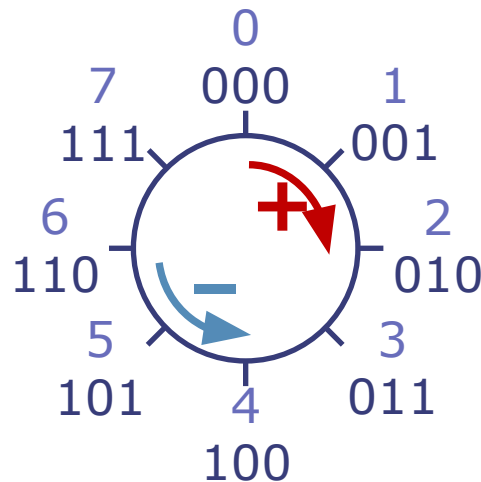
- **uint32_t**: 4 bytes (32 bits): 2^{32} values: 4,294,967,296 nums



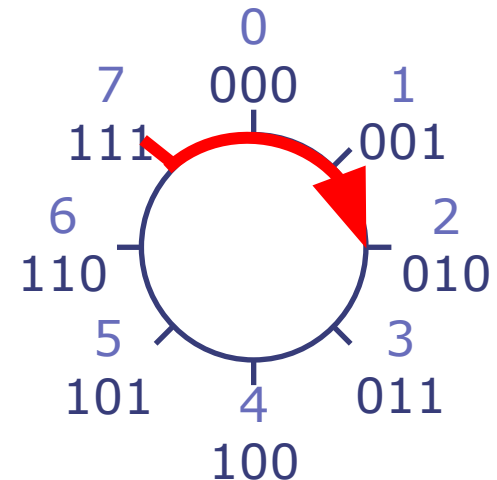
`uint32_t == "unsigned int"`

Inherent Modularity

- What happens when an operation produces a value that requires more bits than the type provides?
 - This is known as an **overflow**
- Common approach: Ignore the extra bit(s)
 - Gives rise to **modular arithmetic**: With N-bit numbers, equivalent to following all operations with **mod 2^N**
 - Visually, numbers “wrap around”:



Example: $(7 + 3) \bmod 2^3$?



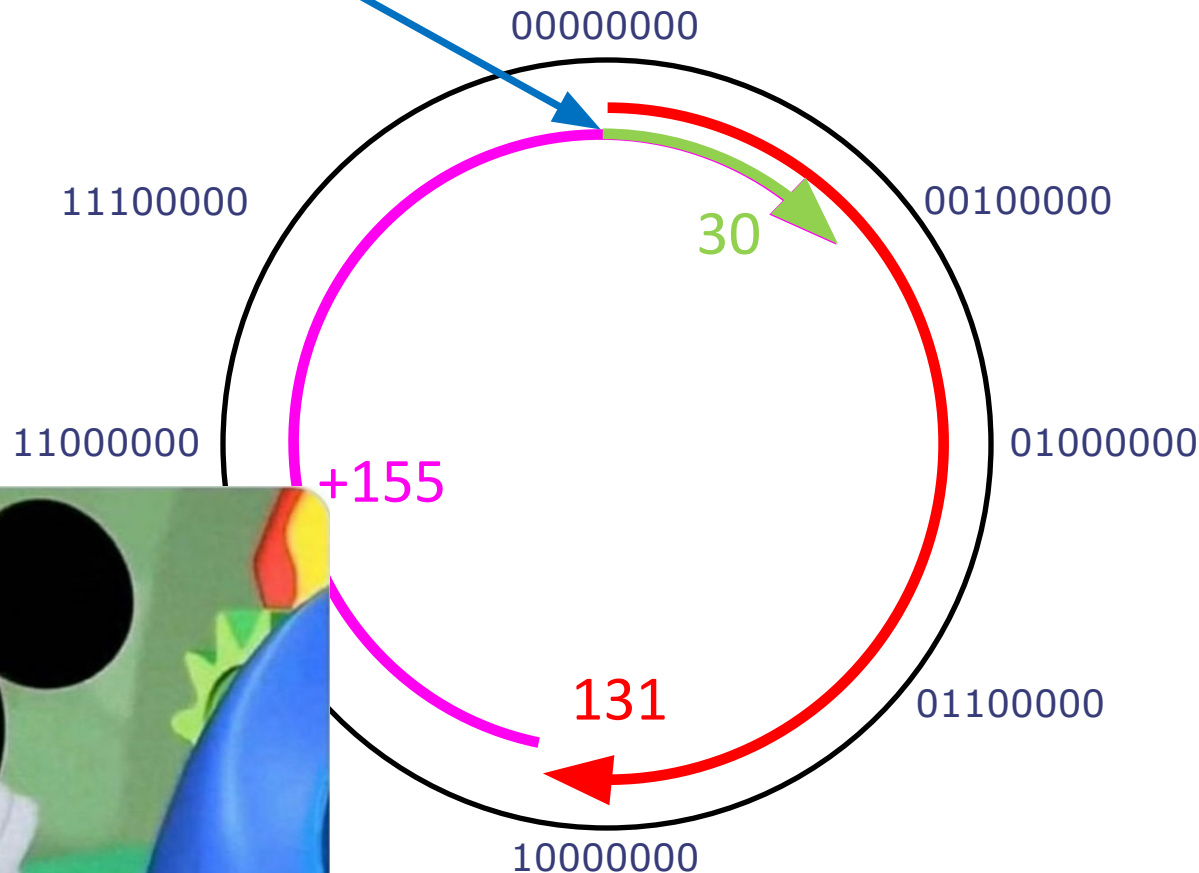
Happens with more bits too (8 bits)

- What happens if you add 131 to 155 with 8 bit `uint8_t`?

```
  10000011  (131)
+ 10011011  (155)
+-----
  100011110 (286)
  00011110  (30)
```

overflow

overflow



-It's a surprise tool that will help us later

What About Negatives?

- Our Number Schemes so far only allow representation of positive numbers (and zero).
- What about negatives? How can we do this in an efficient manner?

One Solution: "Sign Bit"

- If most-significant-bit (msb) is 1, interpret it like a negative sign:

- If 0, lower bits are from a positive number
- If 1, lower bits are from a negative number

- To get the negative of the number, flip the msb:

$$\begin{aligned} 0b00010001 &== +(16+1) == 17 \\ 0b10010001 &== -(16+1) == -17 \end{aligned}$$

- Major problem(s)?

- Signed zero (you can have +0 and -0)...big problem

$$\begin{aligned} 0b00000000 &== 0 \\ 0b10000000 &== -0 \end{aligned}$$

- Math tricky since one bit is no longer a number but a symbol

Another Solution: “One’s Complement”

- If most-significant-bit (msb) is 0, interpret it like an unsigned value.
- If msb is 1, then number is negative, else positive.
- To get the negative of the number flip all the bits (easy):

$$-A = \sim A$$

bitflip

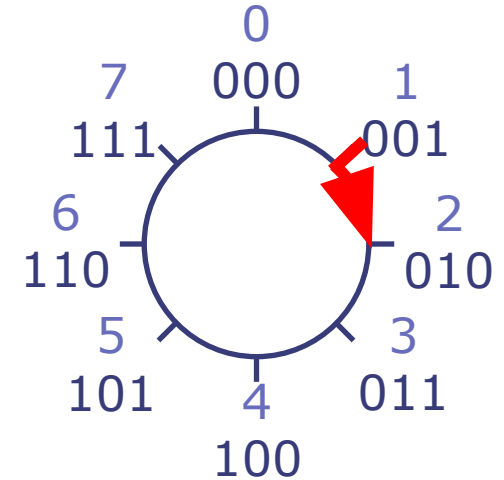
$$0b00010001 == +(16+1) == 17$$

$$0b11101110 == \text{bitflip of } 17 == -17$$

- Major problem(s)?
 - Signed zero (you can have +0 and -0)...big problem
$$0b00000000 == 0$$
$$0b11111111 == -0$$
 - Also addition and subtraction need this annoying wrap-around carry thing.

Inherent Modularity to the Rescue

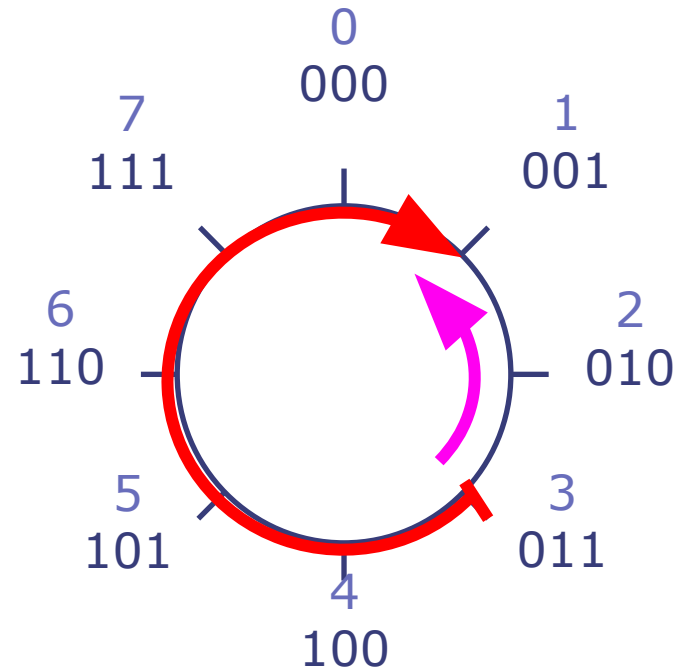
- Return to our 3-bit* number system:
- If I want to add 1, I just add 1 and move clockwise by 1 unit
- If I want to subtract 2, is there a number I could add to get the same result? If so, that number could be called “-2” right?



*3 bits here since easy to think about and draw, but could do with any number of bits

A Negative Number

- If I start at "3" aka 0b011, what could I add to get to 1?
- To go back 2, I can add:
 - $2^3 - 2 = 6$
- $(3+6)\%8 = 1$.
- Or: $\text{"-010"} = 110$



Negating a 3-bit Number

- The negative of a number A can be expressed as:

$$-A = 2^3 - A \qquad -A = 8 - A$$

- Or written a different way:

$$-A = 0b001 + 0b111 - A$$

- $0b111$ *minus* any 3 bit value will be the same as the bitflip aka bitwise inversion of that value ($\sim A$)

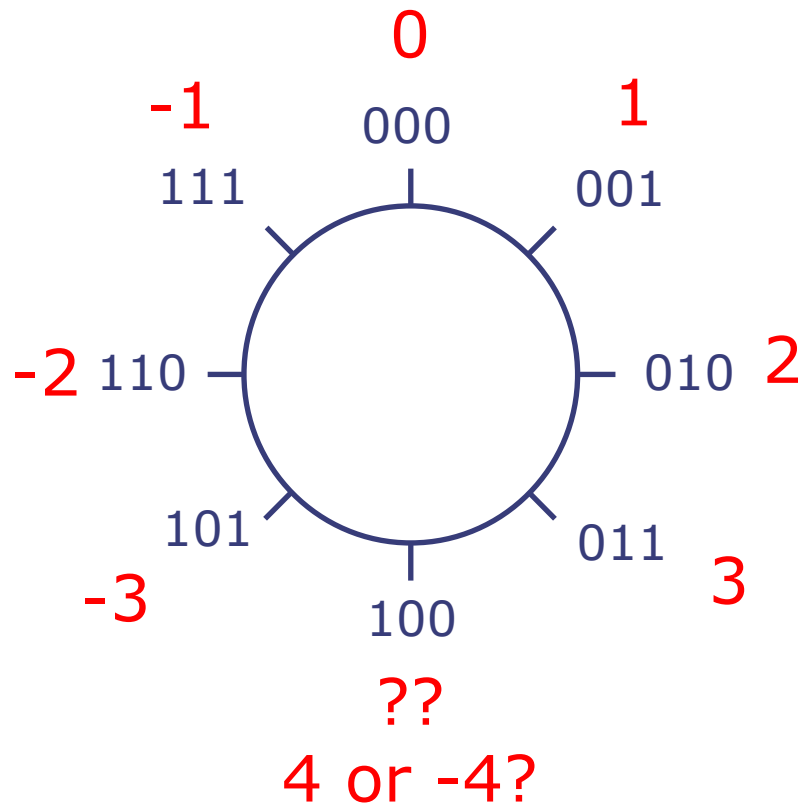
$$-A = 0b001 + (0b111 - A)$$

- So the negative of any value must be:

$$-A = 1 + \sim A$$

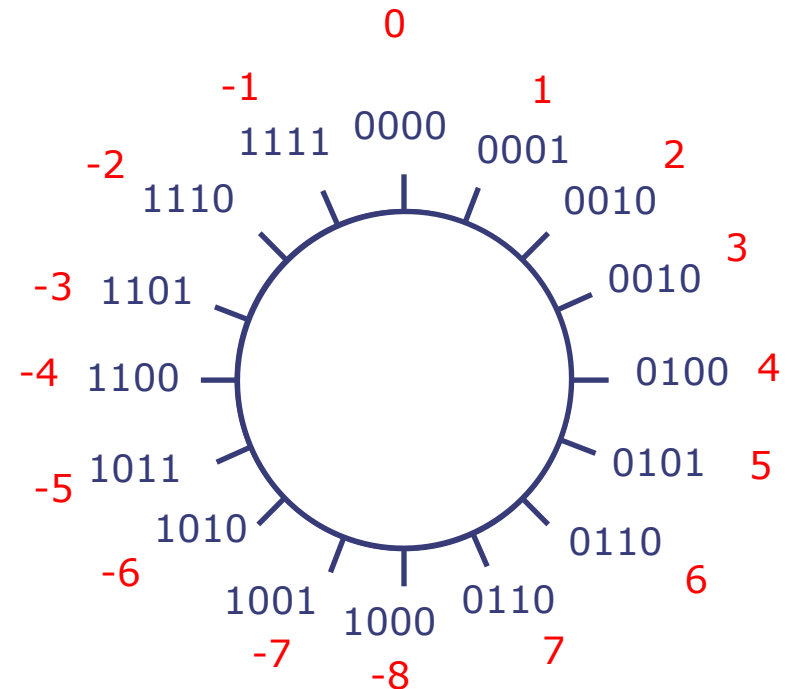
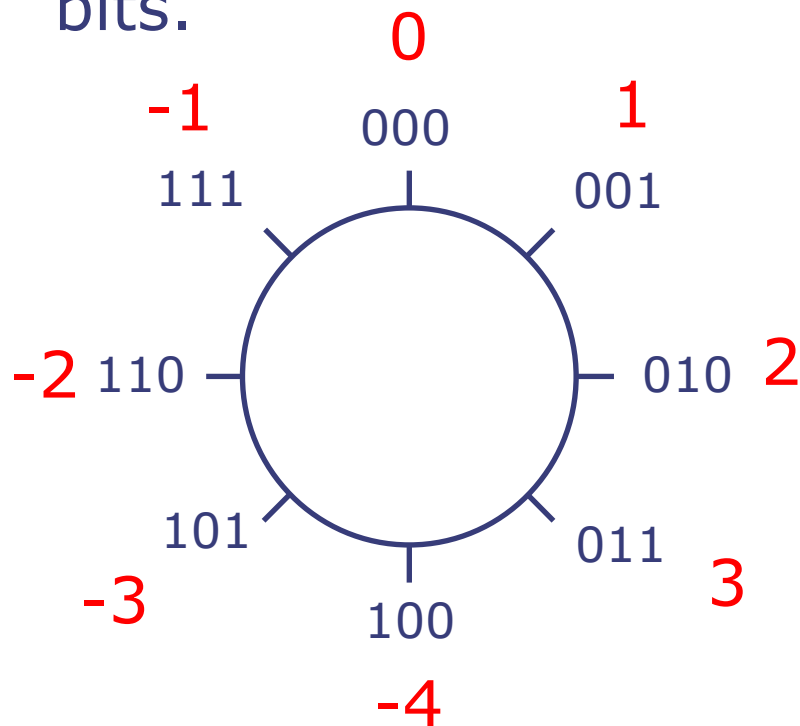
The Solution: 2's Complement

- For 000 to 111 what numbers do we get in this scheme?



Interesting...

- If we make 0b100 into -4, the system of numbers becomes consistent and easily extensible to more bits.



- With this model we can come up with some rules/observations...

Two's Complement Encoding

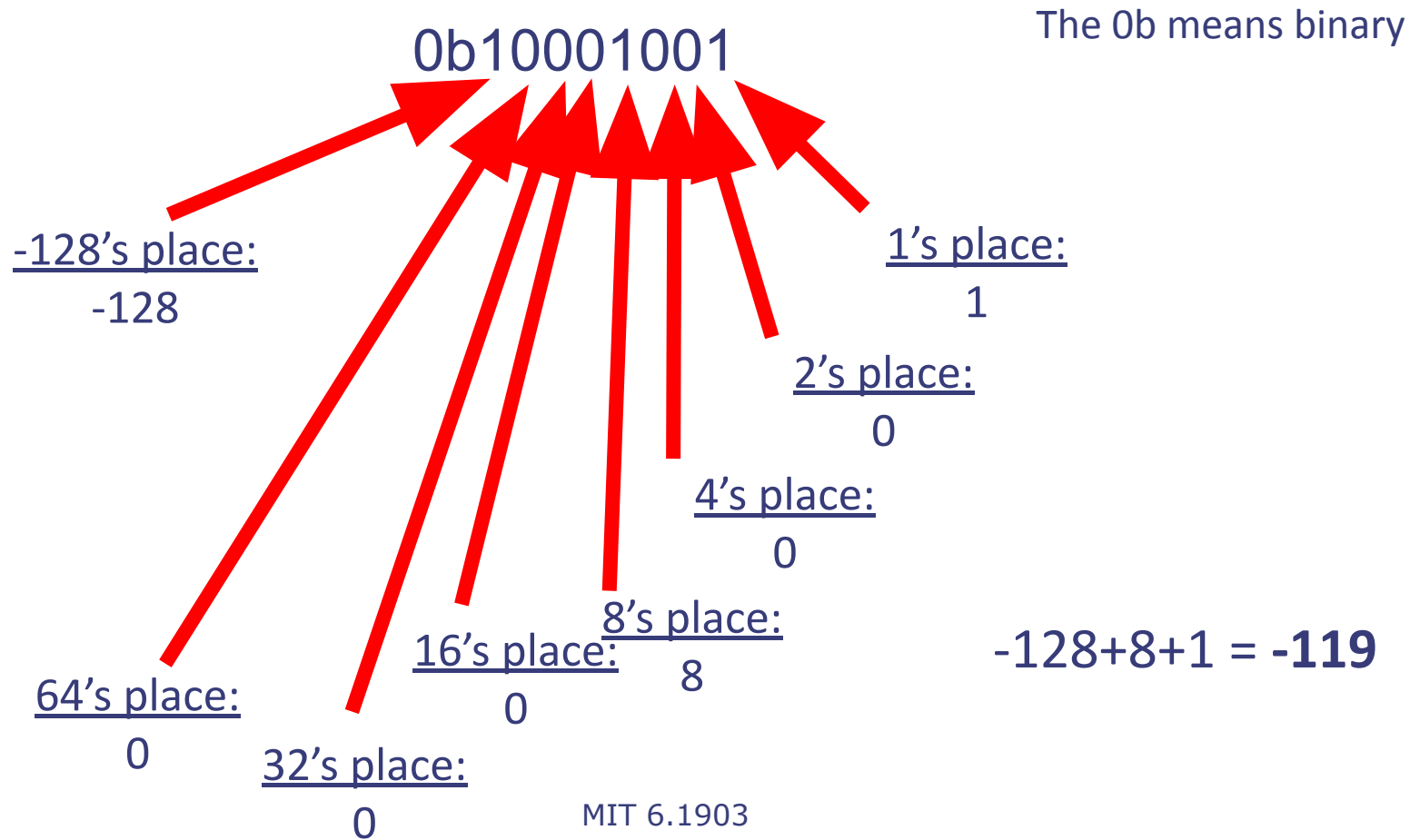
In two's complement encoding, the high-order bit of the N-bit representation has negative weight:

$$v = -2^{N-1}b_{N-1} + \sum_{i=0}^{N-2} 2^i b_i$$

- Negative numbers have "1" in the high-order bit
- *Most negative number?* 0b**1**0...0000 -2^{N-1}
- *Most positive number?* 0b**0**1...1111 $+2^{N-1} - 1$
- *If all bits are 1?* 0b11...1111 -1
- $-A = 1 + \sim A$

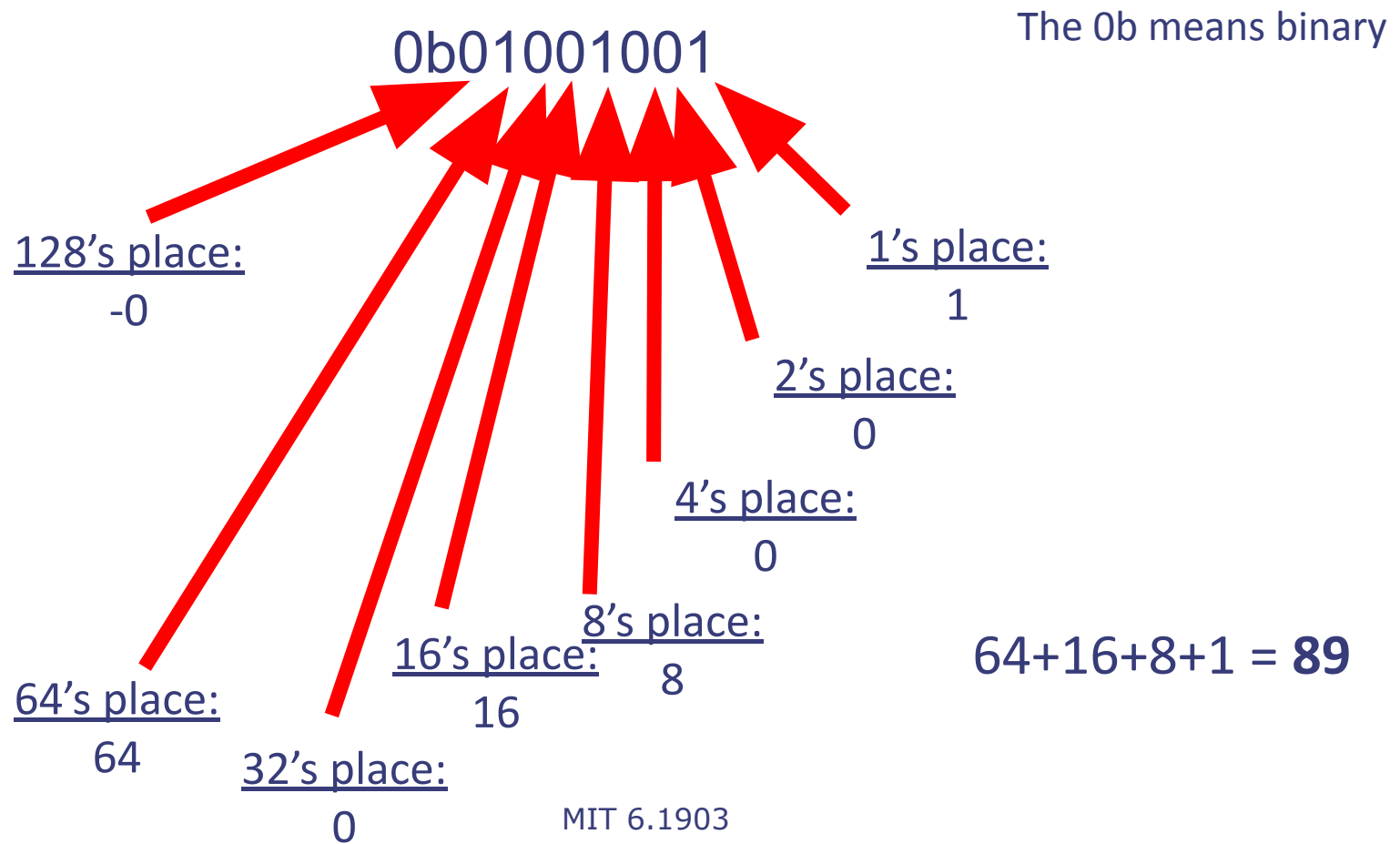
So you find a number on the sidewalk

- If you want to interpret it as a 2's complement



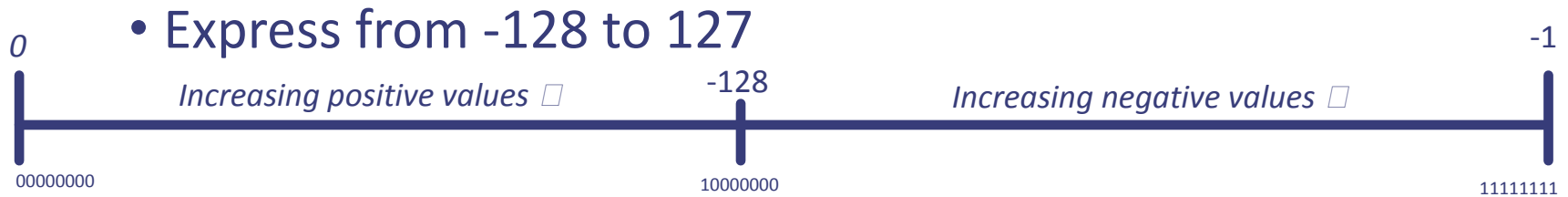
You find another number further down Vassar Street

- If you want to interpret it as a 2's complement

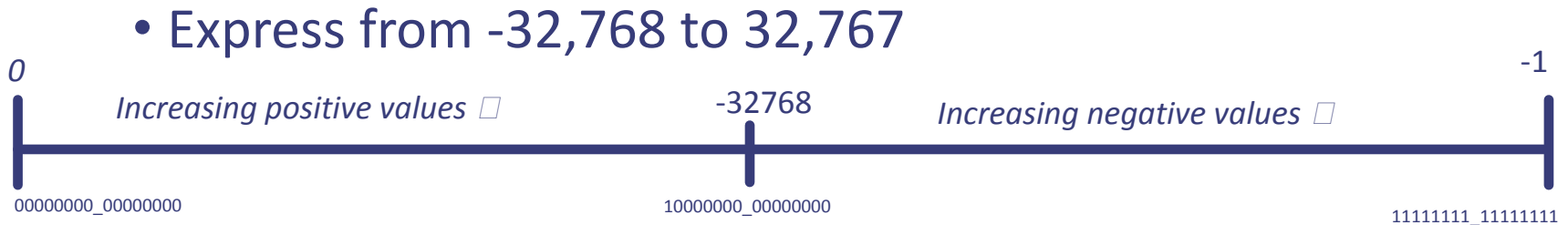


Signed Ints (ints):

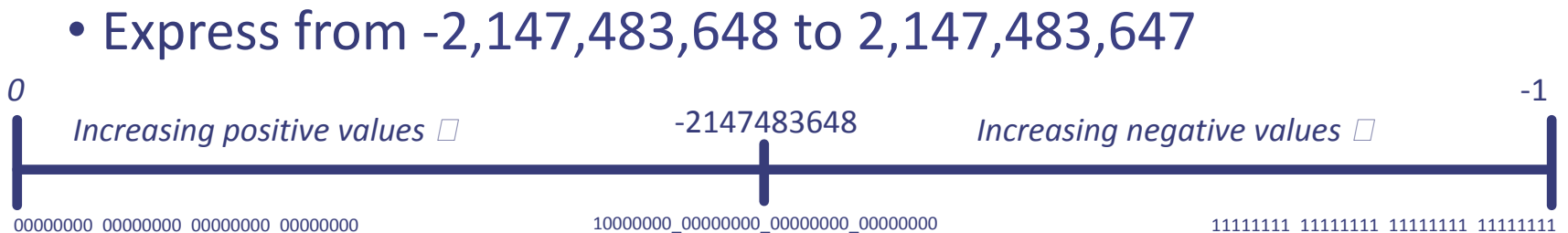
- `int8_t`: 1 byte (8 bits): 2^8 values: 256 numbers to rep



- `int16_t`: 2 bytes (16 bits): 2^{16} values: 65,536 numbers



- `int32_t`: 4 bytes (32 bits): 2^{32} values: 4,294,967,296 nums



`int32_t == "int"`

Extending Digits

- The sign of a number can be ascertained just by looking the most significant bit!
- In unsigned integer if you want to make it take more bits, you just tack 0's on front
- For signed integer you need to *sign extend*...which means replicate the leading bit

8 bit "-64" = 0b11000000

16 bit "-64" = 0b11111111_11000000

Math Operations Still Work

- Two's Complement is pretty nice because you can still do all your regular math operations pretty easily and it just works! (multiply and divide beyond scope of class)
- Also No double-zero!
- Pretty much all modern digital systems use two's complement math to represent signed integers

Shift Left

- Consider two signed numbers: +32 and -32 as 8 bit signed values:

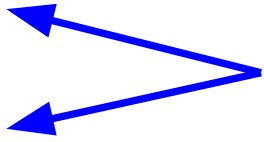
$$+32 = 0b00100000$$

$$-32 = 0b11100000$$

- If we left shift (<<) what do we get?

$$+64 = 0b01000000$$

$$-64 = 0b11000000$$



New digits that
show up are
zeros/same

Shift Right

- Consider two signed numbers: +32 and -32 as 8 bit signed values:

$$+32 = 0b00100000$$

$$-32 = 0b11100000$$

- If we right-shift ($\gg$) what do we get?

$$+16 = 0b00010000$$

$$-16 = 0b11110000$$



New digits that show
up are different

Logical vs. Arithmetic Shift

- Shift operators will base what they do on if the variable/type is signed vs unsigned:
 - Signed value? $\gg$ is *arithmetic* shift:
 - Means leading digit is replicated
 - Unsigned value? $\gg$ is *logical* shift:
 - Means leading digit(s) gets set to 0
 - For $\ll$, in both cases, 0 is used for filler so it is *logical*

Why still have unsigned ints?

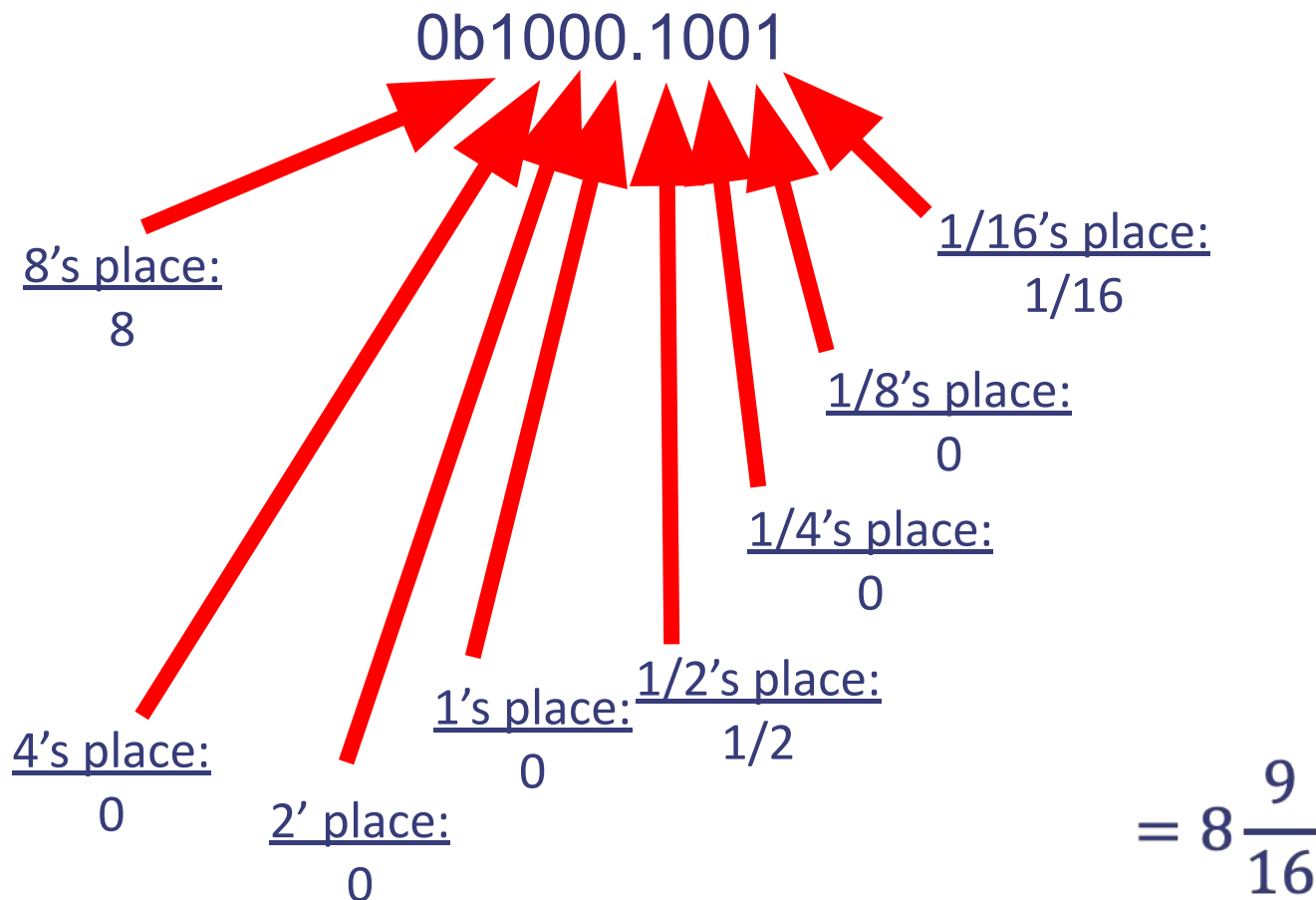
- If Two's Complement is so amazing why do we still have unsigned integers?
- An `uint8_t` and an `int8_t` both represent 256 things, but different sets of things:
 - `uint8_t`: 0 to +255
 - `int8_t`: -128 to +127
- If I know my data ranges from 0 to 140, it would be a waste to have to use 16 bits because I have to use only signed integers. `uint8_t` is a better choice for some things so keep them around.

What about Non-Integer Numbers?

- Can only describe integers so far.
- That's not all numbers.
- Need some ability to represent numbers with decimals.
 - Attempt #1: Fixed point

Fixed Point Representation

- Pretend there's a radix marker (".") at a fixed location in your binary...then reinterpret meaning of bits.



Fixed Point

- Everything works out of the box (add, subtract, multiply, divide, 2's complement negatives)...just like when you learned decimals point math as a child*
- However!
 - Precision is limited!
 - Range is limited!
 - The more digits to the right of the radix marker, The fewer you have to left for a given number of bits.

0b1000.1001

*remember from elementary school you could kinda just line up the "." then forget it...there'd be a few special case rules, but basically the math worked out totally fine.

An Exponential Format

- If you needed to encode something that is best represented by orders of magnitude?
- Have an 8-bit type called `exp8_t` where the 8 bits represent a unsigned 8 bit number **exp** which is used in the following encoding formula*:

$$\text{Value} = 2^{\text{exp}-127}$$

Base-two exponentials are easy to calculate in binary!

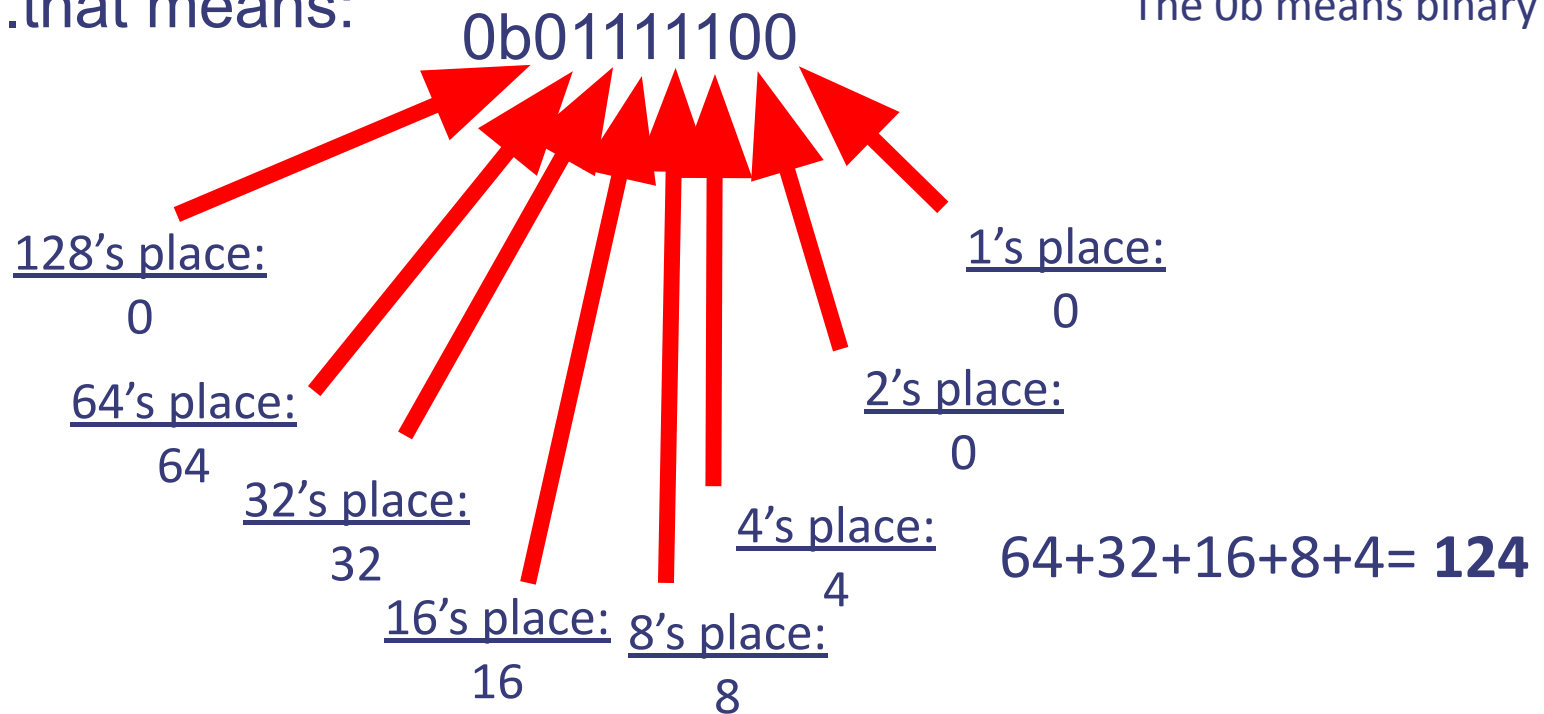
- Represent from:
 - Up to: $2^{255-127} = 2^{128} \approx 3.403 \times 10^{38}$
 - Down to: $2^{0-127} = 2^{-127} \approx 5.877 \times 10^{-39}$

**made this up but I'm going somewhere with this so bear with me...*

Example

- So if you had an `exp8_t` with `0b01111100` as its value...that means:

The 0b means binary



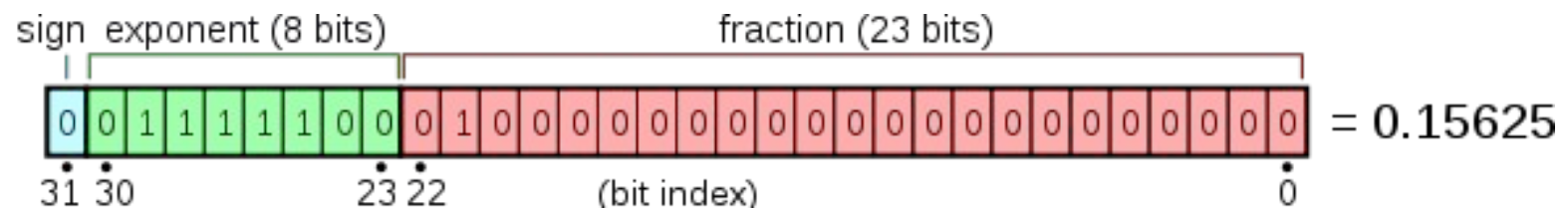
- So actual = $2^{124-127} = 2^{-3} = 0.125$

Exponential Conclusion

- Allows you to encode a massive *range* of numbers, which can be nice for some things.
- The precisions of the numbers, is somewhat limited...there's really only one number for each order of magnitude.
- It sure would be nice to have range and some precision/wiggle-room. Could we do more?
- Yes. Yes we could.

IEEE 754 Floating Point Representation

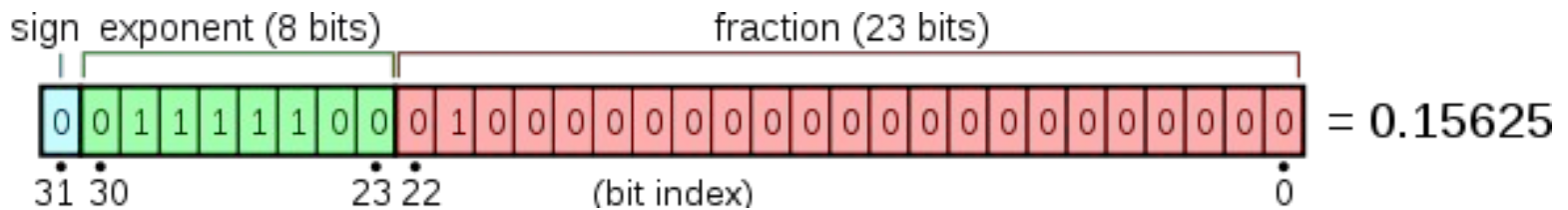
- A Scientific notation approach to number representation
- float: 4 bytes (32 bits): 2^{32} values: 4,294,967,296
 - Express from approx. -3.4×10^{38} to $+3.4 \times 10^{38}$
 - Also express small numbers...down to 1.17×10^{-38}
- Decimal point is “floating” as a result (as opposed to fixed)



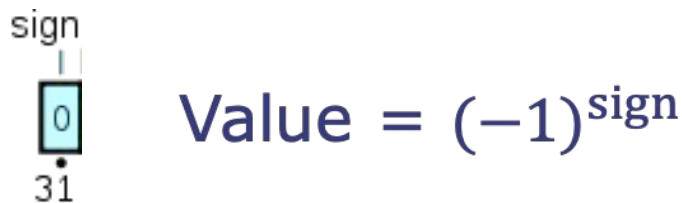
$$\text{Value} = (-1)^{\text{sign}} \cdot 2^{\text{exp}-127} \cdot \left(1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}\right)$$

IEEE 754 is Not Scary

- Just break it down!



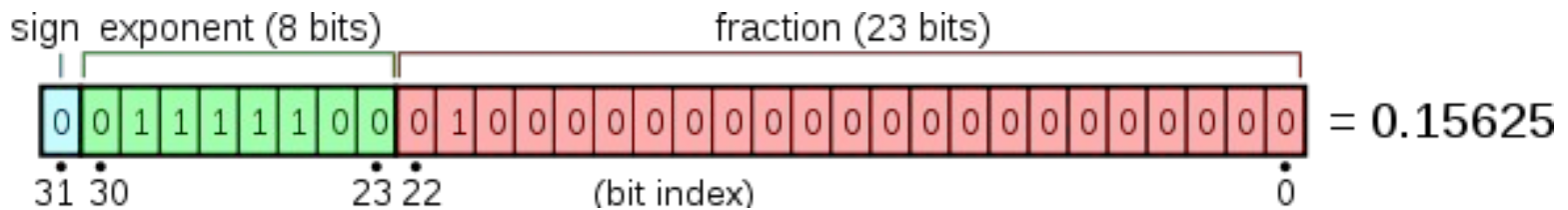
- First up we have a sign bit!



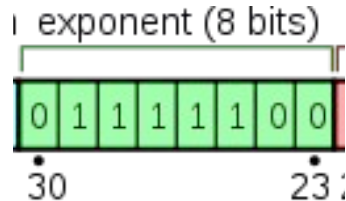
- So value is +1 if sign=0 and -1 if sign = 1

IEEE 754 is Not Scary

- Just break it down!



- Next we have exponent byte

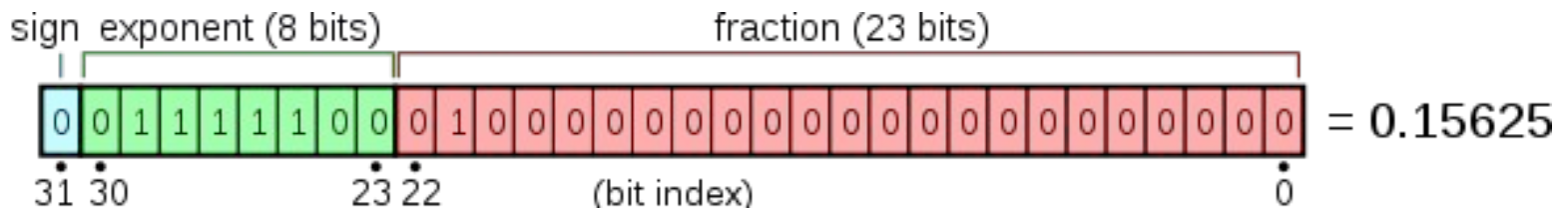


$$\text{Value} = 2^{\text{exp}-127}$$

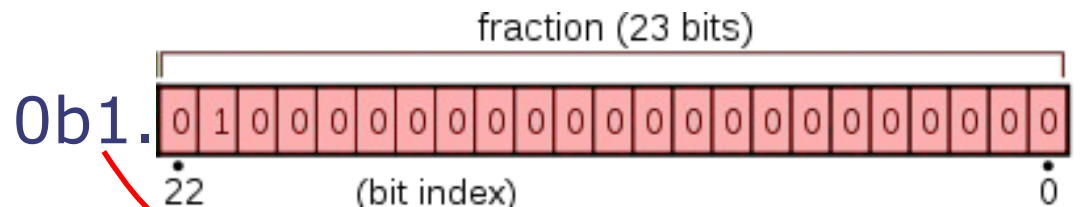
- Already did this a few slides ago!...
- Represent 256 scaling terms exponentially spaced from:
 - Up to: $2^{255-127} = 2^{128} \approx 3.403 \times 10^{38}$
 - Down to: $2^{0-127} = 2^{-127} \approx 5.877 \times 10^{-39}$

IEEE 754 is Not Scary

- Just break it down!



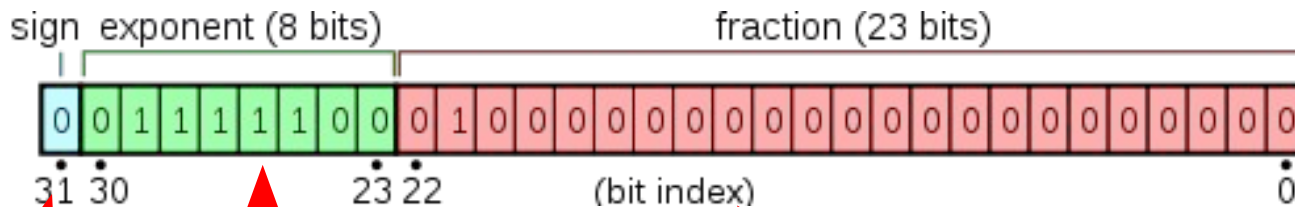
- Finally we have a **24 bit fixed-point number** (23 bits of which are specifiable)



- Eq is Value = $1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}$ *Implied 1 always in the 1's place*

All together

$$\text{Value} = (-1)^{\text{sign}} \cdot 2^{\text{exp}-127} \cdot \left(1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}\right)$$



Sign bit
+1

Exponent*:
1/8 (0.125)

*already did earlier

Mantissa:
 $1 + 2^{-1} \cdot 0 + 2^{-2} \cdot 1 + 0 \dots = 1.25$

ALL TOGETHER:

$$= +1 \cdot \frac{1}{8} \cdot \frac{5}{4} = \frac{5}{32} = 0.15625$$

How do I do math with Floats?

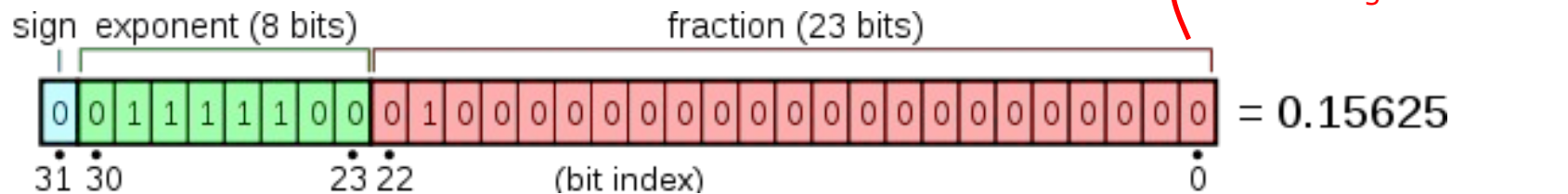
- You can't just add them like we did with integers.
- You need more complicated algorithms and circuits....
 - Add the exponents
 - Multiply the mantissas,
 - Scale/merge their results
 - Do if/else logic check on signs
- Floating point math is expensive!
- **Don't worry about it in this class! Just worry about the encoding!**

Now...Remember about Bits!

- A 32 bit float does NOT let us represent more numbers. It still only let's us represents 2^{32} unique values.
- float: 4 bytes (32 bits): 2^{32} values:
4,294,967,296
 - Express from approx. -3.4×10^{38} to $+3.4 \times 10^{38}$
 - Also express small numbers...down to 1.17×10^{-38}
 -
- What a 32 bit float does let us represent is a far wider range of numbers compared to an integer or a fixed point number.
- Floating Point is a *compromise*. We sacrifice precision for range

IEEE 754: Floating Point Precision

- The amount of “change” caused by toggling the lsb of the fraction part (“mantissa”) is also affected by what the exponent value is



- Starting at 0.15625, the “next” value expressible is: 0.156250014901, a jump of 1.4901×10^{-8}
- Starting at $5.31691198314 \times 10^{37}$, the next value expressible is $5.3169124902 \times 10^{37}$, a jump of 5.0706×10^{30} !!!!

Floating point can have massive massive “gaps”!

Floating Point Use is Approximation Based!

- 32 Bit floats represent 2^{32} (4.3×10^9) values between -3.4×10^{38} to $+3.4 \times 10^{38}$
- There is NO WAY to represent every single number between these to limits with only 32 bits.
- There is NO WAY to represent *even* every integer between those limits

```
#include<stdio.h>
int main(void){
    float x = 123456789;
    float y = 123456788;
    printf("The value of x is %f\n",x);
    printf("The value of y is %f\n",y);
}
```

Output:

```
The value of x is 123456792.000000
The value of y is 123456784.000000
```

Floating Point Can be Problematic

- Exponential representation means:
 - Merging (+/-, etc...) two numbers *of vastly different scales* can result in the smaller one getting consumed by the larger one and lost

```
int main(void){  
    float x = 1.35e9;    // create float that is "1.35 billion"  
    float y = 23.0;      // create float that is "23.0"  
    float sum = x+y;     // sum them  
    float sum_wo_x = sum - x; // remove x again  
    printf("x is %f\ny is %f\nx+y is %f\nx+y-x is %f\n",x,y,sum,sum_wo_x);  
}
```

Output:

```
x is 1350000000.000000  
y is 23.000000  
x+y is 1350000000.000000  
x+y-x is 0.000000
```

Floating Point Can be Problematic

- A float is usually an approximation of the number we want

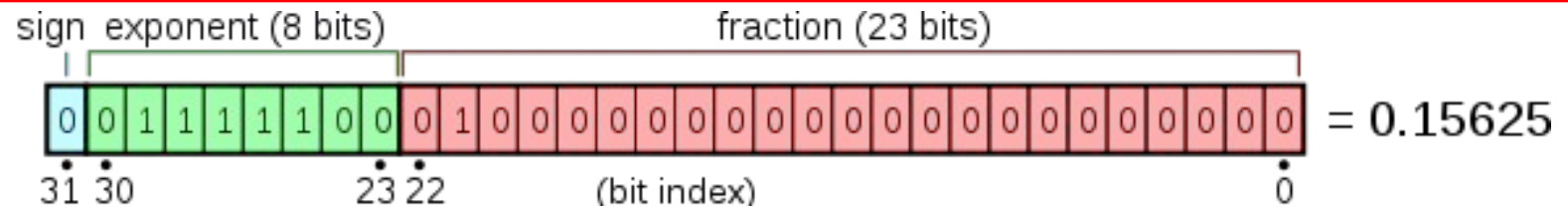
```
int main(void){
    float x = 1e-4; // create float that "is" 0.0001
    float sum = 0.0; // create float that is "0.0"
    float expected_sum = 1.0;
    for (int i=0; i<10000; i++){
        sum +=x;
    }
    printf("Adding %f 10000 times totals %f\n",x,sum);
    printf("%d\n",sum==expected_sum);
}
```

Output:

```
Adding 0.000100 10000 times totals 1.000054
```

NEVER use floats to represent money. EVER

Float has a Double Zero!



$$\text{Value} = (-1)^{\text{sign}} \cdot 2^{\text{exp}-127} \cdot \left(1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}\right)$$

- Is that a problem?
- Actually not necessarily, this can be beneficial in doing things like limits and so on

Old News, Right?

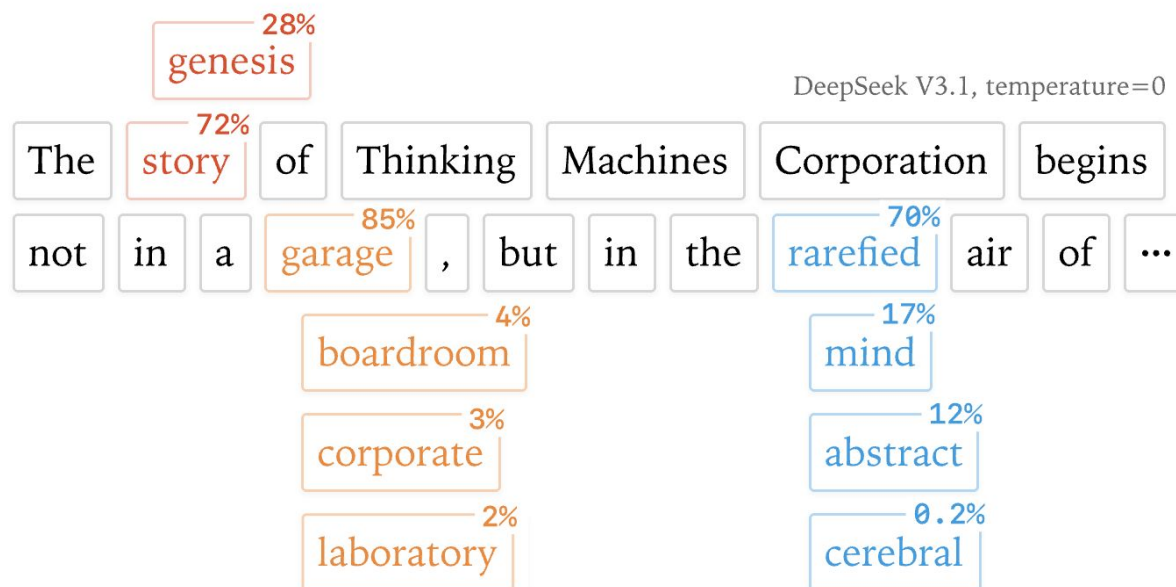
- Pssshh Floating point is old. I'm a new kid. I'm an ML baby. Digital Native. I Chat with GPT all the time. I only like new things. This is irrelevant and not connected to my Gen Z existence in the year 2026.
- But wait...

Floating Point Can be Problematic

Defeating Nondeterminism in LLM Inference

Horace He in collaboration with others at Thinking Machines

Sep 10, 2025



The FP8 debate

- The last ~five years have been a very exciting time for data representation.
- In the ML field, we're building massive systems with billions of weights.
- Those weights need to be stored and used.
- The fewer bits we need to represent those weights, the more we can store.
- Also the fewer bits we need, the faster we can compute.
- General consensus is a 32 bit float is overkill in a lot of applications.
- Do we use ints? Or smaller floats? Or something else?

So there's been a lot of research in formats

- Finding the best mini-float representation for ML:
 - Various types of 8-bit floating points (FP8)

 **Untether AI FP8 – Designed for Inference Acceleration**

AI Inference requires precision – but full floating point is expensive to achieve needed accuracy targets

Untether AI researched various datatype and found FP8 provided the best balance of precision, throughput, and energy efficiency

- But requires both range (FP8r) and precision (FP8p)
- Each additional mantissa doubles the precision, and reduces mean squared error by $(1/2)^2$

FP8 is 2x times more energy/die size efficient than if designed for native INT8

UAI FP8 precision	S	E	E	E	M	M	M	M
UAI FP8 range	S	E	E	E	E	M	M	M
	S	E	E	E	E	E	M	M

Training FP8 precision

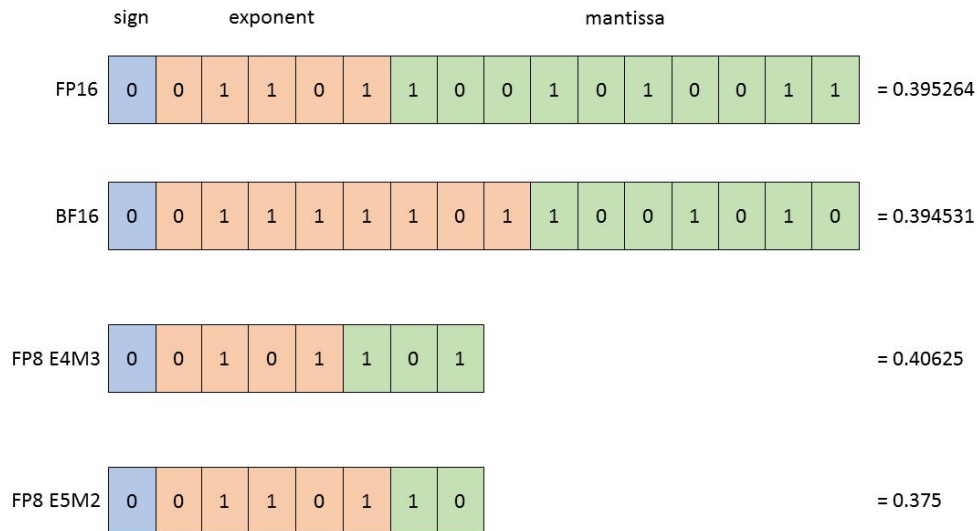
Training FP8 range

UNTETHER AI Copyright © 2022 UNTETHER AI Corp.

<https://www.servethehome.com/untether-ai-boqueria-1458-risc-v-core-ai-accelerator-hc34/hc34-untether-ai-boqueria-fp8-for-ai-inference/>

Different Variations!

- More bits on mantissa? More on Exponent? More/less overall?
- NVIDIA really likes E4M3 and E5M2 formats



- Field is gradually coalescing on things

https://docs.nvidia.com/deeplearning/transformer-engine/user-guide/examples/fp8_primer.html

Different Variations!

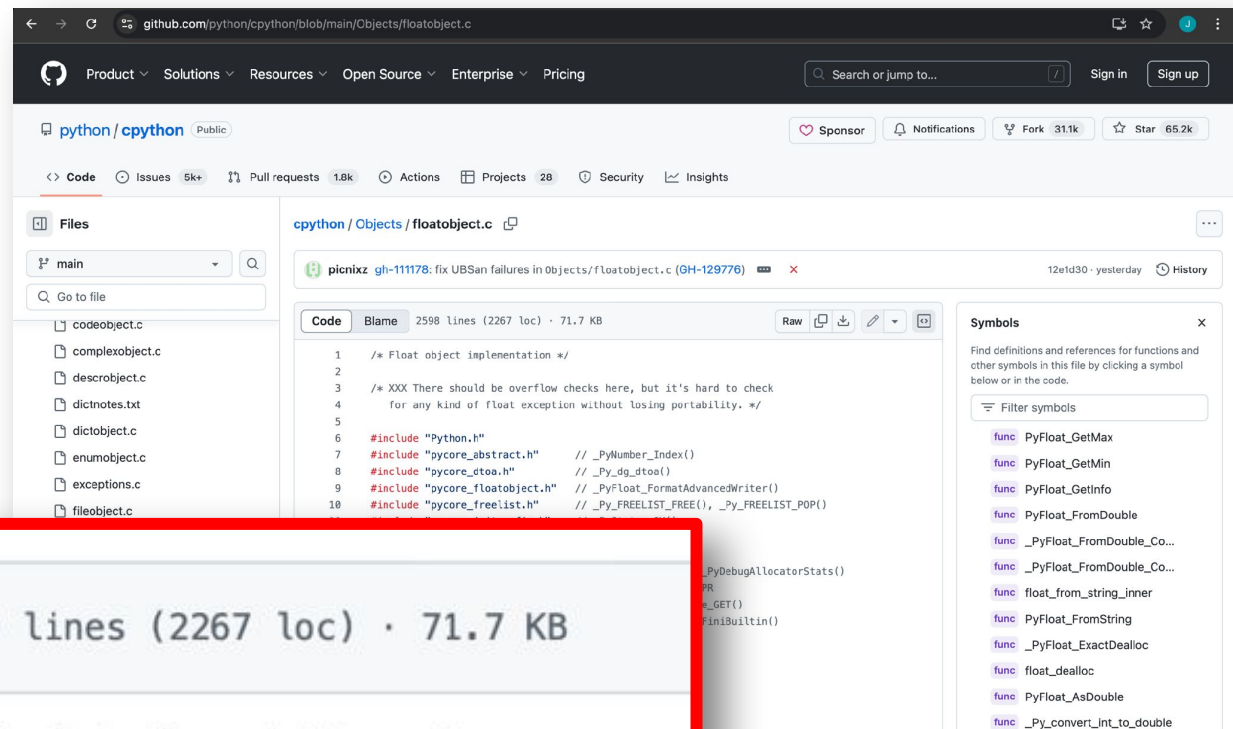
Architecture	A100	H100	H200	GH200	B100	B200	Full B200	GB200
	Ampere	Hopper			Blackwell			
VRAM Size	80GB	80GB	141GB	96GB/144GB	180/192GB	180/192GB	192GB	384GB
VRAM Bandwidth	2TB/s	3.35TB/s	4.8TB/s	4TB/s / 4.9TB/s	8TB/s	8TB/s	8TB/s	16TB/s
FP16 (FLOPS)	312T	1P	1P	1P	1.75P	2.25P	2.5P	5P
INT8 (OPS)	624T	2P	2P	2P	3.5P	4.5P	5P	10P
FP8 (FLOPS)	X	2P	2P	2P	3.5P	4.5P	5P	10P
FP6 (FLOPS)	X	X	X	X	3.5P	4.5P	5P	10P
FP4 (FLOPS)	X	X	X	X	7P	9P	10P	20P
NVLink Bandwidth	600GB/s	900GB/s	900GB/s	900GB/s	1.8TB/s	1.8TB/s	1.8TB/s	3.6TB/s
Power Consumption	400w	700w	700w	1000w	700w	1000w	1200w	2700w
Notes	Die*1	Die*1	Die*1	Grace CPU*1 H200 GPU*1	Die*2	Die*2	Die*2	Grace CPU*1 Blackwell GPU*2

Also...Python's "Float"

- As more advanced languages were developed, the ability to represent wide-ranging numbers more effectively was a priority.
- In using Python's "Float" you're still using the same underlying data type/binary representation.
- But it has a lot more supporting infrastructure built around that "object" to protect you from the limitations.

Python's Float

- Python's "Float" object is described in its floatobject.c file



<https://github.com/python/cpython/blob/main/Objects/floatobject.c>