
32 Bits
Sign
Exponent
Mantissa
1 Bit
8 Bits
23 Bits
Single Precision
IEEE 754 Floating-Point Standard
Look what they need to mimic a fraction

TODO

- Assignment 1 is assigned (due September 24)

Overview for today

- Finishing up floating point
- Strings

```
public class Loop {
    public static void main(String[] args) {
        for (double i = 0; i != 1; i += 0.1) {
            System.out.println(i);
        }
    }
}
```

Floating point is dangerous

Alternatives

- Fixed point
  - $1343.77 → 134377 cents
- Rational numbers
  - Keep track of numerator and denominator

Rationals in Python

```
>>> from fractions import Fraction
>>> Fraction(1, 10) + Fraction(1, 10) + Fraction(1, 10)
Fraction(3, 10)
```

Rationals downside

- 47985420975943853245439857324095742390875942307508943750984237509874320859742309857340289573249365921749023748239017408923740892374098237104982310894723091748390214732912349823190 47 / 23978
- Some numbers cannot be represented as rationals
  - Transcendental: π
  - Irrational: √2

How do we fix this?

```
public class Loop {
    public static void main(String[] args) {
        for (double i = 0; i != 1; i += 0.1) {
            System.out.println(i);
        }
    }
}
```

Not all hardware supports FP

- Specialized hardware: FPU
- Included in all consumer desktops/laptops these days
- In 90s, an optional expansion coprocessor
- Embedded processors often don’t have an FPU


intel®
i387™SX™
N80387SX
(16-25MHZ)
C2200508 A1
Ⓜ© 1986

No FPU?

- Fixed point (easy to represent as an int)
- C compilers can often emulate floating point on non-FPU-capable CPUs
  - Think the kind of bit stuff we are doing in Lab 1!
  - But the kind of hardware that doesn’t have an FPU is already on the slow side…


50
AMMO
100%
HEALTH
2 3 4
5 6 7
ARMS
0%
ARMOR
BULL 50 / 200
SHEL 0 / 50
RCKT 0 / 50
CELL 0 / 300

Structure of course

- First third:
  - Data Representation ← here
  - Memory
- Second third (Exam 2):
  - Machine Code
  - Memory hierarchy
- Final third (Final exam):
  - Processes
  - Other miscellaneous topics

Data Representation roadmap

- ~~Integers~~
- ~~Floating point numbers~~
- Strings ← here

Strings

- Not as closely tied to hardware
- Represented differently in different programming languages / files

C strings

- char: a byte (8 bits)
- Why is it called a char?
- Because it is also a character!

C chars

```c
#include <stdio.h>

int main(void) {
    char c;
    c = 'h'; printf("%c\n", c);
    c = 'e'; printf("%c\n", c);
    c = 'l'; printf("%c\n", c);
    c = 'l'; printf("%c\n", c);
    c = 'o'; printf("%c\n", c);
    return 0;
}
```

C chars

```c
#include <stdio.h>

int main(void) {
    char c;
    c = 0x68; printf("%c\n", c);
    c = 0x65; printf("%c\n", c);
    c = 0x6c; printf("%c\n", c);
    c = 0x6c; printf("%c\n", c);
    c = 0x6f; printf("%c\n", c);
    return 0;
}
```

Chars ↔ integer values

- `'h'` is syntactic sugar for `0x68`
- The C compiler automatically maps char literals to their equivalent integer values

ASCII Table

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(	72	48	110	H	104	68	150	h
9	9	11		41	29	51	)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[	123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	|
29	1D	35		61	3D	75	=	93	5D	135	]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

ASCII

- “American Standard Code for Information Interchange”
- Maps numbers 0 through 127 to characters
- (High order bit unused)

C strings

```c
#include <stdio.h>

int main(void) {
    char s[] = "hello";
    printf("%s\n", s);
    return 0;
}
```

C strings

```c
#include <stdio.h>

int main(void) {
    char s[] = {'h', 'e', 'l', 'l', 'o', '\0'};
    printf("%s\n", s);
    return 0;
}
```

C strings

```c
#include <stdio.h>

int main(void) {
    char s[] = {0x68, 0x65, 0x6c, 0x6c, 0x6f, 0x00};
    printf("%s\n", s);
    return 0;
}
```

Null terminator

- A string is an array of chars
- But arrays in C don’t keep track of their length
- So we need a way for the computer to know when the string ends

What is missing?

- Are there characters missing from ASCII?

ASCII Table

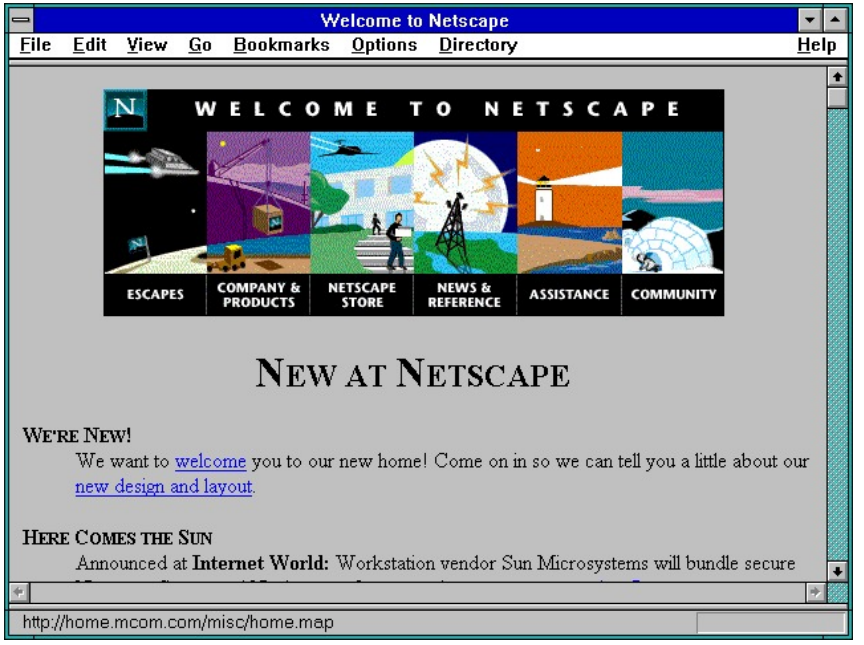
Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(	72	48	110	H	104	68	150	h
9	9	11		41	29	51	)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[	123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	|
29	1D	35		61	3D	75	=	93	5D	135	]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

Internationalization of strings

- 1980s:
  - Unicode consortium founded
  - Consortium believed that 16 bits was enough to hold every commonly used language in the world
- 1993:
  - Unicode: map characters to numbers
  - UCS-2: mapped characters to 2-byte encodings

Internationalization of strings

- 1990s:
  - The Internet happened
  - More “commonly used languages” than anticipated
  - 16 bits not enough!
  - Unicode expanded beyond



Modern language encodings

- ASCII – 8 bit encoding for English
- UTF-8 – maps a Unicode character to 1–4 bytes
- UTF-16 – maps a Unicode character to 1–2 16-bit ints
- UTF-32 – maps a Unicode character to a 32-bit int

Unicode vs. UTF-*

- Unicode decides how to map characters to ints
- UTF-8, -16, and -32 say how to encode those ints in one or more 8/16/32 bit ints

Encoding examples

Encoding	hello	你好	👋
ASCII	68 65 6C 6C 6F	–	–
UTF8	68 65 6C 6C 6F	E4 BD A0 E5 A5 BD	F0 9F 91 8B
UTF16LE	68 00 65 00 6C 00 6C 00 6F 00	60 4F 7D 59	3D D8 4B DC
UTF32LE	68 00 00 00 65 00 00 00 6C 00 00 00 6C 00 00 00 6F 00 00 00	60 4F 00 00 7D 59 00 00	4B F4 01 00
GBK	68 65 6C 6C 6F	C4 E3 BA C3	–

Pros/cons

- ASCII
  - Pro: length in bytes is length in characters
  - Con: English only
- UTF-8
  - Pro: backwards compatible with ASCII
  - Con: length in bytes is not length in characters

Pros/cons

- UTF-16
  - Pro: none really
  - Con: length in bytes is not length in chars
  - Con: not backwards compatible with ASCII
- UTF-32
  - Pro: length in 32-bit ints is length in characters
  - Con: not backwards compatible with ASCII

Example environments

- UTF-8: C (Unix), Rust
- UTF-16: C (Windows), Java, Javascript
- UTF-32: Python

Warning: platform dependent

```c
#include <stdio.h>

int main(void) {
    char s[] = "hello👋";
    printf("%s\n", s);
    return 0;
}
```

Warning: platform dependent

```c
#include <stdio.h>

int main(void) {
    char s[] = {'h', 'e', 'l', 'l', 'o', 0xf0, 0x9f, 0x91, 0x8b, '\0'};
    printf("%s\n", s);
    return 0;
}
```

Warning: platform dependent

```c
#include <stdio.h>

int main(void) {
    char s[] = {0x68, 0x65, 0x6c, 0x6c, 0x6f, 0xf0, 0x9f, 0x91, 0x8b, 0x00};
    printf("%s\n", s);
    return 0;
}
```

Why Windows, Java, Javascript…

- … use UTF-16?
- They historically used UCS-2
- They switched to UTF-16 for backwards compatibility

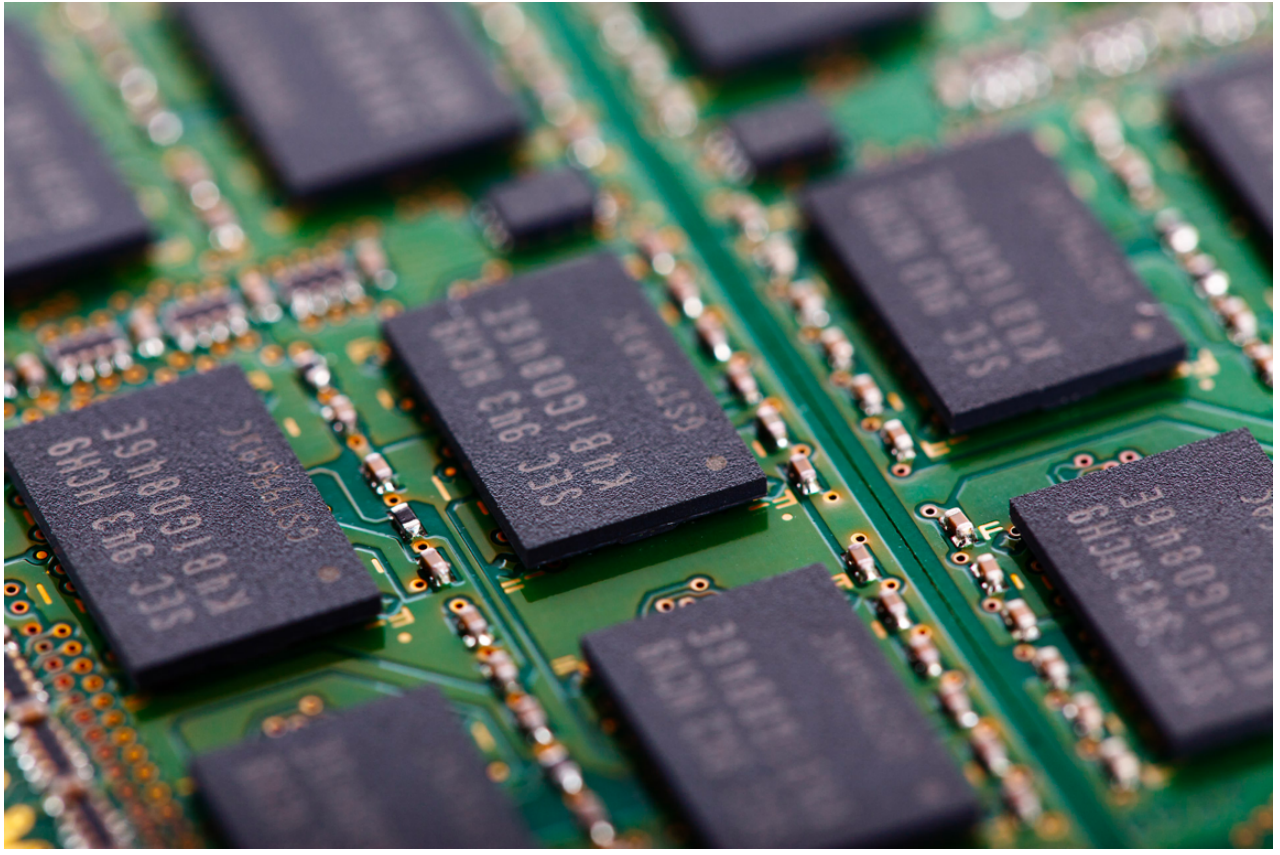
Data Representation roadmap

- ~~Integers~~
- ~~Floating point numbers~~
- ~~Strings~~

Structure of course

- First third:
  - Data Representation
  - Memory ← here
- Second third (Exam 2):
  - Machine Code
  - Memory hierarchy
- Final third (Final exam):
  - Processes
  - Other miscellaneous topics

Memory



Memory

- You’ve used memory
- But you’ve never *managed* it
- In C, we will get to finally do this!

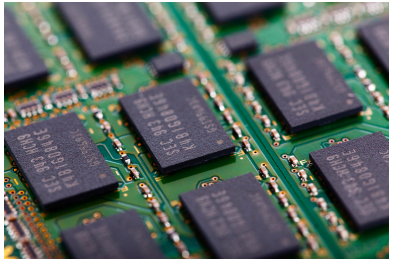
Manual memory management

- Pro: can write faster programs
- Con: can write horribly broken programs
- We will see examples of each!

Memory

- We will spend the next module learning about memory from a **systems** perspective
- And a **C** perspective

What is memory?



- Implemented in hardware by RAM
- Effectively an array of bytes
- The index into this array is the memory *address*
- E.g., if we have 256 bytes of memory:

Content:	0xFF	0x00	0x57	0x92	0xB3	0x8A	…	0xDC
Address:	0x00	0x01	0x02	0x03	0x04	0x05	…	0xFF

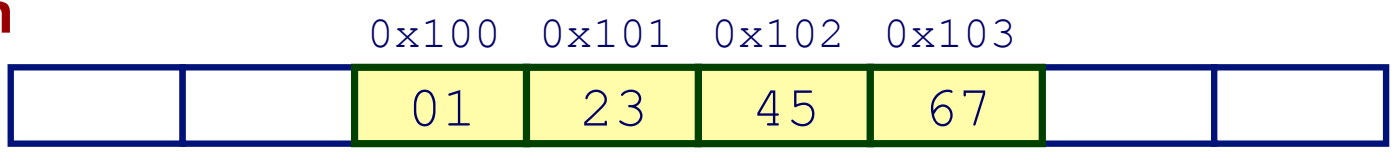
Memory is *byte*-addressable

- **Not** bit-addressable!
- **Not** int-addressable!
- If we want a specific *bit*:
  - Get the entire corresponding byte
  - Use bitwise operations to extract the bit
- How do we store an int?

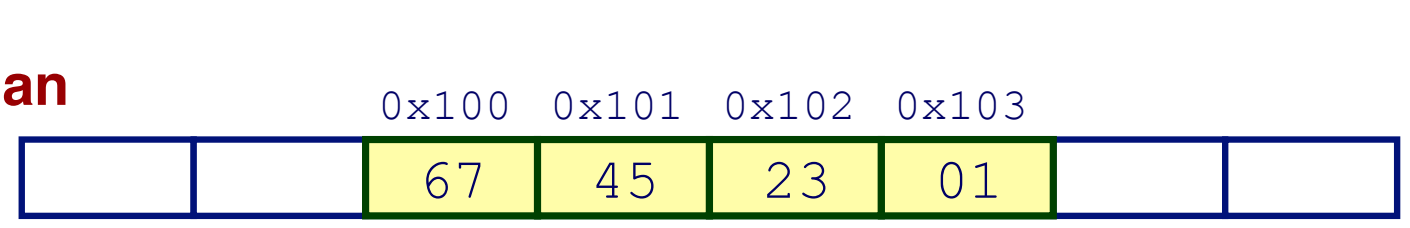
Endianness

```
int x = 0x01234567; // 32-bit num at address 0x100
```

Big Endian



Little Endian



Endianness

- A.k.a. byte order
- Big endian: *big* end comes first
- Little endian: *little* end comes first
- A property of the hardware of the machine
- Your laptops, krebs, etc. are little-endian

Endianness

- Computers prefer little-endian
- But humans prefer big-endian – why?

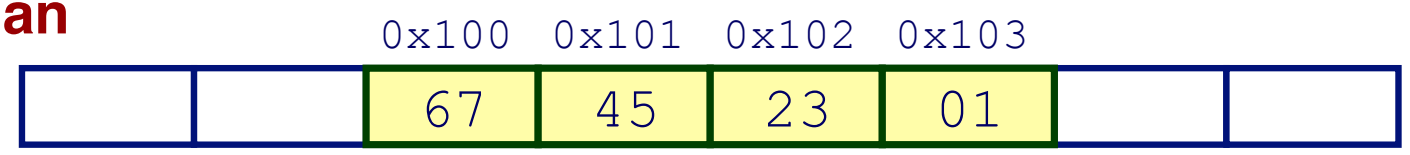
Endianness

- Computers prefer little-endian
- But humans prefer big-endian – why?
  - Purely customary
  - By custom, we draw memory from left-to-right
  - By custom, numbers grow from right-to-left

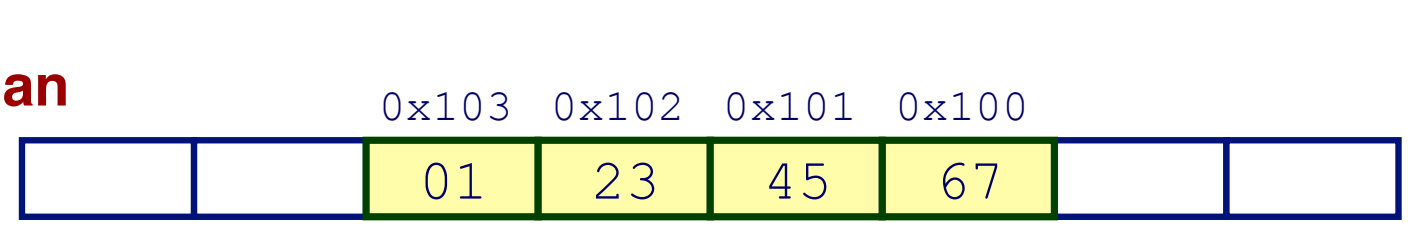
We have to break at least one custom

```
int x = 0x01234567; // 32-bit num at address 0x100
```

Little Endian



Little Endian



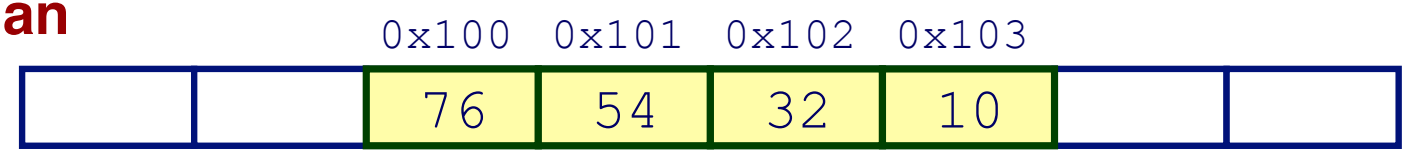
Remember!

- Endianness is *byte* order, **not** bit order
- Memory is byte-addressable, not bit-addressable
- Because we can’t address bits, we can’t measure their order

Is this correct?

```
int x = 0x01234567; // 32-bit num at address 0x100
```

Little Endian



Remember!

- Endianness is *byte* order, **not** nibble order
- For the same reason byte order doesn’t affect bits inside a byte, it doesn’t affect nibble order

Applies to all multi-byte data types

- Shorts (01 23 vs. 23 01)
- Ints (01 23 45 67 vs. 67 45 23 01)
- Longs (01 23 45 67 89 AB CD EF vs. EF CD AB 89 67 45 23 01)
- Floats and doubles

When does endianness matter?

- If we interpret the same memory as both an int and as bytes inside of our program
- If we write ints to disk
- If we send ints over the network

Word size

- *32-bit* computer, *64-bit* computer, etc.
- The word size is the natural unit of data
- It determines the number of bits in an address
- 32-bit machine → 32-bit addresses
- 64-bit machine → 64-bit addresses

Does address size matter?

- How many bytes can 32-bit addresses address?
- How many bytes can 64-bit addresses address?

Does address size matter?

- How many bytes can 32-bit addresses address? 2^{32} = 4 GB
- How many bytes can 64-bit addresses address? 2^{64} = 17.18 billion GB

Does address size matter?

- Suppose my word size is 8 bits
- I can address only 256 bytes!

Content:	0xFF	0x00	0x57	0x92	0xB3	0x8A	…	0xDC
Address:	0x00	0x01	0x02	0x03	0x04	0x05	…	0xFF

Typical word sizes

- Most laptops, desktops, phones are 64-bit
- 32-bit computers are overall most common
  - Remember: most computers are in everyday devices like cars, projectors, etc.

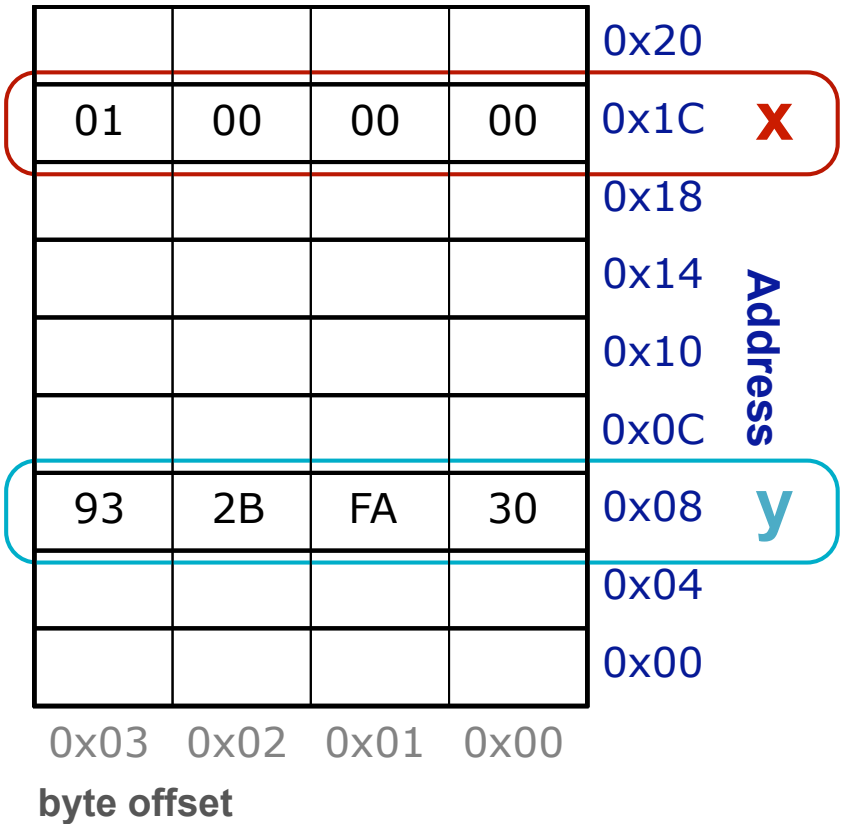

50
AMMO
100%
HEALTH
2 3 4
5 6 7
ARMS
0%
ARMOR
BULL 50 / 200
SHEL 0 / 50
RCKT 0 / 50
CELL 0 / 300

Addresses in C

- Every variable in C has a *value*
- Every variable in C has an *address*

Addresses in C

```
int x; // x at 0x1C
int y; // y at 0x08

x = 1;
y = 0x30FA2B93;
```



C pointers

- Address – index of a byte of memory
- Pointer – variable storing an address
- `T *p; // declare a pointer p pointing to type T`
- `&x // address of x`
- `*p // dereference p (evaluates to what p points to)`

Java “pointers”

- Rather *references*
- Java does a good job of hiding pointers

How many Blobs are there?

```
class Blob {
    // ... instance variables ...
}
public void doStuff() {
    Blob b1 = new Blob();
    Blob b2 = new Blob();
}
```

How many Blobs are there?

```
class Blob {
    // ... instance variables ...
}
public void doStuff() {
    Blob b1 = new Blob();
    Blob b2 = b1;
}
```

Java “pointers”

- If we have two variables but only one blob…
- Then b1 and b2 cannot be blobs…
- Because they are *references* to blobs
- C’s pointers are more powerful… and more dangerous!

C pointers

- Since addresses are just indexes, we can do arithmetic on them just like any array index
- You can point to any variable, not just class instantiations

C pointers

- Address – index of a byte of memory
- Pointer – variable storing an address
- `T *p; // declare a pointer p pointing to type T`
- `&x // address of x`
- `*p // dereference p (evaluates to what p points to)`

Declaring a pointer variable

- `T *p; // declare a pointer p pointing to type T`
- E.g., `int *p1; // pointer to an int`
- We say that the type of p1 is pointer to an int
- Or `double *p2; // pointer to a double`
- We say that the type of p2 is pointer to a double

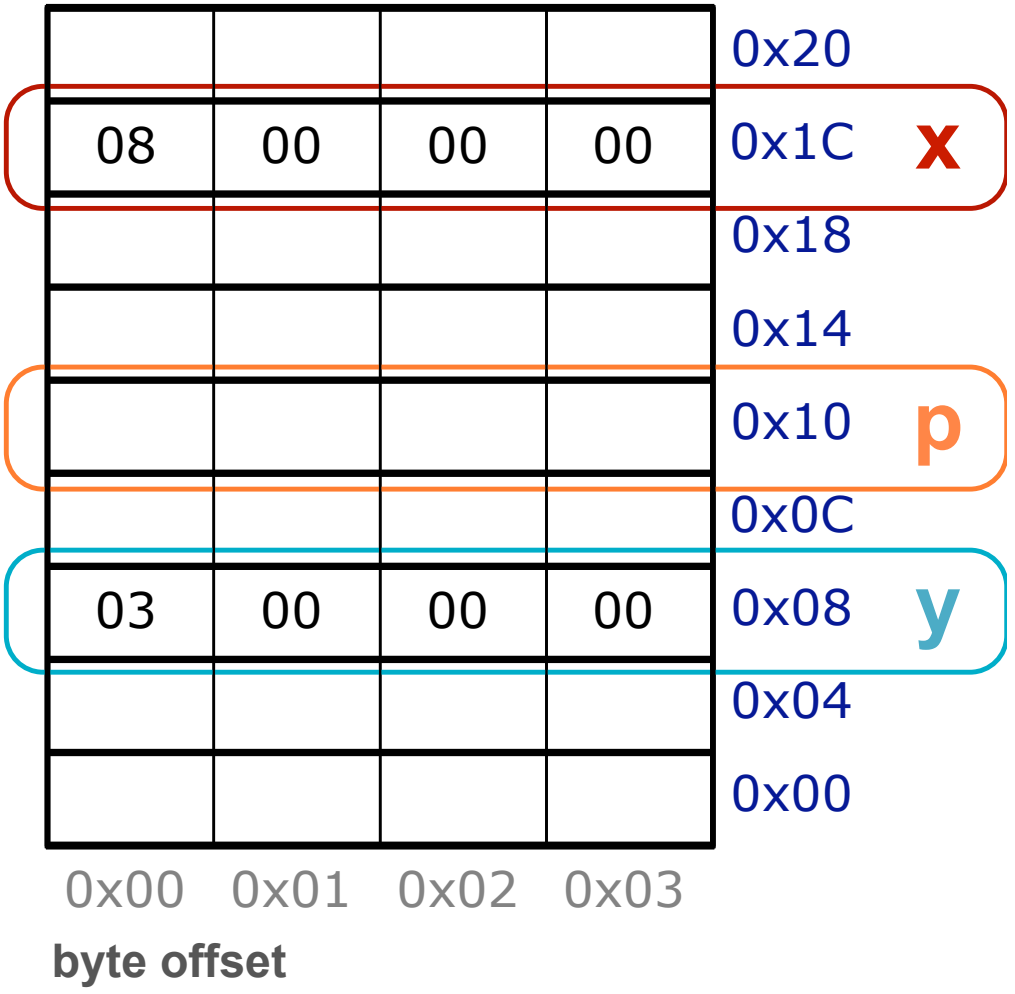
Creating an address

```
int i = 5;
double x = 0.25;
int *p1 = &i; // p1 now points to address of i
double *p2 = &x; // p2 now points to address of d
```

Dereferencing a pointer

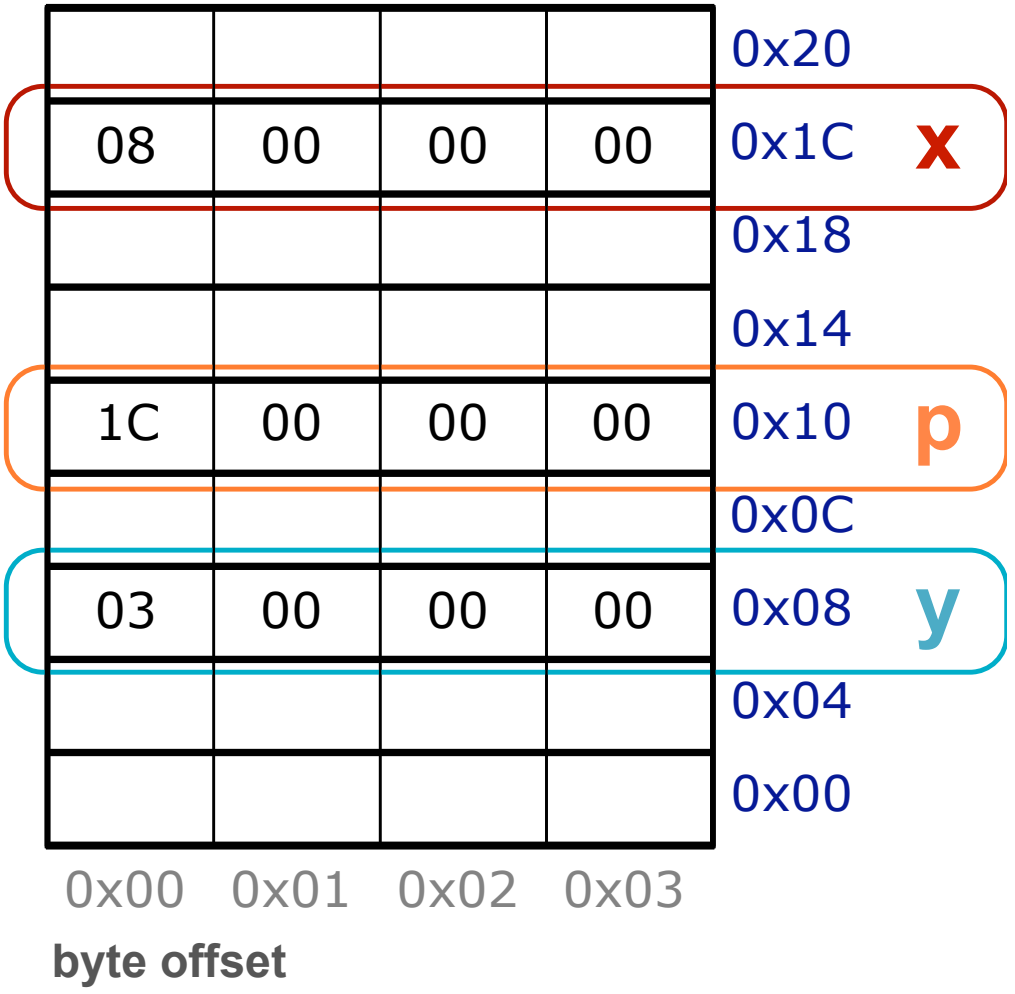
```
int i = 5;
double x = 0.25;
int *p1 = &i; // p1 now points to address of i
double *p2 = &x; // p2 now points to address of d
int j = *p1; // j is now 5
double y = *p2; // y is now 0.25
```

```
int *p; // p: 0x10

int x = 8; // x: 0x1C
int y = 3; // y: 0x08
```



```
int *p; // p: 0x10

int x = 8; // x: 0x1C
int y = 3; // y: 0x08
```

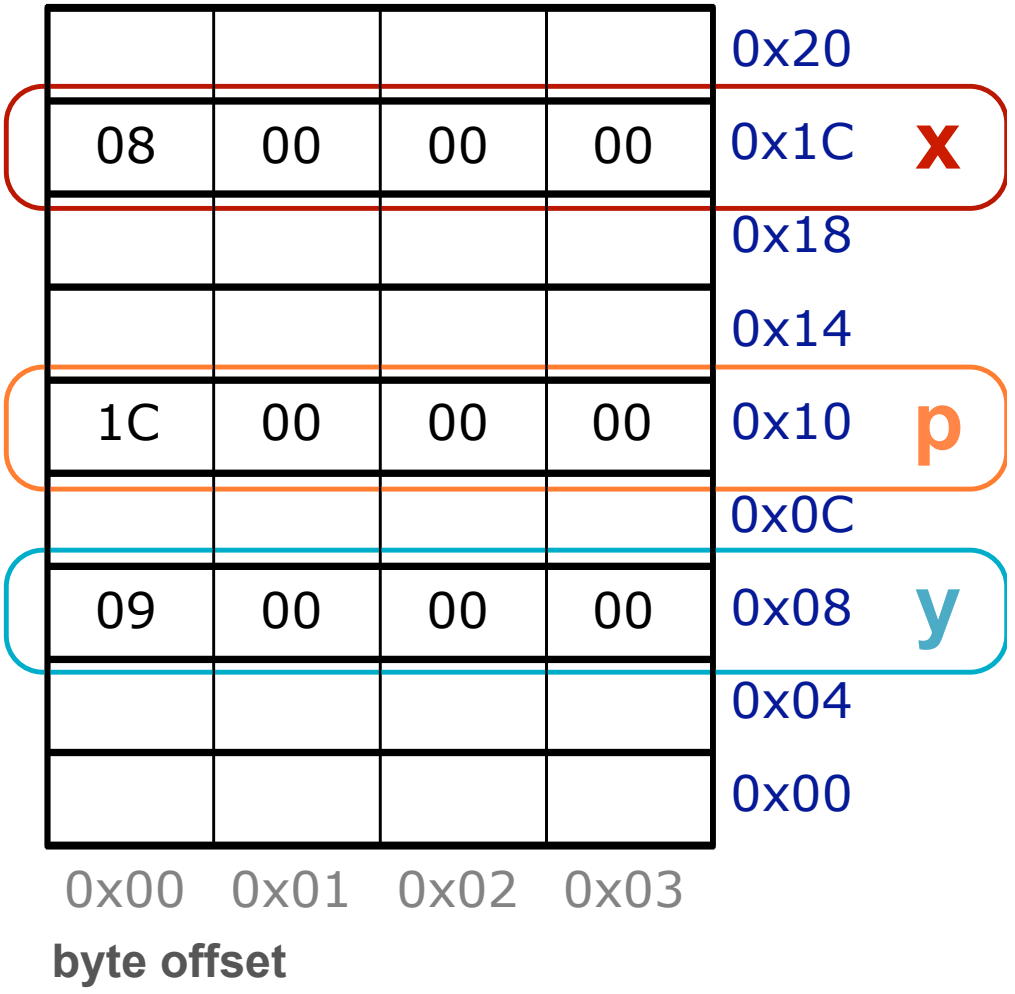
```
p = &x;
```



```
int *p; // p: 0x10

int x = 8; // x: 0x1C
int y = 3; // y: 0x08
```

```
p = &x;
```

```
y = 1 + *p;
```

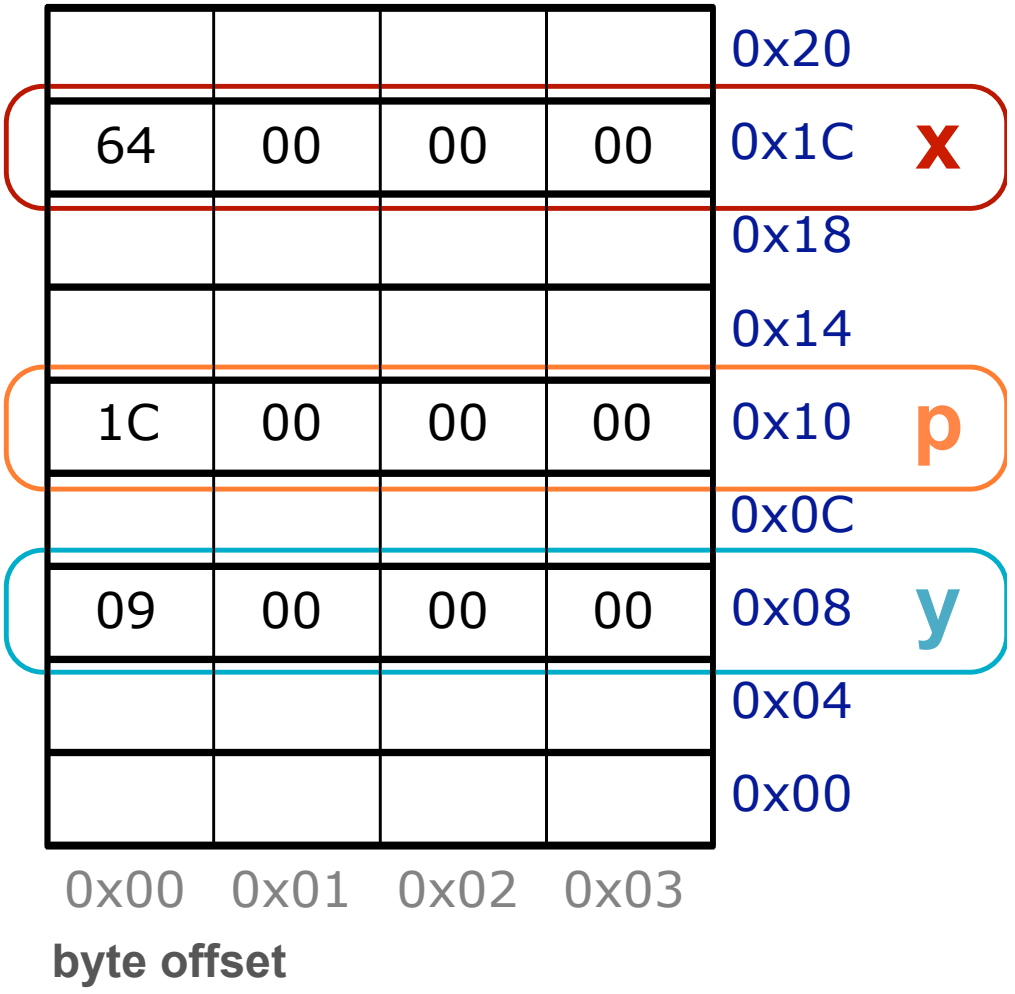


```
int *p; // p: 0x10

int x = 8; // x: 0x1C
int y = 3; // y: 0x08
```

```
p = &x;
```

```
y = 1 + *p;
```

```
*p = 100;
```



Remember!

- Variables have both:
  - An address
  - A value
- The variable x:
  - Has address 0x1c
  - Had value 8 (later value 100)
- Note: we replaced x’s value without an “x = …”!

TODO

- Assignment 1 is assigned (due September 24)