ECE 250 / CS 250
Computer Architecture

Bit Operations and Memory

Benjamin Lee

Slides based on those from Alvin Lebeck, Daniel Sorin, Andrew Hilton, Amir Roth, Gershon Kedem

Admin

- Homework 1
  - Due Jan 30, 11:55pm
  - Code must compile and run for credit
    - 10% for reasonable, commented code
    - 20% for code that compiles
    - Additional credit for satisfying each of 5 test cases
  - Start early and plan ahead

- Today's Outline
  - Floating point representations
  - Character representations
  - Bit operations
  - Memory: pointer arithmetic

Floating Point

- Option 1: Fixed Point
  - Binary Point (6 bits)
    - 001010
    - 0010.10
    - Right of binary point weight is $1/2^{-i}$ (i starts at 1 -> ½, ¼, 1/8,…)
  - General Fixed Point representation specifies <width, point postion>
    - eg., fixed<6,2>
  - Range limited (no good rep for large + small)
- Scientific notation is good
  - $6.82 * 10^{23}$
  - One digit, decimal point, some number, base 10, exponent (+/-)
- Can we do something similar in Binary?

Option 2: Floating Point Representation

- How about:
  - $+/- X.YYYYYY * 2^{+/-N}$

- Big numbers: large positive N
- Small numbers (<1): negative N
- Numbers near 0: small N

- This is "floating point" : most common representation for non-integer numbers (type float)

IEEE Single Precision Floating Point

- Specific format called IEEE single precision:
- $+/-\ 1.YYYYY * 2^{(N-127)}$
- “float” in Java, C, C++,…

- Sign: 1 sign bit (+ = 0, 1 = -)
- Exponent: 8-bit biased exponent (N-127)
  - N = E + 127 where E is actual exponent
- Mantissa: 23-bit mantissa (YYYY)
  - implicit 1 before binary point to save a bit

Floating Point Example

- Binary fraction example:
  - 101.101 = 4 + 1 + ½ + $^1/_8$ = 5.625
- For floating point, needs normalization:
  - 1.01101 * 2^2
- Sign is +, which = 0
- Exponent = 127 + 2 = 129 = 1000 0001
- Mantissa = 1.011 0100 0000 0000 0000 0000

31	30 … 23	22 … 0
0	1000 0001	011 0100 0000 0000 0000 0000
S	N	M

Floating Point Representation

Example:

What floating-point number is:

0xC1580000?

Answer

What floating-point number is

0xC1580000?

1100 0001 0101 1000 0000 0000 0000 0000

31	30 – 23	22 – 0
1	1000 0010	101 1000 0000 0000 0000 0000
S	N	M

- Sign = 1 means this is a negative number
- Exponent = (128+2)-127 = 3
- Mantissa = 1.1011
- -1.1011×2^3 = -1101.1 = -13.5

Trick question

- How do you represent 0.0 in IEEE Floating Point?
  - Why is this a trick question?

Trick question

- How do you represent 0.0 in IEEE Floating Point?
  - Why is this a trick question?
  - 0.0 = 000000000
  - But need 1.XXXXX representation?

Trick question

- How do you represent 0.0 in IEEE Floating Point?
  - Why is this a trick question?
  - 0.0 = 000000000
  - But need 1.XXXXX representation?
- Exponent of 0 is denormalized
  - Zero exponent and zero mantissa
  - Implicit 0. instead of implicit 1. in mantissa
  - Allows 0000….0000 to be 0
  - Helps with very small numbers near 0
- Results in +/- 0 in FP (but they are “equal”)

Other Special FP Values

- If exponent = 1111 1111 …
  - And if mantissa is zero, value is ∞
  - 1/0 = +∞; -1/0 = -∞
- If exponent = 1111 1111 …
  - And if mantissa is non-zero, value is NaN
  - sqrt(-42) = NaN

Floating Point Arithmetic

- Example in Decimal: 99.5 + 0.8
  - Step I: align exponents (if necessary)
    - Temporarily de-normalize operand with smaller exponent
    - Add 2 to exponent → Shift significand right by 2
    - $8.0*10^{-1} \rightarrow 0.08*10^{1}$
  - Step II: add significands
    - $9.95*10^{1} + 0.08*10^{1} \rightarrow 10.03*10^{1}$
  - Step III: normalize result
    - Shift significand right by 1
    - $10.03*10^{1} \rightarrow 1.003*10^{2}$

Floating Point Arithmetic

- Now a binary "quarter" example: 7.5 + 0.5
  - 8 bits: 1-bit sign, 3-bit exponent, 4-bit significand, bias is 3=(2^{N-1}-1)
  - 7.5 = 1.875*2^2 = 0 101 11110 (the 1 is the implicit leading 1)
    - 1.875 = $1*2^0+1*2^{-1}+1*2^{-2}+1*2^{-3}$
  - 0.5 = $1*2^{-1}$ = 0 010 10000

- Step I: align exponents (if necessary)
  - 0 010 10000 → 0 101 00010
  - Add 3 to exponent → shift significand right by 3
- Step II: add significands
  - 0 101 11110 + 0 101 00010 = 0 101 100000
- Step III: normalize result
  - 0 101 100000 → 0 110 10000
  - Shift significand right by 1 → add 1 to exponent

Rounding Errors

- We only have 32-bits to represent floats
  - Must approximate some values
  - Limited bits for mantissa
- Does (x+y)*z = (x*z+y*z)?
  - Mathematically yes, but assumes infinite precision
- Example in base 10,
  - four digits available (two to left, two to right of decimal point)
  - $x = 99.96 \times 10^3$
  - $x = x + 0.07$
  - $x = 100.03 \times 10^3$
  - $x = 10.00 \times 10^4$
- Numerical Analysis (CS 220) studies these issues

Floating Point Representation

- Double Precision Floating point:

  64-bit representation:
  - 1-bit sign
  - 11-bit (biased) exponent
  - 52-bit fraction (with implicit 1).

- “double” in Java, C, C++, …

S	Exp	Mantissa
1	11-bit	52 - bit

What About Strings?

Recall Strings

- char str1[256] = “hi”;
- str1[0] = ‘h’, str1[1] = ‘i’,str1[2] = 0;
- 0 is value of NULL character ‘\0’, identifies end of string

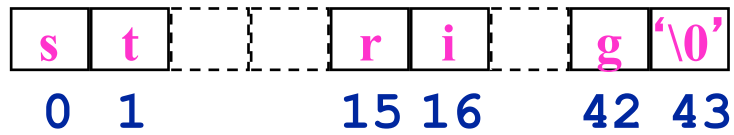
- A string is an array of characters
- So we need to represent characters

ASCII Character Representation

Oct.	Char														
000	nul	001	soh	002	stx	003	etx	004	eot	005	enq	006	ack	007	bel
010	bs	011	ht	012	nl	013	vt	014	np	015	cr	016	so	017	si
020	dle	021	dc1	022	dc2	023	dc3	024	dc4	025	nak	026	syn	027	etb
030	can	031	em	032	sub	033	esc	034	fs	035	gs	036	rs	037	us
040	sp	041	!	042	"	043	#	044	$	045	%	046	&	047	'
050	(	051	)	052	*	053	+	054	,	055	-	056	.	057	/
060	0	061	1	062	2	063	3	064	4	065	5	066	6	067	7
070	8	071	9	072	:	073	;	074	<	075	=	076	>	077	?
100	@	101	A	102	B	103	C	104	D	105	E	106	F	107	G
110	H	111	I	112	J	113	K	114	L	115	M	116	N	117	O
120	P	121	Q	122	R	123	S	124	T	125	U	126	V	127	W
130	X	131	Y	132	Z	133	[	134	\	135	]	136	^	137	_
140	`	141	a	142	b	143	c	144	d	145	e	146	f	147	g
150	h	151	i	250	j	153	k	154	l	155	m	156	n	157	o
160	p	161	q	162	r	163	s	164	t	165	u	166	v	167	w
170	x	171	y	172	z	173	{	174	|	175	}	176	~	177	del

- Each character represented by 7-bit ASCII code (packed into 8-bits)
- Convert upper to lower case ‘A’ + 32 = ‘a’

Review: Strings as Arrays



- A string is an array of characters with '\0' at the end
- Each element is one byte, ASCII code
- '\0' is null (ASCII code 0)
- Char str1[256]
- Char *str
- Str = (char *) str

Unicode

- Many types
- UTF-8: variable length encoding backward compatible with ASCII
  - Linux
- UTF-16: variable length
  - Windows, Java
- UTF-32: fixed length

Bit Manipulations

Problem

- 32-bit word contains many values
  - e.g., input device, sensors, etc.
  - current x,y position of mouse and which button (left, mid, right)
- Assume x, y position is 0-255
  - How many bits for position?
  - How many for button?

Goal

- Extract position and button from 32-bit word
- Need operations on individual bits of word

Bitwise AND / OR / XOR

- & operator performs bitwise AND
- | operator performs bitwise OR
- ^ operator performs bitwise Exclusive OR (XOR)
- Per bit

0 & 0 = 0	0 | 0 = 0	0 ^ 0 = 0
0 & 1 = 0	0 | 1 = 1	0 ^ 1 = 1
1 & 0 = 0	1 | 0 = 1	1 ^ 0 = 1
1 & 1 = 1	1 | 1 = 1	1 ^ 1 = 0

- For multiple bits, apply operation to individual bits in same position

AND	OR	XOR
011010	011010	011010
101110	101110	101110
001010	111110	110100

Mouse Example

- 32-bit word with x,y and button fields
  - bits 0-7 contain x position
  - bits 8-15 contain y position
  - bits 16-17 contain button (0 = left, 1 = middle, 2 = right)

- Use bitwise operations to extract specific fields from bit string…

button y x

`0x1a34c = 01 1010 0011 0100 1100`

Mouse Solution

- AND with a bit mask
  - specific values that clear some bits, but pass others through
- To extract x position use mask 0x000ff
  - xpos = 0x1a34c & 0x000ff

```
               button        y              x
0x1a34c = 01 1010 0011 0100 1100
0x000ff = 00 0000 0000 1111 1111
0x0004c = 00 0000 0000 0100 1100
```

More of the Mouse Solution

- Extract y position with mask 0x0ff00
  - ypos = 0x1a34c & 0x0ff00

- Extract button with mask 0x30000
  - button = 0x1a34c & 0x30000

- Not quite done…why?

The Shift Operator

- >> shifts right, << shifts left,
- operands are int and number of positions to shift
  - Shifting signed integers requires sign extension

- (1 << 3) shifts …001 -> …1000 (it's 2^3)
- 0xff << 8 = 0xff00
- 0xff00 >> 8 = 0x00ff if integer is unsigned
- 0xff00 >> 8 = 0xffff if integer is signed

- Example: shift to extract ypos and button values

  ypos = (0x1a34c & 0x0ff00) >> 8

  button = (0x1a34c & 0x30000) >> 16

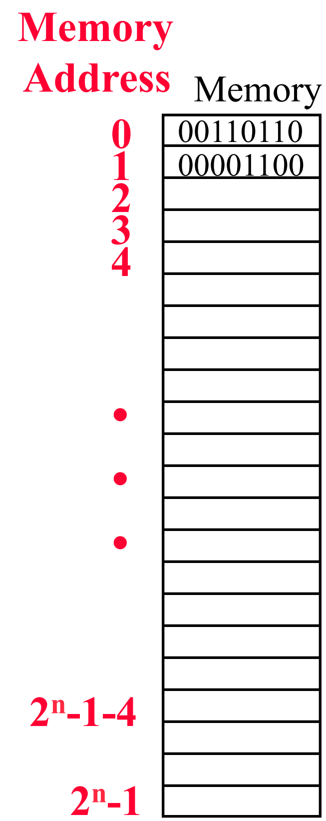
Extracting Parts of Floating Point Number

- x is a 32-bit word

```
#define EXP_BITS 8
#define FRACTION_BITS 23
#define SIGN_MASK 0x80000000
#define EXP_MASK  0x7f800000
#define FRACTION_MASK  0x007fffff
Struct myfloat {
  int sign;
  unsigned int exp;
  unsigned int fraction;
};
struct myfloat x;

num->sign = (x & SIGN_MASK) >> (EXP_BITS + FRACTION_BITS);
num->exp  = (x & EXP_MASK) >> FRACTION_BITS;
num->fraction = x & FRACTION_MASK;
```

A Program's View of Memory

- What is Memory?
  - A large linear array of bits
- Find things by indexing into array
  - memory address (unsigned integer)
  - read to and write from address
- Processor issues commands to read/write specific locations
  - Read from memory location 0x1400
  - Write 0xff to memory location 0x8675309
- Array of …
  - Bytes? 32-bit ints? 64-bit ints?



Processor Word Size

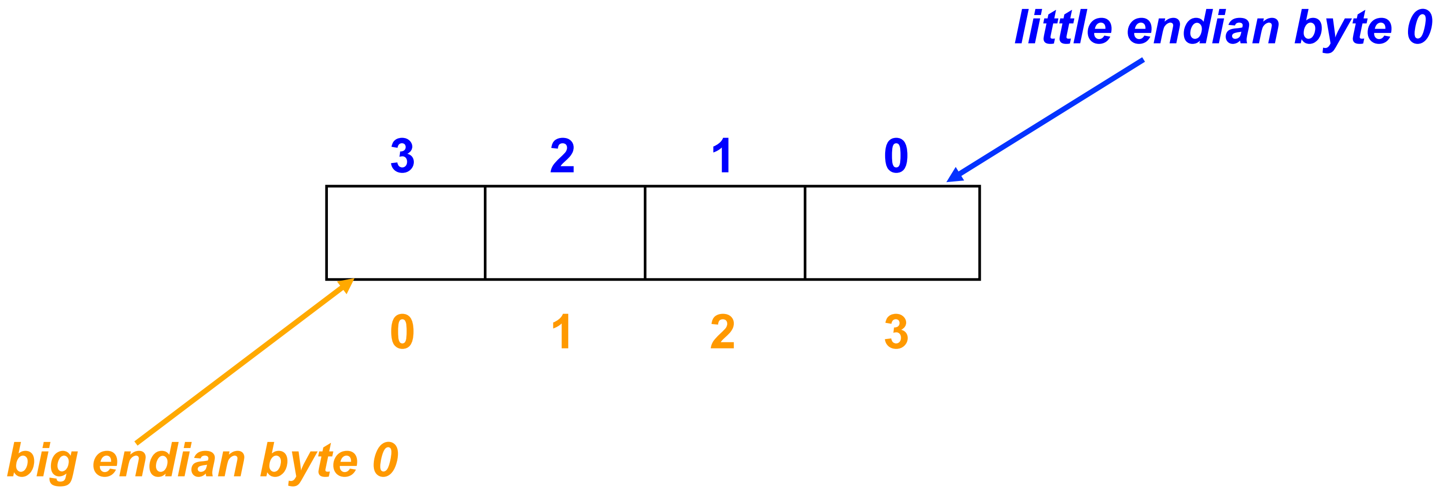
- Processor has word size
  - Nominal size of integer-valued data, addresses
  - 32-bit vs. 64-bit addresses
  - 32-bit words addressed 0x100 and 0x104

- Most systems are byte (8-bit) addressed
  - Support to load/store 16, 32, 64 bit quantities
    - short, int, long long, etc. data types
  - What is order of bytes in memory?
  - Byte ordering varies from system to system

Endianess and Byte Ordering

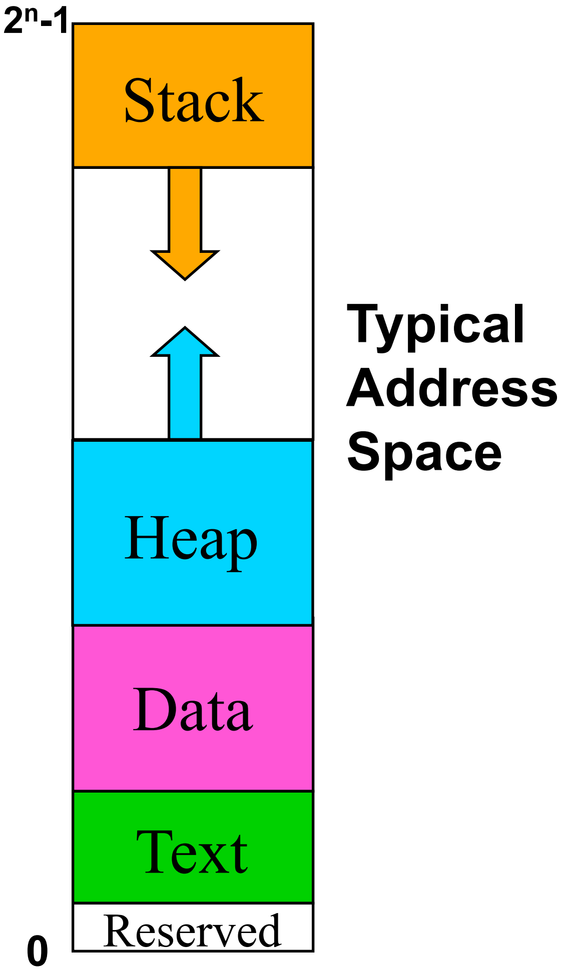
<u>Byte Order</u>

- Big Endian: byte 0 is 8 most significant bits
  - IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- Little Endian: byte 0 is 8 least significant bits
  - Intel 80x86, DEC Vax, DEC Alpha



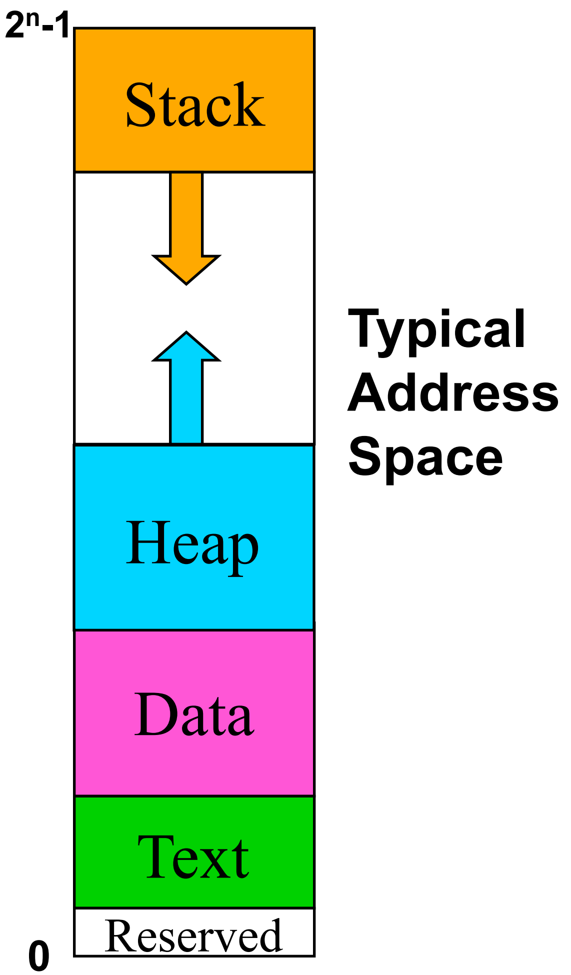
Memory Partitions

- Text for instructions
  - add dest, src1, src2
  - mem[dest] = mem[src1] + mem[src2]
- Data
  - static (constants, global variables)
  - dynamic (heap, new allocated)
  - grows up
- Stack
  - local variables
  - grows down
- Variables are names for memory locations
  - int x;



Memory Layout: Example

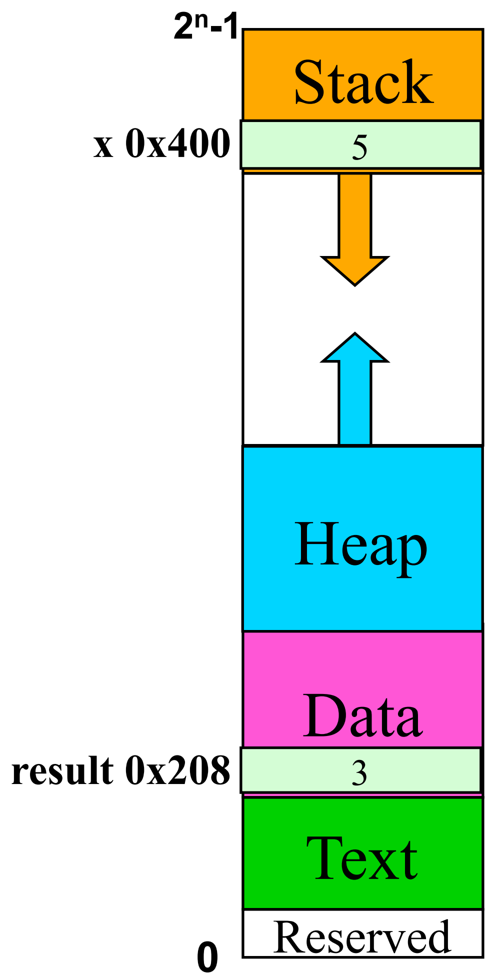
```
int anumber = 3;

int factorial (int x) {
  if (x == 0) {
    return 1;
  }
  else {
    return x * factorial (x – 1);
  }
}

int main (void) {
  int z = factorial (anumber);
  printf("%d\n", z);
  return 0;
}
```



A Simple Program's Memory Layout

```
...
int result; // global var
main()
{
  int x;
  ...
  result = x + result;
  ...
}
```

mem[0x208] = mem[0x400] + mem[0x208]



Review: Pointers

- "address of" operator &
  - don't confuse with bitwise AND operator (later)

<u>Given</u>

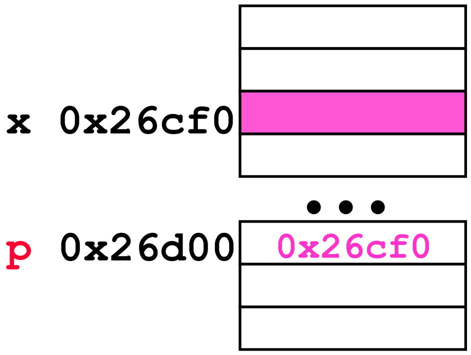
```
int x; int* p;  // p points to an int
p = &x;
```

<u>Then</u>

```
*p = 2;  and x = 2; produce the same result
 Note: p is a pointer, *p is an int
```

- What happens for `p = 2?;`

On 32-bit machine, p is 32-bits



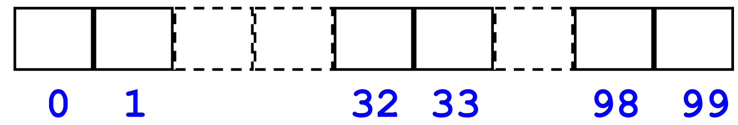
Example Array malloc() & free()

```
#include <stdio.h>
#include <stdlib.h> /* so we get malloc and free definitions */

main() {
        char *str;
        int *ar;

        str = (char *) malloc(256);
        ar = (int *) malloc(100*sizeof(int));

        str[0] = 'H'; str[1] = 'i'; str[2] = 0;
        ar[24] = 272;
        printf("str = %s, ar[24]= %d\n",str,ar[24]);

        free(str);
        free(ar);
}
```

Address Calculation

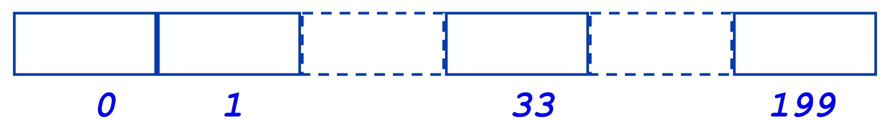
- x is a pointer, what is x+33?
- A pointer, but where?
  - what does calculation depend on?
- Result of adding an int to a pointer depends on size of object pointed to
  - One reason why we tell compiler what type of pointer we have, even though all pointers are really the same thing (and same size)

```
int* a=malloc(100*sizeof(int));
```




```
a[33] is the same as *(a+33)
if a is 0x00a0, then a+1 is
0x00a4, a+2 is 0x00a8
(decimal 160, 164, 168)
```

```
double* d=malloc(200*sizeof(double));
```




```
*(d+33) is the same as d[33]
if d is 0x00b0, then d+1 is
0x00b8, d+2 is 0x00c0
(decimal 176, 184, 192)
```

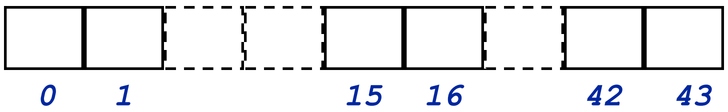
More Pointer Arithmetic

- what's at `*(begin+44)?`

- what does `begin++` mean?

- how are pointers compared using < and using == ?

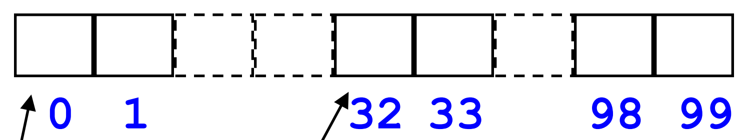
- what is value of `end - begin`?




```
char* a = new char[44];
char* begin = a;
char* end = a + 44;

while (begin < end)
{
    *begin = 'z';
    begin++;
}
```

More Pointers & Arrays

```
int * a = new int[100];
```




```
a is a pointer
*a is an int
a[0] is an int (same as *a)
a[1] is an int
a+1 is a pointer
a+32 is a pointer
*(a+1) is an int (same as a[1])
*(a+99) is an int
*(a+100) is trouble
```

Array Example

```
#include <stdio.h>

main()
{
  int *a = (int*)malloc (100 * sizeof(int));
  int *p = a;
  int k;

  for (k = 0; k < 100; k++)
    {
      *p = k;
      p++;
    }
  printf("entry 3 = %d\n", a[3])
}
```

C Array of Structures: Linked List

```
#include <stdio.h>
#include <stdlib.h>
struct node {
  int me;
  struct node *next;
};
int main()
{
  struct node *ar;
  struct node *p;
  int k;
  ar = (struct node *)
   malloc(10*sizeof(struct node));
  p = ar;
for (k = 0; k < 9; k++)
    {
      p->me = k;
      p->next = ar + k + 1;
      p++;
    }
   p->me = 9;
   p->next = NULL;
   p = &ar[0];
   while (p != NULL) {
      printf("%d 0x%lx 0x%lx\n", \
         p->me, (unsigned long) p,
         (unsigned long) p->next);
         p = p->next;
   }
   return(0);
}}
```

- Given ar = 0x10000, what does memory layout look like?
  - What is stored at each address?

Memory Layout

Output

Me	p	p->next
0	0x26ca8	0x26cb0
1	0x26cb0	0x26cb8
2	0x26cb8	0x26cc0
3	0x26cc0	0x26cc8
4	0x26cc8	0x26cd0
5	0x26cd0	0x26cd8
6	0x26cd8	0x26ce0
7	0x26ce0	0x26ce8
8	0x26ce8	0x26cf0
9	0x26cf0	0x0

- NOTE: If you run this program twice you'll get different addresses!

Memory Address	Memory Contents	Source Symbol
0x26ca8	0	me } ar[0]
	0x26cb0	next } ar[0]
0x26cb0	1	
	0x26cb8	
0x26cb8	2	
	0x26cc0	
0x26cc0	3	
	0x26cc8	
0x26cc8	4	
	0x26cd0	
0x26cd0	5	
	0x26cd8	
0x26cd8	6	
	0x26ce0	
0x26ce0	7	
	0x26ce8	
0x26ce8	8	
	0x26cbf0	
0x26cf0	9	me } ar[9]
	0x0	next } ar[9]

me is int (4 bytes)

next is node* (4 bytes)

Summary: From C to Binary

- Everything must be represented in binary!
- There are issues for numbers
  - Max, min, rounding, etc.
- Computer memory is linear array of bytes
- Pointer is memory location that contains address of another memory location
- We'll visit these topics again throughout semester
- Next week
  - Assembly Programming