ECE 250 / CPS 250
Computer Architecture

From C to Binary

Benjamin Lee

Slides based on those from

Andrew Hilton (Duke), Alvy Lebeck (Duke)
Benjamin Lee (Duke), and Amir Roth (Penn)

Outline

- Previously:
  - Computer is machine that does what we tell it to do
- Next:
  - How do we tell computers what to do?
  - How do we represent data objects in binary?
  - How do we represent data locations in binary?

Representing High Level Things in Binary

- Computers represent everything in binary
- Instructions are specified in binary
- Instructions must be able to describe
  - Operation types (add, subtract, shift, etc.)
  - Data objects (integers, decimals, characters, etc.)
  - Memory locations
- Example:

```
int x, y;              // Where are x and y?  How to represent an int?
bool decision;         // How do we represent a bool?  Where is it?
y = x + 7;             // How do we specify "add"?  How to represent 7?
decision=(y>18);  //  Etc.
```

Representing Operation Types

- How do we tell computer to add? Shift? Read from memory? Etc.
- Arbitrarily! ☺
- Each Instruction Set Architecture (ISA) has its own binary encodings for each operation type
- E.g., in MIPS:
  - Integer add is: 00000 010000
  - Read from memory (load) is: 010011
  - Etc.

Representing Data Types

- How do we specify an integer? A character? A floating point number? A bool? Etc.
- Same as before: binary!
- Key Idea: the same 32 bits might mean one thing if interpreted as an integer but another thing if interpreted as a floating point number

Basic Data Types

Bit (bool): 0, 1

Bit String: sequence of bits of a particular length
- 4 bits is a nibble
- 8 bits is a byte
- 16 bits is a half-word
- 32 bits is a word
- 64 bits is a double-word
- 128 bits is a quad-word

Integers (int, long):
- "2's Complement" (32-bit or 64-bit representation)

Floating Point (float, double):
- Single Precision (32-bit representation)
- Double Precision (64-bit representation)
- Extended (Quad) Precision (128-bit representation)

Character (char):
- ASCII 7-bit code

Issues for Binary Representation of Numbers

- There are many ways to represent numbers in binary
  - Binary representations are encodings → many encodings possible
  - What are the issues that we must address?
- Issue #1: Complexity of arithmetic operations
- Issue #2: Negative numbers
- Issue #3: Maximum representable number
- Choose representation that makes these issues easy for machine, even if it's not easy for humans (i.e., ECE/CS 250 students)
  - Why? Machine has to do all the work!

Sign Magnitude

- Use leftmost bit for + (0) or – (1):
- 6-bit example (1 sign bit + 5 magnitude bits):
- +17 = 010001
- -17 = 110001
- Pros:
  - Conceptually simple
  - Easy to convert
- Cons:
  - Harder to compute (add, subtract, etc) with
  - Positive and negative 0: 000000 and 100000

1's Complement Representation for Integers

- Use largest positive binary numbers to represent negative numbers
- To negate a number,

  invert ("not") each bit:

  0 → 1

  1 → 0
- Cons:
  - Still two 0s (yuck)
  - Still hard to compute with

0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-7
1001	-6
1010	-5
1011	-4
1100	-3
1101	-2
1110	-1
1111	-0

2's Complement Integers

- Use large positives to represent negatives
- $(-x) = 2^n - x$
- This is 1's complement + 1
- $(-x) = 2^n - 1 - x + 1$
- So, just invert bits and add 1

<u>6-bit examples:</u>

$010110_2 = 22_{10}$; $101010_2 = -22_{10}$

$1_{10} = 000001_2$; $-1_{10} = 111111_2$

$0_{10} = 000000_2$; $-0_{10} = 000000_2$ → good!

0000	**0**
0001	**1**
0010	**2**
0011	**3**
0100	**4**
0101	**5**
0110	**6**
0111	**7**
1000	**-8**
1001	**-7**
1010	**-6**
1011	**-5**
1100	**-4**
1101	**-3**
1110	**-2**
1111	**-1**

Pros and Cons of 2's Complement

- Advantages:
  - Only one representation for 0 (unlike 1's comp): 0 = 000000
  - Addition algorithm is much easier than with sign and magnitude
    - Independent of sign bits

- Disadvantage:
  - One more negative number than positive
  - Example: 6-bit 2's complement number 100000_2 = -32_{10}; but 32_{10} could not be represented

All modern computers use 2's complement for integers

2's Complement Precision Extension

- Most computers today support 32-bit (int) or 64-bit integers
  - Specify 64-bit using gcc C compiler with long long
- To extend precision, use sign bit extension
  - Integer precision is number of bits used to represent a number

Examples

14_{10} = 001110_2 in 6-bit representation.

14_{10} = 000000001110_2 in 12-bit representation

-14_{10} = 110010_2 in 6-bit representation

-14_{10} = 111111110010_2 in 12-bit representation.

Binary Math : Addition

- Suppose we want to add two numbers:

```
     00011101
+    00101011
-------------
```

- How do we do this?

Binary Math : Addition

- Suppose we want to add two numbers:

```
    00011101           695
+   00101011         + 232
```

- How do we do this?
  - Let’s revisit decimal addition
  - Think about the process as we do it

Binary Math : Addition

- Suppose we want to add two numbers:

```
      00011101              695
+     00101011            + 232
  ------------            -----
                              7
```

- First add one’s digit 5+2 = 7

Binary Math : Addition

- Suppose we want to add two numbers:

$$\begin{array}{r} \\ 00011101 \\ +\quad 00101011 \\ \hline \end{array} \qquad\qquad \begin{array}{r} 1 \\ 695 \\ +\ \ 232 \\ \hline 27 \end{array}$$

- First add one’s digit 5+2 = 7
- Next add ten’s digit 9+3 = 12 (2 carry a 1)

Binary Math : Addition

- Suppose we want to add two numbers:

$$\begin{array}{rr} & 00011101 \\ + & 00101011 \\ \hline \end{array} \qquad\qquad \begin{array}{rr} & 695 \\ + & 232 \\ \hline & 927 \end{array}$$

- First add one’s digit 5+2 = 7
- Next add ten’s digit 9+3 = 12 (2 carry a 1)
- Last add hundred’s digit 1+6+2 = 9

Binary Math : Addition

- Suppose we want to add two numbers:

```
     00011101
 +   00101011
```

- Back to the binary:
- First add 1’s digit 1+1 = …?

Binary Math : Addition

- Suppose we want to add two numbers:

```
            1
      00011101
 +    00101011
--------------
             0
```

- Back to the binary:
- First add 1’s digit 1+1 = 2 (0 carry a 1)

Binary Math : Addition

- Suppose we want to add two numbers:

```
         11
    00011101
 +  00101011
------------
          00
```

- Back to the binary:
- First add 1’s digit 1+1 = 2 (0 carry a 1)
- Then 2’s digit: 1+0+1 =2 (0 carry a 1)
- You all finish it out….

Binary Math : Addition

- Suppose we want to add two numbers:

```
      111111
     00011101     =  29
 +   00101011     =  43
 ---------------
     01001000     =  72
```

- Can check our work in decimal

Binary Math : Addition

- What about this one:

```
       01011101
  +    01101011
  -------------
```

Binary Math : Addition

- What about this one:

$$\begin{array}{rlcr} & 1111111 & & \\ & 01011101 & = & 93 \\ + & 01101011 & = & 107 \\ \hline & 11001000 & = & -56 \end{array}$$

- But… that can’t be right?
  - What do you expect for the answer?
  - What is it in 8-bit signed 2’s complement?

Integer Overflow

- Answer should be 200
    - Not representable in 8-bit signed representation
    - No right answer
- Call Integer Overflow
- Real problem in programs

Subtraction

- 2's complement makes subtraction easy:
  - Remember: A - B = A + (-B)
  - And: -B = ~B + 1

    ↑ that means flip bits ("not")
  - So we just flip the bits and start with carry-in (CI) = 1
  - Later: No new circuits to subtract (re-use adder hardware!)

```
                                 1
  0110101       ->         0110101
- 1010010                + 0101101
---------                ---------
```

What About Non-integer Numbers?

- There are infinitely many real numbers between two integers
- Many important numbers are real
  - Speed of light ~= $3\text{x}10^8$
  - Pi = 3.1415…
- Fixed number of bits limits range of integers
  - Can't represent some important numbers
- Humans use Scientific Notation
  - $1.3\text{x}10^4$

Option 1: Fixed point

- Represent non-integer in two parts
  - Integer and fraction parts separated by binary point
  - Example: 8 bit fixed-point number with 3 fractional bits
  - $(00010.110)_2 = 1*2^1 + 1*2^{-1} + 1*2^{-2} = (2.75)_{10}$

- Pros:
  - Similar to integer representation, except for binary point
  - Addition/subtraction just like integers

- Cons:
  - Loss of range and precision
  - Example: 1 fractional bit gives precision to within 0.5

Can we do better?

- Think about scientific notation for a second:
- For example:

  $6.82 * 10^{23}$

- Real number, but comprised of ints:
  - 6 generally only 1 digit here
  - 82 any number here
  - 10 always 10 (base we work in)
  - 23 can be positive or negative
- Can we do something like this in binary?

Option 2: Floating Point

- 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 way

IEEE single precision floating point

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

- Assume X is always 1 (saves us a bit)
- 1 sign bit (+ = 0, 1 = -)
- 8 bit biased exponent (do N-127)
- Implicit 1 before *binary point*
- 23-bit *mantissa* (YYYYY)

Binary fractions

- 1.YYYY has a binary point
  - Like a decimal point but in binary
  - After a decimal point, you have
    - tenths
    - hundredths
    - Thousandths
    - ….
- So after a binary point you have…

Binary fractions

- 1.YYYY has a binary point
  - Like a decimal point but in binary
  - After a decimal point, you have
    - Tenths
    - Hundredths
    - Thousandths
    - ….
- So after a binary point you have…
  - Halves
  - Quarters
  - Eighths
  - ….

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

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	E	F

X = 1 1000 0010 101 1000 0000 0000 0000 0000

Sign = 1 which is negative

Exponent = (128+2)-127 = 3

Mantissa = 1.1011

-1.1011x2^3 = -1101.1 = -13.5

Trick question

- How do you represent 0.0?
  - Why is this a trick question?

Trick question

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

Trick question

- How do you represent 0.0?
    - Why is this a trick question?
    - 0.0 = 000000000
    - But need 1.XXXXX representation?
- Exponent = 0000 0000 is denormalized
    - Implicit 0. instead of 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 Weird FP numbers

- Exponent = 1111 1111 also not standard
  - All 0 mantissa: +/- ∞

    1/0 = +∞

    -1/0 = -∞
  - Non zero mantissa: Not a Number (NaN)

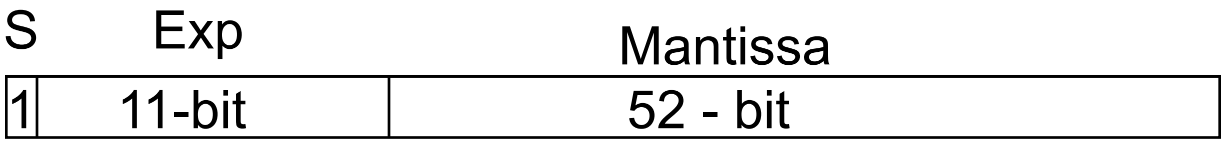
    sqrt(-42) = NaN

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?

- Many important things stored as strings…
  - E.g., your name
- How should we store strings?

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

Outline

- Previously:
  - Computer is machine that does what we tell it to do
- Next:
  - How do we tell computers what to do?
  - How do we represent data objects in binary?
  - How do we represent data locations in binary?

Computer Memory

- Where do we put the data (and instructions)?
  - Registers [more on these later]
    - In the processor core
    - Compute directly on them
    - Relatively few of them (~16-64)

  - Memory

Computer Memory

- Where do we put these numbers?
  - Registers [more on these later]
    - In the processor core
    - Compute directly on them
    - Few of them (~16 or 32 registers, each 32-bit or 64-bit)
  - Memory   [Our focus now]
    - External to processor core
    - Load/store values to/from registers
    - Very large (multiple GB)

Memory Organization

- Memory: billions of locations…how to get the right one?
  - Each memory location has an address
  - Processor asks to read or write specific address
    - Memory, please load address 0x123400
    - Memory, please write 0xFE into address 0x8765000
  - Kind of like a giant array

Memory Organization

- Memory: billions of locations…how to get the right one?
  - Each memory location has an address
  - Processor asks to read or write specific address
    - Memory, please load address 0x123400
    - Memory, please write 0xFE into address 0x8765000
  - Kind of like a giant array
    - Array of what?
      - Bytes?
      - 32-bit ints?
      - 64-bit ints?

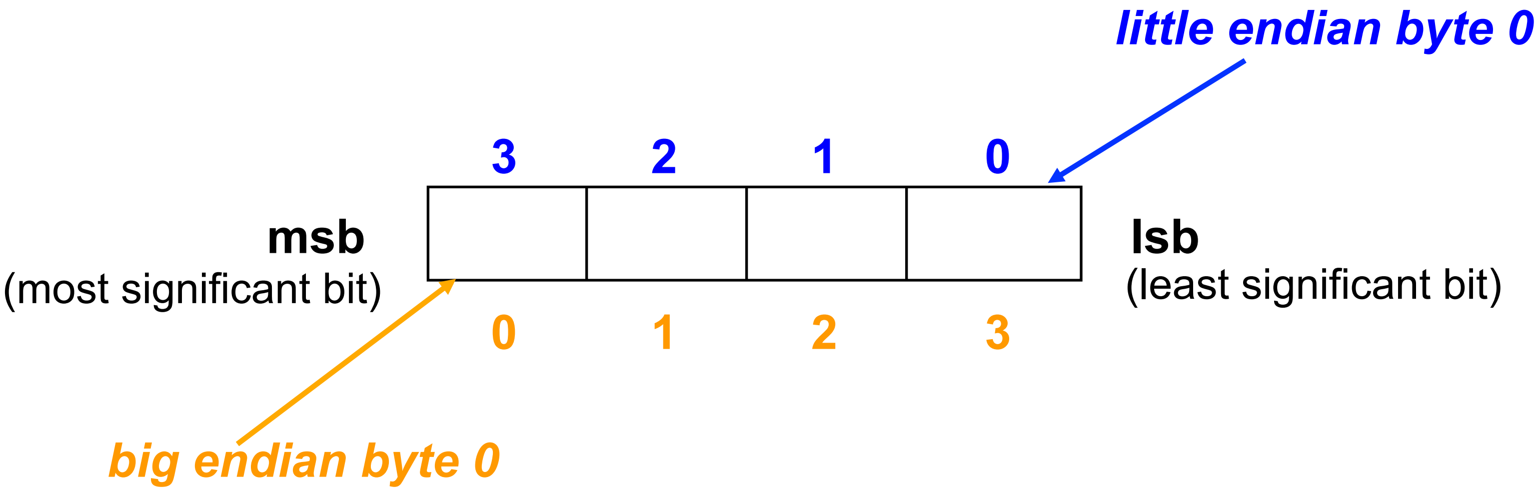
Memory Organization

- Most systems: byte (8-bit) addressed
- Memory is “array of bytes”
    - Each address specifies 1 byte
- Support to load/store 16, 32, 64 bit quantities
    - Byte ordering varies from system to system

Word of the Day: Endianess

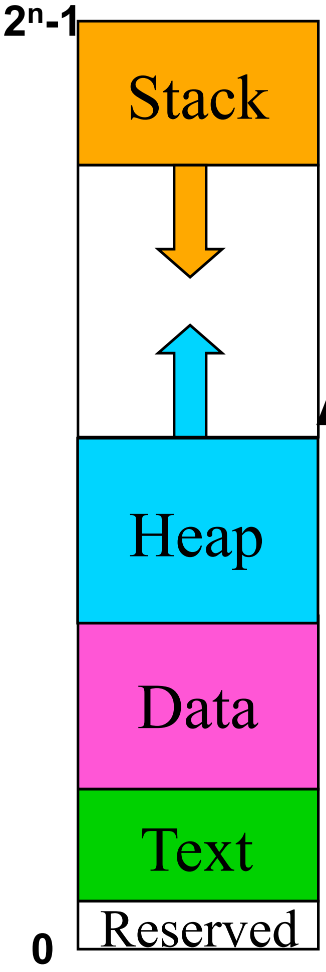
Byte Order

- 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 Layout

- Memory is array of bytes, but there are conventions as to what goes where in this array
- Text: instructions (the program to execute)
- Data: global variables
- Stack: local variables and other per-function state; starts at top & grows downward
- Heap: dynamically allocated variables; grows upward
- What if stack and heap overlap????



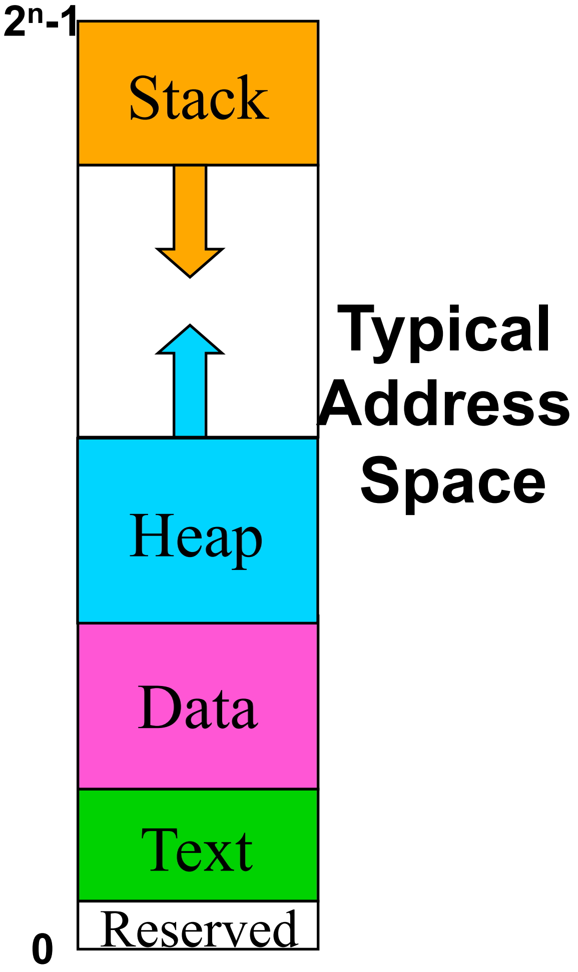
Typical Address Space

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;
}
```



Let’s do a little Java…

```
public class Example {
  public static void swap (int x, int y) {
    int temp = x;
    x = y;
    y = temp;
  }
  public static void main (String[] args) {
    int a = 42;
    int b = 100;
    swap (a, b);
    System.out.println(“a =“ + a + “ b = “ + b);
  }
}
```

- What does this print? Why?

Let’s do a little Java…

```
public class Example {
  public static void swap (int x, int y) {
    int temp = x;
    x = y;
    y = temp;
  }
  public static void main (String[] args) {
    int a = 42;
    int b = 100;
  → swap (a, b);
    System.out.println(“a =“ + a + “ b = “ + b);
  }
}
```

- What does this print? Why?

Stack

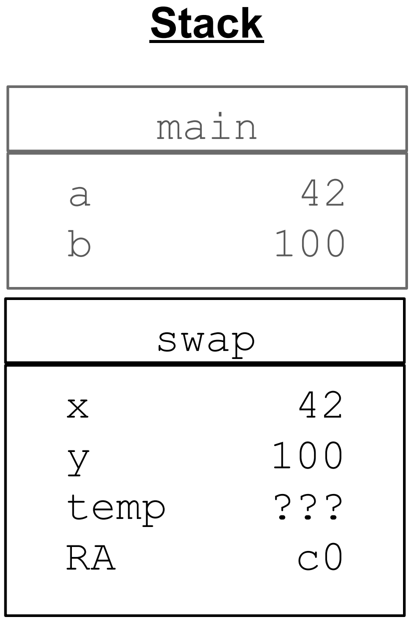
main

a 42

b 100

Let’s do a little Java…

```
public class Example {
  public static void swap (int x, int y) {
    int temp = x;
    x = y;
    y = temp;
  }
  public static void main (String[] args) {
    int a = 42;
    int b = 100;
c0  swap (a, b);
    System.out.println(“a =“ + a + “ b = “ + b);
  }
}
```



Stack

main	
a	42
b	100

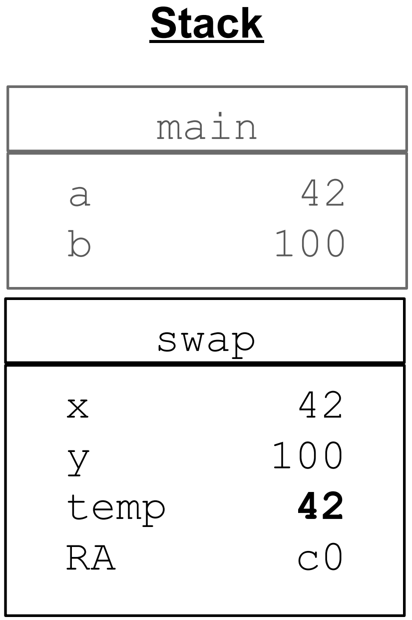
swap	
x	42
y	100
temp	???
RA	c0

- What does this print? Why?

Let’s do a little Java…

```
public class Example {
  public static void swap (int x, int y) {
    int temp = x;
    x = y;
    y = temp;
  }
  public static void main (String[] args) {
    int a = 42;
    int b = 100;
c0  swap (a, b);
    System.out.println(“a =“ + a + “ b = “ + b);
  }
}
```

- What does this print? Why?



Stack

main

a	42
b	100

swap

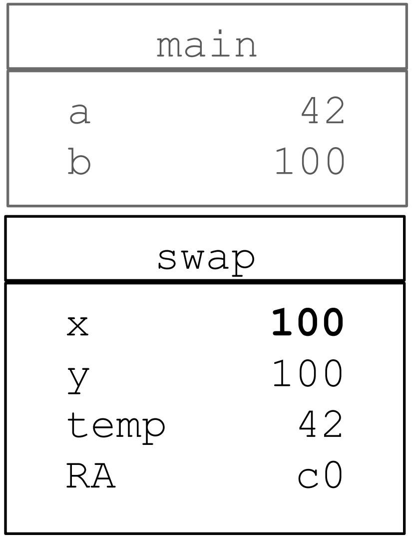
x	42
y	100
temp	**42**
RA	c0

Let’s do a little Java…

```
public class Example {
  public static void swap (int x, int y) {
    int temp = x;
    x = y;
    y = temp;
  }
  public static void main (String[] args) {
    int a = 42;
    int b = 100;
c0  swap (a, b);
    System.out.println(“a =“ + a + “ b = “ + b);
  }
}
```

- What does this print? Why?

Stack



main
a 42
b 100

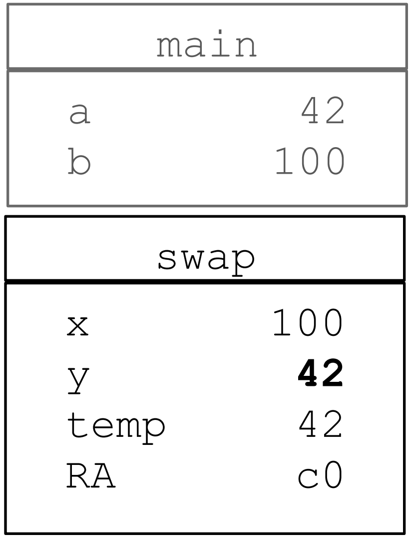
swap
x **100**
y 100
temp 42
RA c0

Let’s do a little Java…

```
public class Example {
  public static void swap (int x, int y) {
    int temp = x;
    x = y;
    y = temp;
  }
  public static void main (String[] args) {
    int a = 42;
    int b = 100;
    swap (a, b);
c0  System.out.println(“a =“ + a + “ b = “ + b);
  }
}
```

- What does this print? Why?

Stack



main

a 42
b 100

swap

x 100
y **42**
temp 42
RA c0

Let’s do a little Java…

```
public class Example {
  public static void swap (int x, int y) {
    int temp = x;
    x = y;
    y = temp;
  }
  public static void main (String[] args) {
    int a = 42;
    int b = 100;
    swap (a, b);
➡   System.out.println(“a =“ + a + “ b = “ + b);
  }
}
```

Stack

- What does this print? Why?

Let’s do some different Java…

```
public class Ex2 {
  int data;
  public Ex2 (int d) { data = d; }
  public static void swap (Ex2 x, Ex2 y) {
    int temp = x.data;
    x.data = y.data;
    y.data = temp;
  }
  public static void main (String[] args) {
    Example a = new Example (42);
    Example b = new Example (100);
    swap (a, b);
    System.out.println(“a =“ + a.data +
                       “ b = “ + b.data);
  }
}
```

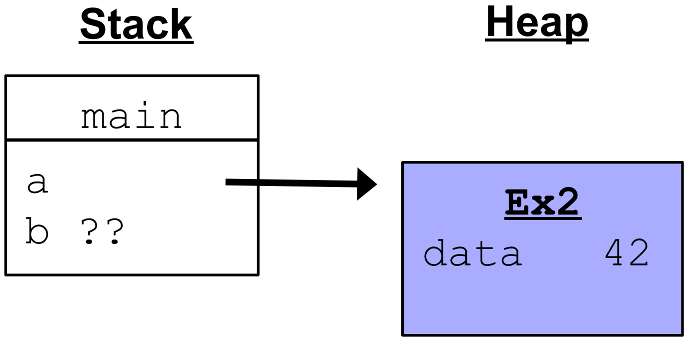
Stack

main
a ?? b ??

- What does this print? Why?

Let’s do some different Java…

```
public class Ex2 {
  int data;
  public Ex2 (int d) { data = d; }
  public static void swap (Ex2 x, Ex2 y) {
    int temp = x.data;
    x.data = y.data;
    y.data = temp;
  }
  public static void main (String[] args) {
    Example a = new Example (42);
    Example b = new Example (100);
    swap (a, b);
    System.out.println(“a =“ + a.data +
                          “ b = “ + b.data);
  }
}
```



Stack

Heap

main

a

b ??

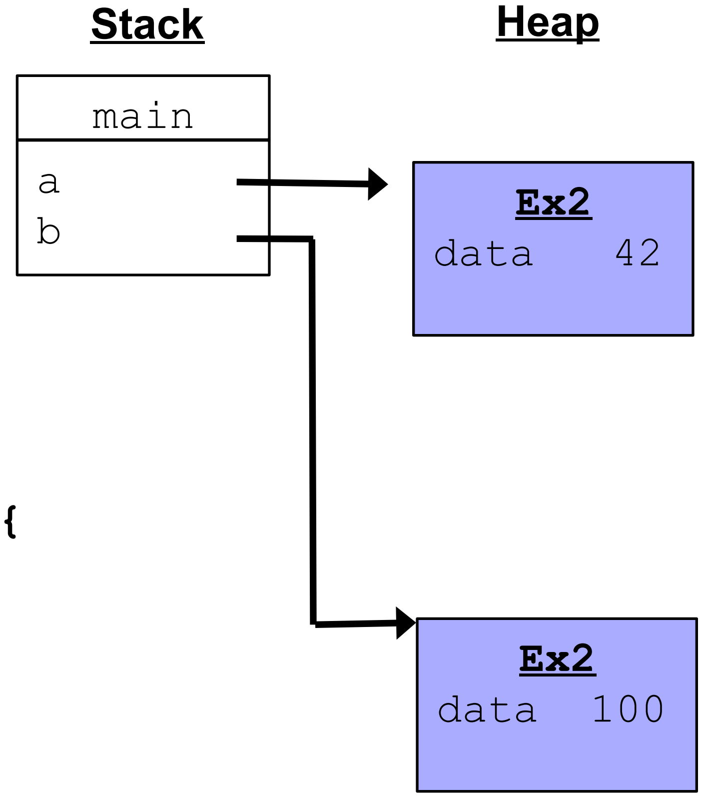
Ex2

data 42

- What does this print? Why?

Let’s do some different Java…

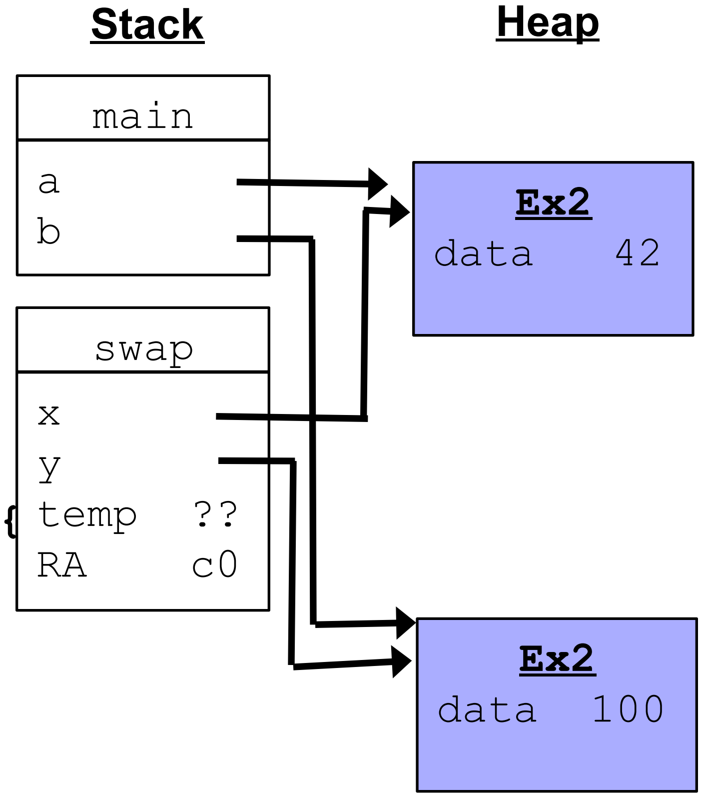
```
public class Ex2 {
  int data;
  public Ex2 (int d) { data = d; }
  public static void swap (Ex2 x, Ex2 y) {
    int temp = x.data;
    x.data = y.data;
    y.data = temp;
  }
  public static void main (String[] args) {
    Example a = new Example (42);
    Example b = new Example (100);
    swap (a, b);
    System.out.println(“a =“ + a.data +
                          “ b = “ + b.data);
  }
}
```



- What does this print? Why?

Let’s do some different Java…

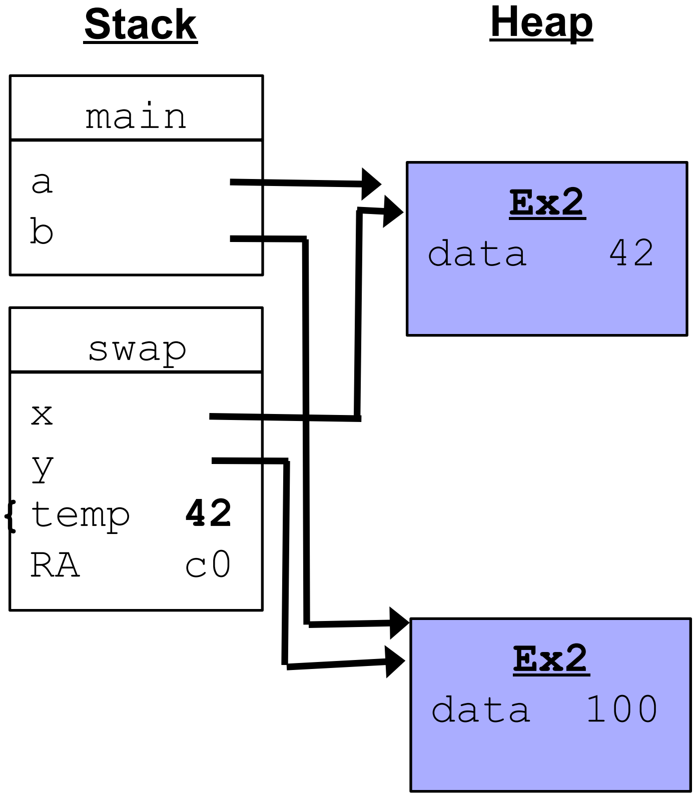
```
  public class Ex2 {
    int data;
    public Ex2 (int d) { data = d; }
    public static void swap (Ex2 x, Ex2 y) {
      int temp = x.data;
      x.data = y.data;
      y.data = temp;
    }
    public static void main (String[] args) {
      Example a = new Example (42);
      Example b = new Example (100);
c0    swap (a, b);
      System.out.println(“a =“ + a.data +
                               “ b = “ + b.data);
    }
  }
```



- What does this print? Why?

Let’s do some different Java…

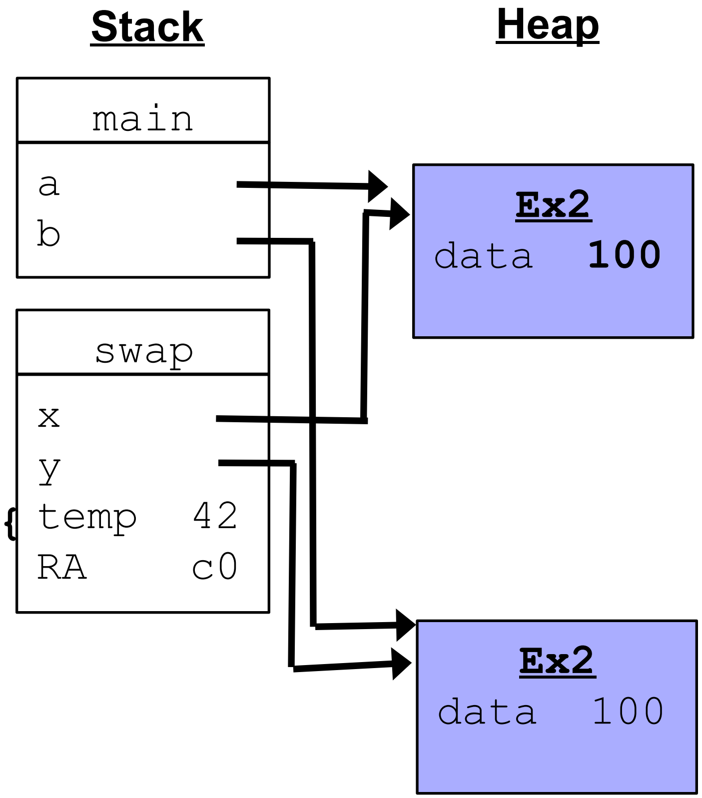
```
public class Ex2 {
  int data;
  public Ex2 (int d) { data = d; }
  public static void swap (Ex2 x, Ex2 y) {
    int temp = x.data;
    x.data = y.data;
    y.data = temp;
  }
  public static void main (String[] args) {
    Example a = new Example (42);
    Example b = new Example (100);
    swap (a, b);
c0  System.out.println(“a =“ + a.data +
                       “ b = “ + b.data);
  }
}
```



- What does this print? Why?

Let’s do some different Java…

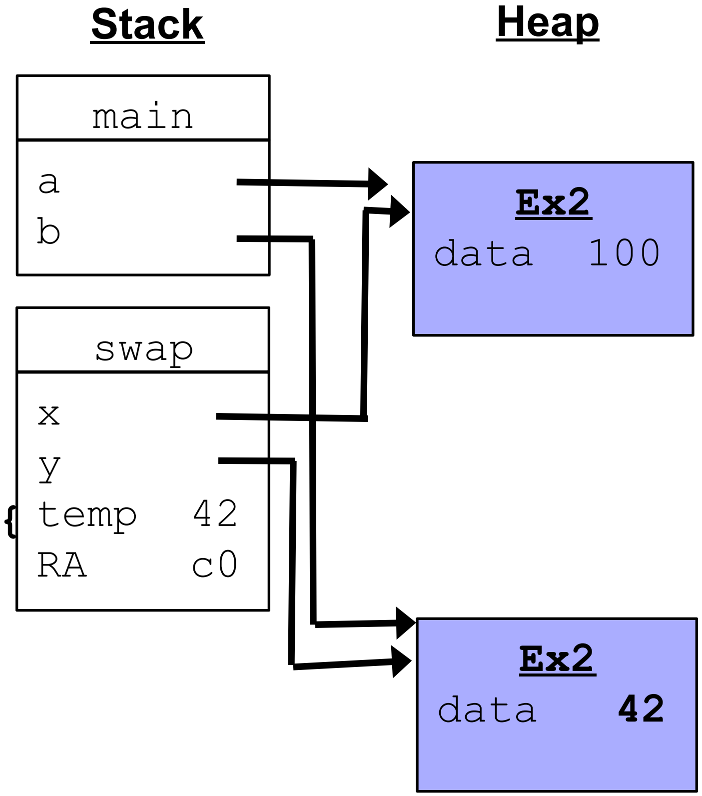
```
public class Ex2 {
   int data;
   public Ex2 (int d) { data = d; }
   public static void swap (Ex2 x, Ex2 y) {
      int temp = x.data;
      x.data = y.data;
      y.data = temp;
   }
   public static void main (String[] args) {
      Example a = new Example (42);
      Example b = new Example (100);
c0    swap (a, b);
      System.out.println(“a =“ + a.data +
                            “ b = “ + b.data);
   }
}
```



- What does this print? Why?

Let’s do some different Java…

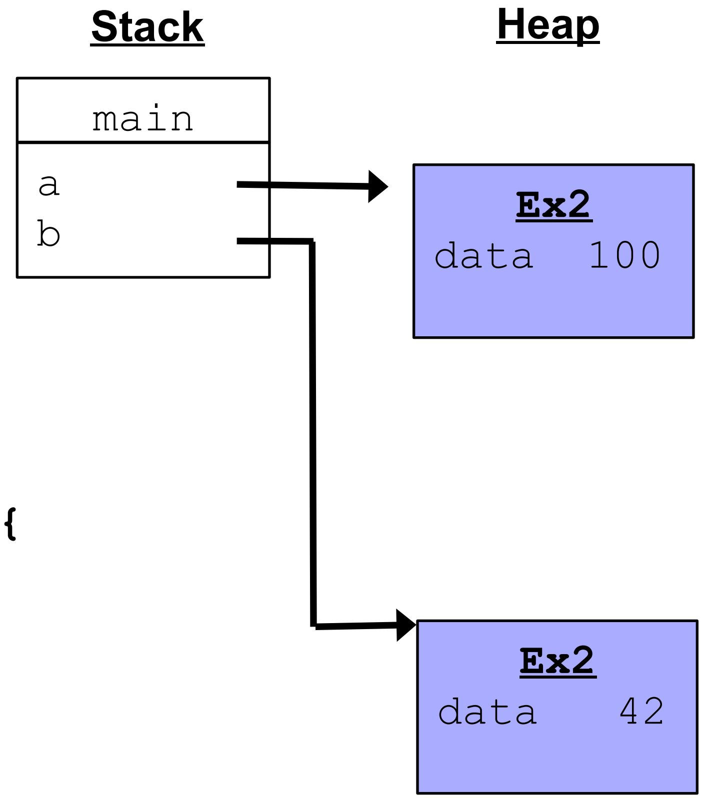
```
public class Ex2 {
   int data;
   public Ex2 (int d) { data = d; }
   public static void swap (Ex2 x, Ex2 y) {
      int temp = x.data;
      x.data = y.data;
      y.data = temp;
   }
   public static void main (String[] args) {
      Example a = new Example (42);
      Example b = new Example (100);
c0    swap (a, b);
      System.out.println(“a =“ + a.data +
                       “ b = “ + b.data);
   }
}
```



- What does this print? Why?

Let’s do some different Java…

```
public class Ex2 {
  int data;
  public Ex2 (int d) { data = d; }
  public static void swap (Ex2 x, Ex2 y) {
    int temp = x.data;
    x.data = y.data;
    y.data = temp;
  }
  public static void main (String[] args) {
    Example a = new Example (42);
    Example b = new Example (100);
    swap (a, b);
➡   System.out.println(“a =“ + a.data +
                          “ b = “ + b.data);
  }
}
```



- What does this print? Why?

References and Pointers

- Java has references:
  - Any variable of object type is a reference
  - Point at objects (which are all in the heap)
    - Under the hood: is the memory address of the object
  - Cannot explicitly manipulate them (*e.g.,* add 4)

References and Pointers (review)

- Java has references:
  - Any variable of object type is a reference
  - Point at objects (which are all in the heap)
    - Under the hood: is the memory address of the object
  - Cannot explicitly manipulate them (*e.g.,* add 4)

- Some languages (C,C++,assembly) have explicit pointers:
  - Hold the memory address of something
  - Can explicitly compute on them
  - Can de-reference the pointer (*ptr) to get thing-pointed-to
  - Can take the address-of (&x) to get something's address
  - Can do very unsafe things, shoot yourself in the foot

Pointers

- “address of” operator &
- don’t confuse with bitwise AND operator (&&)

<u>Given</u>

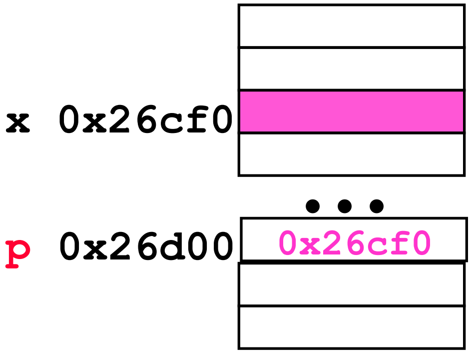
```
int x; int* p;  // p points to an int
p = &x;
```

<u>Then</u>

`*p = 2;  and x = 2; produce the same result`

`Note: p is a pointer, *p is an int`

- What happens when stating **`p = 2?`**



On 32-bit machine, p is 32 bits
On 64-bit machine, p is 64 bits

Back to Arrays

- Java:

```
int [] x = new int [nElems];
```

- C:

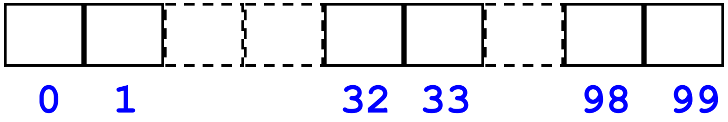
```
int data[42]; //if size is known constant
int* data = (int*) malloc (nElem * sizeof(int));
```

- `sizeof` tells how many bytes something takes
- `malloc` takes number of bytes
- `malloc` returns pointer to first allocated byte

Arrays, Pointers, and Address Calculation

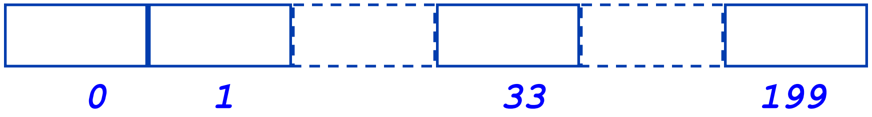
- x is a pointer, what is x+33?
- A pointer, but where?
    - what does calculation depend on?
- Result of adding an int to a pointer depends on size of object pointed to
- One reason why we tell compiler what type of pointer we have, even though all pointers are really the same thing (and same size)

```
int* a=
(int*)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)
```

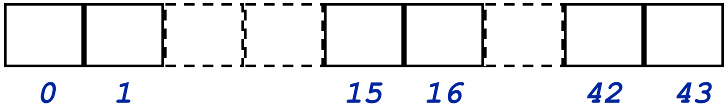
```
double* d =
(double*) malloc(200*sizeof(double));
```



```
*(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

- address one past the end of an array is ok for pointer comparison only



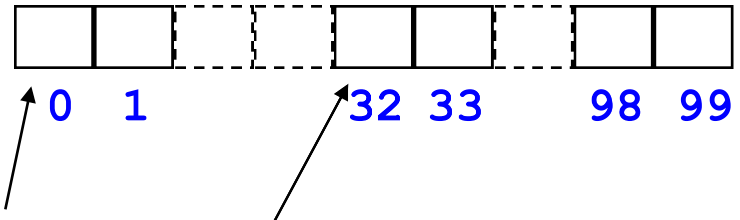
- what's at `*(begin+44)?`
- what does `begin++` mean?
- how are pointers compared using < and using == ?
- what is value of `end - begin`?

```
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

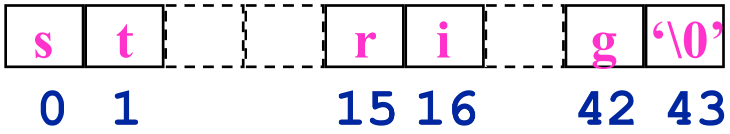
Array Example

```
#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])
}
```

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)

Strlen()

- `strlen()` returns the number of characters in a string
  - same as number elements in char array?

```
int strlen(char * s)
 // pre: ‘\0’ terminated
 // post: returns # chars
 {
      int count=0;
      while (*s++)
             count++;
      return count;
 }
```

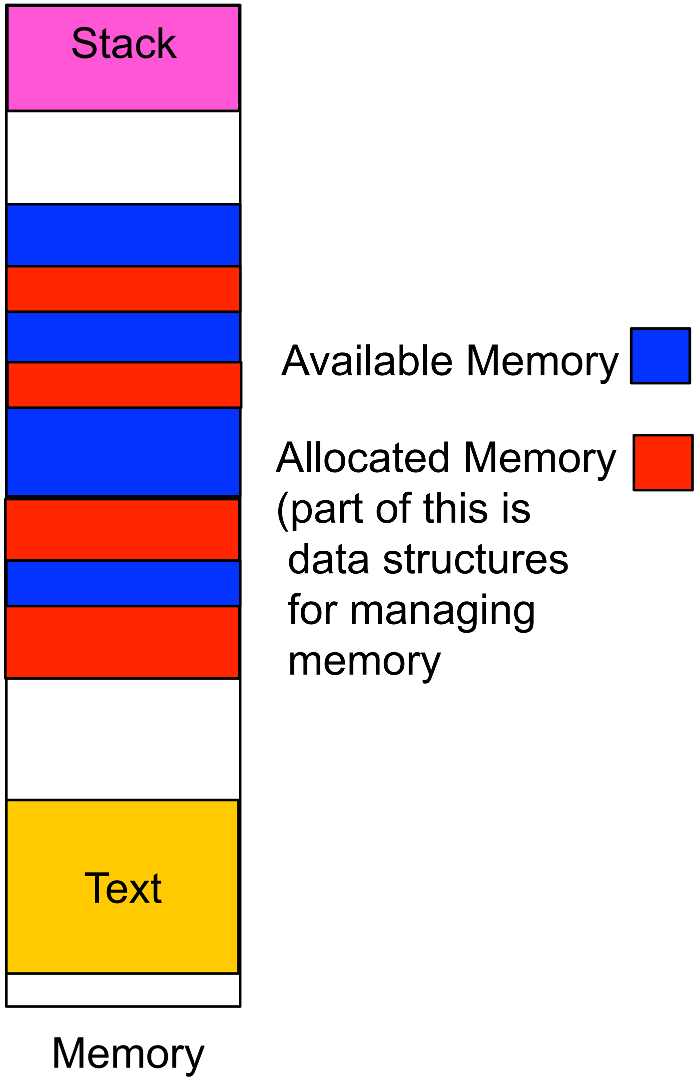
Vector Class vs. Arrays

- Vector Class
  - insulates programmers
  - array bounds checking
  - automagically growing/shrinking when more items are added/deleted

- How are Vectors implemented?
  - Arrays, re-allocated as needed
- Arrays can be more efficient

Memory Manager (Heap Manager)

- malloc() and free()
- Library routines that handle memory management (allocation, deallocation) for heap
- Java has garbage collection to reclaim memory of unreferenced objects
- C must use free, else memory leak



Available Memory 🟦

Allocated Memory 🟥 (part of this is data structures for managing memory)

Summary: From C to Binary

- Everything must be represented in binary!
- 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