

# **CSE 31**

# **Computer Organization**

**Lecture 10 – C Memory Management (cont.),  
Integer Representation**



# Announcement

## ▶ Labs

- Lab 4 due this week (with 7 days grace period after due date)
  - Demo is REQUIRED to receive full credit
- Lab 5 out **next** week

## ▶ Reading assignment

- Reading 03 (zyBooks 3.1 - 3.7, 3.9) due 11-OCT
  - Complete Participation Activities in each section to receive grade towards Participation
  - IMPORTANT: Make sure to submit score to CatCourses by using the link provided on CatCourses

# Announcement

## ▶ Homework assignment

- Homework 02 (zyBooks 2.1 - 2.9) due **tonight**, 04-OCT and Homework 03 (zyBooks 3.1 - 3.7, 3.9) due 18-OCT
  - Complete *Challenge Activities* in each section to receive grade towards Homework
  - IMPORTANT: Make sure to submit score to CatCourses by using the link provided on CatCourses

## ▶ Midterm 01 Exam details released on CatCourses

- See Announcement 23

# Automatic Memory Management (review)

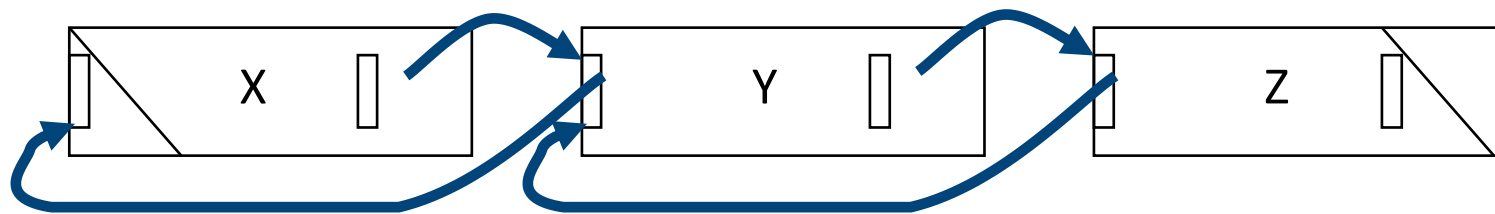
- ▶ Dynamically allocated memory is difficult to track
  - Why not track it **automatically**?
- ▶ If we can keep track of what memory is in use, we can reclaim everything else.
  - Unreachable memory is called **garbage**, the process of reclaiming it is called **garbage collection**.
- ▶ So how do we track what is in use?

# Scheme 1: Reference Counting (review)

- ▶ For every chunk of dynamically allocated memory, keep a count of number of pointers that point to it.
- ▶ When the count reaches 0, reclaim the memory.
- ▶ Simple assignment statements can result in a lot of work, since may update reference counts of many items

# Reference Counting Flaws

- ▶ Extra overhead added to assignments, as well as ending a block of code.
- ▶ Does not work for circular structures!
  - E.g., doubly linked list:



# Scheme 2: Mark and Sweep Garbage Collection

- ▶ Keep allocating new memory until memory is exhausted, then try to find unused memory.
- ▶ Consider objects in a graph, chunks of memory (objects) are graph nodes, pointers to memory are graph edges.
  - Edge from A to B → A stores pointer to B
- ▶ Can start with the root set, perform a graph traversal, find all usable memory!
- ▶ 2 Phases:
  1. Mark used nodes
  2. Sweep free ones, returning list of free nodes

# Mark and Sweep

- ▶ Graph traversal is relatively easy to implement recursively

```
void traverse(struct graph_node *node) {  
    /* visit this node */  
    foreach child in node->children {  
        traverse(child);  
    }  
}
```

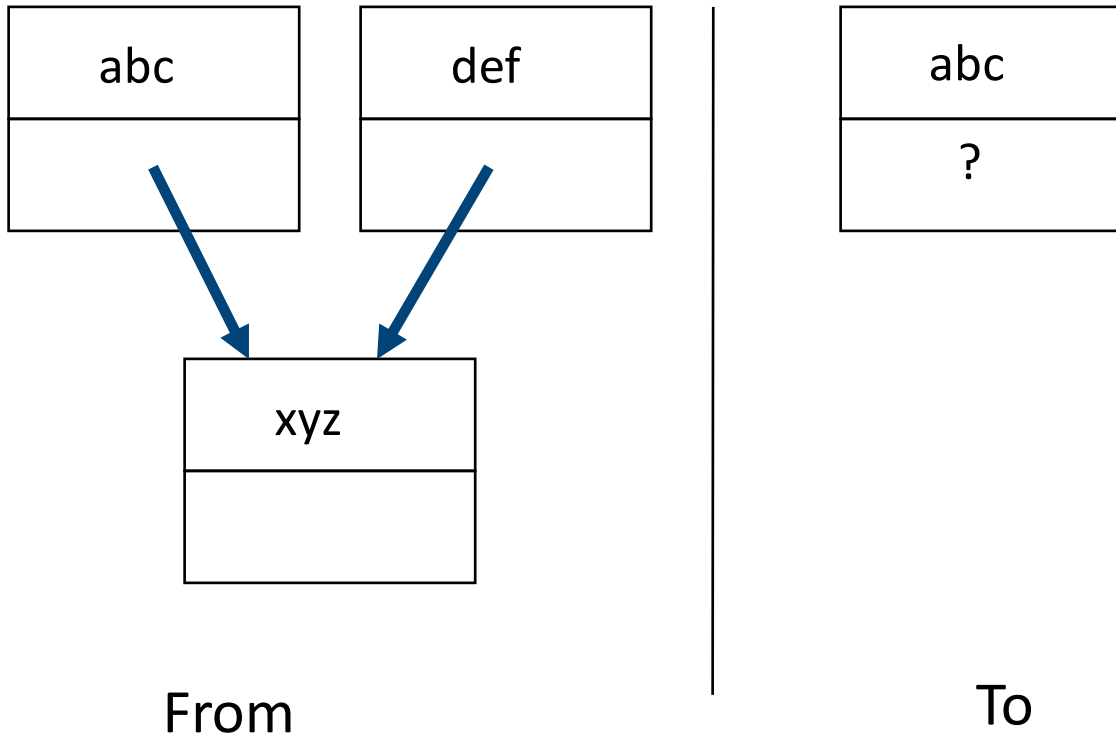
- ▶ But with recursion, state is stored on the execution stack.
  - Garbage collection is invoked when not much memory left
  - ...Oops!
- ▶ As before, we could traverse in constant space (by reversing pointers)



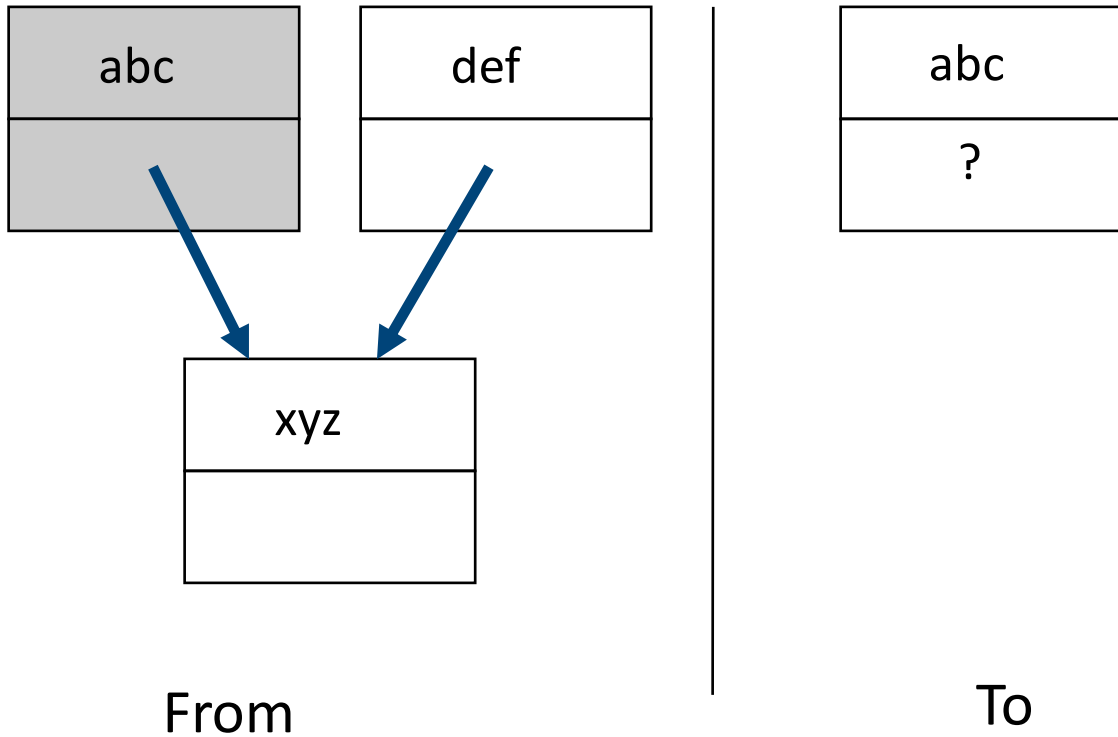
# Scheme 3: Copying Garbage Collection

- ▶ Divide memory into two spaces, only one in use at any time.
- ▶ When active space is exhausted, traverse the active space, copying all objects to the other space, then make the new space active and continue.
  - Only reachable objects are copied!
- ▶ Use “forwarding pointers” to keep consistency
  - Simple solution to avoiding having to have a table of old and new addresses, and to mark objects already copied

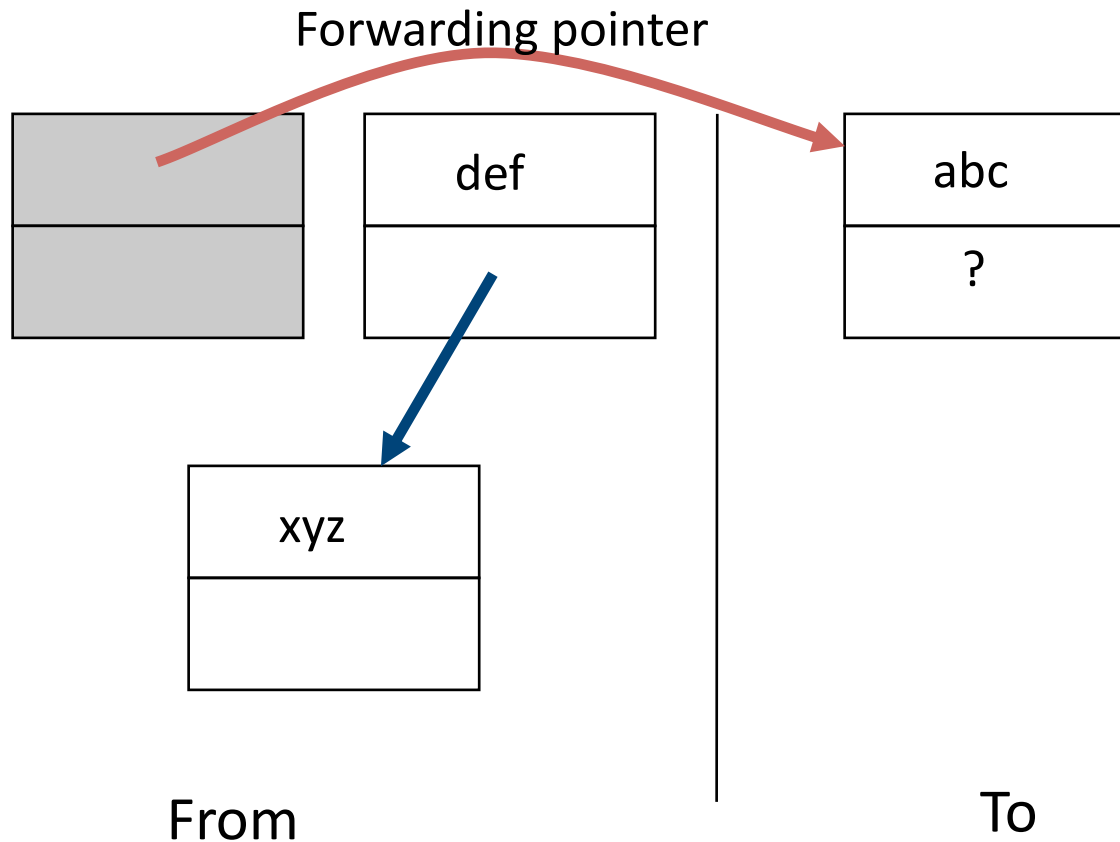
# Forwarding Pointers: 1<sup>st</sup> copy “abc”



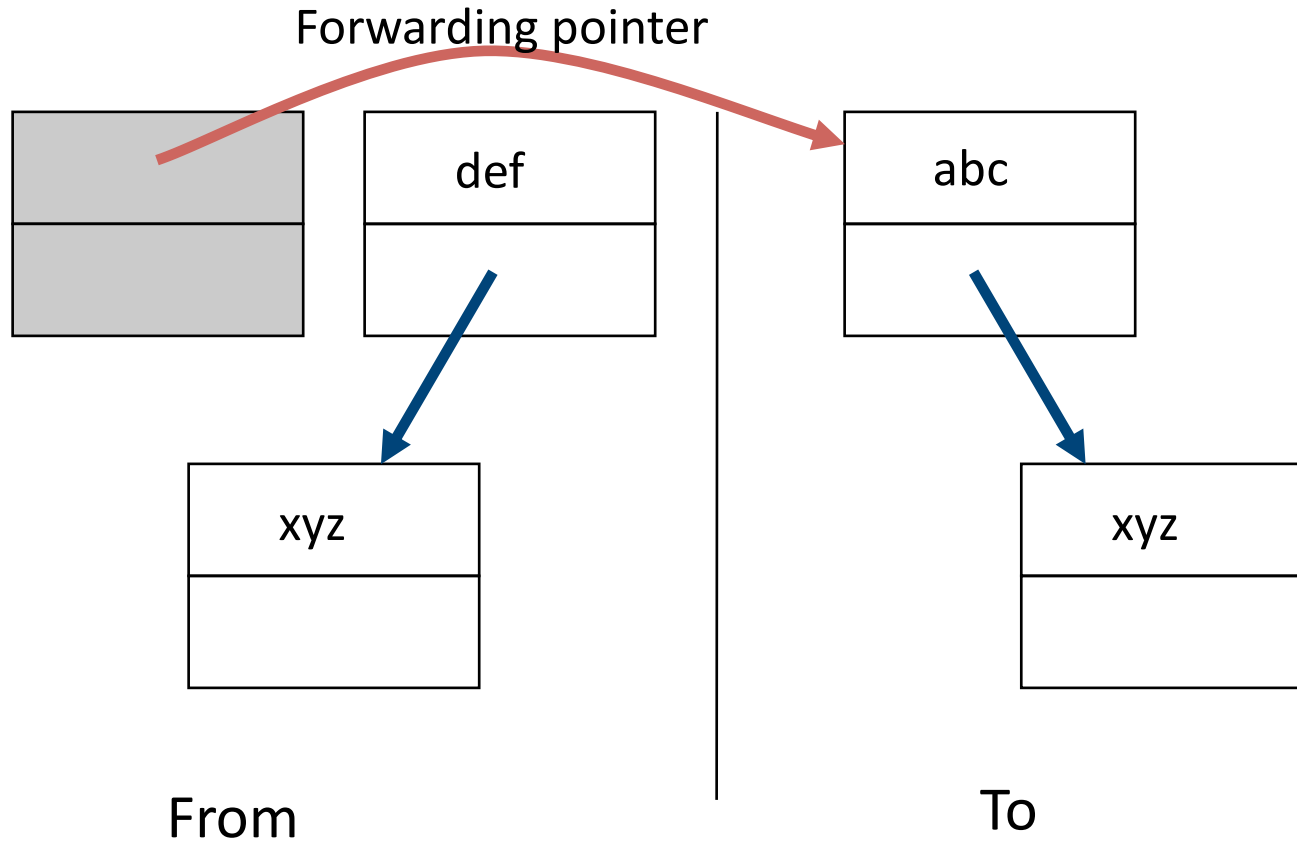
# Forwarding Pointers: leave ptr to new abc



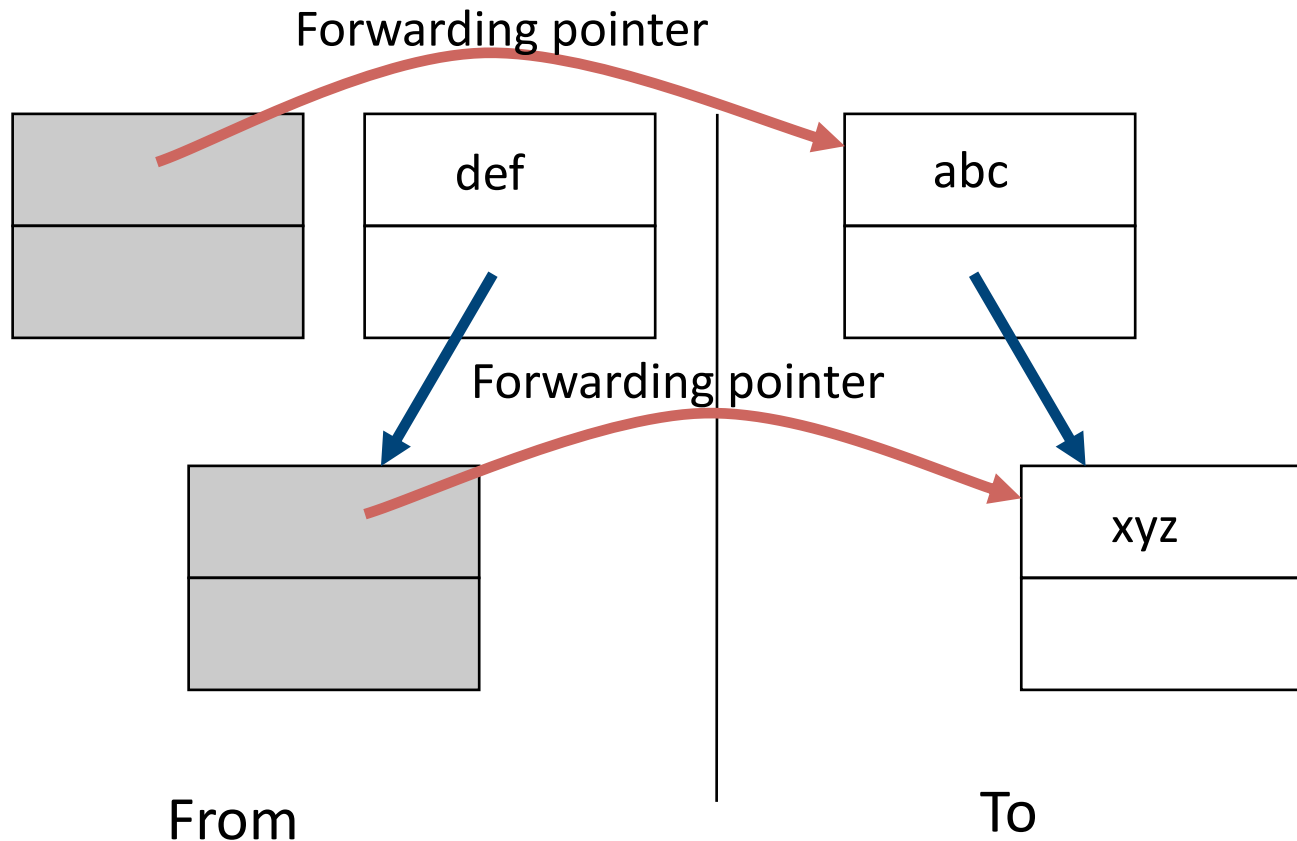
# Forwarding Pointers : now copy “xyz”



# Forwarding Pointers: leave ptr to new xyz

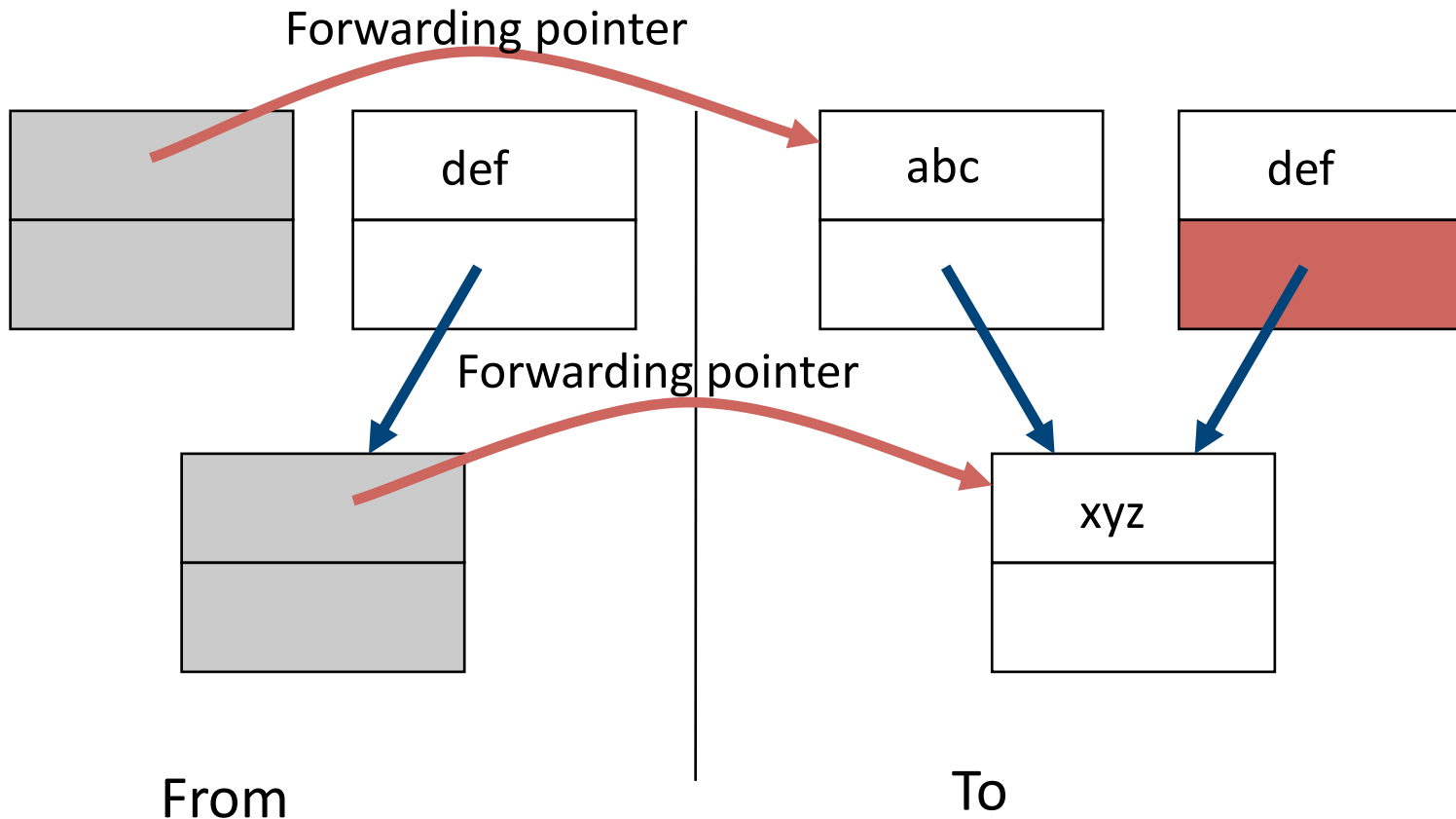


# Forwarding Pointers: now copy “def”



*Since xyz was already copied,  
def uses xyz's forwarding pointer  
to find its new location*

# Forwarding Pointers



*Since xyz was already copied,  
def uses xyz's forwarding pointer  
to find its new location*

# Summary

- ▶ Several techniques for managing heap via malloc and free: best-, first-, next-fit
  - 2 types of memory fragmentation: internal & external; all suffer from some kind of frag.
  - Each technique has strengths and weaknesses, none is definitively best
- ▶ Automatic memory management relieves programmer from managing memory.
  - All require help from language and compiler
  - **Reference Count**: not for circular structures
  - **Mark and Sweep**: complicated and slow, works
  - **Copying**: Divides memory to copy good stuff

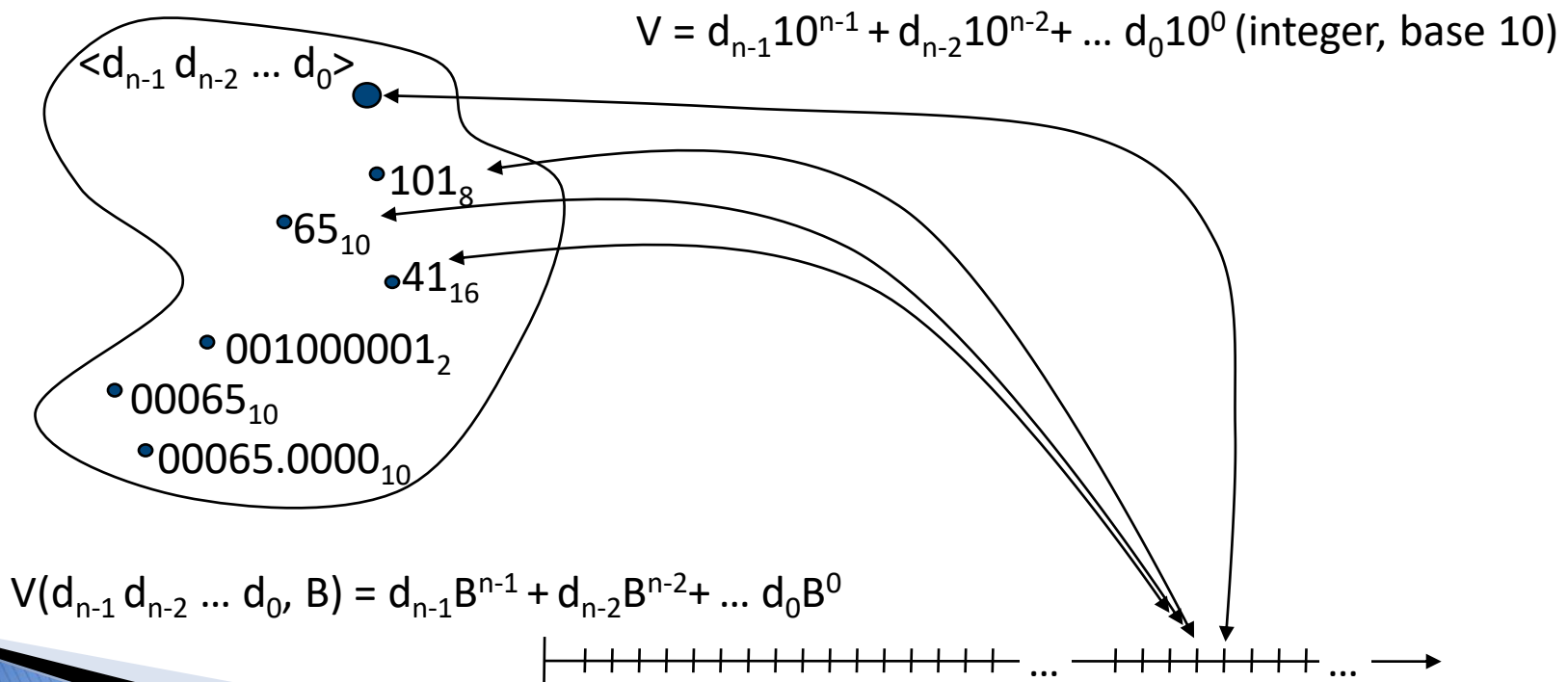


# Number Representations

- ▶ What do these numbers mean?
  - 101
  - 0101
- ▶ Depends on what representation!

# Representation and Meaning

- ▶ Objects are represented as collections of symbols (bits, digits)
- ▶ Their meaning is derived from what you do with them.



# Representation (how many bits?)

## ▶ Characters?

- 26 letters → 5 bits ( $2^5 = 32$ )
- upper/lower case + punctuation → 7 bits (in 8 bits) ("ASCII")
- standard code to cover all the world's languages → 8,16,32 bits ("Unicode") [www.unicode.com](http://www.unicode.com)



## ▶ Logical values?

- 0 → False, 1 → True

## ▶ Colors?

Ex: *Red (00)*

*Green (01)*

*Blue (11)*

## ▶ Remember: N bits → at most $2^N$ things

# How many bits to represent $\pi$ ?

- a) 1
- b) 9 ( $\pi = 3.14$ , so that's 011 . 001 100)
- c) 64 (Since modern computers are 64-bit machines)
- d) Every bit the machine has!
- e)  $\infty$

We are going to learn how to represent floating point numbers later!

# What to do with representations of numbers?

## ► Just what we do with numbers!

- Add them
- Subtract them
- Multiply them
- Divide them
- Compare them

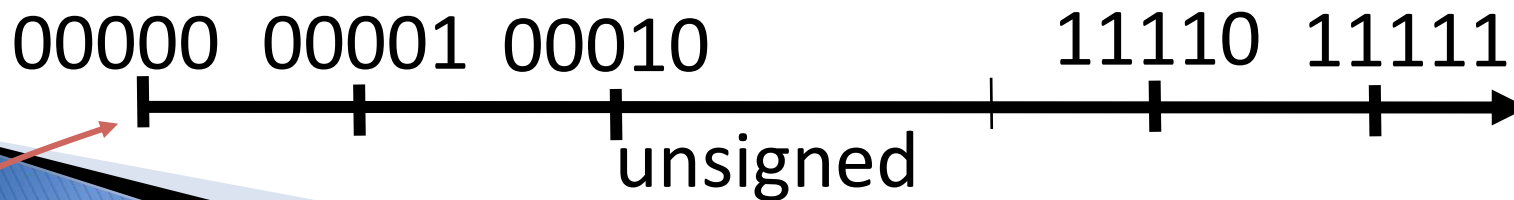
## ► Example: $10 + 7 = 17$

- ...so simple to add in binary that we can build circuits to do it!
- subtraction just as you would in decimal
- Comparison: How do you tell if  $X > Y$  ?

$$\begin{array}{rcccc} & 1 & 1 & & \\ & 1 & 0 & 1 & 0 \\ + & 0 & 1 & 1 & 1 \\ \hline 1 & 0 & 0 & 0 & 1 \end{array}$$

# What if too big?

- ▶ Binary bit patterns are simply representatives of numbers. Strictly speaking they are called “numerals”
- ▶ Numbers really have an  $\infty$  number of digits
  - with almost all being same (00...0 or 11...1) except for a few of the rightmost digits
  - Just don't normally show leading digits
- ▶ If result of add (or -, \*, / ) cannot be represented by these rightmost HW bits, **overflow** is said to have occurred.



# Negative Numbers

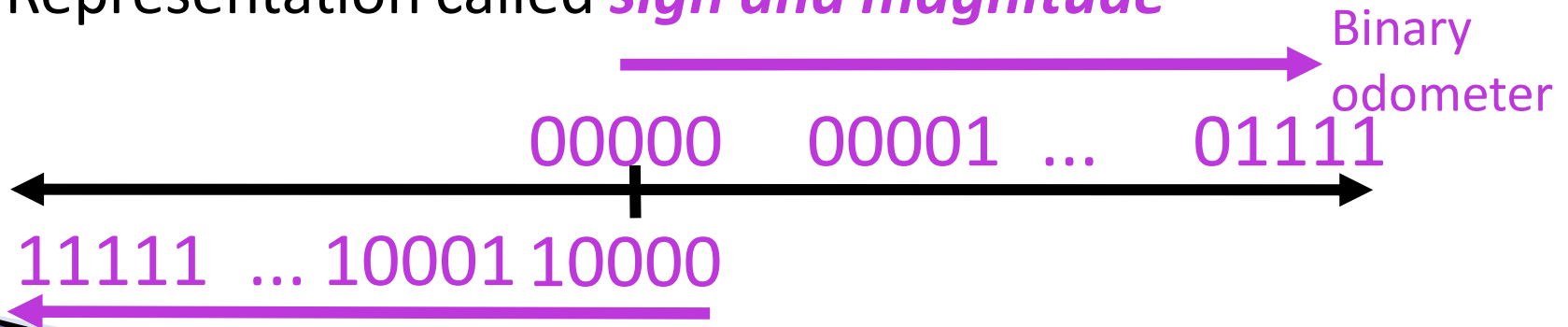
- ▶ So far, *unsigned numbers*



- ▶ Obvious solution: define leftmost bit to be sign!

- $0 \rightarrow +$  ,  $1 \rightarrow -$
- Rest of bits can be numerical value of number

- ▶ Representation called *sign and magnitude*



# Shortcomings of Sign Magnitude?

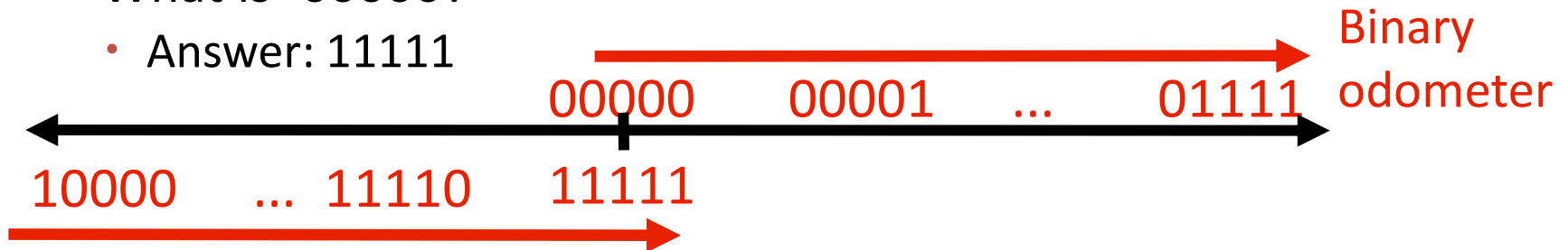
- ▶ Arithmetic circuit complicated
  - Special steps depending whether signs are the same or not
- ▶ Also, **two zeros**
  - $0x00000000 = +0_{\text{ten}}$
  - $0x80000000 = -0_{\text{ten}}$
  - What would two 0s mean for programming?
- ▶ Also, incrementing “binary odometer”, sometimes increases values, and sometimes decreases!
- ▶ Therefore sign and magnitude abandoned



# Another try

## ► Complement the bits

- Example:  $7_{10} = 00111_2$     $-7_{10} = 11000_2$
- Called **One's Complement**
- Note: positive numbers have leading 0s, negative numbers have leading 1s.
- What is -00000?
  - Answer: 11111



- How many positive numbers in N bits?  $2^{N-1}$
- How many negative numbers?  $2^{N-1}$

# Shortcomings of One's complement?

- ▶ Arithmetic is less complicate than sign & magnitude.
- ▶ Still two zeros
  - $0x00000000 = +0_{\text{ten}}$
  - $0xFFFFFFFF = -0_{\text{ten}}$
- ▶ Although used for a while on some computer products, one's complement was eventually abandoned because another solution was better.

# Standard Negative # Representation

- ▶ Problem is the negative mappings “overlap” with the positive ones (the two 0s). Want to shift the negative mappings left by one.
  - Solution! For negative numbers, complement, then add 1 to the result
- ▶ As with sign and magnitude, & one’s complement, leading 0s → positive, leading 1s → negative
  - 000000...xxx is  $\geq 0$ , 111111...xxx is  $< 0$
  - except 1...1111 is -1, not -0
- ▶ This representation is **Two’s Complement**
- ▶ This makes the hardware simple!

In C: short, int, long, intN\_t (C99) are all signed integers.

# Two's Complement Formula

- ▶ Can represent positive and negative numbers in terms of the bit value times a power of 2:

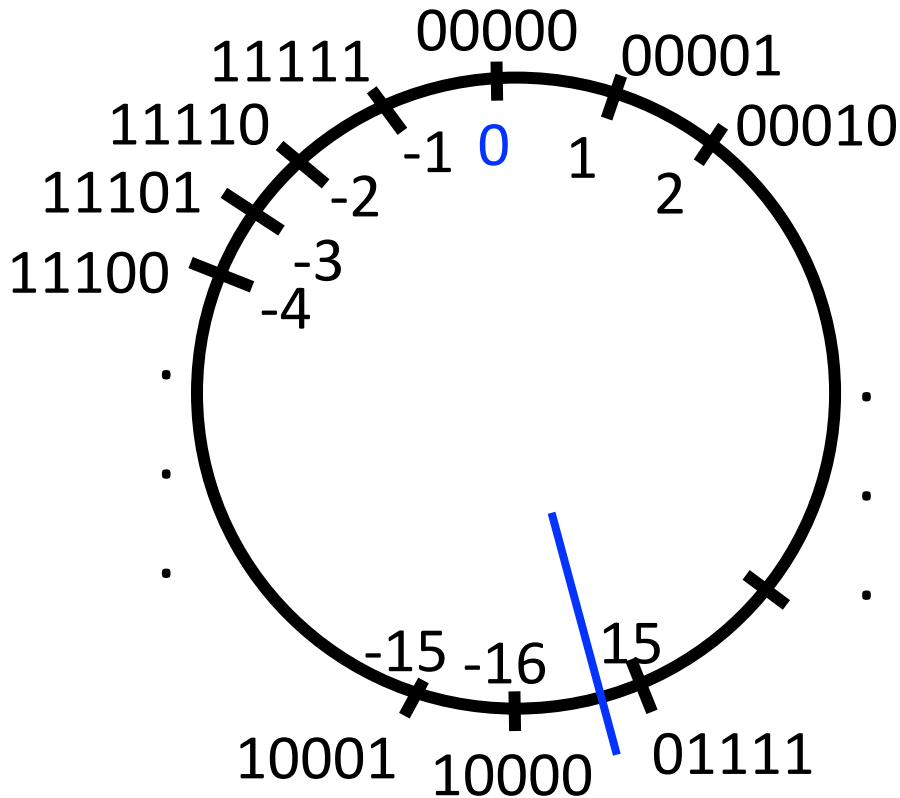
$$d_{31} * -(2^{31}) + d_{30} * 2^{30} + \dots + d_2 * 2^2 + d_1 * 2^1 + d_0 * 2^0$$

- ▶ Example:  $1101_{\text{two}}$   
 $= 1x - (2^3) + 1 * 2^2 + 0 * 2^1 + 1 * 2^0$   
 $= -2^3 + 2^2 + 0 + 2^0$   
 $= -8 + 4 + 0 + 1$   
 $= -8 + 5$   
 $= -3_{\text{ten}}$

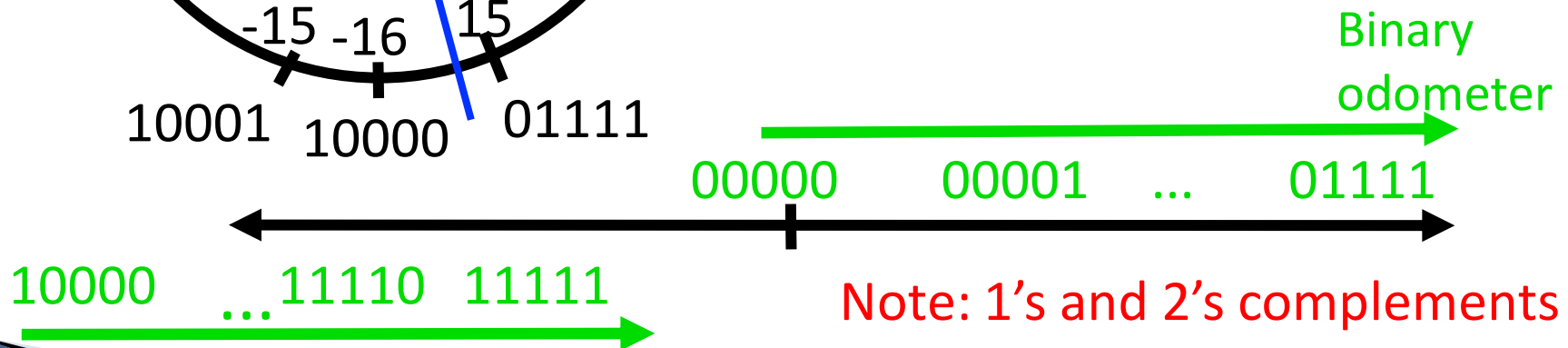
Example: -3 to +3 to -3:

x :	$1101_{\text{two}}$	(-3)
x' :	$0010_{\text{two}}$	
+1 :	$0011_{\text{two}}$	(3)
()' :	$1100_{\text{two}}$	
+1 :	$1101_{\text{two}}$	(-3)

# 2's Complement Number "line": N = 5

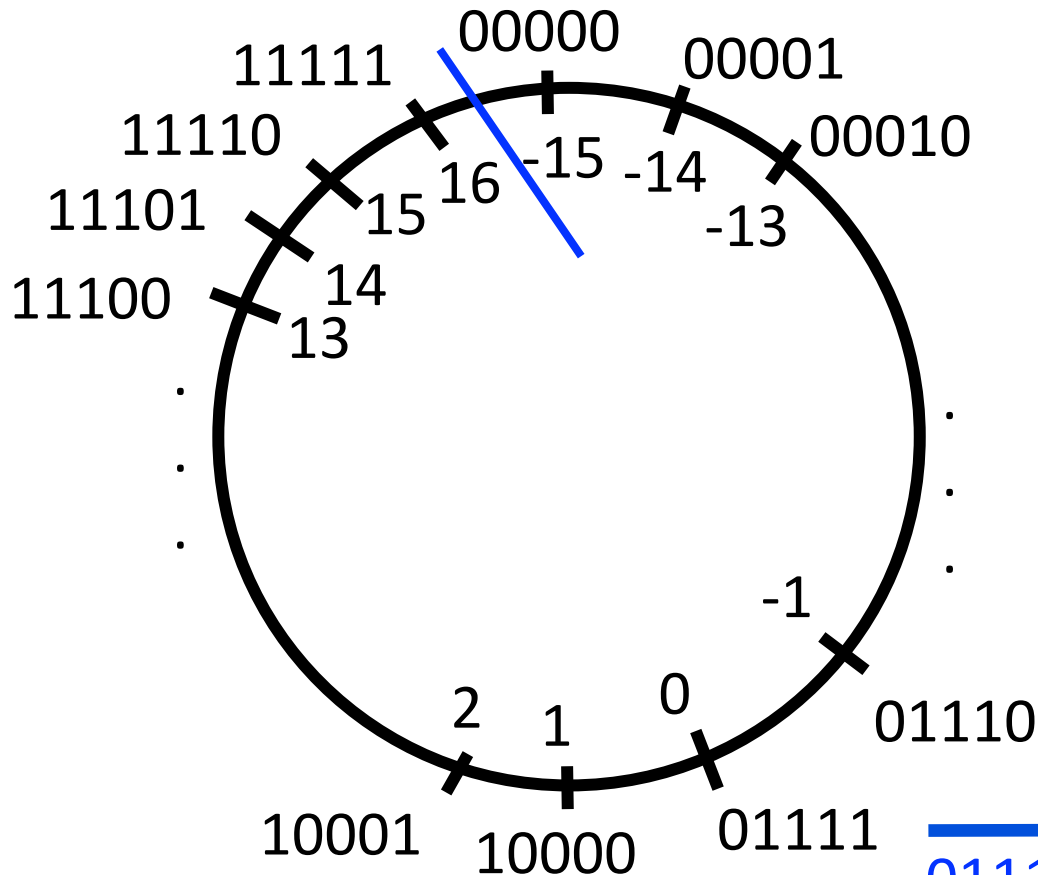


- ▶  $2^{N-1}$  non-negatives
- ▶  $2^{N-1}$  negatives
- ▶ **one zero**
- ▶ how many positives?
  - $2^{N-1} - 1$

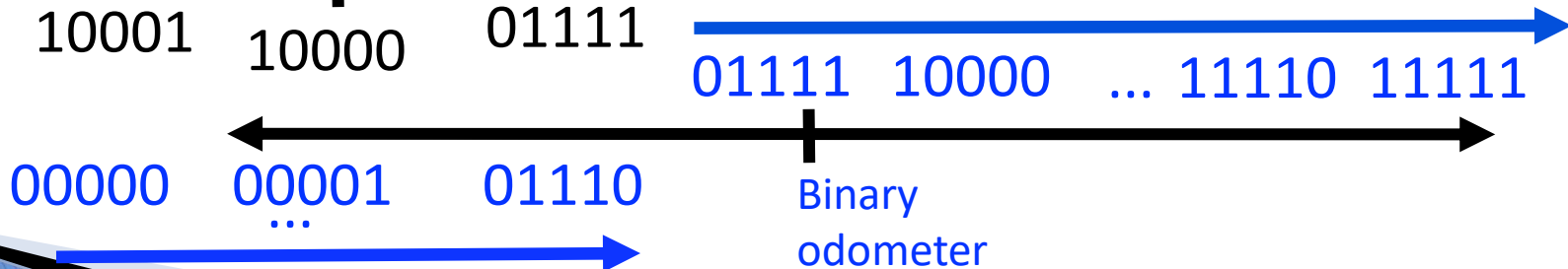


Note: 1's and 2's complements are used to represent negative numbers only!

# Bias Encoding: N = 5 (bias = 15)



- ▶ Want 00... to represent the smallest number
- ▶ value = unsigned - bias
- ▶ Bias for N bits =  $2^{N-1} - 1$
- ▶ one zero
- ▶ how many positives?
  - $2^{N-1}$
  - (more than 2's complement)



# Summary

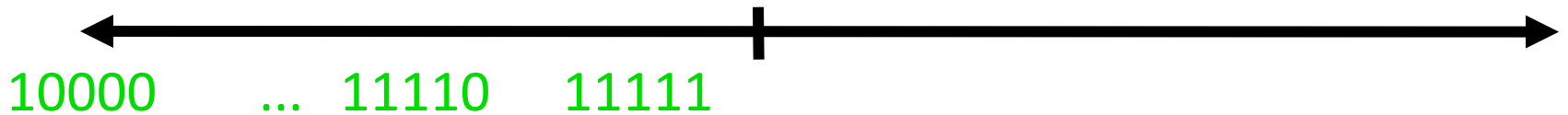
- ▶ We represent “things” in computers as particular bit patterns:
  - $N$  bits  $\rightarrow 2^N$  things
- ▶ Different integer encodings have different benefits; 1s complement and sign/mag have most problems.
- ▶ **unsigned** (C99's `uintN_t`):

00000      00001      ...      01111      10000      ...      11111



- ▶ **2's complement** (C99's `intN_t`): universal, learn it!

00000      00001      ...      01111



- ▶ Overflow: numbers  $\infty$ ; computers finite  $\rightarrow$  errors!