CSE 31
Computer Organization

Lecture 9 – C Memory Management (cont.), Integer Representation

Announcement

- Labs
  - Lab 3 due this week (with 7 days grace period after due date)
    - Demo is REQUIRED to receive full credit
  - Lab 4 out this week
    - Due at 11:59pm on the same day of your next lab
    - You must demo your submission to your TA within 14 days
- 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

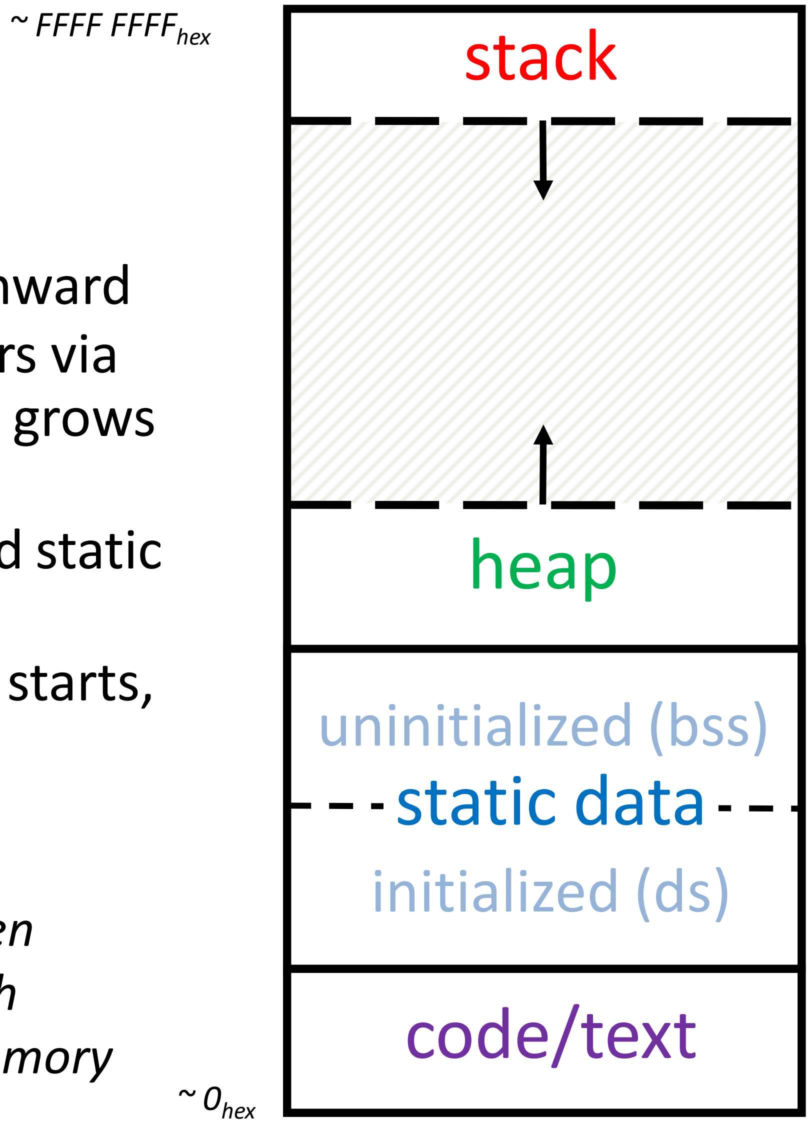
Announcement

- Homework assignment
  - Homework 02 (zyBooks 2.1 - 2.9) due 04-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

Normal C Memory Management (review)

- A program's address space contains 4 regions:
  - stack: local variables, grows downward
  - heap: space requested for pointers via `malloc()` ; resizes dynamically, grows upward
  - static data: Initialized/uninitialized static and global variables
  - code/text: loaded when program starts, does not change

For now, OS somehow prevents accesses between stack and heap (gray hash lines). Wait for virtual memory



K&R Malloc/Free Implementation (review)

- Each block of memory is preceded by a header that has two fields:
  - size of the block
  - a pointer to the next block
- All free blocks are kept in a circular linked list, the pointer field is unused in an allocated block
- `malloc()` searches the free list for a block that is big enough. If none is found, more memory is requested from the operating system. If what it gets can't satisfy the request, it fails.
- `free()` checks if the blocks adjacent to the freed block are also free
  - If so, adjacent free blocks are merged (coalesced) into a single, larger free block
  - Otherwise, the freed block is just added to the free list

Choosing a block in `malloc()`

- If there are multiple free blocks of memory that are big enough for some request, how do we choose which one to use?
  - best-fit: choose the smallest block that is big enough for the request
  - first-fit: choose the first block we see that is big enough
  - next-fit: like first-fit but remember where we finished searching and resume searching from there

Tradeoffs of allocation policies

- Best-fit: Tries to limit fragmentation but at the cost of time (must examine all free blocks for each malloc).
  - Leaves lots of small blocks (why?)
- First-fit: Quicker than best-fit (why?) but potentially more fragmentation.
  - Tends to concentrate small blocks at the beginning of the free list (why?)
- Next-fit: Does not concentrate small blocks at front like first-fit, should be faster as a result.

Quiz – Pros and Cons of fits

1) first-fit results in many small blocks at the beginning of the free list
2) next-fit is slower than first-fit, since it takes longer in steady state to find a match
3) best-fit leaves lots of tiny blocks

	123
a)	FFT
b)	FTT
c)	TFF
d)	TFT
e)	TTT

Quiz – Pros and Cons of fits

1) first-fit results in many small blocks at the beginning of the free list
2) next-fit is slower than first-fit, since it takes longer in steady state to find a match
3) best-fit leaves lots of tiny blocks

	123
a)	FFT
b)	FTT
c)	TFF
d)	**TFT**
e)	TTT

Summary

- C has 3 pools of memory
  - Static storage: global and static variable storage, basically permanent, entire program run
  - The Stack: local variable storage, parameters, return address
  - The Heap (dynamic storage): `malloc()` grabs space from here, `free()` returns it.

- `malloc()` handles free space with freelist. Three different ways to find free space when given a request:
  - First fit (find first one that's free)
  - Next fit (same as first, but remembers where left off)
  - Best fit (finds most "snug" free space)

Slab Allocator

- A different approach to memory management (used in GNU `libc`)
- Divide blocks in to “large” and “small” by picking an arbitrary threshold size (say 128kB). Blocks larger than this threshold are managed with a **freelist** (as before).
- For small blocks, allocate blocks in sizes that are powers of 2
  - e.g., if program wants to allocate 20 bytes, actually give it 32 bytes

Slab Allocator

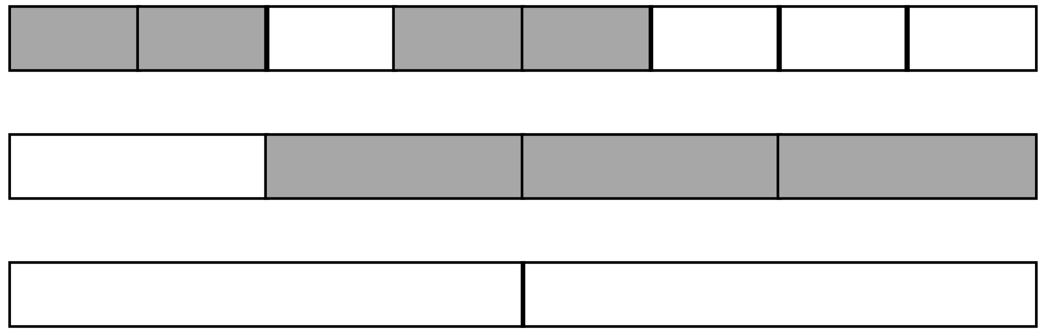
- Bookkeeping for small blocks is relatively easy
  - Use a bitmap for each range of blocks of the same size
- Allocating is easy and fast
  - Compute the size of the block to allocate and find a free bit in the corresponding bitmap.
- Freeing is also easy and fast
  - Figure out which slab the address belongs to and clear the corresponding bit.

Slab Allocator

16 byte blocks:

32 byte blocks:

64 byte blocks:



16 byte block bitmap: 11011000

32 byte block bitmap: 0111

64 byte block bitmap: 00

Slab Allocator Tradeoffs

- Extremely fast for small blocks.
- Slower for large blocks
  - But presumably the program will take more time to do something with a large block, so the overhead is not as critical.
- Minimal space overhead
- No external fragmentation (as we defined it before)
  - For small blocks, but still have wasted space!

Internal vs. External Fragmentation

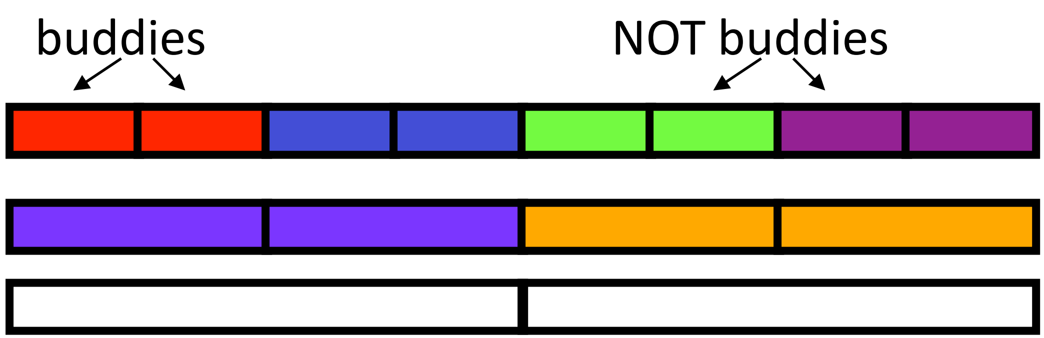
- With the slab allocator, difference between requested size and next power of 2 is wasted
  - e.g., if program wants to allocate 20 bytes and we give it a 32 byte block, 12 bytes are unused.
- We also refer to this as fragmentation, but call it **internal fragmentation** since the wasted space is actually within an allocated block.
- **External fragmentation**: wasted space between allocated blocks.

Buddy System

- Yet another memory management technique (used in Linux kernel)
- Like GNU’s “slab allocator”, but only allocate blocks in sizes that are powers of 2 (internal fragmentation is possible)
- Keep separate free lists for each size
  - e.g., separate free lists for 16 byte, 32 byte, 64 byte blocks, etc.

Buddy System

- If no free block of size `n` is available, find a block of size `2n` and split it in to two blocks of size `n`
- When a block of size `n` is freed, if its neighbor of size `n` is also free, combine the blocks into a single block of size `2n`
  - Buddy is a block in other half larger block



- Same speed advantages as slab allocator

Allocation Schemes

- So which memory management scheme (K&R, slab, buddy) is best?
  - There is no single best approach for every application.
  - Different applications have different allocation / deallocation patterns.
  - A scheme that works well for one application may work poorly for another application.

Automatic Memory Management

- 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?

Tracking Memory Usage

- Techniques depend heavily on the programming language and rely on help from the compiler.
- Start with all pointers in global variables and local variables (root set).
- Recursively examine dynamically allocated objects we see a pointer to.
  - We can do this in constant space by reversing the pointers on the way down
- How do we recursively find pointers in dynamically allocated memory?

Tracking Memory Usage

- Again, it depends heavily on the programming language and compiler.
- Could have only a single type of dynamically allocated object in memory
  - E.g., simple Lisp/Scheme system with only `cons` cells
- Could use a strongly typed language (e.g., Java)
  - Don't allow conversion (casting) between arbitrary types.
  - C/C++ are not strongly typed.
- We will cover 3 schemes to collect garbage

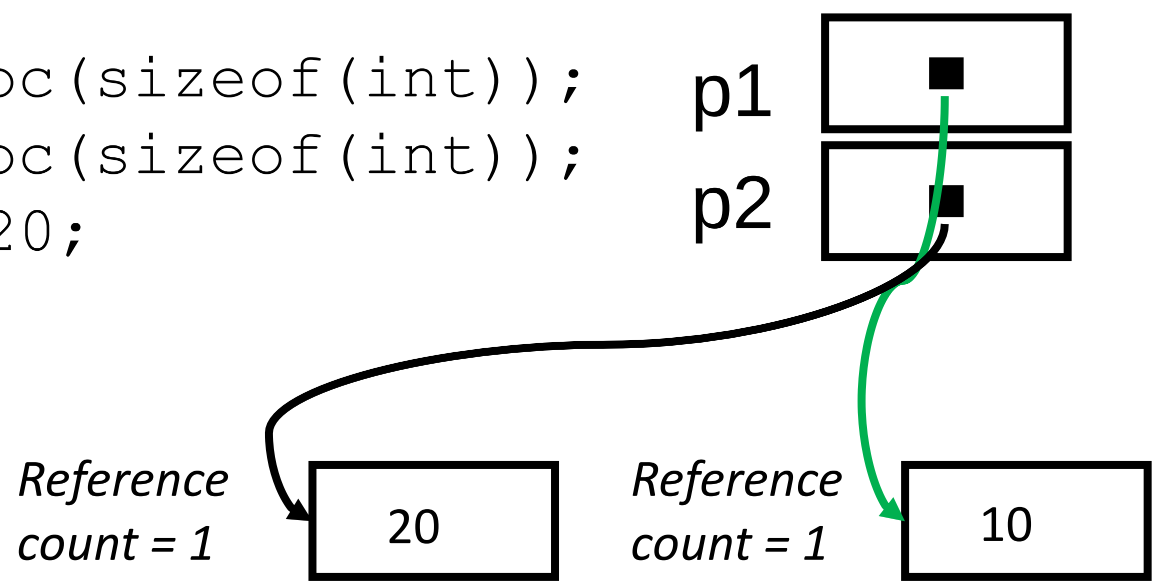
Scheme 1: Reference Counting

- 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 Example

- For every chunk of dynamically allocated memory, keep a count of number of pointers that point to it.
  - When the count reaches 0, reclaim.

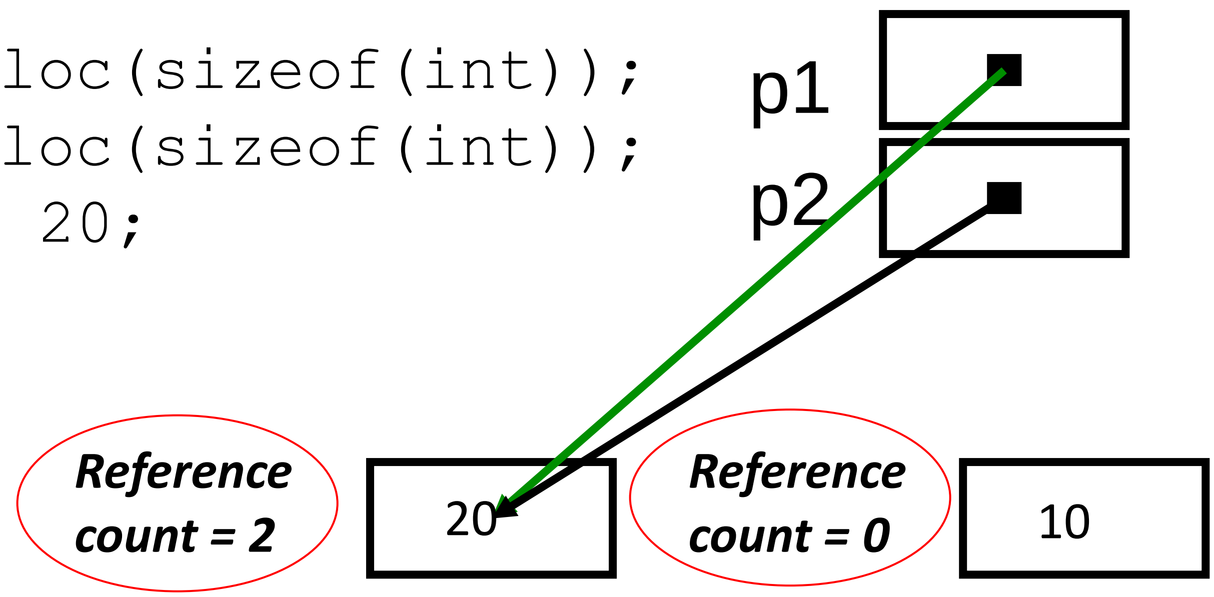
```
int *p1, *p2;
p1 = (int *)malloc(sizeof(int));
p2 = (int *)malloc(sizeof(int));
*p1 = 10; *p2 = 20;
```



Reference Counting Example

- For every chunk of dynamically allocated memory, keep a count of number of pointers that point to it.
  - When the count reaches 0, reclaim.

```
int *p1, *p2;
p1 = (int *)malloc(sizeof(int));
p2 = (int *)malloc(sizeof(int));
*p1 = 10; *p2 = 20;
p1 = p2;
```



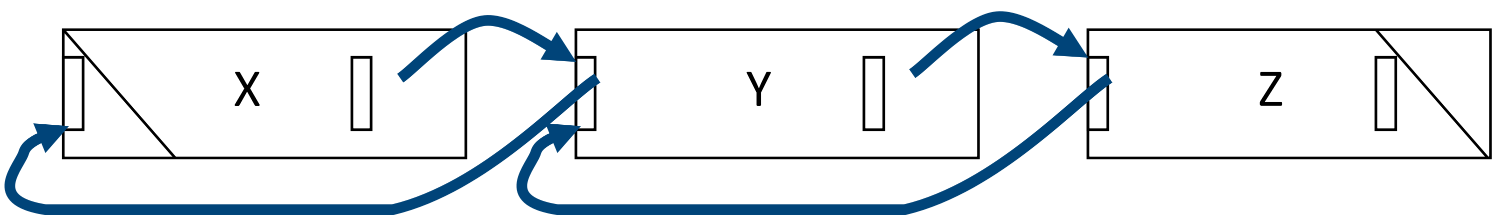
Reference Counting (p1, p2 are pointers)

```
p1 = p2;
```

- Increment reference count for `p2`
- If `p1` held a valid value, decrement its reference count
- If the reference count for `p1` is now 0, reclaim the storage it points to.
  - If the storage pointed to by `p1` was pointing to other pointers, decrement all their reference counts, and so on…
- Must also decrement reference count when local variables cease to exist.

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 $\rightarrow$ 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

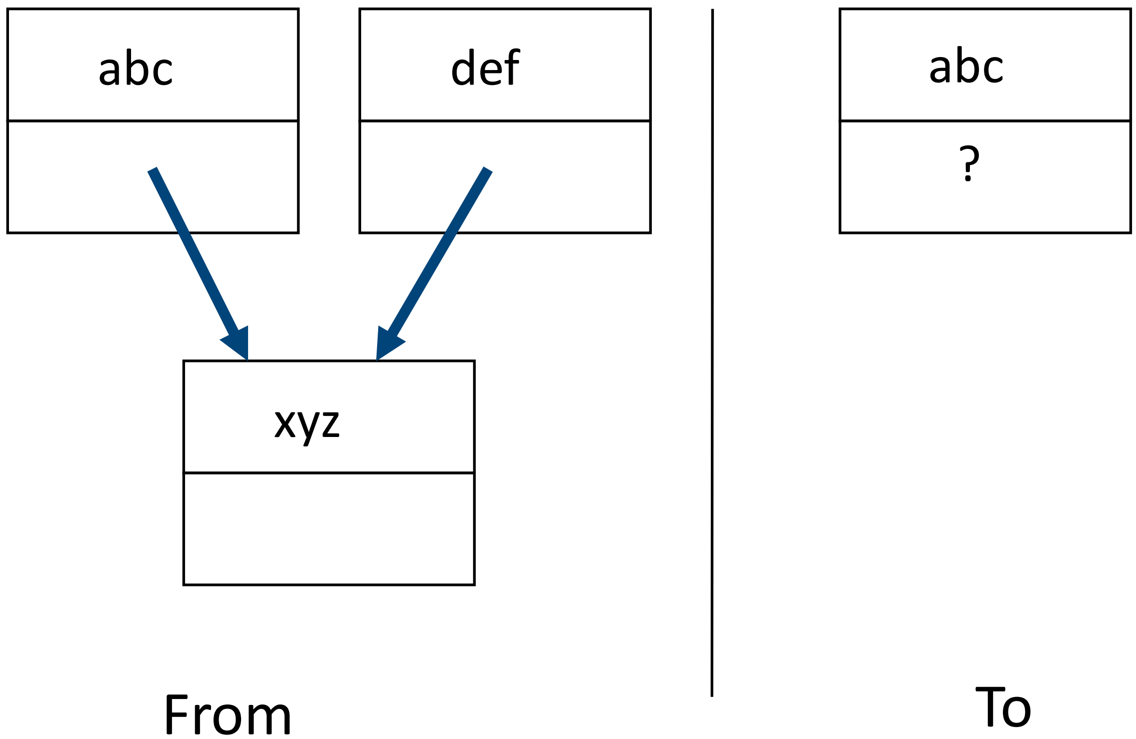
```
void traverse(struct graph_node *node) {
    /* visit this node */
    foreach child in node->children {
        traverse(child);
    }
}
```

- 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)

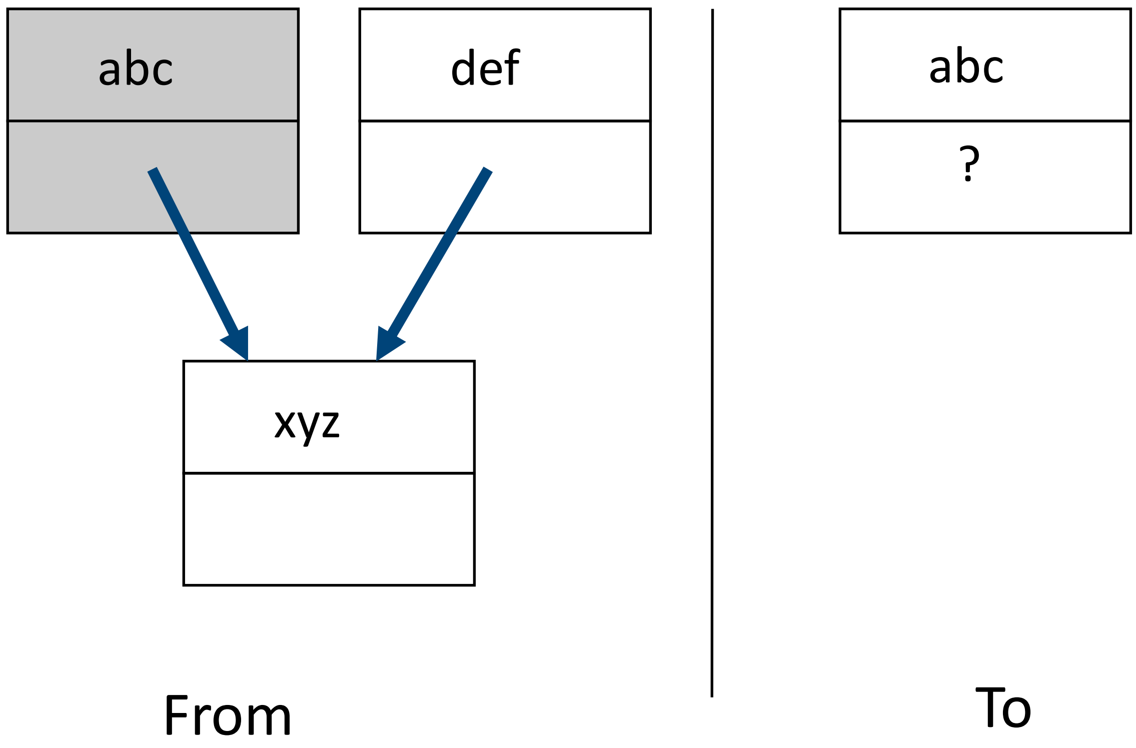
Scheme 3: Copying Garbage Collection

- 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

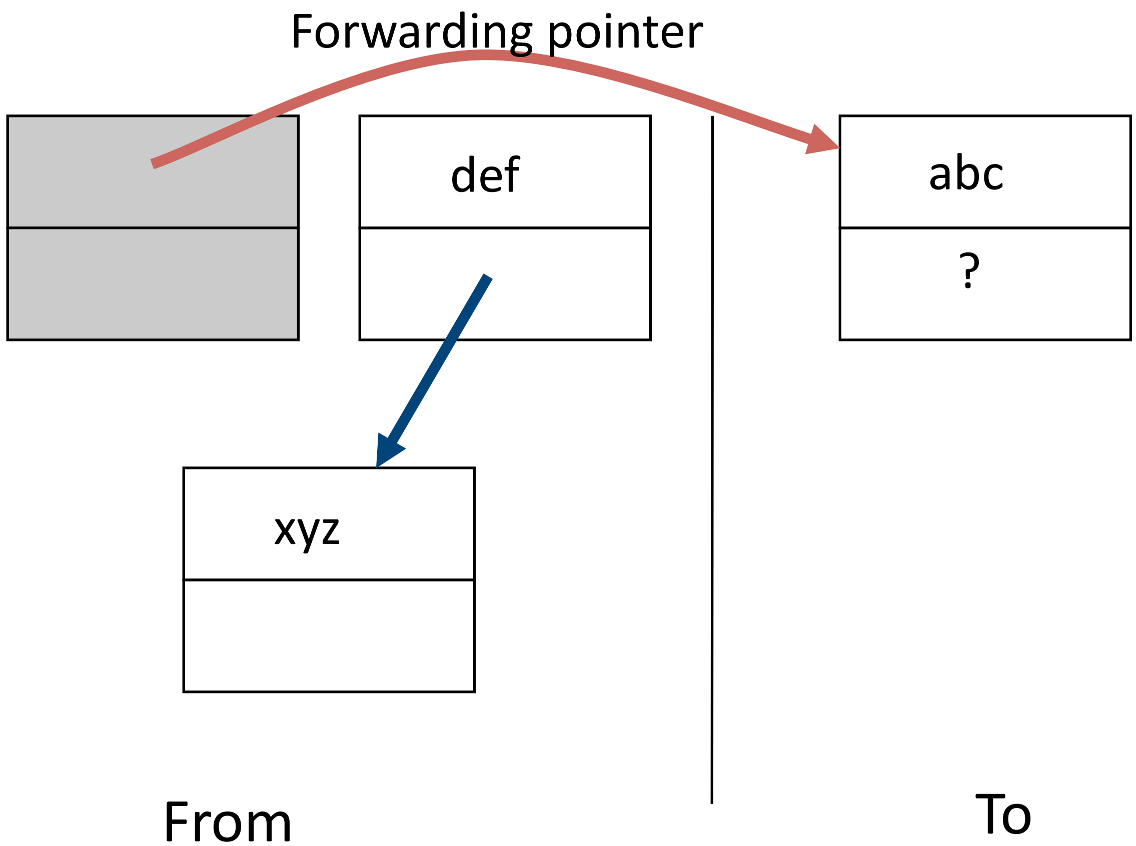
Forwarding Pointers: 1st copy “abc”



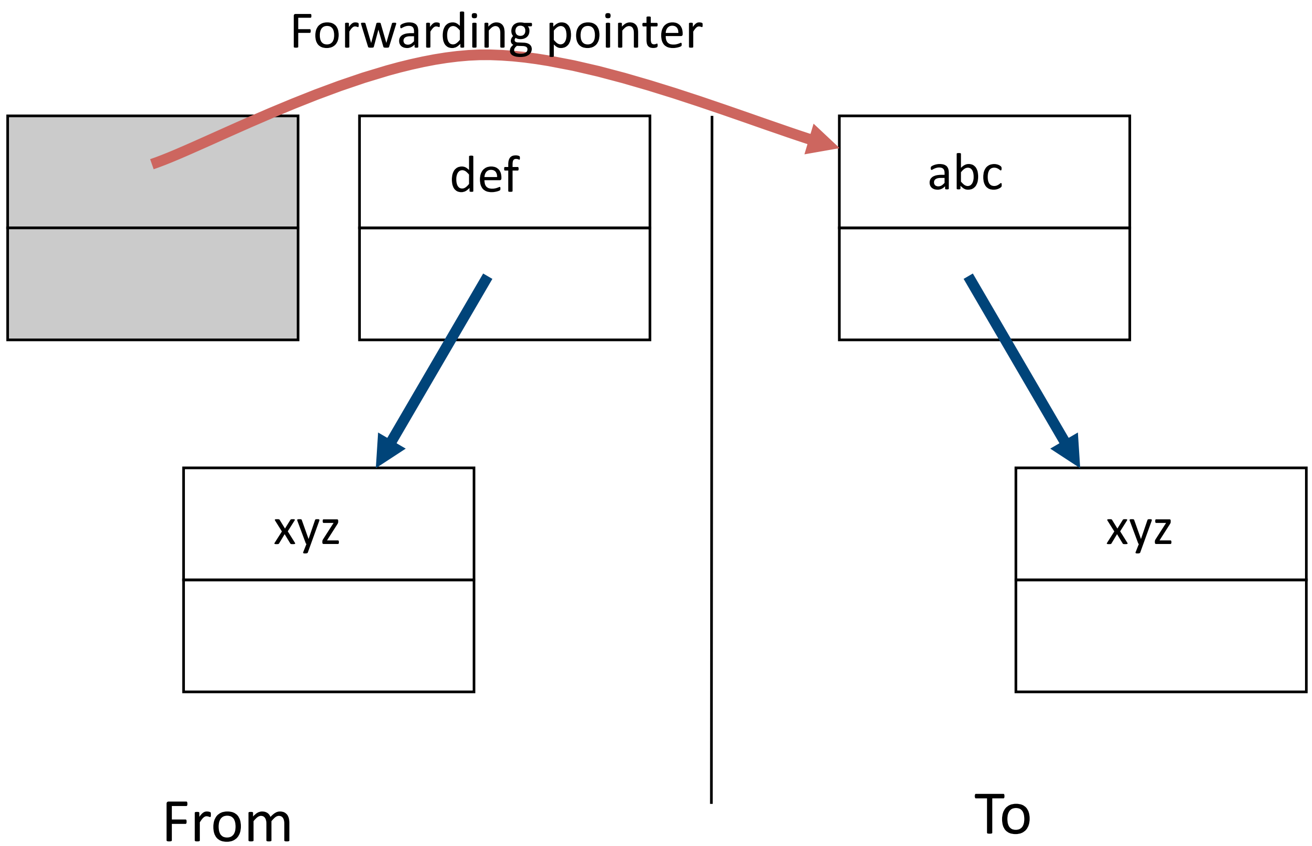
Forwarding Pointers: leave ptr to new abc



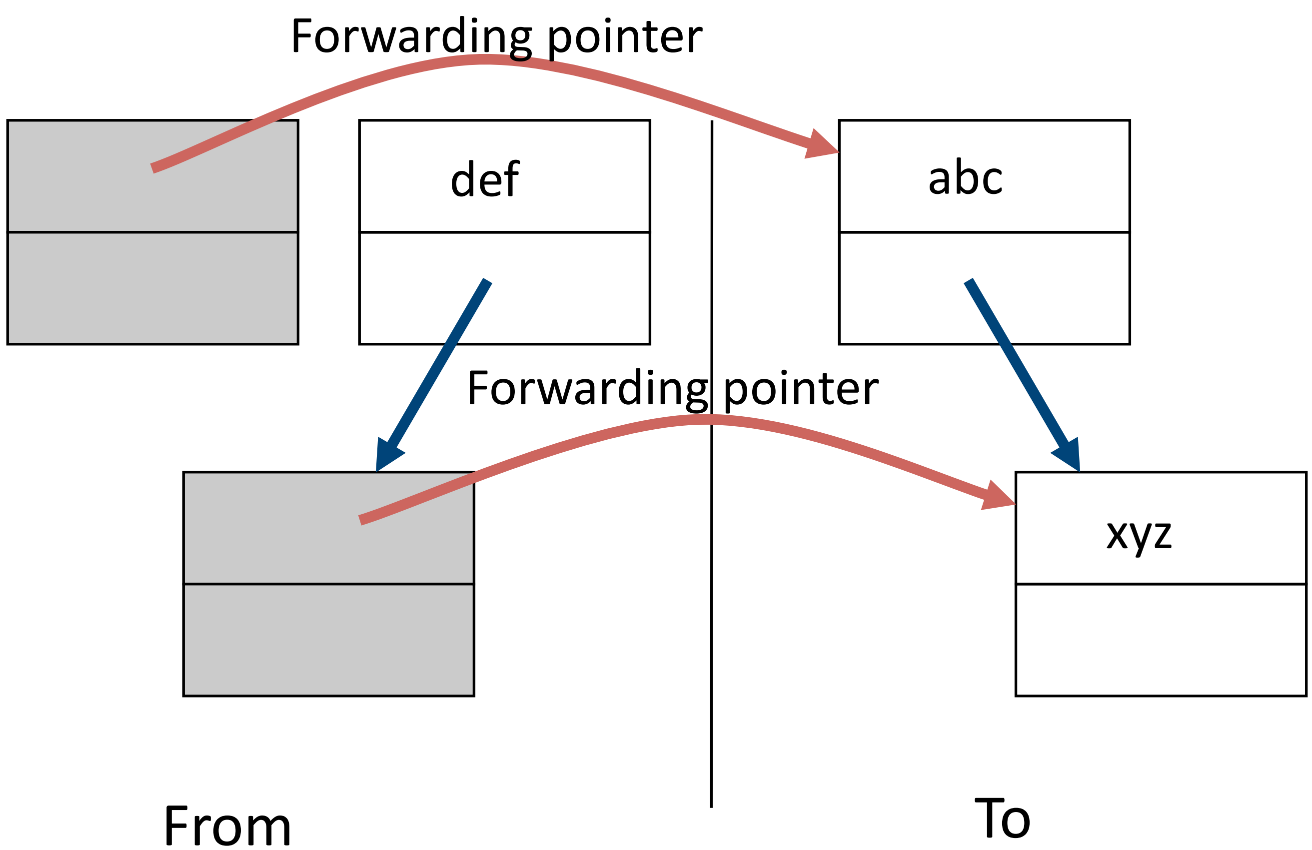
Forwarding Pointers : now copy “xyz”



Forwarding Pointers: leave ptr to new xyz

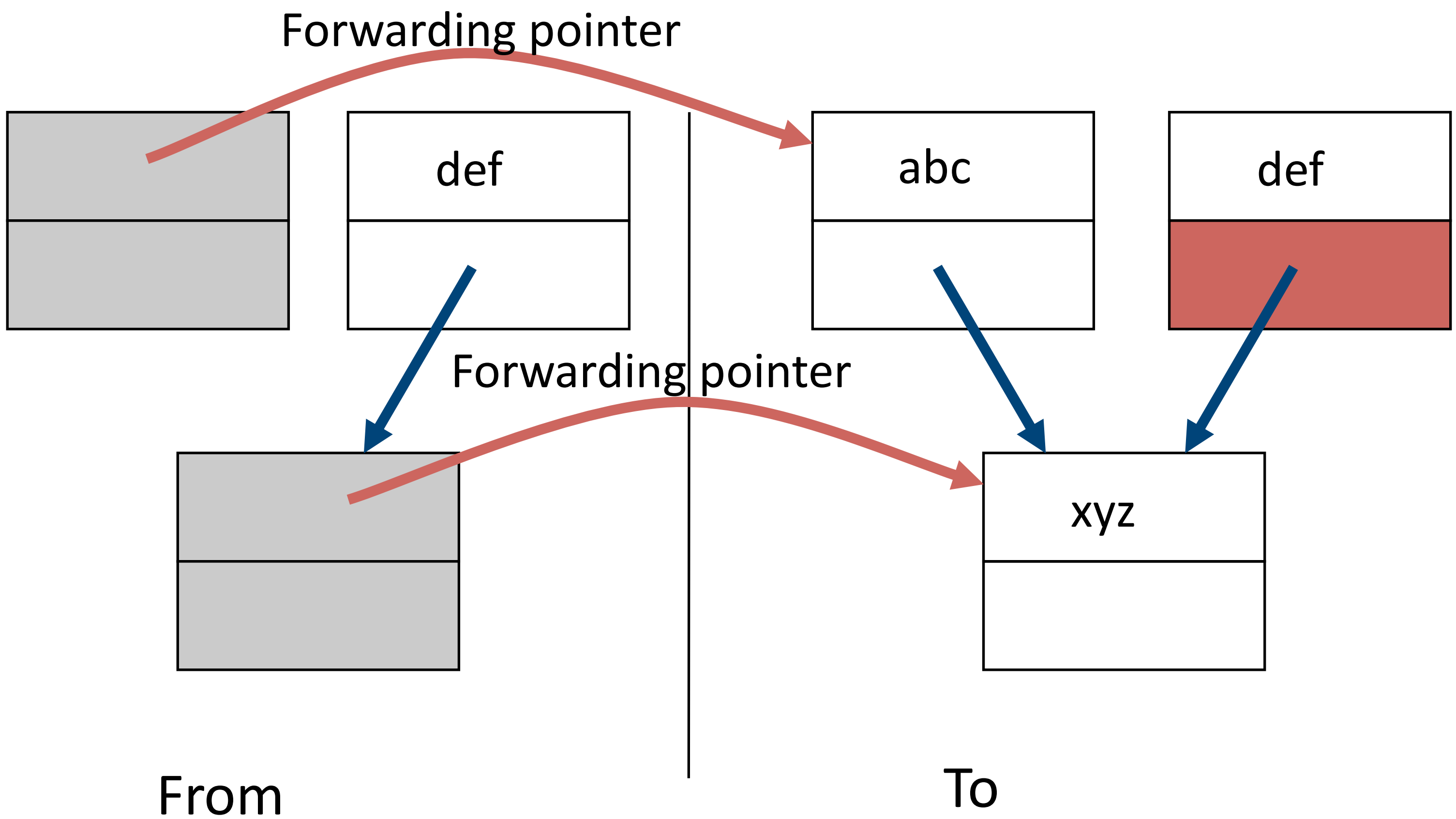


Forwarding Pointers: now copy "def"



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

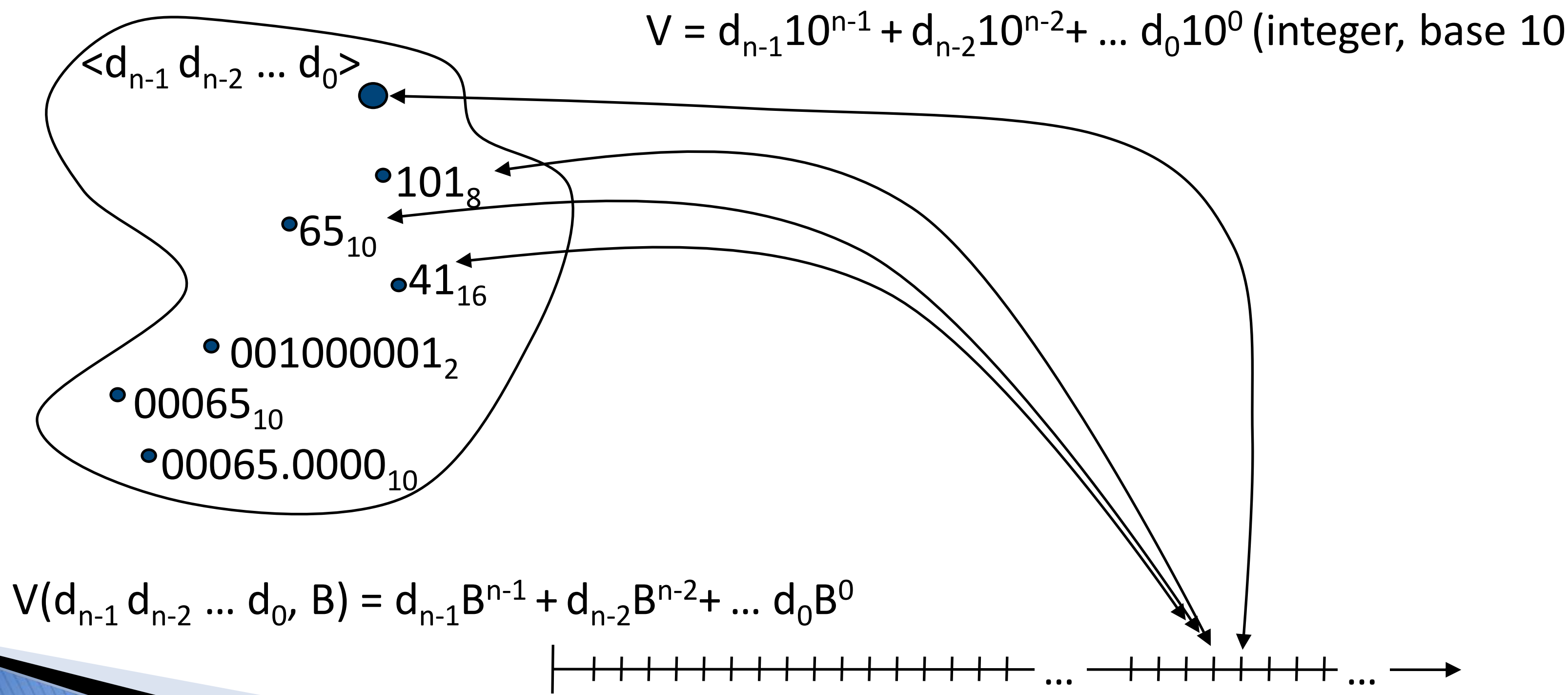
- Several techniques for managing heap via malloc and free: best-, first-, next-fit
  - 2 types of memory fragmentation: internal & external; all suffer from some kind of frag.
  - Each technique has strengths and weaknesses, none is definitively best
- Automatic memory management relieves programmer 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

- 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 π ?

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

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

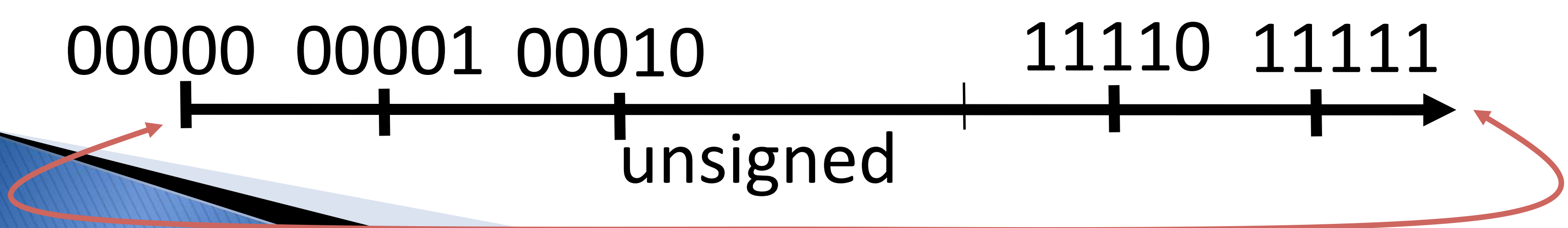
What to do with representations of numbers?

- 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 ?

```
     1   1
     1   0   1   0

+    0   1   1   1

------------------------

1    0   0   0   1
```

What if too big?

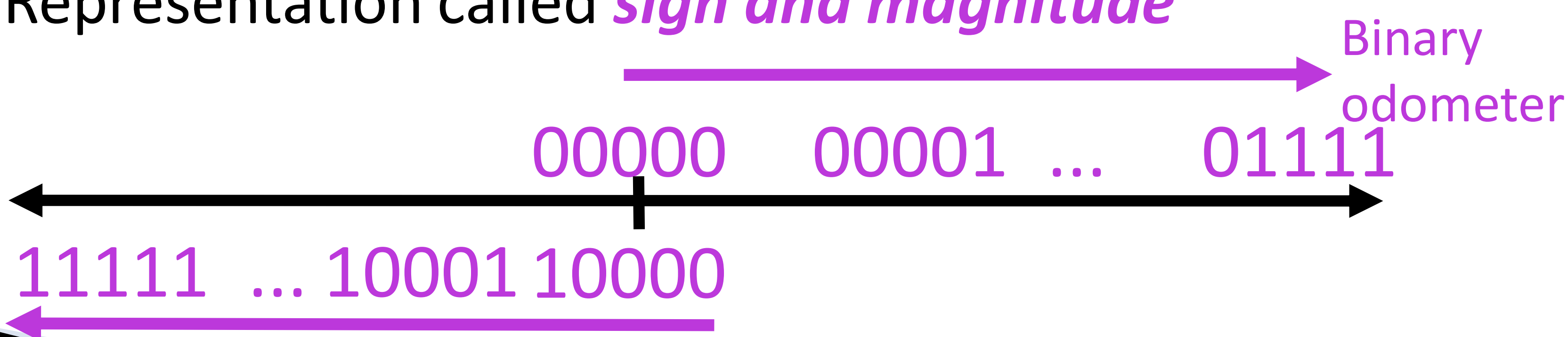
- Binary bit patterns are simply representatives of numbers. Strictly speaking they are called "numerals"
- Numbers really have an ∞ number of digits
  - with almost all being same (00…0 or 11…1) except for a few of the rightmost digits
  - Just don't normally show leading digits
- If result of add (or -, *, /) 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 → + , 1 → –
  - Rest of bits can be numerical value of number
- Representation called ***sign and magnitude***

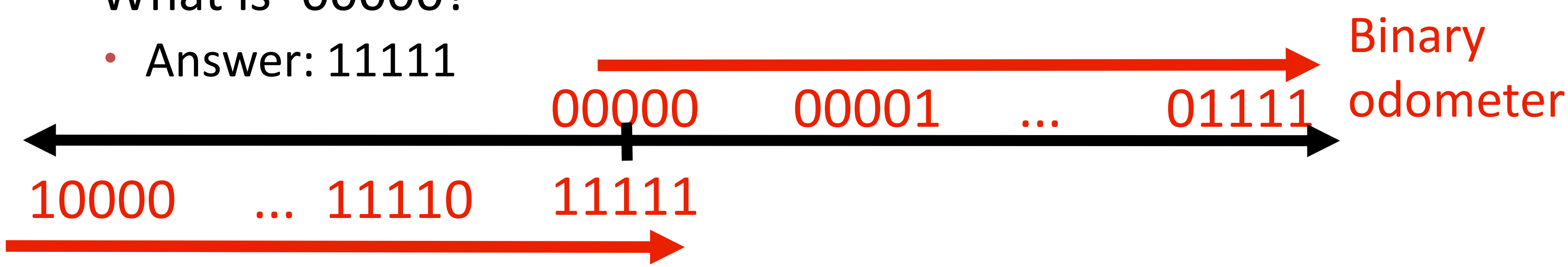


Shortcomings of Sign Magnitude?

- Arithmetic circuit complicated
  - Special steps depending whether signs are the same or not
- Also, two zeros
  - 0x00000000 = $+0_{ten}$
  - 0x80000000 = $–0_{ten}$
  - 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: $7_{10} = 00111_2 \quad -7_{10} = 11000_2$
  - Called ***One's Complement***
  - Note: positive numbers have leading 0s, negative numbers have leadings 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 = $+0_{ten}$
  - 0xFFFFFFFF = -0_{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 ≥ 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:

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

- Example: 1101_{two}

  $= 1x\text{-}(2^3) + 1*2^2 + 0*2^1 + 1*2^0$

  $= -2^3 + 2^2 + 0 + 2^0$

  $= -8 + 4 + 0 + 1$

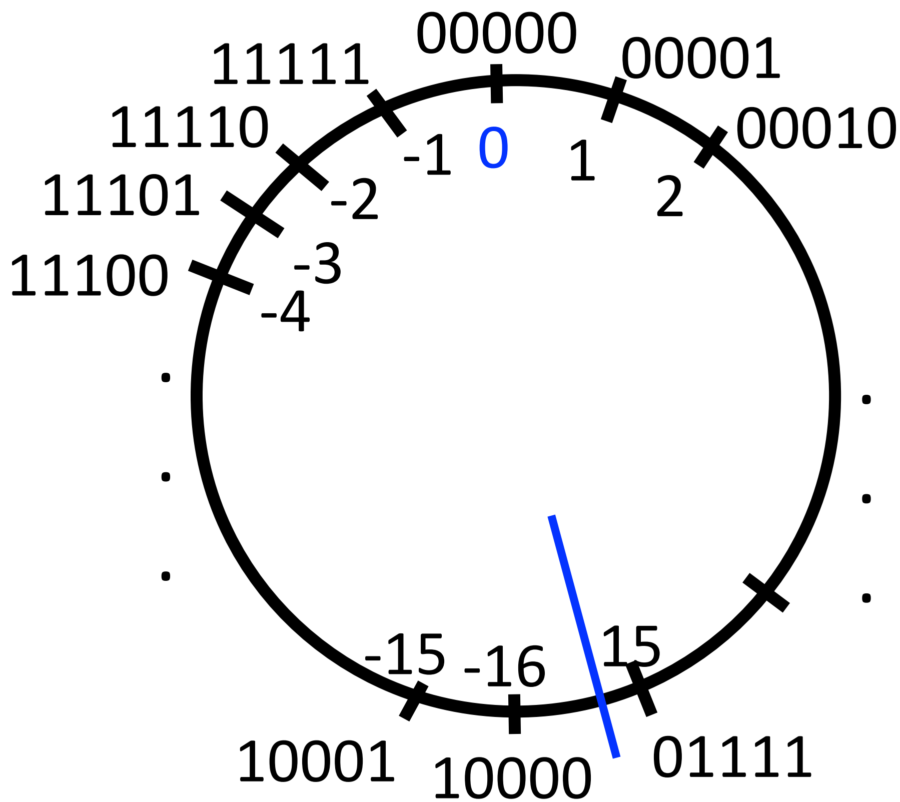
  $= -8 + 5$

  $= -3_{ten}$

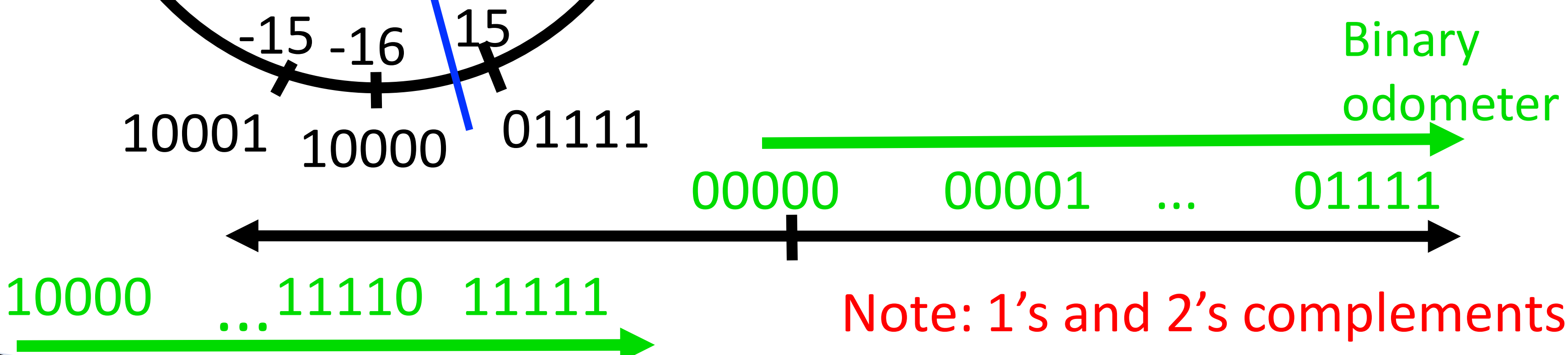
Example: -3 to +3 to -3:

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

2’s Complement Number “line”: N = 5

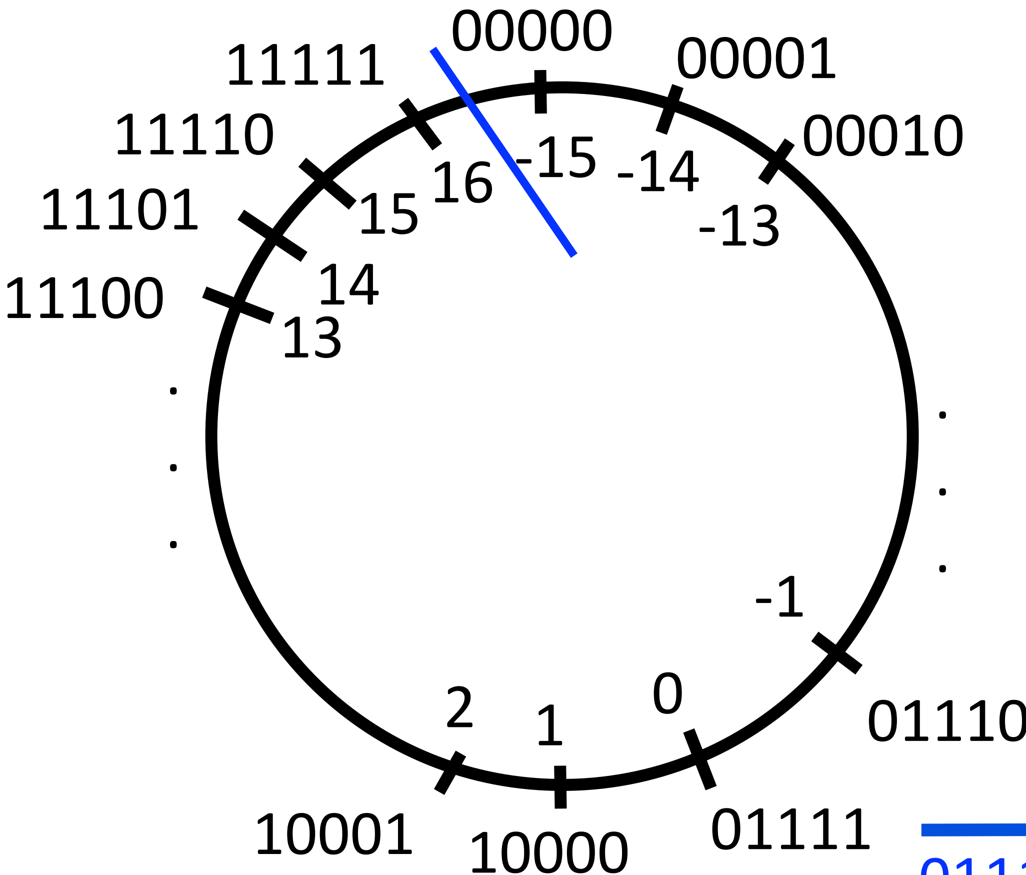


- 2^{N-1} non-negatives
- 2^{N-1} negatives
- one zero
- how many positives?
  - $2^{N-1} – 1$

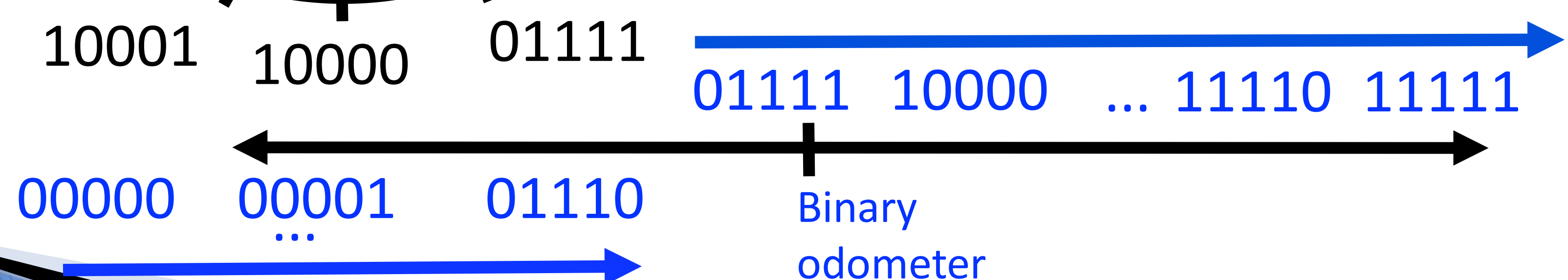


Note: 1’s and 2’s complements are used to represent negative numbers only!

Bias Encoding: N = 5 (bias = 15)



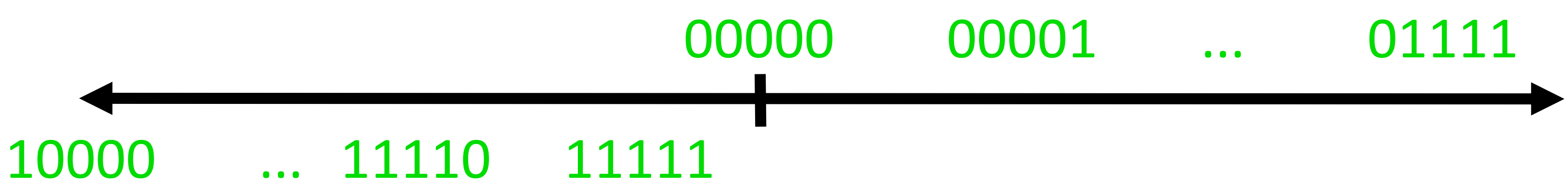
- Want 00… to represent the smallest number
- value = unsigned - bias
- Bias for N bits = $2^{N-1} – 1$
- one zero
- how many positives?
  - 2^{N-1}
  - (more than 2’s complement)



Summary

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

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



- Overflow: numbers ∞ ; computers finite → errors!