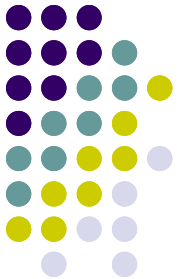
Pointer Basics



Optimized C++

Ed Keenan

23 January 2019

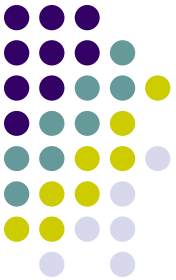
13.0.6.3.4 Mayan Long Count

Goals

- Tweak your understanding of pointers
  - Hopefully it's a review
- Learn a cool geek tricks
  - Still great party material
    - Even works at lunches
    - Multi-useful material

This class is great!

Welcome to the Club!

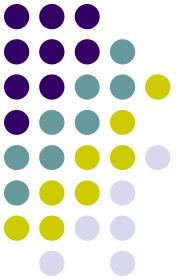


- Pointers are very important
  - You need to understand completely
  - Be proficient
  - Understanding offsets
  - Understanding arrays
  - And more
- Important Code Ninja Skills
  - Train, Train, Train
  - Drill, Drill, Drill

R Lee Ermey - Wisdom

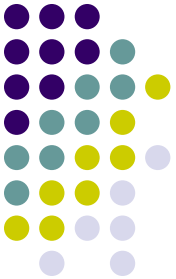
- Full Metal Jacket
  - Rifleman's Creed

Programmer's Prayer



- This is my compiler.
  - There are many like it, but this one is mine.
  - My compiler is my best friend. It is my life. I must master it, as I must master my life.
- Without me, my compiler is useless.
  - Without my compiler, I am useless.
- I must write my code true.
  - I must debug better than my competition who are trying to take my job.
  - I must release to market before they reach beta.
- I will.

Never under estimate



- Never under estimate
  - A motivated software developer
  - Who knows pointers!
    - They can code anything!

Reality Check!

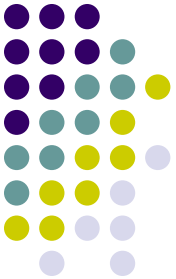
- I really need to get everyone's attention
  - Yes you in the back of the room

- Here's a Quiz!
  - http://www.online-stopwatch.com/countdown-clock/full-screen

Answers

1. 0x3F
2. 0xb5
3. 0xcd
4. 0x44
5. 0xcd
6. 0x75
7. 0x11
8. 0x29
9. 0x56
10. 0xa9
11. 0xa9
12. 0x3f12cdab
13. 0x11225529
14. 0x35d3b533
15. 0x13a98856
16. 0x11225529
17. 0x0366
18. 0x3375
19. 0x2668
20. 0x8856
21. 0x13
22. 0x29
23. 0x14

Pointers == Arrays?



```
int *p;

p[3] = 5;

int a[10];

a[3] = 5;
```

- Are pointer and arrays the same?
  - Both access the 3rd element
    - **p[3]** accesses the 3rd element
    - **a[3]** accesses the 3rd element
  - Syntax looks the same
    - let's look into this a little deeper
  - a[3] translates to:
    - ***(a + 3)**
  - p[3] translates to:
    - ***(p+3)**

Arrays

int a[10];

- *(a + 3)
  1. dereference the following
  2. address a
  3. plus the 3 * sizeof (a type)

- *(***a** + 3 * sizeof(***int***))
- Array is the literal storage
  - It's the address the first element

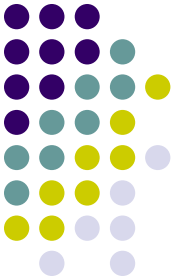
Pointers

- ***Pointer*** is a location in memory that contains an address to the memory

int *p;

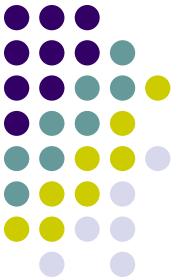
- *(p+3)
  1. dereference the following
  2. address that is contained in **p**
  3. plus the 3 * sizeof (what p points to)
- *(***p*** + 3 * sizeof(***int***))

Shocking truth



- Pointers and Arrays are different!
  - Keep those pictures in mind.
- Ask yourself,
  - Is the data or a register that points to the data?

Pointer arithmetic



- *(p + 3);
- P is a register that contains an address
- 3 is the *number* of locations of the type

- 3 * sizeof (***what p points to***)

- Double *p;
  - *(p + 3)
  - *(p + 3 * sizeof(double))
  - *(p + 3 * 8)
  - *(p + 24)
- Address in variable p + 24 bytes is the address that is dereference

[Bracket]

- p[3]
- Translates to
  - *(p + 3)
- Addition is Commutative
  - *(3 + p) is same
- Negative numbers are allowed
  - *(p-3)
  - p[-3]

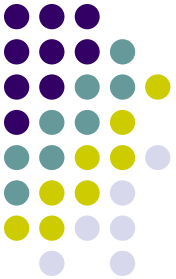
- So it party time….
- if
  - *(p+3) == p[3]
- then
  - *(3+p) == 3[p]

How Kool is That?
Why the K in cool?
Mortal Kombat team.

Casting

- static_cast<***dest type***> (***source variable***)
  - Converts to the destination
  - Old style cast, *float x = (float) d_int;*
- reiterpret_cast<***dest type***> (***source variable***)
  - Does NOT convert data.
  - Shut's up compiler warning, just copy data literally
- dynamic_cast<***dest type***> (***source variable***)
  - Casting down in hierarchy through polymorphism
  - Cast a derived object to a base object
  - Can be dangerous
  - Needs Run-Time Type Information (RTTI)
- const_cast<***dest type***> (***source variable***)
  - Removes const off of a variable
  - If you use this, generally something is wrong

Pre-increment *++p



*++p

- Translates to:
  1. p = p + 1;
  2. *p;

*--p

- Translates to:
  1. p = p - 1;
  2. *p;

- Parenthesis does nothing
- *(++p) == *++p
- Remember both
  - pre-increment
  - pre-decrement
  - Address is processed ***before*** dereferencing

Cave Formations

- Stalagmite
  - Forms for deposits starting from the floor
  - Be careful, you “might” trip on them,
  - Stalagmite

Cave Formations

- Stalactites
  - Forms for deposits starting on the ceiling
  - They “hold on tight” to the ceiling
  - stalactite

Pre-increment *++p

- Pre-increment / pre-decrement
  - Think of it prematurely modifying the address.
- It's a bit of a stretch
  - But you'll remember it.

Post-increment *p++

*p++

- Translates to:
  1. *p
  2. p = p + 1;

*p--

- Translates to:
  1. *p;
  2. p = p - 1;

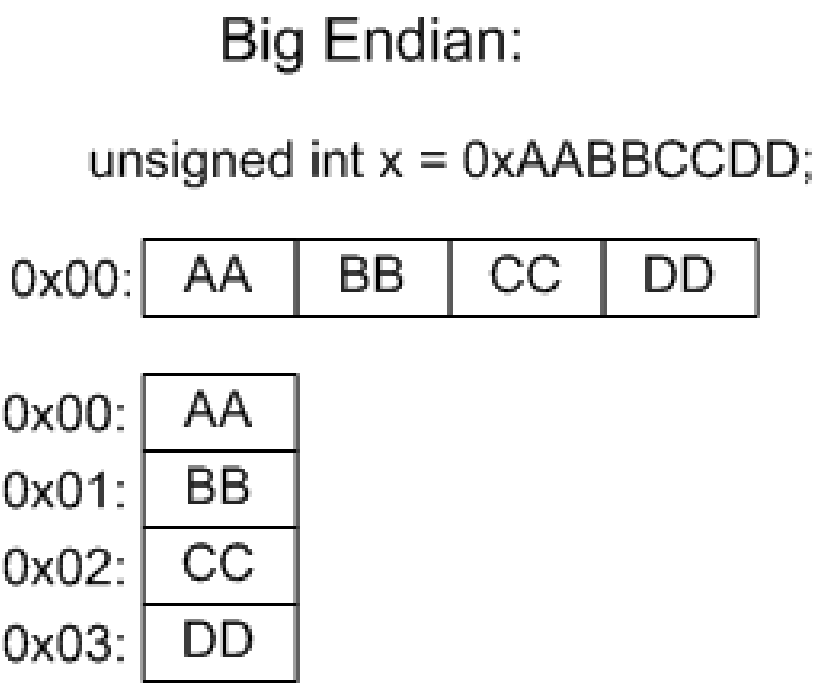
- Parenthesis does nothing
  - *(p++) == *p++
- Remember both
  - post-increment
  - post-decrement
  - Dereferencing happen ***before*** address processing.
- Faster than pre-increment, (yes!)
  - since the dereference is immediate

Post-increment *p++

- Post-increment / post-decrement
  - Give me the pointer NOW
  - We fix it in ***POST***
    - modifying the address
- It's a bit of a stretch
  - But you'll remember it.

Big Endian

- Most significant byte is at the first address.
- PowerPC, Motorola Chips, and other processors
- It's the layout I think instantly
  - Makes sense

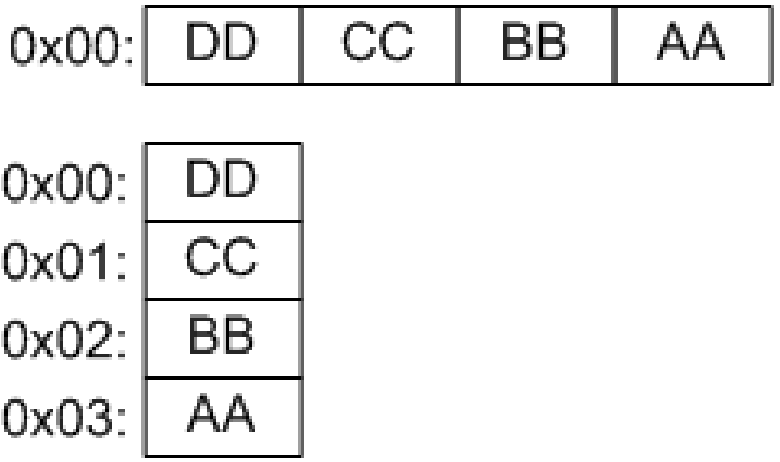


Little Endian

- ***Least*** significant byte is at the first address.
- Intel, Mips, and other processors
- Backwards, feels weird.
- Created out of legacy issues from the 8088.

Little Endian:

unsigned int x = 0xAABBCCDD;



Adding 7 bytes to int *

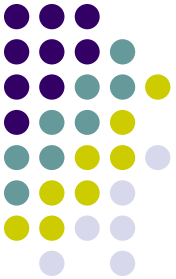
- Example:
  - int *p;
  - p = p + 7;
    - Adds 28 bytes to address p not 7
      - Why?
- Don't ask why, here's how you do it.

```
p = (int *)( (char *)p + 7);
```

or

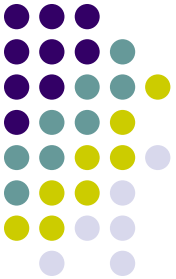
```
p = (int *)( (unsigned int)p + 7);
```

Subtracting pointers



- Can subtract two pointers of the same type.
  - (p2 – p1)
- The result is the distance (in array elements) between the two elements.
  - NOT in BYTES
- Can be confusing… I convert to char * or unsigned ints.
  - difference_in_bytes = (char *)p2 – (char *)p1;

Useful information



- printf (“ %p “, p);
  - %p – prints pointers
- Do Not use NULL
  - Use 0 for Null pointers
  - It’s the ANSI spec
- Always test your pointers before derefencing them

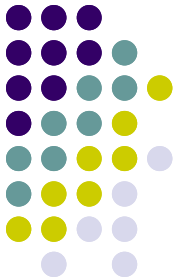
Examples

- On Board:

Thank You!

- Questions?

Simple Memory System



Optimized C++

Ed Keenan

Goals

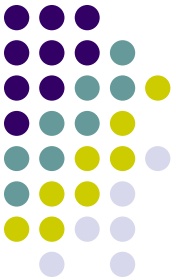
- Understand the internals of a simple memory system
- Create your own Heap Memory system
  - Not so great party material, too long to describe
    - 3-monkey joke is better

About time!

3 Monkey Joke

- There are 3 monkeys playing in a tree.
  - They climb down from the tree and play in the dirt so they are dirty. ***DIRTY MONKEYs***
  - Now the dirty monkeys need to take a bath in the bath tub next to the tree.
- The monkeys are now in the bath tub.
  - The 1st monkey says to the 2nd monkey,
    - ***"Pass the brush"***
  - The 2nd monkey says to the 3rd monkey,
    - ***"Pass the soap"***
  - And the 1st monkey says to the 3rd monkey,
    - ***"Pass the shampoo"***
  - And then the 2nd monkey says,
    - ***"What do you think I am, a typewriter?"***

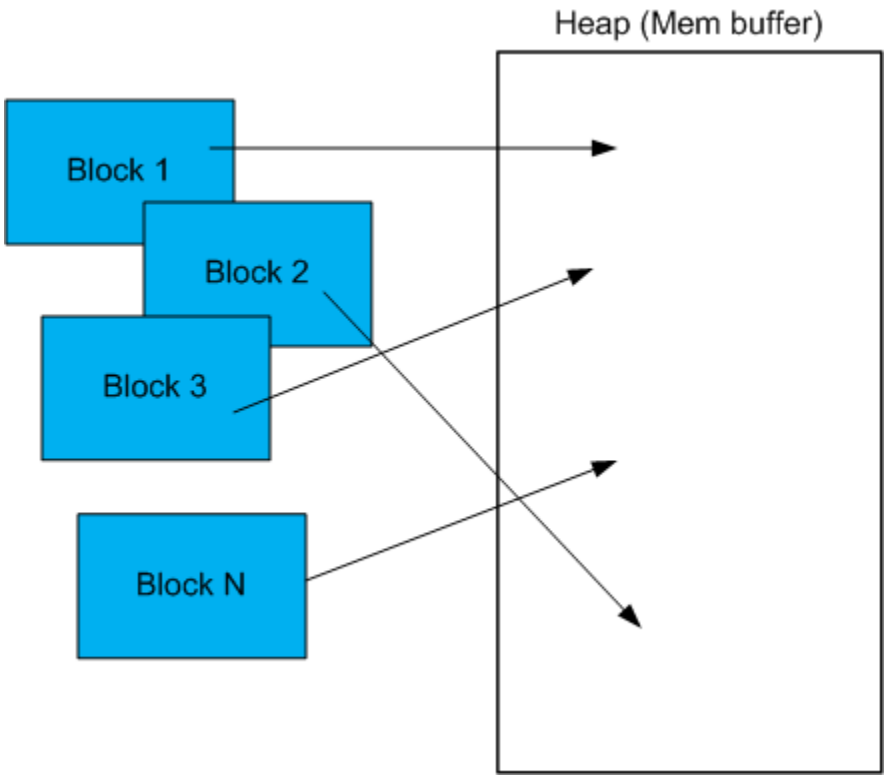
Heaps



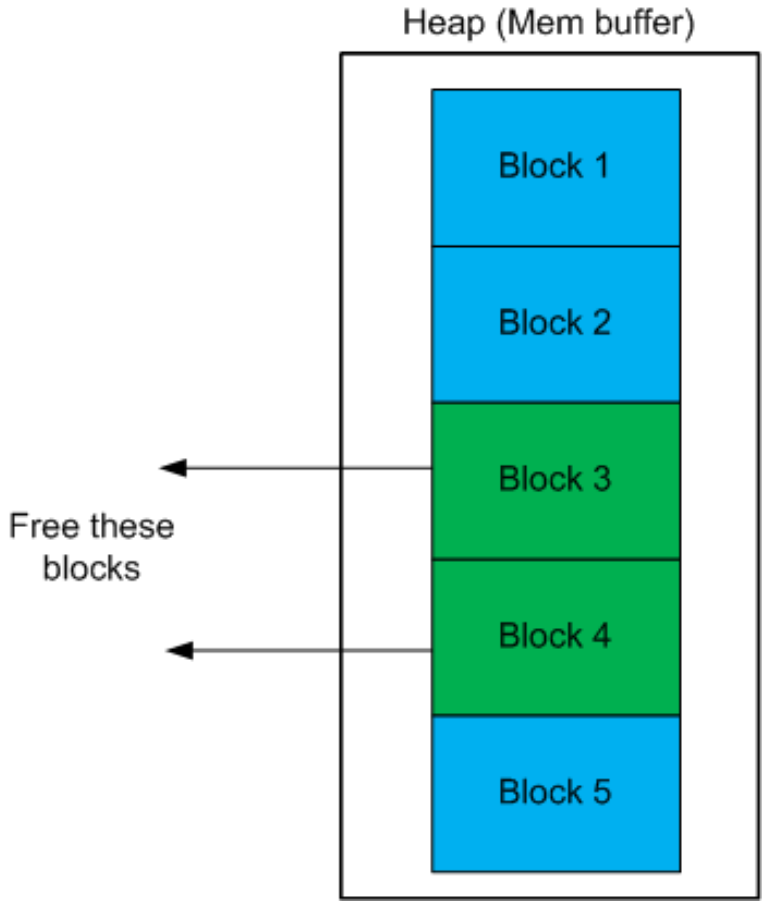
- Heap is a stand-alone buffer that contains dynamic allocations
  - Every allocation comes out of one bucket
  - User has control of Heap creation and lifetime
  - Can create memory allocations schemes to suite the code use patterns.
  - It's faster, faster, faster
- This class is all about memory usage is some way or another.
- Macro Benefits
  - Benefits, all the allocations are contained in the bucket.
  - We are not fragmenting other memory buckets.
- Micro Benefits
  - Create different allocation schemes
  - Can be very fast
  - Can have very low fragmentation
  - Can use temporal relationship data
  - Generally faster than global new and delete

Problem of management

- Start off with a buffer, we'll call that the heap.
- Now we need to add usable blocks to the heap.
- How do we manage them?



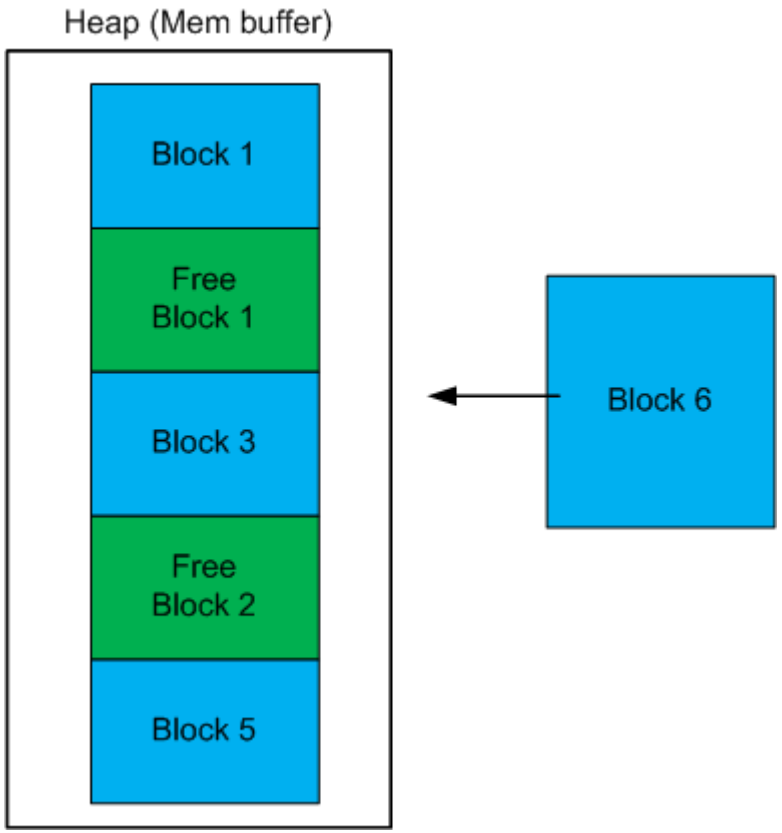
How do we Free Blocks?



- If we free adjacent blocks
- Coalesce the blocks into one larger free block
  - Gives us more options for future allocations

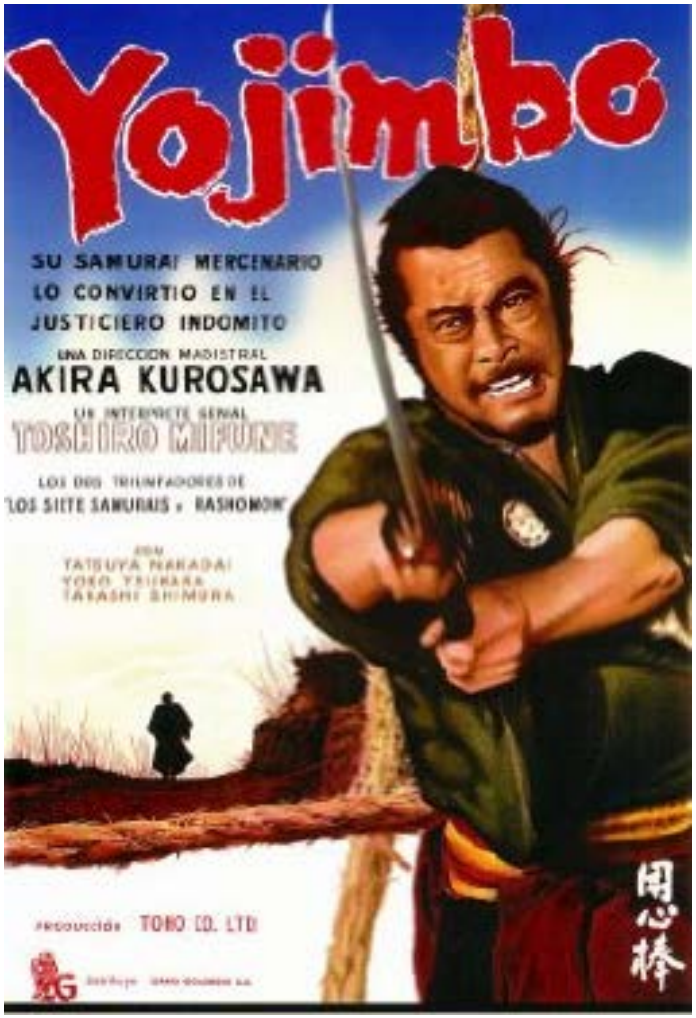
Problem of Fragmentation

- There's empty space, but I can't find a contiguous block available
- Too many allocations sub-dividing the space.



Where's Yojimbo?

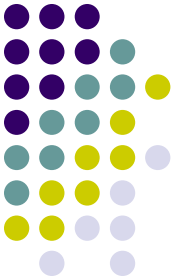
- Thanks, now I have a name.



Time to Allocate / Deallocate

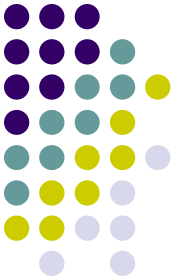
- How efficient is the request for allocation, (malloc).
  - Searching for blocks
  - Best strategy
- How much time does it take to free an block?
  - Time is money!
  - We are about speed

What is a Memory System



- Dispenses dynamic memory blocks (***malloc***) and reuses memory blocks discarded through (***free***).
- Physical data structures, management and data layout are very important
- Solving several problems at once:
  - Fragmentation
  - Coalescing of free blocks
  - Searching
  - Algorithms
  - Allocation strategies
  - Management of data structures

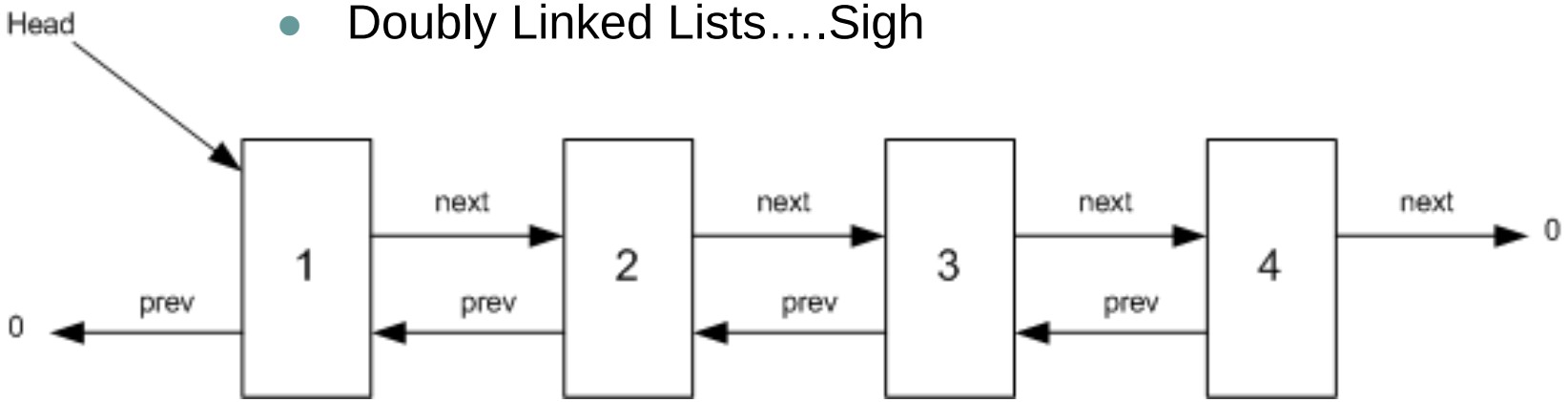
Let's start



- Using ***malloc*** and ***free***
- Yes, I'm old,
  - But that's how it really works at an implementation level
- Operator ***new*** and ***delete*** deal with the instantiation of the objects,
  1. Call system malloc / free under the hood.
  2. Call constructors (new, reverse order for delete)

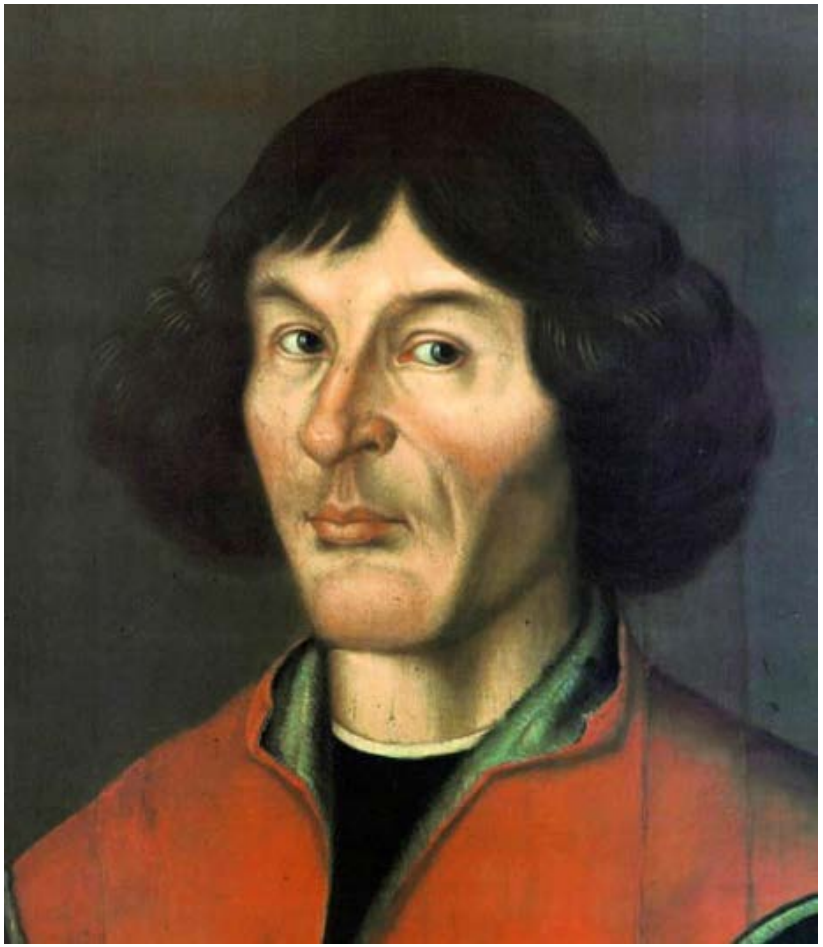
Tracking blocks

- Need to determine which blocks are allocated
- Requirements
  - Easily insert
  - Easy to remove
  - Tracking
- Simple data structure that we all understand
  - Doubly Linked Lists….Sigh

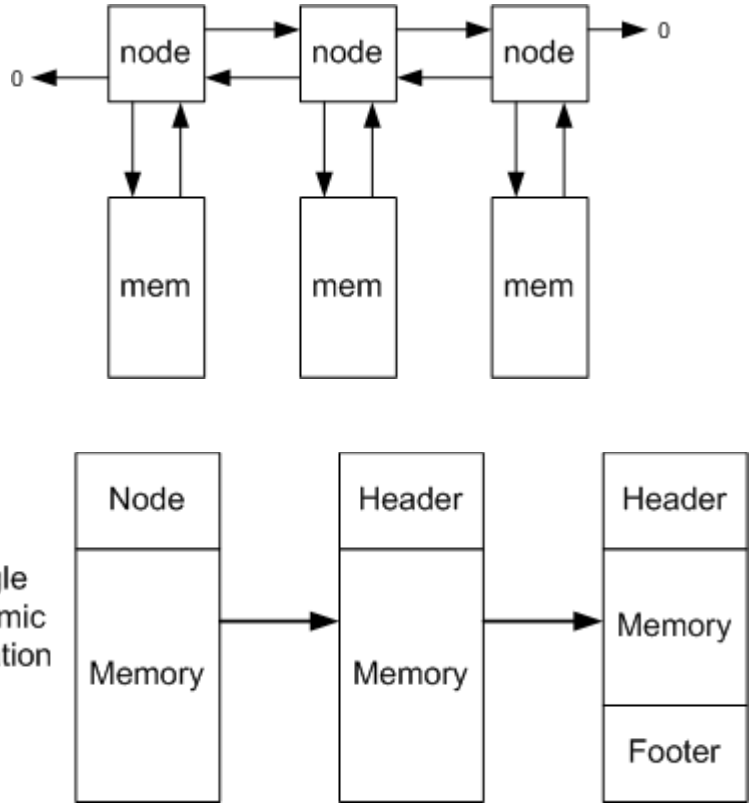


Copernicus

- Does everyone understand doubly linked lists?
  - Speak up.
  - I have a marker board
- Who knows about Copernicus?
  - He's Dead
  - You will be if you don't know these lists inside & out.
- Final Call?

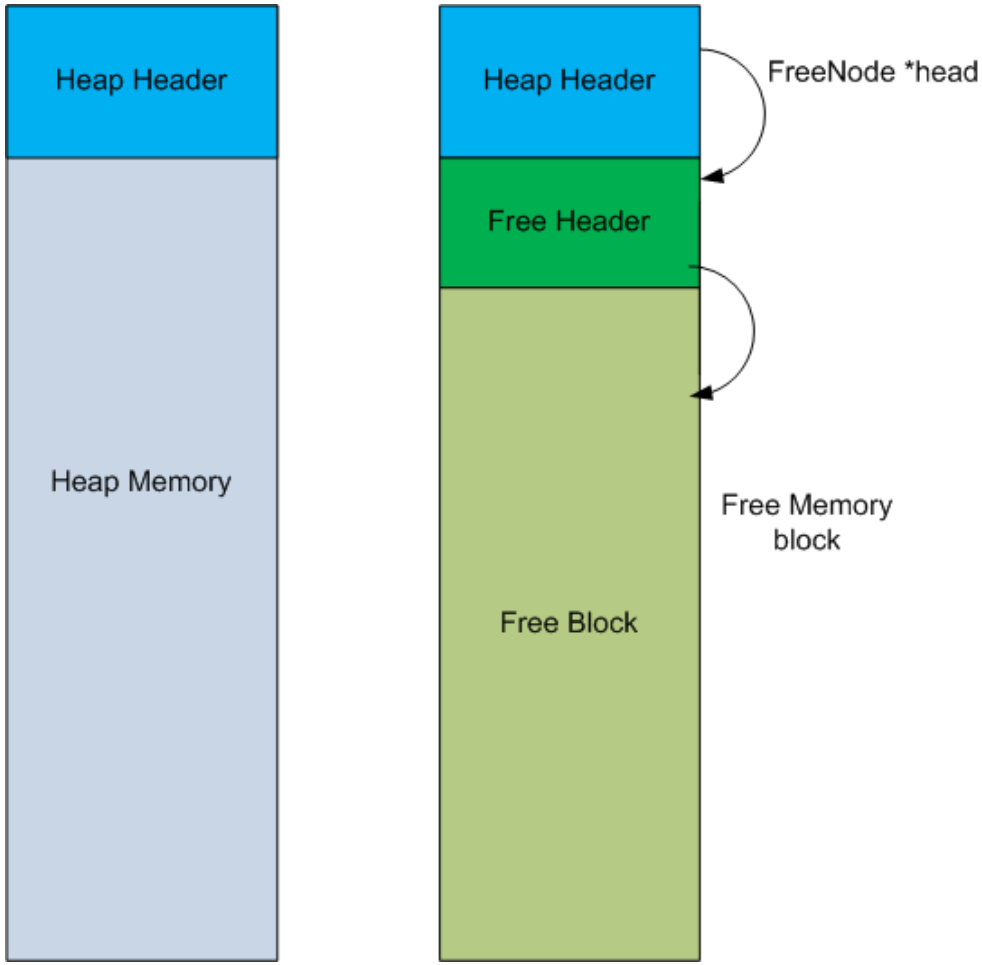


Concept of Headers



- Disjointed memory
  - Controlling node, dynamic memory allocated elsewhere
  - Remember Hot / Cold structures?
    - McDLT (video please)
- Jointed memory
  - Gluing them together
  - Controlling structure + dynamic memory
- Call Node header

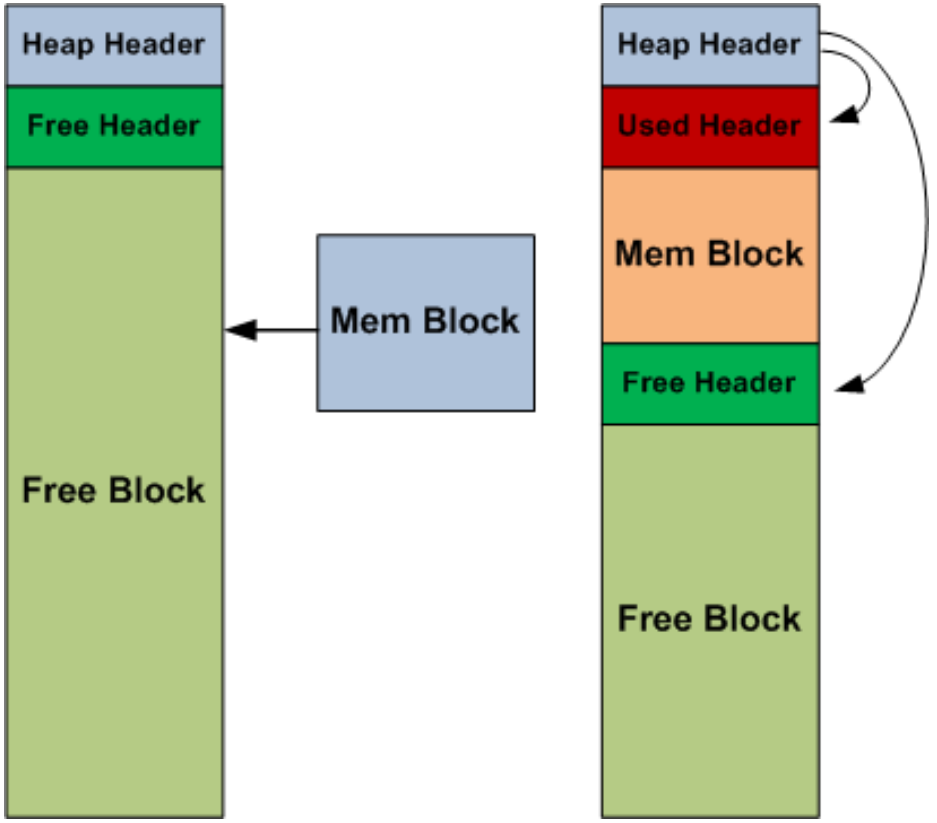
Heap Header



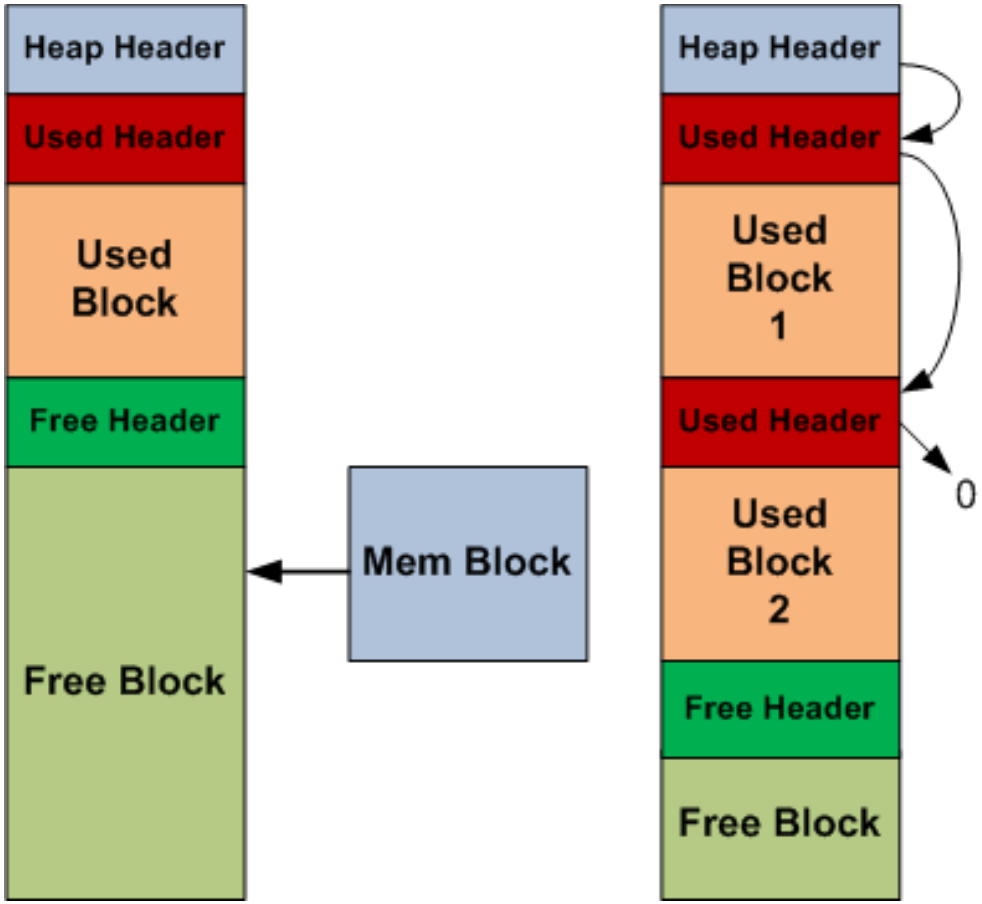
- Heap Header stores
  - Link lists head pointers
  - Tracking data
    - Number allocations
    - Number free blocks
  - Debugging data

Allocate a block

- Allocating a block from the free block
  - Will the requested block fit?
- Sub-divide the free block
  - Used block & header
  - Free block & header
- Update all the pointers in the heap header



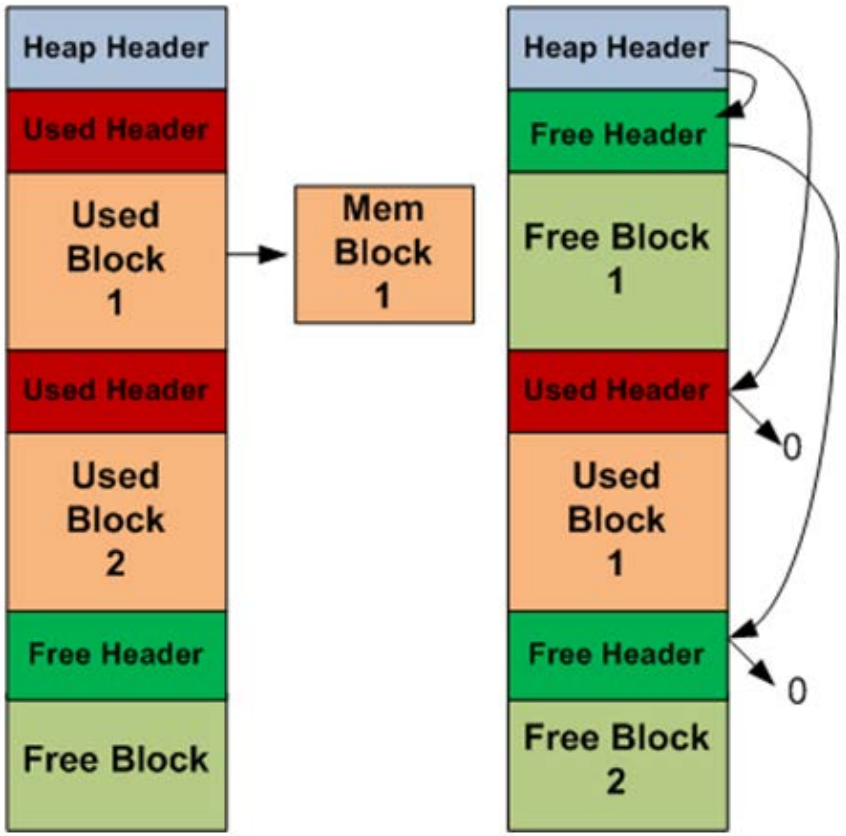
Add another block



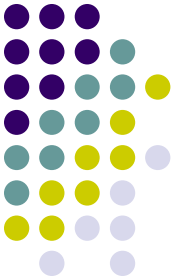
- Add another block
  - Subdivide the free block.
  - Add the 2nd block on to used list.
  - Correct all the linked lists and heap data

Free a used block

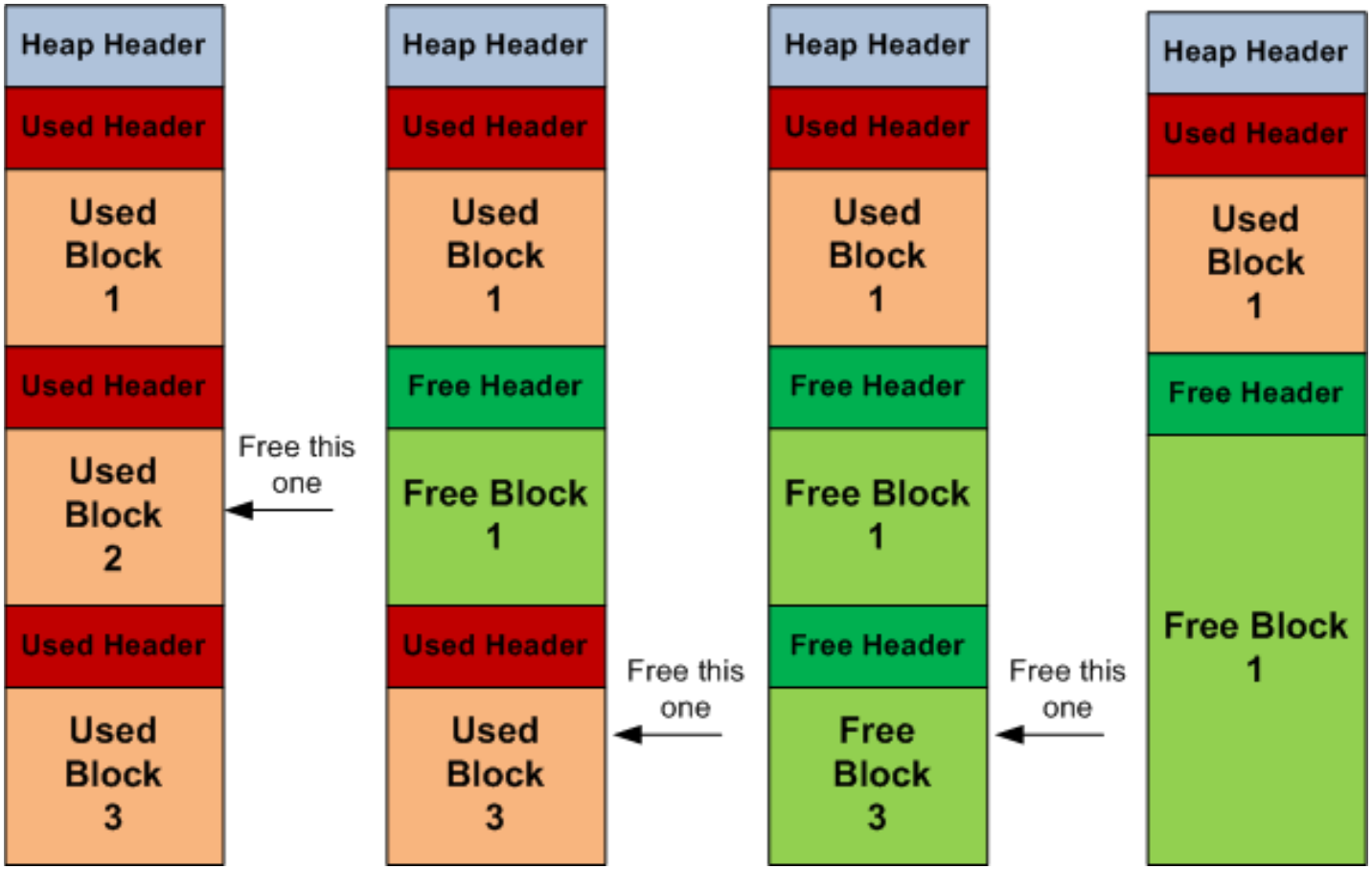
- Free a block
  - Mark the used back as free.
  - Convert the block to free
  - Fix up linked lists



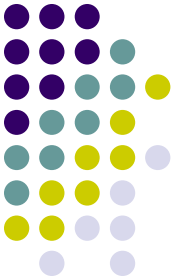
Questions so far



Coalescing


Heap Header
Used Header
Used Block 1
Used Header
Used Block 2
Used Header
Used Block 3
Free this one
Heap Header
Used Header
Used Block 1
Free Header
Free Block 1
Used Header
Used Block 3
Free this one
Heap Header
Used Header
Used Block 1
Free Header
Free Block 1
Free Header
Free Block 3
Free this one
Heap Header
Used Header
Used Block 1
Free Header
Free Block 1

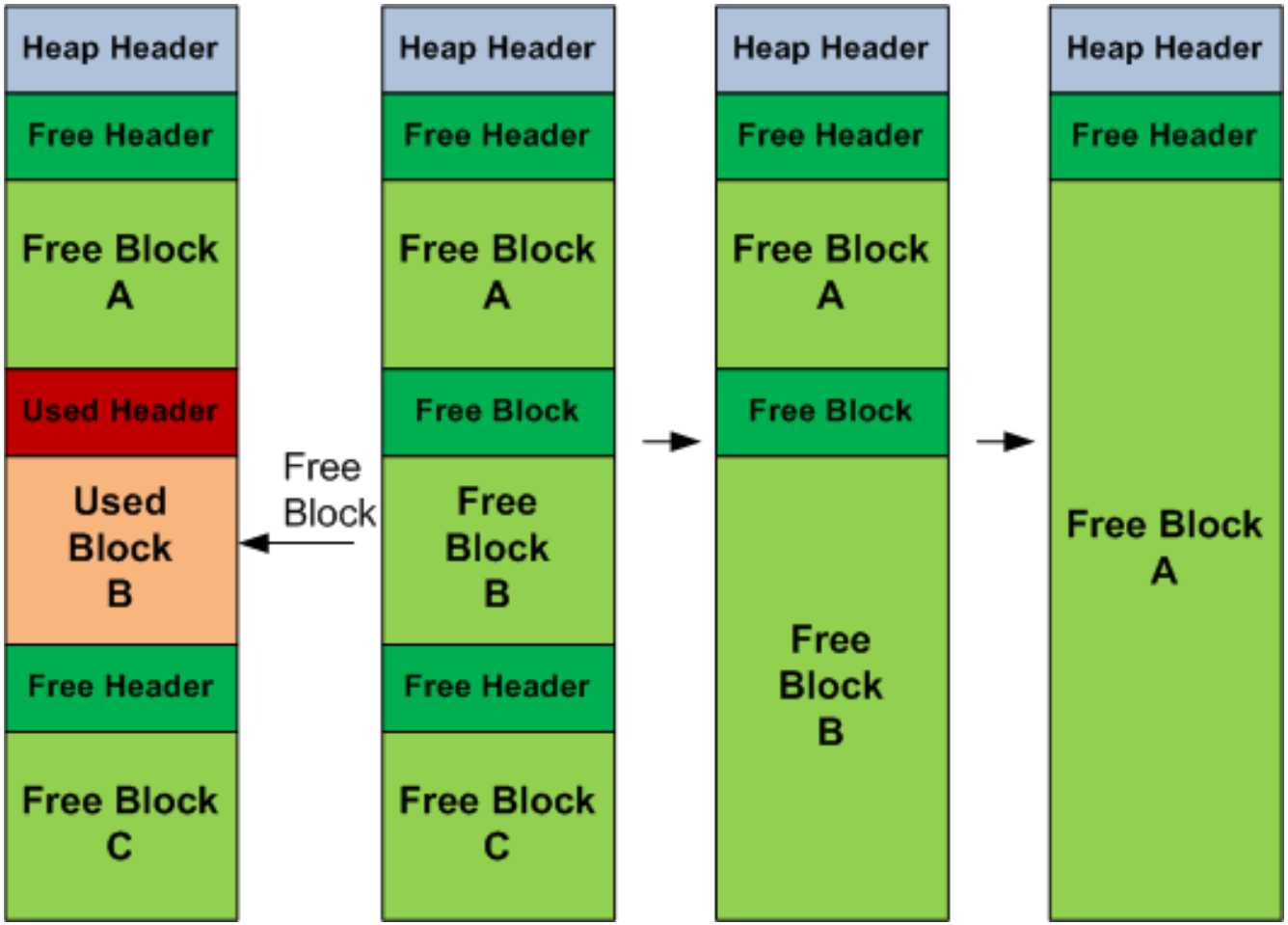
Don’t be a cowboy



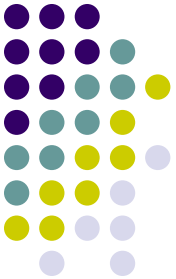
- Free blocks up in discrete steps
  1. Free the block
  2. Coalesce the blocks
     - Do them one at a time
     - Trust me.

- Kyle’s 2 Step Process
  - 6 x better than the famous 12 step process

Multiple-sided Coalescing

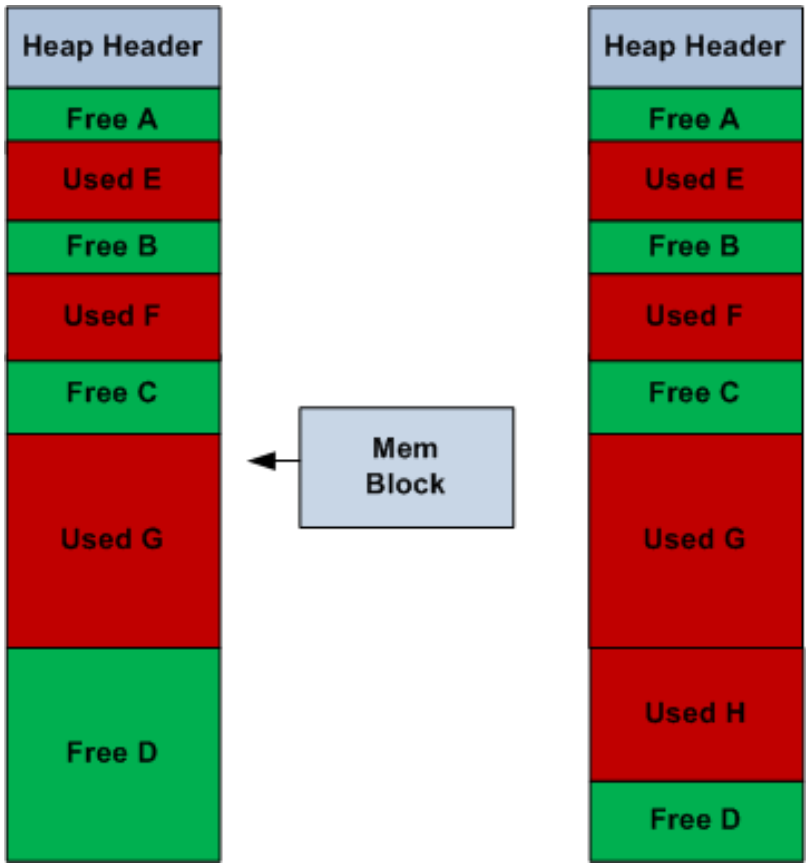


Final thoughts on Coalescing



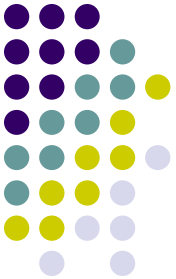
- Edge conditions will trip you up.
  - My tests will drill into these areas
  - So do the testing.
- Ideas
  - Treat the heap header as a Used Block
  - Some people put another Used Block on the bottom of the heap to reduce edge conditions
    - WE DON'T
- Some books call this compaction

Finding a block



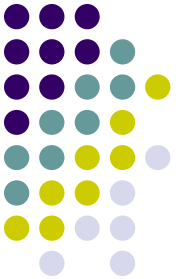
- You might have to search the Free linked list until you find a large enough empty block.
- The amount you search directly affects performance.
- Reduce fragmentation
  - Improve performance

Allocation strategies



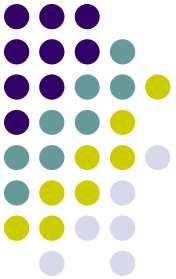
- Best fit
  - Find the hole the fits the block the best
  - Least amount of waste.
  - Slower
  - Has low fragmentation
- Worse fit
  - Find the biggest Free block
  - Take your allocation from that block

Allocation strategies



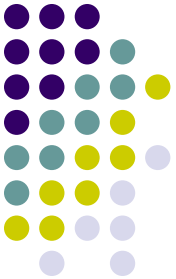
- First Fit
  - Find the first open block
    - subdivide that block for your allocation
  - Simplest, pretty effective (Use this in class)
- Next Fit
  - Find the 1st hole the fits the block
    - Save the location
  - Start from last location down the heap
    - Wrap back to top
  - Faster
  - → Use this in class PA

Practical issues



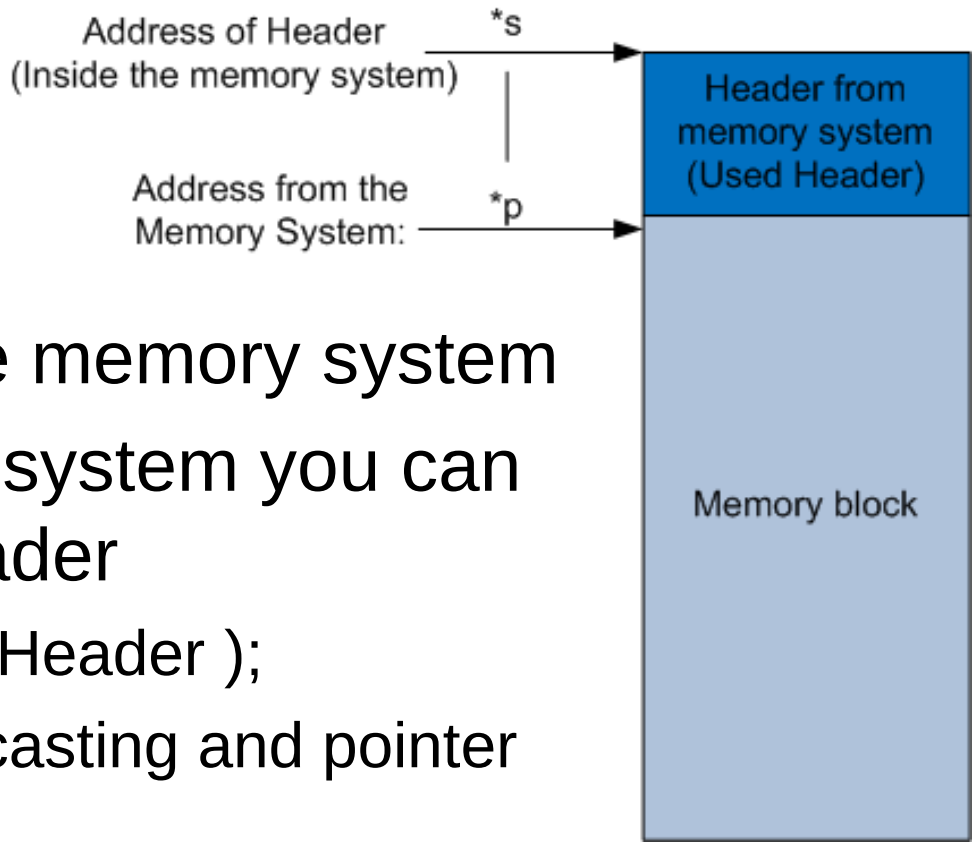
- High level was discussed
- Concepts were conveyed
  - ***Devil lies in the details***
    - Many problems come out of this assignment
- Advice:
  - You need to debug
  - Build and develop incrementally
    - Trust me

Find Header from Malloc block



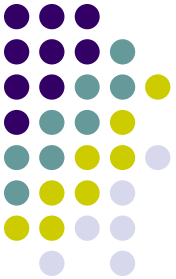
- You have a dynamically allocated block
  - Now find the header in the heap
- Naïve way:
  - Search the memory range until you can find the address of the block in heap.
  - Walk the Used pointers until you find address
- Cool way:
  - Move back up from the pointer the size of header.
  - Cast that address to be header
  - Pointer math is useful

Getting Header pointer



- You get *p from the memory system
- Inside the memory system you can get *s the used header
  - s = p – sizeof(usedHeader);
  - Do forget all those casting and pointer stuff.

Ideas for your headers



- Free Header
  - Node Linkages
    - Next pointer
    - Previous pointer
  - Size
    - Size of free block
  - Flags
    - Miscellaneous flags and fields
    - Any additional data that helps you with your design
- Used Header
  - Node Linkages
    - Next pointer
    - Previous pointer
  - Size
    - Size of free block
  - Flags
    - Flags on adjacency
    - Above block is Free or Used
    - Below block is free or used
    - Miscellaneous flags and fields that help you with your design

Thank You!

- Questions?