Photo: Dennis Ritchie and Ken Thompson

COMP201

Computer Systems & Programming

Lecture #28 – More Heap Allocators, Wrapping Up

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2021

Recap

- The heap so far
- What is a heap allocator?
- Heap allocator requirements and goals
- Method 0: Bump Allocator
- Method 1: Implicit Free List Allocator

Recap: Heap Allocator Goals

- Goal 1: Maximize **throughput**, or the number of requests completed per unit time. This means minimizing the average time to satisfy a request.
- Goal 2: Maximize memory **utilization**, or how efficiently we make use of the limited heap memory to satisfy requests.

Other desirable goals:
Locality ("similar" blocks allocated close in space)
Robust (handle client errors)
Ease of implementation/maintenance

Recap: Bump Allocator

- A **bump allocator** is a heap allocator design that simply allocates the next available memory address upon an allocate request and does nothing on a free request.
- Throughput: each `malloc` and `free` execute only a handful of instructions:
  - It is easy to find the next location to use
  - Free does nothing!
- Utilization: we use each memory block at most once. No freeing at all, so no memory is ever reused. ☹

Recap: Implicit Free List Allocator

- **Key idea:** in order to reuse blocks, we need a way to track which blocks are allocated and which are free.
- We could store this information in a separate global data structure, but this is inefficient.
- Instead: let's allocate extra space before each block for a **header** storing its payload size and whether it is allocated or free.
- When we allocate a block, we look through the blocks to find a free one, and we update its header to reflect its allocated size and that it is now allocated.
- When we free a block, we update its header to reflect it is now free.
- The header should be 8 bytes (or larger).
- By storing the block size of each block, we *implicitly* have a *list* of free blocks.

Plan for Today

- Explicit Free List Allocator
- Garbage Collection
- **Recap:** Where We've Been
- COMP201 Tools and Techniques
- What's Next?
- Q&A

Disclaimer: Slides for this lecture were borrowed from

—Nick Troccoli's Stanford CS107 class

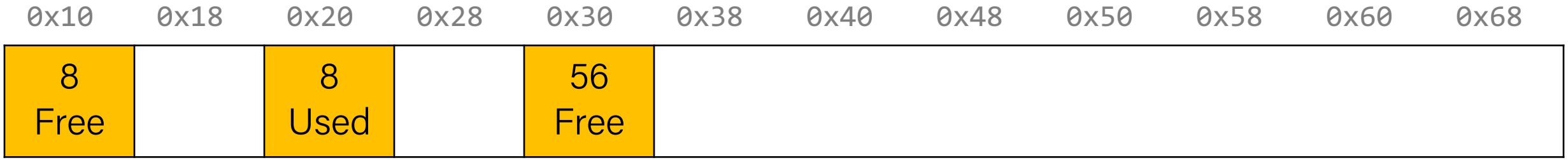
—Ruth Anderson's UW CSE 351 class

Lecture Plan

- Explicit Free List Allocator
- Garbage Collection
- **Recap:** Where We've Been
- COMP201 Tools and Techniques
- What's Next?
- Q&A

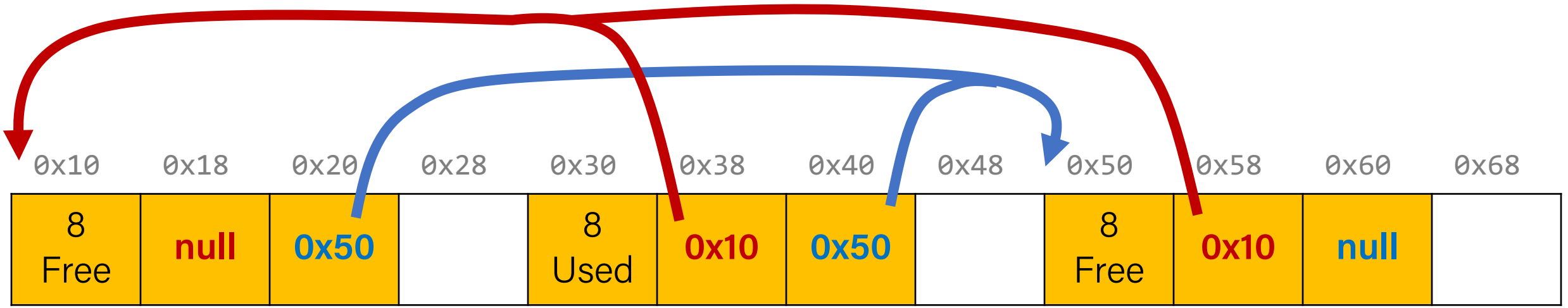
Can We Do Better?

- It would be nice if we could jump *just between free blocks*, rather than all blocks, to find a block to reuse.
- **Idea:** let's modify each header to add a pointer to the previous free block and a pointer to the next free block.



Can We Do Better?

- It would be nice if we could jump *just between free blocks*, rather than all blocks, to find a block to reuse.
- **Idea:** let's modify each header to add a pointer to the **previous** free block and a pointer to the **next** free block.



Can We Do Better?

- It would be nice if we could jump *just between free blocks*, rather than all blocks, to find a block to reuse.
- **Idea:** let's modify each header to add a pointer to the **previous** free block and a pointer to the **next** free block.

This is inefficient – it triples the size of *every* header, when we just need to jump from one free block to another. And even if we just made free headers bigger, it's complicated to have two different header sizes.

Can We Do Better?

- It would be nice if we could jump *just between free blocks*, rather than all blocks, to find a block to reuse.
- **Idea:** let's modify each header to add a pointer to the previous free block and a pointer to the next free block.

Can We Do Better?

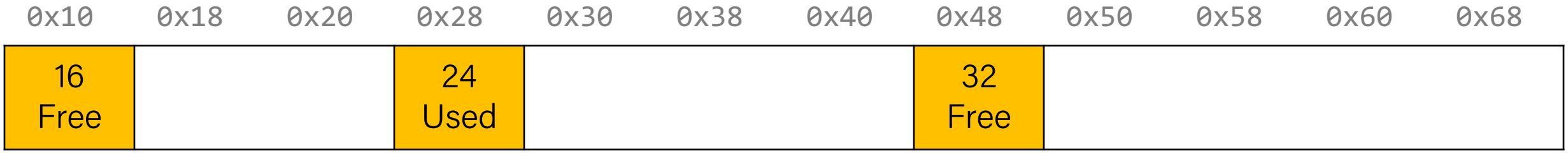
- It would be nice if we could jump *just between free blocks*, rather than all blocks, to find a block to reuse.
- **Idea:** let's modify each header to add a pointer to the previous free block and a pointer to the next free block. *This is inefficient / complicated.*
- **Where can we put these pointers to the next/previous free block?**
- **Idea:** In a separate data structure?

Can We Do Better?

- It would be nice if we could jump *just between free blocks*, rather than all blocks, to find a block to reuse.
- **Idea:** let's modify each header to add a pointer to the previous free block and a pointer to the next free block. *This is inefficient / complicated.*
- **Where can we put these pointers to the next/previous free block?**
- **Idea:** In a separate data structure? *More difficult to access in a separate place – prefer storing near blocks on the heap itself.*

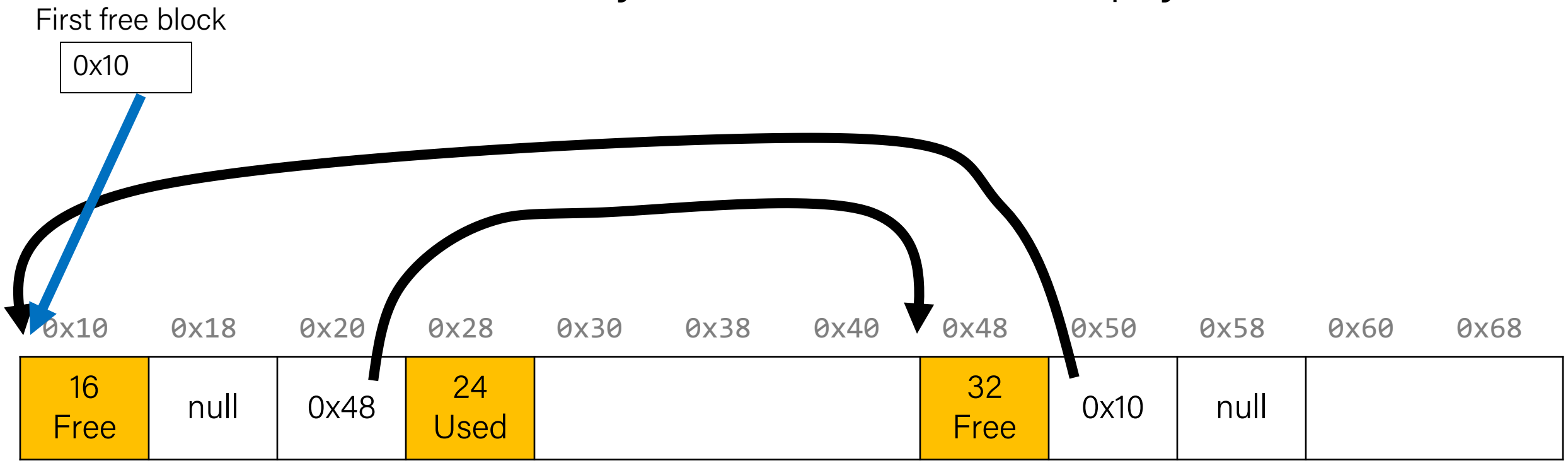
Can We Do Better?

- **Key Insight:** the payloads of the free blocks aren't being used, because they're free.
- **Idea:** since we only need to store these pointers for free blocks, let's store them in the first 16 bytes of each free block's payload!



Can We Do Better?

- **Key Insight:** the payloads of the free blocks aren't being used, because they're free.
- **Idea:** since we only need to store these pointers for free blocks, let's store them in the first 16 bytes of each free block's payload!

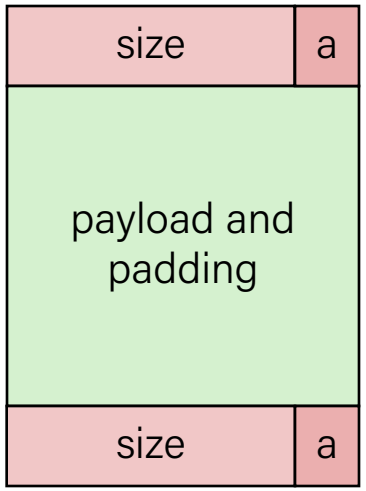


Can We Do Better?

- **Key Insight:** the payloads of the free blocks aren't being used, because they're free.
- **Idea:** since we only need to store these pointers for free blocks, let's store them in the first 16 bytes of each free block's payload!
- This means each payload must be big enough to store 2 pointers (16 bytes). So we must require that for every block, free and allocated. (why?)

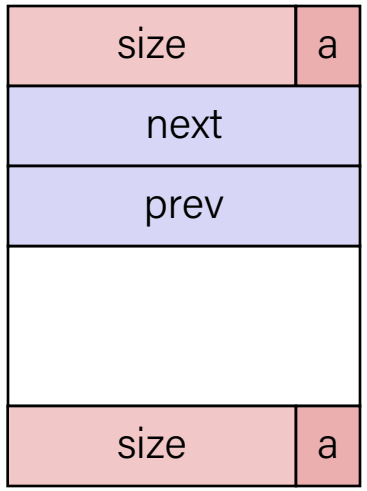
Explicit Free Lists

Allocated block:



(same as implicit free list)

Free block:



- Use list(s) of free blocks, rather than implicit list of all blocks
  - The "next" free block could be anywhere in the heap
    - So we need to store next/previous pointers, not just sizes
  - Since we only track free blocks, so we can use "payload" for pointers
  - Still need boundary tags (header/footer) for coalescing

Explicit Free List Allocator

- This design builds on the implicit allocator, but also stores pointers to the next and previous free block inside each free block's payload.
- When we allocate a block, we look through just the free blocks using our linked list to find a free one, and we update its header and the linked list to reflect its allocated size and that it is now allocated.
- When we free a block, we update its header to reflect it is now free and update the linked list.

This **explicit** list of free blocks increases request throughput, with some costs (design and internal fragmentation)

Explicit Free List: List Design

How do you want to organize your explicit free list?
(compare utilization/throughput)

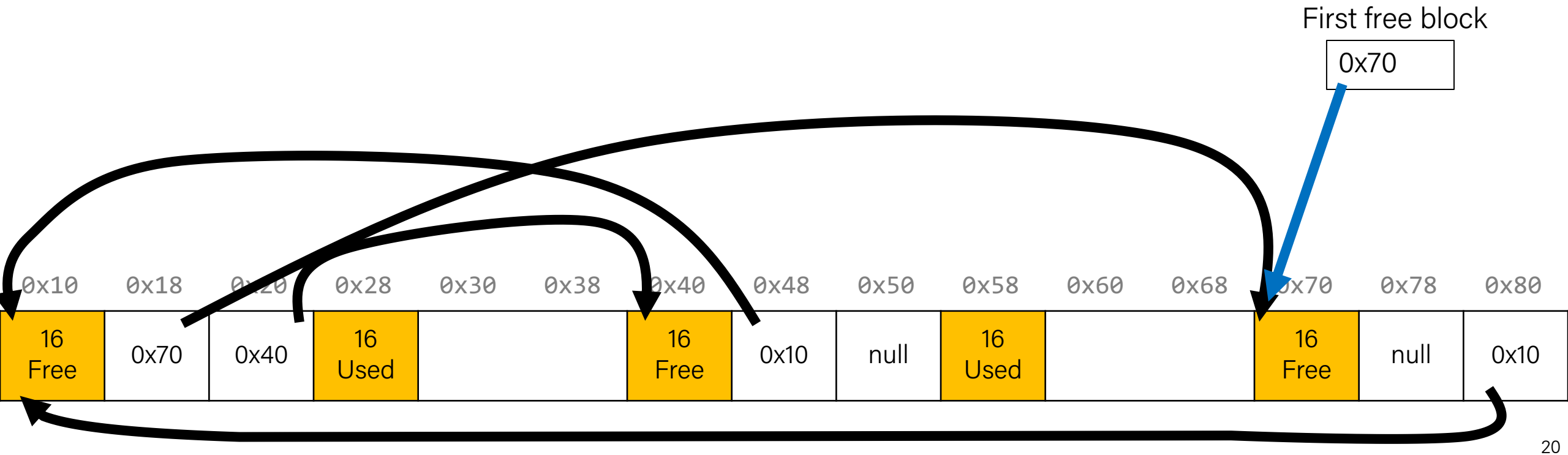
A. Address-order (each block's address is less than successor block's address) — Better memory util, Linear free

B. Last-in first-out (LIFO)/like a stack, where newly freed blocks are at the beginning of the list — Constant free (push recent block onto stack)

C. Other (e.g., by size, etc.) — (more later)

Explicit Free List: List Design

Note that the doubly-linked list *does not have to be in address order*.



Implicit vs. Explicit: So Far

Implicit Free List

- 8B header for size + alloc/free status
- Allocation requests are worst-case linear in total number of blocks
- Implicitly address-order

Explicit Free List

- 8B header for size + alloc/free status
- Free block payloads store prev/next free block pointers
- Allocation requests are worst-case linear in number of free blocks
- Can choose block ordering

Revisiting Our Goals

Can we do better?

1. Can we avoid searching all blocks for free blocks to reuse? **Yes! We can use a doubly-linked list.**
2. Can we merge adjacent free blocks to keep large spaces available?
3. Can we avoid always copying/moving data during `realloc`?

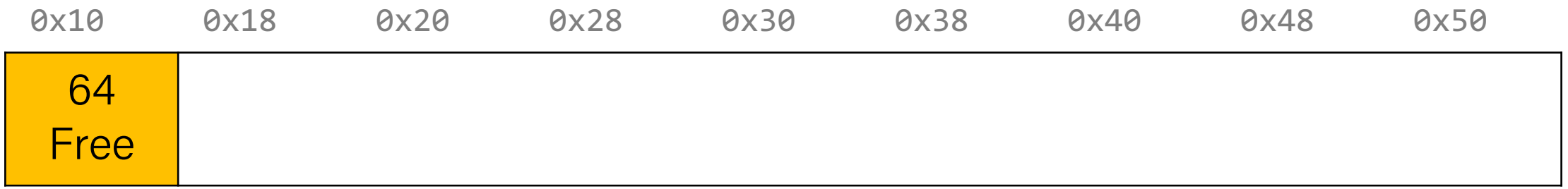
Revisiting Our Goals

Can we do better?

1. Can we avoid searching all blocks for free blocks to reuse? **Yes! We can use a doubly-linked list.**
2. Can we merge adjacent free blocks to keep large spaces available?
3. Can we avoid always copying/moving data during `realloc`?

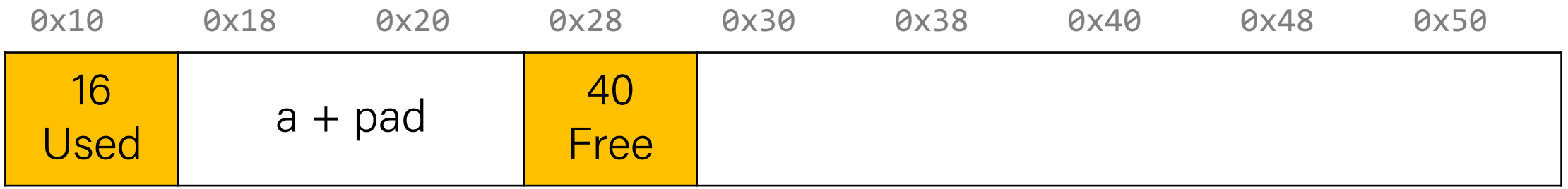
Coalescing

```
void *a = malloc(8);
void *b = malloc(8);
void *c = malloc(16);
free(b);
free(a);
void *d = malloc(32);
```



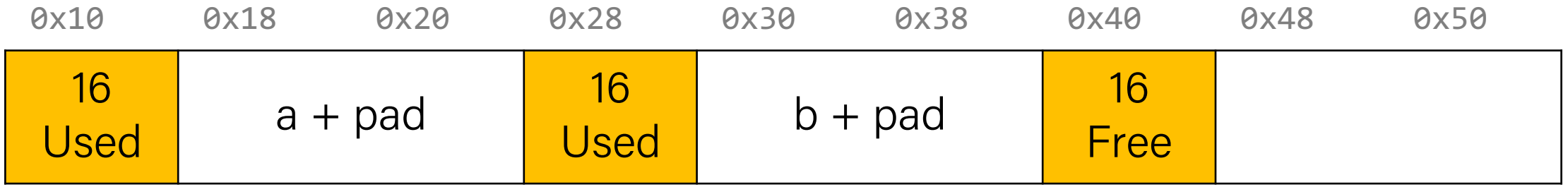
Coalescing

```
void *a = malloc(8);
void *b = malloc(8);
void *c = malloc(16);
free(b);
free(a);
void *d = malloc(32);
```



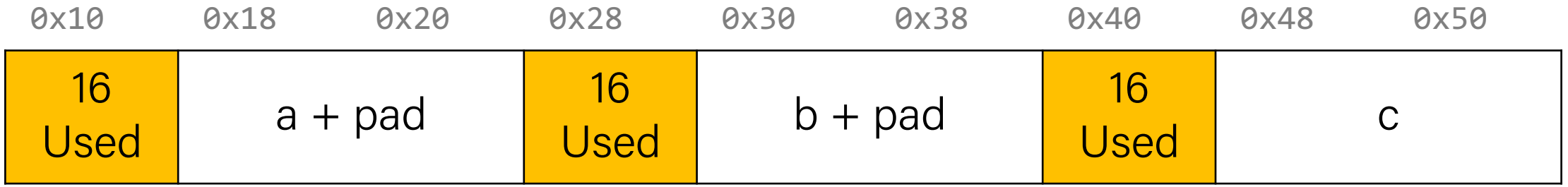
Coalescing

```
void *a = malloc(8);
void *b = malloc(8);
void *c = malloc(16);
free(b);
free(a);
void *d = malloc(32);
```



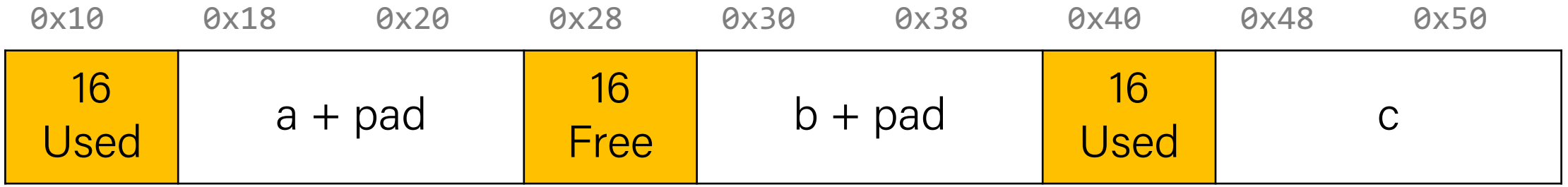
Coalescing

```
void *a = malloc(8);
void *b = malloc(8);
void *c = malloc(16);
free(b);
free(a);
void *d = malloc(32);
```



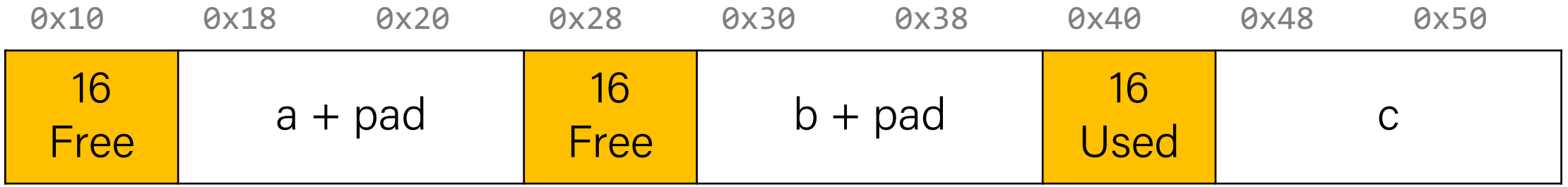
Coalescing

```
void *a = malloc(8);
void *b = malloc(8);
void *c = malloc(16);
free(b);
free(a);
void *d = malloc(32);
```



Coalescing

```
void *a = malloc(8);
void *b = malloc(8);
void *c = malloc(16);
free(b);
free(a);
void *d = malloc(32);
```

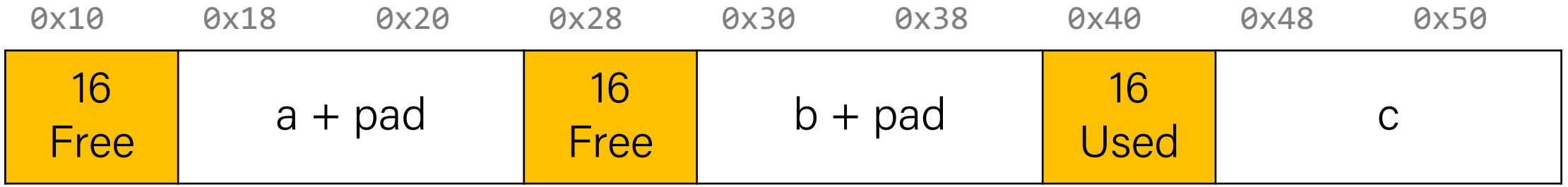


Coalescing

```
void *a = malloc(8);
void *b = malloc(8);
void *c = malloc(16);
free(b);
free(a);
void *d = malloc(32);
```

We have enough memory space, but it is fragmented into free blocks sized from earlier requests!

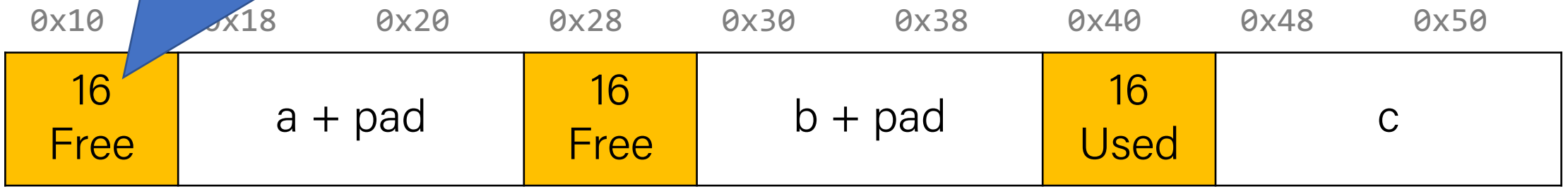
We'd like to be able to merge adjacent free blocks back together. How can we do this?



Coalescing

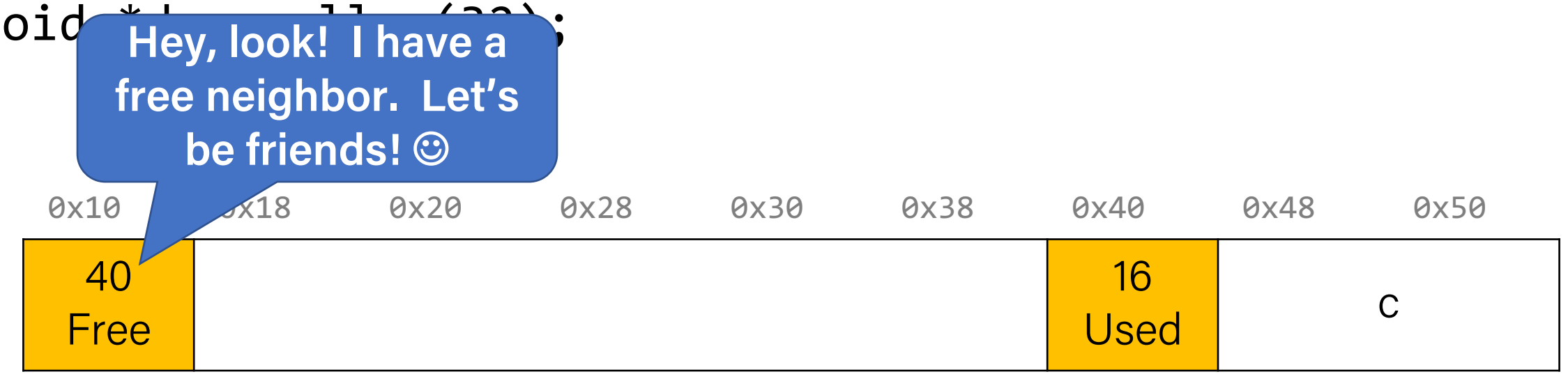
```
void *a = malloc(8);
void *b = malloc(8);
void *c = malloc(16);
free(b);
free(a);
void *d = malloc(32);
```

Hey, look! I have a free neighbor. Let's be friends! ☺



Coalescing

```
void *a = malloc(8);
void *b = malloc(8);
void *c = malloc(16);
free(b);
free(a);
void *d = malloc(32);
```

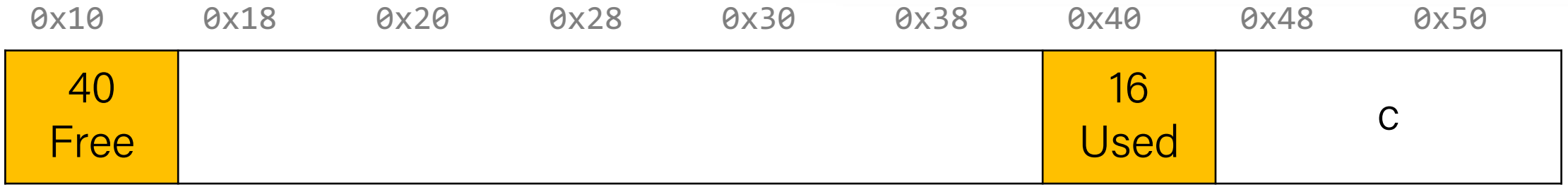


Coalescing

```
void *a = malloc(8);
void *b = malloc(8);
void *c = malloc(16);
free(b);
free(a);
void *d = malloc(32);
```

The process of combining adjacent free blocks is called coalescing.

For your explicit heap allocator, you should coalesce if possible when a block is freed. **You only need to coalesce the most immediate right neighbor.**



Revisiting Our Goals

Can we do better?

1. Can we avoid searching all blocks for free blocks to reuse? **Yes! We can use a doubly-linked list.**
2. Can we merge adjacent free blocks to keep large spaces available? **Yes! We can coalesce on** `free()`.
3. Can we avoid always copying/moving data during `realloc`?

Revisiting Our Goals

Can we do better?

1. Can we avoid searching all blocks for free blocks to reuse? **Yes! We can use a doubly-linked list.**
2. Can we merge adjacent free blocks to keep large spaces available? **Yes! We can coalesce on `free()`.**
3. Can we avoid always copying/moving data during `realloc`?

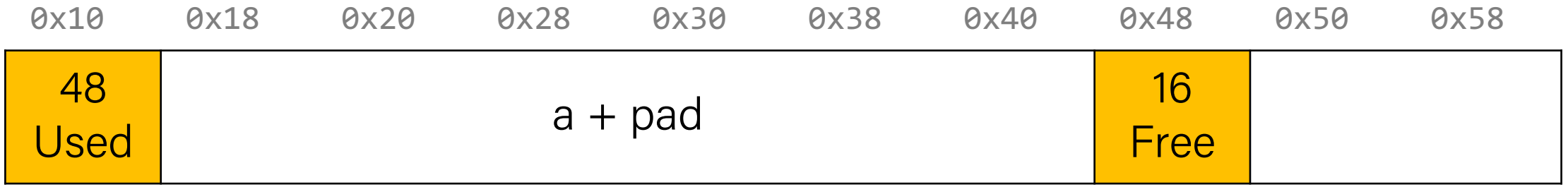
`realloc`

- For the implicit allocator, we didn't worry too much about `realloc`. We always moved data when they requested a different amount of space.
  - Note: `realloc` can grow *or* shrink the data size.
- But sometimes we may be able to keep the data in the same place. How?
  - **Case 1:** size is growing, but we added padding to the block and can use that
  - **Case 2:** size is shrinking, so we can use the existing block
  - **Case 3:** size is growing, and current block isn't big enough, but adjacent blocks are free.

`realloc`: Growing In Place

```
void *a = malloc(42);
...
void *b = realloc(a, 48);
```

a's earlier request was too small, so we added padding. Now they are requesting a larger size we can satisfy with that padding! So `realloc` can return the same address.

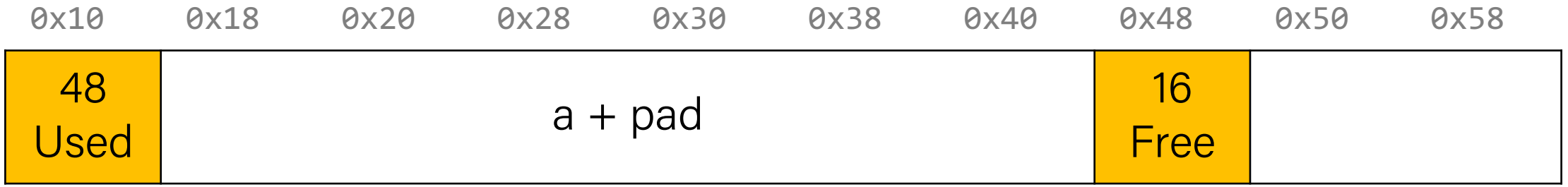


`realloc`: Growing In Place

```
void *a = malloc(42);
...
void *b = realloc(a, 16);
```

If a `realloc` is requesting to shrink, we can still use the same starting address.

If we can, we should try to recycle the now-freed memory into another freed block.

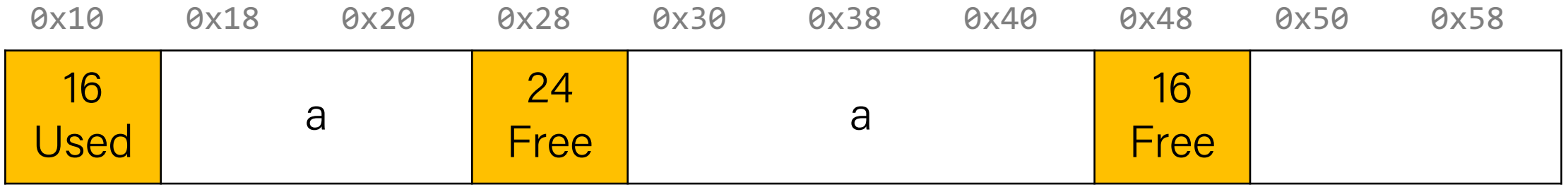


`realloc`: Growing In Place

```
void *a = malloc(42);
...
void *b = realloc(a, 16);
```

If a `realloc` is requesting to shrink, we can still use the same starting address.

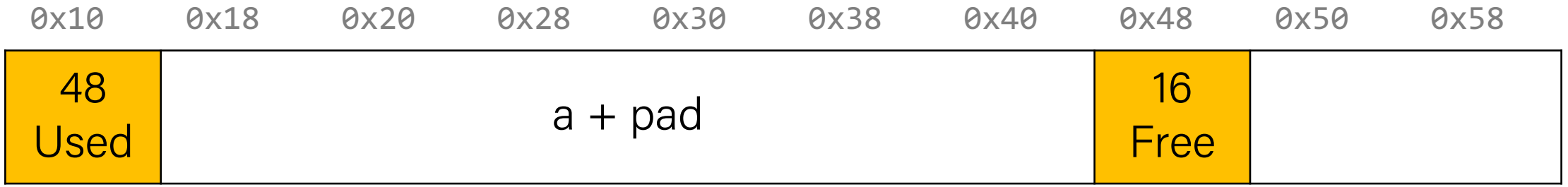
If we can, we should try to recycle the now-freed memory into another freed block.



realloc: Growing In Place

```
void *a = malloc(42);
...
void *b = realloc(a, 72);
```

Even with the padding, we don't have enough space to satisfy the larger size. But we have an adjacent neighbor that is free – let's team up!

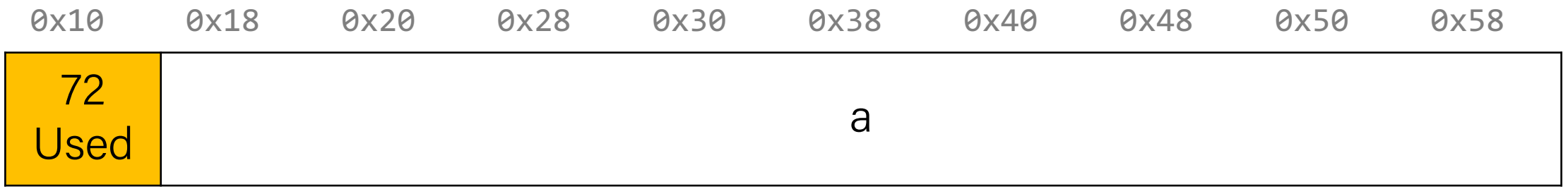


`realloc`: Growing In Place

```
void *a = malloc(42);
...
void *b = realloc(a, 72);
```

Even with the padding, we don't have enough space to satisfy the larger size. But we have an adjacent neighbor that is free – let's team up!

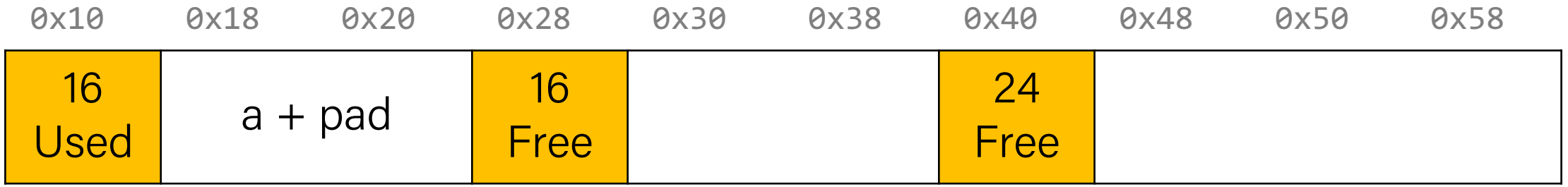
Now we can still return the same address.



`realloc`: Growing In Place

```
void *a = malloc(8);
...
void *b = realloc(a, 72);
```

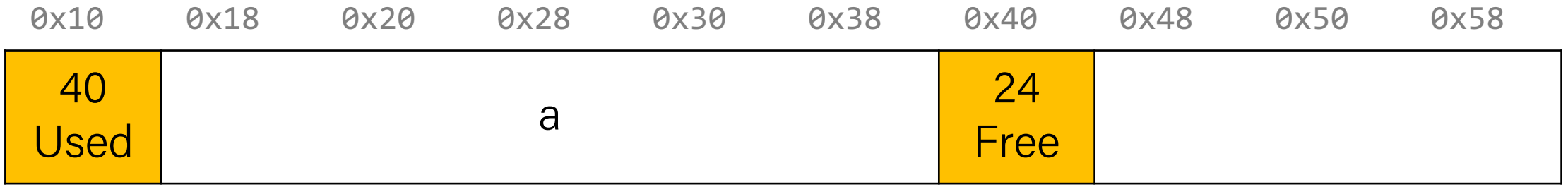
Here, you should combine with your *right* neighbors as much as possible until we get enough space, or until we know we cannot get enough space.



realloc: Growing In Place

```
void *a = malloc(8);
...
void *b = realloc(a, 72);
```

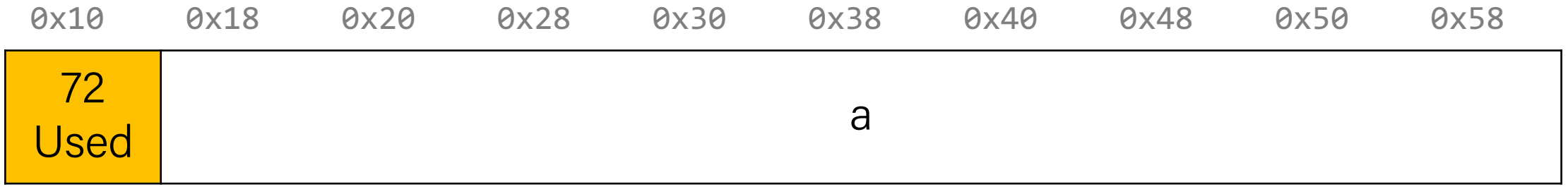
Here, you should combine with your *right* neighbors as much as possible until we get enough space, or until we know we cannot get enough space.



realloc: Growing In Place

```
void *a = malloc(8);
...
void *b = realloc(a, 72);
```

Here, you should combine with your *right* neighbors as much as possible until we get enough space, or until we know we cannot get enough space.



`realloc`

- For the implicit allocator, we didn't worry too much about `realloc`. We always moved data when they requested a different amount of space.
  - Note: `realloc` can grow *or* shrink the data size.
- But sometimes we may be able to keep the data in the same place. How?
  - **Case 1:** size is growing, but we added padding to the block and can use that
  - **Case 2:** size is shrinking, so we can use the existing block
  - **Case 3:** size is growing, and current block isn't big enough, but adjacent blocks are free.
- If you can't do an in-place `realloc`, then you should move the data elsewhere.

Practice 3

- For the following heap layout, what would the heap look like after the following request is made, assuming we are using an **explicit** free list allocator with a **first-fit** approach and **coalesce on free + realloc in-place**?

[24 byte payload, allocated for B] [16 byte payload, free] [16 byte payload, allocated for A]

```
free(B);
```

[48 byte payload, free] [16 byte payload, allocated for A]

Practice 4

- For the following heap layout, what would the heap look like after the following request is made, assuming we are using an **explicit** free list allocator with a **first-fit** approach and **coalesce on free + realloc in-place**?

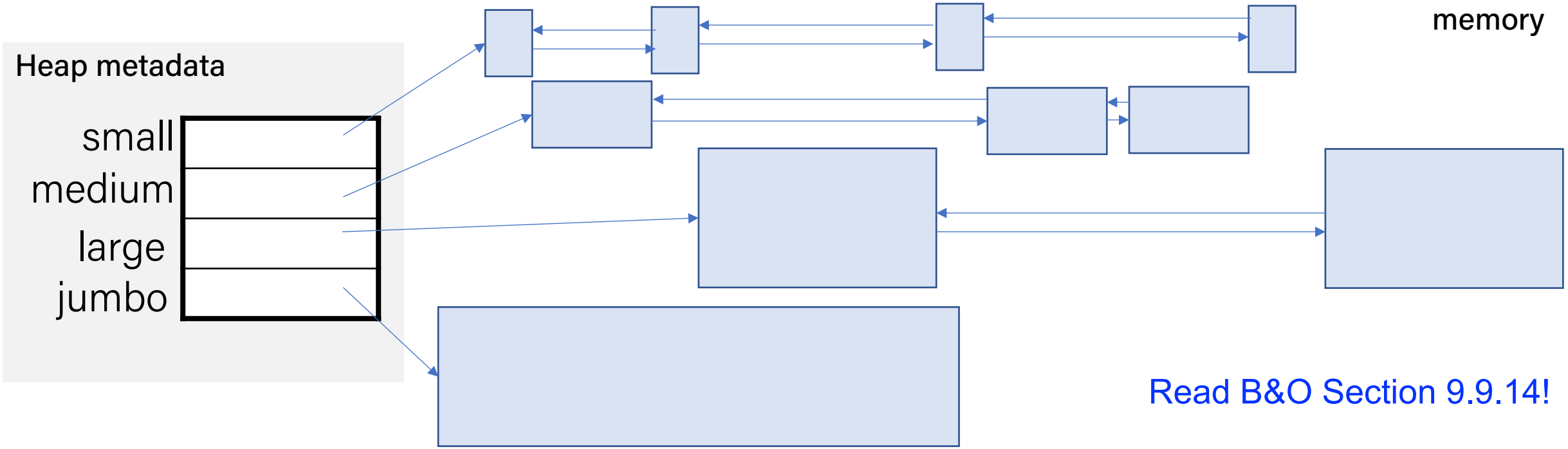
[16 byte payload, allocated for A] [32 byte payload, free] [16 byte payload, allocated for B]

```
realloc(A, 24);
```

[24 byte payload, allocated for A] [24 byte payload, free] [16 byte payload, allocated for B]

Going beyond: Explicit list w/size buckets

- Explicit lists are much faster than implicit lists.
- However, a first-fit placement policy is still linear in total # of free blocks.
- What about an explicit free list **sorted by size** (e.g., as a tree)?
- What about several explicit free lists **bucketed by size**? (below)



Read B&O Section 9.9.14!

More Info on Allocators

- D. Knuth, "*The Art of Computer Programming*", 2nd edition, Addison Wesley, 1973
  - The classic reference on dynamic storage allocation

- Wilson et al, "*Dynamic Storage Allocation: A Survey and Critical Review*", Proc. 1995 Int'l Workshop on Memory Management, Kinross, Scotland, Sept, 1995.
  - Comprehensive survey
  - Available from CS:APP student site (csapp.cs.cmu.edu)

Wouldn’t it be nice…

- If we never had to free memory?
- Do you free objects in Java?
  - Reminder: *implicit* allocator

Lecture Plan

- Explicit Free List Allocator
- Garbage Collection
- **Recap:** Where We've Been
- COMP201 Tools and Techniques
- What's Next?
- Q&A

Garbage Collection (GC) (Automatic Memory Management)

- **Garbage collection:** automatic reclamation of heap-allocated storage – application never explicitly frees memory

```
void foo() {
   int* p = (int*) malloc(128);
   return;  /* p block is now garbage! */
}
```

- Common in implementations of functional languages, scripting languages, and modern object oriented languages:
  - Lisp, Racket, Erlang, ML, Haskell, Scala, Java, C#, Perl, Ruby, Python, Lua, JavaScript, Dart, Mathematica, MATLAB, many more…

- Variants ("conservative" garbage collectors) exist for C and C++
  - However, cannot necessarily collect all garbage

Garbage Collection

- How does the memory allocator know when memory can be freed?
  - In general, we cannot know what is going to be used in the future since it depends on conditionals
  - But, we can tell that certain blocks cannot be used if they are *unreachable* (via pointers in registers/stack/globals)

- Memory allocator needs to know what is a pointer and what is not – how can it do this?
  - Sometimes with help from the compiler

Classical GC Algorithms

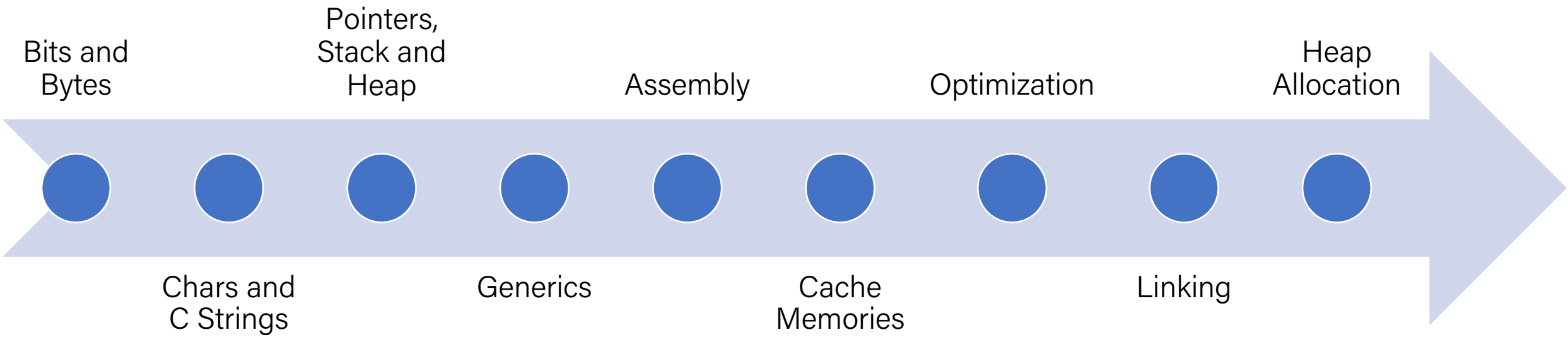
- **Mark-and-sweep collection** (McCarthy, 1960)
  - Does not move blocks (unless you also "compact")
- Reference counting (Collins, 1960)
  - Does not move blocks (not discussed)
- Copying collection (Minsky, 1963)
  - Moves blocks (not discussed)
- Generational Collectors (Lieberman and Hewitt, 1983)
  - Most allocations become garbage very soon, so focus reclamation work on zones of memory recently allocated.
- For more information:
  - Jones, Hosking, and Moss, *The Garbage Collection Handbook: The Art of Automatic Memory Management*, CRC Press, 2012.
  - Jones and Lin, *Garbage Collection: Algorithms for Automatic Dynamic Memory*, John Wiley & Sons, 1996.

Lecture Plan

- Explicit Free List Allocator
- Garbage Collection
- **Recap:** Where We've Been
- COMP201 Tools and Techniques
- What's Next?
- Q&A

We've covered *a lot* in just 14 weeks! Let's take a look back.

Our COMP201 Journey



Course Overview

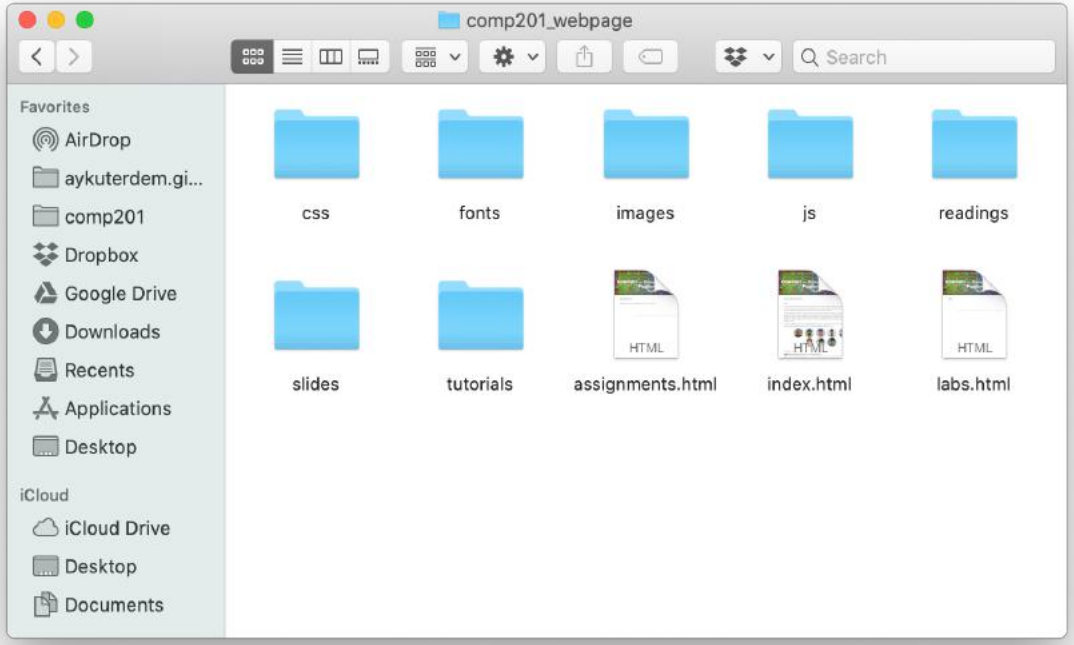
1. **Bits and Bytes** - *How can a computer represent integer and float numbers?*
2. **Chars and C-Strings** - *How can a computer represent and manipulate more complex data like text?*
3. **Pointers, Stack and Heap –** *How can we effectively manage all types of memory in our programs?*
4. **Generics** - *How can we use our knowledge of memory and data representation to write code that works with any data type?*
5. **Assembly** - *How does a computer interpret and execute C programs?*
6. **Cache Memories** - *How does the memory system is organized as a hierarchy of different storage devices with unique capacities?*
7. **Optimization**- *How we can optimize our code to improve efficiency and speed? The optimizations GCC can perform.*
8. **Linking**- *How to construct programs from multiple object files?*
9. **Heap Allocators** - *How do core memory-allocation operations like malloc and free work?*

First Day

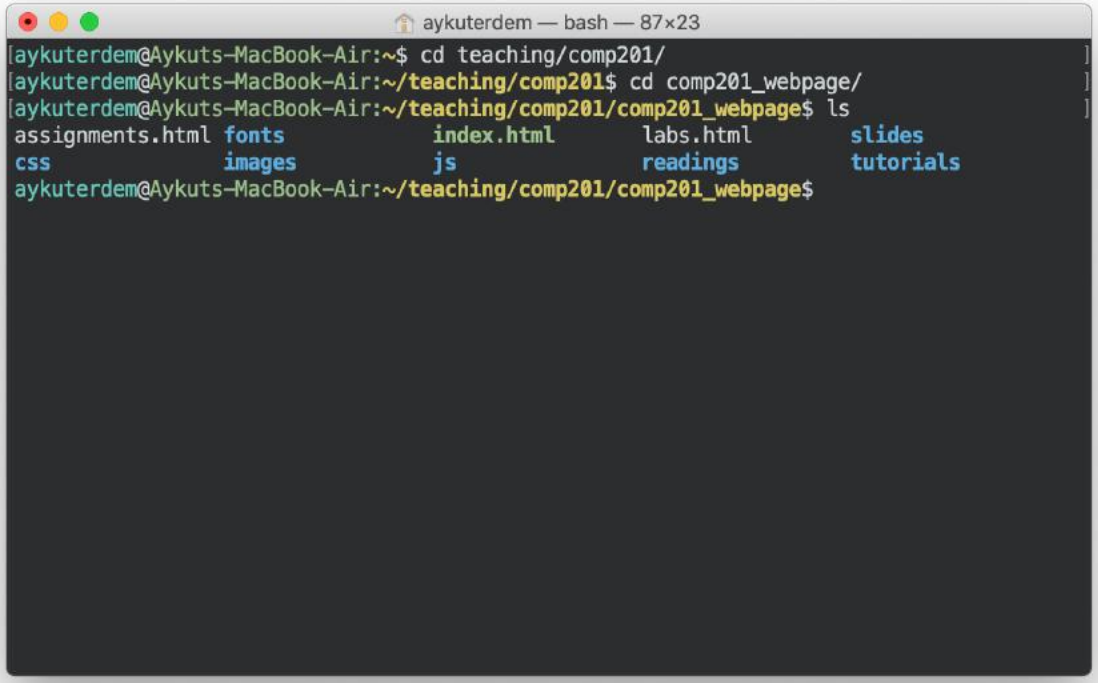
```
/*
 * hello.c
 * This program prints a welcome message
 * to the user.
 */
#include <stdio.h>   // for printf

int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    return 0;
}
```

First Day

- The **command-line** is a text-based interface to navigate a computer, instead of a Graphical User Interface (GUI).



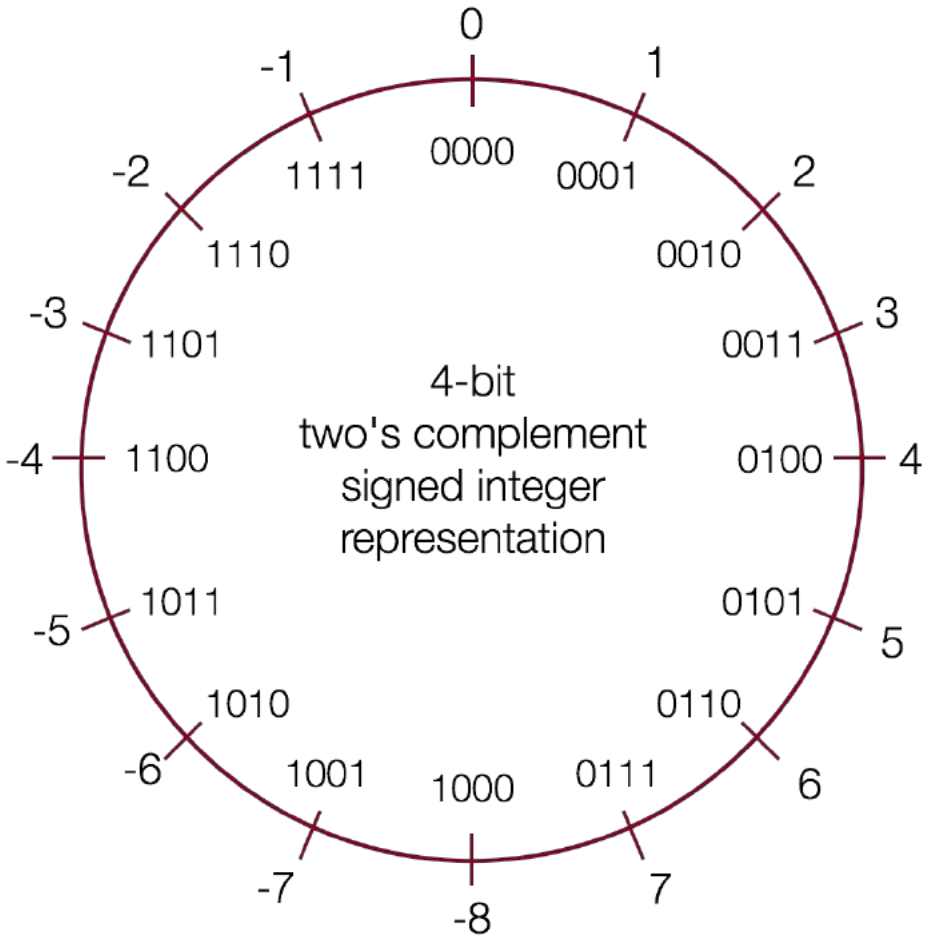
Graphical User Interface



Text-based interface

Bits And Bytes

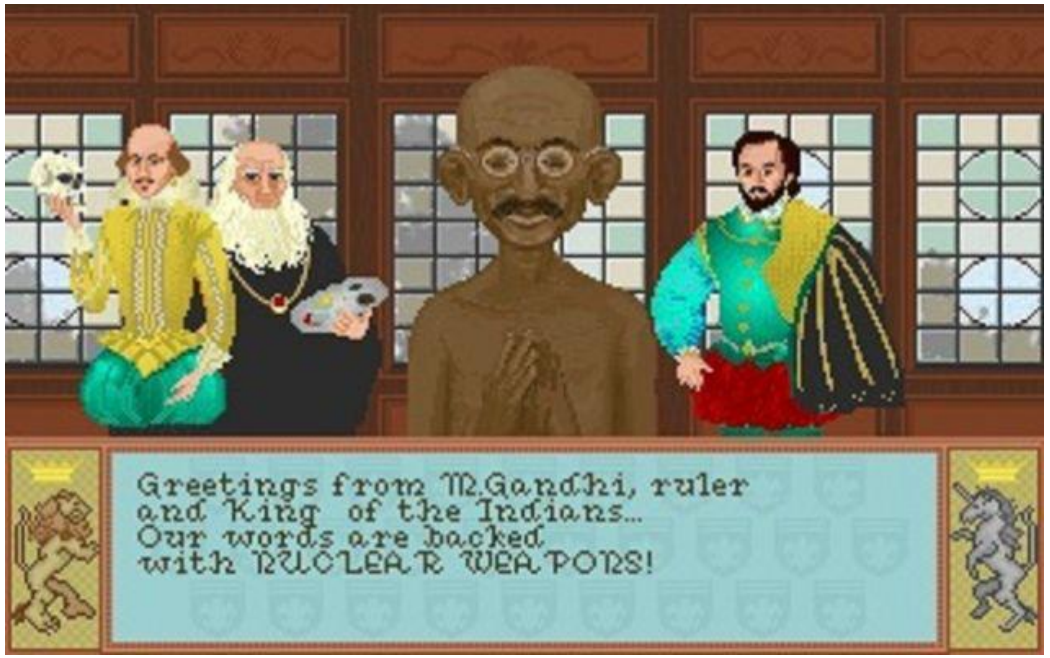
Key Question: *How can a computer represent integer numbers?*



Bits And Bytes

Why does this matter?

- Limitations of representation and arithmetic impact programs!
- We can also efficiently manipulate data using bits.



https://kotaku.com/why-gandhi-is-such-an-asshole-in-civilization-1653818245

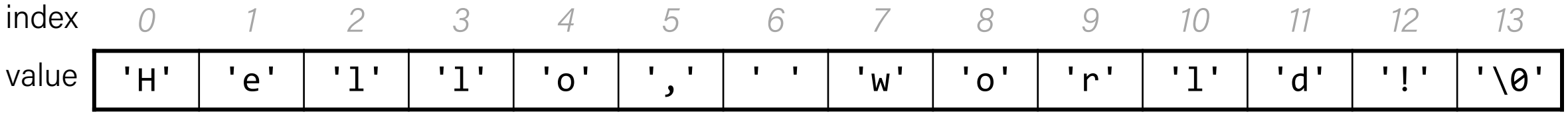
Floats

- IEEE Floating Point is a carefully-thought-out standard. It's complicated, but engineered for their goals.
- Floats have an extremely wide range, but cannot represent every number in that range.
- Some approximation and rounding may occur! This means you definitely don't want to use floats e.g. for currency.
- Associativity does not hold for numbers far apart in the range
- Equality comparison operations are often unwise.

C Strings

Key Question: *How can a computer represent and manipulate more complex data like text?*

- Strings in C are arrays of characters ending with a null terminator!
- We can manipulate them using pointers and C library functions (many of which you could probably implement).



C Strings

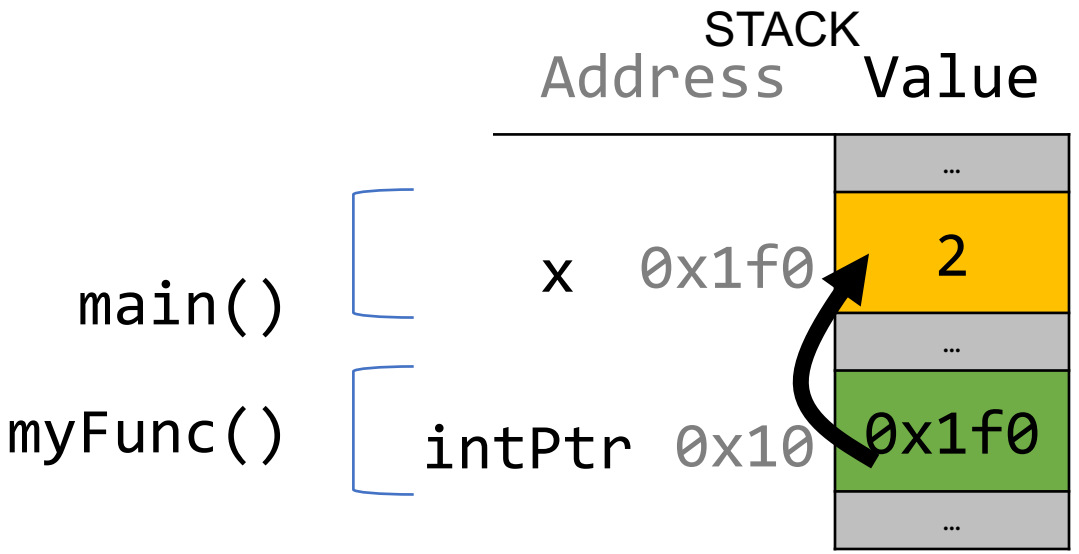
Why does this matter?

- Understanding this representation is key to efficient string manipulation.
- This is how strings are represented in both low- and high-level languages!
  - C++: https://www.quora.com/How-does-C++-implement-a-string
  - Python: https://www.laurentluce.com/posts/python-string-objects-implementation/

Pointers, Stack and Heap

Key Question: *How can we effectively manage all types of memory in our programs?*

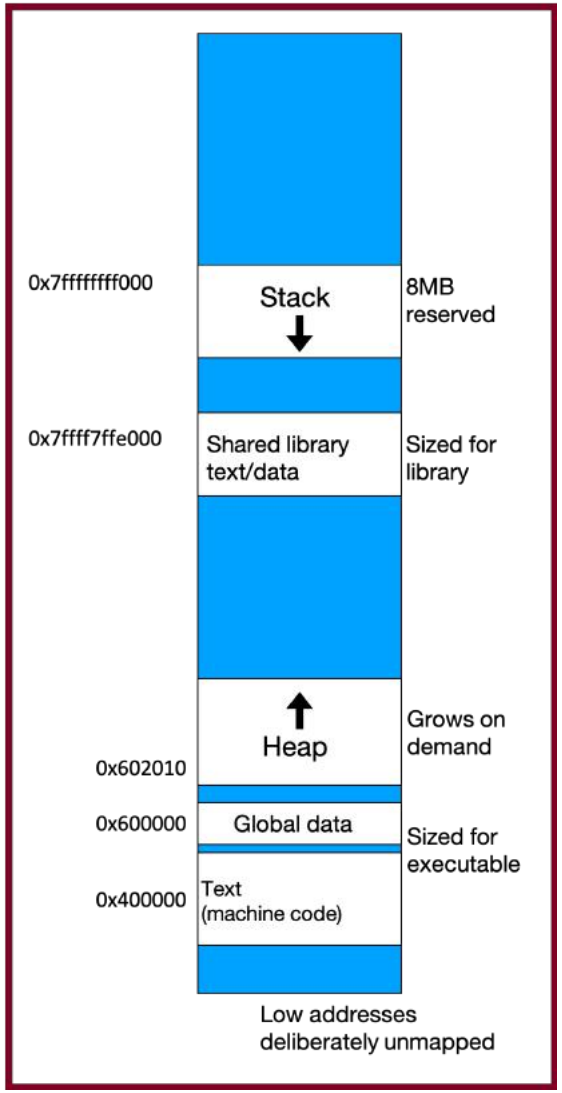
- Arrays let us store ordered lists of information.
- Pointers let us pass addresses of data instead of the data itself.
- We can use the stack, which cleans up memory for us, or the heap, which we must manually manage.



Stack And Heap

Why does this matter?

- The stack and heap allow for two ways to store data in our programs, each with their own tradeoffs, and it's crucial to understand the nuances of managing memory in any program you write!
- Pointers let us pass around references to data efficiency



Generics

Key Question: *How can we use our knowledge of memory and data representation to write code that works with any data type?*

- We can use `void *` to circumvent the type system, `memcpy`, etc. to copy generic data, and function pointers to pass logic around.

Why does this matter?

- Working with any data type lets us write more generic, reusable code.
- Using generics helps us better understand the type system in C and other languages, and where it can help and hinder our program.

Assembly Language

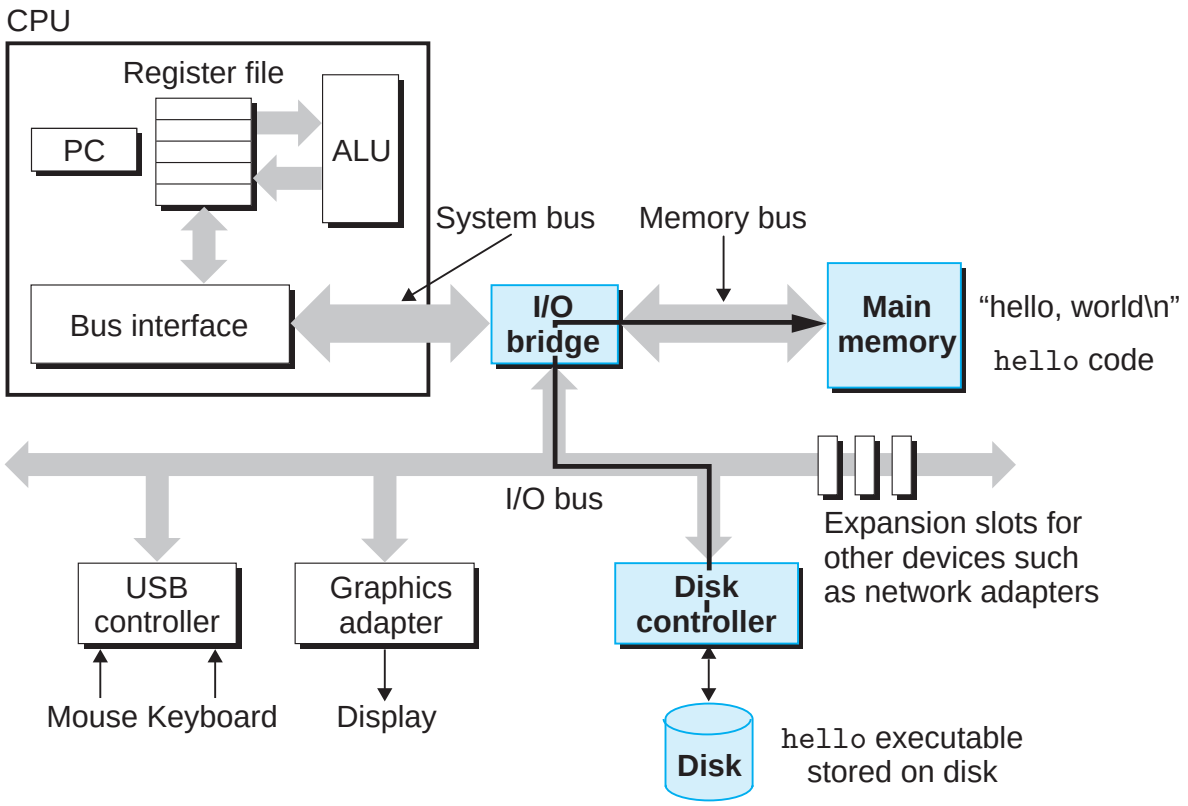
Key Question: *How does a computer interpret and execute C programs?*

- GCC compiles our code into *machine code instructions* executable by our processor.
- Our processor uses registers and instructions like **mov** to manipulate data.

Assembly Language

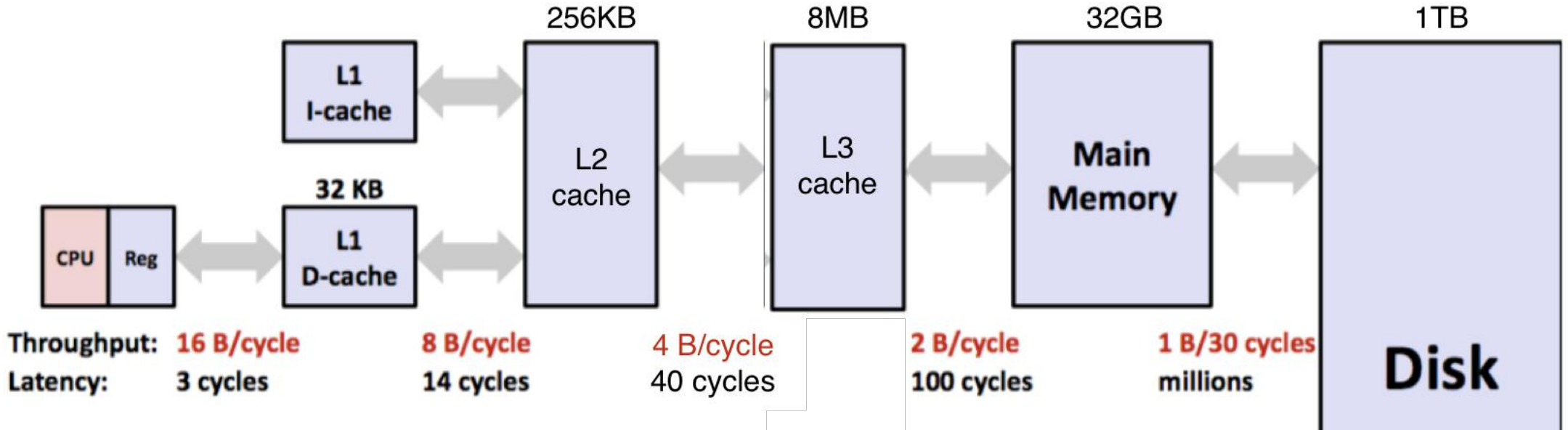
Why does this matter?

- We write C code because it is higher level and transferrable across machines. But it is not the representation executed by the computer!
- Understanding how programs are compiled and executed, as well as computer architecture, is key to writing performant programs (e.g. fewer lines of code is not necessarily better).
- We can reverse engineer and exploit programs at the assembly level!



Caching

- Processor speed is not the only bottleneck in program performance – memory access is perhaps even more of a bottleneck!
- Memory exists in levels and goes from *really fast* (registers) to *really slow* (disk).
- As data is more frequently used, it ends up in faster and faster memory.



Caching

Why does this matter?

Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer.

- **Temporal locality**
  - Repeat access to the same data tends to be co-located in TIME
  - Intuitively: things I have used recently, I am likely to use again soon
- **Spatial locality**
  - Related data tends to be co-located in SPACE
  - Intuitively: data that is near a used item is more likely to also be accessed

Linking

- Linking is a technique that allows programs to be constructed from multiple object files.
- Linking can happen at different times in a program's lifetime:
  - Compile time (when a program is compiled)
  - Load time (when a program is loaded into memory)
  - Run time (while a program is executing)

Why does this matter?

- Understanding linking can help you avoid nasty errors and make you a better programmer.

Heap Allocators

Key Question: *How do core memory-allocation operations like malloc and free work?*

- A *heap allocator* manages a block of memory for the heap and completes requests to use or give up memory space.
- We can manage the data in a heap allocator using headers, pointers to free blocks, or other designs

Why does this matter?

- Designing a heap allocator requires making many design decisions to optimize it as much as possible. There is no perfect design!
- All languages have a "heap" but manipulate it in different ways.

COMP201 Learning Goals

The goals for COMP201 are for students to gain **mastery** of

- writing C programs with complex use of memory and pointers
- an accurate model of the address space and compile/runtime behavior of C programs

to achieve **competence** in

- translating C to/from assembly
- writing programs that respect the limitations of computer arithmetic
- finding bottlenecks and improving runtime performance
- working effectively in a Unix development environment

and have **exposure** to

- a working understanding of the basics of cache memories

Lecture Plan

- Explicit Free List Allocator
- Garbage Collection
- **Recap:** Where We've Been
- COMP201 Tools and Techniques
- What's Next?
- Q&A

Tools and Techniques

- Unix and the command line
- Coding Style
- Debugging (GDB)
- Testing (Sanity Check)
- Memory Checking (Valgrind)

Unix And The Command Line

Unix and command line tools are extremely popular tools outside of COMP201 for:

- Running programs (web servers, python programs, remote programs…)
- Accessing remote servers (Amazon Web Services, Microsoft Azure, Heroku…)
- Programming embedded devices (Raspberry Pi, etc.)

Our goal for COMP201 was to help you become proficient in navigating Unix

Coding Style

- Writing clean, readable code is crucial for any computer science project
- Unfortunately, a fair amount of existing code is poorly-designed/documented

Our goal for COMP201 was to help you write with good coding style, and read/understand/comment provided code.

Debugging (GDB)

- Debugging is a crucial skill for any computer scientist
- Our goal for COMP201 was to help you become a better debugger
  - narrow in on bugs
  - diagnose the issue
  - implement a fix
- Practically every project you work on will have debugging facilities
  - Python: “PDB”
  - IDEs: built-in debuggers (e.g. QT Creator, Eclipse)
  - Web development: in-browser debugger

Testing (Sanity Check)

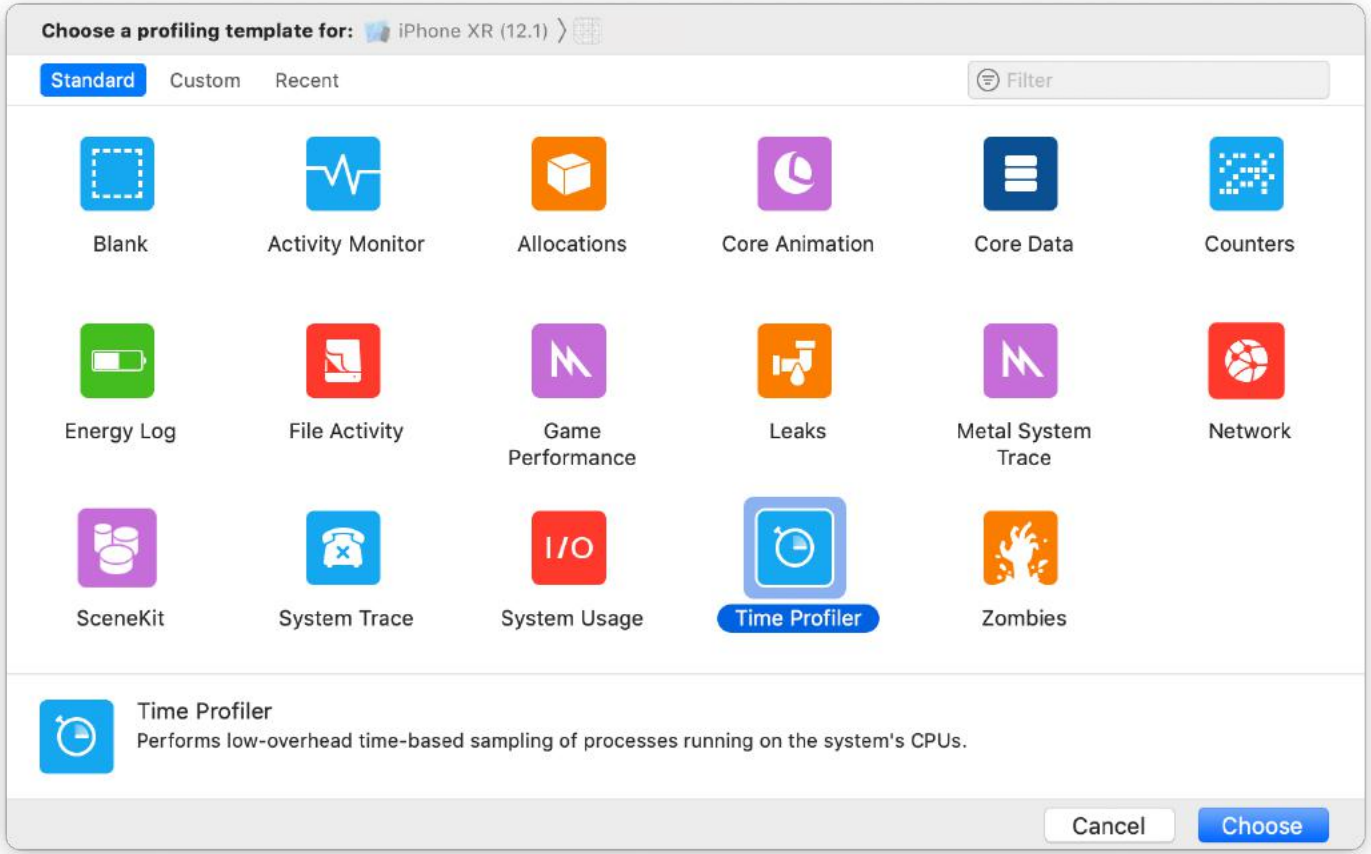
- Testing is a crucial skill for any computer scientist
- Our goal for COMP201 was to help you become a better tester
  - Writing targeted tests to validate correctness
  - Use tests to prevent regressions
  - Use tests to develop incrementally

Memory Checking and Profiling

- Memory checking and profiling are crucial for any computer scientist to analyze program performance and increase efficiency.
- Many projects you work on will have profiling and memory analysis facilities:
  - Mobile development: integrated tools (XCode Instruments, Android Profiler, etc.)
  - Web development: in-browser tools

Tools

You'll see manifestations of these tools throughout projects you work on. We hope you can use your COMP201 knowledge to take advantage of them!

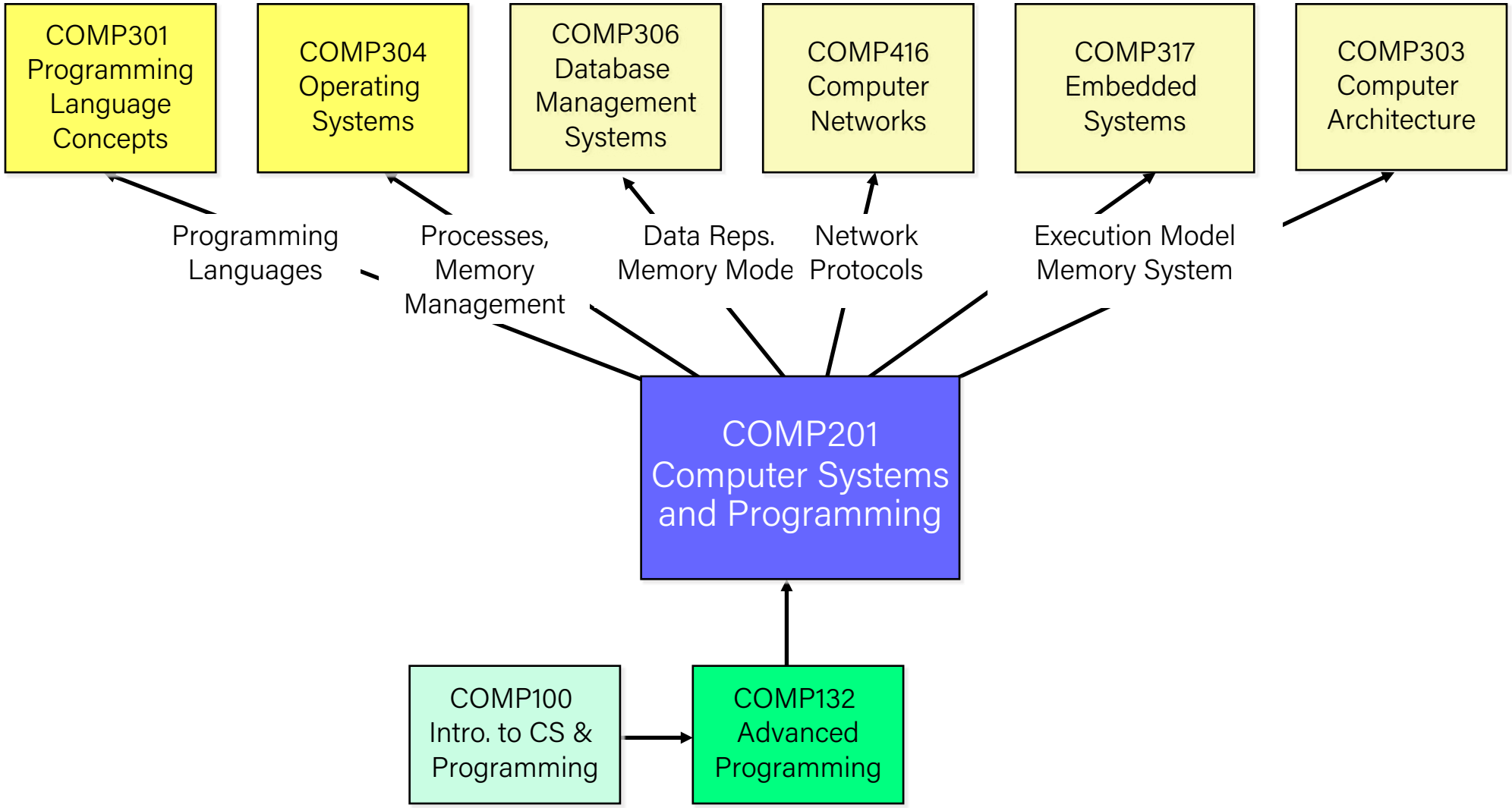


Lecture Plan

- Explicit Free List Allocator
- Garbage Collection
- **Recap:** Where We've Been
- Larger Applications
- What's Next?
- Q&A

After COMP201, you are prepared to take a variety of classes in various areas. What are some options?

Role within COMP Curriculum



Courses

How is an operating system implemented? (take COMP304!)

- Threads
- User Programs
- Virtual Memory
- Filesystem

How is a programming language designed? (take COMP301!)

- Lexical analysis
- Parsing
- Semantic Analysis
- Code Generation

Courses

What are the principles of designing computer hardware (take COMP303)

- Computer organization

How can we write programs that execute on special hardware? (take COMP317!)

- Embedded systems

How can applications communicate over a network? (take COMP416!)

- How can we weigh different tradeoffs of network architecture design?
- How can we effectively transmit bits across a network?

Machine Learning

Can we speed up machine learning training with reduced precision computation?

- https://www.top500.org/news/ibm-takes-aim-at-reduced-precision-for-new-generation-of-ai-chips/
- https://devblogs.nvidia.com/mixed-precision-training-deep-neural-networks/

How can we implement performant machine learning libraries?

- Popular tools such as TensorFlow and PyTorch are implemented using C!
- https://pytorch.org/blog/a-tour-of-pytorch-internals-1/
- https://www.tensorflow.org/guide/extend/architecture

Web Development

How can we efficiently translate Javascript code to machine code?

- The Chrome V8 JavaScript engine converts Javascript into machine code for computers to execute: https://medium.freecodecamp.org/understanding-the-core-of-nodejs-the-powerful-chrome-v8-engine-79e7eb8af964
- The popular Node.js web server tool is built on Chrome V8

How can we compile programs into an efficient binary instruction format that runs in a web browser?

- WebAssembly is an emerging standard instruction format that runs in browsers: https://webassembly.org
- You can compile C/C++/other languages into WebAssembly for web execution

Programming Languages / Runtimes

How can programming languages and runtimes efficiently manage memory?

- Manual memory management (C/C++)
- Reference Counting (Swift)
- Garbage Collection (Java)

How can we design programming languages to reduce the potential for programmer error?

- Haskell/Swift "Optionals"

How can we design portable programming languages?

- Java Bytecode: https://en.wikipedia.org/wiki/Java_bytecode

Theory

How can compilers output efficient machine code instructions for programs?

- Languages can be represented as regular expressions and context-free grammars
- We can model programs as control-flow graphs for additional optimization

Security

How can we find / fix vulnerabilities at various levels in our programs?

- Understand machine-level representation and data manipulation
- Understand how a computer executes programs
- macOS High Sierra Root Login Bug: https://objective-see.com/blog/blog_0x24.html

Lecture Plan

- Explicit Free List Allocator
- Garbage Collection
- **Recap:** Where We've Been
- Larger Applications
- What's Next?
- Q&A

Course Evaluations

- I hope you can take the time to fill out the end-semester COMP201 course evaluation.
- I sincerely appreciate any feedback you have about the course and read every piece of feedback we receive.
- I am always looking for ways to improve!

Q&A