
# Memory Allocation

2023-04-13 — 29 MINUTE READ

One thing that all programs on your computer have in common is a need for memory. Programs need to be loaded from your hard drive into memory before they can be run. While running, the majority of what programs do is load values from memory, do some computation on them, and then store the result back in memory.

In this post I'm going to introduce you to the basics of memory allocation. Allocators exist because it's not enough to have memory available, you need to use it effectively. We will visually explore how simple allocators work. We'll see some of the problems that they try to solve, and some of the techniques used to solve them. At the end of this post, you should know everything you need to know to write your own allocator.

## › `malloc` and `free`

To understand the job of a memory allocator, it's essential to understand how programs request and return memory. **`malloc`** and **`free`** are functions that were first introduced in a recognisable form in UNIX v7 in 1979(!). Let's take a look at a short C program demonstrating their use.

*Woah, hold on. I've never written any C code before. Will I still be able to follow along?*

If you have beginner-level familiarity with another language, e.g. JavaScript, Python, or C#, you should have no problem following along. You don't need to understand every word, as long as you get the overall idea. This is the only C code in the article, I promise.

```c
#include <stdlib.h>

int main() {
  void *ptr = malloc(4);
  free(ptr);
  return 0;
}
```

In the above program we ask for 4 bytes of memory by calling **`malloc(4)`**, we store the value returned in a variable called **`ptr`**, then we indicate that we're done with the memory by calling **`free(ptr)`**.

These two functions are how almost all programs manage the memory they use. Even when you're not writing C, the code that is executing your Java, Python, Ruby, JavaScript, and so on make use of **`malloc`** and **`free`**.

## › What is memory?

The smallest unit of memory that allocators work with is called a "byte." A byte can store any number between 0 and 255. You can think of memory as being a long sequence of bytes. We're going to represent this sequence as a grid of squares, with each square representing a byte of memory.

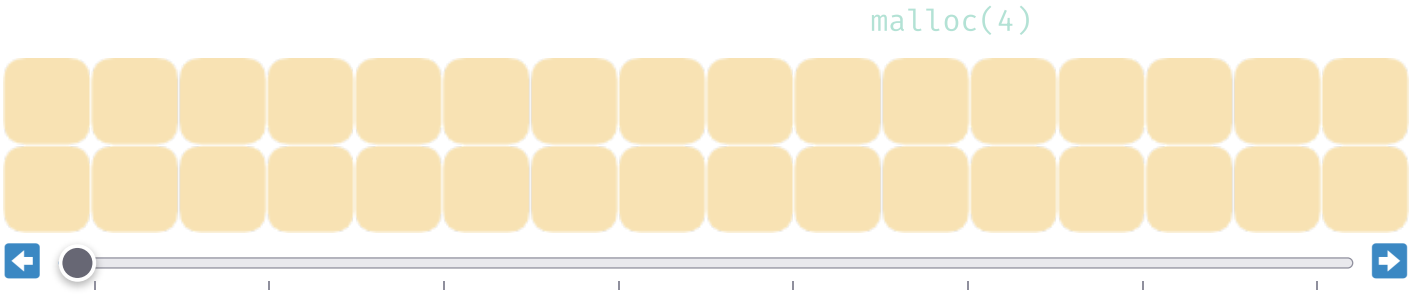


In the C code from before, **`malloc(4)`** allocates 4 bytes of memory. We're going to

represent memory that has been allocated as darker squares.

Then **`free(ptr)`** tells the allocator we're done with that memory. It is returned back to the pool of available memory.

Here's what 4 **`malloc`** calls followed by 4 **`free`** calls looks like. You'll notice there's now a slider. Dragging the slider to the right advances time forward, and dragging it left rewinds. You can also click anywhere on the grid and then use the arrow keys on your keyboard, or you can use the left and right buttons. The ticks along the slider represent calls to **`malloc`** and **`free`**.




*Wait a sec... What is **`malloc`** actually returning as a value? What does it mean to "give" memory to a program?*

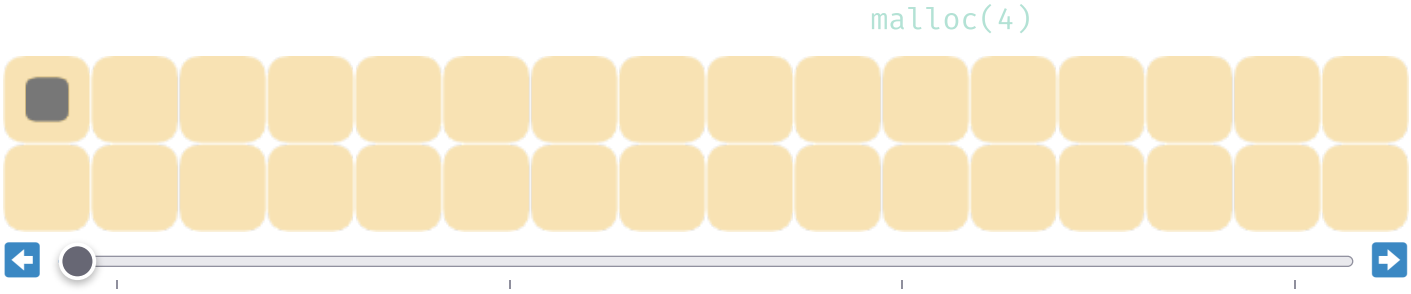
What **`malloc`** returns is called a "pointer" or a "memory address." It's a number that identifies a byte in memory. We typically write addresses in a form called "hexadecimal." Hexadecimal numbers are written with a **`0x`** prefix to distinguish them from decimal numbers. Move the slider below to see a comparison between decimal numbers and hexadecimal numbers.

Here's our familiar grid of memory. Each byte is annotated with its address in hexadecimal form. For space reasons, I've omitted the `0x` prefix.


The examples we use in this article pretend that your computer only has a very small amount of memory, but in real life you have billions of bytes to work with. Real addresses are much larger than what we're using here, but the idea is exactly the same. Memory addresses are numbers that refer to a specific byte in memory.

## › The simplest `malloc`

The "hello world" of `malloc` implementations would hand out blocks of memory by keeping track of where the previous block ended and starting the next block right after. Below we represent where the next block should start with a grey square.




You'll notice no memory is `free`d. If we're only keeping track of where the next block should start, and we don't know where previous blocks start or end, `free` doesn't have enough information to do anything. So it doesn't. This is called a "memory leak" because, once allocated, the memory can never be used again.

Believe it or not, this isn't a completely useless implementation. For programs that use a known amount of memory, this can be a very efficient strategy. It's extremely fast and extremely simple. As a general-purpose memory allocator, though, we can't get away with having no `free` implementation.

# › The simplest general-purpose `malloc`

In order to **`free`** memory, we need to keep better track of memory. We can do this by saving the address and size of all allocations, and the address and size of blocks of free memory. We'll call these an "allocation list" and a "free list" respectively.

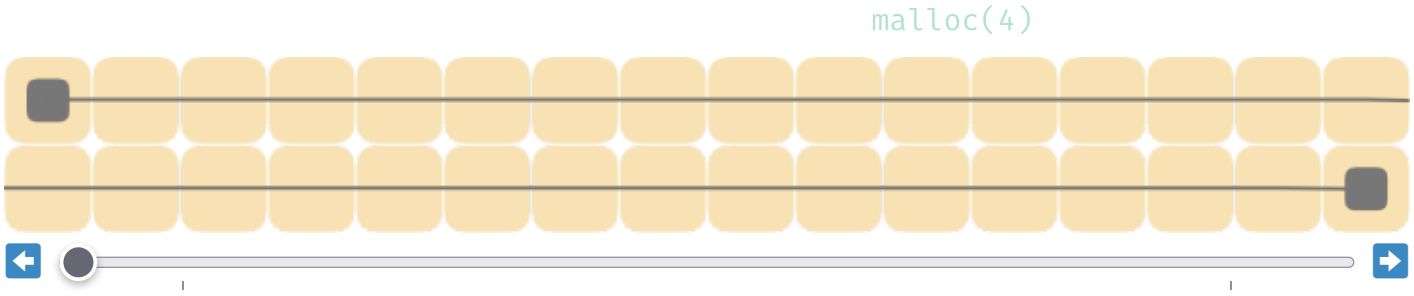
We're representing free list entries as 2 grey squares linked together with a line. You can imagine this entry being represented in code as **`address=0`** and **`size=32`**. When our program starts, all of memory is marked as free. When **`malloc`** is called, we loop through our free list until we find a block large enough to accommodate it. When we find one, we save the address and size of the allocation in our allocation list, and shrink the free list entry accordingly.

*Where do we save allocations and free list entries? Aren't we pretending our computer only has 32 bytes of memory?*

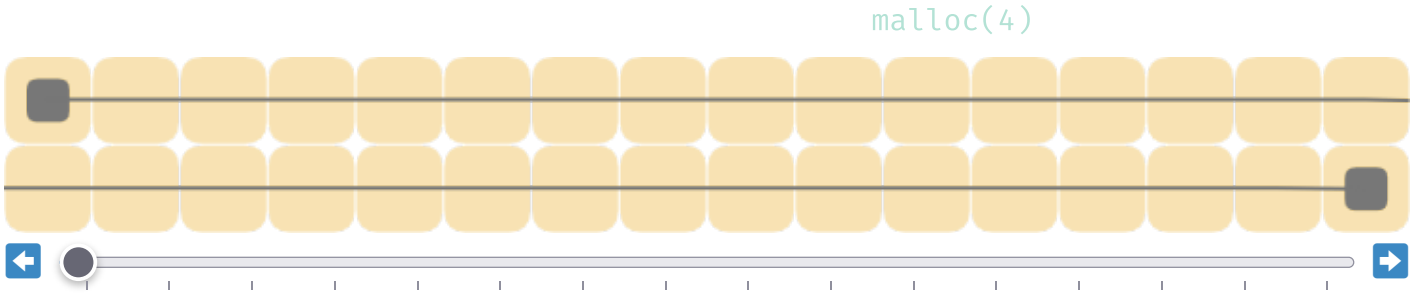
You caught me. One of the benefits of being a memory allocator is that you're in charge of memory. You could store your allocation/free list in a reserved area that's just for you. Or you could store it inline, in a few bytes immediately preceding each allocation. For now, assume we have reserved some unseen memory for ourselves and we're using it to store our allocation and free lists.

So what about **`free`**? Because we've saved the address and size of the allocation in our allocation list, we can search that list and move the allocation back in to the free list.

Without the size information, we wouldn't be able to do this.

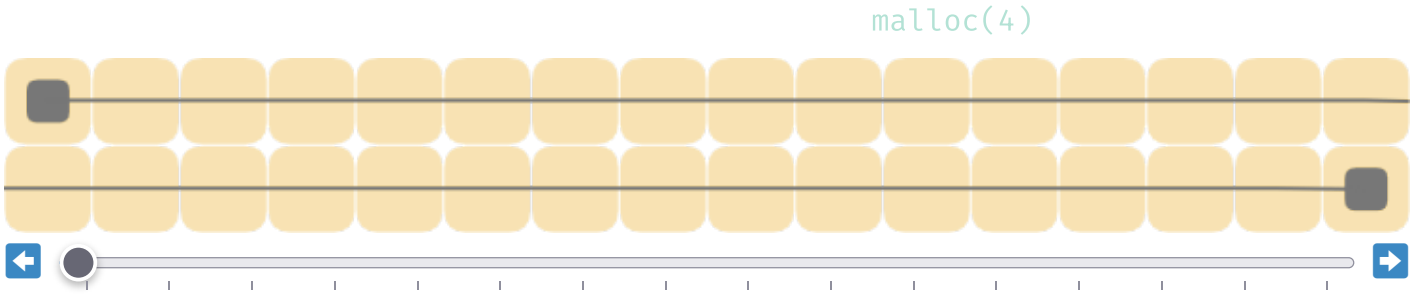



Our free list now has 2 entries. This might look harmless, but actually represents a significant problem. Let's see that problem in action.




We allocated 8 blocks of memory, each 4 bytes in size. Then we **`free`**d them all, resulting in 8 free list entries. The problem we have now is that if we tried to do a **`malloc(8)`**, there are no items in our free list that can hold 8 bytes and the **`malloc(8)`** will fail.

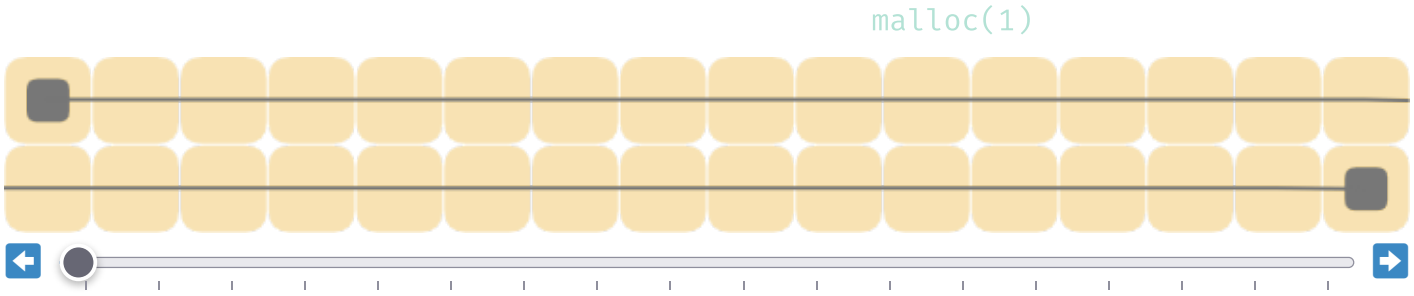
To solve this, we need to do a bit more work. When we **`free`** memory, we should make sure that if the block we return to the free list is next to any other free blocks, we combine them together. This is called "coalescing."




Much better.

## › Fragmentation

A perfectly coalesced free list doesn't solve all of our problems. The following example shows a longer sequence of allocations. Have a look at the state memory is in at the end.



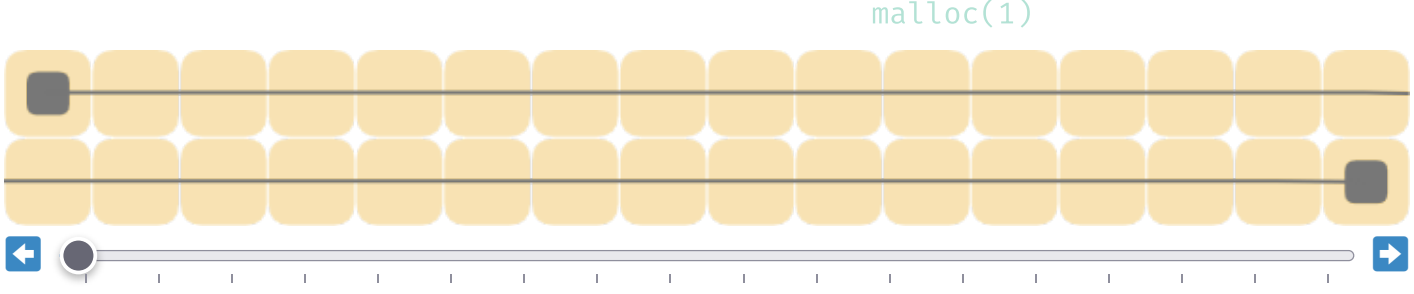

We end this sequence with 6 of our 32 bytes free, but they're split into 2 blocks of 3 bytes. If we had to service a **`malloc(6)`**, while we have enough free memory in theory, we wouldn't be able to. This is called "fragmentation."

*Couldn't we rearrange the memory to get a block of 6 contiguous bytes? Some sort of defragmentation process?*

Sadly not. Remember earlier we talked about how the return value of **`malloc`** is the address of a byte in memory? Moving allocations won't change the pointers we have already returned from **`malloc`**. We would change the value those pointers are pointed at, effectively breaking them. This is one of the downsides of the **`malloc`**/**`free`** API.

If we can't move allocations after creating them, we need to be more careful about where we put them to begin with.

One way to combat fragmentation is, confusingly, to overallocate. If we always allocate a minimum of 4 bytes, even when the request is for 1 byte, watch what happens. This is the exact same sequence of allocations as above.



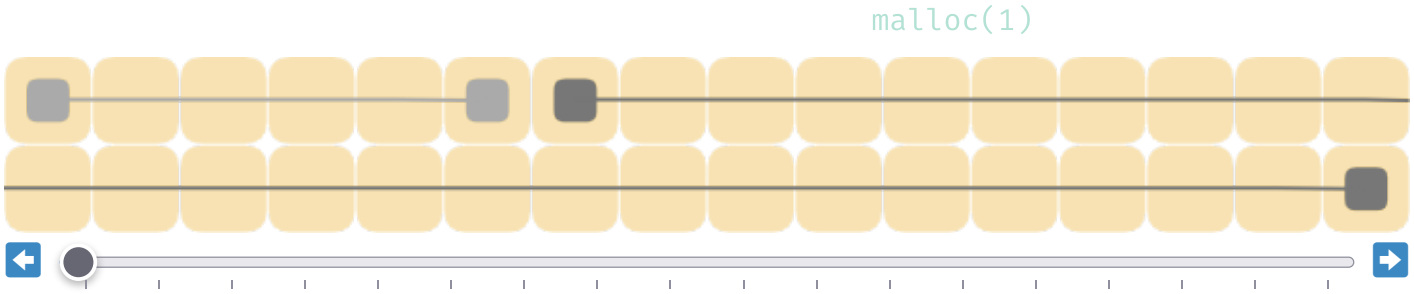
Now we can service a **`malloc(6)`**. It's worth keeping in mind that this is just one example. Programs will call **`malloc`** and **`free`** in very different patterns depending on what they do, which makes it challenging to design an allocator that always performs well.

*After the first **`malloc`**, the start of the free list seems to fall out of sync with allocated memory. Is that a bug in the visualisation?*

No, that's a side-effect of overallocating. The visualisation shows "true" memory use, whereas the free list is updated from the allocator's perspective. So when the first **`malloc`** happens, 1 byte of memory is allocated but the free list entry is moved forward 4 bytes. We trade some wasted space in return for less fragmentation.

It's worth noting that this unused space that results from overallocation is another form of fragmentation. It's memory that cannot be used until the allocation that created it is freed. As a result, we wouldn't want to go too wild with overallocation. If our program only ever allocated 1 byte at a time, for example, we'd be wasting 75% of all memory.

Another way to combat fragmentation is to segment memory into a space for small allocations and a space for big ones. In this next visualisation we start with two free lists. The lighter grey one is for allocations 3 bytes or smaller, and the darker grey one is for allocations 4 bytes or larger. Again, this is the exact same sequence of allocations as before.




Nice! This also reduces fragmentation. If we're strictly only allowing allocations of 3 bytes or less in the first segment, though, then we can't service that **`malloc(6)`**. The trade-off here is that reserving a segment of memory for smaller allocations gives you less memory to work with for bigger ones.

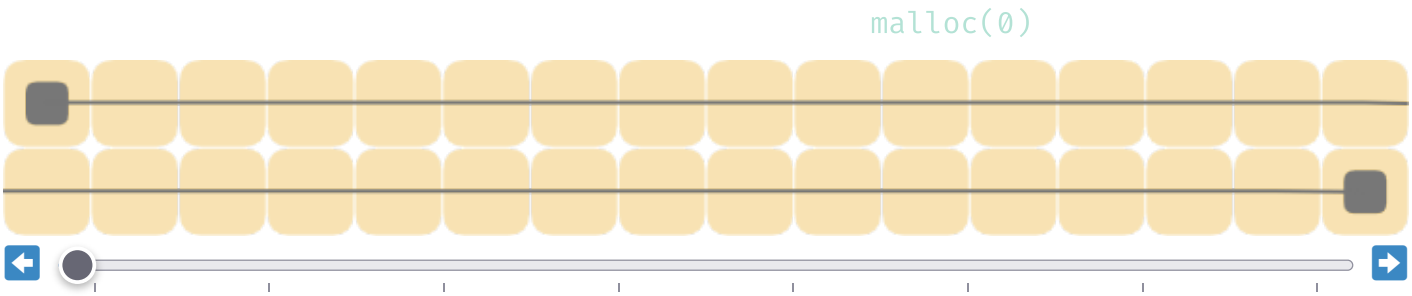
*Hey, the first allocation in the dark grey free list is 3 bytes! You said this was for allocations 4 bytes and up. What gives?*

Got me again. This implementation I've written will put small allocations in the dark grey space when the light grey space is full. It will overallocate when it does this, otherwise we'd end up with avoidable fragmentation in the dark grey space thanks to small allocations.

Allocators that split memory up based on the size of allocation are called "slab allocators." In practice they have many more size classes than the 2 in our example.

## › A quick `malloc` puzzle

What happens if you **`malloc(0)`**? Have a think about this before playing with the slider below.




This is using our free list implementation that mandates a minimum size of 4 bytes for allocations. All memory gets allocated, but none is actually used. Do you think this is correct behaviour?

It turns out that what happens when you **`malloc(0)`** differs between implementations. Some of them behave as above, allocating space they probably didn't have to. Others will return what's called a "null pointer", a special pointer that will crash your program if you try to read or write the memory it points to. Others pick one specific location in memory and return that same location for all calls to **`malloc(0)`**, regardless how many times it is called.

Moral of the story? Don't **`malloc(0)`**.

## › Inline bookkeeping

Remember earlier on when you asked about where allocation list and free list information gets stored, and I gave an unsatisfying answer about how it's stored in some other area of memory we've reserved for ourselves?

*Yes...*

This isn't the only way to do it. Lots of allocators store information right next to the blocks of memory they relate to. Have a look at this.


What we have here is memory with no allocations, but free list information stored inline in that memory. Each block of memory, free or used, gets 3 additional bytes of bookkeeping information. If **`address`** is the address of the first byte of the allocation, here's the layout of a block:

1. `address + 0` is the **size** of the block
2. `address + 1` is whether the block is **free (1)** or **used (2)**
3. `address + 2` is where the **usable memory** starts
4. `address + 2 + size` -- the **size** of the block again

So in this above example, the byte at **`0×0`** is storing the value 29. This means it's a block containing 29 bytes of memory. The value 1 at **`0×1`** indicates that the block is free memory.

*We store the **size** twice? Isn't that wasteful?*

It seems wasteful at first, but it is necessary if we want to do any form of coalescing. Let's take a look at an example.


Here we've allocated 4 bytes of memory. To do this, our **`malloc`** implementation starts at the beginning of memory and checks to see if the block there is used. It knows that at **`address + 1`** it will find either a 1 or a 2. If it finds a 1, it can check the value at **`address`** for how big the block is. If it is big enough, it can allocate into it. If it's not big enough, it knows it can add the value it finds in **`address`** to **`address`** to get to the start of the next block of memory.

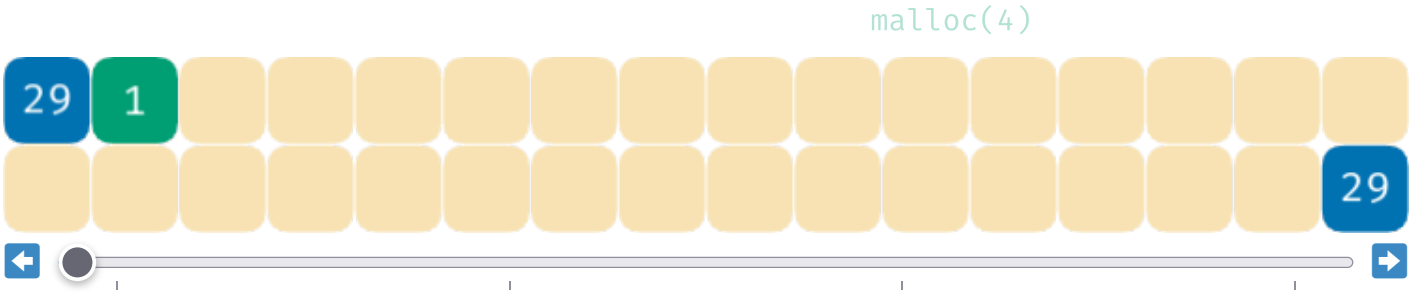
This has resulted in the creation of a used block (notice the 2 stored in the 2nd byte), and it has pushed start of the free block forward by 7 bytes. Let's do the same again and allocate another 4 bytes.


Next, let's **`free`** our first **`malloc(4)`**. The implementation of **`free`** is where storing information inline starts to shine. In our previous allocators, we had to search the allocation list to know the size of the block being **`free`**d. Now we know we'll find it at **`address`**. What's better than that is that for this **`free`**, we don't even need to know how big the allocation is. We can just set **`address + 1`** to 1!

How great is that? Simple, fast.

What if we wanted to free the 2nd block of used memory? We know that we want to coalesce to avoid fragmentation, but how do we do that? This is where the seemingly wasteful bookkeeping comes into play.

When we coalesce, we check to see the state of the blocks immediately before and immediately after the block we're **`free`**ing. We know that we can get to the next block by adding the value at **`address`** to **`address`**, but how do we get to the previous block? We take the value at **`address - 1`** and *subtract* that from **`address`**. Without this duplicated size information at the end of the block, it would be impossible to find the previous block and impossible to coalesce properly.




Allocators that store bookkeeping information like this alongside allocations are called "boundary tag allocators."

*What's stopping a program from modifying the bookkeeping information? Wouldn't that completely break memory?*

Surprisingly, nothing truly prevents this. We rely heavily, as an industry, on the correctness of code. You might have heard of "buffer overrun" or "use after free" bugs before. These are when a program modifies memory past the end of an allocated block, or accidentally uses a block of memory after **`free`**ing it. These are indeed catastrophic. They can result in your program immediately crashing, they can result in your program crashing in several minutes, hours, or days time. They can even result in hackers using the bug to gain access to systems they shouldn't have access to.

We're seeing a rise in popularity of "memory safe" languages, for example Rust. These

languages invest a lot in making sure it's not possible to make these types of mistake in the first place. Exactly how they do that is outside of the scope of this article, but if this interests you I highly recommend giving Rust a try.

You might have also realised that calling `free` on a pointer that's in the middle of a block of memory could also have disastrous consequences. Depending on what values are in memory, the allocator could be tricked into thinking it's `free`ing something but what it's really doing is modifying memory it shouldn't be.

To get around this, some allocators inject "magic" values as part of the bookkeeping information. They store, say, `0x55` at `address + 2`. This would waste an extra byte of memory per allocation, but would allow them to know when a mistake has been made. To reduce the impact of this, allocators often disable this behaviour by default and allow you to enable it only when you're debugging.

# Playground

If you're keen to take your new found knowledge and try your hand at writing your own allocators, you can click here to go to my allocator playground. You'll be able to write JavaScript code that implements the `malloc`/`free` API and visualise how it works!

# Conclusion

We've covered a lot in this post, and if it has left you yearning for more you won't be disappointed. I've specifically avoided the topics of virtual memory, `brk` vs `mmap`, the role of CPU caches, and the endless tricks real `malloc` implementations pull out of their sleeves. There's no shortage of information about memory allocators on the Internet, and if you've read this far you should be well-placed to dive in to it.

Join the discussion on Hacker News!

## Acknowledgments

Special thanks to the following people:

- Chris Down for lending me his extensive knowledge of real-world memory

allocators.

- Anton Verinov for lending me his extensive knowledge of the web, browser developer tools, and user experience.
- Blake Becker, Matt Kaspar, Krista Horn, Jason Peddle, and Josh W. Comeau for their insight and constructive reviews.

Home