

# What's in a ring buffer? And using them in Rust

Monday, February 10, 2025

Working on my cursed MIDI project, I needed a way to store the most recent messages without risking an unbounded amount of memory usage. I turned to the trusty ring buffer for this! I started by writing a very simple one myself in Rust, then I looked and what do you know, of *course* the standard library has an implementation.

While I was working on this project, I was pairing with a friend. She asked me about ring buffers, and that's where this post came from!

## What's a ring buffer?

Ring buffers are known by a few names. Circular queues and circular buffers are the other two I hear the most. But this name just has a nice ring to it.

It's an array, basically: a buffer, or a queue, or a list of items stacked up against each other. You can put things into it and take things out of it the same as any buffer. But the front of it connects to the back, like any good ring<sup>1</sup>.

Instead of adding on the end and popping from the end it, like a stack, you can add to one end and remove from the start, like a queue. And as you add or remove things, where the start and end of the list *move around*. Your data eats its own tail.

This lets us keep a fixed number of elements in the buffer without running into reallocation. In a regular old buffer, if you use it as a queue—add to the end, remove from the front—then you'll eventually need to either reallocate the entire thing or shift all the elements over. Instead, a ring buffer lets you just keep adding from either end and removing from either end and you never have to reallocate!

## Uses for a ring buffer

My MIDI program is one example of where you'd want a ring buffer, to keep the most recent items. There are some general situations where you'll run into this:

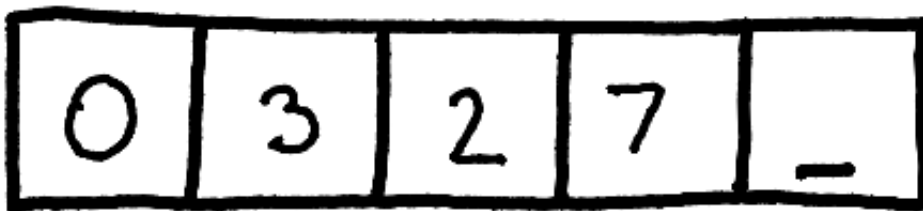
- You want a fixed number of the most recent things, like the last 50 items seen
- You want a queue, especially with a fixed maximum number of queued elements<sup>2</sup>
- You want a buffer for data coming in with an upper bound, like with streaming audio, and you want to overwrite old data if the consumer can't keep up for a bit

A lot of it comes down to performance, and streaming data. Something is producing data, something else is consuming it, and you want to make sure insertions and removals are fast. That was exactly my use: a MIDI device produces messages, my UI consumes them, but I don't want to fill up *all* my memory, forever, with more of them.

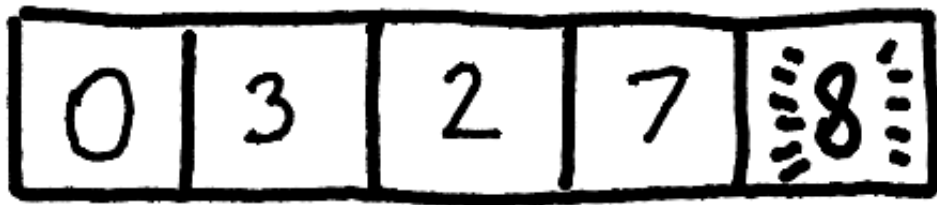
## How ring buffers work

So how do they work?

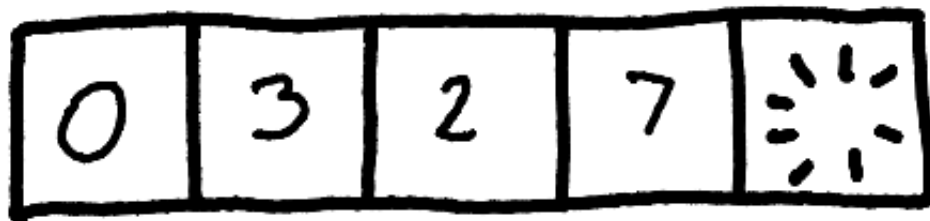
This is a normal, non-circular buffer:



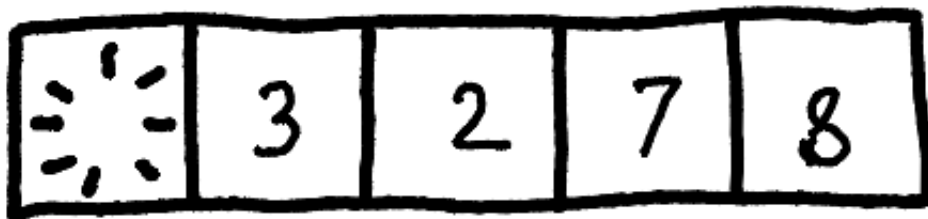
When you add something, you put it on the end.



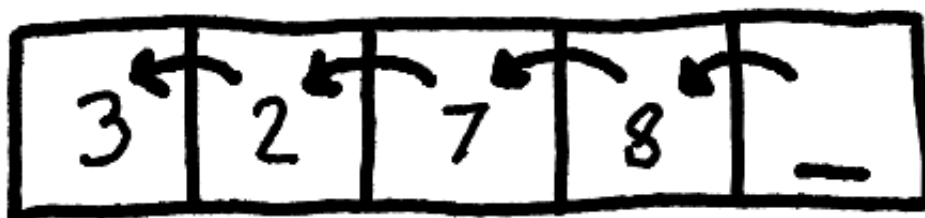
If you're using it as a stack, then you can pop things off the end.



But if you're using it as a queue, you pop things off the front...



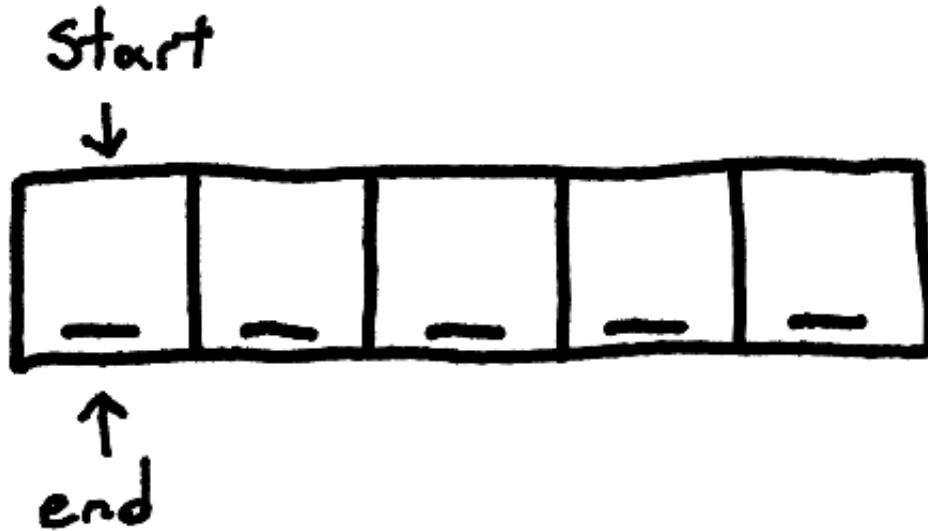
And then you have to shuffle everything to the left.



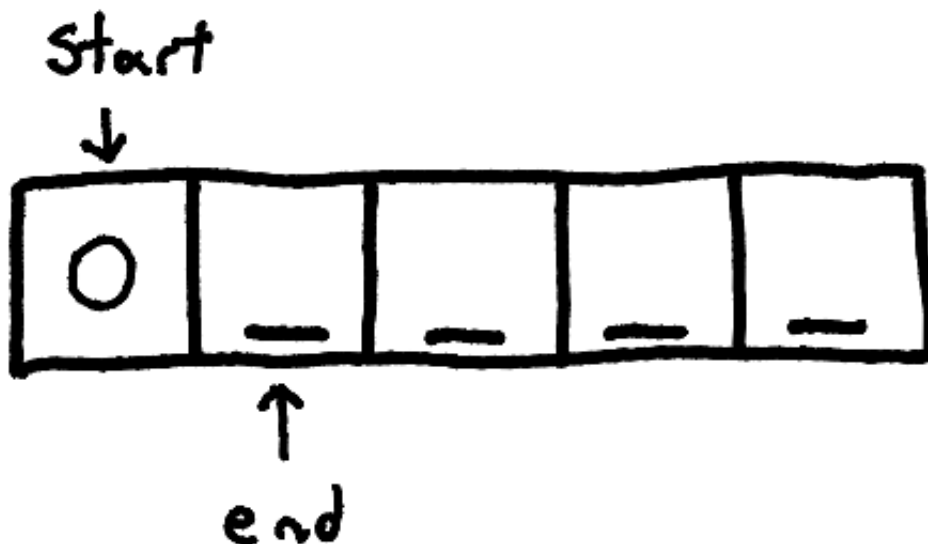
That's why we have ring buffers! I'll draw it as a straight line here, to show how it's laid out in memory. But the end loops back to the front logically, and we'll see how it

wraps around.

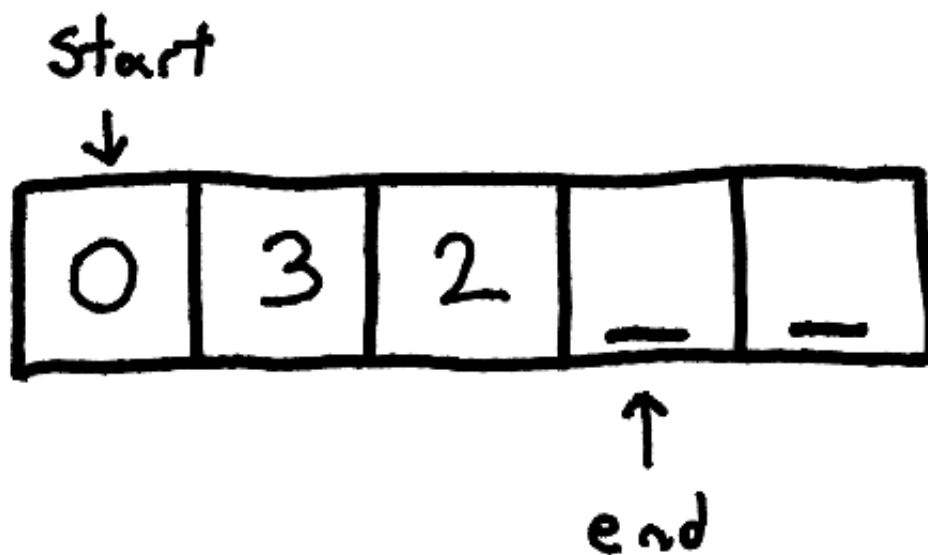
We start with an empty buffer, and `start` and `end` both point at the first element. When `start == end`, we know the buffer is empty.



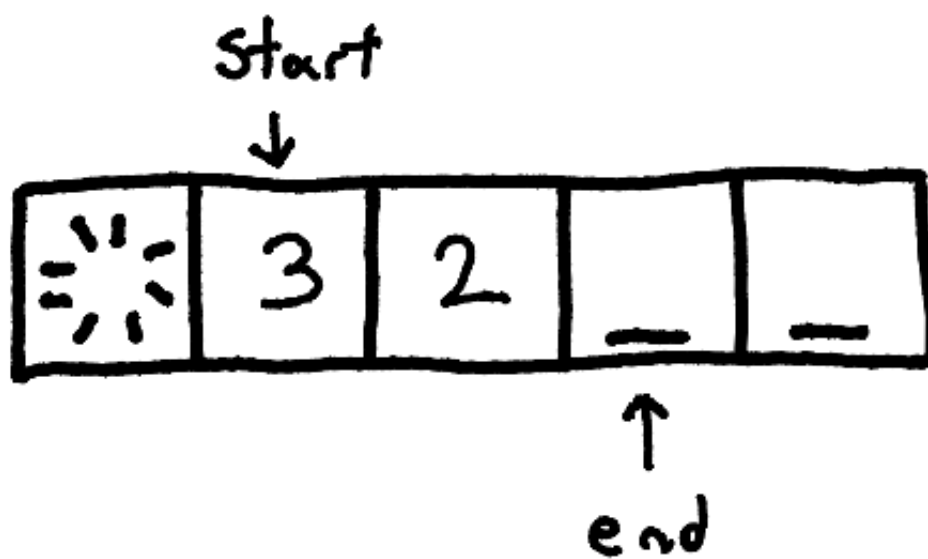
If we insert an element, we move `end` forward.



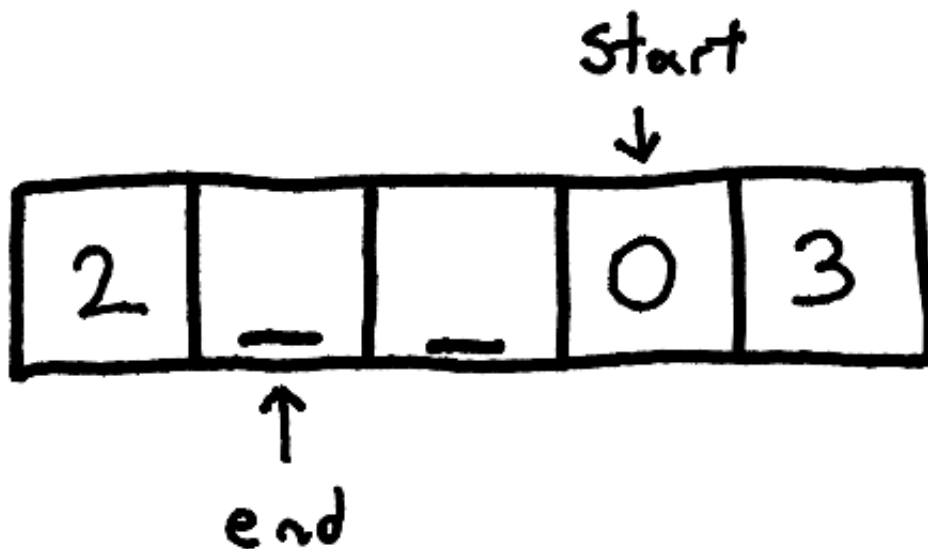
And it gets repeated as we insert multiple times.



If we remove an element, we move `start` forward.



We can also start the buffer at any point, and it crosses over the end gracefully.



Ring buffers are pretty simple when you get into their details, with just a few things to wrap your head around. It's an incredibly useful data structure!

## Rust options

If you want to use one in Rust, what are your options?

There's the standard library, which includes [VecDeque](#). This implements a ring buffer, and the name comes from "Vec" (Rust's growable array type) combined with "Deque" (a double-ended queue). As this is in the standard library, this is quite accessible from most code, and it's a *growable* ring buffer. This means that the pop/push operations will always be efficient, but if your buffer is full it will result in a resize operation. The amortized running time will still be  $O(1)$  for insertion, but you incur the cost all at once when a resize happens, rather than at each insertion.

You can enforce size limits in your code, if you want to avoid resize operations. Here's an example of how you could do that. You check if it's full first, and if so, you remove something to make space for the new element.



```
let buffer: VecDeque<u32> = VecDeque::with_capacity(10);

for thing in 0..15 {
    // if the buffer is already full, remove the first element to make
    space
    if buffer.len() == 10 {
        buffer.pop_front();
    }
}
```

```
buffer.push_back(thing);  
}
```

There are also libraries that can enforce this for you! For example, [circular-buffer](#) implements a ring buffer. It has a fixed max capacity and won't resize on you, instead overwriting elements when you run out of space. The size is set at compile time, though, which can be great or can be a constraint that's hard to meet. There is also [ringbuffer](#), which gives you a fixed size buffer that's heap allocated at runtime. That buys you some flexibility, with the drawback of being heap-based instead of stack-based.

I'm definitely reaching for a library when I need a non-growable ring buffer in Rust now. There are some good choices, and it saves me the trouble of having to manually enforce size limits.

- 
1. One of my favorite rings represents the sun and the moon, with the sun nestling inside this crescent moon shape. Unfortunately, the ring is *open* there. It makes for a very nice visual effect, but it kept getting snagged on things and bending. So it's not a very good ring for living an active hands-on life. 
  2. These can also be resizable! The Rust standard library one is. This comes with reallocation cost, which amortizes to a low cost but can be expensive in individual moments. 
-