Inside STL: The deque, design

devblogs.microsoft.com/oldnewthing/20230809-00

August 9, 2023

Raymond Chen

The C++ standard library `deque` is a double-ended queue that supports adding and removing items efficiently at either the front or the back.

All three of the major implementations of the C++ standard library use the same basic structure for a deque, but they vary in both policy and implementation details.

First, let's design a simple version of a deque that stores its elements in an array.

```
template<typename T>
struct simple_deque
{
    T* elements;
    T* first;
    T* last;
    size_t capacity;
};
```

For example, a deque of three integers might look like this:

					↓ first			↓ last	
elements	→	?	?	?	1	2	3	?	?
capacity = 8									
size = last − first = 3									

The `elements` points to an array whose length is given by the `capacity` member's value of 8. In that array, the first three elements are not in use, but which could be used in the future. We'll call them *spares*. Next come three elements holding the values 1, 2, and 3, followed by two more spares. The first element in use (1) is pointed to by `first`, and one past the last element in use is pointed to by `last`.

With this design, the four basic deque operations are straightforward:

- Remove from front: Increment `first`.
- Remove from back: Decrement `last`.
- Add to front: Decrement `first` and store the new value there.
- Add to back: Store the new value at `last`, and then increment `last`.

When you run out of spares, you have a few choices.

- If there is a spare at the opposite end, you can move the elements over, so that they consume one or more of the spares at the opposite end, and free up spares on the end you are trying to expand.
- If there are no spares anywhere, then you need to allocate a new, bigger array and then move the existing elements out of the old array into the new one, leaving space to be used as new spares.

Now, this is an inefficient data structure because both of the alternatives for freeing up spares are *O*(*n*), which is a problem when the dequeue gets large.

To solve this, the implementations don't use a giant array. Instead, the giant array is chopped up into fixed-sized blocks. That way, expanding the array entails just allocating a new block.

			↓ first			↓ last	
?	?	?	1	2	3	?	?
block		block		block		block	

This is the same diagram we had earlier, except that instead of eight contiguous elements, we have four blocks, each of which holds two elements.

The four basic deque operations are still the same. It's just that incrementing and decrementing the `first` and `last` pointers is more complicated because they have to know to jump to the next block when incrementing past the end of the current block, or jump to the previous block when decrementing past the beginning of the current block.

If you need to expand the array, you can allocate a new block and add it to the beginning or end. No elements need to be moved.

Next time, we'll dig into the implementations. That's where things get messy.