Inside STL: The deque, implementation

devblogs.microsoft.com/oldnewthing/20230810-00

August 10, 2023

Raymond Chen

Now that we understand the design behind the common STL dequeue implementations, we can peek into the implementation details.

All three of the major implementations of the standard library maintain an array of pointers to blocks, which they confusingly call a "map" even though it is unrelated to `std::map`. (Even more confusingly, gcc internally uses the term "node" instead of "block".) Initially, all the pointers in the map are `nullptr`, and the blocks are allocated only on demand.

We will say that a block is *spare* if it contains only spare elements.

	gcc	**clang**	**msvc**
Block size	as many as fit in 512 bytes but at least 1 element	as many as fit in 4096 bytes but at least 16 elements	power of 2 that fits in 16 bytes but at least 1 element
Initial map size	8	2	8
Map growth	2×	2×	2×
Map shrinkage	On request	On request	On request
Initial first/last	Center	Start	Start

Members	`block** map;`
	`size_t map_size;`
	`iterator first;`
	`iterator last;`
		`block** map;`
		`block** first_block;`
		`block** last_block;`
		`block** end_block;`
		`size_t first;`
		`size_t size;`
			`block** map;`
			`size_t map_size;`
			`size_t first;`
			`size_t size;`

Map layout	counted array	`simple_deque`	counted array

Valid range	Pair of iterators	Start and count	Start and count
Iterator	`T* current;` `T* current_block_begin;` `T* current_block_end;` `block** current_block;`	T* current; block** current_block;	`deque* parent;` `size_t index;`
`begin()` / `end()`	Copy `first` and `last`.	Break `first` and `first + size` into block index and offset.	Break `first` and `first + size` into block index and offset.
Spare blocks	Aggressively pruned	Keep one on each end	Keep all

The Microsoft Visual C++ implementation has a tiny block size. I suspect this decision was made a long time ago, and the Visual C++ folks are anxious to bump up the block size at the next ABI break.

The gcc implementation doesn't believe in spare blocks; it frees blocks as soon as they become empty. The clang and Visual C++ implementations hold onto spare blocks in anticipation that they will be needed again soon. The clang implementation retains one spare block at each end, whereas the Visual C++ implementation retains all spare blocks. (I suspect this is another old design decision that the Visual C++ team wants to change at the next ABI break.)

If the clang implementation needs to allocate a new block at one end, it first checks to see if there is a spare block at the other end. If so, then it moves that block from one end to the other instead of allocating a new block.

The Microsoft Visual C++ implementation treats the map as a *circular* array of blocks. This means that once the map is filled with blocks, no further allocations or element movement is needed to satisfy any further deque operations, assuming the size of the dequeue never exceeds the capacity.

None of the three implementations shrinks the map automatically. You have to call `shrink_to_fit()` to get the map to shrink.

All three implementations keep track of the blocks differently. The gcc and Visual C++ implementations use a simple counted array. If the gcc implementation runs out of map entries at one end, it slides the in-use blocks so that they are centered in the block map. The Visual C++ implementation uses a circular buffer, so the only time it runs out of map entries is when the entire map needs to be expanded. The clang implementation gets fancy and uses the equivalent of our `simple_deque` for its map. This allows it to use the “slide” trick to shift an empty slot from one end to the other.

All three implementations choose different formats for their iterators. The gcc and clang implementations use a pointer to the current element and a pointer to the current block. When incrementing or decrementing off the end of the block, they increment or decrement the `current_block` to find the next block. The gcc implementation carries extra values `current_block_begin` and `current_block_end`, which are technically redundant, since you can derive

- `current_block_begin = *current_block;`
- `current_block_end = current_block_begin + block_size;`

Not only are these iterators bulky (presumably for performance?), but the gcc implementation stores these bulky iterators in its `deque`, whereas clang and Visual C++ use an index and length. Accessing an element by index involves dividing by the block size. This means that clang’s `begin()` and `end()` incur a runtime division to calculate which block the element is in and using the remainder to get the offset within the block. Visual C++ forces block sizes to be powers of two, so it can use bitwise operations to break the index into a block and offset.

Here’s a visualization of what the three `deque` implementations look like if you alternately pop from the front and push to the back:

First, here’s what gcc’s implementation does.

map[0]	map[1]	map[2]	map[3]
	A	B	
	1	2 3	
pop 1 from the front: Free block A			
		B	
		2 3	
push 4 to the back: Allocate block C			
		B	C
		2 3	4
pop 2 from the front			

		B	C
		3	4
push 5 to the back			
		B	C
		3	4 5
pop 3 from the front: Free block B			
		B	C
			4 5
push 6 to the back: Allocate block D and slide			
	C	D	
	4 5	6	

The eager pruning of empty blocks in gcc means that we free a block, only to immediately allocate a new one.

Here's what clang does when you alternate pops and pushes:

map[0]	map[1]	map[2]	map[3]
	A	B	
	1	2 3	
pop 1 from the front: Block A becomes spare			
	A	B	
		2 3	
push 4 to the back: Steal block A			
		B	A
		2 3	4
pop 2 from the front			
		B	A
		3	4

push 5 to the back			
		B	A
		3	4 5
pop 3 from the front: Block B becomes spare			
		B	A
			4 5
push 6 to the back: Steal block B and slide			
	A	B	
	4 5	6	

Since clang keeps a one-block spare, the growth on the back can be satisfied by stealing the spare block from the front. No new memory allocations are performed; just block sliding.

And here’s what Visual C++ does. Note that Visual C++ always pushes starting at the start of the map, so getting to the initial state means that we have already popped three elements from the front, leaving a spare block A.

map[0]	map[1]	map[2]	map[3]
A	B	C	
	1	2 3	
pop 1 from the front: Block B becomes spare			
A	B	C	
		2 3	
push 4 to the back: Allocate block D			
A	B	C	D
		2 3	4
pop 2 from the front			
A	B	C	D
		3	4

push 5 to the back							
A		B		C		D	
					3	4	5
pop 3 from the front: Block C becomes spare							
A		B		C		D	
						4	5
push 6 to the back: Wrap around to block A							
A		B		C		D	
6						4	5

Once you reach the steady state where all the map entries have blocks, further deque operations can be performed without any memory allocation or rearranging.

Here’s what the Microsoft Visual C++ deque looks like in the debugger with the visualizer:

```
d : { size = 0x2 }
    [0x0] = 1 [Type: __int64]
    [0x1] = 2 [Type: __int64]
```

The raw view reveals the soft underbelly of the implementation.

```
0:000> ?? d
class std::deque<__int64,std::allocator<__int64> >
   +0x000 _Mypair          :
std::_Compressed_pair<std::allocator<__int64>,std::_Deque_val<std::_Deque_simple_types
 >,1>
0:000> ?? d._Mypair
class
std::_Compressed_pair<std::allocator<__int64>,std::_Deque_val<std::_Deque_simple_types
 >,1>
   +0x000 _Myval2          : std::_Deque_val<std::_Deque_simple_types<__int64> >
0:000> ?? d._Mypair._Myval2
class std::_Deque_val<std::_Deque_simple_types<__int64> >
   +0x000 _Myproxy         : 0x000001c9`ee615540 std::_Container_proxy
   +0x008 _Map             : 0x000001c9`ee6178b0  -> 0x000001c9`ee615000  -> 0n42
   +0x010 _Mapsize         : 8
   +0x018 _Myoff           : 3
   +0x020 _Mysize          : 2
0:000> dps 0x000001c9`ee6178b0 L 8
000001c9`ee6178b0  000001c9`ee615000
000001c9`ee6178b8  000001c9`ee615160
000001c9`ee6178c0  000001c9`ee615380
000001c9`ee6178c8  000001c9`ee6153c0
000001c9`ee6178d0  000001c9`ee615720
000001c9`ee6178d8  000001c9`ee6155e0
000001c9`ee6178e0  000001c9`ee615240
000001c9`ee6178e8  000001c9`ee615020
0:000> dps 000001c9`ee615160 L2
000001c9`ee615160  00000000`0000002a
000001c9`ee615168  00000000`00000001
0:000> dps 000001c9`ee615380 L2
000001c9`ee615380  00000000`00000002
000001c9`ee615388  00000000`00000099
```

Since this is a dequeue of `__int64` (8 bytes), there will be two values per 16-byte block. From the raw dump, the `_Map` tells us where the block pointers are, and the `_Mapsize` tells us how many. We dump the block pointers, and the `_Myoff` tells us that the first valid entry is at offset 3, and the `_Mysize` tells us that there are two valid entries. Therefore, the first valid entry is at offset 1 in block 1, and the second valid entry is at offset 0 in block 2.

We dump the block at index 1 and see that the value at offset 1 is `00000000`00000001`. And then we dump the block at index 2 and see that the value at offset 0 is `00000000`00000002`. We also see that the other (unused) elements in the block hold values that had previously been pushed and then popped from the deque.

Bonus chatter: As I noted at the start of the series, the primary purpose of these articles is to explain how to extract the contents of these collection classes when you encounter them in a crash dump. I'm taking the designs as given and showing how to use them to find the data.