Queues

Learning Objectives

- Use the STL queue class to implement queue-based algorithms such as scheduling first-come, first-served tasks
- Use the STL double-ended queue classes in applications
- Implement the queue and double-ended queue classes using either an array or a linked-list data structure

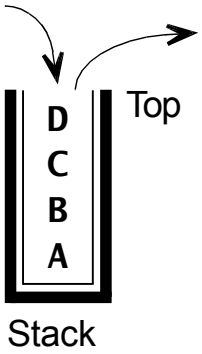
Introduction

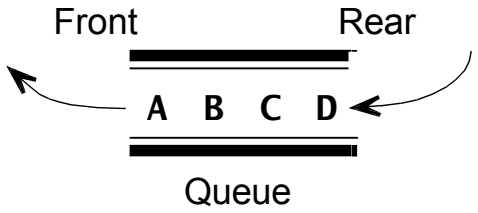
- First-In/First-Out data structure (FIFO)
  - Because entries must be removed in exactly the same order that they were added to the queue

Stack vs. Queue

Input: ABCD

- With a queue: they are removed in the same order: A, B, C, D
- With a stack: they are removed in the reverse order: D, C, B, A





The Standard Library Queue Class

- The C++ Standard Library has a queue template class

```
template <class T, class Container = deque<T> >
class queue;
```

- Queues are implemented as containers adaptors
- This underlying container must support at least the following operations:
  - empty; size; front; push_back; pop_front
- The standard container classes deque and list fulfill these requirements

IMPLEMENTATIONS OF THE QUEUE CLASS

Array Implementation of a Queue

- With a queue, we add entries at one end of the array and remove them from the other end
- We access the used portion of the array at both ends

  Note: Using stack, we access just one end of the partially filled array
- Because we now need to keep track of both ends of the used portion of the array, we will have two variables to keep track of how much of the array is used:
  1. `first`: indicates the first index currently in use
  2. `last`: indicates the last index currently in use

```
data[first], data[first + 1], ... data[last]
```

Array Implementation of a Queue (cont.)

- Add an entry:

1. increment last by one
2. store the new entry in `data[last]`

- Get the next entry:

1. retrieve `data[first]`
2. increment first by one, so that `data[first]` is the entry that used to be second

- One problem with this plan: `last` is incremented but never decremented
  - it will quickly reach the end of the array
- In a normal application, `first` would also be incremented when entries are removed from the queue
  - this will free up the array locations with index values less than `first`

Array Implementation of a Queue (cont.)

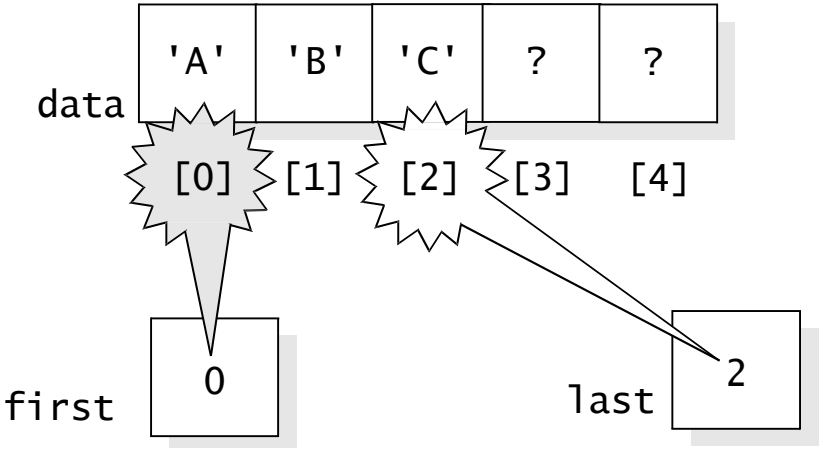
- There are several ways to reuse these freed locations

1. A straightforward approach for using the freed array locations
   - Maintain all the queue entries so that first is always equal to 0
   - When `data[0]` is removed, we move all the entries in the array down one location
   - This approach is inefficient: every time we remove an entry from the queue, we must move every entry in the queue

2. When the last index reaches the end of the array, simply start reusing the available locations at the front of the array

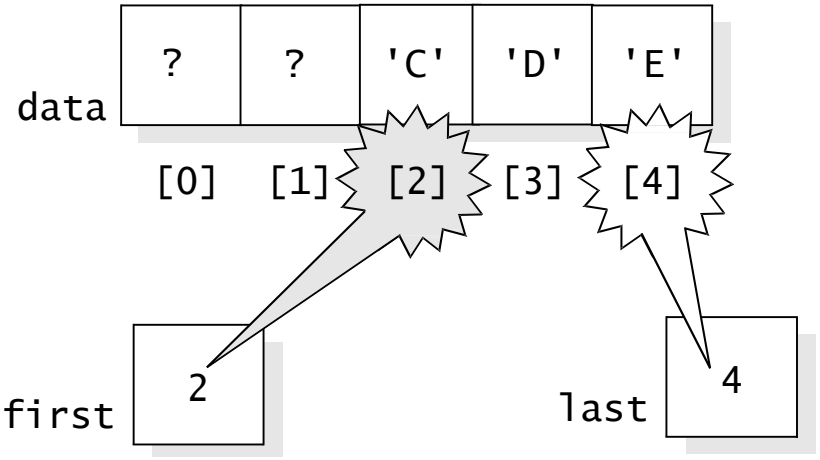
Array Implementation of a Queue (cont.)

- Suppose that we have a queue of characters with a capacity of five, and the queue currently contains three entries, 'A', 'B', and 'C’, these values are stored with first equal to 0 and last equal to 2



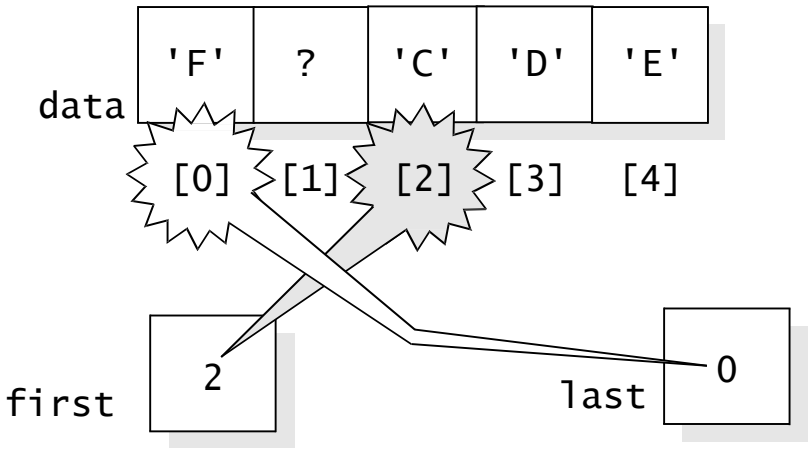
Array Implementation of a Queue (cont.)

- Let's remove two entries (the 'A' and 'B'), and add two more entries to the rear of this queue



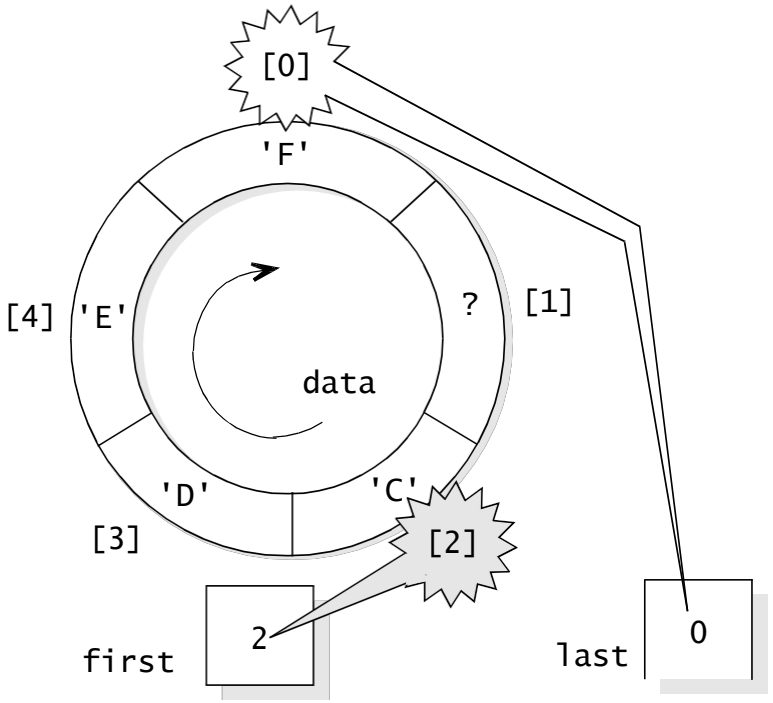
Array Implementation of a Queue (cont.)

- Suppose that we add another character, 'F', to the queue
- Go to the front of the array, adding the new 'F' at location `data[0]`



Array Implementation of a Queue (cont.)

- With circular view of the array, the queue's entries start at `data[first]` and continue forward
- If you reach the end of the array, you come back to `data[0]` and keep going until you find the rear



- It is called a circular array

Circular Array Implementation of a Queue

- Private member variables
  - `data`: holds the queue's entries in an array
  - `first` and `last`: hold the indexes for the front and the rear of the queue
    - ✓ Whenever the queue is non-empty, the entries begin at `data[first]`
    - ✓ If the entries reach the end of the array, then they continue at the first location, `data[0]`
    - ✓ In any case, `data[last]` is the last entry in the queue
  - `count`: records the number of items that are in the queue
    - ✓ count will be used to check whether the queue is empty or full, and also to produce the value returned by the size member function

```
template <class Item>
class queue
{
 public:
    typedef std::size_t size_type;
    typedef Item value_type;
    static const size_type CAPACITY = 30;

    // CONSTRUCTOR
    queue( );
    // MODIFICATION MEMBER FUNCTIONS
    void pop( );
    void push(const Item& entry);
    // CONSTANT MEMBER FUNCTIONS
    bool empty( ) const { return (count == 0); }
    Item front( ) const;
    size_type size( ) const { return count; }
 private:
    Item data[CAPACITY]; // Circular array
    size_type first;     // Index of item at front of the queue
    size_type last;      // Index of item at rear of the queue
    size_type count;     // Total number of items in the queue
    // HELPER MEMBER FUNCTION
    size_type next_index(size_type i) const {return (i+1)%CAPACITY;}
 };
```

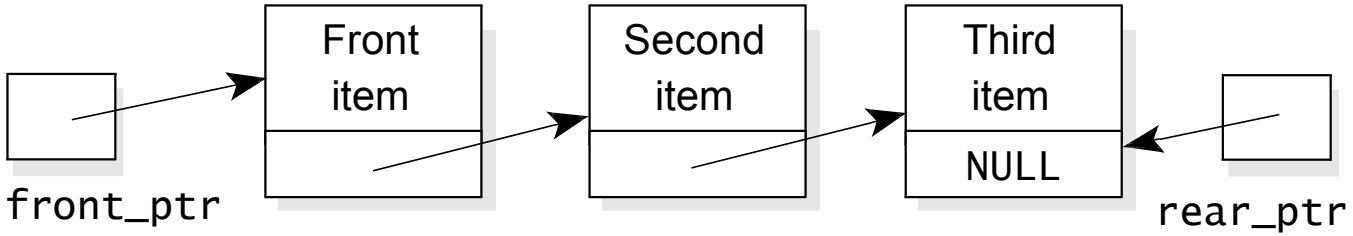
```
template <class Item>
queue<Item>::queue( )
{
    count = 0;
    first = 0;
    last = CAPACITY - 1;
}

template <class Item>
Item queue<Item>::front( ) const
// Library facilities used: cassert
{
    assert(!empty( ));
    return data[first];
}

template <class Item>
void queue<Item>::pop( )
// Library facilities used: cassert
{
    assert(!empty( ));
    first = next_index(first);
    --count;
}
```

Linked-List Implementation of a Queue

- A queue can also be implemented as a linked list
- One end of the linked list is the front, and the other end is the rear of the queue
- The approach uses two pointers:
- front_ptr: To the first node
- rear_ptr: To the last node

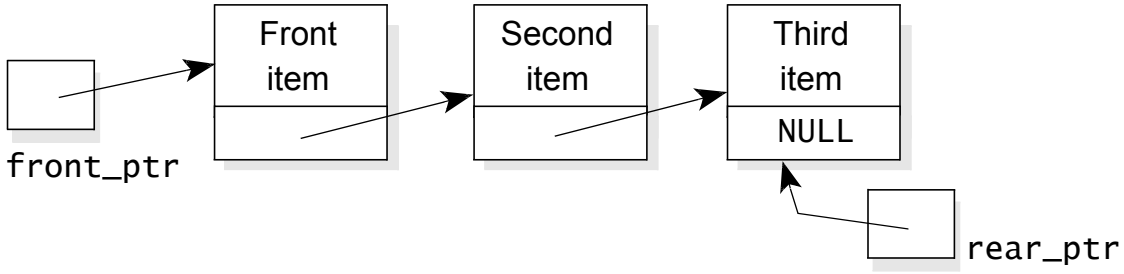


Invariant of the Queue Class (Linked-List Version)

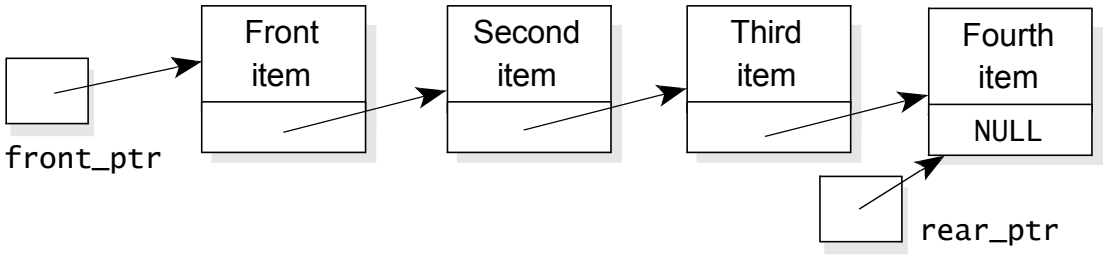
1. The number of items in the queue is stored in the member variable `count`
2. The items in the queue are stored in a linked list, with the front of the queue stored at the head node, and the rear of the queue stored at the final node
3. Pointers:
   - `front_ptr` is the head pointer
   - `rear_ptr` is the tail pointer

Implementation Details

- Push member function. Adds a node at the rear of the queue



- After adding a fourth item, the list would look like this:



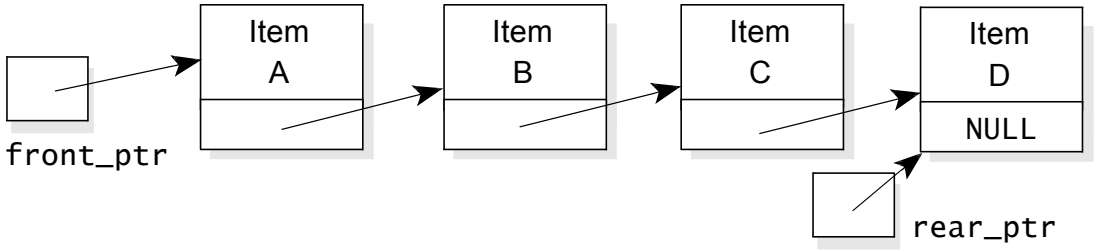
- The item is added at the end of the list through:

```
list_insert(rear_ptr, entry);
rear_ptr = rear_ptr->link( );
```

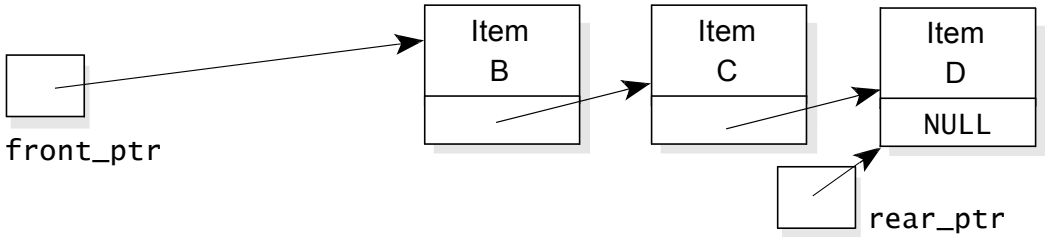
- To add the first item, we need a slightly different approach because the empty list has no rear pointer

```
if (empty( ))
{ // Insert first entry.
    list_head_insert(front_ptr, entry);
    rear_ptr = front_ptr;
}
else
{ // Insert an entry that is not the first.
    list_insert(rear_ptr, entry);
    rear_ptr = rear_ptr->link( );
}
```

- Pop member function. Removes a node from the front of the queue



- The pop function will remove the item that is labeled “Item A”



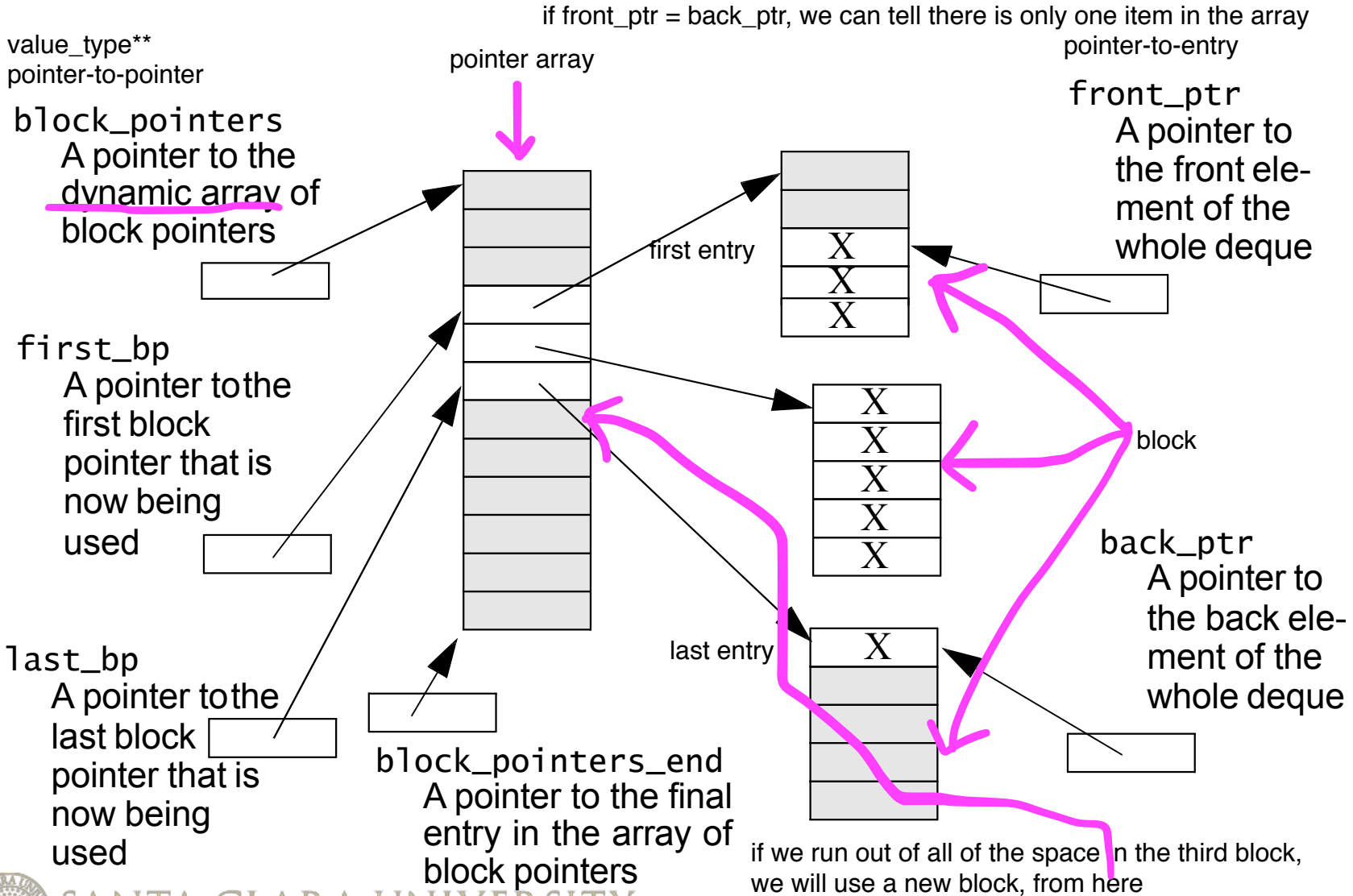
- The implementation of pop uses list_head_remove

IMPLEMENTING THE STL DEQUE CLASS

Implementing the STL deque Class

- A variation of queue is a double-ended queue, also called a deque
- Entries can be quickly inserted and removed at both ends
- This differs from:
  - A stack (which uses only one end)
  - An ordinary queue (in which items enter at one end and leave at the other)
- The most straightforward implementations of a deque are similar to the queue implementations using a circular array or linked list
- STL uses a more complicated deque implementation
  - A dynamic array of a fixed size called block
  - If more space is needed, the block is not resized
  - Instead, a second block of the same fixed size is allocated
  - Pointers to all blocks are kept together in a separate array of pointers

A Possible Data Structure for the STL deque Class



A Possible Data Structure for the STL deque Class (Cont.)

- This deque currently uses three blocks
- Each block is a dynamic array containing a constant number of items (five in our example, although a real implementation would have a larger block size)
- The shaded array locations at the start of the first block and the end of the last block are not currently being used

- The dynamic array on the left is an array of pointers to the blocks
- Currently capable of holding up to 12 pointers, although only 3 are being used now
- If more than 12 blocks are needed, then that array of pointers can be expanded

A Possible Data Structure for the STL deque Class (Cont.)

- The complexity (efficiency) of common deque operations:
  - Random access: constant O(1)
  - Insertion or removal of elements at the end or beginning: constant O(1)
  - Insertion or removal of elements: linear O(n)

A Possible Data Structure for the STL deque Class (Cont.)

```
// Number of value_type items in each block
static const size_t block_size = 5;
// The elements in the deque
typedef int value_type;
// A pointer to a dynamic array of value_type items
typedef value_type* vtp;
// A pointer to the dynamic array of block pointers
vtp* block_pointers;
// A pointer to the final entry in the array of block pointers
vtp* block_pointers_end;
// A pointer to the first block pointer that's now being used
vtp* first_bp;
// A pointer to the last block pointer that's now being used
vtp* last_bp;
// A pointer to the front element of the whole depue
vtp front_ptr;
// A pointer to the back element of the whole depue
vtp back_ptr;
```

pointer to pointer

pointer to item

SANTA CLARA UNIVERSITY
THE JESUIT UNIVERSITY IN SILICON VALLEY

The deque's pop_back Function

```
void mydeque::pop_back( )
{
   // An empty deque has a NULL back_ptr
   assert (back_ptr != NULL);

   if (back_ptr == front_ptr)
   {  // Case 1: There is just one block with just one element,
      // so delete it, and  reset things back to the start
      clear( );
   }
   else if (back_ptr == *last_bp)
   {  // Case 2: back_ptr is pointing to the first element of
      // the last block in the deque
      // 1. Remove the entire last block. Note that this will call
      // the destructor for each item in the block
      // 2. The last block pointer that we are using is now
      // one spot earlier in the array  of block pointers
      // 3. The new back element is the last element in the last block
      delete [] back_ptr;
      --last_bp;
      back_ptr = (*last_bp) + (BLOCK_SIZE - 1);
   }
```

The deque's pop_back Function

```
    else
    {  // Case 3: The element we are removing is not the only element
       // in its block, so we just move the back_ptr backward
       --back_ptr;
    }
}
```

Priority Queues

- A priority queue is a container class that allows entries to be retrieved according to some specified priority levels.
  - The highest priority entry is removed first
  - Entries with equal priority can be removed according some criterion e.g. FIFO as an queue.
- STL priority_queue<Item> template class
  - #include <queue>
  - priority_queue<int> q2;
  - Functions push, pop, empty, size , **top** *(not **front**!)*

Reference Return Values for the stack, queue, and priority queue classes

- In STL, the top (for stack) and front (for queue) functions have reference return values, e.g. in stack class definition:
  - Item& top ( );
  - const Item& top( ) const;
- Can be used to change the top item
  - If we declare
    - ✓ stack<int> b;
    - ✓ const stack<int> c;
  - Which ones are correct? =>

1. int i = b.top(); V
2. b.push(i); V
3. b.top() = 18; V
4. c.top() = 18; X
5. b.push(c.top()); V

Summary

- A queue is a First-In/First-Out data structure
- A queue can be implemented as a partially filled circular array
- A queue can be implemented as a linked list
- When implementing a queue, you need to keep track of both ends of the list
- A deque (or "double-ended queue") allows quick removal and insertion of elements from both ends
- It can be implemented with a circular array, a doubly linked list or more complex structures of pointers