COEN 79
Object-Oriented Programming and Advanced Data Structures
Department of Computer Engineering, Santa Clara University
Behnam Dezfouli, PhD

Queues

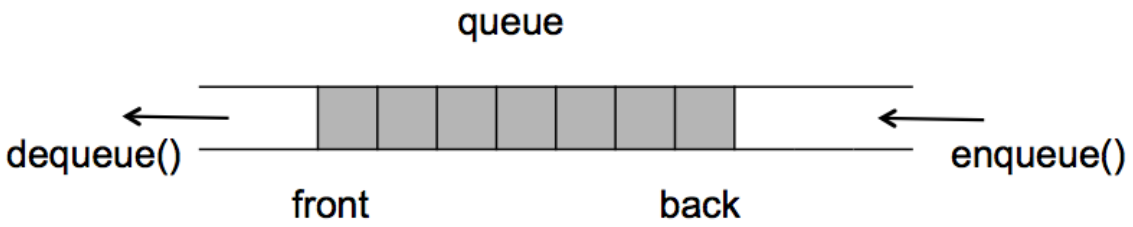
Slide Set: 8

Learning Objectives

- Implement the queue and double-ended queue classes using either an array or a linked-list data structure
- The STL deque implementation

Introduction

- A queue is **a data structure of ordered entries** such that entries can only be inserted at one end (called the **rear**) and removed at the other end (called the **front**)
- The entry at the front end of the queue is called the first entry
- Designed to operate in a FIFO context (first-in first-out)
- Items are taken out of the queue in the same order that they were put into the queue



Implementations of the Queue Class

Implementations of the Queue Class

- We will give two implementations of our queue class:
  - A static implementation using a fixed-sized array
  - A dynamic implementation using a linked list

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

- 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

- If `data` is the array name, then the queue entries will be in the array components:

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

Implementations of the Queue Class

Array Implementation of a Queue (Cont'd)

- **To add an entry:**
  1. increment `last` by one
  2. store the new entry in the component `data[last]`
- **To get the next entry:**
  1. retrieve `data[first]`
  2. increment `first` by one, so that `data[first]` is then the entry that used to be second

What is the problem with this plan?

Implementations of the Queue Class

Array Implementation of a Queue (Cont'd)

- 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

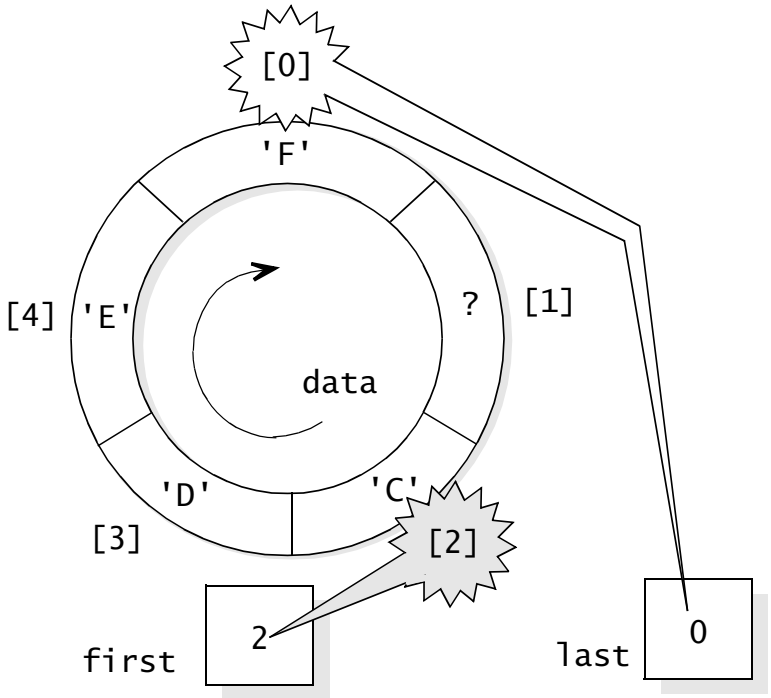
- **Another approach: We can simply start reusing the available locations at the front of the array**
- Think of this arrangement as if the array were bent into a circle with the first component of the array immediately after the last component
- In this **circular arrangement**, the free index positions are always "right after" `data[last]`

Implementations of the Queue Class

Array Implementation of a Queue (Cont'd)

- **Circular view of the array**
- It may help to actually view the array as bent into a circle, with the final array element attached back to the beginning:

- An array used in this way is called a **circular array**



Implementations of the Queue Class

Array Implementation of a Queue (Cont'd)

Implementation of queue using a circular array:

- `data`: A private member variable that holds the queue's entries in an array

- The private member variables `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 the location `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`: A private member variable that records the number of items that are in the queue
  - **`count` will be used to check whether the queue is empty or full**

Implementations of the Queue Class

Array Implementation of a Queue (Cont'd)

- The queue class definition uses a **helper function** called `next_index`
  - This is a private member function
  - Allows us to easily step through the array, one index after another, with wraparound at the end
  - The reason for declaring `next_index` is to make the other implementations easier to read

- **`next_index = (i+1) % CAPACITY`**

Implementations of the Queue Class

Header File for the Array Version of the Queue Template Class (Cont'd)

```
#ifndef SCU_COEN79_QUEUE1_H
#define SCU_COEN79_QUEUE1_H

#include <cstdlib>   // Provides size_t

namespace scu_coen79_8B
{

    template <class Item>
    class queue
    {

     public:
        typedef std::size_t size_type;
        typedef Item value_type;
        static const size_type CAPACITY = 30;

       // CONSTRUCTOR
       queue( );
```

Implementations of the Queue Class

Header File for the Array Version of the Queue Template Class (Cont'd)

```
        // 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;}
    };
}
#include "queue1.template"
#endif
```

When a class requires some small operation that is implemented as a formula, consider implementing the formula with a **helper function**

Implementations of the Queue Class

Implementation of the Array Version of the Queue Template Class

```
// TEMPLATE CLASS IMPLEMENTED: queue<Item>
// INVARIANT for the queue class:
// 1. The number of items in the queue is in the member variable count;
// 2. For a non-empty queue, the items are stored in a circular array
// beginning at data[first] and continuing through data[last].
// The array's total capacity is CAPACITY.
// 3. For an empty array, last is some valid index, and first is
// always equal to next_index(last).

#include <cassert>
namespace scu_coen79_8B {
    template <class Item>
    const typename queue<Item>::size_type queue<Item>::CAPACITY;

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

Satisfies the invariant: For an empty queue, `last` is some valid index, and `first = next_index(last)`

Implementations of the Queue Class

Implementation of the Array Version of the Queue Template Class (Cont'd)

```
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;
}
```

Implementations of the Queue Class

Implementation of the Array Version of the Queue Template Class (Cont'd)

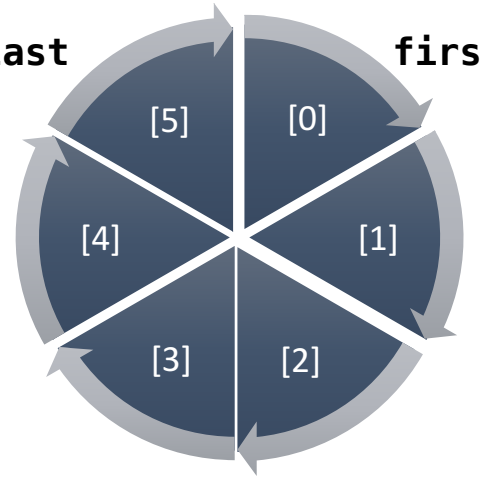
```
template <class Item>
void queue<Item>::push(const Item& entry)
// Library facilities used: cassert
{
    assert(size( ) < CAPACITY);
    last = next_index(last);
    data[last] = entry;
    ++count;
}


|| implementation of other functions…

}
```

Implementations of the Queue Class

Implementation of the Array Version of the Queue Template Class (Cont'd)

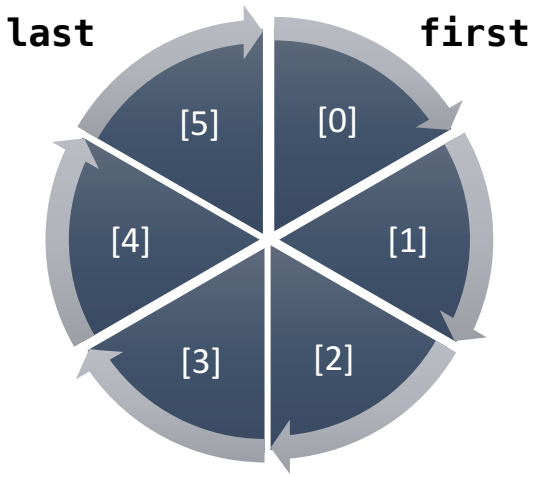



```
template <class Item>
queue<Item>::queue( )
{
   count = 0;
   first = 0;
   last = CAPACITY - 1;
}
```

Creating an empty queue

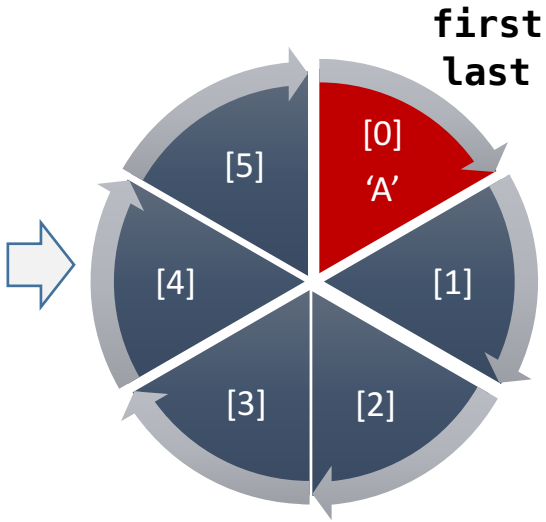
Implementations of the Queue Class

Implementation of the Array Version of the Queue Template Class (Cont'd)



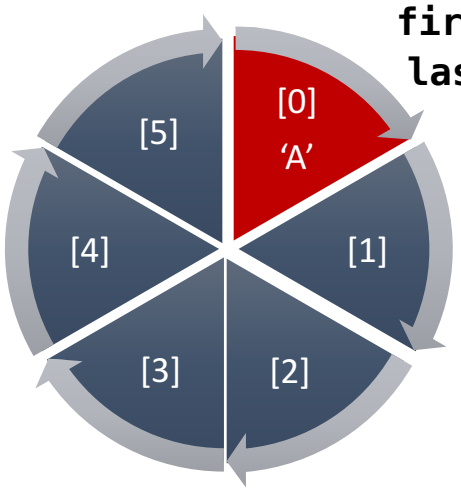
Adding an entry to the queue

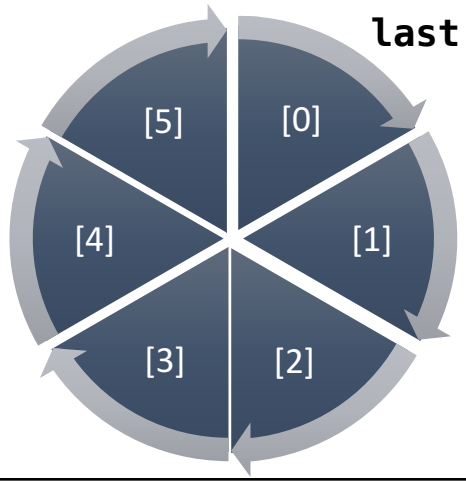
```
template <class Item>
void queue<Item>::push(const Item& entry)
{
    assert(size( ) < CAPACITY);
    last = next_index(last);
    data[last] = entry;
    ++count;
}
```



Implementations of the Queue Class

Implementation of the Array Version of the Queue Template Class (Cont'd)



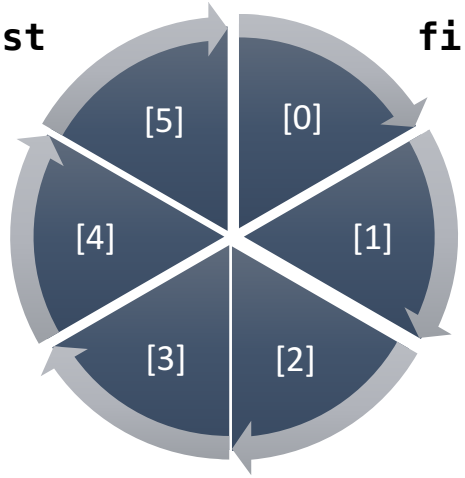



```
template <class Item>
void queue<Item>::pop( )
  {
      assert(!empty( ));
      first = next_index(first);
      --count;
  }
```

Removing an entry from the queue

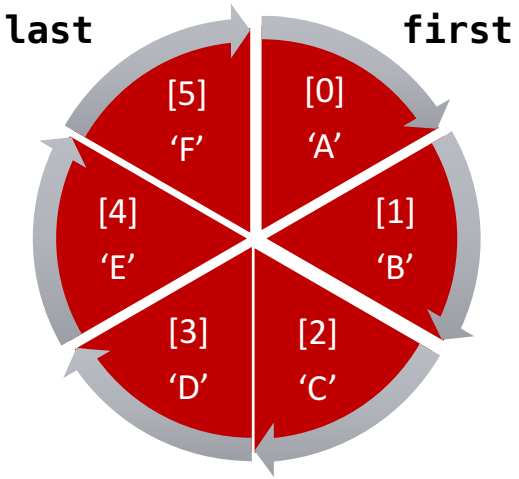
Implementations of the Queue Class

Implementation of the Array Version of the Queue Template Class (Cont'd)



- **An empty queue**
- `first = next_index (last)`

`count = 0`



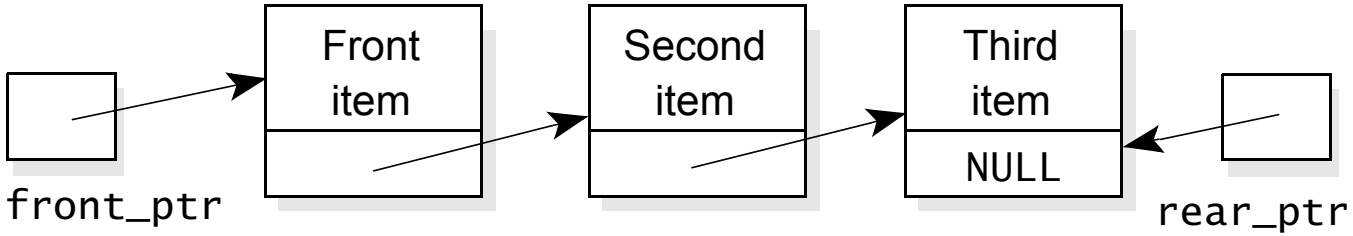
- **A full queue**
- `first = next_index (last)`

`count = 4`

Implementations of the Queue Class

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



Implementations of the Queue Class

Header File for the Linked-List Version of the Queue Template Class

```
// FILE: queue2.h (part of the namespace
// TEMPLATE CLASS PROVIDED: queue<Item> (a queue of items)

#ifndef SCU_COEN79_QUEUE2_H    // Prevent duplicate definition
#define SCU_COEN79_QUEUE2_H
#include <cstdlib>             // Provides std::size_t
#include "node2.h"             // Node template class

namespace scu_coen79_8C
{
    template <class Item>
    class queue
    {
    public:
        // TYPEDEFS
        typedef std::size_t size_type;
        typedef Item value_type;
```

Implementations of the Queue Class

Header File for the Linked-List Version of the Queue Template Class (Cont'd)

```
        // CONSTRUCTORS and DESTRUCTOR
        queue( );
        queue(const queue<Item>& source);
        ~queue( );
        // MODIFICATION MEMBER FUNCTIONS
        void pop( );
        void push(const Item& entry);
        void operator =(const queue<Item>& source);
        // CONSTANT MEMBER FUNCTIONS
        bool empty( ) const { return (count == 0); }
        Item front( ) const;
        size_type size( ) const { return count; }
    private:
        scu_coen79_6B::node<Item> *front_ptr;
        scu_coen79_6B::node<Item> *rear_ptr;
        size_type count;       // Total number of items in the queue
    };
}

#include "queue2.template" // Include the implementation
#endif
```

Implementations of the Queue Class

Invariant of the Queue Class (Linked-List Version)

Invariants:

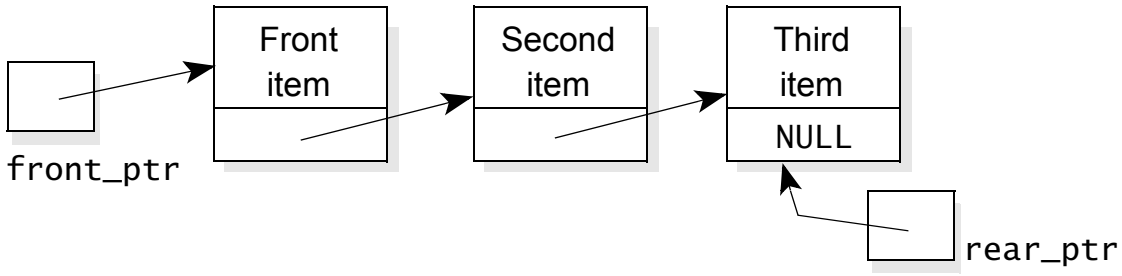
1. The number of items in the queue is stored in the member variable `count`
2. The items in the queue are stored in a linked list, with the front of the queue stored at the head node, and the rear of the queue stored at the final node
3. Pointers:
   - `front_ptr` is the head pointer
   - For a non-empty queue, the member variable `rear_ptr` is the tail pointer of the linked list; for an empty list, we don't care what's stored in `rear_ptr` (as `front_ptr` is `NULL`)

Implementations of the Queue Class

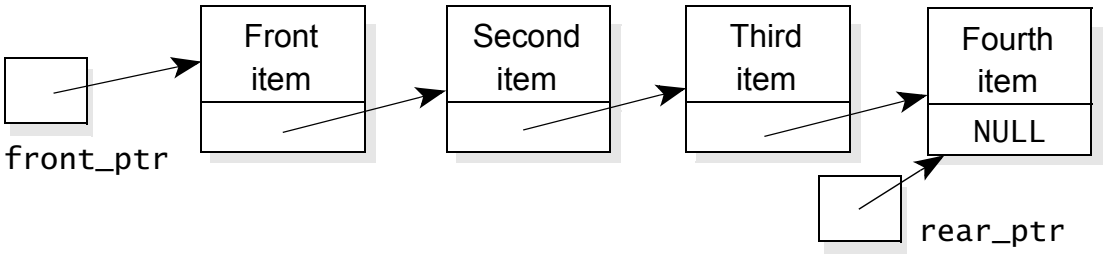
Implementation Details (Cont'd)

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

❑Example: Suppose we start with three items:



- 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( );
```

Implementations of the Queue Class

Implementation Details (Cont'd)

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

```
if (empty( ))
{ // Insert first entry.
    scu_coen79_6B::list_head_insert(front_ptr, entry);
    rear_ptr = front_ptr;
}

else
{ // Insert an entry that is not the first.
    scu_coen79_6B::list_insert(rear_ptr, entry);
    rear_ptr = rear_ptr->link( );
}
```

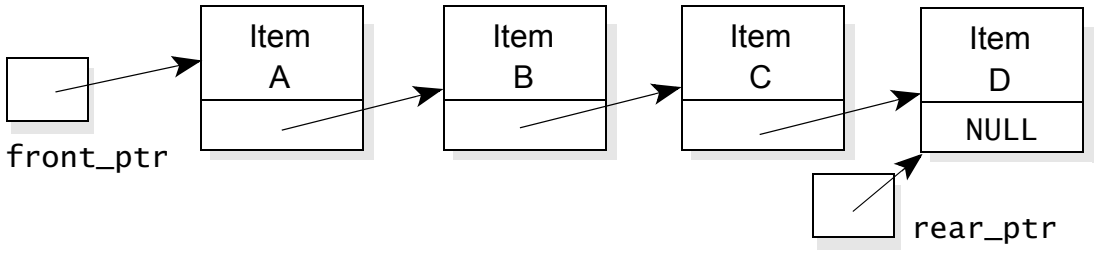
- You also need to add one to the `count` member variable

Implementations of the Queue Class

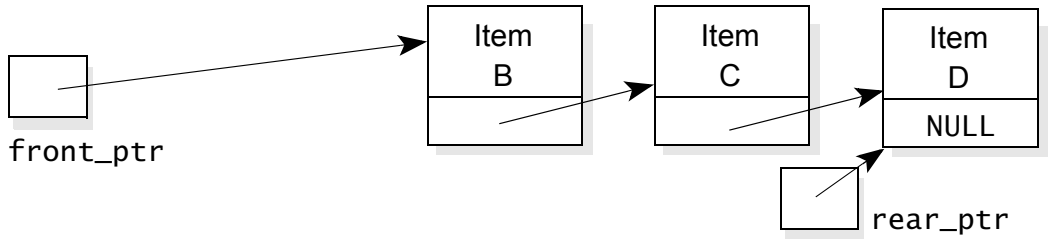
Implementation Details (Cont'd)

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

❑Example: We start with this queue:



- The `pop` function will remove the item that is labeled "`Item A`", when `pop` returns, the list will have only three items



- The implementation of pop uses `list_head_remove`

Implementations of the Queue Class

Linked-List Version Implementation of the Queue Template Class

```
// FILE: queue2.template
// TEMPLATE CLASS IMPLEMENTED: Queue<Item> (see queue2.h for
// documentation)

// INVARIANT for the Queue class:
// 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. The member variable front_ptr is the head pointer of the linked
// list of items. For a non-empty queue, the member variable rear_ptr
// is the tail pointer of the linked list; for an empty list, we don't
// care what's stored in rear_ptr.
```

Note that it is important to document this "Don't Care" situation

Implementations of the Queue Class

Linked-List Version Implementation of the Queue Template Class (Cont'd)

```
#include <cassert>   // Provides assert
#include "node2.h"   // Node template class

namespace scu_coen79_8C
{
    template <class Item>
    queue<Item>::queue( )
    {
        count = 0;
        front_ptr = NULL;
    }


    template <class Item>
    queue<Item>::queue(const queue<Item>& source)
    // Library facilities used: node2.h
    {
        count = source.count;
        list_copy(source.front_ptr, front_ptr, rear_ptr);
    }
```

For an empty list, we don't care what's stored in `rear_ptr`

Implementations of the Queue Class

Linked-List Version Implementation of the Queue Template Class (Cont'd)

```
template <class Item>
queue<Item>::~queue( )
{
    list_clear(front_ptr);
}


template <class Item>
void queue<Item>::operator =(const queue<Item>& source)
// Library facilities used: node2.h
{
    if (this == &source) // Handle self-assignment
        return;

    list_clear(front_ptr);
    list_copy(source.front_ptr, front_ptr, rear_ptr);
    count = source.count;
}
```

Implementations of the Queue Class

Linked-List Version Implementation of the Queue Template Class (Cont'd)

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

```
template <class Item>
void queue<Item>::pop( )
// Library facilities used: cassert, node2.h
{
    assert(!empty( ));
    list_head_remove(front_ptr);
    --count;
}
```

Implementations of the Queue Class

Linked-List Version Implementation of the Queue Template Class (Cont'd)

```
template <class Item>
void queue<Item>::push(const Item& entry)
// Library facilities used: node2.h
{
    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( );
    }

    ++count;
}
}
```

Implementations of the Queue Class

Implementation of the Array Version of the Queue Template Class (Cont'd)

- **What are the iterator invalidation rules for the linked-list version of the queue class?**

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)

Implementing the STL deque Class

```
template< class T, class Allocator = std::allocator<T> >
class deque;
```

- std::deque (double-ended queue) is **an indexed sequence** container that **allows fast insertion and deletion at both its beginning and its end**
- **Insertion and deletion at either end of a deque never invalidates pointers or references to the rest of the elements**
- **As opposed to std::vector, the elements of a deque are not stored contiguously**
  - Typical implementations use a sequence of individually allocated fixed-size arrays (we will show the implementation later)
- **Expansion of a deque is cheaper than the expansion of a std::vector because it does not involve copying of the existing elements to a new memory location**

Implementing the STL deque Class

```
#include <iostream>
#include <deque>
int main()
{
    // Create a deque containing integers
    std::deque<int> d = {1, 2, 3, 4};

    // Add an integer to the beginning and end of the deque
    d.push_front(0);
    d.push_back(5);

    // Iterate and print values of deque
    for(std::deque<int>::iterator it = d.begin(); it != d.end(); ++it)
    {
        std::cout << *it << '\n';
    }
}
```

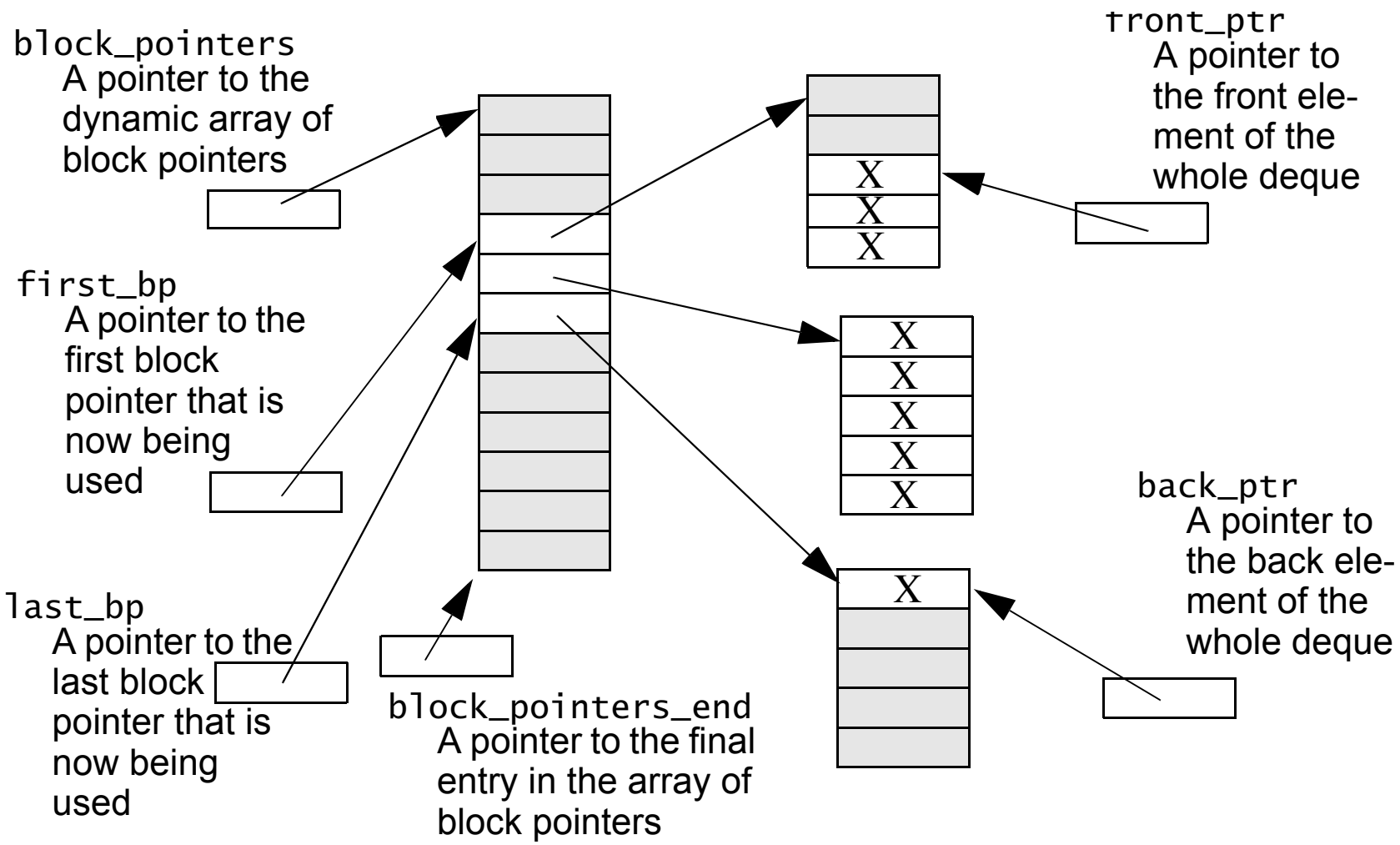
```
Output:
0
1
2
3
4
5
```

Implementing the STL deque Class

- STL uses a more complicated deque implementation
- A dynamic array of a fixed size called **block** is initially allocated to hold the elements
- **If more space is needed, the block is not resized, instead, a second block of the same fixed size is allocated**
- More and more blocks are allocated as needed, and pointers to all these blocks are kept together in a separate array of pointers

Implementing the STL deque Class

A Possible Data Structure for the STL deque Class



Implementing the STL deque Class

A Possible Data Structure for the STL deque Class

- The deque of the previous slide is using 3 **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

Implementing the STL deque Class

A Possible Data Structure for the STL deque Class

- **The complexity (efficiency) of common dequeu 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)

Implementing the STL deque Class

A Possible Data Structure for the STL deque Class

- deque provides a functionality similar to vectors, but with efficient insertion and deletion of elements also at the beginning of the sequence, and not only at its end
- **deques are not guaranteed to store all its elements in contiguous storage locations**
  - vectors use a single array that needs to be occasionally reallocated for growth
  - The elements of a deque can be scattered in different chunks of storage
- For operations that involve frequent insertion or removals of elements at positions other than the beginning or the end, deques perform worse and have less consistent iterators and references than lists and forward lists

Implementing the STL deque Class

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

- The private member pointers might be declared in this way:

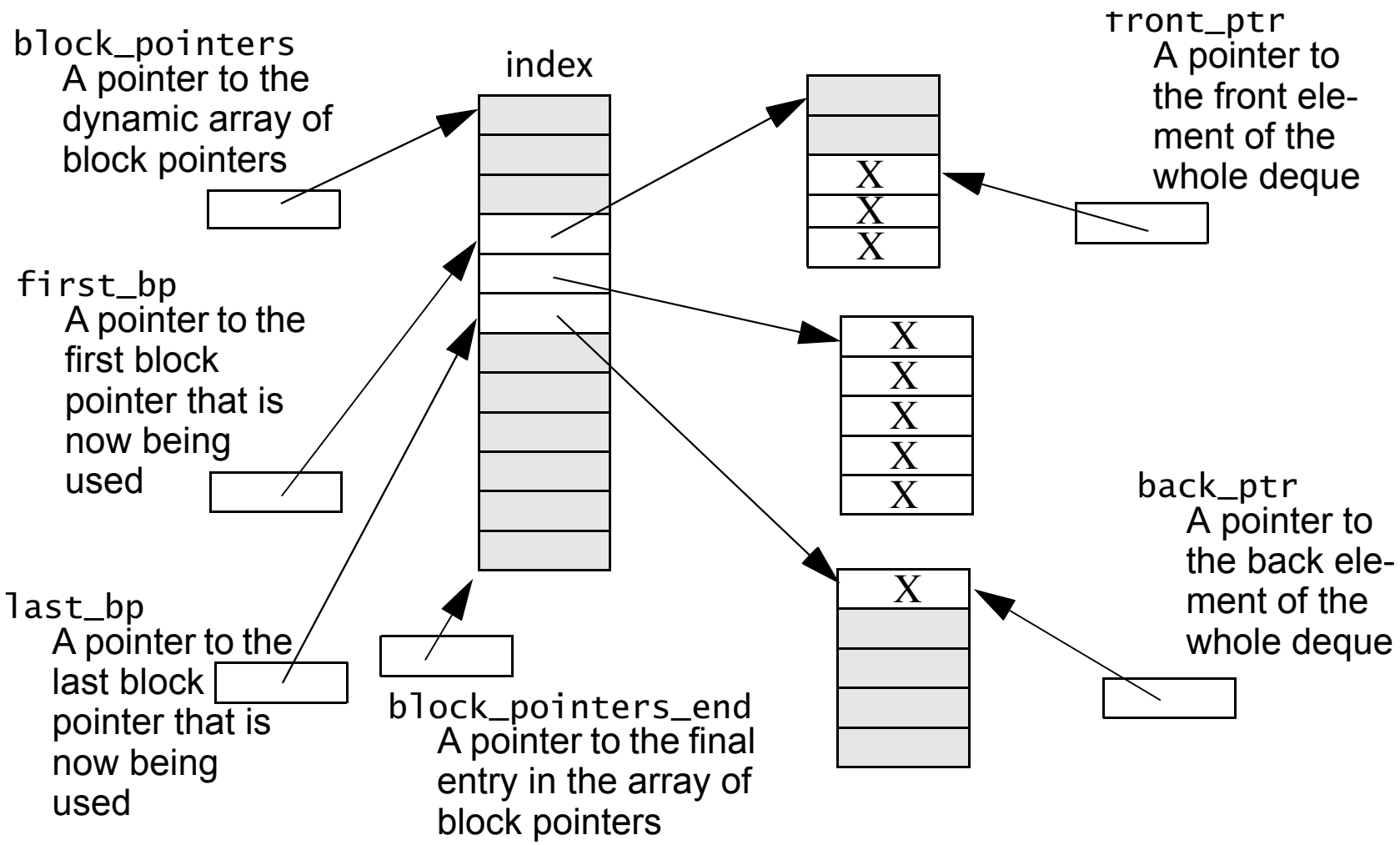
```
// Number of value_type items in each block
static const size_t block_size = 5;
// The elements in the deque
typedef int value_type;
// A pointer to a dynamic array of value_type items
typedef value_type* vtp;

// 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;
vtp front_ptr; // A pointer to the front element of the whole deque
vtp back_ptr;  // A pointer to the back element of the whole deque
```

Implementing the STL deque Class

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



Implementing the STL deque Class

Pointer Arithmetic used in the Implementation of the deque Class

Use pointer arithmetic in the implementation:

- If `p` is a pointer into an array and `n` is an integer, then the expression `p+n` gives a new pointer that is `n` elements beyond `*p`
- In this case, the expression `++p` changes `p` so that it points to the next element in the array, and `--p` changes `p` so that it points to the previous element
- If `p` and `q` are both pointers into the same array, then the expression `q - p` gives an integer that indicates the distance between `q` and `p`

❑Example: If `*q` is the element after `*p`, then `q - p` is 1

Implementing the STL deque Class

Constructor

```
template <class Item>
deque<Item>::deque (int init_bp_array_size, int init_block_size)
{
        bp_array_size = init_bp_array_size;
        block_size = init_block_size;

        //set a pointer the start of the array of block pointers
        block_pointers = new value_type* [bp_array_size];

        for (size_type index = 0; index < bp_array_size; ++index)
        {
            block_pointers[index] = NULL;
        }

        //set a pointer the end of the array of block pointers
        block_pointers_end = block_pointers + (bp_array_size - 1);

        first_bp = last_bp = NULL;
        front_ptr = back_ptr = NULL;
    }
```

Implementing the STL deque Class

Destructor

```
emplate <class Item>
deque<Item>::~deque ()
{

   for (size_type index = 0; index < bp_array_size; ++index)
   {
        if (block_pointers[index] != NULL)
              delete [] block_pointers[index];
    }

    delete [] block_pointers;

    first_bp = last_bp = NULL;
    block_pointers_end = block_pointers = NULL;
    front_ptr = back_ptr = NULL;

}
```

Implementing the STL deque Class

Clear

```
template <class Item>
void deque<Item>::clear ()
{

    for (size_type index = 0; index < bp_array_size; ++index)
    {
         delete [] block_pointers[index];
         block_pointers[index] = NULL;
     }

     first_bp = last_bp = NULL;
     front_ptr = back_ptr = NULL;

}
```

Implementing the STL deque Class

Implementation of the deque's pop_back Function

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

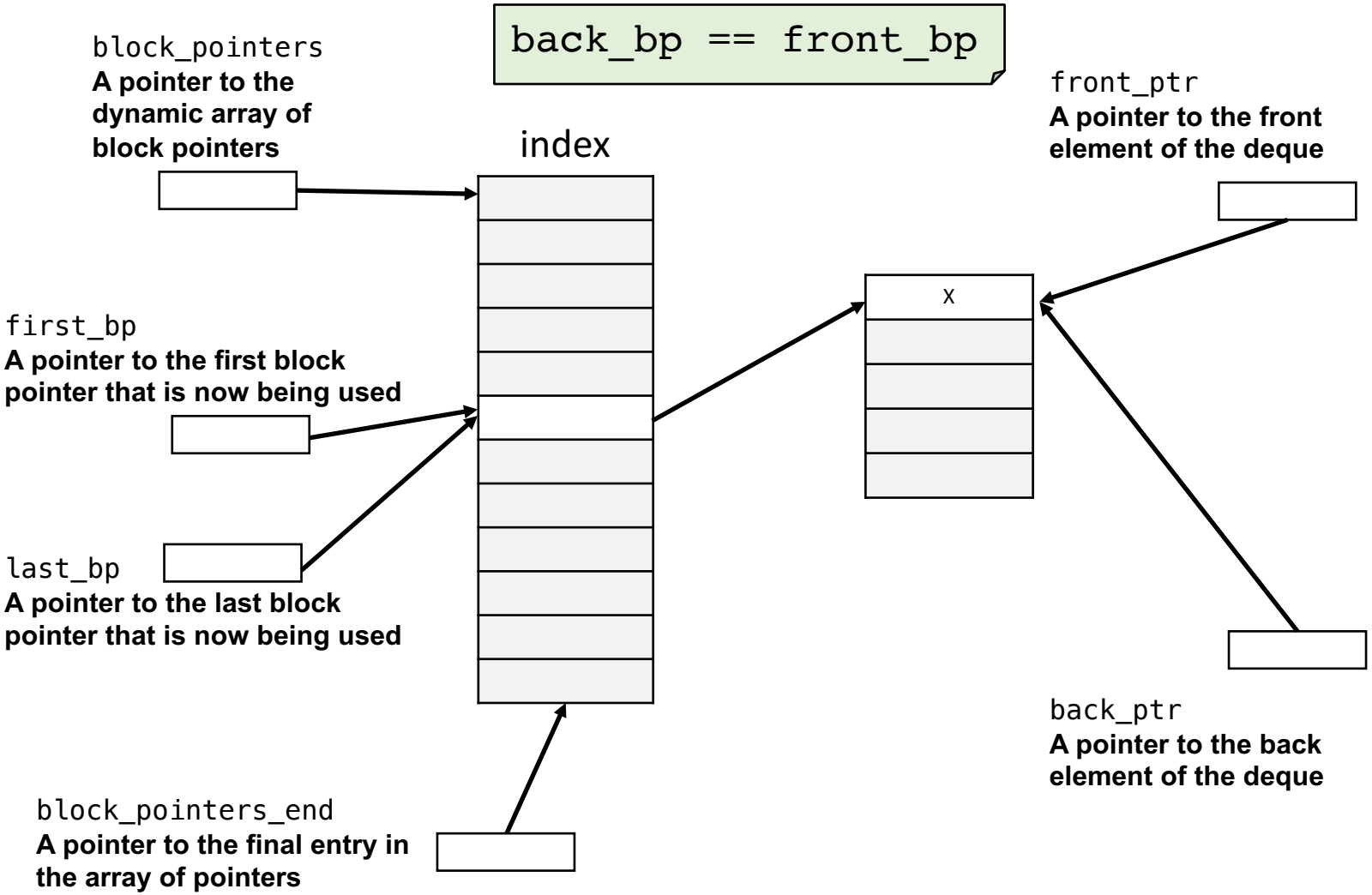
Case 1: There is just one block with just one element, so delete it, and reset things back to the start

See the next slide

```
if (back_ptr == front_ptr)
 {
   clear( );
 }
```

Implementing the STL deque Class

Implementation of the deque's pop_back Function



Implementing the STL deque Class

Implementation of the deque's pop_back Function

Case 2: `back_ptr` is pointing to the first element of the last block in the deque:

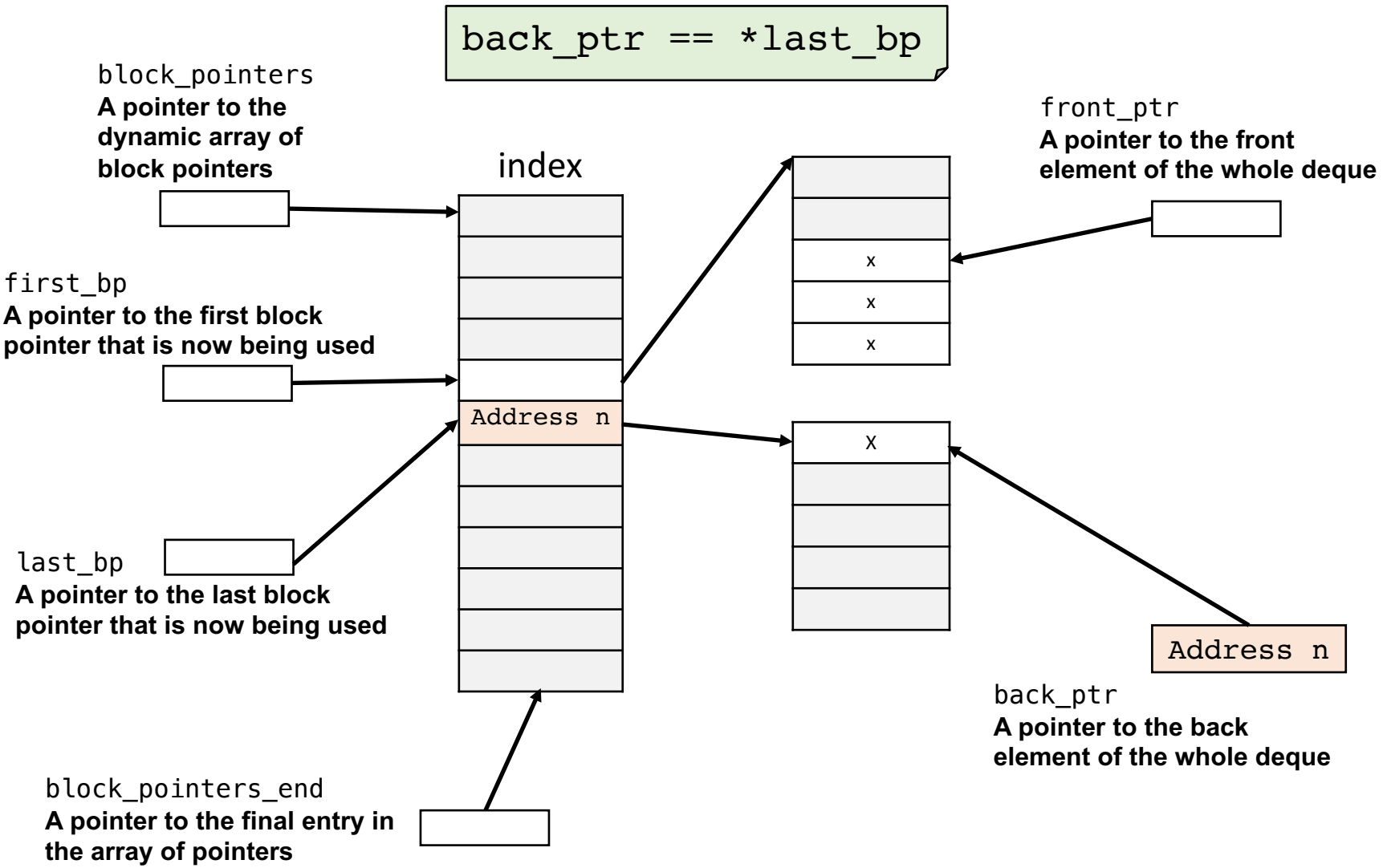
- Remove the entire last block. Note that this will call the destructor for each item in the block: `delete [] back_ptr;`
- The last block pointer that we are using is now one spot earlier in the array of block pointers: `--last_bp;`
- The new back element is now the last element in the last block

```
else if (back_ptr == *last_bp)
 {
    delete [] *last_bp;
    *last_bp = NULL;
    --last_bp;
    back_ptr = (*last_bp) + (BLOCK_SIZE - 1);
 }
```

See the next slide

Implementing the STL deque Class

Implementation of the deque's pop_back Function



Implementing the STL deque Class

Implementation of the deque's pop_back Function

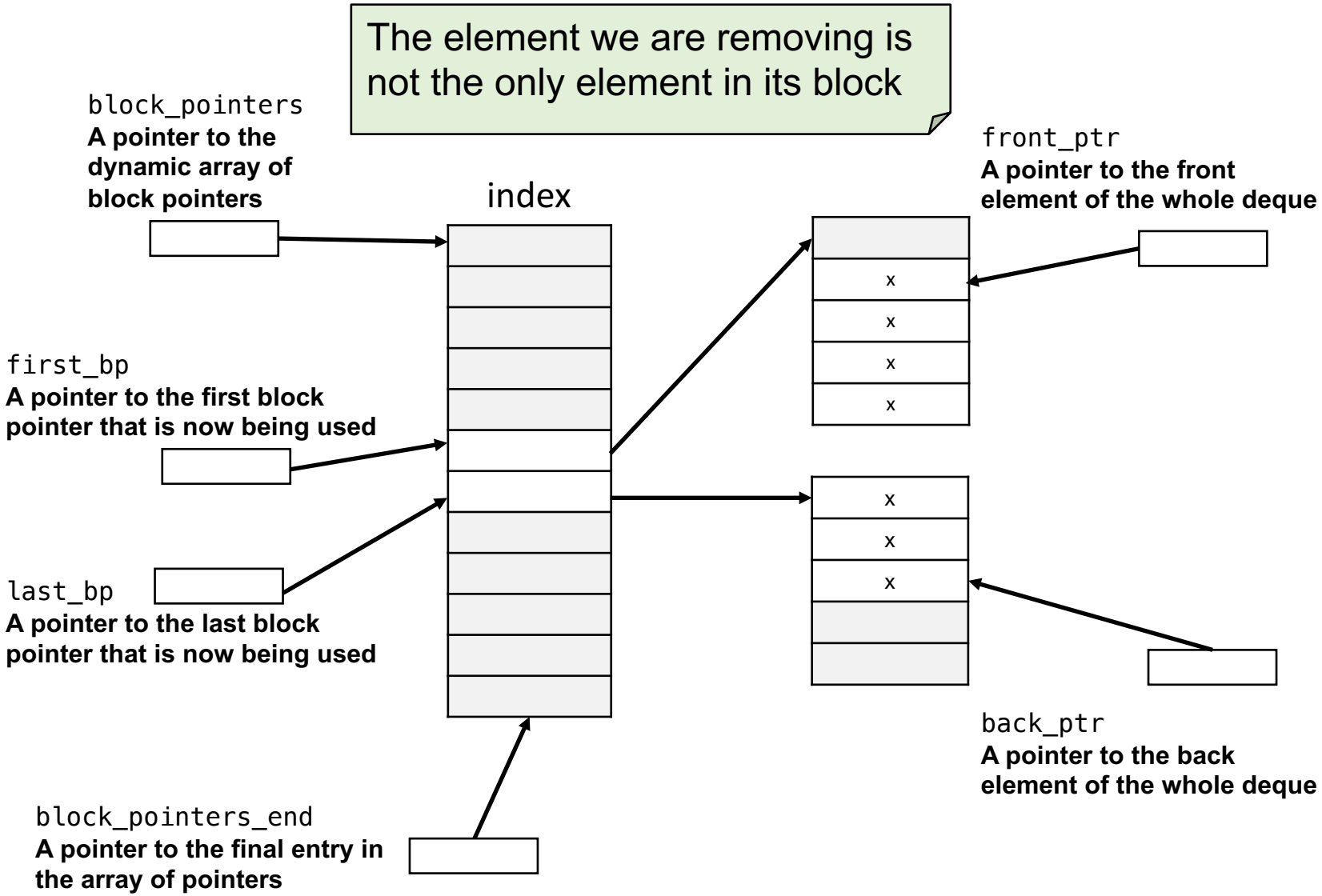
Case 3: The element we are removing is not the only element in its block, so we just move the `back_ptr` backward

```
else
{
      --back_ptr;
}
```

See the next slide

Implementing the STL deque Class

Implementation of the deque's pop_back Function



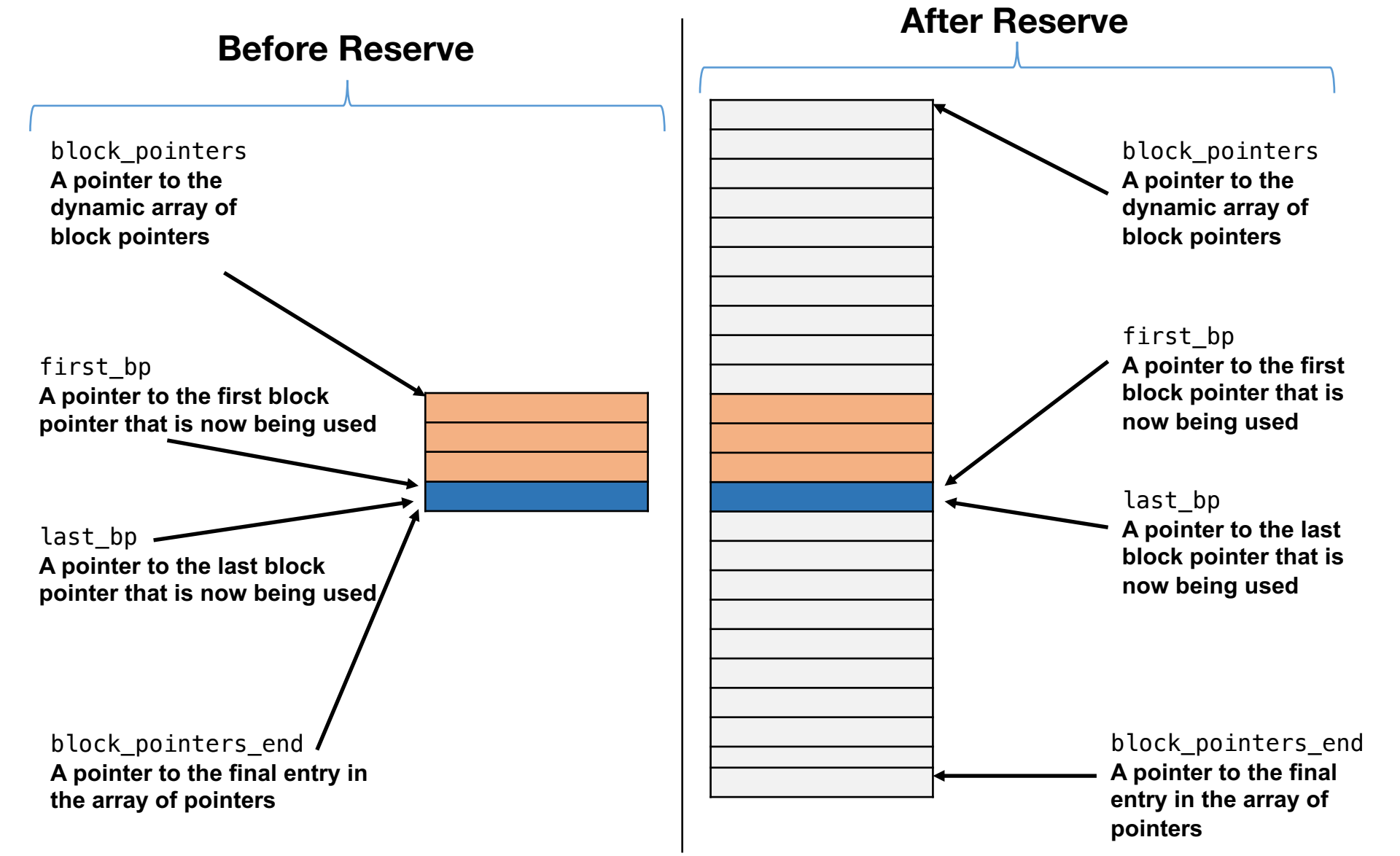
Implementing the STL deque Class

Reserve

- To increase the size of the deque (i.e., `reserve` function)
  - We plan to increase the size of the deque by 20 X `BLCOK_SIZE`
  - Therefore, we need to increase the size of the array of block pointers by 20
  - We also copy the existing array of block pointers to the middle of the new array of block pointers to improve the flexibility of adding at the two ends of the deque

Implementing the STL deque Class

Reserve



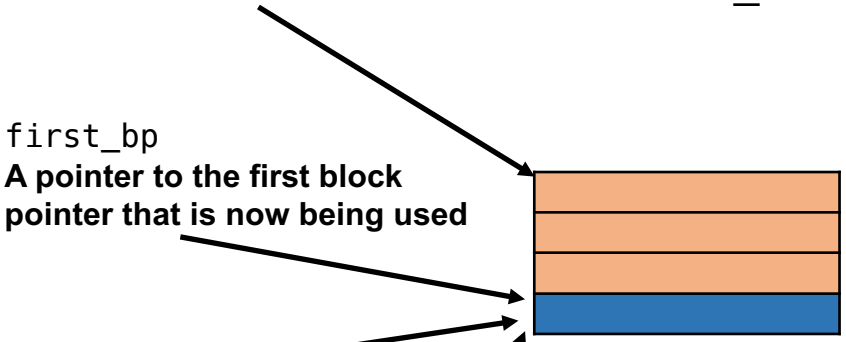
Implementing the STL deque Class

Increasing the Size of deque

Before Reserve

`block_pointers`
A pointer to the dynamic array of block pointers

```
offset_first_bp = first_bp - block_pointers = 3
offset_last_bp = last_bp - block_pointers = 3
```



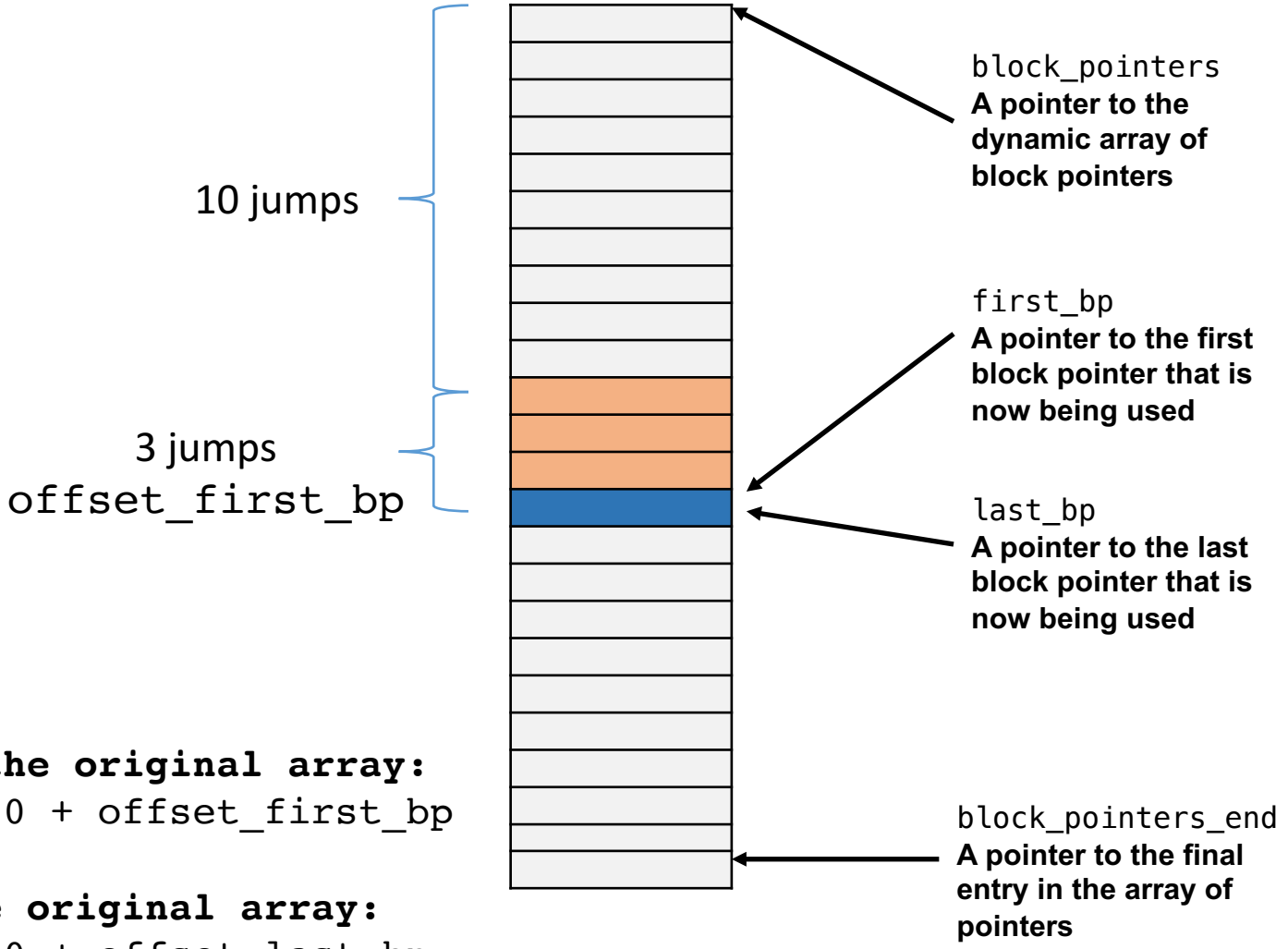
`first_bp`
A pointer to the first block pointer that is now being used

`last_bp`
A pointer to the last block pointer that is now being used

`block_pointers_end`
A pointer to the final entry in the array of pointers

Implementing the STL deque Class

Increasing the Size of deque



Start of copying the original array:

```
block_pointers + 10 + offset_first_bp
```

End of copying the original array:

```
block_pointers + 10 + offset_last_bp
```

Implementing the STL deque Class

Reserve

```
template <class Item>
void deque<Item>::reserve() {
   size_type newSize = bp_array_size + 20;

   value_type** new_block_pointers = new value_type* [newSize];

   for (size_type index = 0; index < newSize; ++index)
        new_block_pointers[index] = NULL;

   size_type offsett_first_bp = first_bp - block_pointers;
   size_type offsett_last_bp = last_bp - block_pointers;

   std::copy(first_bp, last_bp + 1,
             new_block_pointers + 10 + offsett_first_bp);

        delete [] block_pointers;

        block_pointers = new_block_pointers;
        bp_array_size = newSize;
        block_pointers_end = block_pointers + bp_array_size - 1;
        first_bp = block_pointers + offsett_first_bp + 10;
        last_bp = block_pointers + offsett_last_bp + 10;
}
```

Implementing the STL deque Class

Iterator Invalidation Rules for STL's deque

- **Note that an iterator pointing to an item of a deque cannot be implemented by only using a pointer to an item**
- The iterator needs to know which array it is part of, and where the index is, so it can find the next/previous arrays

Insertion:

- All iterators and references are invalidated because existing elements must be shifted
- If inserted member is at the front or back of the deque, all iterators are invalidated, but references to elements are unaffected
  - It fills out the first/last array, and then adds another one when necessary, so it never needs to move *existing* elements
  - However, the index vector might be copied and reallocated when resized

Iterator Invalidation Rules for STL's deque

Erasure:

- All iterators and references are invalidated, unless the erased members are at the front or back of the deque (in which case only iterators and references to the erased members are invalidated)

Queue Applications

Queue Applications

Uses for Queues (Cont'd)

- **Queues are frequently used in simulation programs**
  - ❑Example: A program to simulate the traffic at an intersection might use a software queue to simulate the real-life situation of a growing line of automobiles waiting for a traffic light to change from red to green
- Queues also appear in computer system software, such as the operating system that runs on your PC
  - ❑Example: For buffering input characters
- In a computer system in which more than one process or component uses a single resource, a queue is often used so that the processes or components wait in line and are served on a "first-come, first-served" basis

Queue Applications

Programming Example: Recognizing Palindromes

- **Palindrome** is a string that reads the same forward and backward
  - ❑ Example: “radar” is a palindrome
  - ❑ A more complicated example: `Able was I ere I saw Elba`
- Suppose we want a program to read a line of text and tell us if the line is a palindrome
- **We can do this by using: a stack and a queue**
- **We will read the line of text into both a stack and a queue**, and then write out the contents of the stack and the contents of the queue
- The line that is written using the queue is written forward, and the line that is written using the stack is written backward
- Now, if those two output lines are the same, then the input string must be a palindrome

Queue Applications

A Program to Recognize Palindromes (Cont'd)

```
// FILE: pal.cxx
// Program to test whether an input line is a palindrome. Spaces,
// punctuation, and the difference between upper- and lowercase are
// ignored.

#include <cassert>    // Provides assert
#include <cctype>     // Provides isalpha, toupper
#include <cstdlib>    // Provides EXIT_SUCCESS
#include <iostream>   // Provides cout, cin, peek
#include <queue>      // Provides the queue template class
#include <stack>      // Provides the stack template class
using namespace std;

int main( )
{
    queue<char> q;
    stack<char> s;
    char letter;
    queue<char>::size_type mismatches = 0; // Mismatches between queue
                                           // and stack
```

Queue Applications

A Program to Recognize Palindromes (Cont'd)

```
cout << "Enter a line and I will see if it's a palindrome:" << endl;

while (cin.peek( ) != '\n')
  {
      cin >> letter;
      if (isalpha(letter))
      {
          q.push(toupper(letter));
          s.push(toupper(letter));
      }
  }

while ((!q.empty( )) && (!s.empty( )))
  {
      if (q.front( ) != s.top( ))
          ++mismatches;
      q.pop( );
      s.pop( );
  }
```

returns true if its single argument is one of the alphabetic characters

converts a lowercase letter to corresponding uppercase letter and returns this value

Queue Applications

A Program to Recognize Palindromes (Cont'd)

```
    if (mismatches == 0)
        cout << "That is a palindrome." << endl;
    else
        cout << "That is not a palindrome." << endl;

    return EXIT_SUCCESS;
}
```

A Program to Recognize Palindromes (Cont'd)

- Sample Dialogues from the Palindrome Program

First Sample Dialogue

```
Enter a line and I will see if it's a palindrome:
Straw? No, too stupid a fad. I put soot on warts.
That is a palindrome.
```

Second Sample Dialogue

```
Enter a line and I will see if it's a palindrome:
Able were you ere you saw Elba.
That is not a palindrome.
```

A Program to Recognize Palindromes (Cont'd)

- Our program also ignores spaces and punctuation, requiring only that the letters on the line read the same forward and backward

❑Example: You might not immediately recognize the following as a palindrome: `Straw? No, too stupid a fad. I put soot on warts`

- Our program ignores blanks and punctuation, so, according to our program, the above is a palindrome
- The function, called `isalpha`, returns true if its single argument is one of the alphabetic characters
  - From `<cctype>`

Summary

Summary

- A queue is a First-In/First-Out data structure
- A queue can be used to buffer data that is being sent from a fast computer component to a slower component
- Queues have many other applications: in simulation programs, operating systems, and elsewhere
- 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

References

1) Data Structures and Other Objects Using C++, Michael Main, Walter Savitch, 4th Edition
2) Data Structures and Algorithm Analysis in C++, by Mark A. Weiss, 4TH Edition
3) C++: Classes and Data Structures, by Jeffrey Childs
4) http://en.cppreference.com
5) http://www.cplusplus.com
6) https://isocpp.org