Linked Lists: Implementation continued
Sequences in the C++ STL

CS 311 Data Structures and Algorithms

Lecture Slides

Wednesday, April 1, 2009

Glenn G. Chappell

Department of Computer Science
University of Alaska Fairbanks
CHAPPELLG@member.ams.org

Unit Overview
Sequences & Their Implementations

Major Topics

- ✓ Introduction to Sequences
- ✓ Smart arrays
  - ✓ Interface
  - ✓ A basic implementation
  - ✓ Exception safety
  - ✓ Allocation & efficiency
  - ✓ Generic containers
- Linked Lists
  - ✓ Node-based structures
  - (part) Implementation
- Sequences in the C++ STL
- Stacks
- Queues

Review
Smart Arrays: Allocation & Efficiency

An operation is **amortized constant time** if *k* operations require *O(k)* time.

- Thus, over many consecutive operations, the operation averages constant time.
- *Not* the same as constant-time average case.
- Quintessential amortized-constant-time operation: insert-at-end for a well written (smart) array.
- Amortized constant time is not something we can easily compare with (say) logarithmic time.

Review
Generic Containers [1/2]

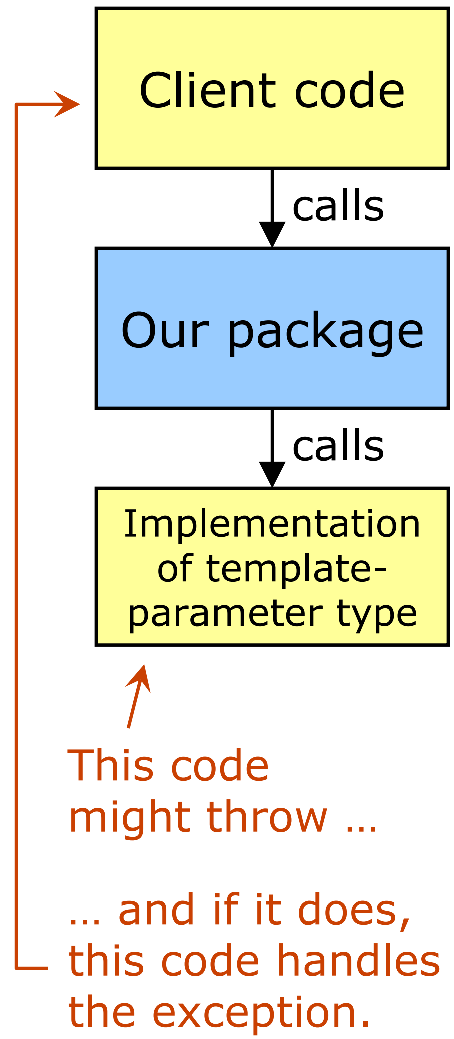
A **generic container** is a container that can hold a client-specified data type.

- In C++ we usually implement a generic container using a **class template**.

A function that allows exceptions thrown by a client's code to propagate unchanged, is said to be **exception-neutral**.

When exception-neutral code calls a client-provided function that may throw, it does one of two things:

- Call the function outside a try block, so that any exceptions terminate our code immediately.
- Or, call the function inside a try block, then catch all exceptions, do any necessary clean-up, and re-throw.



Review
Generic Containers [2/2]

We can use catch-all, clean-up, re-throw to get both exception safety and exception neutrality.

```
arr = new MyType[10];
try
{
    std::copy(a, a+10, arr);
}
catch (...)
{
    delete [] arr;
    throw;
}
```

Called outside any **`try`** block. If this fails, we exit immediately, throwing an exception.

Called inside a **`try`** block. If this fails, we need to deallocate the array before exiting.

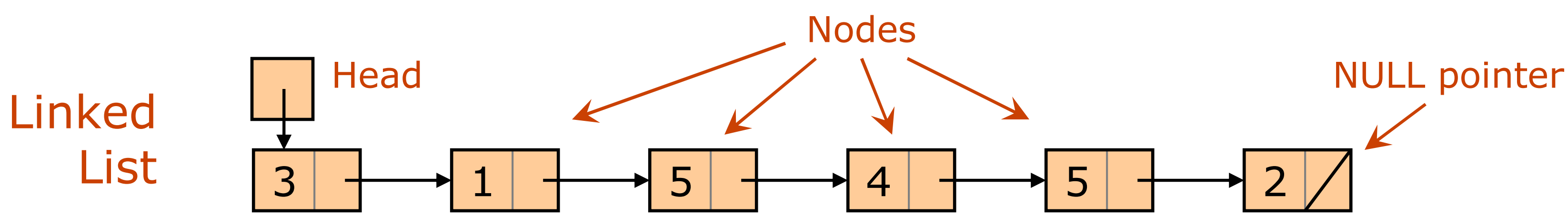
This helps us meet the Basic Guarantee (also the Strong Guarantee if this function does nothing else).

This makes our code exception-neutral.

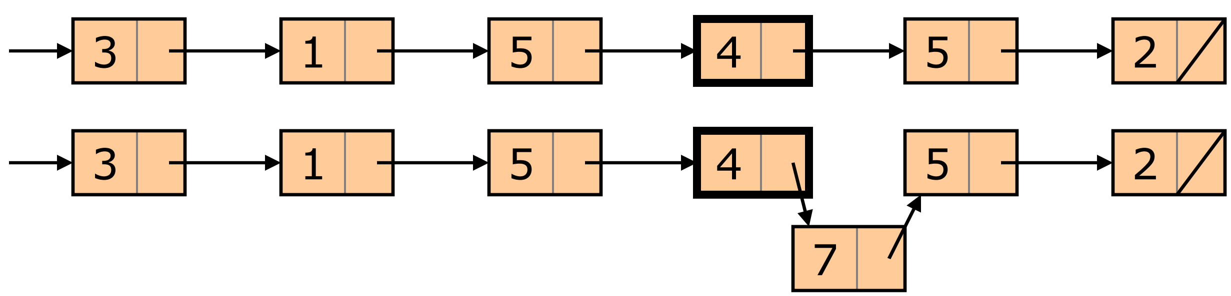
Review
Node-Based Structures — Linked Lists [1/5]

Our first node-based data structure is a (**Singly**) **Linked List**.

- A Linked List is composed of nodes. Each has a single data item and a pointer to the next node.

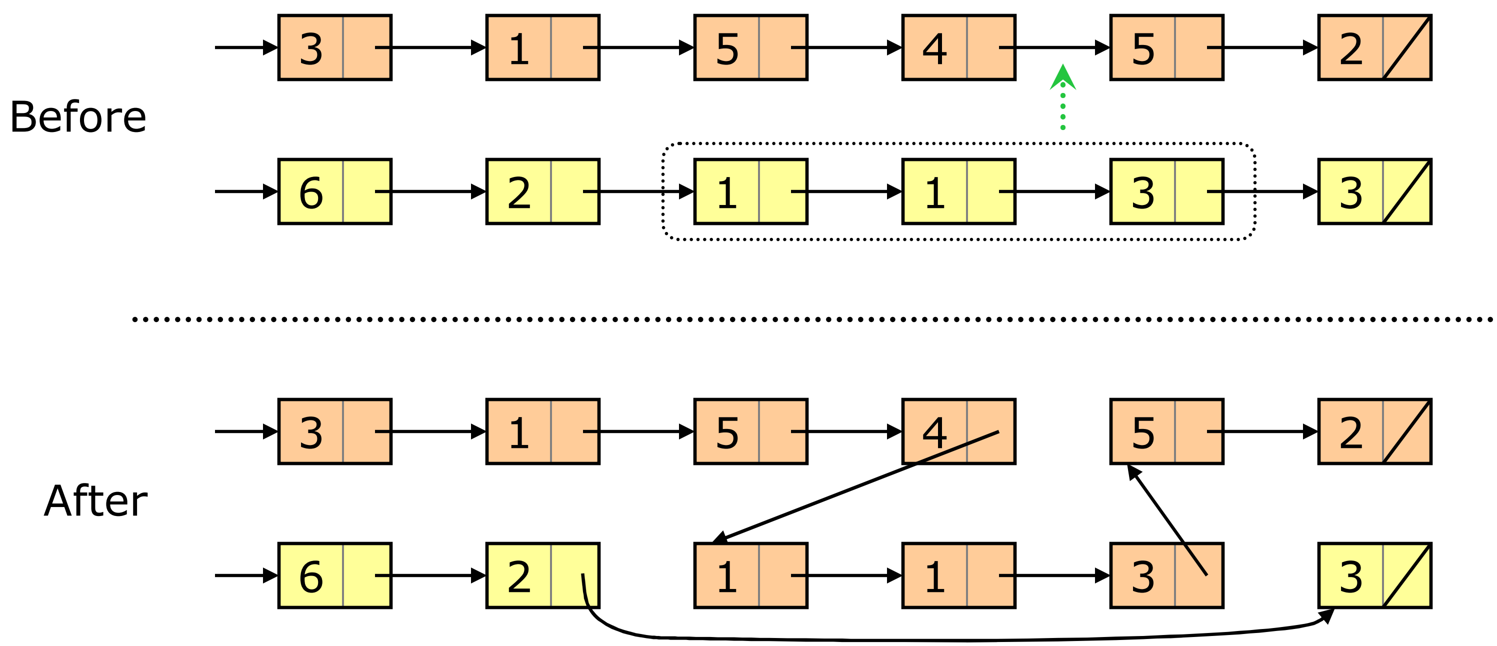


- These pointers are the **only** way to find the next data item.
- Once we have found a position within a Linked List, we can insert and delete in constant time.



Review
Node-Based Structures — Linked Lists [2/5]

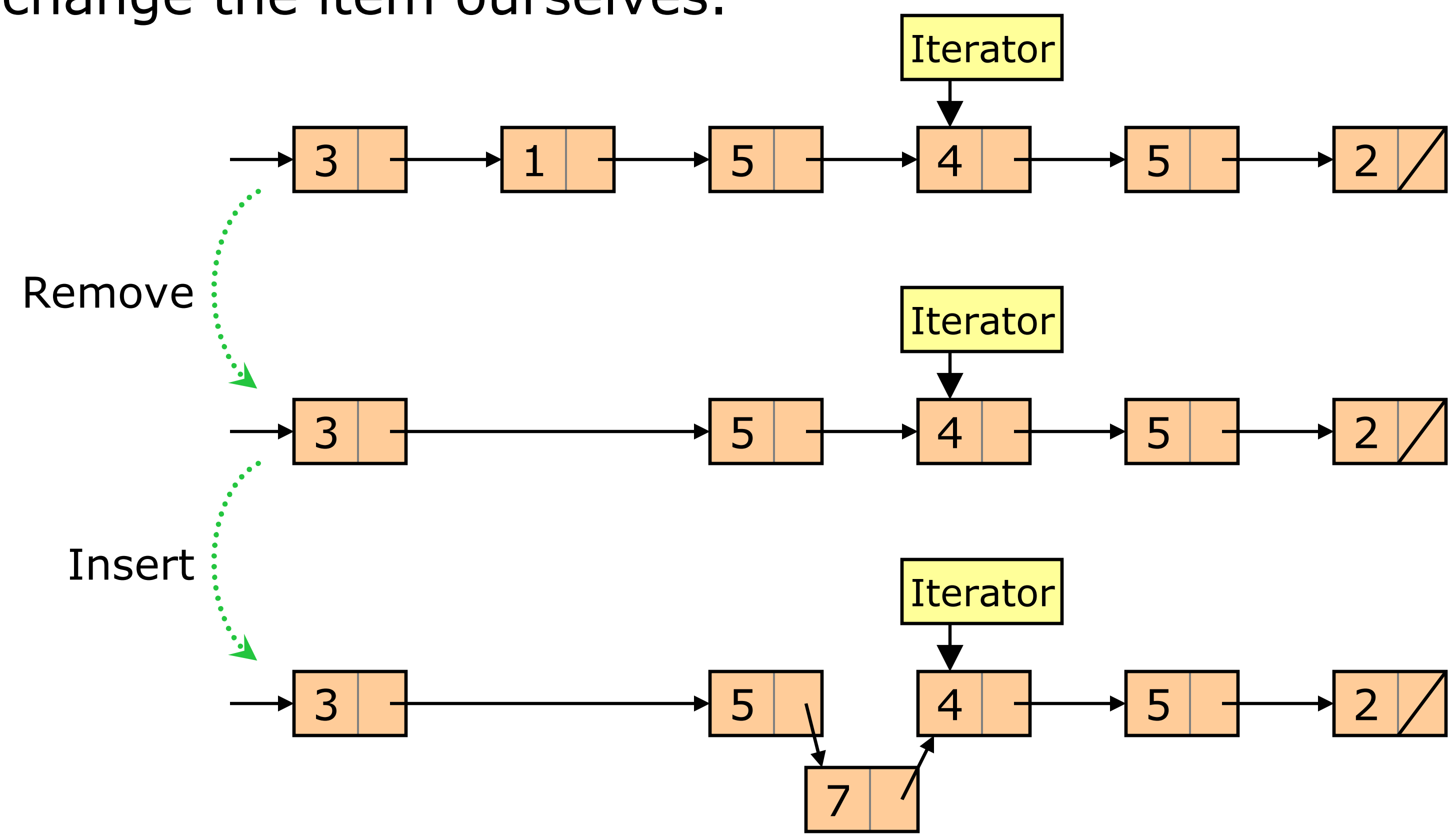
Also, with Linked Lists, we can do a fast **splice**:



Note: If we allow for efficient splicing, then we cannot efficiently keep track of a Linked List's size.

Review
Node-Based Structures — Linked Lists [3/5]

Further, with Linked Lists, iterators, pointers, and references to items will always stay valid and never change what they refer to, as long as the Linked List exists — unless we remove or change the item ourselves.



Review
Node-Based Structures — Linked Lists [4/5]

	Smart Array	Linked List
Look-up by index	$\boldsymbol{O(1)}$	$\boldsymbol{O(n)}$
Search sorted	$\boldsymbol{O(\log n)}$	$\boldsymbol{O(n)}$
Search unsorted	$O(n)$	$O(n)$
Sort	$O(n \log n)$	$O(n \log n)$
Insert @ given pos	$\boldsymbol{O(n)}$	$\boldsymbol{O(1)}$*
Remove @ given pos	$\boldsymbol{O(n)}$	$\boldsymbol{O(1)}$*
Splice	$\boldsymbol{O(n)}$	$\boldsymbol{O(1)}$
Insert @ beginning	$\boldsymbol{O(n)}$	$\boldsymbol{O(1)}$
Remove @ beginning	$\boldsymbol{O(n)}$	$\boldsymbol{O(1)}$
Insert @ end	$O(1)$ or $O(n)$** amortized const	$O(1)$ or $O(n)$***
Remove @ end	$O(1)$	$O(1)$ or $O(n)$***
Traverse	$O(n)$	$O(n)$
Copy	$O(n)$	$O(n)$
Swap	$O(1)$	$O(1)$

*For Singly Linked Lists, we mean inserting or removing just *after* the given position.
- Doubly Linked Lists can help.

**$O(n)$ if reallocation occurs. Otherwise, $O(1)$. Amortized constant time.
- Pre-allocation can help.

***For $O(1)$, need a pointer to the end of the list. Otherwise, $O(n)$.
- This is tricky.
- Doubly Linked Lists can help.

Find faster with an array

Rearrange faster with a Linked List

Review
Node-Based Structures — Linked Lists [5/5]

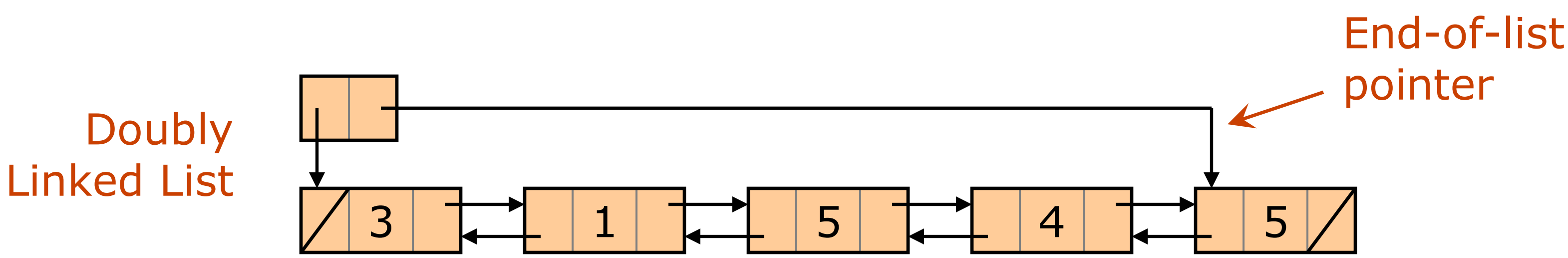
Other Issues

- ☹ Linked Lists use **more memory**.
- ☹ When order is the same, Linked Lists are almost always **slower**.
  - Arrays might be 2–10 times faster.
- ☹ Arrays keep consecutive items in **nearby memory locations**.
  - Many algorithms have the property that when they access a data item, the following accesses are likely to be to the same or nearby items.
    - This property of an algorithm is called **locality of reference**.
  - Once a memory location is accessed, a memory cache will automatically load nearby memory locations. With an array, these are likely to hold nearby data items.
  - Thus, when a memory cache is used, an array can have a significant speed advantage over a Linked List, when used with an algorithm that has good locality of reference.
- ☺ With an array, iterators, pointers, and references to items can be **invalidated** by reallocation. Also, insert/remove can change the item they reference.

Review
Node-Based Structures — Linked List Variations [1/2]

In a **Doubly Linked List**, each node has a data item & **two pointers**:

- A pointer to the next node.
- A pointer to the previous node.



Doubly Linked Lists often have an end-of-list pointer.

- This can be efficiently maintained, resulting in constant-time insert and remove at the end.

Doubly Linked Lists are generally considered to be a good basis for a **general-purpose** generic container type.

- Singly-Linked Lists are not. Remember all those asterisks?

Review
Node-Based Structures — Linked List Variations [2/2]

	Smart Array	**Doubly** Linked List
Look-up by index	$\boldsymbol{O(1)}$	$\boldsymbol{O(n)}$
Search sorted	$\boldsymbol{O(\log n)}$	$\boldsymbol{O(n)}$
Search unsorted	$O(n)$	$O(n)$
Sort	$O(n \log n)$	$O(n \log n)$
Insert @ given pos	$\boldsymbol{O(n)}$	$\boldsymbol{O(1)}$
Remove @ given pos	$\boldsymbol{O(n)}$	$\boldsymbol{O(1)}$
Splice	$\boldsymbol{O(n)}$	$\boldsymbol{O(1)}$
Insert @ beginning	$\boldsymbol{O(n)}$	$\boldsymbol{O(1)}$
Remove @ beginning	$\boldsymbol{O(n)}$	$\boldsymbol{O(1)}$
Insert @ end	$O(1)$ or $O(n)$* amortized const	$O(1)$
Remove @ end	$O(1)$	$O(1)$
Traverse	$O(n)$	$O(n)$
Copy	$O(n)$	$O(n)$
Swap	$O(1)$	$O(1)$

Find faster
with an array

Rearrange faster
with a Linked List

With Doubly Linked Lists, we can get rid of most of our asterisks.

*$O(n)$ if reallocation occurs. Otherwise, $O(1)$. Amortized constant time.

- Pre-allocation can help.

Review
Linked Lists: Implementation

Two approaches to implementing a Linked List:

- A Linked List package to be used by others.
- A Linked List as part of some other package, and not exposed to clients.

Linked Lists: Implementation — continued
Write It

TO DO

- Write an insert-at-beginning operation for a Linked List.

Done. See **`linked_list.cpp`**, *on the web page.*

Sequences in the C++ STL
Generic Sequence Types — Introduction

The C++ STL has four generic Sequence container types.

- Class template **`std::vector`**.
  - A “smart array”.
  - Much like what we wrote, but with more member functions.
- Class template **`std::basic_string`**.
  - Much like **`std::vector`**, but aimed at character string operations.
  - Mostly we use **`std::string`**, which is really **`std::basic_string<char>`**.
  - Also **`std::wstring`**, which is really **`std::basic_string<std::wchar_t>`**.
- Class template **`std::list`**.
  - A Doubly Linked List.
    - Note: The Standard does not specify implementation. It specifies the semantics and order of operations. These were written with a Doubly Linked List in mind, and a D.L.L. is the usual implementation.
- Class template **`std::deque`**.
  - Deque stands for **D**ouble-**E**nded **QUE**ue.
  - Say “deck”.
  - Like **`std::vector`**, but a bit slower. Allows fast insert/remove at both beginning and end.

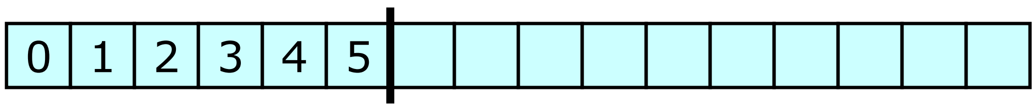
Sequences in the C++ STL
Generic Sequence Types — **`std::deque`** [1/4]

We are familiar with smart arrays and Linked Lists. How is **`std::deque`** implemented?

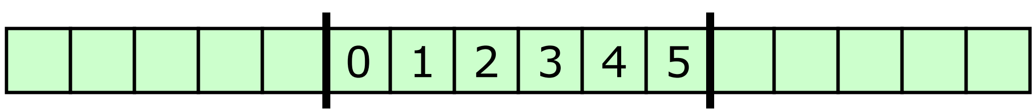
- There are two big ideas behind it.

First Idea

- A **`vector`** uses an array in which data are stored at the beginning.



  - This gives linear-time insert/remove at beginning, constant-time remove at end, and, if we do it right, amortized-constant-time insert at end.
- What if we store data in the middle? When we reallocate-and-copy, we move our data to the middle of the new array.

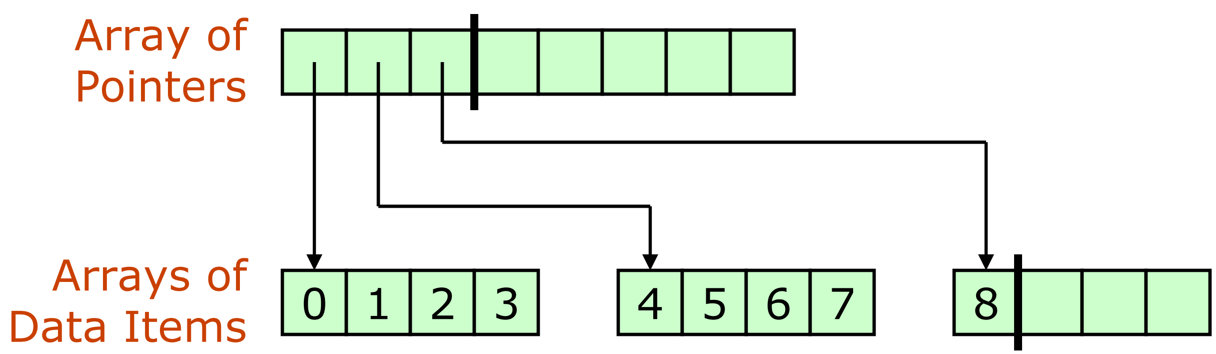


  - Result: Amortized-constant-time insert, and constant-time remove, at **both** ends.

Sequences in the C++ STL
Generic Sequence Types — `std::deque` [2/4]

Second Idea

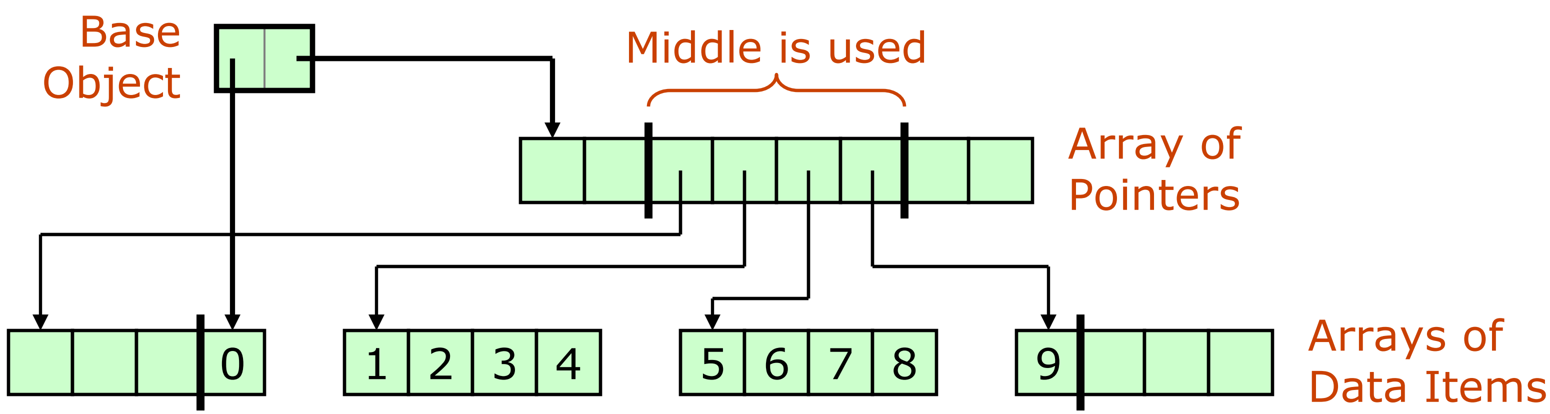
- Doing reallocate-and-copy for a **`vector`** requires calling either the copy constructor or copy assignment for **every data item**.
  - For large, complex data items, this can be time-consuming.
- Instead, let our array be an **array of pointers to arrays**, so that reallocate-and-copy only needs to move the pointers.
  - This still lets us keep most of the locality-of-reference advantages of an array, when the data items are small.



Sequences in the C++ STL
Generic Sequence Types — **`std::deque`** [3/4]

An implementation of **`std::deque`** typically uses both of these ideas.

- It probably uses an array of pointers to arrays.
  - This *might* go deeper (array of pointers to arrays of pointers to arrays).
- The arrays may not be filled all the way to the beginning or the end.
- Reallocate-and-copy moves the data to the middle of the new array of pointers.



Thus, **`std::deque`** is an array-ish container, optimized for:

- Insert/remove at either end.
- Possibly large, difficult-to-copy data items.

The cost is complexity, and a slower [but still $O(1)$] look-up by index.

Sequences in the C++ STL
Generic Sequence Types — **`std::deque`** [4/4]

Essentially, **`std::deque`** is an array.

- Iterators are random-access.
- But it has some complexity to it, so it is a slow-ish array.

However, insertions at the beginning do not require items to be moved up.

- We speed up insert-at-beginning by allocating extra space before existing data.

And reallocate-and-copy leaves the data items alone.

- We also speeds up insertion by trading value-type operations for pointer operations.
- Pointer operations can be much faster than value-type operations. A **`std::deque`** can do reallocate-and-copy using a raw memory copy, with no value-type copy ctor calls.

Like **`vector`**, **`deque`** *tends* to keep consecutive items in nearby memory locations.

- So it avoids cache misses when used with algorithms having good locality of reference.

The Bottom Line

- A **`std::deque`** is generally a good choice when you need fast insert/remove at both ends of a Sequence.
- Especially if you also want fast-ish look-up.
- Some people also recommend **`std::deque`** whenever you will be doing a lot of resizing, but do not need fast insert/remove in the middle.

Sequences in the C++ STL
Generic Sequence Types — Efficiency [1/2]

We determine efficiency by counting operations. How do we count operations for a generic container type?

- We count both built-in operations and value-type operations.
- However, we typically expect that the most time-consuming operations are those on the value type.

The C++ Standard, on the other hand, counts **only** value-type operations.

- For example, “constant time” in the Standard means that at most a constant number of value-type operations are performed.

Sequences in the C++ STL
Generic Sequence Types — Efficiency [2/2]

	`vector`, `basic_string`	`deque`	`list`
Look-up by index	Constant	Constant	Linear
Search sorted	Logarithmic	Logarithmic	Linear
Insert @ given pos	Linear	Linear	Constant
Remove @ given pos	Linear	Linear	Constant
Insert @ beginning	Linear	Linear/ Amortized Constant*	Constant
Remove @ beginning	Linear	Constant	Constant
Insert @ end	Linear/ Amortized Constant**	Linear/ Amortized Constant*	Constant
Remove @ end	Constant	Constant	Constant

*Only a constant number of value-type operations are required.

- The C++ standard counts only value-type operations. Thus, it says that insert at beginning or end of a **`std::deque`** is constant time.

**Constant time if sufficient memory has already been allocated.

All have $O(n)$ traverse, copy, and search-unsorted, $O(1)$ swap, and $O(n \log n)$ sort.

Sequences in the C++ STL
Generic Sequence Types — Common Features

All STL Sequence containers have:

- **iterator**, **const_iterator**
  - Iterator types. The latter acts like a pointer-to-const.
  - **vector**, **basic_string**, and **deque** have random-access iterators.
  - **list** has bidirectional iterators.
- *iterator* **begin()**, *iterator* **end()**
- *iterator* **insert(***iterator*, *item***)**
  - Insert before. Returns position of new item.
- *iterator* **erase(***iterator***)**
  - Remove this item. Returns position of next item.
- **push_back(***item***)**, **pop_back()**
  - Insert & remove at the end.
- *reference* **front()**, *reference* **back()**
  - Return reference to first, last item.
- **clear()**
  - Remove all items.
- **resize(***newSize***)**
  - Change the size of the container.
  - Not the same as **vector::reserve**, which sets capacity.

In addition, **deque** and **list** also have:

- **push_front(***item***)**, **pop_front()**
  - Insert & remove at the beginning.

In addition, **vector**, **basic_string**, and **deque** also have:

- *reference* **operator[](***index***)**
  - Look-up by index.

In addition, **vector** also has:

- **reserve(***newCapacity***)**
  - Sets capacity to at least the given value.

And there are other members …

Sequences in the C++ STL
Iterator Validity — The Idea

One of the trickier parts of using container types is making sure you do not use an iterator that has become "invalid".

- Generally, *valid* iterators are those that can be dereferenced.
- We also call things like **`container.end()`** valid.
  - These are "past-the-end" iterators.

Consider the smart-array class in Assignment 5. When is one of its iterators invalidated?

- When reallocate-and-copy occurs.
- When the container is destroyed.
- When the container is resized so that the iterator is more than one past the end.

Now consider a (reasonable) Linked-List class with iterators. When are such iterators invalidated?

- Only when the item referenced is erased.
  - This includes container destruction.

Sequences in the C++ STL
Iterator Validity — Rules

We see that different container types have different iterator-validity rules.

- When using a container, it is important to **know the associated rules**.

A related topic is **reference validity**.

- Items in a container can be referred to via iterators, but also via pointers and references.
- *Reference-validity* rules indicate when pointers and references remain usable.
- Often these are the same as the iterator-validity rules, but not always.

Sequences in the C++ STL
Iterator Validity — `std::vector`

For **`std::vector`**

- Reallocate-and-copy invalidates **all** iterators and references.
- When there is no reallocation, the Standards says that insertion and erasure invalidate all iterators and references except those **before** the insertion/erasure.
  - Apparently, the Standard counts an iterator as invalidated if the item it points to changes.

A **`vector`** can be forced to pre-allocate memory using **`std::vector::reserve`**.

- The amount of pre-allocated memory is the vector's *capacity*.
- We have noted that pre-allocation makes insert-at-end a constant-time operation. Now we have another reason to do pre-allocation: preserving iterator and reference validity.

Sequences in the C++ STL
Iterator Validity — `std::deque`

For **`std::deque`**

- Insertion in the **middle** invalidates **all** iterators and references.
- Insertion at either **end** invalidates all iterators, but no **references**.
  - Why?
- Erasure in the middle invalidates **all** iterators and references.
- Erasure at the either end invalidates only iterators and references to items erased.

So deques have some validity advantages over vectors.

Sequences in the C++ STL
Iterator Validity — `std::list`

For **`std::list`**

- An iterator or reference always remains valid until the item it points to goes away.
  - When the item is erased.
  - When the list is destroyed.

In some situations, these validity rules can be a big advantage of **`std::list`**.

Sequences in the C++ STL
Iterator Validity — Example

```
// v is a variable of type vector<int>
// Insert a 1 before each 2 in v:
for (vector<int>::iterator iter = v.begin();
     iter != v.end();
     ++iter)
{
   if (*iter == 2)
      v.insert(iter, 1);
}
```

What is wrong with the above code?

- The **`insert`** call invalidates iterator **`iter`**.
- Even if **`iter`** stays valid, after an insertion, it points to the 1 inserted. After being incremented, it points to the 2 again. Infinite loop.

How can we fix it? *Some ideas (most of which were discussed in class):*

- *Replace the "**`if`**" body with: "**`iter = v.insert(iter, 1); ++iter;`**".*
- *Use **indices** in the loop, instead of iterators.*
- *Use **`std::list`**, instead of **`std::vector`**.*
- *Pre-allocate using **`reserve`** (and increment **`iter`** in the "**`if`**").*