More on Linked Lists continued
Sequences in the C++ STL

CS 311 Data Structures and Algorithms
Lecture Slides
Monday, October 26, 2020

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
ggchappell@alaska.edu
© 2005–2020 Glenn G. Chappell
Some material contributed by Chris Hartman
Review
Our problem for most of the rest of the semester:

- **Store:** A collection of data items, all of the same type.
- **Operations:**
  - Access items [single item: retrieve/find, all items: traverse].
  - Add new item [insert].
  - Eliminate existing item [delete].
- Time & space efficiency are desirable.

A solution to this problem is a **container**.

In a **generic container**, client code can specify the value type.
Unit Overview
Data Handling & Sequences

Major Topics
✓ Data abstraction
✓ Introduction to Sequences
✓ Interface for a smart array
✓ Basic array implementation
✓ Exception safety
✓ Allocation & efficiency
✓ Generic containers
✓ Node-based structures
✓ More on Linked Lists
(part) Sequences in the C++ STL
✓ Stacks
✓ Queues

Smart Arrays
Linked Lists
Many data structures are built out of *nodes*.

- **A node** is usually a small block of memory that is referenced via a pointer, and which may reference other nodes via pointers.

- To find a node, follow a chain of pointers. Look-up can be slow.
- Operations that require rearrangement *might* be very fast.

Our pictures are usually prettier than the actual arrangement of nodes in memory.
The C++ Standard Library includes RAII class templates called **smart pointers**, which automatically handle ownership of dynamic objects.

**std::unique_ptr<T> (<memory>)**
- One-owner-at-a-time ownership of a dynamic object of type T.
- The destructor of an owning `unique_ptr` destroys the object pointed to.
- Movable but not copyable. Moving transfers ownership.

**std::shared_ptr<T> (<memory>)**
- Allows shared ownership of a dynamic object of type T.
- Uses a **reference count**. Destroys object when the count hits 0.
  - “The last one to leave turns out the lights.”
- Copyable. Copying grants shared ownership.
Create using \texttt{make\_unique/make\_shared}. Then we do neither the \texttt{new} nor the \texttt{delete} directly.

\begin{verbatim}
auto unp = make_unique<Foo>(5, "xy", 3.2);
\end{verbatim}

Dereference just like a pointer.

\begin{verbatim}
Foo x = *unp;
unp->bar();
\end{verbatim}

To obtain a regular pointer to the referenced object, call member function \texttt{get}.

\begin{verbatim}
Foo * p = unp.get()
\end{verbatim}
An **empty** smart pointer is one that does not point to anything. Check for this by treating a smart pointer like a `bool`.

```cpp
if (unp) // Nonempty?
    unp->bar();
```

Relinquish ownership *early* using member function `reset`.

```cpp
unp.reset();
```

With a `unique_ptr`, transfer ownership by passing/returning a `unique_ptr` by value (`std::move` may be required). Otherwise, pass a `unique_ptr` by reference or reference-to-const.

We can pass a `shared_ptr` by value arbitrarily, sharing ownership.
Programmers have found that shared ownership is rarely needed. It is true that, whenever you might use `unique_ptr`, it will also work to use `shared_ptr`. On the other hand, using `unique_ptr` helps the compiler find bugs for you (and it is more efficient!).

Suggestions

- When you want a smart pointer, start by using `unique_ptr` and `make_unique`.
- Pass a smart pointer by reference or reference-to-const when you do not want to transfer/share ownership.
- If code does not compile because it tries to copy a `unique_ptr`:
  - If you want to transfer ownership, then you might simply need to wrap the `unique_ptr` in `std::move`.
  - If you do not want to transfer ownership, then you can call `get` to obtain a non-owning regular pointer and use that instead.
  - If it turns out that you really need shared ownership, then do a global search/replace: `unique → shared`
Our first node-based data structure is the **(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 (think *iterator*) within a Linked List, we can insert or remove an item in constant time.

We can also **splice** in constant time.
**Review**  
**More on Linked Lists [2/5]**

<table>
<thead>
<tr>
<th></th>
<th>Smart Array</th>
<th>Linked List</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Look-up by index</strong></td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td><strong>Search sorted</strong></td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Search unsorted</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Sort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td><strong>Insert @ given pos</strong></td>
<td>$O(n)$</td>
<td>$O(1)^*$</td>
</tr>
<tr>
<td><strong>Remove @ given pos</strong></td>
<td>$O(n)$</td>
<td>$O(1)^*$</td>
</tr>
<tr>
<td>Splice</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td><strong>Insert @ beginning</strong></td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td><strong>Remove @ beginning</strong></td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Insert @ end</td>
<td>$O(n)^{**}$</td>
<td>$O(1)$ or $O(n)^{***}$</td>
</tr>
<tr>
<td></td>
<td>amortized const</td>
<td></td>
</tr>
<tr>
<td>Remove @ end</td>
<td>$O(1)$</td>
<td>$O(1)$ or $O(n)^{***}$</td>
</tr>
<tr>
<td>Traverse</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

*For Singly Linked Lists, insert/remove just *after* the given position.*
- Doubly Linked Lists can help.

**$O(1)$ if no reallocate-and-copy.**
- Pre-allocate to ensure this.

***For $O(1)$, need a pointer to end of list. Otherwise, $O(n)$.***
- This can be tricky.
- And, for remove @ end, it is mostly impossible.
- Doubly Linked Lists can help.

**Find** faster with an array  
**Rearrange** faster with a Linked List
Arrays store consecutive items in *nearby memory locations*.

Modern processors **prefetch** memory locations near those accessed, storing them in a fast on-chip **cache**.

An algorithm has **locality of reference** if, when it accesses a data item, the following accesses are likely to be nearby items.

Therefore:

Array + Algorithm with good locality of reference → significant speed advantage over a Linked List
In a **Doubly Linked List**, each node has a data item & **two pointers**:
- A pointer to the next node (null at the end).
- A pointer to the previous node (null at the beginning).

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 of the list.
With Doubly Linked Lists, we can eliminate asterisks.

* \( O(1) \) if no reallocate-and-copy.
  - Pre-allocate to ensure this.

**Find** faster with an array

**Rearrange** faster with a Linked List
More on Linked Lists

continued
Two Approaches to Implementing a Linked List

- A Linked List package to be used by others.
- An internal-use Linked List: part of a larger package, and not exposed to client code.

Q. How would these be different? In particular, what classes might we define in each case?

A1. In the first case, many classes: container, node, iterator.
A2. In the second case, a node class, but possibly nothing else.

I bring this up because you will probably never need to write the former, except perhaps as homework. But you may have occasion to write the latter.
We have already written an internal-use-style Singly Linked List, back in *A Little About Linked Lists*. But now we can improve it, by using smart pointers.

TO DO

- Update our Linked List to include the following. Follow the suggestions in the *Node-Based Structures/Smart Pointers* subtopics.
  - An owning smart pointer.
  - An insert-at-beginning operation.
  - A remove-at-beginning operation.
  - Exception-safety information.

*Done. See llnode2.h.*
*See use_list2.cpp for a program that uses this Linked List.*
More on Linked Lists
Implementation — Problem

Twice now, we have written a Linked List in which the node destructor recursively destroys the rest of the list. For a list that may be arbitrarily long, this is a bad idea! Why? The recursive node destructor has linear recursion depth. Stack overflow awaits.

The phrase “linear recursion depth” should strike terror into your heart.

<gasp> No! Not—LINEAR RECURSION DEPTH!
Anything but that!!
TO DO

- Rewrite our Linked List so that it no longer has a recursive destructor.

Done. See llnode2.h. See use_list2.cpp for a program that uses this Linked List.
Sequences in the C++ STL
The C++ STL includes six generic Sequence containers.

- **std::vector**
  - Smart array.
- **std::basic_string**
  - Much like `std::vector`, but aimed at character string operations.
  - `string` is `basic_string<char>`; other string-ish types are defined.
- **std::array**
  - Kinda-smart array. Not resizable. Size is part of the type.
  - Not the same as a C++ built-in array.
  - Data items are stored in the object.
  - A little faster than `vector`.
- **std::forward_list**
  - Singly Linked List.
- **std::list**
  - Doubly Linked List.
- **std::deque** (stands for Double-Ended QUEue; say “deck”)
  - Somewhat like `vector`, but slower. Fast insert/remove at both ends.

We will not say much more about `std::array` & `std::forward_list`.
Sequences in the C++ STL
std::deque [1/3]

We are familiar with arrays and Linked Lists. How is std::deque implemented? There are two big ideas behind it.

Big Idea #1

- vector uses an array with data stored at the beginning. This gives linear-time insert/remove at beginning, constant-time remove at end, and, if we are careful, amortized constant-time insert at end.

```
  0 1 2 3 4 5
```

- What if we store data in the middle? When we reallocate-and-copy, we move our data to the middle of the new array.

```
  0 1 2 3 4 5
```

- Result: amortized $\Theta(1)$ insert and $\Theta(1)$ remove at beginning & end.
Big Idea #2

- Doing reallocate-and-copy for a `vector` requires a copy/move call for every data item. For large, complex data items, this can be time-consuming.
- Instead, use an array of pointers to arrays. Only the secondary arrays hold data items. Reallocation-and-copy only deals with the array of pointers.
  - We still get most of the locality-of-reference advantages of an array.
  - We can do the `copy` portion of reallocate-and-copy using a raw-memory copy—no copy/move constructor calls.

![Diagram showing an array of pointers and arrays of data items. When space runs out, reallocate-and-copy this, but leave these alone.](image-url)
A `std::deque` implementation typically uses both ideas.

- 2 levels: array of pointers to arrays. Or maybe 3 levels.
- Reallocate-and-copy relocates to the middle of the new space.

Result: `std::deque` is an array-ish container.

- Iterators are random-access.
- Look-up by index is constant time—but slower than `vector`.
- Locality-of-reference advantages are *almost* as good as `vector`.
- Reallocate-and-copy is likely to be faster than `vector`.
- Insert/remove is fast at both beginning and end.
- Large, difficult-to-copy data items are handled more efficiently.
Sequences in the C++ STL

Common Features

vector, basic_string, deque, list:
- iterator, const_iterator
  - list has bidirectional iterators.
  - Others are random-access.
- iterator begin(), iterator end()
- reference front(), reference back()
  - Return reference to first/last item.
- iterator insert(iterator, item)
  - Insert before. Return iterator to new item.
- iterator erase(iterator)
  - Remove this item. Return iterator to following item—or end.
- push_back(item), pop_back()
  - Insert/remove at end.
- clear()
  - Remove all items.
- resize(newSize)
  - Set size.

Additionally:

d deque, list:
- push_front(item), pop_front()
  - Insert/remove at beginning.

vector, basic_string, deque:
- reference operator[](index)
  - Look-up by index.

vector, basic_string:
- reserve(newCapacity)
  - Set capacity to at least this.
  - Not the same as resize. Does not change the size.

And others ...
Now we compare the efficiency characteristics of STL generic Sequence containers.

In the model of computation used in the official description of C++ STL, the basic operations are value-type operations *only*. Things like pointer operations are not counted.

- So “constant time” in the Standard means that at most a constant number of *value-type* operations are performed.
- An algorithm that the Standard calls “constant time” may still perform a large number of pointer operations.

Our analyses will be based on the same model of computation that we have been using. When the efficiency characteristics specified in the C++ Standard appear to be different, we will note this.
Sequences in the C++ STL
Efficiency [2/2]

<table>
<thead>
<tr>
<th>Operation</th>
<th>vector, basic_string</th>
<th>deque</th>
<th>list</th>
</tr>
</thead>
<tbody>
<tr>
<td>Look-up by index</td>
<td>Constant</td>
<td>Constant</td>
<td>Linear</td>
</tr>
<tr>
<td>Search sorted</td>
<td>Logarithmic</td>
<td>Logarithmic</td>
<td>Linear</td>
</tr>
<tr>
<td>Insert @ given pos</td>
<td>Linear</td>
<td>Linear</td>
<td>Constant</td>
</tr>
<tr>
<td>Remove @ given pos</td>
<td>Linear</td>
<td>Linear</td>
<td>Constant</td>
</tr>
<tr>
<td>Insert @ end</td>
<td>Linear/Amortized constant*</td>
<td>Linear/Amortized constant**</td>
<td>Constant</td>
</tr>
<tr>
<td>Remove @ end</td>
<td>Constant</td>
<td>Constant</td>
<td>Constant</td>
</tr>
<tr>
<td>Insert @ beginning</td>
<td>Linear</td>
<td>Linear/Amortized constant**</td>
<td>Constant</td>
</tr>
<tr>
<td>Remove @ beginning</td>
<td>Linear</td>
<td>Constant</td>
<td>Constant</td>
</tr>
</tbody>
</table>

The way `vector` acts at the end is the way `deque` acts at beginning and end.

*Θ(1) if sufficient memory has already been allocated. We can pre-allocate.

**Only a constant number of value-type operations are required. The C++ Standard says these are constant-time.

All four have Θ(n) traverse & search-unsorted and Θ(n log n) sort.
This completes our discussion of Sequences *in full generality*.

Next, we look at two *restricted* versions of Sequences, that is, ADTs that are much like Sequence, but with fewer operations:

- Stack.
- Queue.

For each of these, we look at:

- What it is.
- Implementation.
- Availability in the C++ STL.
- Applications.