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

CS 311 Data Structures and Algorithms

Lecture Slides

Monday, October 28, 2024

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

ggchappell@alaska.edu

Some material contributed by Chris Hartman

Unit Overview
Data Handling & Sequences

Topics

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

Smart Arrays

Linked Lists

Review

Where Are We? — The Big Challenge

Our problem for most of the rest of the semester:

- Store: A collection of data items, all of the same type.
- Things we need to be able to do:
  - Access items [single item: retrieve/find, all items: traverse].
  - Add new item [insert].
  - Eliminate existing item [delete].
- Time & space efficiency are desirable.

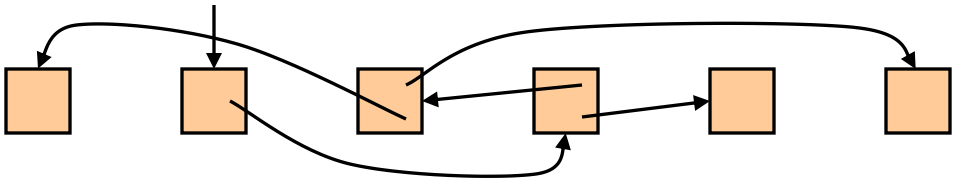
A solution to this problem is a **container**.

In a **generic container**, client code can specify the value type.

Review
Node-Based Structures — Introduction

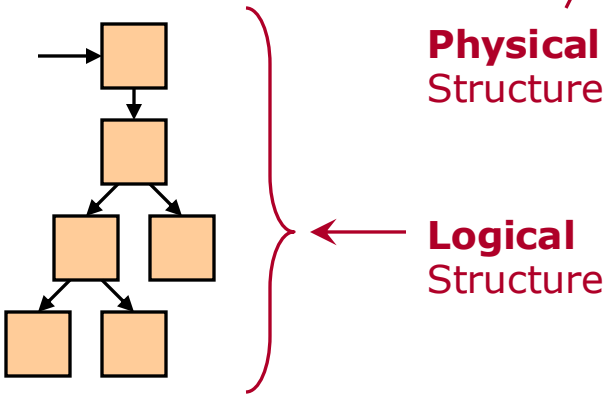
Many data structures are built out of *nodes*.

- 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. But they do show the logical structure correctly.



Node-Based Structures — Smart Pointers [1/4]

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.

Node-Based Structures — Smart Pointers [2/4]

Create using `make_unique`/`make_shared`. Then we do neither the `new` nor the `delete` directly.

```
auto unp = make_unique<Foo>(5, "xy", 3.2);
```

Dereference just like a pointer.

```
Foo x = *unp;
unp->bar();
```

To obtain a regular pointer to the referenced object, call member function `get`.

```
Foo * p = unp.get()
```

Node-Based Structures — Smart Pointers [3/4]

A **null** smart pointer is one that does not point to anything. Check for this by treating a smart pointer like a `bool`.

```
if (unp)  // Non-null?
    unp->bar();
```

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.

Node-Based Structures — Smart Pointers [4/4]

Suggestions

- When you want a smart pointer, start by using `unique_ptr`.
- To transfer ownership, pass/return by value.
- Otherwise, pass by reference or reference-to-const, OR call `get` and pass a regular pointer.
- 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`

Review
More on Linked Lists [1/2]

We often **find** faster with an array.

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

We often **rearrange** faster with a Linked List.

*For Singly Linked Lists, insert/remove just after the given position.

***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.)

More on Linked Lists [2/2]

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

More on Linked Lists

continued

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.

TO DO

- Revise the smart-pointer-based Linked List so that it no longer has a recursive destructor.

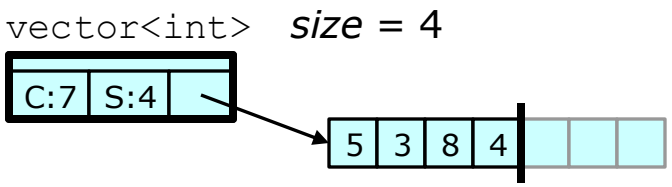
Done. See `llnode2.hpp`*.*
See `use_list2.cpp` *for a program that uses this Linked List.*

Sequences in the C++ STL

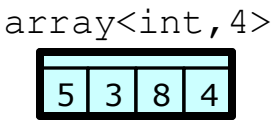
Sequences in the C++ STL
Generic Sequence Containers

The C++ STL includes six generic Sequence containers.

- `std::vector`
  - Smart resizable array.



- `std::basic_string`
  - Much like `vector`, but aimed at character string operations.
  - `string` is `basic_string<char>`; other string-ish types are defined.
- `std::array`
  - A-little-bit-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.

We will not say much more about `std::array` & `std::forward_list`.

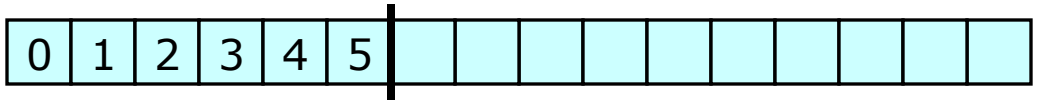
- `std::list`
  - Doubly Linked List.
- `std::deque` (stands for Double-Ended QUEue; say "deck")
  - Like `vector`, but a bit slower. Fast insert/remove at both ends.

Sequences in the C++ STL
std::deque [1/3]

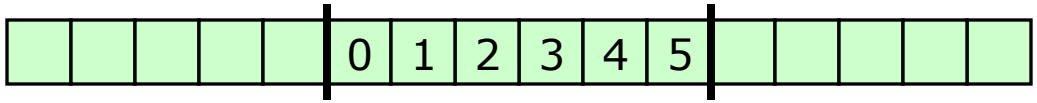
We are familiar with arrays and Linked Lists. What is `std::deque`? There are two big ideas behind it.

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.



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

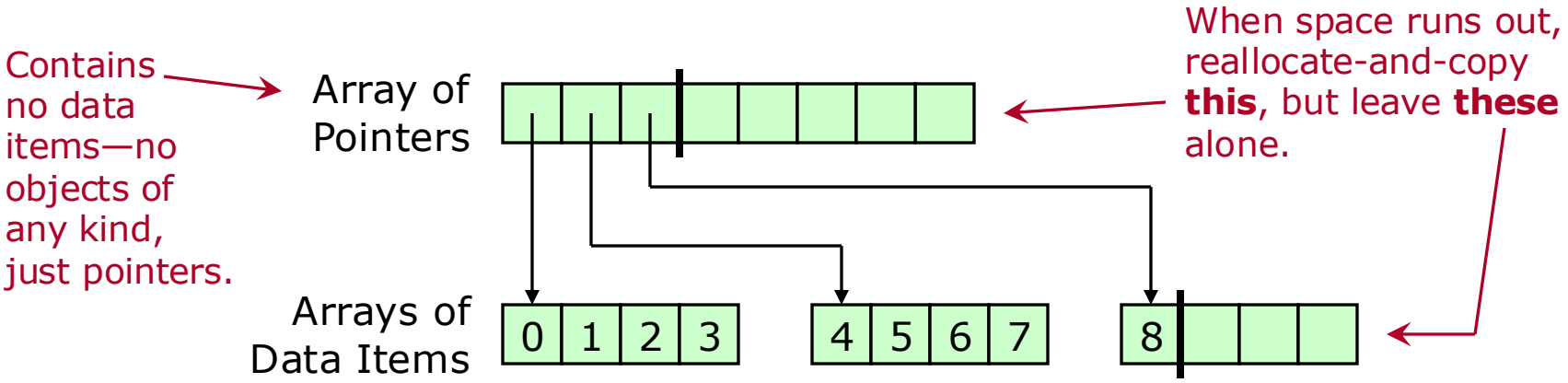


- Result: amortized Θ(1) insert and Θ(1) remove at beginning & end.

Sequences in the C++ STL
std::deque [2/3]

Big Idea #2

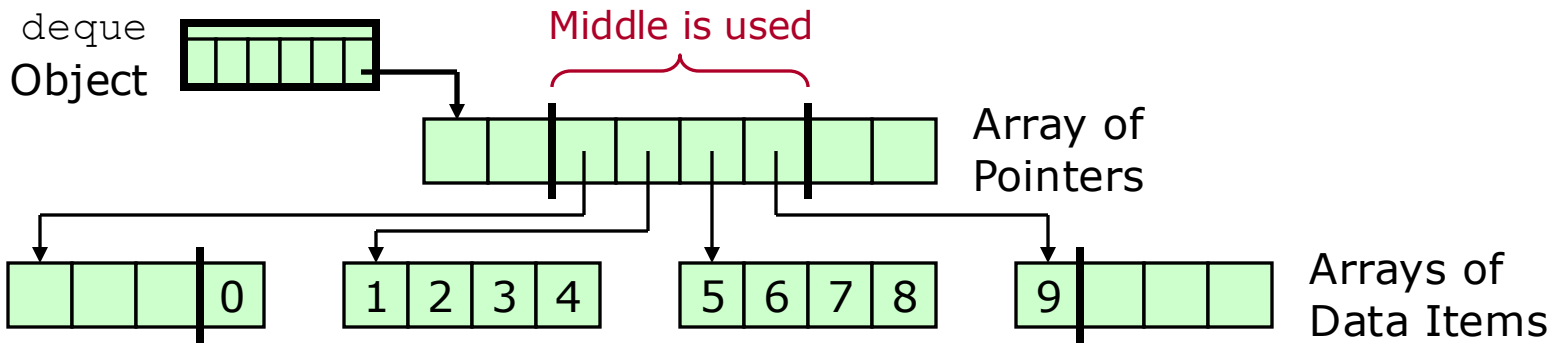
- Doing reallocate-and-copy for a `vector` requires a copy/move call for every data item. For large, complex data items, this can be time-consuming.
- Instead, use an array of pointers to arrays. Only the secondary arrays hold data items. Reallocate-and-copy only deals with the array of pointers.
  - 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 ctor calls.



Sequences in the C++ STL
std::deque [3/3]

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—(amortized) Θ(1)—at beginning and end.
- Large, difficult-to-copy data items are handled more efficiently.

Sequences in the C++ STL
Efficiency [1/2]

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]

	`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 @ end	Linear/ Amortized constant*	Linear/ Amortized constant**	Constant
Remove @ end	Constant	Constant	Constant
Insert @ beginning	Linear	Linear/ Amortized constant**	Constant
Remove @ beginning	Linear	Constant	Constant

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.