Stacks (cont.)
Applications of Stacks

CS 311 Data Structures and Algorithms
Lecture Slides
Wednesday, April 3, 2013

Chris Hartman
Department of Computer Science
University of Alaska Fairbanks
cmhartman@alaska.edu
Based on material by Glenn G. Chappell

Unit Overview
Handling Data & Sequences

Major Topics

- ✓ Data abstraction
- ✓ Introduction to Sequences
- ✓ Smart arrays
  - ✓ Array interface
  - ✓ Basic array implementation
  - ✓ Exception safety
  - ✓ Allocation & efficiency
  - ✓ Generic containers
- ✓ Linked Lists
  - ✓ Node-based structures
  - ✓ More on Linked Lists
- ✓ Sequences in the C++ STL
- Stacks
- Queues

Review
Sequences in the C++ STL [1/3]

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

- Class template **`std::vector`**.
  - A "smart array".
- 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.
- 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.

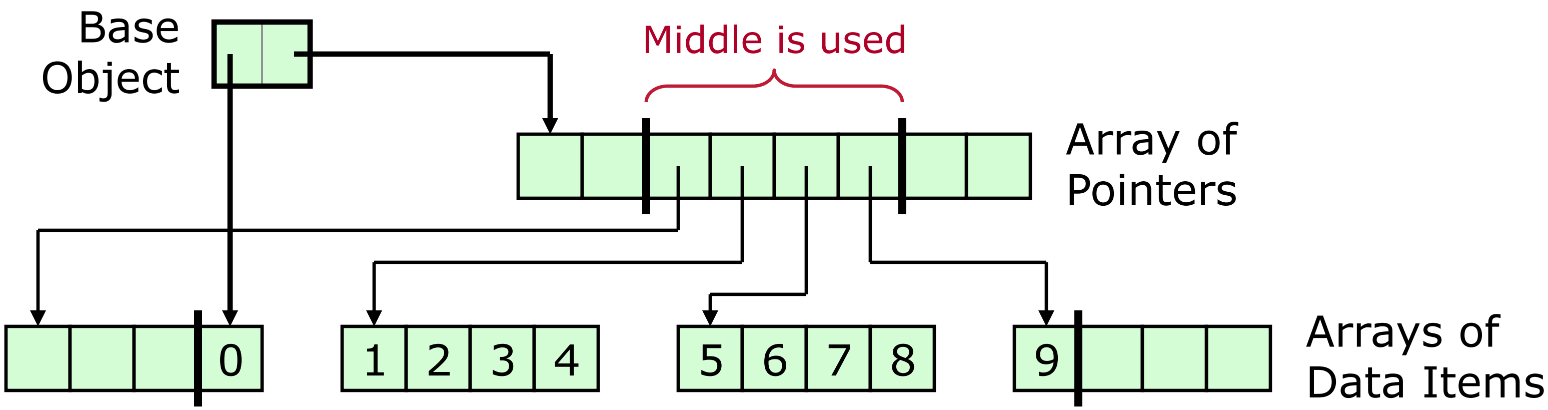
Review
Sequences in the C++ STL [2/3]

STL's **`std::deque`** is a random-access container optimized for:

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

A typical implementation:

- Uses an array of pointers to arrays.
- Has storage that may not be filled all the way to the beginning or the end, with a reallocate-and-copy that moves the data to the middle of the new array of pointers.



Review
Sequences in the C++ STL [3/3]

	`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 the 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.

Review
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.

Review
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.

Review
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`**.

Stacks
In the C++ STL — Review

The STL has a Stack: **`std::stack`**, in **`<stack>`**.

- STL documentation does not call **`std::stack`** a "container", but rather a "container adapter".
- This is because **`std::stack`** is explicitly a **wrapper** around some other container.

You get to pick what that container is.

`std::stack<T,` ***container*** **`<T> >`**

- "**`T`**" is the value type.
- "*container*" can be **`std::vector`**, **`std::deque`**, or **`std::list`**.
- "*container*<**`T`**>" can be **any** standard-conforming container with member functions **`back`**, **`push_back`**, **`pop_back`**, **`empty`**, **`size`**, along with comparison operators (==, <, etc.).

container defaults to **`std::deque`**.

```
std::stack<T>  // = std::stack<T, std::deque<T> >
```

Stacks
In the C++ STL — Operations Review

`std::stack` implements the various ADT operations as follows.

ADT Operation	What to Call
Push	Member function **`push`**
Pop	Member function **`pop`**
GetTop	Member function **`top`**
IsEmpty	Member function **`empty`**
Create	Default constructor
Destroy	Destructor
Copy	Copy constructor, copy assignment

`std::stack` also has member function size, which returns the size of the Stack, and the various comparison operators (**==**, <, etc.).

Stacks
In the C++ STL — Efficiency

Is the default container, **`std::deque`**, a good idea?

Using a **`std::deque`** is, **on the average**, faster than using a **`std::list`**.

- A **`std::deque`** has much less memory management to do, and it does no more value-type operations than **`std::list`**.
- Thus, a **`deque`**'s amortized constant time for insert-at-end should result in a smaller constant than a **`list`**'s constant time.

However, **worst-case** performance of **`std::deque`** may be worse.

- Linear time for insert-at-end, if **all** operations are counted, vs. constant time for **`std::list`**.

This does not mean that **`deque`**s are "bad". It does mean that you should use them with care.

The typical vs. worst-case performance tradeoff is not uncommon.

- This *used to be* an issue with Quicksort vs. Merge Sort.
- We will see it again when we cover Hash Tables.

Stacks
In the C++ STL — Comparisons

We can **compare** two **`std::stack<T>`** objects, using "**==**", "**<**", etc. Why are these operations available?

Hint: When do we use an ordering, even though we might not care what order things are in?

The two operators: "**==**" and "**<**" are those required by various STL types and algorithms.

- "**<**" lets us (for example) make a **`std::set`** of **`stack`**s.
- "**==**" lets us (for example) do **`std::find`** in a **`vector`** of **`stack`**s.
- More generally, these two operators make **`std::stack`** usable with just about any STL container or algorithm.
- All STL containers/adapters (except **`std::priority_queue`**, for some reason) have "**==**", "**<**", and the other comparisons defined.

Stacks — In the C++ STL, Applications

The STL has a Stack: **`std::stack`**, in **`<stack>`**.

- This is a "container adapter". The data is stored in some other container.

`vector`, **`deque`**, or **`list`**

```
std::stack<T, container<T> >
std::stack<T>  // = std::stack<T, std::deque<T> >
```

We look at two applications of Stacks.

- Expression evaluation.
  - A Stack can be used to do a very simple evaluator for **Reverse Polish Notation**.
    - Normal (infix): (2 + 3) * (7 – 5). RPN (postfix): 2 3 + 7 5 – *.
- Eliminating recursion.
  - We'll use the "brute force method" to eliminate the recursion in **`fibo1.cpp`**.
    Result: long, slow, but correct.

*See **`rpn.cpp`**, on the web page.*

Stacks
Applications — Expressions [1/3]

One important application of Stacks is **parsing**: determining the structure of input.

- Parsing a source file is one step in compilation.
- It is also used in expression evaluation.

Full-scale parsing is beyond the scope of this class.

- However, we can do some very simple expression evaluation.

We will use a Stack to write an expression evaluator for “Reverse Polish Notation”.

Reverse Polish Notation (RPN) is a way of writing mathematical expressions so that operators come after the numbers they operate on.

- Normal (**infix**): "`1 + 2`". RPN (**postfix**): "`1 2 +`".
- We can operate on expressions as well:
  - "`(2 – 3) * 7`" becomes "`2 3 – 7 *`".
  - "`2 – (3 * 7)`" becomes "`2 3 7 * –`".
  - "`(2 – 3) * (7 + 5)`" becomes "`2 3 – 7 5 + *`".
- Notice that RPN never needs parentheses!

How to evaluate:

- Use a Stack, which holds numbers.
- When you see a number in the input, push it on the Stack.
- When you see a (binary) operator in the input, pop two values, apply the operator to them, and push the result.
  - Operators of other **arities** can be handled similarly.
- When you are done, the result is the top value on the Stack.

Stacks
Applications — Expressions [3/3]

TO DO

- Implement a simple RPN evaluator.

Stacks
Applications — Eliminating Recursion: Refresher [1/2]

From the "Eliminating Recursion" slides:

Fact. Every recursive function can be rewritten as an iterative function that uses essentially the same algorithm.

- Think: How does the system help you do recursion?
  - It provides a **Stack**, used to hold return addresses for function calls, and values of automatic local variables.
- We can implement such a Stack ourselves. We need to be able to store:
  - Values of automatic local variables, including parameters.
  - The return value (if any).
  - Some indication of where we have been in the function.
- Thus, we can eliminate recursion by mimicking the system's method of handling recursive calls using Stack frames.

Stacks
Applications — Eliminating Recursion: Refresher [2/2]

“Brute-force” method

To rewrite **any** recursive function in iterative form:

- Declare an appropriate Stack.
  - A Stack item holds all automatic variables, an indication of what location to return to, and the return value (if any).
- Replace each automatic variable with its field in the top item of the Stack.
  - Set these up at the beginning of the function.
- Put a loop around the *rest* of the function body: “**`while (true) { … }`**”.
- Replace each recursive call with:
  - Push an object with parameter values and current execution location on the Stack.
  - Restart the loop (“**`continue`**”).
  - A label marking the current location.
  - Pop the stack, using the return value (if any) appropriately.
- Replace each “**`return`**” with:
  - If the “return address” is the outside world, really **`return`**.
  - Otherwise, set up the return value, and skip to the appropriate label (“**`goto`**”?).

This method is primarily of theoretical interest.

- *Thinking* about the problem often gives better solutions than this.
- We will look at this method further when we study **Stacks**. **NOW**