

Stacks, cont'd

CS 311 Data Structures and Algorithms

Lecture Slides

Wednesday, April 9, 2008

Glenn G. Chappell

Department of Computer Science
University of Alaska Fairbanks

CHAPPELLG@member.ams.org

© 2005–2008 Glenn G. Chappell

Unit Overview

Sequences and Their Implementations

Major Topics

- ✓ • Data Abstraction
- ✓ • Sequence Data
- ✓ • Interfaces to Data
- ✓ • Data Structure Implementation
- ✓ • Exception Safety
- ✓ • Allocation & Efficiency
- ✓ • Generic Containers
- ✓ • Node-Based Structures
- ✓ • Linked Lists
- ✓ • Sequences in Practice

Review

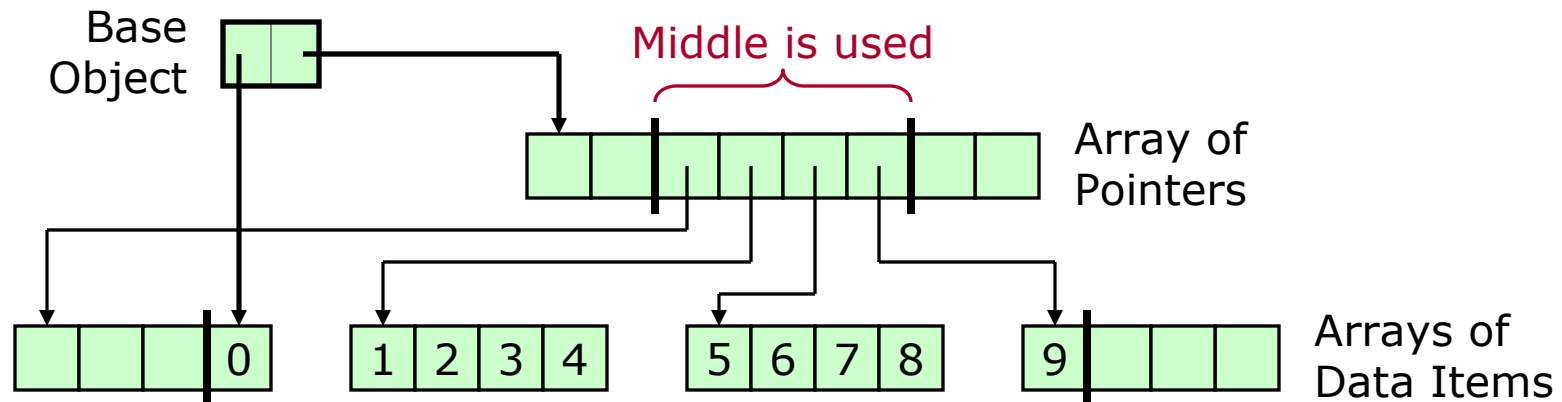
Sequences in Practice — STL Sequence Types [1/2]

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 arrays 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 Practice — STL Sequence Types [2/2]

	<code>vector</code> , <code>basic_string</code>	<code>deque</code>	<code>list</code>
Look-up by index	Constant	Constant	Linear
Binary Search	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 Sequential Search, $O(1)$ swap, and $O(n \log n)$ sort.

Unit Overview

Stacks & Queues

Major Topics

- Stacks
 - ✓ ▪ What a Stack is.
 - ✓ ▪ The ADT Stack interface.
 - ✓ ▪ Implementing a Stack.
 - Stacks in the C++ STL.
 - Applications of Stacks.
- Queues

Review Stacks

A Stack is:

- A kind of container.
- A Last-In-First-Out (LIFO) structure.
- A restricted version of a Sequence.

Conceptually, a Stack carries out the idea of **top-down design**.

Three primary operations:

- **push**
- **pop**
- **getTop**

A Stack can be implemented simply as a wrapper around some existing Sequence type.

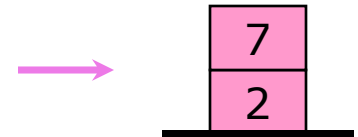
1. Start:
Empty Stack.



2. Push 2.



3. Push 7.



4. Pop.



5. Pop.
Empty again.



6. Push 5.



Stacks, cont'd

`std::stack` — Introduction

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.

- You say “`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`.
 - You can say just “`std::stack<T>`” to get “`std::stack<T, std::deque<T> >`”.

Stacks, cont'd

`std::stack` — 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, cont'd

`std::stack` — 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 stacks.
- “==” lets us (for example) do `std::find` in a vector of stacks.
- 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, cont'd

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”.

Stacks, cont'd

Applications — Expressions [2/3]

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 + 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.
- When you are done, the result is the top value on the Stack.

Stacks, cont'd

Applications — Expressions [3/3]

TO DO

- Implement a simple RPN evaluator.

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

Stacks, cont'd

Applications — Eliminating Recursion [1/8]

Recall, from the "Recursion vs. Iteration" slides (February 20):

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, cont'd

Applications — Eliminating Recursion [2/8]

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

“Brute-force”
method

- 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**. ...

Stacks, cont'd

Applications — Eliminating Recursion [2/8]

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

“Brute-force”
method

- 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**

Stacks, cont'd

Applications — Eliminating Recursion [3/8]

Here is function `fibonacci` from `fibonacci.cpp`.

```
int fibonacci(int n)
{
    // BASE CASE
    if (n <= 1)
        return n;

    // RECURSIVE CASE
    return fibonacci(n-2) + fibonacci(n-1);
}
```

Let's use the "brute force" recursion elimination procedure to produce a non-recursive version.

- I have already written this. In the following slides we examine the code.

*See `fibonacci5.cpp`,
on the web page.*

Stacks, cont'd

Applications — Eliminating Recursion [4/8]

When we eliminate recursion, all local values will be stored on our Stack. For convenience I rewrote function `fibonacci` so that every computed value has a name.

```
int fibonacci(int n)
{
    int v1, v2;

    // Base Case
    if (n <= 1)
        return n;

    // Recursive call #1
    v1 = fibonacci(n-2);

    // Recursive call #2
    v2 = fibonacci(n-1);

    // Return the result
    return v1 + v2;
}
```

Stacks, cont'd

Applications — Eliminating Recursion [5/8]

We need a Stack. It holds:

- All variables and necessary temporary values.
 - These are `n`, `v1`, `v2`, and the return value.
- Some indication of where to return.
 - Three possibilities: the outside world, recursive call #1, recursive call #2.

We can use a `struct` for our Stack frame:

```
struct FiboStackFrame {
    int n;           // Parameter
    int v1;         // Result of recursive call #1
    int v2;         // Result of recursive call #2
    int returnValue; // Value to return
    int returnAddr; // Return address:
                    //      0: outside world
                    //      1: recursive call #1
                    //      2: recursive call #2
};
```

Stacks, cont'd

Applications — Eliminating Recursion [6/8]

We need to create our Stack when we enter function `fibonacci`.

```
std::stack<FibonacciStackFrame> s;
```

Then we can store our local variables there.

- So, for example, "`n`" becomes "`s.top().n`".

We need one variable to hold values during Stack operations.

- All such values will be `ints`.

```
int tmp;
```

After setting up the initial values, we enter a big `while` loop.

Stacks, cont'd

Applications — Eliminating Recursion [7/8]

To make a recursive call:

- We set up the Stack and restart the loop (`continue`).
- We must enable the function to return here. Use a **label**, and return to it with `goto`.

For example, here is `"v1 = fibo(n-2);"`:

```
    tmp = s.top().n - 2;
    s.push(FiboStackFrame()); // Make new stack frame
    s.top().n = tmp;          // Set parameter
    s.top().returnAddr = 1;  // Set return address
                              // (recursive call #1)
    continue;                // Do "recursive call"
label1:                       // Place to return to
    tmp = s.top().returnValue;
    s.pop();
    s.top().v1 = tmp;         // Put returned value in v1
```

Stacks, cont'd

Applications — Eliminating Recursion [8/8]

To “return”:

- If we were called by the outside world, then really **return**.
- Otherwise, set up the return value, and **goto** the appropriate location.
 - Note: As on the previous slide, I pop the Stack *after* returning.

For example, here is “**return n;**”:

```
s.top().returnValue = s.top().n;
if (s.top().returnAddr == 1)           // Back to recursive call #1
    goto label1;
else if (s.top().returnAddr == 2)     // Back to recursive call #2
    goto label2;
else                                   // Back to outside world
{
    tmp = s.top().returnValue;
    s.pop();
    return tmp;
}
```