Stacks

CS 311 Data Structures and Algorithms
Lecture Slides
Friday, November 6, 2009

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

© 2005–2009 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.

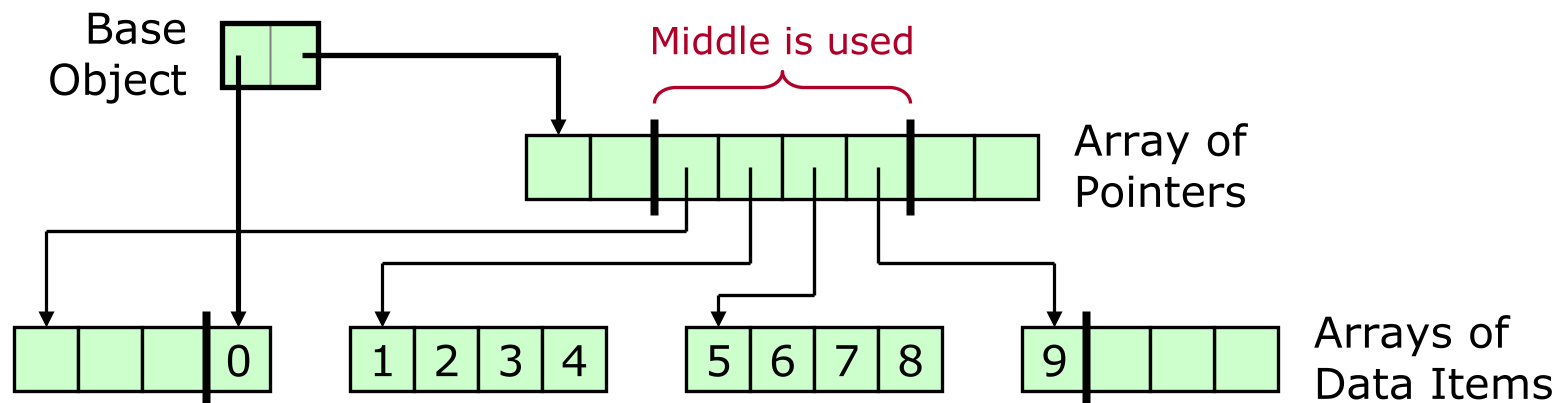
Review
Sequences in the C++ STL [2/3]

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.

Unit Overview
What is Next

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.
  - Covered in chapter 6 in the text.
- Queue.
  - Covered in Chapter 7 in the text.

For each of these, we look at:

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

Stacks
What a Stack Is — Idea

Our third ADT is **Stack**. This is another container ADT; that is, it holds a number of values, all the same type.

A *Stack* is a Last-In-First-Out (LIFO) structure.

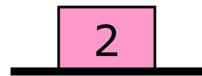
- What we do with a Stack:
  - **Push**: add a new value.
  - **Pop**: Remove a value.
- The last item added is the first removed.
  - Think of a stack of plates.

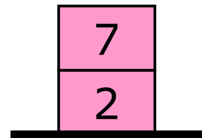
Thus, a Stack is a restricted version of a Sequence.

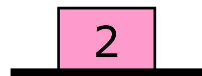
- We can only insert/remove at one end.
- We cannot iterate through the contents.

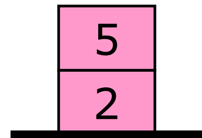
Stacks
What a Stack Is — Illustration

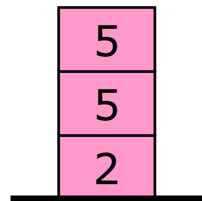
1. Start:
   an empty Stack.

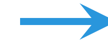
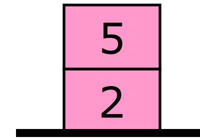
2. Push 2. 

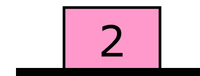
3. Push 7. 

4. Pop. 

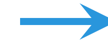
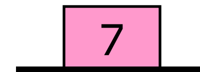
5. Push 5. 

6. Push 5. 

7. Pop.  

8. Pop. 

9. Pop.
   Stack is empty again.

10. Push 7.  

11. Etc. …

Stacks
What a Stack Is — Top-Down Design

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

- In TDD, we want to perform some large task.
- We think of various subtasks as “black boxes”. We call the code to perform them without worrying about its internal details.
  - Even if that code is the same code we are already executing (as is the case in a recursive function).
- When we perform a subtask, we can push our current state on a Stack. When the subtask is finished, pop the state off.
  - The Stack ends up looking exactly as it did before the subtask began.

In practice, nearly every use of a Stack has this idea behind it.

Stacks
What a Stack Is — ADT

When we looked at general Sequences, I defined my own ADT (“Sequence”).

But there is essentially only one good definition of a Stack:

- Data
  - A Sequence of data items. One end is the “top”.
- Operations
  - **getTop**. Look at top item.
  - **push**. Add an item.
  - **pop**. Remove top item.
  - To avoid errors we need information about empty state (or size):
    - **isEmpty**. Returns true if Stack is empty.
  - Then, of course, we need the standard stuff:
    - **create**.
    - **destroy**.
    - I will add the usual **copy** operations.

Stacks
Implementation — Ideas

One can write a Stack "from scratch".

However, in practice, a Stack is often just a wrapper around some Sequence.

- That is, around a Sequence container.
- Which ones would work well?

Once the Sequence is written, making a Stack is easy.

- Write a class with just one data member: the Sequence.
- **All** of the Stack operations are just wrappers around existing Sequence operations.
- Watch out for errors & exception safety!

Stacks
Implementation — Interface Problems?

Why should “pop” **not** return a value (the old top value)?

- We cannot return by reference, since there is nothing left to make a reference to.
- Returning by value may produce an exception in the value type’s copy constructor.
- In this case, we have already left the function. The value to be returned is lost. We also cannot offer the Strong Guarantee.
- Remember: In general, a non-const member function should not return an object by value.

The text’s interface has a “pop” that **does** tell the caller what the top value was [`void pop(value_type & topItem)`]. Why is this **not** a problem?

- The text’s version assigns a reference parameter to the old top value.
- This can be done before the pop. If an exception occurs, we can catch it and avoid the pop. No information is lost.
- Since the Stack is unchanged in the event of an exception, we can also offer the Strong Guarantee.

Stacks
In the C++ STL — 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.

```
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

`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
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.

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

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

From the "Eliminating Iteration" 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**

Stacks
Applications — Eliminating Recursion: Example [1/6]

Here is function **fibo** from **fibo1.cpp**.

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

    // RECURSIVE CASE
    return fibo(n-2) + fibo(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 **fibo6.cpp**, *on the web page.*

Stacks
Applications — Eliminating Recursion: Example [2/6]

When we eliminate recursion, all local values will be stored on our Stack. For convenience I rewrote function **`fibo`** so that computed values have names.

```
bignum fibo(int n)
{
    bignum v1, v2;

    // BASE CASE
    if (n <= 1)
        return bignum(n);

    // RECURSIVE CASE
    v1 = fibo(n-2);  // Recursive call #1
    v2 = fibo(n-1);  // Recursive call #2
    return v1 + v2;  // Return the result
}
```

Stacks
Applications — Eliminating Recursion: Example [3/6]

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

Stacks
Applications — Eliminating Recursion: Example [4/6]

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

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

Then we can store our local variables there.

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

We need variables to hold values during Stack operations.

- Some will be **`int`**s and some will be **`bignum`**s.

```
int tmpi;
bignum tmpb;
```

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

Stacks
Applications — Eliminating Recursion: Example [5/6]

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);`**":

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

Stacks
Applications — Eliminating Recursion: Example [6/6]

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 bignum(n);`**":

```
s.top().returnValue = bignum(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
{
    tmpb = s.top().returnValue;
    s.pop();
    return tmpb;
}
```