Stacks

CS 311 Data Structures and Algorithms
Lecture Slides
Wednesday, October 28, 2020

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
ggchappell@alaska.edu

Some material contributed by Chris Hartman

Review

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.
- 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
- ✓ Sequences in the C++ STL
- Stacks
- Queues

Smart Arrays

Linked Lists

Review
More on Linked Lists — Implementation

Two Approaches to Implementing a Linked List

- A Linked List package to be used by others. Container class, node class, iterator classes.
- An internal-use Linked List: part of a larger package, and not exposed to client code. Probably just a node class.

We updated our internal-use-style Linked List to use smart pointers (`std::unique_ptr`).

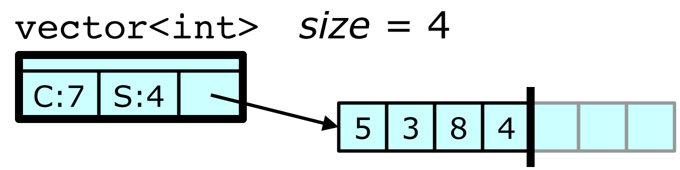
The recursive destructor was problematic, since it had **linear recursion depth**. Destroying a long Linked List would result in stack overflow. To fix this, we wrote an iterative destructor.

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

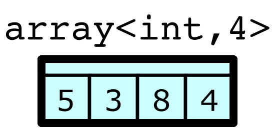
Review
Sequences in the C++ STL [1/3]

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`.



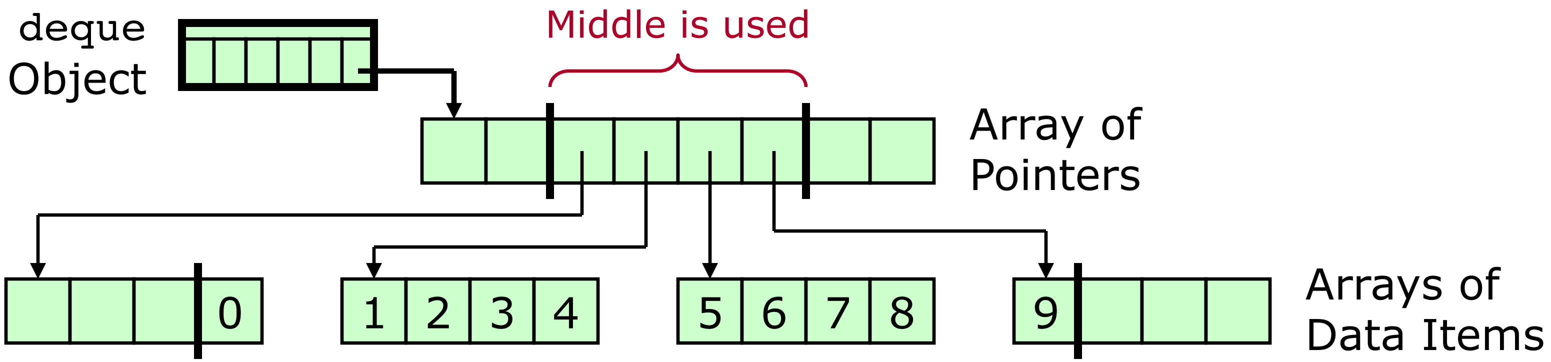
- `std::forward_list`
  - Singly Linked List.
- `std::list`
  - Doubly Linked List.
- `std::deque` (stands for Double-Ended QUEue; say “deck”)
  - Somewhat like `vector`, but slower. Fast insert/remove at both ends.

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

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

`std::deque` is a random-access container optimized for:

- Resizing, including 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. Reallocate-and-copy 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 @ 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.

*$\Theta(1)$ 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 $\Theta(n)$ traverse & search-unsorted and $\Theta(n \log n)$ 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.
- Queue.

For each of these, we look at:

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

Stacks

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* allows **Last-In-First-Out** (**LIFO**) access to data.

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

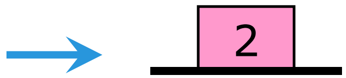
Thus, a Stack is a restricted version of a Sequence.

- We can only insert/remove at one end.
- We cannot iterate through the contents or access any item other than the top item.
- Or *maybe* we can, but if so, then we are using a structure that provides functionality beyond that required by a Stack.

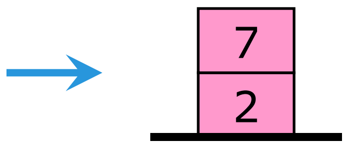
Stacks
What a Stack Is — Illustration

1. Start:
   an empty Stack.

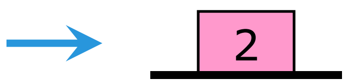
2. Push 2.



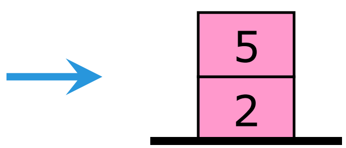
3. Push 7.



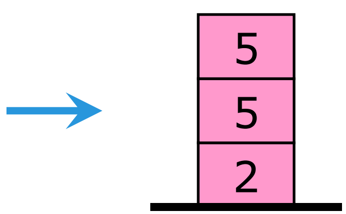
4. Pop.



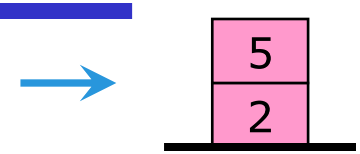
5. Push 5.



6. Push 5.



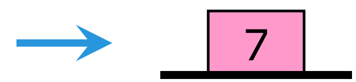
7. Pop.



8. Pop.

9. Pop.
   Stack is empty again.

10. Push 7.



Stacks
What a Stack Is — Top-Down Design

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

- Say we want to perform some large task involving a number of subtasks—each of which may involve sub-subtasks, etc.
- When it is time to perform a subtask, we push our current state on a Stack and then do the subtask. When the subtask is finished, we restore the state by popping it off the Stack. The Stack ends up looking exactly as it did before the subtask began; then we continue performing the main task.
- The subtask may also use the top of the Stack as a place to store data. It can push a new top item when it begins and pop that item when it ends.

The prototypical application of a Stack is the function call stack being using to store return addresses and local variables. But there are many other ways to use a Stack; most involve the above idea.

ADT **Stack**

- Data
  - A finite sequence of data items, all the same type. One end is the **top**.
- Operations
  - **getTop**. Look at top item.
  - **push**. Add a new item at the top.
  - **pop**. Remove top item.
  - To avoid errors we need information about the number of items:
    - **isEmpty**. Return true if Stack is empty.
    - **size**.
  - Then, of course, we need the standard stuff:
    - **create**.
    - **destroy**.
    - **copy**.

Three primary operations
(retrieve, insert, delete)

Stacks
Implementation — Sequence Wrapper

One *can* implement a Stack from the ground up.

However, in practice, a Stack is usually a wrapper around some Sequence container.

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.

Stacks
Implementation — Interface Trickiness

As we have mentioned, it can be a bad idea to have the *pop* operation return the top value that is removed. Why?

- 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.

Stacks
In the C++ STL — Introduction

The STL has a Stack: `std::stack` (`<stack>`).

- The Standard calls `stack` a **container adapter**, not a container. That is, `stack` is 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*`<T>` can be any standard-conforming container with member functions `back`, `push_back`, `pop_back`, `empty`, and `size`, along with comparison operators (`==`, `<`, etc.).
- In particular, *container* can be `vector`, `deque`, or `list`.

container defaults to `std::deque`.

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

Stacks
In the C++ STL — Introduction

The STL has a Stack: `std::stack` (`<stack>`).

The Standard calls `stack` a **container adapter**, not a container.
That is, `stack` is 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*`<T>` can be any standard-conforming container with member functions `back`, `push_back`, `pop_back`, `empty`, and `size`, along with comparison operators (==, <, etc.).
- In particular, *container* can be `vector`, `deque`, or `list`.

container defaults to `std::deque`.

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

Stacks
In the C++ STL — Operations

The `std::stack` interface for the various ADT operations:

ADT Operation	Implementation
push	Member function `push`
pop	Member function `pop`
getTop	Member function `top`
isEmpty	Member function `empty`
size	Member function `size`
create	Default constructor
destroy	Destructor
copy	Copy/move operations

`std::stack` also has:

- Member function `swap`.
- The various comparison operators (==, <, etc.).

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 exactly what order things are in?

> There are things on this slide that we have not covered yet: Sets, Hash Tables, Priority Queues. We will get to these!

Comparisons are used in searching and, generally, in making things *easy to find*.

- “<” lets us (for example) do Binary Search on a `std::vector` of `stack`s, or make a `std::set` of `stack`s.
- “==” lets us (for example) do `std::find` in a `vector` of `stack`s.

Most STL containers & container adapters have all the comparison operators defined, just like `std::stack`.

- Exceptions: the Hash Tables (`std::unordered_map`, `std::unordered_set`, etc.) and `std::priority_queue`.

Applications — Expression Evaluation [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.

Recall:

An **expression** is something that has a value.

To **evaluate** an expression is to compute its value.

In-depth coverage of parsing is beyond the scope of this class. (See CS 331.)

However, we can do simple expression evaluation. We will use a Stack in an expression evaluator for *Reverse Polish Notation*.

Reverse Polish Notation (**RPN**) is a way of writing expressions so that an operator comes after its operands.

- Normal (**infix**): "`1 + 2`". RPN (**postfix**): "`1 2 +`".
- We can translate larger expressions as well:
  - "`(5 - 2) * 7`" becomes "`5 2 - 7 *`".
  - "`5 - (2 * 7)`" becomes "`5 2 7 * -`".
  - "`(5 - 2) * (7 + 5)`" becomes "`5 2 - 7 5 + *`".
- RPN never needs parentheses!

An odd term (IMHO) that goes back to early 20th-century logician Jan Łukasiewicz, who happened to be Polish.

A **binary** operator is one with two operands.

The **arity** of an operator is the number of operands it has.

How to evaluate:

- Use a Stack, which holds numbers.
- When you see a number in the input, push it.
- 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 done, the result is the top value on the Stack.

Try it!

Stacks
Applications — Expression Evaluation [3/3]

TO DO

- Implement a simple evaluator for arithmetic expressions in RPN.

Done. See `rpn_evaluate.cpp`*.*

Applications — Eliminating Recursion: Refresher [1/2]

From the Eliminating Recursion slides:

While it is a useful algorithm-design tool, recursion can have serious drawbacks. Thus, it can sometimes be helpful to **eliminate recursion**—that is, to convert recursion to iteration.

Fact. *Every* recursive function can be rewritten as a non-recursive function that uses essentially the same algorithm.

This is true because we can simulate the call stack ourselves. We can eliminate recursion by mimicking the system’s method of handling recursive calls using stack frames.

We *can* always eliminate recursion, but that does not mean that eliminating it is always a good idea.

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

We discuss this method further when we cover *Stacks*, later in the semester.

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 Stack item.
  - Set these up at the beginning of the function.
- Put a loop around the *rest* of the function: `while (true) { … }`
- Replace each recursive call with:
  - Push an object with parameter values and current execution location.
  - Restart the loop (`continue`).
  - A label marking the current location.
  - Pop the stack. Make use of the return value (if any).
- Replace each `return` with:
  - If the "return address" is the outside world, then really `return`.
  - Otherwise, set the return value, and skip to the proper label (`goto` ?).

NOW

This method is rarely used. *Thinking* often gets better results.

Stacks
Applications — Eliminating Recursion: Example [1/7]

Here is function `fibo` from `fibo_first.cpp` (the slow version).

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

    // RECURSIVE CASE
    // Invariant: n >= 2
    return fibo(n-2) + fibo(n-1);
}
```

I used the brute-force recursion-elimination procedure on this code.

Let’s examine the result.

See `fibo_bf_elim.cpp`*.*

Applications — Eliminating Recursion: Example [2/7]

First, I rewrote `fibo` to store some temporary values in variables.

```
bignum fibo(int n)
{
    bignum r1, r2;

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

    // RECURSIVE CASE
    r1 = fibo(n-2);  // Recursive call #1
    r2 = fibo(n-1);  // Recursive call #2
    return r1 + r2;  // Return the result
}
```

Applications — Eliminating Recursion: Example [3/7]

We need a Stack. It should hold:

- Local variables (`n`, `r1`, `r2`) and the return value.
- Return address (outside world, recursive call #1, recursive call #2).

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

```
struct FiboStackFrame {
    int    n;             // Parameter
    bignum r1;            // Result of recursive call #1
    bignum r2;            // 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
};
```

Applications — Eliminating Recursion: Example [4/7]

We create our Stack when we enter function `fibo`.

```
stack<FiboStackFrame> cs;  // Call stack
```

Then we can store our local variables there.

- For example, “`n`” becomes “`cs.top().n`”.

We need variables to hold values during Stack operations.

- There will be both `int` and `bignum` values.

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

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

Applications — Eliminating Recursion: Example [5/7]

To make a recursive call:

- Set up the Stack and restart the loop (`continue`).
- Enable the function to return to just after where the call was made. Use a **label**, which we can return to with `goto`.

Here is "`r1 = fibo(n-2);`":

```
    tmpi = cs.top().n - 2;
    cs.push(FiboStackFrame());  // Make new stack frame
    cs.top().n = tmpi;          // Set parameter
    cs.top().returnAddr = 1;    // Return addr: call #1
    continue;                   // Do "recursive call"
return_here_1:                  // Place to return to
    tmpb = cs.top().returnValue;
    cs.pop();
    cs.top().r1 = tmpb;         // Returned value -> r1
```

Label

Applications — Eliminating Recursion: Example [6/7]

To "return":

- If we were called by the outside world, then really `return`.
- Otherwise, set up the return value, and `goto` the appropriate label.

Here is "`return bignum(n);`":

```
cs.top().returnValue = bignum(cs.top().n);
if (cs.top().returnAddr == 1)       // Back to call #1
    goto return_here_1;
else if (cs.top().returnAddr == 2)  // Back to call #2
    goto return_here_2;
else                                // Back to outside world
{
    tmpb = cs.top().returnValue;
    cs.pop();
    return tmpb;
}
```

Convention. All pushing and popping is done in the "caller". So when "returning" from a recursive call, we do not need to pop in this code.

Stacks
Applications — Eliminating Recursion: Example [7/7]

And it works! (Try it!)

This example might seem silly. It *is* a bit silly.

So, what is the point?

- Recursion is a powerful theoretical tool. As an implementation method, it is *sometimes* problematic. However, it can *always* be replaced by iteration.
- Computer programming is a discipline in which theoretical knowledge is often closely connected to practical reality. We can *theoretically* eliminate recursion. Applying the theoretical ideas, we can, *in practice*, eliminate recursion.
- For convenience, our operating system and runtime environment provide many useful facilities for us, like the call stack. However, in many cases we can write our own versions, if the provided facilities do not meet our needs.