Programming Abstractions

CS106B

Cynthia Lee

Today’s Topics

ADTs

- Stack
  - Example: Reverse-Polish Notation calculator
- Queue
  - Example: Mouse Events

Stacks

stack.h

New ADT: Stack

```
template <typename ValueType> class Stack {
public:
    Stack();
    virtual ~Stack();
    int size() const;
    bool isEmpty() const;
    void clear();
    void push(ValueType value);
    ValueType pop();
    ValueType peek() const;
    std::string toString();
private:
    -Redacted-

};
```

Source: http://www.flickr.com/photos/35237093334@N01/409465578/
Author: http://www.flickr.com/people/35237093334@N01 Peter Kazanjy]

Using a Stack

```
Stack<int> s;                // {} bottom -> top
s.push(42);                  // {42}
s.push(-3);                  // {42, -3}
s.push(17);                  // {42, -3, 17}
cout << s.pop() << endl;     // 17 (s is {42, -3})
cout << s.peek() << endl;    // -3 (s is {42, -3})
cout << s.pop() << endl;     // -3 (s is {42})
```

Remember this format—we'll use it in section examples and on exams

Using a Stack to *buffer* (*i.e.*, temporarily hold) file input

LIFO

```
void mystery(ifstream& infile) {
    Stack<string> lines;
    string line;
    while (getline(infile,line)) {
        lines.push(line);
    }
    infile.close();
    while (!lines.isEmpty()) {
        cout << lines.pop()
             << endl;
    }
}
```

hi
bye
hello

Reading from a file basics:

- `ifstream` is an *input* stream (like cin) but from a *file*
- `getline` takes the file and a string by reference, and reads one line into string
  - Returns false when there are no more lines to read

What does this code do?

Using a Stack to *buffer* (*i.e.*, temporarily hold) file input

```
void mystery(ifstream& infile) {
    Stack<string> lines;
    string line;
    while (getline(infile,line)) {
        li
    }
    infile
    while (!lines.isEmpty()) {
        cout << lines.pop()
             << endl;
    }
}
```

Why do I need Stack?
I could have done that with a Vector!

Reading from a file basics:

- `ifstream` is an *input* stream (like cin) but from a *file*
- `getline` takes the file and a
  y reference, and reads
  into string
- ns false when there are
  ore lines to read

What does this code do?

Stack or Vector?

```
void mystery(ifstream& infile) {
    Stack<string> lines;
    string line;
    while (getline(infile,line)) {
        lines.push(line);
    }
    infile.close();
    while (!lines.isEmpty()) {
        cout << lines.pop()
             << endl;
    }
}
```

```
Vector<string> lines;
```

```
lines.insert(lines.size(), line);
```

```
cout << lines[lines.size()-1]
     << endl;
lines.remove(lines.size()-1);
```

Vector version

```
void mystery(ifstream& infile) {
    Vector<string> lines;
    string line;
    while (getline(infile,line)) {
        lines.insert(lines.size(),line);
    }
    infile.close();
    while (!lines.isEmpty()) {
        cout << lines[lines.size()-1]
             << endl;
        lines.remove(lines.size()-1);
    }
}
```

This code isn’t terrible, but it is harder to read quickly, and is probably more error prone.

For example, it would be easy to forget the “-1” when you remove “lines.size()-1”.

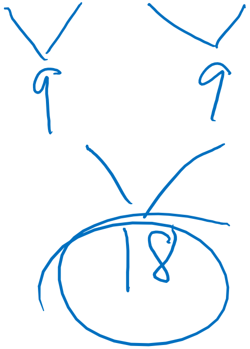
Applications of Stacks

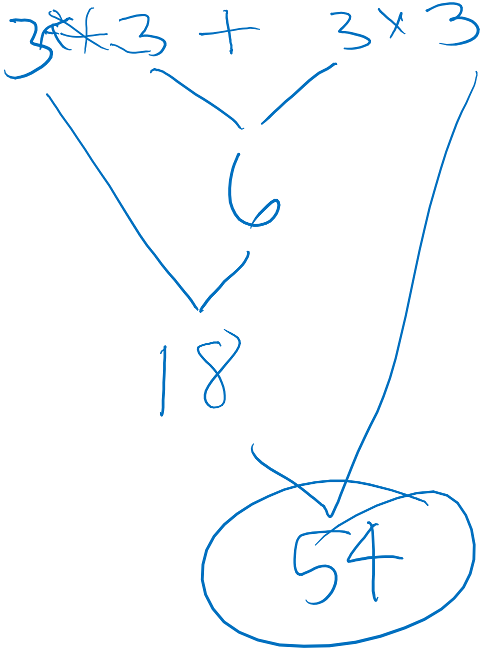
We’ve seen one (buffering input and giving it back in reverse—LIFO—order). What else are Stacks good for?

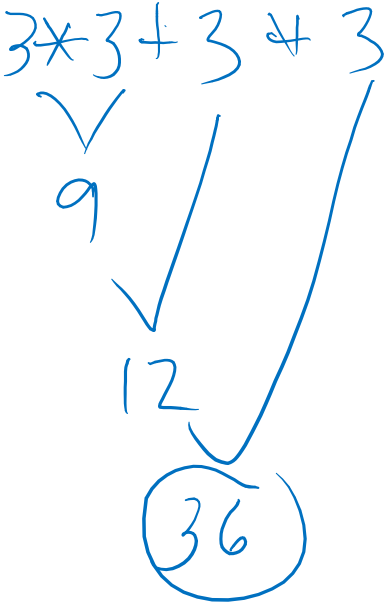
Operator Precedence and Syntax Trees

Ignoring operator precedence rules, how many distinct results are there to the following arithmetic expression?

- 3 * 3 + 3 * 3







Reverse Polish Notation

Ambiguities don’t exist in RPN! ☺

Also called “postfix notation” because the operator goes after the operands

Postfix (RPN):

- 4 3 * 3 +

Equivalent Infix:

- (4*3) + (3)

http://commons.wikimedia.org/wiki/File:Hewlett-Packard_48GX_Scientific_Graphing_Calculator.jpg

Reverse Polish Notation

This postfix expression:

- 4 3 * 7 2 5 * + +

10

Is equivalent to this infix expression:

A. ((4*3) + (7*2)) + 5

B. (4*3) + ((7+2) + 5)

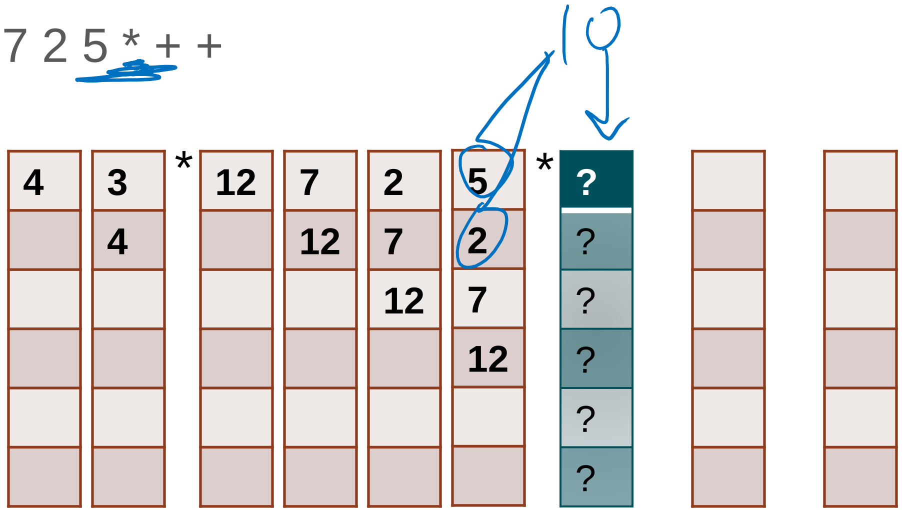
[✓ C. (4*3) + (7 + (2*5))]

D. Other/none/more than one

http://commons.wikimedia.org/wiki/File:Hewlett-Packard_48GX_Scientific_Graphing_Calculator.jpg

Stacks and RPN

- Evaluate this expression with the help of a stack
  - Encounter a **number**: **PUSH** it
  - Encounter an **operator**: **POP** two numbers and **PUSH** result
- 4 3 * 7 2 5 * + +



Contents of the stack, reading from top down:

(A) 7, 12

[✓ (B) 10, 7,12]

(C) 10, 5, 2, 7, 12

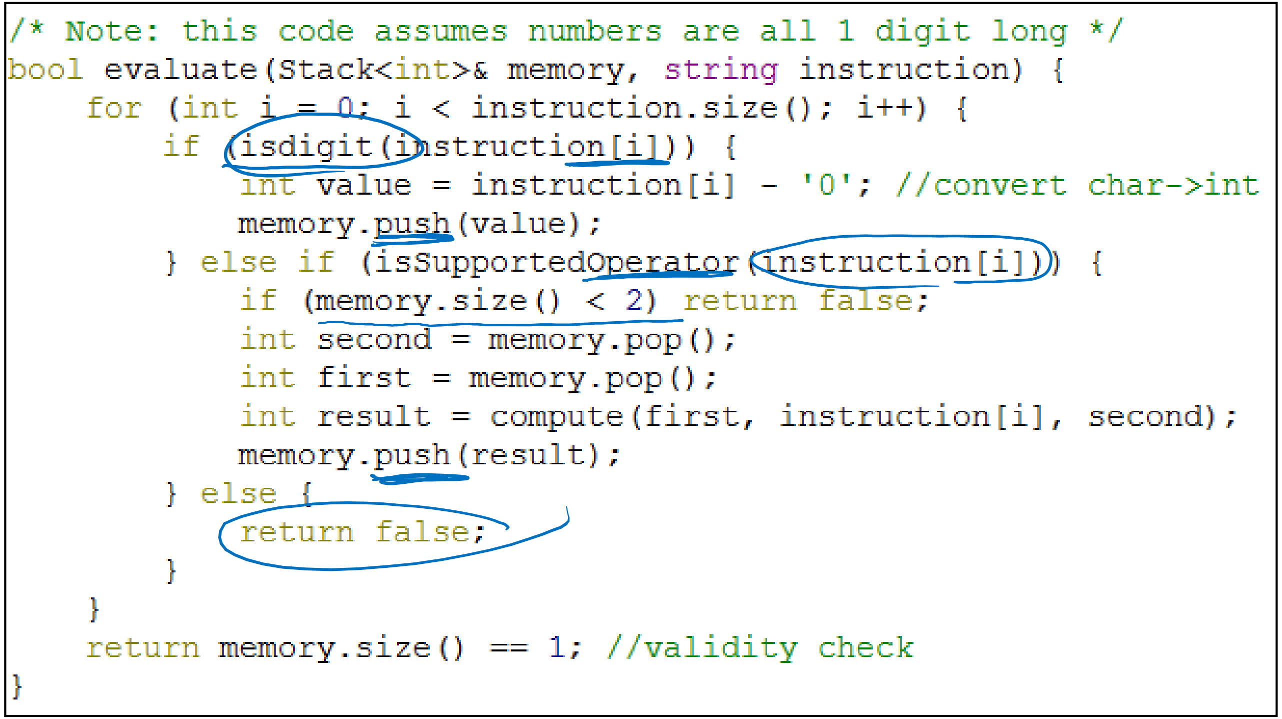
(D) Other

Stacks and RPN: What does that look like in code?

Evaluate this expression with the help of a stack

- › Encounter a **number**: **PUSH** it
- › Encounter an **operator**: **POP** two numbers and **PUSH** result

43*725*++



```
/* Note: this code assumes numbers are all 1 digit long */
bool evaluate(Stack<int>& memory, string instruction) {
    for (int i = 0; i < instruction.size(); i++) {
        if (isdigit(instruction[i])) {
            int value = instruction[i] - '0'; //convert char->int
            memory.push(value);
        } else if (isSupportedOperator(instruction[i])) {
            if (memory.size() < 2) return false;
            int second = memory.pop();
            int first = memory.pop();
            int result = compute(first, instruction[i], second);
            memory.push(result);
        } else {
            return false;
        }
    }
    return memory.size() == 1; //validity check
}
```

Queues

WHAT ARE THEY?
EXAMPLE APPLICATION

Queues

They work the same way a waiting in line (or, if you're in the UK, *queuing*) works.

FIFO = "First in, first out"

"First come, first serve"

Waiting for Apple Watch

New ADT: Queue

queue.h

```
template <typename ValueType> class Queue {
public:
    Queue();
    virtual ~Queue();
    int size() const;
    bool isEmpty() const;
    void clear();
    void enqueue(ValueType value);
    ValueType dequeue();
    ValueType& back();
    ValueType& front();
    ValueType peek() const;
    std::string toString();
private:
```

-Redacted-

Using a Queue

```
Queue<int> q;                    // {} front -> back
q.enqueue(42);                   // {42}
q.enqueue(-3);                   // {42, -3}
q.enqueue(17);                   // {42, -3, 17}
cout << q.dequeue() << endl;     // 42 (q is {-3, 17})
cout << q.peek() << endl;        // -3 (q is {-3, 17})
cout << q.dequeue() << endl;     // -3 (q is {17})
```

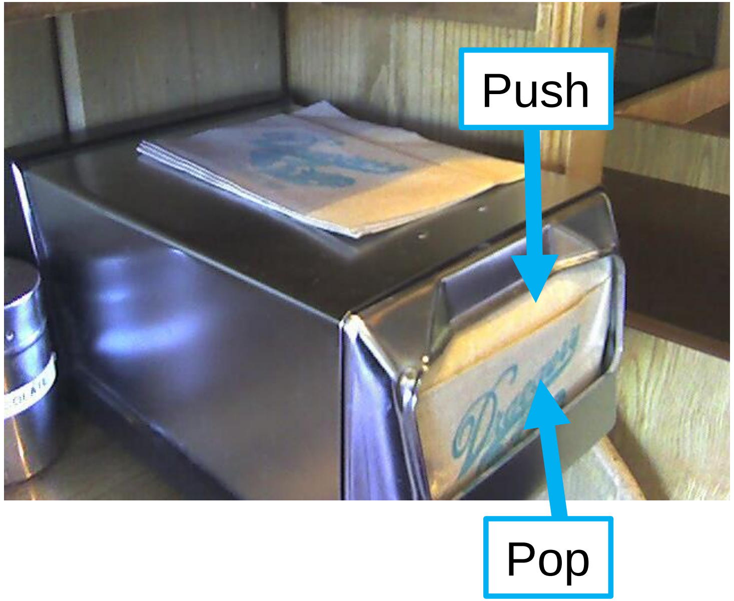
~~q[3]~~

Remember this format—we'll use it in section examples and on exams

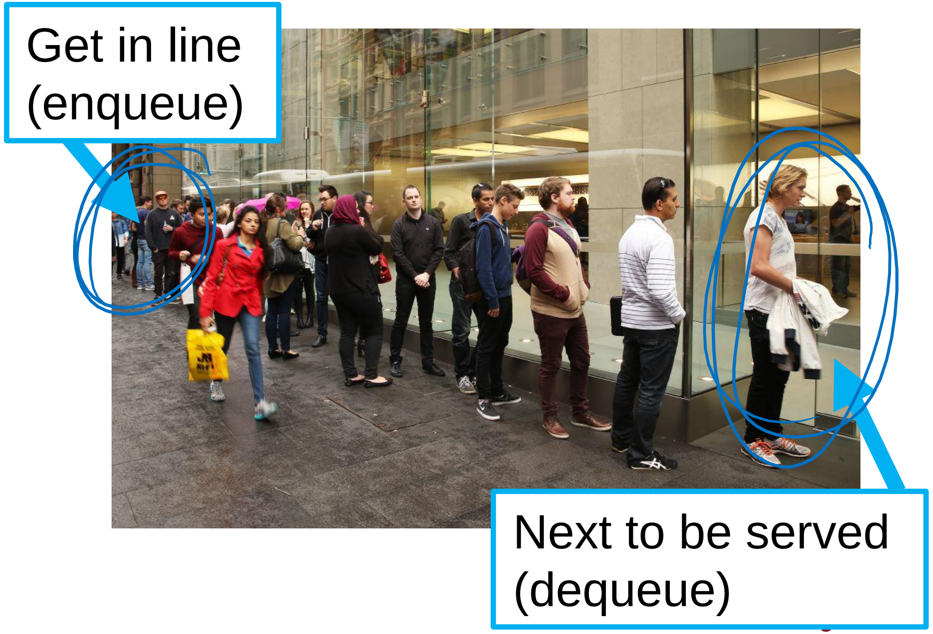
Where Stack and Queue are accessed

This may seem obvious, but it's an important point for the behind-the-scenes code implementation of these data structures:

Stack: only accessed on one end



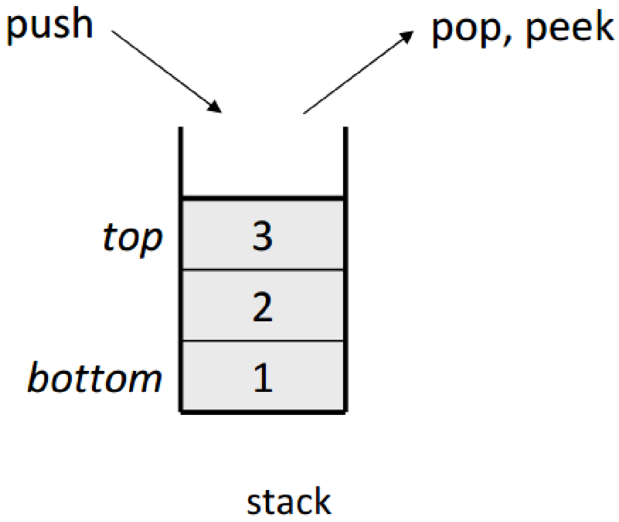
Queue: accessed at both ends



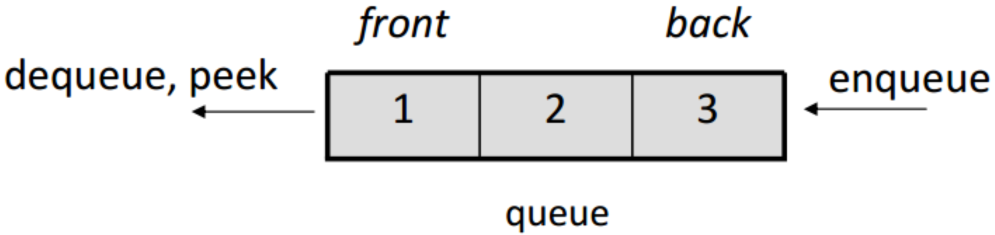
Where Stack and Queue are accessed

This may seem obvious, but it's an important point for the behind-the-scenes code implementation of these data structures:

Stack: only accessed on one end



Queue: accessed at both ends



Event queues (used in Fauxtoshop)

While your code executes, a separate part of the program is constantly running, and its only job is listening for events and recording them

- Mouse moved, mouse clicked, mouse dragged, etc

Every once in a while, your code can call **`getNextEvent()`** to see what has happened

- `getNextEvent()` returns the events one at a time, in the same order they happened
  - In other words, returns them in **FIFO** order
  - When it is “recording” events, it is **enqueuing** events in an event **QUEUE**

Very common use of the Queue ADT