CS180 - Stacks & queues

Note Title 9/10/2010

Announcements

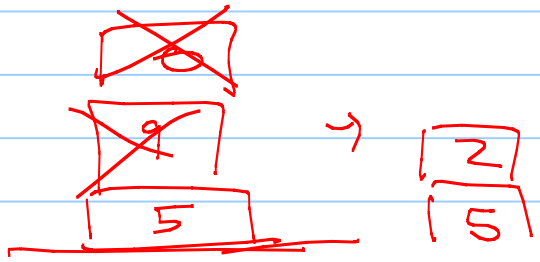
- HW 2 is out, due in 1 week
- Midterm 1 in 2-3 weeks (?)
  (more next time on date)

Last time: Stacks

Operations: - push
- pop

```
push(5)
push(9)
push(0)
pop()  → 0
pop()  → 9
push(2)
```

Stack



int num(5);

We coded the main functions.

What is left?

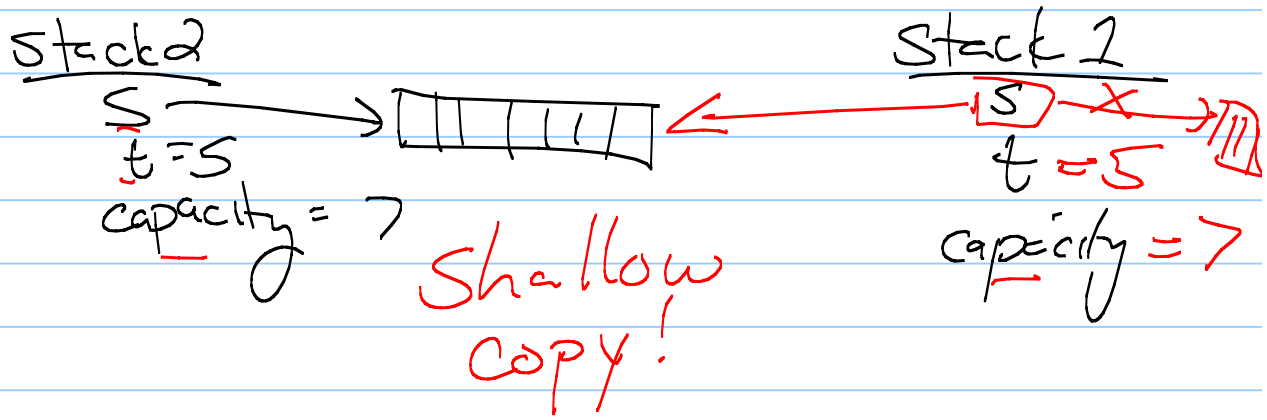
Hint: Consider this command

Stack1 = stack2

As default, C++ will run = for each private piece of data.

~~Problems~~

1. Shallow copy
2. Memory leaks - need a destructor

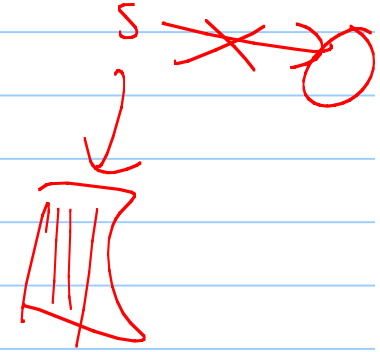


stack1 = stack2 = stack3;

① ②

Operator =

```
ArrayStack & operator=(const ArrayStack& other) {
  if (this != &other) {
    capacity = other.capacity;
    t = other.t;
    delete [] S;
    S = new Object[capacity];
    for (int i=0; i<=t; i++)
    {
      S[i] = other.S[i];
    }
  }
  return *this;
}
```



Destructor: Is called when function ends (or scope ends).

```
~ArrayStack() {
    delete[] S;
}
```

C++ automatically deletes private variables - just not what variables point to.

Aside:

```
while ( . . ) {
    int num;
    :
    :
}
num ← error
```

careful!

Copy Constructor

in main:

```
ArrayStack<int> stack2(stack1);
```

```
ArrayStack(const ArrayStack& other):
    capacity(other.capacity), t(other.t) {
    S = new Object[capacity];
    for (int i=0; i<=t; i++)
    {
        S[i] = other.S[i];
    }
}
```

Final Code

↳ on webpage

Note: Different implementations are possible!

See book versus webpage versus lecture version.

Running Times:

Function	Time
size	O(1)
isEmpty	O(1)
top	O(1)
push	O(1)
pop	O(1)

← return S[t];

copy Constructor: O(n)
(for loop)

Size usage?

not O(n) — O(capacity)

Sec. 4.2.3 - Function calls + stacks

C++ actually keeps a private stack, called the run-time stack, to keep track of local variables.

Why is this data structure ideal?

function {

while {

if {

} } }

(See p. 166-168 for more detail.)

stack: LIFO

Queues: Another way of Storing a list

First in, First out (FIFO)

Two main functions:

enqueue(o): Insert object o at the rear of the queue

push

dequeue(): Remove & return the object at the front of the queue

pop

Other operations

- size()
- isEmpty()
- first() (same idea as top) does not remove, just returns first element

Operation	Output	Queue
enqueue(5)	–	<5>
enqueue(3)	–	<5, 3>
dequeue()	5	<3>
enqueue(7)	–	<3, 7>
dequeue()	3	<7>
front()	7	<7>
dequeue()	7	<>
dequeue()	error	<>
isEmpty()	true	<>
enqueue(9)	–	<9>
enqueue(7)	–	<9, 7>
size()	2	<9, 7>
enqueue(3)	–	<9, 7, 3>
enqueue(5)	–	<9, 7, 3, 5>
dequeue()	9	<7, 3, 5>

Alright - let's think about the setup:

```
template <typename Object>
Class Queue {
    public:

        int size() const;

        bool isEmpty() const;

        const Object& front() const;

        void enquene(Object obj);

        Object dequene();
};
```