CS180 - Lecture 8: Stacks

Note Title 9/10/2009

Announcements

- Program 1 due next Friday (start this weekend!)
- First midterm in 2 weeks: either Wed., Sept. 23 or Thurs., Sept. 24 (with lab on other day)
  Preference?
- HW3 will come out next week, & program 2 will come out right before/after the midterm

Template (Sec. 2.3)

Consider a function:

```
int min(int a, int b) {
    if (a<b)
        return a;
    else
        return b;
}
```

Alternate:

```
int main(int a int b)
  { return (a<b ? a : b); }
```

Seems handy...

Function templates:

this is a parameter list (only one here - called T)

```
template <typename T>
T min(T a, T b) {
    if (a<b)
        return a;
    else
        return b;
}
```

Important: Will work for any class, as long as "<" has been defined!

Class templates: a vector example

```
template <typename Object>
class BasicVector {
private:
    Object* a;          // array of elements
    int capacity;       // length of array a
public:
    BasicVector(int c=10) {   //constructor
        capacity = c;
        a = new Object[capacity]; //allocate storage
    }

    Object& elemAtRank(int r)  //access r^th element
        {return a[r];}
    :
```

Note: in C++, arrays are pointers!

Can always set an array using new, or just put a pointer to first element.

Then pointer is address of first element, so we can add to that number, or just say pointer[index].

(Sec. 1.1.3)

Back to BasicVector: usage

```
BasicVector<int>    intvec(5); //vector of 5 ints
BasicVector<string> strvec(10); //vector of 10 strings

intvec.elementAtRank(3) = 8; //sets 4th element = 8
strvec.elementAtRank(7) = "hello"; //sets 8th elt = "hello"
```

Or even:

```
BasicVector<BasicVector<int> > myvec(5);
        //vector of 5 BasicVectors of integers

myvec.elementAtRank(2).elementAtRank(8) = 15;
        //  myvec[2][8] = 15
```

Exceptions

In C++, exceptions are "thrown" by code that encounters something odd.

(Relatively new addition to C++, so Standard template library doesn't always use them!)

Exceptions are often inherited, since might have a set of similar ones...

Example: Math errors

```
class MathException {
    private:
        string errMsg;
    public:
        MathException(const string& err)
            { errMsg = err; }

        // probably others to access message, etc...
};
```

More specific exceptions:

```
class ZeroDivideException : public MathException {
  public:
    ZeroDivideException(const string &err) :
        MathException(err) {}
};

class NegativeRootException : public MathException {
  public:
    NegativeRootException(const string & err):
        MathException(err) {}
};
```

Throwing & Catching Exceptions

```
try {
    // ... some computations
    if (divisor == 0)
        throw ZeroDivideException("Divide by 0
                                   in Module X");
}
catch (ZeroDivideException& zde) {
    // handle division by 0
}
catch (MathException & me) {
    // handle any others
}
```

will also catch NegativeRootException

What happens?

- When divisor is equal to 0, it immediately "jumps to that "catch".
- If the exception is not caught, the program just aborts.
- In previous example, if we had thrown a NegativeRootException, it would have been caught by the MathException since that is closest matching catch.
- catch(...) ← catches all exceptions (like blank except in Python)

How do we recover?

Depends on type of exception.

- often, print error & end program.
- May require clean-up, such as deallocating memory.

Exceptions in functions

When we declare a function, we should also specify what exceptions might occur.

- lets user know what to expect, so they can handle appropriately
- means we don't have to handle exceptions - will be passed up (see p. 95 of text for details)

Syntax: Exceptions in functions

```
void calculator() throw (ZeroDivideException, NegativeRootException)
{
  // function body
}
```

- Means we can throw only these 2 exceptions in calculator (or any child classes).

```
main {
    try {
        calculator();
    }
    catch ZeroDivideException . ——
```

What do these mean?

```
void funct1()
{
  //body
}
```

] can throw any exceptin

```
void funct2() throw()
{
  //body
}
```

] can't throw any exceptions

One final recap: Recursion

What is it?

a function that calls itself

↑ or object

(See sec. 2.5 and 4.1 for a review)

Stack : a way to store a list of data

Ex: Web browser: Store history

last in is first out

(LIFO)

Ex: Text editors (commands): Store previously executed

undoes most recent action

The Stack Abstract Data Type (ADT)

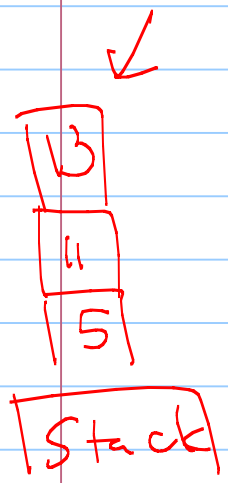
Supports 2 main functions:

- push(o)

  Insert object o at top of stack

- pop()

  Remove top object from stack & return it



① push (5)
② push (11)
③ push (-3)

④ pop() - returns -3
⑤ push (13)

Additional behaviors

- size(): Return # of objects in the stack
- isEmpty(): Returns true if stack is empty, false otherwise
- top(): Returns top object on stack without removing it

(like pop, but does not remove elt)

Notice:

I haven't said what this is made with!

Ideas?

- private data an array
  - keep track of "top" element
- maximum size to our stack
- or, if array is too small, delete it & create larger