CMSC 341
Lecture 7

Announcements

Proj 2 up

Project Preview tonight and tomorrow

Comparing Performance

	Linear	S Linked	D Linked	Cursor
constructor	O(1)	O(1)	O(1)	O(1)
find	O(n)	O(n)	O(n)	O(n)
findPrev	O(n)	O(n)	O(n)	O(n)
insert	O(n)	O(1)	O(1)	O(1)
remove	O(n)	O(n)	O(n)	O(n)
makeEmpty	O(1)	O(n)	O(n)	O(n)

Stack ADT

Restricted List

- only add to top
- only remove from top

Examples

- pile of trays
- partial results
- local state

Relation to Previous List

Using inheritance:

- Stack Is-A List
- Need to pay special intention that List operations are properly performed

Using aggregation:

- Stack Has-A List

Stack.H

```
#include “LinkedList.H”
template <class Object>
class Stack {
public:
   Stack();
   Stack(const Stack &coll);
   ~Stack();
   bool isEmpty() const;
   bool isFull() const;
   const Object &top() const;
   void makeEmpty();
   void pop();
   void push(const Object &x);
   Object topAndPop();
   const Stack &operatr=(const Stack &stk);
```

Stack.H (cont)

```
protected:
   const List<Object> &getList() const;
   List<Object> &getList();
   void setList(List<Object> &lst
   ListItr<Object> &getZeroth();

private:
   List<Object> _theList;
   ListItr<Object> _zeroth;
};
```

Stack.C

```
template <class Object>
Stack<Object>::Stack(){
   _zeroth = getList().zeroth();
   }
template <class Object>
Stack<Object>::Stack(const Stack &stk) {
   _theList = stk.getList();
   _zeroth = getList().zeroth();
}
template <class Object>
Stack<Object>::~Stack() {}

template <class Object>
bool Stack<Object>::isEmpty() const{
   return getList().isEmpty();
   }
```

Stack.C (cont)

```
template <class Object>
bool Stack<Object>::isFull()const{
   return false;
   }
template <class Object>
const Object & Stack<Object>::top() const{
   if (isEmpty())
        throw StackException("top on empty stack");
   return getList().first().retrieve();
}
template <class Object>
void Stack<Object>::makeEmpty () {
   getList().makeEmpty();
}
```

Stack.C (cont)

```
template <class Object>
 void Stack<Object>::pop() {
   if (isEmpty())
        throw StackException("pop on empty stack");
   getList().remove(top());
   }
template <class Object>
void Stack<Object>::push( const Object &x) {
   getList().insert(x, getZeroth());
}
template <class Object>
Object Stack<Object>::topAndPop () {
   if (isEmpty())
        throw StackException("topAndPop on empty stack");
   Object tmp = top();  pop();
   return tmp;
}
```

Stack.C (cont)

```
template <class Object>
const Stack<Object> &Stack<Object>::operator=( const
   Stack &stk) {
   if (this != &stk){
        setList(stk.getList());
        setZeroth(getList().zeroth());
        }
   return *this;
   }
template <class Object>
const List<Object> &Stack<Object>:: getList() {
   return _theList;
}
template <class Object>
ListItr<Object> &Stack<Object>:: getZeroth () {
   return _zeroth;
}
```

StackException.H

```
class StackException
{
 public:
  StackException(); // Message is the empty string
  StackException(const string & errorMsg);
  StackException(const StackException & ce);
  ~StackException();
  const StackException & operator=(const StackException
   & ce);
const string & errorMsg() const; // Accessor for msg

 private:
  string _msg;
};
```

StackException.C

```
StackException::StackException(){}

StackException::StackException(const string & errorMsg){
  _msg = errorMsg;
}

StackException::StackException(const StackException &ce){
  _msg = ce.errorMsg();
}

StackException::~StackException(){}
```

StackException.C (cont)

```
const StackException &
StackException::operator=(const StackException & ce){
  if (this == &ce)
    return *this;   // don't assign to itself
  _msg = ce.errorMsg();
  return *this;
}

const string & StackException::errorMsg() const {
  return _msg;
}
```

TestStack.C

```
int main (){
  Stack<int> stk;

  stk.push(1);
  stk.push(2);
  printStack(stk);

  Stack<int> otherstk;
  otherstk = stk;
  printStack(otherstk);

  cout << stk.topAndPop() << endl;
  cout << stk.topAndPop() << endl;
```

TestStack.C (cont)

```
  printStack(stk);
  printStack(otherstk);

  try {
      stk.pop();
    }
  catch (StackException & e){
      cout << e.errorMsg() << endl;
    }
}
```

TestStack_aux.C

```
template <class Object>
void printStack( Stack<Object> & theStack ){
  Stack<Object> tmp;

  if( theStack.isEmpty( ) ){
      cout << "Empty stack" << endl;
      return;
    }
  while (theStack.isEmpty() == false) {
      Object topObj = theStack.top();
      cout << topObj;
      tmp.push(topObj);  // save on other stack
      theStack.pop();
      if (theStack.isEmpty() == false)
          cout << ", ";
    }
  cout << endl;
```

TestStack_aux.C

```
while (tmp.isEmpty() == false)
    {
      Object topObj = tmp.top();
      theStack.push(topObj);
      tmp.pop();
    }
}
```

Queue ADT

Restricted List

- only add to end
- only remove from front

Examples

- line waiting for service
- jobs waiting to print

Queue.H

```
template <class Object>
class Queue {
public:
   explicit Queue(int capacity=10);
   bool isEmpty() const;
   bool isFull() const;
   const Object &getFront() const;
   void makeEmpty();
   void dequeue();
private:
   vector<Object> theArray;
   int currentSize;
   int front;
   int back;
   void increment (int &x);
   }
```

Queue.C

```
template <class Object>
void Queue<Object>::enqueue(const Object &x){
   if (isFull())
       throw Overflow();
   increment (back);
   theArray[back] = x;
   currentSize++;
   }
template <class Object>
void Queue<Object>::increment( int &x) {
   if (++x == theArray.size())
       x = 0;
   }
```

theArray

Queue.C (cont)

```
template <class Object>
Object Queue<Object>::dequeue(){
   if (isEmpty())
       throw Underflow();
   currentSize--;
   Object frontItem = theArray[front];
   increment(front);
   return frontItem;
   }
```