EECE.3220: Data Structures

Spring 2017

Lecture 14: Key Questions
February 24, 2017

1. Describe the characteristics of an ADT to store a list.

2. What data members would be necessary for a static array-based list implementation?

3. Describe the basic algorithm for inserting an element into an array-based list.

4. Describe the basic algorithm for removing an element from an array-based list.

5. Describe the key characteristics of the static array-based List implementation (files attached)

6. Explain the use of the `const` keyword with variables, arguments, and methods.

7. Explain the general practice of operator overloading and the specific example (<<) shown for the `List` class, both as a true non-member function and a friend function.

8. Explain the `new` and `delete` operators.

9. If we implement a list using dynamic arrays, what parts of the class stay the same? What's different? What needs to be added?

10. Explain how the constructor needs to change in a dynamically allocated list.

11. Explain the purpose and operation of a destructor.

12. Explain the purpose and operation of copy constructors and the = operator. Why are the default versions of these functions not suitable for objects with dynamically allocated members?

```
//
//  List.h
//  ds_test
//
//  Created by Michael Geiger on 2/24/17.
//  Figure 6.1A from Nyhoff text
//

#include <iostream>
using std::ostream;

#ifndef LIST
#define LIST

const int CAPACITY = 1024;  // Maximum array size
typedef int ElementType;    // Define "ElementType" as generic type name
                            // Can change specific type by changing "int"

class List {
public:
 /******** Function Members ********/
    /***** Class constructor *****/
    List();
    /*-------------------------------------------------------------------
     Construct a List object.

     Precondition:  None
     Postcondition: An empty List object has been constructed;
     mySize is 0.
     -------------------------------------------------------------------*/

    /***** empty operation *****/
    bool empty() const;
    /*-------------------------------------------------------------------
     Check if a list is empty.

     Precondition:  None
     Postcondition: true is returned if the list is empty,
     false if not.
     -------------------------------------------------------------------*/

    /***** insert and erase *****/
    void insert(ElementType item, int pos);
    /*-------------------------------------------------------------------
     Insert a value into the list at a given position.

     Precondition:  item is the value to be inserted; there is room in
     the array (mySize < CAPACITY); and the position satisfies
     0 <= pos <= mySize.
     Postcondition: item has been inserted into the list at the position
     determined by pos (provided there is room and pos is a legal
     position).
     -------------------------------------------------------------------*/

    void erase(int pos);
    /*-------------------------------------------------------------------
```

```
      Remove a value from the list at a given position.

      Precondition:  The list is not empty and the position satisfies
      0 <= pos < mySize.
      Postcondition: element at the position determined by pos has been
      removed (provided pos is a legal position).
      ------------------------------------------------------------------------*/

     /***** output *****/
     void display(ostream & out) const;
     /*-------------------------------------------------------------------------
      Display a list.

      Precondition:  out is a reference parameter
      Postcondition: The list represented by this List object has been
      inserted into ostream out.
      ------------------------------------------------------------------------*/

private:
 /******** Data Members ********/
    int mySize;                     // current size of list stored in myArray
    ElementType myArray[CAPACITY];  // array to store list elements

}; //--- end of List class

//------ Prototype of output operator
ostream & operator<< (ostream & out, const List & aList);

#endif
```

```
//
//  List.cpp
//  ds_test
//
//  Created by Michael Geiger on 2/24/17.
//  Figure 6.1B from Nyhoff text
//

#include "List.h"
using namespace std;

//--- Definition of class constructor
List::List()
: mySize(0)
{}

//--- Definition of empty()
bool List::empty() const {
    return (mySize == 0);
}

//--- Definition of display()
void List::display(ostream & out) const {
    for (int i = 0; i < mySize; i++)
        out << myArray[i] << "  ";
}

//--- Definition of output operator
ostream & operator<< (ostream & out, const List & aList) {
    aList.display(out);
    return out;
}

//--- Definition of insert()
void List::insert(ElementType item, int pos) {
    if (mySize == CAPACITY) {
        cerr << "*** No space for list element -- terminating "
        "execution ***\n";
        exit(1);
    }
    if (pos < 0 || pos > mySize) {
        cerr << "*** Illegal location to insert -- " << pos
            << ".  List unchanged. ***\n";
        return;
    }

    // First shift array elements right to make room for item
    for(int i = mySize; i > pos; i--)
        myArray[i] = myArray[i - 1];

    // Now insert item at position pos and increase list size
    myArray[pos] = item;
    mySize++;
}

// Definition of erase()
```

```
void List::erase(int pos) {
    if (mySize == 0) {
        cerr << "*** List is empty ***\n";
        return;
    }
    if (pos < 0 || pos >= mySize) {
        cerr << "Illegal location to delete -- " << pos
            << ".  List unchanged. ***\n";
        return;
    }

    // Shift array elements left to close the gap
    for(int i = pos; i < mySize; i++)
        myArray[i] = myArray[i + 1];

    // Decrease list size
    mySize--;
}
```

```
//
//  List.h
//  ds_test
//
//  Created by Michael Geiger on 2/24/17.
//  Figure 6.2A from Nyhoff text
//

/*-- List.h -----------------------------------------------------------

 This header file defines the data type List for processing lists.
 Basic operations are:
 Constructor
 Destructor
 Copy constructor
 Assignment operator
 empty:    Check if list is empty
 insert:   Insert an item
 erase:    Remove an item
 display:  Output the list
 << :      Output operator
 -------------------------------------------------------------------*/

#include <iostream>
using std::ostream;

#ifndef LIST
#define LIST

typedef int ElementType;
class List
{
public:
 /******** Function Members ********/
    /***** Class constructor *****/
    List(int maxSize = 1024);
    /*-------------------------------------------------------------------
     Construct a List object.

     Precondition:  maxSize is a positive integer with default value 1024.
     Postcondition: An empty List object is constructed; myCapacity ==
     maxSize (default value 1024); myArrayPtr points to a run-time
     array with myCapacity as its capacity; and mySize is 0.
     -------------------------------------------------------------------*/


    /***** Class destructor *****/
    ~List();
    /*-------------------------------------------------------------------
     Destroys a List object.

     Precondition:  The life of a List object is over.
     Postcondition: The memory dynamically allocated by the constructor
     for the array pointed to by myArrayPtr has been returned to
     the heap.
     -------------------------------------------------------------------*/
```

```
    /***** Copy constructor *****/
    List(const List & origList);
    /*-------------------------------------------------------------------
     Construct a copy of a List object.

     Precondition:  A copy of origList is needed; origList is a const
     reference parameter.
     Postcondition: A copy of origList has been constructed.
     -------------------------------------------------------------------*/

    /***** Assignment operator *****/
    List & operator=(const List & origList);
    /*-------------------------------------------------------------------
     Assign a copy of a List object to the current object.

     Precondition: none
     Postcondition: A copy of origList has been assigned to this
     object.  A reference to this list is returned.
     -------------------------------------------------------------------*/

    /***** empty operation *****/
    bool empty() const;
    /*-------------------------------------------------------------------
     Check if a list is empty.

     Precondition:  None
     Postcondition: true is returned if the list is empty,
     false if not.
     -------------------------------------------------------------------*/

    /***** insert and erase *****/
    void insert(ElementType item, int pos);
    /*-------------------------------------------------------------------
     Insert a value into the list at a given position.

     Precondition:  item is the value to be inserted; there is room in
     the array (mySize < CAPACITY); and the position satisfies
     0 <= pos <= mySize.
     Postcondition: item has been inserted into the list at the position
     determined by pos (provided there is room and pos is a legal
     position).
     -------------------------------------------------------------------*/

    void erase(int pos);
    /*-------------------------------------------------------------------
     Remove a value from the list at a given position.

     Precondition:  The list is not empty and the position satisfies
     0 <= pos < mySize.
     Postcondition: element at the position determined by pos has been
     removed (provided pos is a legal position).
     -------------------------------------------------------------------*/

    /***** output *****/
    void display(ostream & out) const;
```

```
    /*-------------------------------------------------------------------
     Display a list.

     Precondition:  The ostream out is open.
     Postcondition: The list represented by this List object has been
     inserted into out.
     -------------------------------------------------------------------*/

private:
 /******** Data Members ********/
    int mySize;               // current size of list stored in array
    int myCapacity;           // capacity of array
    ElementType * myArrayPtr;  // pointer to dynamically-allocated array

}; //--- end of List class

//------ Prototype of output operator
ostream & operator<< (ostream & out, const List & aList);

#endif
```

```
//
//  List.cpp
//  ds_test
//
//  Created by Michael Geiger on 2/24/17.
//  Figure 6.2B from Nyhoff text
//

#include <cassert>
#include <new>          // Necessary for (nothrow) version of new
using namespace std;

#include "DList.h"

//--- Definition of class constructor
List::List(int maxSize)
: mySize(0), myCapacity(maxSize)
{
    myArrayPtr = new(nothrow) ElementType[maxSize];
    assert(myArrayPtr != 0);
}

//--- Definition of class destructor
List::~List() {
    delete [] myArrayPtr;
}

//--- Definition of the copy constructor
List::List(const List & origList)
: mySize(origList.mySize), myCapacity(origList.myCapacity) {
    //--- Get new array for copy
    myArrayPtr = new(nothrow) ElementType[myCapacity];

    if (myArrayPtr != 0)                // check if memory available
        //--- Copy origList's array into this new array
        for(int i = 0; i < myCapacity; i++)
            myArrayPtr[i] = origList.myArrayPtr[i];
    else {
        cerr << "*Inadequate memory to allocate List ***\n";
        exit(1);
    }
}

//--- Definition of the assignment operator
List & List::operator=(const List & origList) {
    if (this != &origList) {   // check for list = list
        mySize = origList.mySize;
        myCapacity = origList.myCapacity;

        //-- Allocate a new array if necessary
        if (myCapacity != origList.myCapacity)
        {
            delete[] myArrayPtr;
            myArrayPtr = new(nothrow) ElementType[myCapacity];

            if (myArrayPtr == 0)   // check if memory available
```

```
            {
                cerr << "*Inadequate memory to allocate stack ***\n";
                exit(1);
            }
        }
        //--- Copy origList's array into this new array
        for(int i = 0; i < myCapacity; i++)
            myArrayPtr[i] = origList.myArrayPtr[i];
    }
    return *this;
}

//--- Definition of empty()
bool List::empty() const {
    return mySize == 0;
}

//--- Definition of display()
void List::display(ostream & out) const {
    for (int i = 0; i < mySize; i++)
        out << myArrayPtr[i] << "  ";
}

//--- Definition of output operator
ostream & operator<< (ostream & out, const List & aList) {
    aList.display(out);
    return out;
}

//--- Definition of insert()
void List::insert(ElementType item, int pos) {
    if (mySize == myCapacity) {
        cerr << "*** No space for list element -- terminating "
        "execution ***\n";
        exit(1);
    }
    if (pos < 0 || pos > mySize) {
        cerr << "*** Illegal location to insert -- " << pos
        << ".  List unchanged. ***\n";
        return;
    }

    // First shift array elements right to make room for item
    for(int i = mySize; i > pos; i--)
        myArrayPtr[i] = myArrayPtr[i - 1];

    // Now insert item at position pos and increase list size
    myArrayPtr[pos] = item;
    mySize++;
}

//--- Definition of erase()
void List::erase(int pos) {
    if (mySize == 0) {
        cerr << "*** List is empty ***\n";
        return;
```

```
    }
    if (pos < 0 || pos >= mySize) {
        cerr << "Illegal location to delete -- " << pos
        << ".  List unchanged. ***\n";
        return;
    }

    // Shift array elements left to close the gap
    for(int i = pos; i < mySize; i++)
        myArrayPtr[i] = myArrayPtr[i + 1];

    // Decrease list size
    mySize--;
}
```