Data Structures

data object

set or collection of instances

integer = \{0, +1, -1, +2, -2, +3, -3, \ldots\}

daysOfWeek = \{S,M,T,W,Th,F,Sa\}
Data Object

instances may or may not be related

myDataObject = \{\text{apple, chair, 2, 5.2, red, green, Jack}\}
Data Structure

Data object +
  relationships that exist among instances and elements that comprise an instance

Among instances of integer
369 < 370
280 + 4 = 284
Ordering Relationships

You can have ordering relationships

\[-2 < -1 < 0 < 1 \ldots\] (relational operators such as \(<\), \(>\), \(=\), etc). Elements could be related by operations such as add, subtract, etc.
Data Structure

Among elements that comprise an instance

369
3 is more significant than 6
3 is immediately to the left of 6
9 is immediately to the right of 6
Linear (or Ordered) Lists

instances are of the form
(e₀, e₁, e₂, ..., e_{n-1})

where eᵢ denotes a list element
n ⩾ 0 is finite
list size is n
Linear Lists

$L = (e_0, e_1, e_2, e_3, \ldots, e_{n-1})$

relationships

$e_0$ is the zero’th (or front) element
$e_{n-1}$ is the last element
$e_i$ immediately precedes $e_{i+1}$
Linear List Examples/Instances

Students in COP3530 =
(Jack, Jill, Abe, Henry, Mary, …, Judy)

Exams in COP3530 =
(exam1, exam2, exam3)

Days of Week = (S, M, T, W, Th, F, Sa)

Months = (Jan, Feb, Mar, Apr, …, Nov, Dec)
Linear List Operations—size()

determine list size

\[ L = (a,b,c,d,e) \]

size = 5
Linear List Operations—get(theIndex)

get element with given index

$L = (a,b,c,d,e)$

$\text{get}(0) = a$

$\text{get}(2) = c$

$\text{get}(4) = e$

$\text{get}(-1) = \text{error}$

$\text{get}(9) = \text{error}$
Linear List Operations—indexOf(theElement)

determine the index of an element

$L = (a, b, d, b, a)$

$\text{indexOf}(d) = 2$

$\text{indexOf}(a) = 0$

$\text{indexOf}(z) = -1$
Linear List Operations—remove(theIndex)

remove and return element with given index

$L = (a, b, c, d, e, f, g)$

remove(2) returns $c$

and $L$ becomes $(a, b, d, e, f, g)$

index of $d, e, f$, and $g$ decrease by 1
Linear List Operations—remove(theIndex)

remove and return element with given index

$L = (a, b, c, d, e, f, g)$

remove(-1) => error
remove(20) => error
Linear List Operations—
add(theIndex, theElement)

add an element so that the new element has a specified index

\[ L = (a,b,c,d,e,f,g) \]

\[ add(0,h) \Rightarrow L = (h,a,b,c,d,e,f,g) \]

index of \( a, b, c, d, e, f \) and \( g \) increase by 1
Linear List Operations—
add(theIndex, theElement)

\[ L = (a,b,c,d,e,f,g) \]

\[ add(2, h) \Rightarrow L = (a,b,h,c,d,e,f,g) \]
index of \( c,d,e,f, \) and \( g \) increase by 1

\[ add(10, h) \Rightarrow \text{error} \]

\[ add(-6, h) \Rightarrow \text{error} \]
Data Structure Specification

- Language independent
  - Abstract Data Type
- C++
  - Abstract Class
Linear List Abstract Data Type

AbstractDataType LinearList
{
    instances
        ordered finite collections of zero or more elements
    operations
        empty(): return true iff the list is empty, false otherwise
        size(): return the list size (i.e., number of elements in the list)
        get(index): return the index\textsuperscript{th} element of the list
        indexO \text{f}(x): return the index of the first occurrence of \textit{x} in
            the list, return -1 if \textit{x} is not in the list
        erase(index): remove the index\textsuperscript{th} element,
            elements with higher index have their index reduced by 1
        insert(\textit{theIndex}, \textit{x}): insert \textit{x} as the index\textsuperscript{th} element, elements
            with \textit{theIndex} \textgreater= index have their index increased by 1
        output(): output the list elements from left to right
}
Linear List As C++ Abstract Class

template<class T>
class linearList
{
    public:
        virtual ~linearList() {};
        virtual bool empty() const = 0;
        virtual int size() const = 0;
        virtual T& get(int theIndex) const = 0;
        virtual int indexOf(const T& theElement) const = 0;
        virtual void erase(int theIndex) = 0;
        virtual void insert(int theIndex, const T& theElement) = 0;
        virtual void output(ostream& out) const = 0;
};
**Extending A C++ Class**

```cpp
template<class T>
class arrayList : public linearList<T>
{
    // code for all abstract methods of linearList must come here
}
```
Data Type Of Array element[]

Data type of list elements is unknown.

Define element[] to be of template type T.
Length of Array element[]

Don’t know how many elements will be in list.

Must pick an initial length and dynamically increase as needed.

Use variable `arrayLength` to store current length of array `element[]`. 
Increasing Array Length

Length of array `element[]` is 6.

```
  a b c d e f
```

First create a new and larger array

```
T* newArray = new T[15];
```
Increasing Array Length

Now copy elements from old array to new one.

copy(element, element + 6, newArray);
Increasing Array Length

Finally, delete old array and rename new array.

```javascript
delete [] element;

element = newArray;

arrayLength = 15;
```

```
  a  b  c  d  e  f
```

`element[0]`
template<class T>
void changeLength1D(T*& a, int oldLength, int newLength)
{
    if (newLength < 0)
        throw illegalParameterValue();

    T* temp = new T[newLength];
    // new array
    int number = min(oldLength, newLength);
    // number to copy
    copy(a, a + number, temp);
    delete [] a;
    // deallocate old memory
    a = temp;
}
How Big Should The New Array Be?

At least 1 more than current array length.

Cost of increasing array length when array is full is $\Theta(\text{old length})$.

Cost of $n$ insert operations done on an initially empty linear list increases by $\Theta(n^2)$. 
Space Complexity

element[6]

```
  a  b  c  d  e  f
```

newArray = new char[7];

```
   _  _  _  _  _  _
```

space needed = 2 * newLength – 1

= 2 * maxListSize – 1
Array Doubling

Double the array length.

```
  a  b  c  d  e  f
```

```
newArray = new char[12];
```

```
  a  b  c  d  e  f  
```

Time for $n$ inserts goes up by $O(n)$.

Space needed = $1.5 \times $newLength.

Space needed $\leq 3 \times $maxListSize $- 3$
How Big Should The New Array Be?

Resizing by any constant factor

\[ \text{new length} = c \times \text{old length} \]

doubles the cost of \( n \) inserts by \( \Theta(n) \).

Resizing by an additive constant increases

the cost of \( n \) add operations by \( \Theta(n^2) \).
How Big Should The New Array Be?

Resizing by any constant factor

new length = c * old length

requires at most \((1+c) * (\text{maxListSize} - 1)\) space.

Resizing by an additive constant \(c\) requires

at most \((\text{maxListSize} - 1) + (\text{maxListSize} - 1 + c)\)

\[= 2 * (\text{maxListSize} - 1) + c\] space.
What Does C++ Do?

STL class `vector` … \( c = 1.5 \)

arrayList of text … \( c = 2 \)
The Class array\texttt{List}

- General purpose implementation of linear lists.
- Unknown number of lists.
Create An Empty List

arrayList<int> a(100), b;
arrayList<double> c(10), d;

linearList<int> *d = new ArrayList<int>(1000);
linearList<char> *f = new ArrayList<char>(20);
Using A Linear List

cout << a.size() << endl;
a.insert(0, 2);
d->insert(0, 4);
a.output();
cout << a << endl;
a.erase(0);
if (a.empty()) a.insert(0, 5);
Array Of Linear Lists

linearList<int> * x[4];
x[0] = new ArrayList<int>(20);
x[1] = new chain<int>();
x[2] = new chain<int>();
x[3] = new ArrayList<int>(15);
for (int i = 0; i < 4; i++)
    x[i].insert(0, i);
The Class arrayList

// include statements come here
using namespace std;
template<class T>
class arrayList : public linearList<T> {
    public:
        // constructor, copy constructor and destructor
        arrayList(int initialCapacity = 10);
        arrayList(const arrayList<T>&);
        ~arrayList() {delete [] element;};
The Class arrayList

// ADT methods
bool empty() const { return listSize == 0; }
int size() const { return listSize; }
T& get(int theIndex) const;
int indexOf(const T& theElement) const;
void erase(int theIndex);
void insert(int theIndex, const T& theElement);
void output(ostream& out) const;

// additional method
int capacity() const { return arrayLength; }
The Class arrayList

protected:

    void checkIndex(int theIndex) const;
        // throw illegalIndex if theIndex invalid
    T* element;      // 1D array to hold list elements
    int arrayLength;       // capacity of the 1D array
    int listSize;          // number of elements in list

};
A Constructor

template<class T>
arrayList<T>::arrayList(int initialCapacity)
{// Constructor.
    if (initialCapacity < 1)
    {
        ostringstream s;
        s << "Initial capacity = "
        << initialCapacity << " Must be > 0";
        throw illegalParameterValue(s.str());
    }
    arrayLength = initialCapacity;
    element = new T[arrayLength];
    listSize = 0;
}
template<class T>
arrayList<T>::arrayList(const arrayList<T>& theList)
{
    // Copy constructor.
    arrayLength = theList.arrayLength;
    listSize = theList.listSize;
    element = new T[arrayLength];
    copy(theList.element, theList.element + listSize, element);
}
The Method checkIndex

```cpp
template<class T>
void arrayList<T>::checkIndex(int theIndex) const
{// Verify that theIndex is between 0 and
 // listSize - 1.
    if (theIndex < 0 || theIndex >= listSize)
    {
        ostringstream s;
        s << "index = " << theIndex << " size = "
          << listSize;
        throw illegalIndex(s.str());
    }
}
```
The Method get

template<class T>
T& arraylist<T>::get(int theIndex) const
{// Return element whose index is theIndex.
 // Throw illegalIndex exception if no such
 // element.
    checkIndex(theIndex);
    return element[theIndex];
}
The Method indexOf

template<class T>
int ArrayList<T>::indexOf(const T& theElement)
const
{// Return index of first occurrence of theElement.
   // search for theElement
   int theIndex = (int) (find(element, element
   + listSize, theElement) - element);
   // check if theElement was found
   if (theIndex == listSize)
      return -1; // not found
   else return theIndex;
}
The Method erase

template<class T>
void arrayList<T>::erase(int theIndex)
{// Delete the element whose index is theIndex.
  checkIndex(theIndex);

  // valid index, shift elements with higher
  // index
  copy(element + theIndex + 1, element +
       listSize, element + theIndex);

  element[--listSize].~T(); // invoke destructor
}
The Method insert

template<class T>
void ArrayList<T>::insert(int theIndex,
                           const T& theElement)
{
   // Insert theElement.
   if (theIndex < 0 || theIndex > listSize)
     // invalid index
     // code to throw an exception comes here
   
   // valid index, make sure we have space
   if (listSize == arrayLength)
     // no space, double capacity
     changeLength1D(element, arrayLength,
                     2 * arrayLength);
   arrayLength *= 2;
}
The Method insert

// shift elements right one position
copy_backward(element + theIndex,
    element + listSize,
    element + listSize + 1);

    element[theIndex] = theElement;

    listSize++;
}
template<class T>
void arrayList<T>::output(ostream& out) const
{
    // Put the list into the stream out.
    copy(element, element + listSize,
         ostream_iterator<T>(out, "  "));
}
Overloading « »

// overload « »
template <class T>
ostream& operator<<(ostream& out,
                 const arrayList<T>& x)

{x.output(out); return out;}