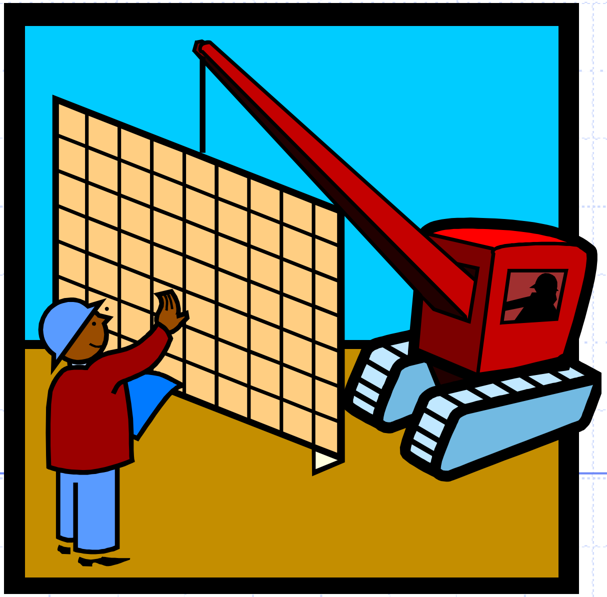
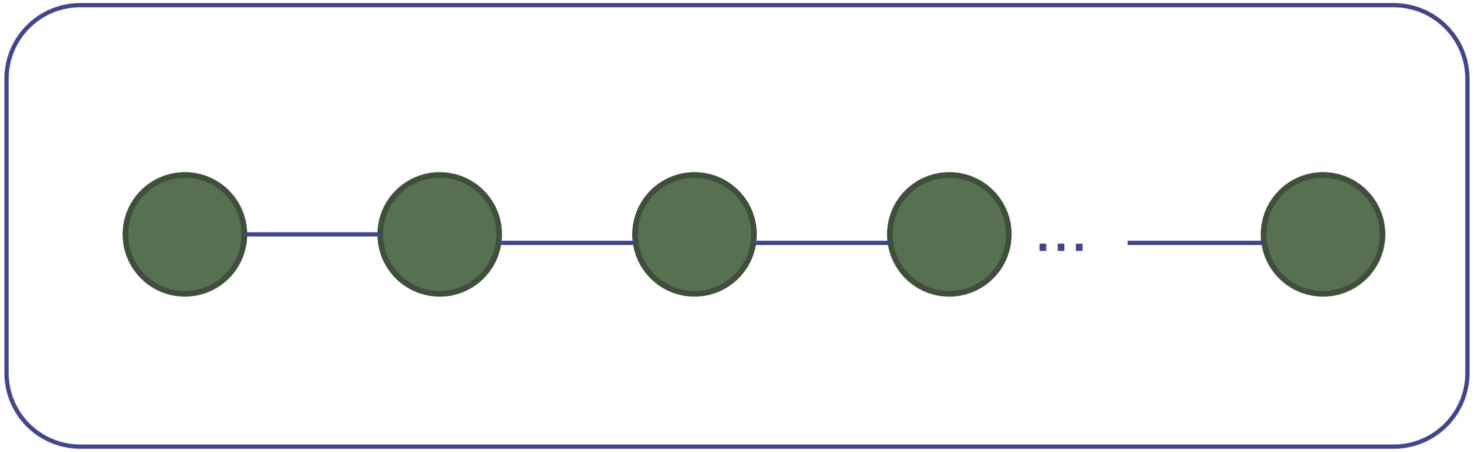
# Vector, List and Sequence



# Overview and Reading

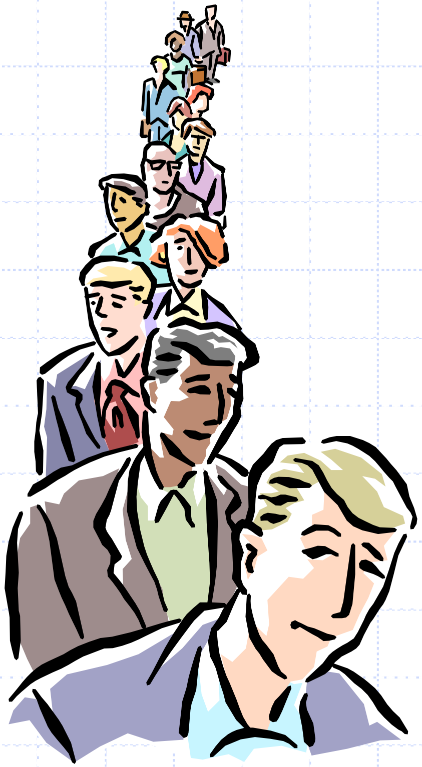
- Reading: Chapters: 6.1, 6.2, and 6.3
- A data structure that stores *n* elements in a linear order
  - Called list or sequence
- Didn't we learn "array" and "linked list"?
  - We are talking about more abstract ADTs than them



# Three ADTs

- Vector (also called Array List)
  - Access each element using a notion of index in [0,n-1]
  - Index of element e: the number of elements that are before e
  - Typically we use the "index" (e.g., [ ])
  - A more general ADT than "array"
- List
  - Not using an index to access, but use a node to access
  - Insert a new element e before some "position" p
  - A more general ADT than "linked list"
- Sequence
  - Can access an element as vector and list (using both index and position)
- (Note) Can implement the above ADTs using various ways
  - array, singly linked list, doubly linked list, circular linked list

# Vectors (or Array Lists)



# The Array List ADT

- The Vector or Array List ADT extends the notion of array by storing a sequence of objects
- An element can be accessed, inserted or removed by specifying its index (number of elements preceding it)
- An exception is thrown if an incorrect index is given (e.g., a negative index)

- Main methods:
  - at(integer i): returns the element at index i without removing it
  - set(integer i, object o): replace the element at index i with o
  - insert(integer i, object o): insert a new element o to have index i
  - erase(integer i): removes element at index i
- Additional methods:
  - size()
  - empty()

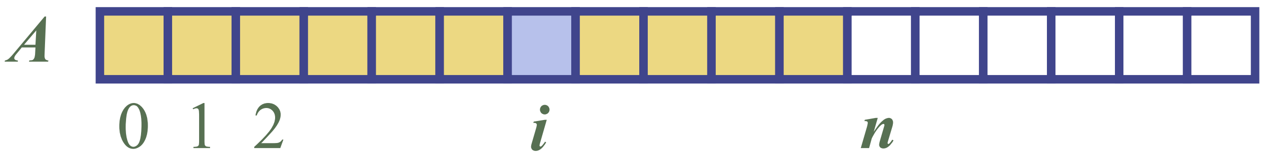
# Applications of Array Lists

- Direct applications
  - Sorted collection of objects (elementary database)
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

- Basically, every place where you can use “array”.

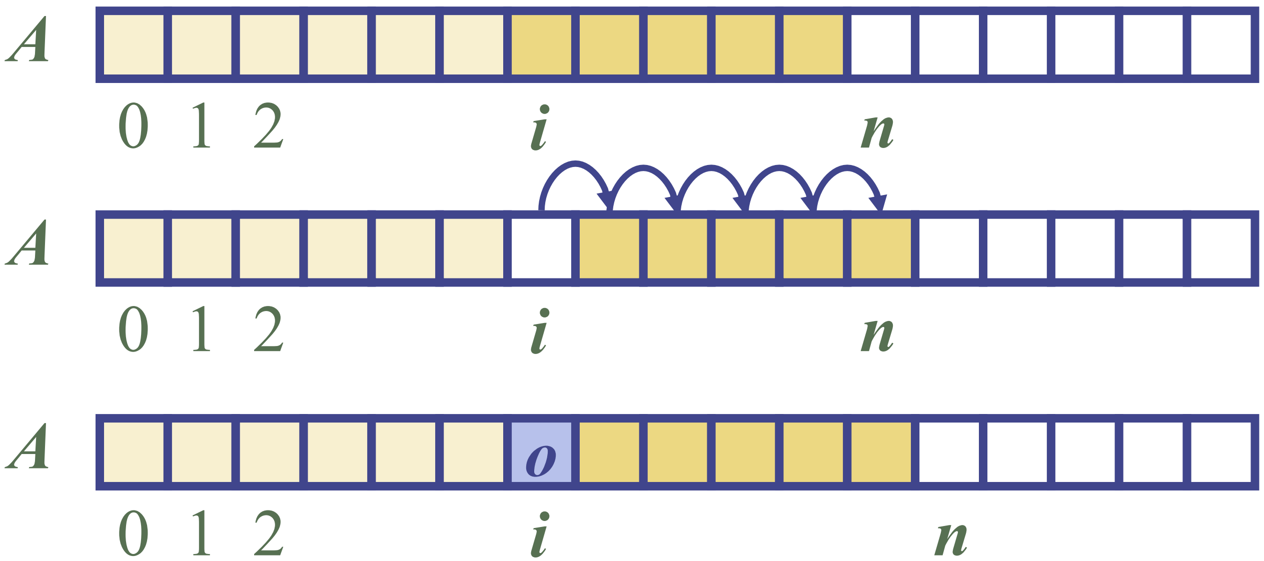
# Array-based Implementation of Vector

- Use an array $A$ of size $N$
- A variable $n$ keeps track of the size of the array list (number of elements stored)
- Operation ***at***($i$) is implemented in $O(1)$ time by returning $A[i]$
- Operation ***set***($i,o$) is implemented in $O(1)$ time by performing $A[i] = o$



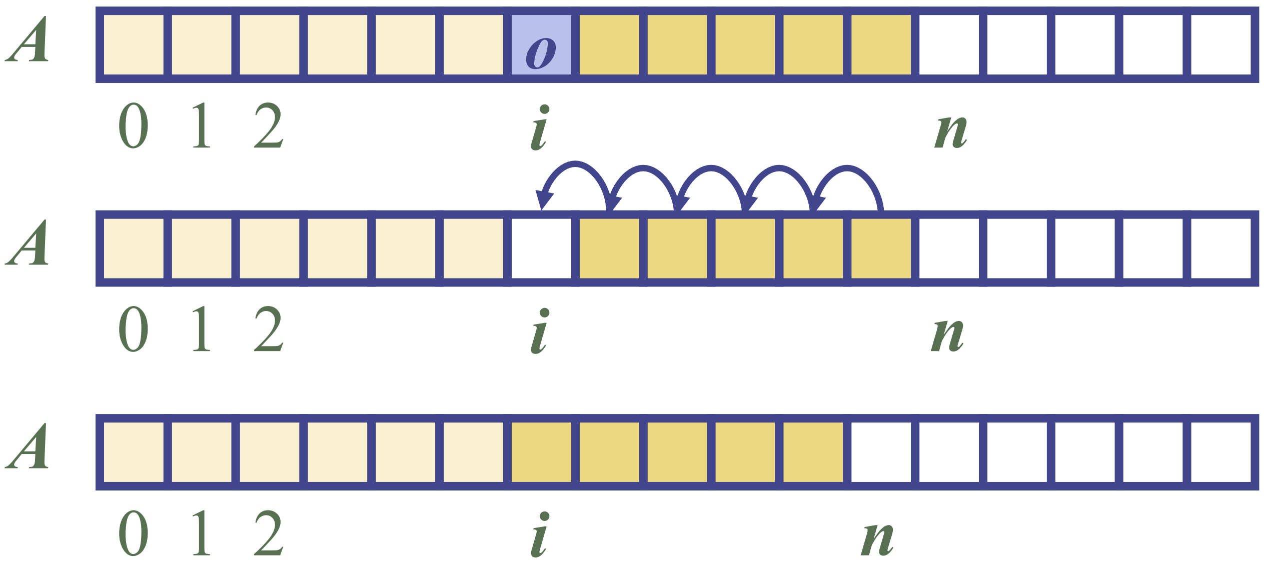
# Insertion

- In operation ***insert***$(i, o)$, we need to make room for the new element by shifting forward the $n - i$ elements $A[i], \ldots, A[n-1]$
- In the worst case ($i = 0$), this takes $O(n)$ time



# Element Removal

- In operation ***erase***(i), we need to fill the hole left by the removed element by shifting backward the $n - i - 1$ elements $A[i + 1], \ldots, A[n - 1]$
- In the worst case ($i = 0$), this takes $O(n)$ time



# Performance

- In the array-based implementation of an array list:
  - The space used by the data structure is $O(n)$
  - ***size***, ***empty***, ***at*** and ***set*** run in $O(1)$ time
  - ***insert*** and ***erase*** run in $O(n)$ time in worst case

- If we use the array in a circular fashion, operations ***insert***$(0, x)$ and ***erase***$(0, x)$ run in $O(1)$ time

- In an ***insert*** operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one

# Growable Array-based Array List

- In an insert(o) operation (without an index), we always insert at the end
- When the array is full, we replace the array with a larger one
- How large should the new array be?
  - Incremental strategy: increase the size by a constant $c$
  - Doubling strategy: double the size

```
Algorithm insert(o)
  if t = S.length − 1 then
    A ← new array of
        size …
    for i ← 0 to n−1 do
      A[i] ← S[i]
    S ← A
  n ← n + 1
  S[n−1] ← o
```

- For size n array, "re-grow" operation requires n copies

# Which is better?: Incremental or Doubling

- Comparison Method 1
  - Given the current size of S = n
  - Worst-case running time
    - Incremental strategy: O(1)
    - Doubling strategy: O(n)

- Are you happy?
  - Happy if your focus is really the worst-case
  - Unhappy
    - For doubling strategy, the total number of resizing array size would be small
- Can we reconsider the analysis method?

# Which is better?: Incremental or Doubling

- Comparison Method2
  - Compute the total time $T(n)$ needed to perform a series of $n$ insert(o) operations
  - Assume that we start with an empty stack represented by an array of size $1$

- We call amortized time of an insert operation the average time taken by an insert over the series of operations, i.e., $T(n)/n$
  - This can be a fairer comparison in some cases
- Amortized analysis (분할상환분석 in Wiki)

# Incremental Strategy Analysis

- We replace the old array with a new one $k = n/c$ times
- The total time $T(n)$ of a series of $n$ insert operations is proportional to

$$n + c + 2c + 3c + 4c + \ldots + kc =$$
$$n + c(1 + 2 + 3 + \ldots + k) =$$
$$n + ck(k + 1)/2$$

- Since $c$ is a constant, $T(n)$ is $O(n + k^2)$, i.e., $O(n^2)$
- The amortized time of an insert operation is $O(n)$

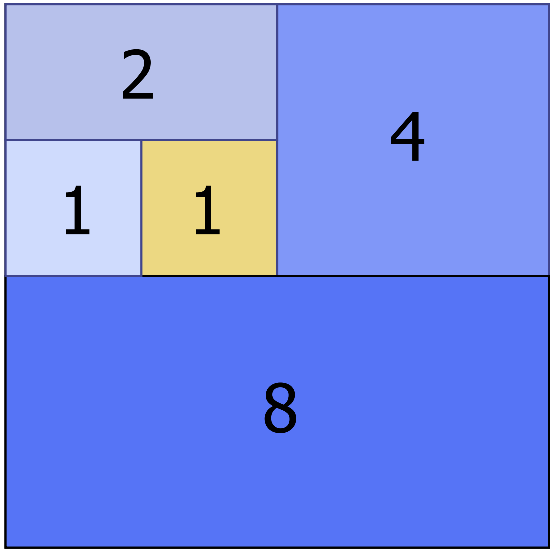
# Doubling Strategy Analysis

- We replace the old array with a new one $k = \log_2 n$ times
- The total time $T(n)$ of a series of $n$ insert operations is proportional to

$$n + 1 + 2 + 4 + 8 + \ldots + 2^k =$$
$$n + 2^{k+1} - 1 =$$
$$3n - 1$$

- $T(n)$ is $O(n)$
- The amortized time of an insert operation is $O(1)$

geometric series



# Professor, I have a question

- In "computing spans", why didn't you do amortized analysis?
- Can we do it?
- Is it meaningful?

- Think about this!
  - I am ready to discuss if you get your version of answer ready.

```
Algorithm spans2(X, n)                          #
    S ← new array of n integers                 n
    A ← new empty stack                         1
    for i ← 0 to n − 1 do                       n
        while (¬A.empty() ∧
                X[A.top()] ≤ X[i] ) do          n
            A.pop()                             n
        if A.empty() then                       n
            S[i] ← i + 1                        n
        else
            S[i] ← i − A.top()                  n
        A.push(i)                               n
    return S                                    1
```

# Vectors in C++ STL

```
#include <vector>                 // provides definition of vector
using std::vector;                // make vector accessible

vector<int> myVector(100);        // a vector with 100 integers
```

- **vector($n$)**: Construct a vector with space for $n$ elements; if no argument is given, create an empty vector.
- **size()**: Return the number of elements in $V$.
- **empty()**: Return true if $V$ is empty and false otherwise.
- **resize($n$)**: Resize $V$, so that it has space for $n$ elements.
- **reserve($n$)**: Request that the allocated storage space be large enough to hold $n$ elements.
- **operator[$i$]**: Return a reference to the $i$th element of $V$.
- **at($i$)**: Same as $V[i]$, but throw an out_of_range exception if $i$ is out of bounds, that is, if $i < 0$ or $i \geq V$.size().
- **front()**: Return a reference to the first element of $V$.
- **back()**: Return a reference to the last element of $V$.
- **push_back($e$)**: Append a copy of the element $e$ to the end of $V$, thus increasing its size by one.
- **pop_back()**: Remove the last element of $V$, thus reducing its size by one.

Difference between resize() and reserve()?

# Logistics

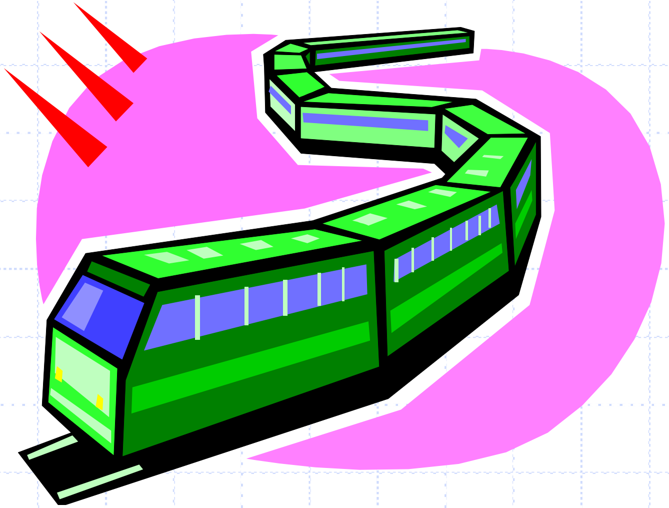
- First programming assignment
  - Deadline: Sep, 19th
- Problem Solving Homework
  - Deadline: Oct, 1st
- You should keep reading the textbook
- Sep 24th, 26th : No class
  - Thanksgiving

# Last Class

- Vector and List

- Vector
  - Access elements by “index”
  - Incremental vs. Doubling Strategy
    - Amortized analysis

# Lists



# (Node) List ADT

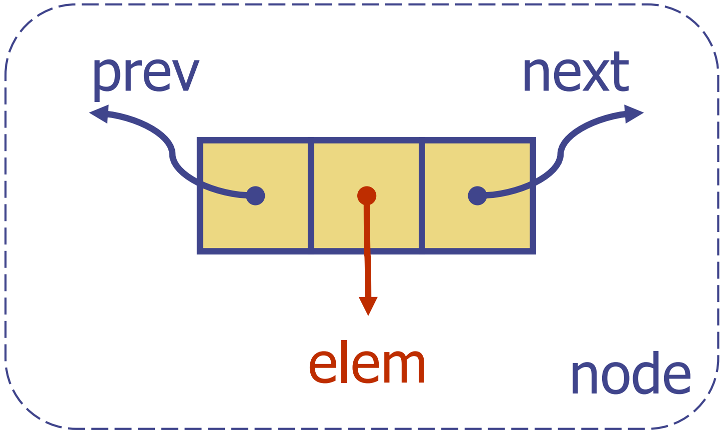
- The Node List ADT models a sequence of positions storing arbitrary objects
- It establishes a before/after relation between positions
- Generic methods:
  - size(), empty()

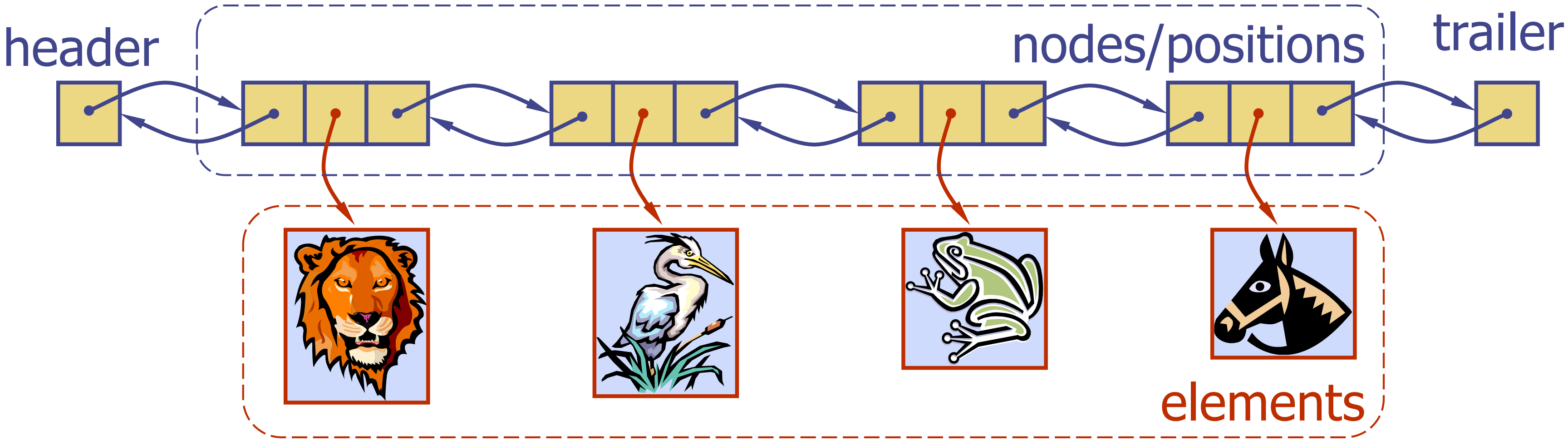
- Iterators:
  - begin(), end()
- Update methods:
  - insertFront(e), insertBack(e)
  - removeFront(), removeBack()
- Iterator-based update:
  - insert(p, e)
  - remove(p)

(Question) No method for accessing a specific node?
We will talk about this later

# Implementation based on DLL (covered this)

- A doubly linked list provides a natural implementation of the Node List ADT
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes



# Performance

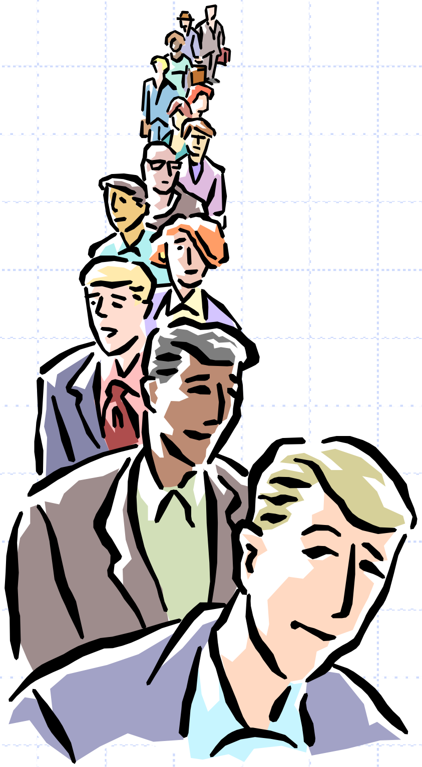
- In the implementation of the List ADT by means of a doubly linked list
  - The space used by a list with $n$ elements is $O(n)$
  - The space used by each position of the list is $O(1)$
  - All the operations of the List ADT run in $O(1)$ time

# Lists in C++ STL

```
#include <list>
using std::list;                    // make list accessible
list<float> myList;                 // an empty list of floats
```

- list($n$): Construct a list with $n$ elements; if no argument list is given, an empty list is created.
- size(): Return the number of elements in $L$.
- empty(): Return true if $L$ is empty and false otherwise.
- front(): Return a reference to the first element of $L$.
- back(): Return a reference to the last element of $L$.
- push_front($e$): Insert a copy of $e$ at the beginning of $L$.
- push_back($e$): Insert a copy of $e$ at the end of $L$.
- pop_front(): Remove the fist element of $L$.
- pop_back(): Remove the last element of $L$.

# Containers, Iterators, and Generic algorithms



# Sorting: Vector and List

- I want to find “yiyung” in Vector or List objects

```
vector<string>  V(100);
list<string> L(100);
// some data insertion to V and L

//Design 1: different function
find_vector(&V);
find_list(&L);

//Design 2: function overloading
find(&V);
find(&L);
```

Do you like these? Why? Why not?

# This is how we can do in C++

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

using namespace::std;

int main()
{
  vector<string> vec_str;
  vec_str.push_back("is");
  vec_str.push_back("of");
  vec_str.push_back("the");
  vec_str.push_back("hello");

  vector<string>::iterator it;

  it =
    find(vec_str.begin(), vec_str.end(), "the");
  cout << "Print: " << *it << endl;

  it++;
  cout << "Print: " << *it << endl;

  return 0;
}
```

```
#include <iostream>
#include <list>
#include <string>
#include <algorithm>

using namespace::std;

int main()
{
  list<string> list_str;
  list_str.push_back("is");
  list_str.push_back("of");
  list_str.push_back("the");
  list_str.push_back("hello");

  list<string>::iterator it;

  it =
    find(list_str.begin(), list_str.end(), "the");
  cout << "Print: " << *it << endl;

  it++;
  cout << "Print: " << *it << endl;

  return 0;
}
```

It is cool. But why is it cool?

# Mysterious things

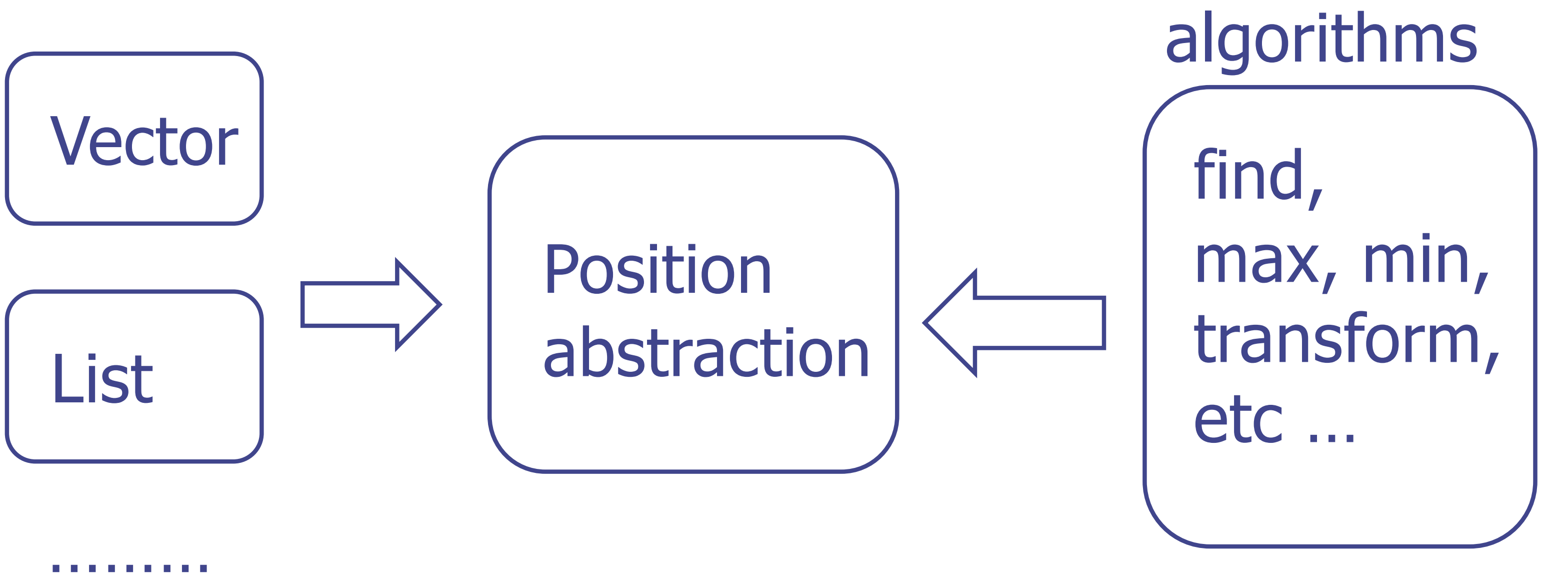
```cpp
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

using namespace::std;

int main()
{
  vector<string> vec_str;
  vec_str.push_back("is");
  vec_str.push_back("of");
  vec_str.push_back("the");
  vec_str.push_back("hello");

  vector<string>::iterator it;

  it =
    find(vec_str.begin(), vec_str.end(), "the");
  cout << "Print: " << *it << endl;

  it++;
  cout << "Print: " << *it << endl;

  return 0;
}
```

```cpp
#include <iostream>
#include <list>
#include <string>
#include <algorithm>

using namespace::std;

int main()
{
  list<string> list_str;
  list_str.push_back("is");
  list_str.push_back("of");
  list_str.push_back("the");
  list_str.push_back("hello");

  list<string>::iterator it;

  it =
    find(list_str.begin(), list_str.end(), "the");
  cout << "Print: " << *it << endl;

  it++;
  cout << "Print: " << *it << endl;

  return 0;
}
```

iterator? Looks like a "position" of vector or list. Hmm.....

# Goal and Design Challenge

- Lots of data structures (or classes in C++) that can contain various types of elements
  - "Container"
  - Examples: Vector, List, deque, set, map, etc …




- How are you going to design this concept?
  - Again, from C++ STL designer's perspective

# Position ADT

- The Position ADT models the notion of place within a data structure where a single object is stored
- It gives a unified view of diverse ways of storing data, such as
  - a cell of an array
  - a node of a linked list
- "A" method of accessing the element at position p:
  - object p.element(): returns the element at position
  - In C++ it is convenient to implement this as *p
    - Operator overloading

- Implemented as "iterator" in C++

# Containers and Iterators in C++

- An iterator abstracts the process of scanning through a collection of elements
- A container is an abstract data structure that supports element access through iterators
  - Data structures that support iterators
  - Examples include Stack, Queue, Vector, List
  - begin(): returns an iterator to the first element
  - end(): return an iterator to an imaginary position just after the last element
- An iterator behaves like a pointer to an element
  - *p: returns the element referenced by this iterator
  - ++p: advances to the next element
- Extends the concept of position by adding a traversal capability

# Example codes again

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

using namespace::std;

int main()
{
  vector<string> vec_str;
  vec_str.push_back("is");
  vec_str.push_back("of");
  vec_str.push_back("the");
  vec_str.push_back("hello");

  vector<string>::iterator it;

  it =
    find(vec_str.begin(), vec_str.end(), "the");
  cout << "Print: " << *it << endl;

  it++;
  cout << "Print: " << *it << endl;

  return 0;
}
```

```
#include <iostream>
#include <list>
#include <string>
#include <algorithm>

using namespace::std;

int main()
{
  list<string> list_str;
  list_str.push_back("is");
  list_str.push_back("of");
  list_str.push_back("the");
  list_str.push_back("hello");

  list<string>::iterator it;

  it =
    find(list_str.begin(), list_str.end(), "the");
  cout << "Print: " << *it << endl;

  it++;
  cout << "Print: " << *it << endl;

  return 0;
}
```

Ah-ha, it's an iterator!

# Various Iterators

- (standard) iterator: allows read-write access to elements
- const iterator: provides read-only access to elements
- bidirectional iterator: supports both ++p and –p
- random-access iterator: supports both p+i and p-i

# STL Iterators in C++

- Each STL container type C supports iterators:
  - C::iterator – read/write iterator type
  - C::const_iterator – read-only iterator type
  - C.begin(), C.end() – return start/end iterators
- This iterator-based operators and methods:
  - *p: access current element
  - ++p, --p: advance to next/previous element
  - C.assign(p, q): replace C with contents referenced by the iterator range [p, q) (from p up to, but not including, q)
  - insert(p, e): insert e prior to position p
  - erase(p): remove element at position p
  - erase(p, q): remove elements in the iterator range [p, q)

# Back to Iterator: STL Iterator-based Functions

| | |
|---|---|
| vector$(p,q)$: | Construct a vector by iterating between $p$ and $q$, copying each of these elements into the new vector. |
| assign$(p,q)$: | Delete the contents of $V$, and assigns its new contents by iterating between $p$ and $q$ and copying each of these elements into $V$. |
| insert$(p,e)$: | Insert a copy of $e$ just prior to the position given by iterator $p$ and shifts the subsequent elements one position to the right. |
| erase$(p)$: | Remove and destroy the element of $V$ at the position given by $p$ and shifts the subsequent elements one position to the left. |
| erase$(p,q)$: | Iterate between $p$ and $q$, removing and destroying all these elements and shifting subsequent elements to the left to fill the gap. |
| clear(): | Delete all these elements of $V$. |

# STL Containers and Algorithms

#include <algorithm>

| | |
|---|---|
| sort$(p,q)$: | Sort the elements in the range from $p$ to $q$ in ascending order. It is assumed that less-than operator ("<") is defined for the base type. |
| random_shuffle$(p,q)$: | Rearrange the elements in the range from $p$ to $q$ in random order. |
| reverse$(p,q)$: | Reverse the elements in the range from $p$ to $q$. |
| find$(p,q,e)$: | Return an iterator to the first element in the range from $p$ to $q$ that is equal to $e$; if $e$ is not found, $q$ is returned. |
| min_element$(p,q)$: | Return an iterator to the minimum element in the range from $p$ to $q$. |
| max_element$(p,q)$: | Return an iterator to the maximum element in the range from $p$ to $q$. |
| for_each$(p,q,f)$: | Apply the function $f$ the elements in the range from $p$ to $q$. |

STL Vector
STL deque

STL List

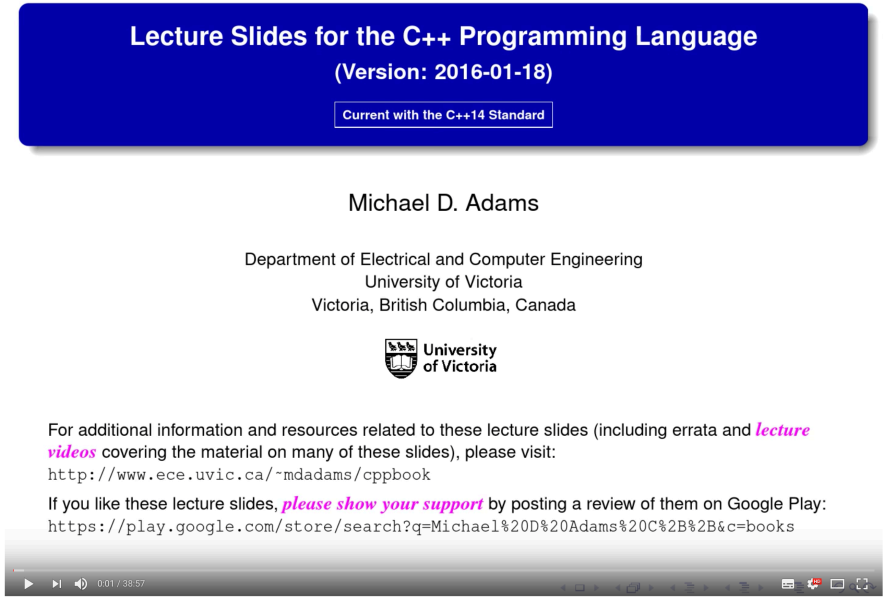
http://www.cplusplus.com/reference/algorithm/

# Example Code

```
#include <cstdlib>                                  // provides EXIT_SUCCESS
#include <iostream>                                 // I/O definitions
#include <vector>                                   // provides vector
#include <algorithm>                                // for sort, random_shuffle

using namespace std;                                // make std:: accessible

int main () {
  int a[] = {17, 12, 33, 15, 62, 45};
  vector<int> v(a, a + 6);                          // v: 17 12 33 15 62 45
  cout << v.size() << endl;                         // outputs: 6
  v.pop_back();                                     // v: 17 12 33 15 62
  cout << v.size() << endl;                         // outputs: 5
  v.push_back(19);                                  // v: 17 12 33 15 62 19
  cout << v.front() << " " << v.back() << endl;     // outputs: 17 19
  sort(v.begin(), v.begin() + 4);                   // v: (12 15 17 33) 62 19
  v.erase(v.end() − 4, v.end() − 2);                // v: 12 15 62 19
  cout << v.size() << endl;                         // outputs: 4

  char b[] = {'b', 'r', 'a', 'v', 'o'};
  vector<char> w(b, b + 5);                         // w: b r a v o
  random_shuffle(w.begin(), w.end());               // w: o v r a b
  w.insert(w.begin(), 's');                         // w: s o v r a b

  for (vector<char>::iterator p = w.begin(); p != w.end(); ++p)
    cout << *p << " ";                              // outputs: s o v r a b
  cout << endl;
  return EXIT_SUCCESS;
}
```

# If you want to know more about iterators,

- Please watch this video

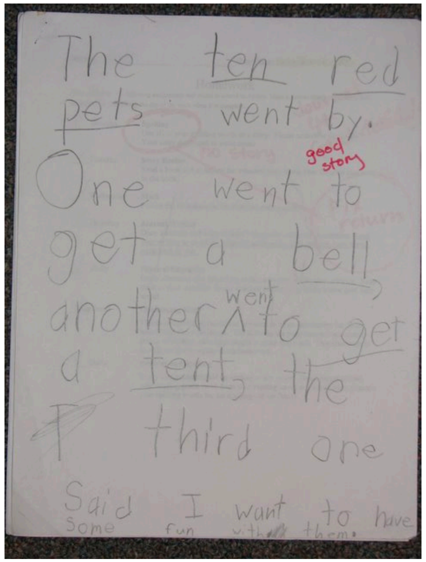


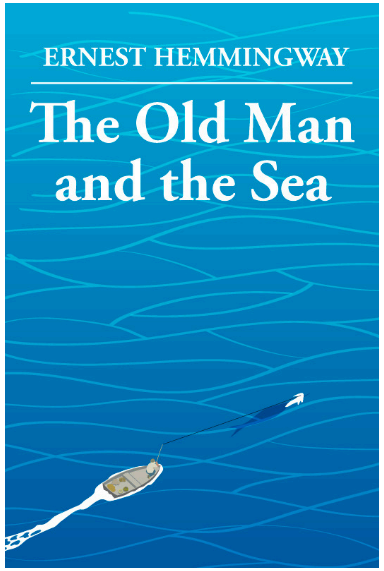

https://www.youtube.com/watch?v=TxufBysSPK0

- Please
  - I hate to answer the question “Is this included in the exam?”

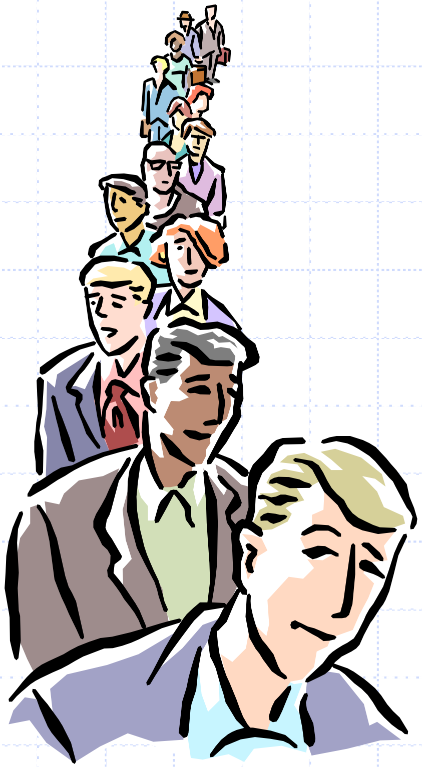
# What should be your next question?

- Can I implement iterators in C++, in addition to just knowing how to use them?
  - Someone like the C++ STL designer
- Ch 6.2.3: Some level of explanation:
  - Beyond the topic of this class
- I will be happy to discuss this if you visit my office.



# Sequences



# Sequence ADT

- The Sequence ADT is the union of the Array List and Node List ADTs
- Elements accessed by
  - Index, or
  - Position
- Generic methods:
  - size(), empty()
- ArrayList-based methods:
  - at(i), set(i, o), insert(i, o), erase(i)
- List-based methods:
  - begin(), end()
  - insertFront(o), insertBack(o)
  - eraseFront(), eraseBack()
  - insert (p, o), erase(p)
- Bridge methods:
  - atIndex(i), indexOf(p)

# Applications of Sequences

- The Sequence ADT is a basic, general-purpose, data structure for storing an ordered collection of elements
- Direct applications:
  - Generic replacement for stack, queue, vector, or list
  - small database (e.g., address book)
- Indirect applications:
  - Building block of more complex data structures

# Questions?