

# CSCI-1200 Computer Science II — Fall 2008

## Lecture 9 — Lists

### Review from Lecture 8

- We wrote several versions of a program to maintain a class enrollment list and an associated waiting list.
  - The first version used vectors to store the information. Unfortunately, erasing items from vectors is inefficient.
  - In the second version, we explored iterators and iterator operations as a different means of manipulating the contents of the vector.
  - This allows us to replace the vector with a list in the third version. There is an `erase` function for both vectors and lists. The vector erase function does pretty much what we did in our enrollment example program. The list erase function is much more efficient (more on this next week!).
- For the enrollment problem, the list is a better sequential container class than the vector.

### Today's Class

- Returning references to member variables from member functions
- Review of iterators and iterator operations
- STL Lists, `erase` and `insert` on lists
- Differences between indices and iterators, differences between lists and vectors
- Introductory example on linked lists.
- Basic linked list operations:
  - Stepping through a list
  - Push back
  - ... & we'll continue on Tuesday

### 9.1 References and Return Values

- A reference is an *alias* for another variable. For example:

```
string a = "Tommy";
string b = a;      // a new string is created using the string copy constructor
string& c = a;    // c is an alias/reference to the string object a

b[1] = 'i';
cout << a << " " << b << " " << c << endl;    // outputs: Tommy Timmy Tommy

c[1] = 'a';
cout << a << " " << b << " " << c << endl;    // outputs: Tammy Timmy Tammy
```

The reference variable `c` refers to the same string as variable `a`. Therefore, when we change `c`, we change `a`.

- Exactly the same thing occurs with reference parameters to functions and the return values of functions. Let's look at the `Student` class from Lecture 4 again:

```
class Student {
public:
    const string& first_name() const { return first_name_; }
    const string& last_name() const { return last_name_; }
    // etc....

private:
    string first_name_;
    string last_name_;
    // etc...
};
```

- In the main function we had a vector of students:

```
vector<Student> students;
```

Based on our discussion of references above and looking at the class declaration, what if we wrote the following. Would the code then be changing the internal contents of the i-th Student object?

```
string & fname = students[i].first_name();
fname[1] = 'i'
```

- The answer is NO! The `Student` class member function `first_name` returns a **const** reference. The compiler will complain that the above code is attempting to assign a const reference to a non-const reference variable.
- If we instead wrote the following, then compiler would complain that you are trying to change a const object.

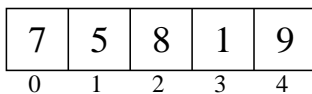
```
const string & fname = students[i].first_name();
fname[1] = 'i'
```

- Hence in both cases the `Student` class would be “safe” from attempts at external modification.
- However, the author of the `Student` class would get into trouble if the member function return type was only a reference, and not a const reference. Then external users could access and change the internal contents of an object! This is a bad idea in most cases.

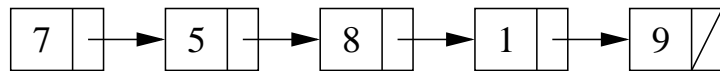
## 9.2 The list Standard Library Container Class

- Lists are formed as a sequentially linked structure instead of the array-like, random-access / indexing structure of vectors.

array/vector:



list:



- Lists have `push_front` and `pop_front` functions in addition to the `push_back` and `pop_back` functions of vectors.
- Erase is very efficient for a list, independent of the size of the list (we’ll see why when we learn the implementation details later in the semester).
- We can’t use the standard `sort` function; we must use a special `sort` function defined by the list type.
- Lists have no subscripting operation (a.k.a. they do not allow “random-access”).

## 9.3 Iterators and Iterator Operations — General

- An iterator type is defined by each container class. For example,

```
vector<double>::iterator v_itr;
list<string>::iterator l_itr;
string::iterator s_itr;
```

- An iterator is assigned to a specific location in a container. For example:

Note: We can add an integer to vector and string iterators, but not to list iterators.

```
v_itr = vec.begin() + i; // i-th location in a vector
l_itr = lst.begin();    // first entry in a list
s_itr = str.begin();    // first char of a string
```

- The contents of the specific entry referred to by an iterator are accessed using the *\* dereference operator*. In the first and third lines, `*v_itr` and `*l_itr` are l-values. In the second, `*s_itr` is an r-value.

```
*v_itr = 3.14;
cout << *s_itr << endl;
*l_itr = "Hello";
```

- Stepping through a container, either forward and backward, is done using increment (++) and decrement (--) operators:

```
++itr; itr++; --itr; itr--;
```

These operations move the iterator to the next and previous locations in the vector, list, or string. The operations do not change the contents of container!

- Finally, we can change the container that a specific iterator is attached to **as long as the types match**. Thus, if `v` and `w` are both `vector<double>`, then the code:

```
v_itr = v.begin();
*v_itr = 3.14; // changes 1st entry in v
v_itr = w.begin() + 2;
*v_itr = 2.78; // changes 3rd entry in w
```

works fine because `v_itr` is a `vector<double>::iterator`, but if `a` is a `vector<string>` then

```
v_itr = a.begin();
```

is a syntax error because of a type clash!

## 9.4 Iterators and Iterator Operations — Vector Iterators

Vector (and string) iterators have special capabilities that most other container iterators do not have:

- Initialization at a random spot in the vector:

```
p = v.begin() + i;
```

- Jumping around inside the vector through addition and subtraction of location counts:

```
p = p + 5;
```

moves `p` 5 locations further in the vector.

- Neither of these is allowed for list iterators (and most other iterators, for that matter) because of the way the corresponding containers are built.

## 9.5 Iterators vs. Indices for Vectors and Strings

- Students are often confused by the difference between iterators and indices for vectors. Consider the following declarations:

```
vector<double> a(10, 2.5);
vector<double>::iterator p = a.begin() + 5;
unsigned int i=5;
```

- Iterator `p` refers to location 5 in vector `a`. The value stored there is directly accessed through the `*` operator:

```
*p = 6.0;
cout << *p << endl;
```

- The above code has **changed the contents** of vector `a`. Here's the equivalent code using subscripting:

```
a[i] = 6.0;
cout << a[i] << endl;
```

## 9.6 Lists vs. Vectors

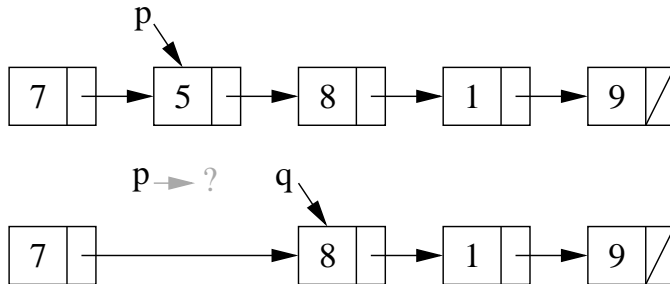
- Lists are a chain of separate memory blocks, one block for each entry.
- Vectors are formed as a contiguous (and bigger) block of memory.
- Lists therefore allow easy/fast insert and remove in the middle, but not indexing.
- Vectors therefore allow indexing (which depends on jumping around inside the block of memory), but slow insert and remove in the middle.

## 9.7 Erase

- Lists and vectors each have a special member function called `erase`. In particular, given list of ints `s`, consider the example:

```
list<int>::iterator p = s.begin();
++p;
list<int>::iterator q = s.erase(p);
```

- After the code above is executed:
  - The integer stored in the second entry of the list has been removed.
  - The size of the list has shrunk by one.
  - The iterator `p` does not refer to a valid entry.
  - The iterator `q` refers to the item that was the third entry and is now the second.



- To reuse the iterator `p` and make it a valid entry, you will often see the code written:

```
list<int>::iterator p = s.begin();
++p;
p = s.erase(p);
```

- Now we can rewrite the `erase_from_vector` function from the Lecture 8 enrollment example:

```
p = v.erase(p);
```

- Even though this has the same syntax for vectors and for list, the vector version is  $O(n)$ , whereas the list version is  $O(1)$ .

## 9.8 Insert

- Similarly, there is an `insert` function for lists that takes an iterator and a value and adds a link in the chain with the new value immediately before the item pointed to by the iterator.
- The call returns an iterator that points to the newly added element. Variants on the basic insert function are also defined.

## 9.9 Exercise: Erase & Insert

Write a function that takes a list of integers, `lst`, and an integer, `x`. The function should 1) remove all negative numbers from the list, 2) verify that the remaining elements in the list are sorted in increasing order, and 3) insert `x` into the list such that the order is maintained.

## 9.10 Motivation

- Thus far our discussion of how `list<T>` is implemented has been only intuitive: it is a “chain” of objects.
- Now we will look at the mechanism — *linked lists*.
- Learning this mechanism is good background for higher-level courses where the design of novel data structures is important.

## 9.11 Objects with Pointers / Linking Objects

- The two fundamental mechanisms of linked lists are:
  - creating objects with pointers as one of the member variables, and
  - making these pointers point to other objects of the same type.
- These mechanisms are illustrated in the following program:

```
#include <iostream>
using namespace std;

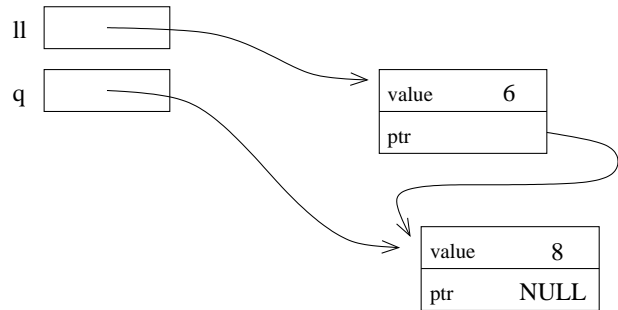
template <class T>
class Node {
public:
    T value;
    Node* ptr;
};

void main() {
    Node<int>* ll;      // ll is a pointer to a (non-existent) Node
    ll = new Node<int>; // Create a Node and assign its memory address to ll
    ll->value = 6;     // This is the same as (*ll).value = 6;
    ll->ptr = NULL;    // NULL == 0, which indicates a "null" pointer

    Node<int>* q = new Node<int>;
    q->value = 8;
    q->ptr = NULL;

    // set ll's ptr member variable to
    // point to the same thing as variable q
    ll->ptr = q;

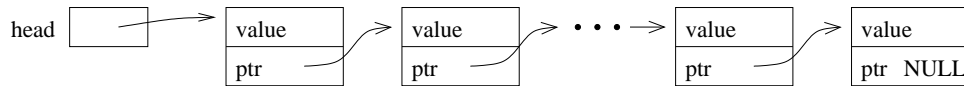
    cout << "1st value: " << ll->value << "\n"
         << "2nd value: " << ll->ptr->value << endl;
}
```



## 9.12 Definition: A Linked List

- The definition is recursive: A linked list is either:
  - Empty, or
  - Contains a node storing a value and a pointer to a linked list.
- The first node in the linked list is called the *head* node and the pointer to this node is called the *head* pointer. The pointer's value will be stored in a variable called *head*.

## 9.13 Visualizing Linked Lists



- The **head** pointer variable is drawn with its own box. It is an individual variable. It is important to have a separate pointer to the first node, since the “first” node may change.
- The objects (nodes) that have been dynamically allocated and stored in the linked lists are shown as boxes, with arrows drawn to represent pointers.
  - Note that this is a conceptual view only. The memory locations could be anywhere, and the actual values of the memory addresses aren’t usually meaningful.
- The last node **MUST** have NULL for its pointer value — you will have all sorts of trouble if you don’t ensure this!
- You should make a habit of drawing pictures of linked lists to figure out how to do the operations.

## 9.14 Basic Mechanisms: Stepping Through the List

- We’d like to write a function to determine if a particular value, stored in **x**, is also in the list.
- You can think of this as a precursor to the **find** function. Our function isn’t yet returning an iterator, however.
- We can access the entire contents of the list, one step at a time, by starting just from the **head** pointer.
  - We will need a separate, local pointer variable to point to nodes in the list as we access them.
  - We will need a loop to step through the linked list (using the pointer variable) and a check on each value.

## 9.15 Exercise: Write `is_there`

```
template <class T> bool is_there(Node<T>* head, const T& x) {
```

## 9.16 Basic Mechanisms: Pushing on the Back

- Goal: place a new node at the end of the list.
- We must step to the end of the linked list, remembering the pointer to the last node.
  - This is an  $O(n)$  operation and is a major drawback to the ordinary linked-list data structure we are discussing now. We will correct this drawback by creating a slightly more complicated linking structure in our next lecture.
- We must create a new node and attach it to the end.
- We must remember to update the **head** pointer variable’s value if the linked list is initially empty.
  - Hence, in writing the function, we must pass the pointer variable **by reference**.

## 9.17 Exercise: Write `push_back`

```
template <class T> void push_back( Node<T>* & head, T const& value ) {
```