

CSCI-1200 Data Structures — Spring 2020

Lecture 9 — Iterators & STL Lists

Review from Lecture 8

- Algorithm Analysis, Formal Definition of Order Notation
- Another `vector` operation: `pop_back`
- *Erasing items* from vectors is inefficient!
- Iterators and iterator operations

Today

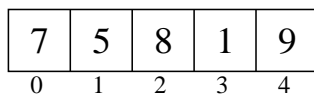
- STL `lists` are a different sequential container class.
- Returning references to member variables from member functions
- `Vec` iterator implementation
- Quick review of iterators
- Implementation of iterators in our homemade `Vec` class (from Lecture 7)
- `const` and reference on return values

Optional Reading: Ford & Topp Ch 6; Koenig & Moo, Sections 5.1-5.5

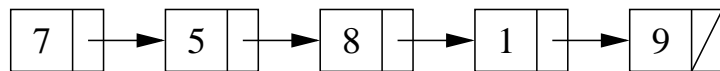
9.1 A New Datatype: The `list` Standard Library Container Class

- Lists are our second standard-library container class. (Vectors were the first.) Both lists & vectors store sequential data that can shrink or grow.
- However, the use of memory is fundamentally different. Vectors are formed as a single contiguous array-like block of memory. Lists are formed as a sequentially linked structure instead.

array/vector:



list:



- Although the interface (functions called) of lists and vectors and their iterators are quite similar, their implementations are VERY different. Clues to these differences can be seen in the operations that are NOT in common, such as:
 - STL `vectors` / arrays allow “random-access” / indexing / `[]` subscripting. We can immediately jump to an arbitrary location within the vector / array.
 - STL `lists` have no subscripting operation (we can’t use `[]` to access data). The only way to get to the middle of a list is to follow pointers one link at a time.
 - Lists have `push_front` and `pop_front` functions in addition to the `push_back` and `pop_back` functions of vectors.
 - `erase` and `insert` in the middle of the STL `list` is very efficient, independent of the size of the list. Both are implemented by rearranging pointers between the small blocks of memory. (We’ll see this when we discuss the implementation details next week).
 - We can’t use the same STL `sort` function we used for `vector`; we must use a special `sort` function defined by the STL `list` type.

```
std::vector<int> my_vec;
std::list<int> my_lst;
// ... put some data in my_vec & my_lst
std::sort(my_vec.begin(), my_vec.end(), optional_compare_function);
my_lst.sort(optional_compare_function);
```

Note: STL `list` `sort` member function is just as efficient, $O(n \log n)$, and will also take the same optional compare function as STL `vector`.

- Several operations invalidate the values of vector iterators, but not list iterators:
 - * `erase` invalidates all iterators after the point of erasure in vectors;
 - * `push_back` and `resize` invalidate ALL iterators in a vector
- The value of any associated vector iterator must be re-assigned / re-initialized after these operations.

9.2 Exercise: Revising the Class List Program to Use Lists (& Iterators)

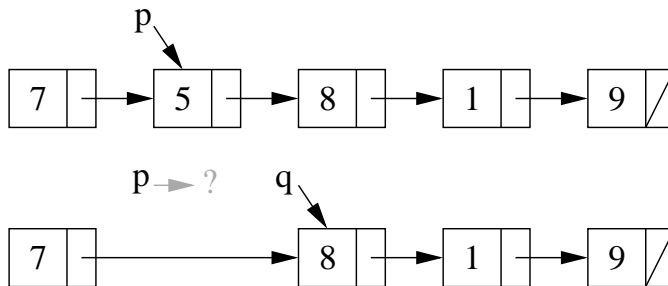
Now let's further modify the program to use lists instead of vectors. Because we've already switched to iterators, this change will be relatively easy. And now the program will be more efficient!

9.3 Erase & Iterators

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

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

```
std::list<int>::iterator p = s.begin();
++p;
p = s.erase(p);
```
- Even though the `erase` function has the same syntax for vectors and for list, the vector version is $O(n)$, whereas the list version is $O(1)$.

9.4 Insert

- Similarly, there is an `insert` function for STL 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.

9.5 Exercise: Using STL list Erase & Insert

Write a function that takes an STL `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.

9.6 Implementing Vec<T> Iterators

- Let's add iterators to our Vec<T> class declaration from last lecture:

```
public:
    // TYPEDEFS
    typedef T* iterator;
    typedef const T* const_iterator;

    // MODIFIERS
    iterator erase(iterator p);

    // ITERATOR OPERATIONS
    iterator begin() { return m_data; }
    const_iterator begin() const { return m_data; }
    iterator end() { return m_data + m_size; }
    const_iterator end() const { return m_data + m_size; }
```

- First, remember that `typedef` statements create custom, alternate names for existing types. `Vec<int>::iterator` is an iterator type defined by the `Vec<int>` class. It is just a `T *` (an `int *`). Thus, internal to the declarations and member functions, `T*` and `iterator` **may be used interchangeably**.
- Because the underlying implementation of `Vec` uses an array, and because pointers *are* the “iterator”s of arrays, the implementation of vector iterators is quite simple. *Note: the implementation of iterators for other STL containers is more involved!*
- Thus, `begin()` returns a pointer to the first slot in the `m_data` array. And `end()` returns a pointer to the “slot” just beyond the last legal element in the `m_data` array (as prescribed in the STL standard).
- Furthermore, dereferencing a `Vec<T>::iterator` (dereferencing a pointer to type `T`) correctly returns one of the objects in the `m_data`, an object with type `T`.
- And similarly, the `++`, `--`, `<`, `==`, `!=`, `>=`, etc. operators on pointers automatically apply to `Vec` iterators.
- The `erase` function requires a bit more attention. We've implemented the core of this function above. The STL standard further specifies that the return value of `erase` is an iterator pointing to the new location of the element just after the one that was deleted.
- Finally, note that after a `push_back` or `erase` or `resize` call some or all iterators referring to elements in that vector may be *invalidated*. Why? You must take care when designing your program logic to avoid invalid iterator bugs!

9.7 Review: Iterators and Iterator Operations

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

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

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

```
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
```

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

- 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 `std::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 `std::vector<double>::iterator`, but if `a` is a `std::vector<std::string>` then

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

is a syntax error because of a type clash!

9.8 Additional Iterator Operations for Vector (& String) Iterators

- Initialization at a random spot in the vector:

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

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

```
v_itr = v_itr + 5;
```

moves `p` 5 locations further in the vector. These operations are constant time, $O(1)$ for vectors.

- These operations are *not allowed* for list iterators (and most other iterators, for that matter) because of the way the corresponding containers are built. These operations would be linear time, $O(n)$, for lists, where n is the number of slots jumped forward/backward. Thus, they are not provided by STL for lists.
- Students are often confused by the difference between iterators and indices for vectors. Consider the following declarations:

```
std::vector<double> a(10, 2.5);
std::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;
```

- Here's another common confusion:

```
std::list<int> lst;      lst.push_back(100); lst.push_back(200);
                        lst.push_back(300); lst.push_back(400); lst.push_back(500);

std::list<int>::iterator itr,itr2,itr3;
itr = lst.begin();// itr is pointing at the 100
++itr;           // itr is now pointing at 200
*itr += 1;       // 200 becomes 201
// itr += 1;     // does not compile!  can't advance list iterator like this

itr = lst.end(); // itr is pointing "one past the last legal value" of lst
itr--;           // itr is now pointing at 500;
itr2 = itr--;    // itr is now pointing at 400, itr2 is still pointing at 500
itr3 = --itr;    // itr is now pointing at 300, itr3 is also pointing at 300
```

```
// dangerous: decrementing the begin iterator is "undefined behavior"
// (similarly, incrementing the end iterator is also undefined)
// it may seem to work, but break later on this machine or on another machine!
itr = lst.begin();
itr--; // dangerous!
itr++;
assert (*itr == 100); // might seem ok... but rewrite the code to avoid this!
```

9.9 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_; }
private:
    string first_name_;
    string last_name_;
};
```

- 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.

9.10 Working towards *our own* version of the STL list

- Our discussion of how the STL `list<T>` is implemented has been intuitive: it is a “chain” of objects.
- Now we will study the underlying mechanism — *linked lists*.
- This will allow us to build custom classes that mimic the STL `list` class, and add extensions and new features (more in the next couple lectures!).

9.11 Objects with Pointers, Linking Objects Together

- 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:

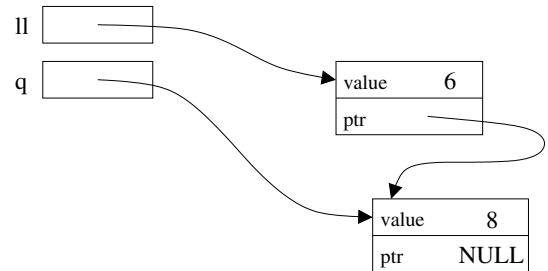
```
template <class T>
class Node {
public:
    T value;
    Node* ptr;
};

int 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;

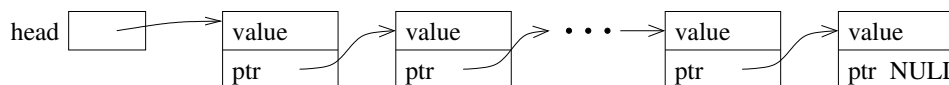
    cout << "1st value: " << ll->value << "\n"
         << "2nd value: " << ll->ptr->value << endl;
}
```



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.