Review from Lecture 6

- 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.

- 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
- Lists
- Review of iterators and iterator operations
- Differences between indices and iterators
- Differences between lists and vectors
- Prime number programming example

7.1 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 6 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:

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

Based on our discussion of references above and looking at the class declaration, what if we wrote:

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

Would the code then be changing the internal contents of the i-th Student object?

• 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:

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

Then compiler would complain that you are trying to change a const object.

• 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.

### 7.2 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:  
\[
\begin{array}{cccccc}
7 & 5 & 8 & 1 & 9 \\
0 & 1 & 2 & 3 & 4
\end{array}
\]
```

```
list:  
\[
\begin{array}{c}
7 \\
5 \\
8 \\
1 \\
9
\end{array}
\]
```

• 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”).

### 7.3 Iterators and Iterator Operations — General

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

```cpp
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:

```cpp
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:

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

In the first and third lines, *v_itr and *l_itr are l-values. In the second, *s_itr is an r-value.

• 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:

```cpp
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

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

is a syntax error because of a type clash!

### 7.4 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 containers are built.

### 7.5 Iterators vs. Indices for Vectors and Strings

Students are often confused by the difference between iterators and indices for vectors.

• Consider the following declarations:

```cpp
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;
```

This has changed the contents of vector a.

• And here's the equivalent code using subscripting:

```
a[i] = 6.0;
cout << a[i] << endl;
```
7.6 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.

7.7 Erase

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

  ```cpp
  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:

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

- Now we can rewrite the `erase_from_vector` function from the Lecture 7 enrollment example:

  ```cpp
  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 $O(1)$. 
7.8 Prime Numbers: Sieve of Eratosthenes

- We will explore the problem of finding all primes less than a given integer, \( n \), and introduce the Sieve of Eratosthenes algorithm.

- The algorithm is a “casting out” algorithm: each new prime is used to cast out all of its multiples from a list of potential primes.

- Finish the skeleton code below which uses lists and iterators.

- Why did we choose to use a list rather than a vector?

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

int main() {

    int n;
    cout << "Enter the upper bound on the set of primes you are interested in: ";
    cin >> n;

    list<int> primes;

    // Initialize with everything from 2 to n
    for (unsigned int i=2; i<=n; ++i)
        primes.push_back(i);

    // p will indicate the current prime
    list<int>::iterator p = primes.begin();

    // step through primes list until it is exhausted
    while (p != primes.end()) {
        // throw out all the numbers that are divisible by the current prime

    }

    // output out all the primes
    cout << primes.size() << " primes: " << endl;
    for (p = primes.begin(); p != primes.end(); p++) {
        cout << *p << " ";
    }
    cout << endl;

    return 0;
}
```
#include <algorithm>
#include <cmath>
#include <iomanip>
#include <iostream>
#include <vector>
using namespace std;

int main() {

    // Input the scores
    cout << "Enter each of the grades, followed by end-of-file: " << endl;
    vector<int> scores;
    int x;
    while (cin >> x) {
        scores.push_back(x);
    }

    // Quit with an error message if too few scores.
    if (scores.size() == 0) {
        cout << "No scores entered. Please try again!" << endl;
        return 1;
    }

    // Sort the values in the vector
    sort(scores.begin(), scores.end());

    // Compute the mode.
    int current_count = 1;
    int mode;
    int mode_count = 0;
    vector<int>::iterator current = scores.begin();
    ++current;
    vector<int>::iterator previous = scores.begin();

    // Loop invariants:
    // (a) current_count == number of occurrences of *previous in the
    //     interval of the vector up until previous
    // (b) mode is the most frequently occurring score that is also less
    //     than *previous in the interval up to previous
    // (c) mode_count is the number of times mode occurs
    for (; current != scores.end(); ++current, ++previous) {
        if (*current == *previous) {
            current_count++;
        } else if (current_count >= mode_count) {
            // Change in the score
            mode = *previous;
            mode_count = current_count;
            current_count = 1;
        } else {
            current_count = 1;
        }
    }

    // Handle the possibility of the last score being the most frequent.
    // The statement of the loop invariant makes it clear that the loop
    // ends without checking if the last value is the mode.
    if (current_count >= mode_count) {
        mode = *previous;
        mode_count = current_count;
    }

    cout << "The most frequent grade, occurring " << mode_count << " times, is " << mode << endl;
    return 0; // Everything ok
}