Computer Science II — CSci 1200
Lecture 8 — Iterators; Programming Examples

Test 1

- 102 total points on the test. Therefore, your score is out of 102.
- Class average: 89.6 / 102
- Distribution:

Range	**Percent**	**Count**
92 - 102	90-100	62
81.5 - 91.5	80-89	40
71.5 - 81	70-79	18
61 - 71	60-69	10
50.5 - 61.5	50-59	10
≤ 50	≤ 50	9

- Solutions are posted on line

Reminder

If you did not receive credit for Checkpoint 3 in last week's lab (Lab 4), you may show it to a TA at the start of lab this week in order to earn credit. This applies to last week's lab only.

Review from Lecture 7

Koenig & Moo: Chapter 5.1-5.5

- 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. We will review this solution today.

  - Of particular note, 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 mush more efficient.

- As we will see, for the enrollment problem the list is a better sequential container class than the vector.

Today's Class — Review; Iterators; Lists; Programming Examples

- 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

References and Return Values

There is one part of the `Student` class that we did not discuss thoroughly in Lecture 6 and I'd like to discuss it now. The topic is returning references to objects. Here is an excerpt from the code from Lecture 6.

```
class Student {
public:
  bool read( istream& in_str,
             unsigned int num_homeworks,
             unsigned int num_tests );

  void compute_averages( double hw_weight );

  const string& first_name() const { return first_name_; }
  const string& last_name() const { return last_name_; }
  //  etc....

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

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

  ```
  string one = "Tommy";
  string& two = one;    //  two is a reference to one
  two[1] = 'i';
  cout << one << endl;   //  This outputs the string Timmy
  ```

  The reference variable `two` refers to the same string as variable `one`. Therefore, when we change `two`, we are changing `one`.

  - Exactly the same thing occurs with reference parameters to functions.

- Returning to the student grades program, in the main function we had a vector of students:

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

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

```
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 reason is that the Student class 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 you instead wrote:

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

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

- Hence in both cases you'd be safe.

- HOWEVER, you'd get yourself in trouble if the member function return type was only a reference, and not a const reference. Then you could access and change the internal contents of an object! This is a bad idea in most cases.

Lists

- Our second standard-library container class
- Lists are formed as a sequentially linked structure instead of the array-like, random-access / indexing structure of vectors.
  - Pictures showing the differences will be drawn in class.
- Lists have `push_front` and `pop_front` functions in addition to the `push_back` and `pop_back` functions of vector
- Erasing is very efficient for a list — this is independent of the size of the list.
- 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.

Iterators and Iterator Operations — General

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

```
vector<double>::iterator p;
list<string>::iterator q;
```

  There are also string iterators, which work just like vector iterators.

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

```
p = v.begin() + i;   //  i-th location in the vector
q = r.begin();       //  first entry in the list
```

  when `v` is a vector of doubles and `r` is a list of strings.

- The contents of the specific entry referred to by an iterator are accessed using the `*` operator. For example,

```
*p = 3.14;
cout << *q << endl;
*q = "Hello";
```

  In the first and third lines, `*p` and $*q$ are l-values. In the second, `*q` is an r-value.

- Stepping through a container, either forward and backward, is done using increment (`++`) and decrement (`--`). Hence,

```
++ p ;
q -- ;
```

  moves `p` to the next location in the vector and moves q to the previous location in the list. Neither operation changes any of the contents of vector `v` or list `r`!

- Finally, we can change the container that an iterator `p` attached to **as long as the types are correct**. Thus, if `v` and `w` are both `vector<double>`, then

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

are all valid because `p` is a `vector<double>::iterator`, but if `a` is a `vector<string>` then

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

is a syntax error because of a type clash!

Iterators and Iterator Operations — Vector Iterators

Vector (and string) iterators have special capabilities that other container iterators (list for now, but others later) 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.

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), b(20, 1.1);
  vector<string> c(3, string("hi"));
  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;
  ```

  This has **changed the contents** of vector `a`.

- We can also access the contents of vector `a` using subscripting.

- We will treat `i` as the subscript. It is not attached to vector `a` in any way. We write,

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

  to access and output the value stored at location 5 of `a`.

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.

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 function `erase`,

  - 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.

- You will often see code like the above written

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

  This makes iterator `p` refer to a valid entry.

- Our `erase_from_vector` function from the Lecture 7 enrollment example becomes simply

```
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 $O(1)$.

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.
- We will implement this **during lecture** using the skeleton program provided on the next page of this handout.
- Central to the code will be the use of iterators.