Computer Science II — CSci 1200
Lecture 9 — Lists, Iterators and Examples

Announcements — Returning Test 1; Lab Next Week

- Because of the snowstorm, grading of Test 1 is not yet complete.
- Graded tests will be available during lab period next week and during class next Friday.
- There is no scheduled lab next week, but the TAs will be available in the lab during the lab period to
  - return Test 1
  - help you with lists, iterators, and HW 4

Review from Lecture 8

Ford&Topp, Ch. 6; Koenig&Moo Ch. 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 allowed us to replace the vector with a list in the third version.

Today's Class — Iterators; Lists; Programming Examples

- Lists
- Review of iterators and iterator operations
- Differences between indices and iterators
- Differences between lists and vectors
- Insert and erase
- Sieve of Eratosthenes, revisited.
- Returning references to member variables from member functions

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.
- Lists have `push_front` and `pop_front` functions in addition to the `push_back` and `pop_back` functions of vector
- Both erase and insert at a particular location in a list are very efficient, independent of the size of the list.
- We can not 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`!

- When we have an iterator that references a list (or vector) of objects, the member functions of the object are accessed through the -> operator. For example, if

```
list<string> words;
```

is a list of words, then

```
for ( list<string>::iterator p = words.begin(); p != words.end(); ++p )
  cout << *p << " " << p->size() << '\n';
```

prints out each of the words and its size. See Lecture 8 for more detail.

- Finally, we can change the container that an iterator p is 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 entry 0 in v
p = w.begin() + 2;
*p = 2.78;   // changes entry 2  in w
```

are all valid because p is a vector<double>::iterator. On the other hand, 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.

- These operators are allowed because vectors are really arrays (underneath the hood) and vector iterators are really just pointers.
- Neither of these is allowed for list iterators (and most other iterators, for that matter) because of the underlying linked structure of lists.
- Vector (and string) iterators may also be compared using <, <=, >, >=. List (and other) iterators may only be compared using == and !=.

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 — the entry **after** the entry that was erased.

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

Exercise

The following problem is very important for HW 4.

> Write code to erase all strings that are longer than `max_size` from a list of strings called `words`.

Insert

- Given a list and an iterator, `p`, referring to an item in the list, the `insert` member function of the list class may be used to insert a new item in the list **before** the item referred to by `p`.

- The insert function returns an iterator referring to the new item that was inserted.

- As an example,

  ```
  list<string>::iterator p = words.begin();
  p = words.insert( p, string("Hello") );
  ```

  inserts the string `"Hello"` at the start of the list. This code is effectively equivalent to

  ```
  words.push_front( string("Hello") );
  list<string>::iterator p = words.begin();
  ```

- Continuing with this example,

  ```
  list<string>::iterator p = words.end();
  p = words.insert( p, string("Bye") );
  p = words.insert( p, string("Good") );
  ```

Now, the string `"Bye"` is the last entry in the list, the string `"Good"` is the second to last entry, and `p` refers to the entry containing `"Good"`.

- To conclude this example, if `words` was initially empty at the start of the foregoing code, then

  ```
  for ( p=words.begin(); p!=words.end(); ++p )
    cout << *p << ' ';
  cout << endl;
  ```

  would output

  ```
  Hello Good Bye
  ```

- Finally, there are other forms of both `list<T>::insert` and `list<T>::erase`, which you can learn about from the textbooks or on-line resources.

Exercise

This exercise is also very useful practice for HW 4.

> Suppose `words` is a list of strings and `words` is in lexicographic order. Given a new string `s`, write code that will insert `s` into `words` in such a way that the resulting list is still in lexicographic order.

Prime Numbers: Sieve of Eratosthenes

If there is time, we will revisit the Sieve of Eratosthenes problem using lists instead of arrays or vectors. The solution will be posted on-line.

```
#include <cstdlib>     //  for use of atoi function
#include <iostream>
#include <list>
using namespace std;

int
main( int argc, char* argv[] )
{
  //  Check a command-line argument is provided.
  if ( argc !=2 )
    {
      cerr << "Usage:  " << argv[0] << " n" << "\n"
   << " where n is a positive integer.\n";
      return 1;
    }

  //  Convert the C-style string to an integer using the function atoi found in cstdlib.
  int n = atoi( argv[1] );
  if ( n <= 0 )
    {
      cerr << "The argument must be a positive integer.\n";
      return 1;
    }

  //  Make a list with n integers
  list<int> primes;
  for ( int i=2; i<=n; ++i ) primes.push_back(i);

  // Fill in here....




  cout << '\n';
  return 0;
}
```

References and Return Values

Here is additional details on the idea of returning a reference to an object. We have already done this several times, once in the Student class from Lecture 4 and once in our Vec<T> class. Here again is the Student class.

```
  // Constructors
  Student();
  Student( std::string const& first, std::string const& last,
           std::string const& id );

  //  Functions for manipulating the information stored in the Student object
  void reset();     // empty and reset all local variables
  void add_name_and_id( std::string const& first, std::string const& last,
                        std::string const& id );
  void add_hw( int hw_score );
  void add_test( int test_score );

  //  Main computation function.
  void compute_averages( double hw_weight );

  //  Access the results
  const std::string& first_name() const { return first_name_; }
  const std::string& last_name() const { return last_name_; }
  const std::string& id_number() const { return id_number_; }

  double hw_avg() const { return hw_avg_; }
  double test_avg() const { return test_avg_; }

  //  Output the final information.
  std::ostream& output_name( std::ostream& out_str ) const;
  std::ostream& output_averages( std::ostream& out_str ) const;

private:
  std::string first_name_;
  std::string last_name_;
  std::string id_number_;
  std::vector<int> hw_scores_;
  double hw_avg_;
  std::vector<int> test_scores_;
  double test_avg_;
  double final_avg_;
};
```

- 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.