Computer Science II — CSci 1200
Lecture 8 — Iterators and Lists

Test 1 Reminder

- Tuesday February 13, 2:00 - 3:30
- West Hall Auditorium
- You may bring a 8.5x11, double-sided crib sheet
- Bring photo id
- Practice problems and test emphasis are covered in review material available at the course web site.

Review from Lecture 7

- Designing our own container classes
- Vectors and dynamic member variables
- Templated classes
- Copy constructors, assignment operators and destructors
- Implementation

We will review the solution to the Lecture 7 exercises and to Lab 4 at the start of class today: `push_back`, `erase`, `resize`, `operator==`.

Today's Class

Ford & Topp Chapter 6; Koenig & Moo Sections 5.1-5.5

- Program to maintain class list and waiting list
- Erasing items from vectors is inefficient
- Iterators and iterator operations
- Lists as a different sequential container class.

Extra examples of using iterators and lists will be posted on the course web page.

Preliminary Discussion: `pop_back` and `erase`

- We have seen how `push_back` adds a value to the end of a vector, increasing the size of the vector by 1.
- There is a corresponding function called `pop_back`, which removes the last item in a vector, reducing the size by 1.
- There are also vector functions called `front` and `back` which denote (and thereby provide access to) the first and last item in the vector, allowing them to be changed
- Here is an example:

  ```
  vector<int> a(5, 1); // a has 5 values, all 1
  a.pop_back();        // a now has 4 values
  a.front() = 3;       // equivalent to the statement, a[0] = 3;
  a.back() = -2;       // equivalent to the statement, a[a.size()-1] = -2;
  ```

- We have also seen, through our lab exercise on the `Vec<T>` templated class, the existence of the function `erase`. In particular,

  ```
  a.erase( a.begin()+i );
  ```

  erases the `i`th entry in vector `a`.

Example: Course Enrollment and Waiting List

Program in `classlist_vec.cpp` to build the class list and the waiting list for a single course.

- The program reads simple requests about additions and deletions from a course from a file.
- Vectors store the enrolled students and the waiting students.
- The main work is done in the two functions `enroll_student` and `remove_student`.
- The invariant on the loop in the main function determines how these functions must behave.

Exercises

1. Give an order notation ("O") estimate of the computational cost of the following vector operations:
   (a) `pop_back`
   (b) `push_back`
   (c) `erase`
2. What does the cost of `erase` say about the cost of the operations in our course enrollment program?

What To Do About the Expense of Erasing From a Vector?

- When items are continually being inserted and removed, vectors are not a good choice for the container.
- Instead we need a different sequential container, called a *list*.
  - This has a "linked" structure that makes the cost of erasing independent of the size.
- We will move toward a list-based implementation of the program in two steps:
  - Rewriting our `classlist_vec.cpp` code in terms of *iterator* operations.
  - Replacing vectors with lists

Iterators

- Here is the definition from Koenig & Moo. An iterator:
  - identifies a container and a specific element stored in the container,
  - lets us examine and change (except for const iterators) the value stored at that element of the container,
  - provides operations for moving (the iterators) between elements in the container,
  - restricts the available operations in ways that correspond to what the container can handle efficiently.
- Iterators for different container classes have many operations in common. As we will see, this often makes the switch between containers fairly straightforward from the programer's viewpoint.
- Iterators in many ways are generalizations of pointers: many operators / operations defined for pointers are defined for iterators. You should use this to guide your beginning understanding and use of iterators.

Iterator Declarations and Operations

- As we have already seen with the `std::vector<T>` class and our own `Vec<t>` implementation, iterator types are declared by the container class. For example,

  ```
  vector<string>::iterator p;
  vector<string>::const_iterator q;
  ```

  defines two (uninitialized) iterator variables. For vectors, these are pointers.
- The *dereference operator* is used to access the value stored at an element of the container. In particular, the code

  ```
  p = enrolled.begin();
  *p = "012312";
  ```

  changes the first entry in the `enrolled` vector.

- The dereference operator is combined with dot operator for accessing member functions of class objects stored in containers. Here is an example using the `Student` class and `students` vector defined in the code associated with Lecture 2:

  ```
  vector<Student>::iterator i = students.begin();
  (*i).compute_averages( 0.45 );
  ```

  Notes:

  - This operation would be illegal if `i` had been defined as a `const_iterator` because `compute_averages` is a non-const member function.
  - The parentheses on the `*i` are **required!**.
- There is a "syntactic sugar" for the combination of the dereference operator and the dot operator, which is exactly equivalent but simpler:

  ```
  vector<StudentRec>::iterator i = students.begin();
  i->compute_averages( 0.45 );
  ```

- Just like pointers, iterators can be incremented and decremented using the `++` and `--` operators to move to the next or previous element of any container.
- Iterators can be compared using the `==` and `!=` operators.
- Iterators can be assigned, just like any other variable.
- Vector iterators have several additional operations:
  - Integer values may be added to them or subtracted from them. This leads to statements like

    ```
    enrolled.erase( enrolled.begin() + loc );
    ```

    which we have already used.
  - Vector iterators may be compared using operators like `<`, `<=`, etc.
  - For most containers (other than vectors), these "random access" iterator operations are not legal and therefore prevented by the compiler. Reasons will gradually become clear.

Revising the Class List Program to Use Iterators

I will demonstrate revising the main function of the class list program to use iterators. As an exercise, you will revise the two functions `enroll_student` and `remove_student` to do so as well. .

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 vectors
- Erase is very efficient for a list, 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.

Revising the Class List Program to Use Lists and Iterators

In class we will rewrite this program again to use lists and list iterators. The resulting program will be posted on the web.

Looking Ahead

- Although the interface (public member functions) 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:
  - No indexing (subscripting) in lists.
  - No `push_front` or `pop_front` operations for vectors.
  - 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` invalidates ALL iterators in a vector

    The value of any associated vector iterator must be re-assigned / re-initialized after either operation.
  - Lists have their own, specialized `sort` functions.
- We have already seen that vectors are implemented in terms of arrays, which explains why `erase` is so expensive and is why `push_front` and `pop_front` are not allowed (even though they could be implemented).
- Soon we will start to see how `std::list<T>` is implemented in terms of a linked-list.