CSCI-1200 Data Structures — Fall 2016
Lecture 9 — Iterators & STL Lists

Review from Lecture 8

- Designing our own container classes
- Dynamically allocated memory in classes
- Copy constructors, assignment operators, and destructors
- Templated classes, Implementation of the DS `Vec` class, mimicking the STL `vector` class

Today

- Another `vector` operation: `pop_back`
- *Erasing items* from vectors is inefficient!
- Iterators and iterator operations
- STL `lists` are a different sequential container class.
- Returning references to member variables from member functions
- `Vec` iterator implementation

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

9.1 Review: Constructors, Assignment Operator, and Destructor

From an old test: Match up the line of code with the function that is called. Each letter is used exactly once.

☐	`Foo f1;`	a)	assignment operator
☐	`Foo* f2;`	b)	destructor
☐	`f2 = new Foo(f1);`	c)	copy constructor
☐	`f1 = *f2;`	d)	default constructor
☐	`delete f2;`	e)	none of the above

9.2 Another STL `vector` operation: `pop_back`

- 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. For 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;
```

```cpp
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
#include <assert.h>
using namespace std;

void erase_from_vector(unsigned int i, vector<string>& v) {
  /* EXERCISE: IMPLEMENT THIS FUNCTION */
}

// Enroll a student if there is room and the student is not already in course or on waiting list.
void enroll_student(const string& id, unsigned int max_students,
                    vector<string>& enrolled, vector<string>& waiting) {
  // Check to see if the student is already enrolled.
  unsigned int i;
  for (i=0; i < enrolled.size(); ++i) {
    if (enrolled[i] == id) {
      cout << "Student " << id << " is already enrolled." << endl;
      return;
    }
  }
  // If the course isn't full, add the student.
  if (enrolled.size() < max_students) {
    enrolled.push_back(id);
    cout << "Student " << id << " added.\n"
         << enrolled.size() << " students are now in the course." << endl;
    return;
  }
  // Check to see if the student is already on the waiting list.
  for (i=0; i < waiting.size(); ++i) {
    if (waiting[i] == id) {
      cout << "Student " << id << " is already on the waiting list." << endl;
      return;
    }
  }
  // If not, add the student to the waiting list.
  waiting.push_back(id);
  cout << "The course is full.  Student " << id << " has been added to the waiting list.\n"
       << waiting.size() << " students are on the waiting list." << endl;
}

// Remove a student from the course or from the waiting list.  If removing the student from the
// course opens up a slot, then the first person on the waiting list is placed in the course.
void remove_student(const string& id, unsigned int max_students,
                    vector<string>& enrolled, vector<string>& waiting) {
  // Check to see if the student is on the course list.
  bool found = false;
  unsigned int loc=0;
  while (!found && loc < enrolled.size()) {
    found = enrolled[loc] == id;
    if (!found) ++loc;
  }
  if (found) {
    // Remove the student and see if a student can be taken from the waiting list.
    erase_from_vector(loc, enrolled);
    cout << "Student " << id << " removed from the course." << endl;
    if (waiting.size() > 0) {
      enrolled.push_back(waiting[0]);
      cout << "Student " << waiting[0] << " added to the course from the waiting list." << endl;
      erase_from_vector(0, waiting);
      cout << waiting.size() << " students remain on the waiting list." << endl;
    } else {
      cout << enrolled.size() << " students are now in the course." << endl;
    }
  } else {
    // Check to see if the student is on the waiting list
    found = false;
    loc = 0;
    while (!found && loc < waiting.size()) {
      found = waiting[loc] == id;
      if (!found) ++loc;
    }
    if (found) {
      erase_from_vector(loc, waiting);
      cout << "Student " << id << " removed from the waiting list.\n"
           << waiting.size() << " students remain on the waiting list." << endl;
    } else {
      cout << "Student " << id << " is in neither the course nor the waiting list" << endl;
    }
  }
}

int main() {
  // Read in the maximum number of students in the course
  unsigned int max_students;
  cout << "\nEnrollment program for CSCI 1200\nEnter the maximum number of students allowed\n";
  cin >> max_students;

  // Initialize the vectors
  vector<string> enrolled;
  vector<string> waiting;

  // Invariant:
  // (1) enrolled contains the students already in the course,
  // (2) waiting contains students who will be admitted (in the order of request) if a spot opens up
  // (3) enrolled.size() <= max_students,
  // (4) if the course is not filled (enrolled.size() != max_students) then waiting is empty
  do {
    // check (part of) the invariant
    assert (enrolled.size() <= max_students);
    assert (enrolled.size() == max_students || waiting.size() == 0);
    cout << "\nOptions:\n"
         << "  To enroll a student type 0\n"
         << "  To remove a student type 1\n"
         << "  To end type 2\n"
         << "Type option ==> ";
    int option;
    if (!(cin >> option)) { // if we can't read the input integer, then just fail.
      cout << "Illegal input.  Good-bye.\n";
      return 1;
    } else if (option == 2) {
      break; // quit by breaking out of the loop.
    } else if (option != 0 && option != 1) {
      cout << "Invalid option.  Try again.\n";
    } else { // option is 0 or 1
      string id;
      cout << "Enter student id: ";
      if (!(cin >> id)) {
        cout << "Illegal input.  Good-bye.\n";
        return 1;
      } else if (option == 0) {
        enroll_student(id, max_students, enrolled, waiting);
      } else {
        remove_student(id, max_students, enrolled, waiting);
      }
    }
  }
  while (true);

  // some nice output
  sort(enrolled.begin(), enrolled.end());
  cout << "\nAt the end of the enrollment period, the following students are in the class:\n\n";
  for (unsigned int i=0; i<enrolled.size(); ++i) { cout << enrolled[i] << endl;  }
  if (!waiting.empty()) {
    cout << "\nStudents are on the waiting list in the following order:\n";
    for (unsigned int j=0; j<waiting.size(); ++j) { cout << waiting[j] << endl; }
  }
  return 0;
}
```

9.3 Motivating Example: Course Enrollment and Waiting List

- This program maintains the class list and the waiting list for a single course. The program is structured to handle interactive input. Error checking ensures that the input is valid.
- 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.

9.4 Exercises

1. Write `erase_from_vector`. This function removes the value at index location i from a vector of strings. The size of the vector should be reduced by one when the function is finished.

```
// Remove the value at index location i from a vector of strings.  The
// size of the vector should be reduced by one when the function is finished.
void erase_from_vector(unsigned int i,  vector<string>& v) {




}
```

2. Give an order notation estimate of the cost of `erase_from_vector`, `pop_back`, and `push_back`.

9.5 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

9.6 Iterators

- Here's 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.
- As we will see, iterators for different container classes have many operations in common. 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.

9.7 Iterator Declarations and Operations

- Iterator types are declared by the container class. For example,

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

  defines two (uninitialized) iterator variables.

- The *dereference operator* is used to access the value stored at an element of the container. 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 the member variables and member functions of elements stored in containers. Here's an example using the `Student` class and `students` vector from Lecture 4:

```
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** (because of operator precedence).

- There is a "syntactic sugar" for the combination of the dereference operator and the dot operator, which is exactly equivalent:

```
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() + 5);
```

  - 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. The reasons will become clear as we look at their implementations.

9.8 Exercise: Revising the Class List Program to Use Iterators

- Now let's modify the class list program to use iterators. First rewrite the `erase_from_vector` to use iterators.

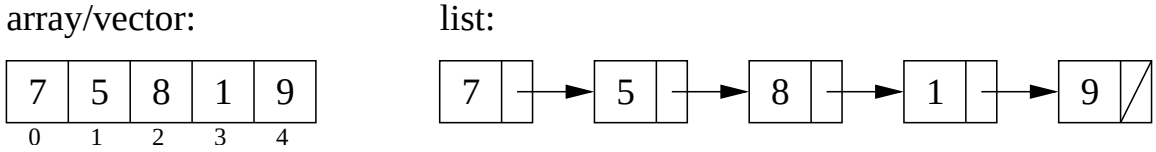
```
void erase_from_vector(vector<string>::iterator itr,  vector<string>& v) {




}
```

  Note: the STL `vector` *class has a function that does just this... called* `erase`*!*

- Now, edit the rest of the file to remove all use of the vector subscripting operator.

9.9 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.



- 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 `vector`s / arrays allow "random-access" / indexing / `[]` subscripting. We can immediately jump to an arbitrary location within the vector / array.
  - STL `list`s 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 memeory. (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.

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

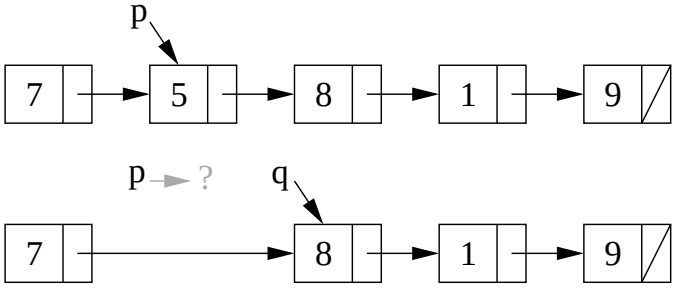
Now let's further modify the program to use lists instead of vectors. Because we've already switched to iterators, this change will be relatively easy. And now the program will be more efficient!

9.11 Erase & Iterators

- STL `list`s and `vector`s 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 $O(1)$.

9.12 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.

9.13 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.

9.14 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!