CSCI-1200 Computer Science II — Fall 2007
Lecture 8 — Iterators

Review from Lecture 7

- Designing our own container classes
- Dynamically allocated memory in classes
- Copy constructors, assignment operators and destructors
- Templated classes
- Implementation of the CS2 `Vec` class, mimicking the STL `vector` class

Today

- *Erasing items* from vectors is inefficient!
- Iterators and iterator operations
- Lists as a different sequential container class.

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

8.1 Another `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;
```

8.2 Example: Course Enrollment and Waiting List

Today we'll look at a program to build and maintain 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.

```
// Build and maintain a list of students enrolled in a class and the waiting list.
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
#include <assert.h>
using namespace std;

void enroll_student(const string& id_number, unsigned int max_students,
    vector<string>& enrolled, vector<string>& waiting);
void remove_student(const string& id_number, unsigned int max_students,
    vector<string>& enrolled, vector<string>& waiting);
void erase_from_vector(unsigned int i, vector<string>& v);
```

```
int main() {
  // Read in the maximum number of students in the course
  unsigned int max_students;
  cout << "\nWelcome to the enrollment program for CSCI 1200\n"
       << "Please enter the maximum number of students allowed\n";
  cin >> max_students;

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

  // Invariant:
  // (1) enrolled contains the students already in the course,
  // (2) waiting contains students who would will to be admitted (in
  //     the order of request) if a spot opens up, in the
  // (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_number;
      cout << "Enter student id: ";
      if (!(cin >> id_number)) {
        cout << "Illegal input.  Good-bye.\n";
        return 1;
      } else if (option == 0) {
        enroll_student(id_number, max_students, enrolled, waiting);
      } else {
        remove_student(id_number, 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\n"
       << "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;
}
```

```
// Enroll a student in the course if there is room and if the student
// is not already in the course or on the waiting list.
void enroll_student(const string& id_number, 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_number) {
      cout << "Student " << id_number << " is already enrolled." << endl;
      return;
    }
  }

  // If the course isn't full, add the student.
  if (enrolled.size() < max_students) {
    enrolled.push_back(id_number);
    cout << "Student " << id_number << " 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_number) {
      cout << "Student " << id_number << " is already on the waiting list." << endl;
      return;
    }
  }

  // If not, add the student to the waiting list.
  waiting.push_back(id_number);
  cout << "The course is full.  Student " << id_number
       << " 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_number, 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_number;
    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_number << " 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_number;
      if (!found) ++loc;
    }
    if (found) {
      erase_from_vector(loc, waiting);
      cout << "Student " << id_number << " removed from the waiting list.\n"
           << waiting.size() << " students remain on the waiting list." << endl;
    } else {
      cout << "Student " << id_number << " is in neither the course nor the waiting list" << endl;
    }
  }
}
```

8.3 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`.

8.4 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

8.5 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.

8.6 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 an earlier lecture:

```
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);
```

    which we will soon use.
  - 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.

8.7 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. Then edit the rest of the file to remove all use of the vector subscripting operator.

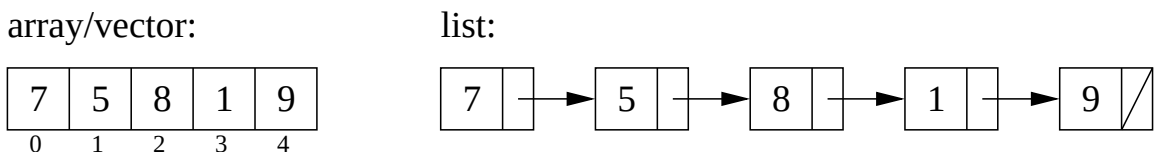
```
void erase_from_vector(vector<string>::iterator itr,  vector<string>& v) {




}
```

- The STL `vector` class has a function that does just this! The return value is an iterator pointing to the new location of the element just after the one that was deleted:

```
iterator erase ( iterator position );
```

8.8 A new datatype: `list`

- Lists are our second standard-library container class. (Vectors were the first.) 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 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").

8.9 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!

8.10 Looking Ahead

- 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:
  - 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).
- Next week we'll look at how to implement lists and the cost of the corresponding operations.