Review from Lecture 7

- Algorithm Analysis, Formal Definition of Big O Notation
- Examples!

Today

- Constructors, Assignment Operator, and Destructor
- Flowchart for determining use of const & reference
- Big ‘O’ Notation for push_back
- Another vector operation: pop_back
- Erasing items from vectors is inefficient!
- Iterators and iterator operations
- Preview: STL lists are a different sequential container class.

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

8.1 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.

\[
\begin{array}{ccc}
\text{Foo } f1; & a) \text{ assignment operator} \\
\text{Foo* } f2; & b) \text{ destructor} \\
\text{f2 = new Foo(f1);} & c) \text{ copy constructor} \\
\text{f1 = *f2;} & d) \text{ default constructor} \\
\text{delete f2;} & e) \text{ none of the above}
\end{array}
\]

8.2 Use of const and reference with functions / parameters / return types

\[
\begin{array}{c}
\text{std::string } \_\_ \text{ my_function} \_\_ (\_\_ \text{ int } \_\_ a, \\
\text{std::vector<int> } \_\_ b, \\
\text{Bar } \_\_ c, \\
\text{Foo* } \_\_ d) \{ \\
\}
\end{array}
\]
8.3 Exercise: Big 'O' Notation Analysis of STL vector / Vec push_back

- // Add an element to the end, resize if necessary.
  template <class T> void Vec<T>::push_back(const T& val) {
    if (m_size == m_alloc) {
      // Allocate a larger array, and copy the old values
      // Calculate the new allocation. Make sure it is at least one.
      m_alloc *= 2;
      if (m_alloc < 1) m_alloc = 1;
      // Allocate and copy the old array
      T* new_data = new T[ m_alloc ];
      for (size_type i=0; i<m_size; ++i)
        new_data[i] = m_data[i];
      // Delete the old array and reset the pointers
      delete [] m_data;
      m_data = new_data;
    }

    // Add the value at the last location and increment the bound
    m_data[m_size] = val;
    ++ m_size;
  }

8.4 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;

8.5 Exercise: Write a non-member helper function erase_from_vector

1. 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.

```cpp
void erase_from_vector(unsigned int i, vector<string>& v) {
}
```

2. What is the big O notation estimate of the average cost of erase_from_vector, pop_back, and push_back.

8.6 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.

```text
array/vector: 7 5 8 1 9 7 9 5 8 1
0 1 2 3 4
list: 7 5 8 1 9
```

- Switching from a vector to a list may improve the performance of programs that make significant use of insert and erase in the middle of sequences.
- ... but before we do that, we need to learn about iterators. Iterators are standardized way to interact with the data elements stored in all of the different STL containers.

8.7 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 programmer’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.

8.8 Iterator Declarations and Operations

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

```cpp
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 3:

```
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.

### 8.9 Example – common confusion/mistake with STL iterators

• NOTE: The example syntax below is the same for STL vector and STL lists.

```
std::vector<int> data;
std::vector<int>::iterator itr,itr2,itr3;
//std::list<int> data;
//std::list<int>::iterator itr,itr2,itr3;

data.push_back(100); data.push_back(200);
data.push_back(300); data.push_back(400); data.push_back(500);

itr = data.begin(); // itr is pointing at the 100
++itr; // itr is now pointing at 200
*itr += 1; // 200 becomes 201

// itr += 1; // NOTE: this syntax only works for vector/vector iterator
// but it does not compile for list/list iterator
// list iterators cannot be advanced like this

itr = data.end(); // itr is pointing "one past the last legal value" of data
itr--; //itr is now pointing at 500;
itr2 = itr--; //itr is now pointing at 400, itr2 is still pointing at 500
itr3 = --itr; //itr is now pointing at 300, itr3 is also pointing at 300

// dangerous: decrementing the begin iterator is "undefined behavior"
```
// (similarly, incrementing the end iterator is also undefined)
// it may seem to work, but break later on this machine or on another machine!
itr = data.begin();
itr--; // dangerous!
itr++;
assert (*itr == 100); // might seem ok... but rewrite the code to avoid this!

8.10 Motivating Example Program: Course Enrollment and Waiting List

- This program (last page of the handout) 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.

8.11 Exercise: Revising the Class List Program to Use Iterators

- Now let’s modify the class list program to use iterators. First rewrite our homemade erase_from_vector to use iterators.

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

- Note: The STL vector class has a member function that does just this... called erase!
  - Give a sample call to our homemade erase_from_vector function:
  ```c
  erase_from_vector(v.begin(), v);
  ```
  - Give the equivalent statement that instead uses the erase member function:

```c
v.erase(v.begin());
```

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

8.12 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.

```
array/vector:    list:
7 5 8 1 9 7 9 5 8 1
0 1 2 3 4
```

- 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 vectors / arrays allow “random-access” / indexing / [] subscripting. We can immediately jump to an arbitrary location within the vector / array.
  - STL lists 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 memory. (We’ll see this when we discuss the implementation details in future lectures).
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.

```cpp
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.

### 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!

#### STL list (and STL vector) has an erase member function

- STL lists and vectors each have a special member function called erase. In particular, given list of ints s, consider the example:

```cpp
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.

![Diagram of list after erase](image)

- To reuse the iterator p and make it a valid entry, you will often see the code written:

```cpp
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 $O(1)$.

### 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.

### 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.
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>

using namespace std;

void erase_from_vector(unsigned int loc, vector<string>& v) {
    assert(loc < v.size());
    v.erase(v.begin() + loc);
}

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 loc = 0;
    while (!found && loc < enrolled.size()) {
        loc = loc + 1;
        if (enrolled[loc] == 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.
" << endl;
    } else {
        cout << "Illegal input. Good-bye.\n"; return;
    }
}

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 waiting list.
    unsigned int loc = 0;
    while (!found && loc < waiting.size()) {
        loc = loc + 1;
        if (waiting[loc] == 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 << "Student " << id << " has been added to the waiting list.\n" << 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()) {
            loc = loc + 1;
            if (found == false) {
                // 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.\n" << endl;
            }
            if (waiting.size() > 0) {
                enrolled.push_back(waiting[0]);
                cout << "Student " << waiting[0] << " added to the course from the waiting list.\n" << endl;
                erase_from_vector(0, waiting);
                cout << "Student " << id << " removed from the waiting list.\n" << endl;
                return;
            }
            else {
                cout << "Students are now in the course.\n" << endl;
                return;
            }
        }
    }
}

int main() {
    // Read in the maximum number of students in the course
    unsigned int max_students;
    cout << \nEnrollment program for CSCI 1200
Enter the maximum number of students allowed\n" << endl;
    cin >> max_students;
    // Initialize the vectors
    vector<string> enrolled;
    vector<string> waiting;
    // (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
    
    // Some nice output
    sort(enrolled.begin(), enrolled.end());
    cout << \nAt the end of the enrollment period, the following students are in the class:\n";
    for (unsigned int i=0; i<enrolled.size(); ++i) { cout << enrolled[i] << endl;  }
    cout << "Students 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;
}