CSCI-1200 Data Structures — Spring 2020
Lecture 8 — Order Notation & 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 DS `Vec` class, mimicking the STL `vector` class

Today's Lecture

- Algorithm Analysis, Formal Definition of Order Notation

- Another `vector` operation: `pop_back`

- *Erasing items* from vectors is inefficient!

- Iterators and iterator operations

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

8.1   Algorithm Analysis

Why should we bother?

- We want to do better than just implementing and testing every idea we have.

- We want to know why one algorithm is better than another.

- We want to know the best we can do. (This is often quite hard.)

How do we do it?   There are several options, including:

- Don't do any analysis; just use the first algorithm you can think of that works.

- Implement and time algorithms to choose the best.

- Analyze algorithms by counting operations while assigning different weights to different types of operations based on how long each takes.

- Analyze algorithms by assuming each operation requires the same amount of time. Count the total number of operations, and then multiply this count by the average cost of an operation.

8.2   Exercise: Counting Example

- Suppose `arr` is an array of `n` doubles. Here is a simple fragment of code to sum of the values in the array:

  ```
  double sum = 0;
  for (int i=0; i<n; ++i)
    sum += arr[i];
  ```

- What is the total number of operations performed in executing this fragment? Come up with a function describing the number of operations *in terms of n*.

8.3 Exercise: Which Algorithm is Best?

A venture capitalist is trying to decide which of 3 startup companies to invest in and has asked for your help. Here's the timing data for their prototype software on some different size test cases:

n	foo-a	foo-b	foo-c
10	10 u-sec	5 u-sec	1 u-sec
20	13 u-sec	10 u-sec	8 u-sec
30	15 u-sec	15 u-sec	27 u-sec
100	20 u-sec	50 u-sec	1000 u-sec
1000	?	?	?

Which company has the "best" algorithm?

8.4 Order Notation Definition

In this course we will focus on the intuition of order notation. This topic will be covered again, in more depth, in later computer science courses.

- Definition: Algorithm A is order $f(n)$ — denoted $O(f(n))$ — if constants k and n_0 exist such that A requires no more than $k * f(n)$ time units (operations) to solve a problem of size $n \geq n_0$.

- For example, algorithms requiring $3n + 2$, $5n - 3$, and $14 + 17n$ operations are all $O(n)$.

  This is because we can select values for k and n_0 such that the definition above holds. (What values?)

  Likewise, algorithms requiring $n^2/10 + 15n - 3$ and $10000 + 35n^2$ are all $O(n^2)$.

- Intuitively, we determine the order by finding the *asymptotically dominant term (function of n)* and throwing out the leading constant. This term could involve logarithmic or exponential functions of n. Implications for analysis:
  - We don't need to quibble about small differences in the numbers of operations.
  - We also do not need to worry about the different costs of different types of operations.
  - We don't produce an actual time. We just obtain a rough count of the number of operations. This count is used for comparison purposes.

- In practice, this makes analysis relatively simple, quick and (sometimes unfortunately) rough.

8.5 Common Orders of Magnitude

- $O(1)$, *a.k.a. CONSTANT*: The number of operations is independent of the size of the problem. e.g., compute quadratic root.

- $O(\log n)$, *a.k.a. LOGARITHMIC*. e.g., dictionary lookup, binary search.

- $O(n)$, *a.k.a. LINEAR*. e.g., sum up a list.

- $O(n \log n)$, e.g., sorting.

- $O(n^2)$, $O(n^3)$, $O(n^k)$, *a.k.a. POLYNOMIAL*. e.g., find closest pair of points.

- $O(2^n)$, $O(k^n)$, *a.k.a. EXPONENTIAL*. e.g., Fibonacci, playing chess.

8.6 Exercise: A Slightly Harder Example

- Here's an algorithm to determine if the value stored in variable `x` is also in an array called `foo`. Can you analyze it? What did you do about the `if` statement? What did you assume about where the value stored in `x` occurs in the array (if at all)?

```
int loc=0;
bool found = false;
while (!found && loc < n) {
  if (x == foo[loc]) found = true;
  else loc++;
}
if (found) cout << "It is there!\n";
```

8.7 Best-Case, Average-Case and Worst-Case Analysis

- For a given fixed size array, we might want to know:
  - The fewest number of operations (best case) that might occur.
  - The average number of operations (average case) that will occur.
  - The maximum number of operations (worst case) that can occur.

- The last is the most common. The first is rarely used.

- On the previous algorithm, the best case is $O(1)$, but the average case and worst case are both $O(n)$.

8.8 Approaching An Analysis Problem

- Decide the important variable (or variables) that determine the "size" of the problem. For arrays and other "container classes" this will generally be the number of values stored.

- Decide what to count. The order notation helps us here.
  - If each loop iteration does a fixed (or bounded) amount of work, then we only need to count the number of loop iterations.
  - We might also count specific operations. For example, in the previous exercise, we could count the number of comparisons.

- Do the count and use order notation to describe the result.

8.9 Exercise: Order Notation

For each version below, give an order notation estimate of the number of operations as a function of `n`:

1.
```
int count=0;
for (int i=0; i<n; ++i)
  for (int j=0; j<n; ++j)
    ++count;
```

2.
```
int count=0;
for (int i=0; i<n; ++i)
  ++count;
for (int j=0; j<n; ++j)
  ++count;
```

3.
```
int count=0;
for (int i=0; i<n; ++i)
  for (int j=i; j<n; ++j)
    ++count;
```

8.10 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

8.11 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;
```

8.12 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.

8.13 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 average cost of erase_from_vector, pop_back, and push_back.

8.14 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

8.15 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.

4

```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) {
  unsigned int i;

  // Check to see if the student is already enrolled.
  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;
}
```

8.16   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.

8.17   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.