CSCI-1200 Computer Science II — Spring 2006
Lecture 4 — Vectors & Sorting; Statistical Computations

Review from Lecture 3

- Strings: subscripting and type declarations
- Problem solving: two methods of thinking about output along a diagonal.

4.1 Today's Class

Koenig & Moo: Chapter 3

- `vector` container class,
- `sort` function,
- Applications in computing statistics

4.2 Problem: Grade Statistics

- Read an unknown number of grades.
- Compute:
  - Mean (average)
  - Standard deviation
  - Median (middle value)
- We will write several programs to accomplish these.

4.3 Example: Reading Numbers and Computing the Average

```
//  Program:  average.cpp
//  Author:   Chuck Stewart
//  Purpose:  Compute the average of an input set of grades.  This
//    demonstrates input of a sequence of integers, computing the
//    average, and manipulating the output precision.

#include <fstream>
#include <iomanip>
#include <iostream>

int main( int argc, char* argv[] ) {

  if ( argc != 2 ) {
    std::cerr << "Usage: " << argv[0] << " grades-file\n";
    return 1;
  }

  std::ifstream grades_str( argv[1] );
  if ( !grades_str ) {
    std::cerr << "Can not open the grades file " << argv[1] << "\n";
    return 1;
  }

  // Counting and summation variables.
  int count = 0;
  int sum = 0;
  //  Input variable
  int x;

  // Read in the scores one at a time.  Add each score to the sum and
  // increment the count.
  //
  // The value of the expression grades_str >> x is a reference to the
  // input stream grades_str.  The while condition uses this to test
  // grades_str to see if it is ok.  If it is not, the test is false
  // and the loop ends.  The most common cause of this is finding the
  // end of the input, but it would be false is something other than
  // an integer (or whitespace), such as a letter, is in the input
  // stream.
  while ( grades_str >> x ) {
      ++ count ;
      sum += x;
    }

  //  Output the result.  Set the precision to 3.
  std::cout << "The average of " << count << " grades is "
    << std::setprecision(3) << double(sum) / count
    << std::endl;

  return 0;
}
```

4.4 Standard Deviation

Definition: if $a_0, a_1, a_2, \ldots, a_{n-1}$ is a sequence of n values, and μ is the average of these values, then the standard deviation is

$$\left[\frac{\sum_{i=0}^{n-1}(a_i - \mu)^2}{n-1}\right]^{\frac{1}{2}}$$

Computing this equation requires two passes through the values:

- Once to compute the average
- A second time to compute the standard deviation

Thus, we need a way to store the values. The only tool we have so far is arrays. But arrays are fixed in size and we don't know in advance how many values there will be. This illustrates one reason why we generally will use *standard library vectors* instead of arrays.

4.5 Vectors

- Standard library "container class" to hold sequences.
- A vector acts like a dynamically-sized, one-dimensional array.
- Capabilities:
  - Holds objects of any type
  - Starts empty unless otherwise specified
  - Any number of objects may be added to the end — there is no limit on size.
  - It can be treated like an ordinary array using the subscripting operator.
  - There is NO automatic checking of subscript bounds.
- Here's how we create an empty vector of integers:

  ```
  vector<int> scores;
  ```

- Vectors are an example of a *templated container class*. The angle brackets `< >` are used to specify the type of object (the "template type") that will be stored in the vector.
- `push_back` is a vector function to append a value to the end of the vector, increasing its size by one. This is an $O(1)$ operation (on average).
  - There is NO corresponding `push_front` operation for vectors.
- `size` is a function defined by the vector type (the vector class) that returns the number of items stored in the vector.
- After vectors are initialized and filled in, they may be treated *just like arrays.*
  - In the line

    ```
    sum += scores[i];
    ```

    `scores[i]` is an "r-value", accessing the value stored at location `i` of the vector.
  - We could also write statements like

    ```
    scores[4] = 100;
    ```

    to change a score. Here `scores[4]` is an "l-value", providing the means of storing 100 at location `4` of the vector.
  - It is the job of the programmer to ensure that any subscript value i that is used is legal —- at least 0 and strictly less than `scores.size()`.
- Finish the code in `average_and_deviation.cpp` to compute and output the standard deviation of the grades.

4.6 Example: Using Vectors to Compute Standard Deviation

```
//  Program:  average_and_deviation.cpp
//  Author:   Chuck Stewart
//  Purpose:  Compute the average and standard deviation of an input
//    set of grades.   This introduces the use of a vector to store
//    the grades upon input.

#include <fstream>
#include <iomanip>
#include <iostream>
#include <vector>         // to access the STL vector class
#include <cmath>          // to use standard math library and sqrt

int main( int argc, char* argv[] ) {

  if ( argc != 2 ) {
    std::cerr << "Usage: " << argv[0] << " grades-file\n";
    return 1;
  }
  std::ifstream grades_str( argv[1] );
  if ( !grades_str ) {
    std::cerr << "Can not open the grades file " << argv[1] << "\n";
    return 1;
  }

  std::vector<int> scores;  // Vector to hold the input scores; initially empty.
  int x;                    // Input variable

  //  Read the scores, appending each to the end of the vector
  while ( grades_str >> x ) {
    scores.push_back(x);
  }

  //  Quit with an error message if too few scores.
  if ( scores.size() == 0 ) {
    std::cout << "No scores entered.  Please try again!" << std::endl;
    return 1;
  }

  //  Compute and output the average value.
  int sum=0;           // Accumulation of the values
  for ( unsigned int i = 0; i < scores.size(); ++ i ) {
    sum += scores[i];
  }

  double average = double(sum) / scores.size();
  std::cout << "The average of " << scores.size()
    << " grades is " << std::setprecision(3)
    << average << std::endl;

  //  Exercise:  compute and output the standard deviation.









  return 0;
}
```

4.7 Median

- Intuitively, a median value of a sequence is a value that is less than half of the values in the sequence, and greater than half of the values in the sequence.
- More technically, if $a_0, a_1, a_2, \ldots, a_{n-1}$ is a sequence of n values AND if the sequence is sorted such that $a_0 \le a_1 \le a_2 \le \cdots \le a_{n-1}$ then the median is

$$\begin{cases} a_{(n-1)/2} & \text{if } n \text{ is odd} \\ \frac{a_{n/2-1} + a_{n/2}}{2} & \text{if } n \text{ is even} \end{cases}$$

- Sorting is therefore the key to finding the median.

4.8 Standard Library Sort Function

- The standard library has a series of algorithms built to apply to container classes.
- The prototypes for these algorithms (actually the functions implementing these algorithms) are in header file `algorithm`.
- One of the most important of the algorithms is `sort`.
- It is accessed by providing the beginning and end of the container's interval to sort.
- As an example, the following code reads, sorts and outputs a vector of doubles

```
double x;
std::vector<double> a;
while ( std::cin >> x ) a.push_back(x);
std::sort( a.begin(), a.end() );
for ( unsigned int i=0; i<a.size(); ++i )
  std::cout << a[i] << '\n';
```

- `a.begin()` is an *iterator* referencing the first location in the vector, while `a.end()` is an *iterator* referencing one past the last location in the vector.
  - We will learn much more about iterators in the next few weeks.
  - Every container has iterators: strings have `begin()` and `end()` iterators defined on them.
- The ordering of values by `std::sort` is least to greatest (technically, non-decreasing). We will see ways to change this.

4.9 Example: Computing the Median

Note the use of functions and parameter passing in this example:

```
//  Program:  median_grade.cpp
//  Author:   Chuck Stewart
//  Purpose:  Compute the median value of an input set of grades.
//    This illustrates the use of vectore, of standard library
//    sorting algorithm, and passing vectors as parameters.

#include <algorithm>
#include <cmath>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <vector>

void read_scores( std::vector<int> & scores, std::ifstream & grade_str ) {
  int x;  //  input variable
  while ( grade_str >> x ) {
    scores.push_back(x);
  }
}

void compute_avg_and_std_dev( const std::vector<int>&  s,
      double & avg, double & std_dev ) {
  //  Compute and output the average value.
  int sum=0;           // Accumulation of the values
  for ( unsigned int i = 0; i < s.size(); ++ i ) {
    sum += s[i];
  }
  avg = double(sum) / s.size();

  // Compute the standard deviation
  double sum_sq = 0.0;
  for ( unsigned int i=0; i < s.size(); ++i ) {
    sum_sq += (s[i]-avg) * (s[i]-avg);
  }
  std_dev = sqrt( sum_sq / (s.size()-1) );
}

double compute_median( const std::vector<int> & scores ) {
  // Create a copy of the vector
  std::vector<int> scores_to_sort( scores );

  //  Sort the values in the vector.  By default this is increasing order.
  std::sort( scores_to_sort.begin(), scores_to_sort.end() );

  //  Now, compute and output the median.
  unsigned int n = scores_to_sort.size();

  if ( n%2 == 0 )  // even number of scores
    return double( scores_to_sort[n/2] + scores_to_sort[n/2-1] ) / 2.0;
  else
    return double( scores_to_sort[ n/2 ] );  // same as (n-1)/2 because n is odd
}

int main( int argc, char* argv[] ) {
  if ( argc != 2 ) {
    std::cerr << "Usage: " << argv[0] << " grades-file\n";
    return 1;
  }
  std::ifstream grades_str( argv[1] );
  if ( !grades_str ) {
    std::cerr << "Can not open the grades file " << argv[1] << "\n";
    return 1;
  }

  std::vector<int> scores;  // Vector to hold the input scores; initially empty.

  //  Read the scores, as before
  read_scores( scores, grades_str );

  //  Quit with an error message if too few scores.
  if ( scores.size() == 0 ) {
    std::cout << "No scores entered.  Please try again!" << std::endl;
    return 1;
  }

  //  Compute the average, standard deviation and median
  double average, std_dev;
  compute_avg_and_std_dev( scores, average, std_dev );
  double median = compute_median( scores );

  //  Output
  std::cout << "Among " << scores.size() << " grades: \n"
    << "  average = " << std::setprecision(3) << average << '\n'
    << "  std_dev = " << std_dev << '\n'
    << "   median = " << median << std::endl;

  return 0;
}
```

4.10 Passing Vectors (and Strings) As Parameters

The following outlines rules for passing vectors as parameters. The same rules apply to passing strings.

- If you are passing a vector as a parameter to a function and you want to make a (permanent) change to the vector, then you should pass it **by reference**.
  - This is illustrated by the function `read_scores` in the program `median_grade`.
  - This is very different from the behavior of arrays as parameters.
- What if you don't want to make changes to the vector or don't want these changes to be permanent?
  - The answer we've learned so far is to pass by value.
  - The problem is that the entire vector is copied when this happens!
- The solution is to pass by **constant reference**: pass it by reference, but make it a constant so that it can not be changed.
  - This is illustrated by the functions `compute_avg_and_std_dev` and `compute_median` in the program `median_grade`.
- As a general rule, you should not pass a container object such as a vector or a string, by value because of the cost of copying. There are rare circumstances in which this rule may be violated, but not in CS II.

4.11 Initializing a Vector — The Use of Constructors

Here are several different ways to initialize a vector:

- This "constructs" an empty vector of integers. Values must be placed in the vector using `push_back`.

  ```
  vector<int> a;
  ```

- This constructs a vector of 100 doubles, each entry storing the value 3.14. New entries can be created using `push_back`, but these will create entries 100, 101, 102, etc.

  ```
  int n = 100;
  vector<double> b( 100, 3.14 );
  ```

- This constructs a vector of 10,000 ints, but provides no initial values for these integers. Again, new entries can be created for the vector using `push_back`. These will create entries 10000, 10001, etc.

  ```
  vector<int> c( n*n );
  ```

- This constructs a vector that is an exact copy of vector `b`.

  ```
  vector<double> d( b );
  ```

- This is a compiler error because no constructor exists to create an int vector from a double vector. These are different types.

  ```
  vector<int> e( b );
  ```

4.12 Exercises

1. After the above code constructing the three vectors, what will be output by the following statement?

   ```
   cout << a.size() << endl
        << b.size() << endl
        << c.size() << endl;
   ```

2. Write code to construct a vector containing 100 doubles, each having the value 55.5.

3. Write code to construct a vector containing 1000 doubles, containing the values 0, 1, $\sqrt{2}$, $\sqrt{3}$, $\sqrt{4}$, $\sqrt{5}$, etc. Write it two ways, one that uses `push_back` and one that does not use `push_back`.

4.13 Example: Compute the Histogram

```
//  Program:  compute_histogram.cpp
//  Author:   Chuck Stewart
//  Purpose:  Compute and output a histogram of a set of grades.
//            Demonstrates fixing the size of the vector when it is
//            constructed.

#include <cmath>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <vector>

const int BIN_SIZE = 10;

int main( int argc, char* argv[] ) {

  if ( argc != 2 ) {
    std::cerr << "Usage: " << argv[0] << " grades-file\n";
    return 1;
  }
  std::ifstream grades_str( argv[1] );
  if ( !grades_str ) {
    std::cerr << "Can not open the grades file " << argv[1] << "\n";
    return 1;
  }

  std::vector<int> scores;  // Vector to hold the input scores; initially empty.
  int x;                    // Input variable

  //  Read the scores, as before
  while ( grades_str >> x ) {
    scores.push_back(x);
  }

  //  Quit with an error message if too few scores.
  if ( scores.size() == 0 ) {
    std::cout << "No scores entered.  Please try again!" << std::endl;
    return 1;
  }

  //  Find the maximum value
  int max_value = scores[0];
  for ( unsigned int i=1; i<scores.size(); ++i ) {
    if ( scores[i] > max_value) max_value = scores[i];
  }

  //  Establish the number of histogram bins
  unsigned int num_bins = max_value / BIN_SIZE + 1;

  //  Initialize the vector called histogram to have size num_bins and
  //  to have a 0 at each entry of the vector.
  std::vector< int > histogram( num_bins, 0 );

  //  Now fill in the histogram.  Each score maps to a location in the
  //  histogram.
  for ( unsigned int i=0; i<scores.size(); ++i ) {
    int bin = scores[i] / BIN_SIZE;
    histogram[ bin ] ++ ;
  }

  //  Output the histogram
  for ( unsigned int b=0; b<num_bins; ++b ) {
    int lower = b * BIN_SIZE;
    int upper = lower + BIN_SIZE - 1;
    std::cout << '[' << std::setw(3) << lower << ".." << std::setw(3) << upper
      << "): " << std::setw(3) << histogram[b] << '\n';
  }

  return 0;  //  Everything ok
}
```

4.14 Example: Alphabetize Strings

```
//  Program:  alphabetize.cpp
//  Author:   Chuck Stewart
//  Purpose:  Demonstrate using a vector of strings and sorting.

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

int main( int argc, char* argv[] ) {

  if ( argc != 3 ) {
    std::cerr << "Usage: " << argv[0] << " names-in names-out\n";
    return 1;
  }
  std::ifstream names_in_str( argv[1] );
  if ( !names_in_str ) {
    std::cerr << "Can not open the names file " << argv[1] << "\n";
    return 1;
  }
  std::ofstream names_out_str( argv[2] );
  if ( !names_out_str ) {
    std::cerr << "Can not open the output names file " << argv[2] << "\n";
    return 1;
  }

  std::vector<std::string> names;
  std::string one_name;

  //  Read the strings one at a time and add them to the back of the
  //  vector.  The reading loop ends when the end of file has been reached.
  while ( names_in_str >> one_name ) {
    names.push_back( one_name );
  }

  //  Sort the vector of strings in the same manner that we sorted the
  //  vector of doubles.  The sort function uses (automatically) the <
  //  operator which is defined on strings.  This operator compares
  //  strings "lexicographically".
  std::sort( names.begin(), names.end() );

  names_out_str << "\n" << "Here are the names in alphabetical order." << std::endl;
  for ( unsigned int i=0; i<names.size(); ++i ) {
    names_out_str << names[i] << std::endl;
  }

  return 0;
}
```