CSCI-1200 Data Structures — Fall 2014
Lecture 2 — STL Strings & Vectors

Announcements

- HW 1 is available on-line through the website (on the "Calendar").
- TA & instructor office hours are posted on website ("Weekly Schedule") and will begin next week.
- If you have not resolved issues with the C++ environment on your laptop, please do so immediately.
- If you cannot access the LMS site or the homework submission server, please email the instructor ASAP.
- Because many students were dealing with lengthy compiler/editor installation, wireless network problems, etc., we will allow (for the first lab only!) students to get checked off for any remaining Lab 1 checkpoints at the beginning of next week's Lab 2.

Today

- Finish Lecture 1
  - Value Parameters & Reference Parameters
  - STL Strings
  - L-values vs. R-values
- STL Vectors as "smart arrays"

2.1 Standard Library (STL) Vectors

- Example Motivating Problem: Read an unknown number of grades and compute some basic statistics such as the *mean* (average), *standard deviation*, *median* (middle value), and *mode* (most frequently occurring value).
- Accomplishing this requires the use of vectors. Why can't it be done (easily) with C-style arrays?

2.2 STL Vectors: "C++ Arrays"

- 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.
  - A vector knows how many elements it stores! (unlike C arrays)
  - 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()`.

2.3 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 );
  ```

2.4 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`.

2.5 Example: Using Vectors to Compute 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

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

```
// Compute the average and standard deviation of an input set of grades.
#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.good()) {
    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;   // program exits with error code = 1
  }

  // Compute and output the average value.
  int sum = 0;
  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.
  double sum_sq_diff = 0.0;
  for (unsigned int i=0; i<scores.size(); ++i) {
    double diff = scores[i] - average;
    sum_sq_diff += diff*diff;
  }
  double std_dev = sqrt(sum_sq_diff / (scores.size()-1));
  std::cout << "The standard_deviation of " << scores.size()
            << " grades is " << std::setprecision(3) << std_dev << std::endl;

  return 0;   // everything ok
}
```

2.6 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.

2.7 Example: Computing the Median

The median value of a sequence is less than half of the values in the sequence, and greater than half of the values in the sequence. 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 $a_0 \leq a_1 \leq a_2 \leq \cdots \leq a_{n-1}$ then the median is

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

```
// Compute the median value of an input set of grades.
#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;
  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_scores(scores, grades_str);   // Read the scores, as before

  // 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;
}
```

2.8 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! Depending on the size of the vector, this can be a considerable waste of memory.
- 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.