CSCI-1200 Data Structures — Fall 2019
Lecture 1 — STL Strings & STL Vectors

Instructor
Professor Barb Cutler, `cutler@cs.rpi.edu`
Lally 302, 518-276-3274

Instructional Support Coordinator
Shianne Hulbert, `hulbes@rpi.edu`
Amos Eaton (AE) 109, 518-276-6911

To ensure a timely response, please use: `ds_instructors@cs.rpi.edu`

Announcements

- See the weekly schedule for office hours:
  http://www.cs.rpi.edu/academics/courses/fall19/csci1200/schedule.php
  - The TAs&mentors will hold normal office hours today 4-6pm (AE 215).
  - There will be no office hours on Monday (holiday).
  - Prof Cutler will hold her Monday office hours on Tuesday Sept 3rd, 1-3pm (Lally 302).
  - The TAs&mentors will hold normal office hours 6-8pm on Tuesday (AE 215).
- Please use the Submitty Discussion Forum to ask questions.
  Check first to see if a similar question has already been asked & answered.

Your TODO list:

1. Be registered. *If needed, bring the instructor a "closed course" form after lecture today.*
2. Obtain an iClicker. *We'll use them starting with Lecture 2 on Sept 6th.*
3. Complete the installation & test your C++ Development Environment.
   http://www.cs.rpi.edu/academics/courses/fall19/csci1200/development_environment.php
4. Work through Lessons 1-6 in the *Crash Course in C++ Syntax.*
   http://www.cs.rpi.edu/academics/courses/fall19/csci1200/crash_course_cpp_syntax.php
5. Start working on Lab 1, Checkpoint 1 (for Wednesday, Sept 4th) – already posted on the Calendar:
   http://www.cs.rpi.edu/academics/courses/fall19/csci1200/calendar.php
6. Additional *Crash Course* Lessons on Strings, Vectors, and Functions will be released this weekend.
7. Lab 1, Checkpoint 2 will be available this weekend.
   Note: Typically 2 of the 3 checkpoints for Wednesday's lab are posted after Tuesday's lecture. The final checkpoint will be available at the start of lab.
8. Homework 1 will be posted this weekend (due Thursday, Sept 5th).

Feeling overwhelmed? Yes, that's alot! We're getting a quick start to the semester.
Most students in the course will also be switching programming languages.

Today

- Additional Discussion of Website & Syllabus
  http://www.cs.rpi.edu/academics/courses/fall19/csci1200/
- STL Strings are "smart" & easy-to-use (vs. C-style `char` arrays)
- STL Vectors are "smart" & easy-to-use (vs. C-style arrays)

1.1 C-style Arrays

Crash Course in C++: Lesson #6

- An array is a fixed-length, consecutive sequence of objects all of the same type. The following declares an array with space for 15 double values. Note the spots in the array are currently *uninitialized*.

```
double a[15];
```

- The values are accessed through subscripting operations. The following code assigns the value 3.14159 to location `i=5` of the array. Here `i` is the *subscript* or *index*.

```
int i = 5;
a[i] = 3.14159;
```

- In C/C++, array indexing starts at 0.
- Arrays are fixed size, and each array knows *NOTHING* about its own size. The programmer must keep track of the size of each array (often storing it in an additional helper variable).

1.2 Character Arrays and String Literals (a.k.a., "C-style strings")

- In the line below `"Hello!"` is a *string literal*. The type of this value is a character array, which can be written as `char*` or `char[]`.

```
cout << "Hello!" << endl;
```

- A char array variable can be initialized as:

```
char h1[] = {'H', 'e', 'l', 'l', 'o', '!', '\0'};
char h2[] = "Hello!";
char* h3 = "Hello!";
```

  In all 3 examples, the variable stores 7 characters, the last one being the special null character, '`\0`'.

- The C language provides many functions for manipulating these "C-style strings". We won't cover them much in this course because "C++ style" STL string library is much more logical and easier to use.
- We will use `char` arrays for file names and command-line arguments, which you will use in Homework 1.

1.3 Editing `char` arrays & L-Values vs. R-Values

- Consider the simple code below. The `char` array `a` becomes `"Tim"`. No big deal, right?

```
char* a = "Kim";
char* b = "Tom";
a[0] = b[0];
```

- Let's look more closely at the line: `a[0] = b[0];` and think about what happens.

  In particular, what is the difference between the use of `a[0]` on the left hand side of the assignment statement and `b[0]` on the right hand side?

- Syntactically, they look the same. But,
  - The expression `b[0]` gets the char value, `'T'`, from location 0 in `b`. This is an *r-value*.
  - The expression `a[0]` gets a reference to the memory location associated with location 0 in `a`. This is an *l-value*.
  - The assignment operator stores the value in the referenced memory location.

  The difference between an *r-value* and an *l-value* will be especially significant when we get to writing our own operators later in the semester

- What's wrong with this code?

```
char* foo = "hello";
foo[2] = 'X';
cout << foo;
'X' = foo[3];
cout << foo;
```

  Your C++ compiler will complain with something like: "`non-lvalue in assignment`"

- Note: Strings in Python are immutable, and there is no difference between a string and a char in Python. Thus, `'a'` and `"a"` are both strings in Python, not individual characters.

  In C++ & Java, single quotes create a character type (exactly one character) and double quotes create a string of 0, 1, 2, or more characters.

1.4 About STL String Objects

- A `string` is an object type defined in the standard library to contain a sequence of characters.
- The string type, like all types (including `int`, `double`, `char`, `float`), defines an interface, which includes construction (initialization), operations, functions (methods), and even other types(!).
- When an object is created, a special function is run called a "constructor", whose job it is to initialize the object. There are several ways of constructing string objects:
  - By default to create an empty string:
    `std::string my_string_var;`
  - With a specified number of instances of a single char:
    `std::string my_string_var2(10, ' ');`
  - From another string:
    `std::string my_string_var3(my_string_var2);`
- The notation `my_string_var.size()` is a call to a function `size` that is defined as a **member function** of the `string` class. There is an equivalent member function called `length`.
- Input to string objects through streams (e.g. reading from the keyboard or a file) includes the following steps:
  1. The computer inputs and discards white-space characters, one at a time, until a non-white-space character is found.
  2. A sequence of non-white-space characters is input and stored in the string. This overwrites anything that was already in the string.
  3. Reading stops either at the end of the input or upon reaching the next white-space character (without reading it in).
- The (overloaded) operator '+' is defined on strings. It concatenates two strings to create a third string, without changing either of the original two strings.
- The assignment operation '=' on strings overwrites the current contents of the string.
- The individual characters of a string can be accessed using the subscript operator `[]` (similar to arrays).
  - Subscript 0 corresponds to the first character.
  - For example, given `std::string a = "Susan";` Then `a[0] == 'S'` and `a[1] == 'u'` and `a[4] == 'n'`.
- Strings define a special type `string::size_type`, which is the type returned by the string function `size()` (and `length()`).
  - The `::` notation means that `size_type` is defined within the scope of the `string` type.
  - `string::size_type` is generally equivalent to `unsigned int`.
  - You may see have compiler warnings and potential compatibility problems if you compare an `int` variable to `a.size()`.
- We regularly convert/cast between C-style & C++-style (STL) strings. For example:

```
std::string s1( "Hello!" );
char* h = "Hello!";
std::string s2( h );
std::string s3 = std::string(h);
```

  You can obtain the C-style string from a standard string using the member function `c_str`, as in `s1.c_str()`.

This seems like a lot to remember. Do I need to memorize this? Where can I find all the details on `string` *objects?*

1.5 Note about Strings in Java & the `new` Keyword

- Standard C++ library `std::string` objects behave like a combination of Java `String` and `StringBuffer` objects. If you aren't sure of how a `std::string` member function (or operator) will behave, check its semantics or try it on small examples (or both, which is preferable).
- Java objects must be created using `new`, as in:

  ```
  String name = new String("Chris");
  ```

  This is not necessary in C++. The C++ (approximate) equivalent to this example is:

  ```
  std::string name("Chris");
  ```

  Note: There is a `new` operator in C++ and its behavior is somewhat similar to the `new` operation in Java. We will study it in a couple weeks.

1.6 Standard Template Library (STL) Vectors: Motivation

- Example 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).
- Our solution to this problem will be much more elegant, robust, & less error-prone if we use the STL `vector` class. Why would it be more difficult/wasteful/buggy to try to write this using C-style (dumb) arrays?

1.7 STL Vectors: a.k.a. "C++-Style", "Smart" Arrays

Crash Course in C++: Lesson #8

- 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:

  ```
  std::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()`.

1.8 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`.

```
std::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;
std::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.

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

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

```
std::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.

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

1.9 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`.

1.10 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
}
```

1.11 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.

1.12 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;
}
```

1.13 Passing Vectors (and Strings) As Parameters

Crash Course in C++: Lesson #10

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`.
  - Note: This is very different from the behavior of C-style arrays as parameters, which are always passed by pointer (even without the & reference). (We'll talk about pointers in a couple weeks!)
- 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.