CSCI-1200 Data Structures — Fall 2022
Lecture 1 — Introduction to C++, Strings, and Vectors

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

Instructional Support Coordinator
Shianne Hulbert, `hulbes@rpi.edu`

Personal/confidential matters – contact both of us: `ds_instructors@cs.rpi.edu`

For general course questions, please use the Discussion Forum:
https://submitty.cs.rpi.edu/courses/f22/csci1200/forum

Today

- Brief Discussion of Website & Syllabus
  http://www.cs.rpi.edu/academics/courses/fall22/csci1200/
- Crash Course in C++ Syntax
  http://www.cs.rpi.edu/academics/courses/fall22/csci1200/crash_course_cpp_syntax.php
- Introduction to C++
- 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 Transitioning from Python to C++ (from CSCI-1100 Computer Science 1)

- Python is a great language to learn the power and flexibility of programming and computational problem solving. This semester we will work in C++ and study lower level programming concepts, focusing on details including efficiency and memory usage.
- Outside of this class, when working on large programming projects, you will find it is not uncommon to use a mix of programming languages and libraries. The individual advantages of Python and C++ (and Java, and Perl, and C, and bash, ...) can be combined into an elegant (or terrifyingly complex) masterpiece.

1.2 Compiled Languages vs. Interpreted Languages

- C/C++ is a *compiled language*, which means your code is processed (compiled & linked) to produce a low-level machine language executable that can be run on your specific hardware. You must re-compile & re-link after you edit any of the files – although a smart development environment or `Makefile` will figure out what portions need to be recompiled and save some time (especially on large programming projects with many lines of code and many files). Also, if you move your code to a different computer you will usually need to recompile. Generally the extra work of compilation produces an efficient and optimized executable that will run fast.
- In contrast, many newer languages including Python, Java, & Perl are *interpreted languages*, that favor incremental development where you can make changes to your code and immediately run all or some of your code without waiting for compilation. However, an interpreted program will almost always run slower than a compiled program.
- These days, the process of compilation is almost instantaneous for simple programs, and in this course we encourage you to follow the same incremental editing & frequent testing development strategy that is employed with interpreted languages.
- Finally, many interpreted languages have a Just-In-Time-Compiler (JIT) that can run an interpreted programming language and perform optimization on-the-fly resulting in program performance that rivals optimized compiled code. Thus, the differences between compiled and interpreted languages are somewhat blurry.
- You will practice the cycle of coding & compilation & testing during Lab 1 & Homework 1. You are encouraged to try out different development environments (code editor & compiler) and quickly settle on one that allows you to be most productive. Ask the your lab TAs & mentors about their favorite programming environments! The course website includes many helpful links as well.
- As you see in today's handout, C++ has more required punctuation than Python, and the syntax is more restrictive. The compiler will proofread your code in detail and complain about any mistakes you make. Even long-time C++ programmers make mistakes in syntax, and with practice you will become familiar with the compiler's error messages and how to correct your code.

1.3 A Sample C++ Program: Find the Roots of a Quadratic Polynomial

```
#include <iostream>   // library for reading & writing from the console/keyboard
#include <cmath>      // library with the square root function & absolute value
#include <cstdlib>    // library with the exit function


// Returns true if the candidate root is indeed a root of the polynomial a*x*x + b*x + c = 0
bool check_root(int a, int b, int c, float root) {
  // plug the value into the formula
  float check = a * root * root + b * root + c;
  // see if the absolute value is zero (within a small tolerance)
  if (fabs(check) > 0.0001) {
    std::cerr << "ERROR:  " << root << " is not a root of this formula." << std::endl;
    return false;
  } else {
    return true;
  }
}

/* Use the quadratic formula to find the two real roots of polynomial.   Returns
true if the roots are real, returns false if the roots are imaginary.  If the roots
are real, they are returned through the reference parameters root_pos and root_neg. */
bool find_roots(int a, int b, int c, float &root_pos, float &root_neg) {
  // compute the quantity under the radical of the quadratic formula
  int radical = b*b - 4*a*c;
  // if the radical is negative, the roots are imaginary
  if (radical < 0) {
    std::cerr << "ERROR:  Imaginary roots" << std::endl;
    return false;
  }
  float sqrt_radical = sqrt(radical);
  // compute the two roots
  root_pos = (-b + sqrt_radical) / float(2*a);
  root_neg = (-b - sqrt_radical) / float(2*a);
  return true;
}

int main() {
  // We will loop until we are given a polynomial with real roots
  while (true) {
    std::cout << "Enter 3 integer coefficients to a quadratic function: a*x*x + b*x + c = 0" << std::endl;
    int my_a, my_b, my_c;
    std::cin >> my_a >> my_b >> my_c;
    // create a place to store the roots
    float root_1, root_2;
    bool success = find_roots(my_a,my_b,my_c, root_1,root_2);
    // If the polynomial has imaginary roots, skip the rest of this loop and start over
    if (!success) continue;
    std::cout << "The roots are: " << root_1 << " and " << root_2 << std::endl;
    // Check our work...
    if (check_root(my_a,my_b,my_c, root_1) && check_root(my_a,my_b,my_c, root_2)) {
      // Verified roots, break out of the while loop
      break;
    } else {
      std::cerr << "ERROR:  Unable to verify one or both roots." << std::endl;
      // if the program has an error, we choose to exit with a
      // non-zero error code
      exit(1);
    }
  }
  // by convention, main should return zero when the program finishes normally
  return 0;
}
```

1.4 Some Basic C++ Syntax

Crash Course in C++: Lesson #3

- Comments are indicated using `//` for single line comments and `/*` and `*/` for multi-line comments.
- `#include` asks the compiler for parts of the standard library and other code that we wish to use (e.g. the input/output stream function `std::cout`).
- `int main()` is a necessary component of all C++ programs; it returns a value (integer in this case) **and** it may have parameters.
- `{ }`: the curly braces indicate to C++ to treat *everything* between them as a unit.

1.5 The C++ Standard Library, a.k.a. "STL"

Crash Course in C++: Lesson #4

- The standard library contains types and functions that are important extensions to the core C++ language. We will use the standard library to such a great extent that it will feel like part of the C++ core language. `std` is a *namespace* that contains the standard library.
- I/O streams are the first component of the standard library that we see. `std::cout` ("console output") and `std::endl` ("end line") are defined in the standard library header file, `iostream`

1.6 Variables and Types

Crash Course in C++: Lesson #1

- A variable is an object with a name. A name is C++ identifier such as "`a`", "`root_1`", or "`success`".
- An object is computer memory that has a type. A type (e.g., `int`, `float`, and `bool`) is a structure to memory and a set of operations.
- For example, a `float` is an object and each `float` variable is assigned to 4 bytes of memory, and this memory is formatted according IEEE floating point standards for what represents the exponent and mantissa. There are many operations defined on floats, including addition, subtraction, printing to the screen, etc.
- In C++ and Java the programmer must specify the data type when a new variable is declared. The C++ compiler enforces type checking (a.k.a. *static typing*). In contrast, the programmer does not specify the type of variables in Python and Perl. These languages are *dynamically-typed* — the interpreter will deduce the data type at runtime.

1.7 Expressions, Assignments, and Statements

Crash Course in C++: Lesson #2 & #3

Consider the *statement*: `root_pos = (-b + sqrt_radical) / float(2*a);`

- The calculation on the right hand side of the `=` is an expression. You should review the definition of C++ arithmetic expressions and operator precedence from any reference textbook. The rules are pretty much the same in C++ and Java and Python.
- The value of this expression is assigned to the memory location of the float variable `root_pos`. Note also that if all expression values are type `int` we need a *cast* from `int` to `float` to prevent the truncation of integer division.

1.8 Conditionals and IF statements

Crash Course in C++: Lesson #5 & #7

- The general form of an if-else statement is

```
if (conditional-expression)
   statement;
else
   statement;
```

- Each `statement` may be a single statement, such as the `cout` statement above, or multiple statements delimited by curly braces `{ ... }` (a.k.a. scope).

1.9 `for` & `while` Loops

Crash Course in C++: Lesson #6 & #7

- Here is the basic form of a `for` loop:

```
for (expr1; expr2; expr3)
   statement;
```

- `expr1` is the initial expression executed at the start before the loop iterations begin;
- `expr2` is the test applied before the beginning of each loop iteration, the loop ends when this expression evaluates to `false` or `0`;
- `expr3` is evaluated at the very end of each iteration;
- `statement` is the "loop body"

- Here is the basic form of a `while` loop:

```
while (expr)
   statement;
```

`expr` is checked before entering the loop and after each iteration. If `expr` evaluates the false the loop is finished.

1.10 Functions and Arguments

Crash Course in C++: Lesson #11

- Functions are used to:
  - Break code up into modules for ease of programming and testing, and for ease of reading by other people (never, ever, under-estimate the importance of this!).
  - Create code that is reusable at several places in one program and by several programs.
- Each function has a sequence of parameters and a return type. The function *prototype* below has a return type of `bool` and five parameters.

```
bool find_roots(int a, int b, int c, float &root_pos, float &root_neg);
```

- The order and types of the parameters in the calling function (the main function in this example) must match the order and types of the parameters in the function prototype.

1.11 Value Parameters and Reference Parameters

Crash Course in C++: Lesson #12

- What's with the `&` symbol on the 4th and 5th parameters in the `find_roots` function prototype?
- Note that when we call this function, we haven't yet stored anything in those two root variables.

```
float root_1, root_2;
bool success = find_roots(my_a,my_b,my_c, root_1,root_2);
```

- The first first three parameters to this function are *value parameters*.
  - These are essentially local variables (in the function) whose initial values are *copies* of the values of the corresponding argument in the function call.
  - Thus, the value of `my_a` from the main function is used to initialize `a` in function `find_roots`.
  - Changes to value parameters within the called function do NOT change the corresponding argument in the calling function.
- The final two parameters are *reference parameters*, as indicated by the `&`.
  - Reference parameters are just aliases for their corresponding arguments. No new objects are created.
  - As a result, changes to reference parameters are changes to the corresponding variables (arguments) in the calling function.
- In general, the "Rules of Thumb" for using value and reference parameters:
  - When a function (e.g., `check_root`) needs to provide just one result, make that result the return value of the function and pass other parameters by value.
  - When a function needs to provide more than one result (e.g., `find_roots`, these results should be returned using multiple reference parameters.
- We'll see more examples of the importance of value vs. reference parameters as the semester continues.

1.12 C-style Arrays

Crash Course in C++: Lesson #8

- 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).

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

Crash Course in C++: Lesson #9

- 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.

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

Crash Course in C++: Lesson #12

- 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.

1.15 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?*

1.16 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?

1.17 STL Vectors: "C++-Style", "Smart" 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:

```
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()`.

1.18 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 );
```

1.19 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;
  // 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
}
```

1.20 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.

1.21 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;
}
```

1.22 Passing Vectors (and Strings) As Parameters

Crash Course in C++: Lesson #12

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.