CSCI-1200 Data Structures — Fall 2012
Lecture 2 — Strings, Vectors and Recursion

Announcements

- HW 1 is available on-line through the course website.
- If you have not resolved issues with the C++ environment on your laptop, please do so immediately.
- If you are still having any problems with the homework submission server, please see the instructor ASAP.

Today

- Finish Lecture 1:
  - Pass-by-value vs. Pass-by-reference
  - Algorithm Analysis & Order Notation
- STL Strings
- Loop Invariants
- STL Vectors as "smart arrays"
- Basic recursion

2.1 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 will have compiler warnings and potential compatibility problems if you compare an `int` variable to `a.size()`.

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

2.2 C++ vs. Java

- 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.

2.3 Example Problem: Writing a Name Along a Diagonal

Let's write a simple program to read in a name and then write it along a diagonal, framed by asterisks. Here's how the program should behave:

```
What is your first name? Bob

*******
*     *
* B   *
*  o  *
*   b *
*     *
*******
```

2.4 Problem Solving

- It's often a good idea to start by solving a simpler version of the problem and then look for ways to extend the solution solve the whole problem. For example, first tackle writing the name diagonally, test it and make sure it works, then add the frame around the name.
- For the whole problem, there are two main difficulties:
  - Making sure that we can put the characters in the right places on the right lines.
  - Getting the asterisks in the right positions and getting the right number of blanks on each line.
- There are often many ways to solve a programming problem. Sometimes you can think of several, while sometimes you struggle to come up with one.
- When you have finished a problem or when you are thinking about programming examples, it is useful to think about the core ideas used. If you can abstract and understand these ideas, you can later apply them to other problems.

2.5 A Sample Solution: Grid of Characters

- For the first solution, let's think about the main output: think of the output region as a 2D grid of rows and columns: How big is this region? What gets output where?
- This leads to an implementation with two nested loops, and conditionals used to guide where characters should be printed.

```
#include <iostream>
#include <string>
using namespace std;

int main() {
  cout << "What is your first name? ";
  string first;
  cin >> first;
  const string star_line(first.size()+4, '*');
  const string blanks(first.size()+2, ' ');
  const string empty_line = '*' + blanks + '*';
```

```
  cout << endl << star_line << endl << empty_line << endl;
  // Output the interior of the framed greeting, one line at a time.
  for (string::size_type i = 0; i < first.size(); i++) {
    // Outputs the i-th row: *'s in the first (0-th) and last columns,
    // the i-th letter in column i+2, and a blank everywhere else.
    for (string::size_type j = 0; j < first.size()+4; ++ j) {
      if (j == 0 || j == first.size()+3)
        cout << '*';
      else if (j == i+2)
        cout << first[i];
      else
        cout << ' ';
    }
    cout << endl;
  }
  cout << empty_line << endl << star_line << endl;
  return 0;
}
```

2.6 Aside: Ending a Line of Output

- There are two common ways to end a line of output in a C++ program: `std::cout << '\n';` and `std::cout << std::endl;` What is the difference?
- C++ streams store their output in an *output buffer*. This buffer is not immediately written to a file or displayed on your screen. The reason is that the writing process is much slower than the other computations. It is much faster overall when output is buffered and then done "all at once" — in large chunks — when the buffer is full.
- Just outputting the `'\n'` — the end-of-line character — just adds one more character to the buffer. Outputting `std::endl` has two effects: outputting the `'\n'` character, **and** causing the buffer to be "flushed" — actually output to the file or screen.
- If your program crashes, the contents of the output buffer are lost and not actually output. As a result, when looking at your output, if you use `'\n'` it may appear that your program crashed much earlier than it actually did. Therefore, using `std::endl` helps greatly with debugging.
- However, using `std::endl` can slow down a program. Therefore, when a program is fully debugged (and needs to run at a reasonable speed), `std::endl` should be replaced by `'\n'`.

2.7 L-Values and R-Values

- Consider the simple code below. String `a` becomes `"Tim"`. No big deal, right? Wrong!

```
string a = "Kim";
string b = "Tom";
a[0] = b[0];
```

- Let's look 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 string location 0 in `b`. This is an *r-value*.
  - The expression `a[0]` gets a reference to the memory location associated with string 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

- Has anyone seen the error message: "`non-lvalue in assignment`"? What's wrong with this code?

```
std::string foo = "hello";
foo[2] = 'X';
cout << foo;
'X' = foo[3];
cout << foo;
```

2.8 Loop Invariants

- Definition: a *loop invariant* is a logical assertion that is true at the start of each iteration of a loop.
- An invariant can be stated in a comment, but it is not part of the actual code. It helps determine:
  - The conditions that may be assumed to be true at the start of each iteration.
  - What should happen in each iteration.
  - What must be done before the next iteration to restore the invariant.
- Analyzing the code relative to the stated invariant also helps explain the code and think about its correctness.
- The `assert` function (`#include <cassert>`) can be used to verify loop invariants and help debug programs.

2.9 Second Sample Solution: Loop Invariant Practice

- Think about what changes from one line to the next. Suppose we had a "blank line" string, containing only the beginning and ending asterisks and the spaces between. We could overwrite the appropriate blank character, output the string, and then replace the blank character (and restoring the loop invariant).

```
#include <iostream>
#include <string>

using std::cin;
using std::cout;
using std::endl;
using std::string;

int main() {
  cout << "What is your first name? ";
  string first;
  cin >> first;

  const string star_line(first.size()+4, '*');
  const string blanks(first.size()+2, ' ');
  const string empty_line = '*' + blanks + '*';
  string one_line = empty_line;

  cout << '\n' << star_line << '\n' << empty_line << endl;









  cout << empty_line << '\n' << star_line << endl;

  return 0;
}
```

Be sure to practice adding assertions and/or comments with your assumptions about the loop invariant.

2.10 Standard Library (STL) Vectors

- 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 arrays?

2.11 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.

2.12 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()`.

2.13 Example: Using Vectors to Compute Standard Deviation

```
// 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) {
    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;
  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;
}
```

2.14 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 $a_0 \leq a_1 \leq a_2 \leq \cdots \leq a_{n-1}$ then the median is

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

- Sorting is therefore the key to finding the median.

2.15 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.

2.16 Example: Computing the Median

```
// 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;
}
```

2.17 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.

2.18 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`.

```
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 );
```

2.19 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`.

2.20 Recursive Definitions of Factorials and Integer Exponentiation

- Factorial is defined for non-negative integers as

$$n! = \begin{cases} n \cdot (n-1)! & n > 0 \\ 1 & n == 0 \end{cases}$$

- Computing integer powers is defined as:

$$n^p = \begin{cases} n \cdot n^{p-1} & p > 0 \\ 1 & p == 0 \end{cases}$$

- These are both examples of *recursive definitions*.

2.21 Recursive C++ Functions

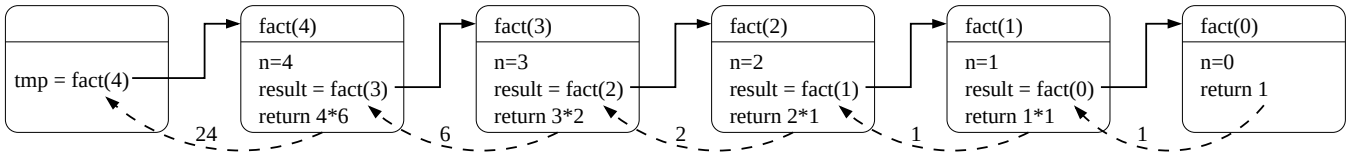
C++, like other modern programming languages, allows functions to call themselves. This gives a direct method of implementing recursive functions. Here are the recursive implementations of factorial and integer power:

```
int fact(int n) {
  if (n == 0) {
    return 1;
  } else {
    int result = fact(n-1);
    return n * result;
  }
}
```

```
int intpow(int n, int p) {
  if (p == 0) {
    return 1;
  } else {
    return n * intpow( n, p-1 );
  }
}
```

2.22 The Mechanism of Recursive Function Calls

- For each recursive call (or any function call), a program creates an *activation record* to keep track of:
  - **Completely separate instances** of the parameters and local variables for the newly-called function.
  - The location in the calling function code to return to when the newly-called function is complete. (Who asked for this function to be called? Who wants the answer?)
  - Which activation record to return to when the function is done. For recursive functions this can be confusing since there are multiple activation records waiting for an answer from the same function.
- This is illustrated in the following diagram of the call `fact(4)`. Each box is an activation record, the solid lines indicate the function calls, and the dashed lines indicate the returns. Inside of each box we list the parameters and local variables and make notes about the computation.



- This chain of activation records is stored in a special part of program memory called *the stack*.

2.23 Iteration vs. Recursion

- Each of the above functions could also have been written using a `for` or `while` loop, i.e. *iteratively*. For example, here is an iterative version of factorial:

```
int ifact(int n) {
  int result = 1;
  for (int i=1; i<=n; ++i)
    result = result * i;
  return result;
}
```

- Often writing recursive functions is more natural than writing iterative functions, especially for a first draft of a problem implementation.
- You should learn how to recognize whether an implementation is recursive or iterative, and practice rewriting one version as the other. Note: We'll see that not all recursive functions can be *easily* rewritten in iterative form!
- Note: The order notation for the number of operations for the recursive and iterative versions of an algorithm is usually the same. However in C, C++, Java, and some other languages, *iterative functions are generally faster than their corresponding recursive functions*. This is due to the overhead of the function call mechanism. Compiler optimizations will sometimes (but not always!) reduce the performance hit by automatically eliminating the recursive function calls. This is called *tail call optimization*.

2.24 Exercises

1. Draw a picture to illustrate the activation records for the function call

   ```
   cout << intpow(4, 4) << endl;
   ```

2. Write an iterative version of `intpow`.

2.25 Rules for Writing Recursive Functions

Here is an outline of five steps that are useful in writing and debugging recursive functions. Note: You don't have to do them in exactly this order...

1. Handle the base case(s).
2. Define the problem solution in terms of smaller instances of the problem. Use *wishful thinking*, i.e., if someone else solves the problem of `fact(4)` I can extend that solution to solve `fact(5)`. This defines the necessary recursive calls. It is also the hardest part!
3. Figure out what work needs to be done before making the recursive call(s).
4. Figure out what work needs to be done after the recursive call(s) complete(s) to finish the computation. (What are you going to do with the result of the recursive call?)
5. Assume the recursive calls work correctly, but make sure they are progressing toward the base case(s)!