

CSCI-1200 Computer Science II — Spring 2006

Lecture 3 — Strings

Review from Lectures 1 and 2

- Standard library and I/O streams
- Expressions and statements
- Variables, objects, types
- Loops (for & while)
- If-else conditionals, logic, switch statements
- Arrays
- Functions and parameter passing (value & reference)
- Scope
- Algorithm analysis and order notation

What's Next? C++ classes and the C++ standard library

- Strings
- Vectors
- Classes
- Lists and iterators
- Maps

It will take us several weeks to cover all of this material, but by the time we are done you will have much of the preparation you need to solve challenging computational problems.

Readings for Today

- **Accelerated C++ (Koenig and Moo):** Chapters 1 and 2
- **C++ Programming (Malik):** pp 399-410

3.1 Example from Chapter 1: Strings

```
// ask for a person's name, and generate a framed greeting
#include <iostream>
#include <string>

int main()
{
    std::cout << "Please enter your first name: ";
    std::string name;
    std::cin >> name;

    // build the message that we intend to write
    const std::string greeting = "Hello, " + name + "!";

    // build the second and fourth lines of the output
    const std::string spaces( greeting.size(), ' ' );
    const std::string second = "* " + spaces + " *";

    // build the first and fifth lines of the output
    const std::string first(second.size(), '*');
```

```

    // write it all
    std::cout << std::endl;
    std::cout << first << std::endl;
    std::cout << second << std::endl;
    std::cout << "*" << greeting << " *" << std::endl;
    std::cout << second << std::endl;
    std::cout << first << std::endl;

    return 0;
}

```

3.2 About 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. The greeting example code exhibits three ways of constructing string objects:
 - By default to create an empty string
 - With a specified number of instances of a single char
 - From another string

- The notation

```
greeting.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 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.

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

3.3 Short Exercises

1. What will be the values of strings `a`, `b` and `c` at the end of the following code fragment:

```
std::string a, b, c;
std::cin >> a >> b >> c;
```

for the input:

```
all-cows   eat123
           grass.  every good boy
deserves fudge!
```

2. Write a C++ code fragment that reads in two strings, outputs the shorter string on one line of output, and then outputs the two strings concatenated together with a space between them on the second line of output.

3.4 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");
```

On the other hand, there is a `new` operator in C++ and its behavior is somewhat similar to the `new` operation in Java. We will study it later in the semester.

3.5 More on Strings

- Strings behave like arrays when using the subscript operator `[]`.
 - This gives access to the individual characters in the string
 - 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()`.

3.6 Example Problem: Writing a Name Along a Diagonal

We will spend the rest of lecture on the following problem: read in a name and then write it along a diagonal, framed by asterisks. Here's how the program might behave:

What is your first name? Peter

```
*****
*       *
* P     *
*  e    *
*   t   *
*    e  *
*     r *
*       *
*****
```

We will start by solving a simpler versions and then look at two ways to solve the whole problem.

3.7 Exercises: Writing the Name Diagonally

Write a program to output the name diagonally, so that the program interaction looks like this:

What is your first name? Juanita

```
J
 u
  a
   n
    i
     t
      a
```

Here is a start to the code:

```
#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;

    // Fill in here...

    return 0;
}
```

You may need to use nested `for` loops here, although it is possible to use just a single loop if you exploit properties of the `string` class.

3.8 Thinking About the Whole Problem

Now we are better prepared to address the original problem.

- Here are the 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.
- In the previous example, we addressed a simplified version of the first. With that practice, we are ready to try the full problem, and we will look at two different ways of solving it.
- It is good practice in general to start by solving simpler problems that address the core issue (or issues) in a larger problem.

3.9 Basic Approach for 1st Solution

- Initial stuff: read the name, and output a blank line, as in an earlier example.
- Let's think about the main output: think of the output region as a 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.

3.10 1st Solution

```
// Program:  diagonal_name
// Author:   Chuck Stewart
// Purpose:  A program that outputs an input name along a diagonal.

#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.
// The use of the while loop instead of a for loop is just for
// illustration purposes. A for loop is a more intuitive and
// therefore cleaner choice here.

string::size_type i = 0;
while ( i < first.size() ) {
    // Job of loop body is to write a row of output containing *'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;
    ++ i;
}

cout << empty_line << endl
    << star_line << endl;
return 0;
}

```

3.11 Loop Invariants

- Definition: a *loop invariant* is a logical assertion that is true at the start of each iteration of a loop.
 - In **for** loops, the “start” is defined as coming after the initialization/increment and termination test, but before execution of the next loop iteration.
- An invariant is stated in a comment; 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 must be done in each iteration.
 - What must be done at the end of each iteration to restore the invariant.
- Analyzing the code relative to the stated invariant also helps explain the code and think about its correctness.

3.12 L-Values and R-Values

- Consider the simple code

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

String `a` is now "Tim".

- No big deal, right? Wrong!
- 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.

- Anyone seen the error message: “non-lvalue in assignment”?

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

3.13 Ideas for a 2nd Solution

Here are ideas for a second solution:

- 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 intervening blanks.
- We could overwrite the appropriate blank character, output the string, and then restore the blank character.

3.14 Exercise: Finish the 2nd Solution

```
#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 << endl
         << star_line << endl
         << empty_line << endl;

    // Fill in here....

    cout << empty_line << endl
         << star_line << endl;

    return 0;
}
```

3.15 Thinking About Problem Solving

- We began by working on simplified versions of the problem to get a “feel” for the core issues.
- Then we introduced two solution approaches:
 - Thinking of the output as a two-dimensional grid and using logical operations to figure out what to output at each location.
 - Thinking of the output as a series of strings, one string per line, and then thinking about the differences between lines.
- 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.