Some Things about Strings

We've seen how to use the C++ built-in `int` type. You can create an object `n` of type `int` though a declaration: `int n;` At any point in time, such an object has one value drawn from a large (but finite) set: *some big negative integer*, ..., -3, -2, -1, 0, 1, 2, 3, ..., *some big positive integer*. The language enables you to do certain operations with `int`s, among them:

```
int k = 2 * 6; // create and initialize one int
int m = k;     // and another
int n;         // create an int with an unspecified value
n = k + 3;     // add ints and assign ints
if (m == n)    // compare ints with ==
    ...
```

The Standard C++ library enables additional operations:

```
#include <iostream>

std::cout << k;   // write a sequence of characters representing
                  // the value of an int
std::cin >> k;    // read a sequence of characters representing
                  // the value of an int, and store that value
                  // in an int variable
```

C++ has a built-in type `double`, with similar operations (but not identical, of course, because `double`s have values drawn from a different set, and the detailed semantics (i.e., meaning) of operations like division, output, and input are not the same).

The C++ *language* has little in the way of support for conveniently dealing with character strings. However, the C++ *library* does. By saying `#include <string>` you enable the creation of `std::string` objects. At any point in time, such an object has a value drawn from a large set: the set of all sequences of characters that can fit in the computer's finite memory. Examples of such string values are the character sequences between the vertical bars below:

```
|hello|         // 5 characters
|R2D2|          // 4 characters
|  #@ %*  |     // 9 characters (blanks are characters)
||              // 0 characters (the so-called "empty string")
```

You can do an awful lot with strings, but we will limit ourselves for now to a few operations:

```
using namespace std; // so we can leave off the  std::
string s = "hello";  // create and initialize one string
string t = s;        // and another
```

```
string u;            // create a string and initialize it to
                     // the empty string!
u = s;               // assign strings
if (s == "goodbye")  // compare strings with ==
```

If we also say `#include <iostream>` we can write and read strings:

```
cout << s;         // write the sequence of characters
getline(cin, s);   // read a sequence of characters up to and
                   // including the next newline, and store that
                   // sequence (without the newline) in a string
                   // variable
```

While you can also say `cin >> s;` it's less useful and more prone to not doing what you really intended. It will read and discard leading whitespace, then read and store non-whitespace characters into the string variable, stopping at (without consuming) the next whitespace character. In other words, it reads in only the next word. If the input were `This is a test` , the value stored in `s` would be `This`, and further input operations would start by reading the blank character before the `i` of `is`. We're not going to use this way of doing string input in this class; use `getline` instead.

Now for a problem. Suppose we want the following interaction:

```
How many place settings would you like to buy? 12
In which style? Floral Tapestry
```

If our program says

```
cout << "How many place settings would you like to buy? ";
int numberOfSettings;
cin >> numberOfSettings;

cout << "In which style? ";
string style;
getline(cin, style);
```

we end up with the empty string as the value of `style`. This is because the expression `cin >> numberOfSettings` consumes the `1` and `2` characters, but not the newline we typed right after the `2`. The expression `getline(cin, style)` picks up right where the previous input operation left off, so it reads all (zero) characters up to that newline, and that newline itself, and stores the zero-character sequence in `style`. To throw away any input after the `12`, we should have said

```
cout << "How many place settings would you like to buy? ";
int numberOfSettings;
cin >> numberOfSettings;
cin.ignore(10000, '\n');       // this is new

cout << "In which style? ";
```

```
string style;
getline(cin, style);
```

The expression `cin.ignore(10000, '\n')` causes characters up to and including the next newline character to be consumed and discarded. Technically, it will consume and discard either all characters up to the next newline or 10000 characters, whichever comes first. We pick a huge number so that the latter situation will never occur. Notice that we use single quotes, not double quotes, to denote the single newline character here.

The variety of syntaxes for doing input (`cin >> ...; getline(cin,...); cin.ignore(...);`) admittedly seems haphazard, but that's because we're getting only glimpses of the bigger, more systematic picture. There is certainly much more that can be done with strings other than what we've shown here, but the information in this tutorial will suffice to enable you to do Project 2.