Next: TCS 8.

Strings:
Strings are a useful data type that is found in almost all programming languages.

Strings are not actually built into C++; they provided in one of the Standard Template Libraries (STLs) and you get them by putting in your program: #include <string>

There is a primitive sort of string built into C++, and these are called "native C strings" or just "C-strings." They are kind of a hold-over from the C language, of which C++ is an extension.

The string STL is more convenient for string handling than C-strings. C++ strings can grow (i.e., strings can increase in size), they have better error checking, they have useful built-in operators (e.g., comparisons <, >, <=, >=, ==, !=; concatenation +), and member functions, such as size() and find().

When you a write a literal string, such as "Hello", that is actually a C-string.

You can declare or declare and initialize a string variable in the usual way:

```
string fred;
string phil = "Hello"; /* in this case the compiler is actually converting the C-string
"Hello" to type string so you can use it to initialize phil. */
```

You can explicitly convert a C-string to a string by using a conversion function, e.g., string("Hello").

Strings behave a lot like vectors of chars. In particular, you can index the elements of a string. For example, if

```
string fred = "Hello";
```

then fred[0] has the value 'H'; fred[1] == 'e'; and so on.
What is fred[5]?
It's undefined. That means the language/compiler makes no commitment about what it is.
Unfortunately, C++ does not check to make sure that an index is in bounds. So generally, it will let you get away with this and give you some garbage.
If you want, you can write fred.at(5) and this will do bounds checking. That means if you use an out-of-bounds subscript, you will get an error message (instead of your program doing mysterious things).

You can change elements of a string:

```
fred[0] = 'h';
```

and now fred == "hello"

What do you suppose

```
fred[5] = 'h';
```

does?
It stores the 'h' in a memory location beyond the end of fred. It could be another variable, some machine code, or some information needed by the operating system.

This is a cause of a common security loophole in operating systems called "buffer overrun." A buffer is an area of memory (think of it like string or vector) usually used for holding input coming from some device, such as a keyboard. Some programs don't check to make sure there is enough space in the buffer before they put something else into it. So you can give these programs a very long string of input that overflows the buffer and writes over the code, letting you put your code into the computer in place of what was there.

Implementing Find:

As you've seen, strings have a find() member function, which you invoke by s.find (c). It returns either the index of the first location of c in s, or it returns a special value string::npos ("no position") if c is not in s. So for example, fruit.find('q') == string::npos. You can test for that in an if-statement.

Nevertheless it's a good exercise to program your own Find function, since it's a simple example of one of the most important things computers do, namely, search.

What we want to do is program a function:

```
int Find (string s, char c) {
  // return the index of the first occurrence of c in s,
  // and return the size of the string if there's no occurrence.
}
```

Think about how you or someone else would do it. You would keep a counter, and look at each character in turn. If you found it, you would return the counter value. If you got to the end of the string, you would return the counter value (which by then is the size of the string).

We've implemented a sequential search. For any kind of search, it's important to analyze its efficiency for a string of length N.

Best case: One step, if c is the first character in s.

Worst case: N+1 steps, if c is not in the string.

Average case: Assuming that all positions are equally likely, is

[1 + 2 + … + (N+1)] / (N+1), which is approximately N/2.

This is called a linear or O(N) (order N) algorithm because its running time is proportional to the size of the input N.

There are many algorithmic complexities, e.g., O(log N) or logarithmic algorithms run in time proportional to the log of the input size.

If the data are sorted, you can search in log time. (Keep dividing the data set in half; the number of such divisions will be $\log_2$ N.)

There are also O(N^2) or quadratic algorithms, where the running time is proportional to N^2. These are not so good, but sometimes the best we have.

Exponential algorithms are O(c^N), for constant c. Very bad!

If you want to see how these compare, write a program to print out N, log N, N^2, 2^N for N = 1 to 10.

In other CS classes you will learn how to analyze the runtime complexity of algorithms.