Pop Quiz:

Define a function getTens that returns the tens-digit of a positive integer. For example getTens(1234) returns 3, getTens(6) returns 0, and getTens(87) returns 8.

Ans:

```
int getTens (int X) {
	return X % 100 / 10;
//	return X / 10 % 10; // alternate answer
}
```

Strings:

Strings are not part of the C++ core language; they are defined in one of the standard libraries. To get them you have to #include <string>.

There is a kind string data type built into C++, and they called C-strings or native C-strings. They they are kind of a primitive version of C++ strings and were inherited from the C language. By "primitive" I mean that they do less, they are more complicated to use, and they are more error-prone.
For example, C-strings cannot grow automatically, whereas C++ strings can. You can concatenate C++ strings using the "+" operator, but you cannot do this directly with C-strings.

You have been using C-strings already. Whenever you use a string literal, such as

```
	"Hello world!"
```

you are writing a C-string. When you write something like:

```
	string fred = "Hello world!";
```

the C++ compiler is automatically converting the C-string "Hello world!" to a string, which is used to initialize the string variable fred.

What can you do with strings?

(1) Strings behave a lot like vectors of characters. And so you can index a string just like you index a vector. For example, fred[2] refers to the "two-th" element of fred. And fred[N] refers to the N-th element (whatever the value of N may be).

I can change the elements of a string:

```
      fred[1] = 'a';
```

after which fred == "Hallo world!"

(2) You can concatenate strings using "+".

```
string firstName, lastName;
….
string revName = lastName + ", " + firstName;
```

(3) You can use push_back() to add characters to the end of a string.

```
string sentence;
….
sentence.push_back('.'); // adds a "." top the end of sentence
```

(4) You can compare strings, using ==, !=, <, >, <=, >=.

These compare according to the "collating sequence" of the character set.

(5) As for vectors, the size() member function returns the length of the string. For example, fred.size() == 12.

How would I print the last character of fred?

```
      cout << fred [fred.size() - 1] << endl;
```

(6) If you accidentally write fred [fred.size()], **which is always incorrect**, then you will get something mysterious.
Unfortunately, C++ does not by default check to make sure that the index is in range. It just calculates where that string element would be, if it existed, and pulls out of memory whatever is there. Even worse, if you store into that location, C++ will do the same thing, assigning to some location not in the string. It might be part of another variable, computer instructions, or something else.

Why doesn't C++ check it? It takes a couple extra instructions and a little extra time, and the designers figured that if you wanted it checked, you could do it yourself.

There is a way to do safe indexing: fred.at(N) is the same as fred[N], except that it does bounds checking.

(7) Strings have a built-in find() function, which allows you to find the position of a character in the string.

How would we define our own find function?

int find (string S, char C)

We want this to return the index of the first occurrence of C in S, or to return S.size() if C does not occur in S.
find(fred, 'l') == 2
find(fred, 'G') == 12

A common kind of sequential search procedure.

```
int find (string S, char C) {
    int pos;
    for (pos = 0; pos < S.size(); pos++) {
        if (C == S[pos]) { return pos; }
    }
    return S.size(); // return pos; would work too
}
```

This is the basic template for a sequential search.

How long does it take to run?
Best case: what I'm looking for is in position 0, so it takes 1 step.
Worst case: is it's not in the string at all, and it takes N+1 steps (where N is the length of the string).

What is the average case, assuming that what I'm looking for is somewhere in the string?

Assume that what I'm looking for is equally likely to be in any one of the N positions in the string. So there are N possibilities, each with probability 1/N. If it's in the first position it takes 1 step, in the second, 2 times through the loop, … , in the last position, N times through. So the average number of times I have to go through the loop is:

$1/N \, [\, 1 + 2 + 3 + \ldots + (N - 1) + N]$
$= (1/N) \, [N(N+1) / 2]$
$= (N+1) / 2$
$\approx N/2$ (if N is big).

This is called a linear or order-N algorithm, O(N), because in the long run the running time is proportional to N, the size of the data.