Homework (turn in at your lab next week):

Write a complete C++ program to generate 50 random dates in 2010 and prints them on the console. For example, your output might look like this:

1/25/2010
5/16/2010
12/31/2010
6/1/2010
….

Make sure your dates are legal (i.e., no Feb. 30 or Sep. 31).
Hint: use an if-ladder.

Ensembles of Data:

Often in programming, we are not dealing with single data items (single ints or doubles) but with ensembles (groups) of data. (What computers do well is deal with large amounts of data.) The data in such an ensemble can be organized in various ways, which we call a data structure (how is the data structured or arranged?).

How the data is structured determines what things can be done with it easily or efficiently, and what cannot. So an important part of any programming project is deciding the data structures to use.

"Pick the right data structure and the algorithm will design itself."
If you are careful to pick (or invent) the right data structure, then it will be very easy to design a correct, efficient algorithm to do what you need to do.
If you pick the wrong data structure, you will be fighting it and it will be difficult or

impossible to write an efficient program.

What makes data structures different from each other?

(1) How is the data accessed? Is it random access (i.e., it's equally easy to get any particular data value out of the structure)?
Is it sequential access? Which means I have to "read through it" from the beginning, which means it's easier to access things at the beginning than the end. Maybe I can read through in either direction, but still sequentially (a bidirectional or two-way sequential data structure). Are there "keys" that give me access to the data (e.g., a student number to look up student data).

(2) Is the data read-only or can it be modified? And if it can be modified, can I modify any data value without affecting the rest, or am I restricted in where I can modify it (e.g., only at the end).

(3) Can the data structure change in size (i.e., is it dynamic as opposed to static). Some structures can grow or shrink to accommodate the data, whereas others can't. The flexibility you get in one way, you will probably pay for in another way.

(4) Does all the data in the structure have to be of the same type (e.g., all ints, all strings, all doubles), or can it be mixed? (Homogeneous or heterogeneous?)

(5) Does the order of the data matter?

These are things you have to think about when you pick your data structures for some application. There are pros and cons to each, so you have to make engineering tradeoffs.

C++ has a number of built-in data structures, which have been found to be useful for programming (and that have to be built into a high-level language). (E.g. structs, C-strings, C-arrays.)

C++ also has a Standard Template Library (STL), which contains other useful data structures programmed in C++ (e.g., lists, stacks, queues, strings, vectors).

Beyond that there are whatever data structures you or anyone else has implemented. E.g. Myro.

In C++ different kinds of data structure are called classes.

LCR 5 discusses two standard C++ data structures: vectors and lists. These are classes defined in the STL.

We need to understand each data structure in terms of its characteristics (access, readable/writable, dynamic/static, etc.).

Vectors:

A vector (in C++) is an ordered collection of data of one type that can be randomly accessed to read or write. C++ vectors are dynamic (can grow or shrink).

Think of 2D vectors in math. E.g. (2, 3) is a vector whose X component is 2 and whose Y component is 3.
Or of 3D vectors. E.g. (-5, 2, 0) is a vector whose X component is -5, whose Y is 2, and whose Z is 0.
Notice that order matters: (2, 3) and (3, 2) are different vectors.
In math we can index the components of a vector. If V = (-5, 2, 0), then V_1 = -5

(the first component of V). $V_2 = 2$.

What's different about C++:

(1) The vectors can have any number of dimensions (i.e., any number of components), and often have very many. For example, if I had a vector of the amount of rain that fell on each day of the year, that would be a 365-element vector.
rainFall = (1.0, 0.0, 0.5, …)

(2) The components of a vector are numbered from 0 rather than from 1. (Some programming languages have this "0-origin indexing"; others have "1-origin indexing"; others let you pick your origin). So we have to get used to counting from zero.
In C++ we do indexing this way: V[0], V[1], V[2], ….
This talks about its parts. If we want to talk about the whole vector, it's just V.

(3) The components of a vector do not have to be numbers. They can be numbers (e.g., ints, doubles), but they can also be chars, strings, bools, or all sorts of other things, including vectors.

(4) I can calculate the index for a component of a vector. So for example V[i] is the i-th element of the vector, depending on the current value of the variable i. Or I could have V[2*k - 3]. The expression "2*k - 3" is evaluated to get the index that will be used to access a particular vector element. This ability to compute indexes is one of the most important characteristics of vectors.