For next time read LCR 5.

<u>Program Development</u>

<u>Top-down Development</u>:

You start with a goal or a task you want to accomplish.
And you break it into subtasks (subgoals).
And you keep doing this, until you get to something you know how to do.
This could be operations that are provided in the PL, or it could be things that are done in a library that you already have (e.g., compute sin, cos, log; do console I/O; etc.).
Libraries: Don't reinvent the wheel unless there is a very good reason.

Advantages of TD development: you get a well-organized, goal-oriented program. Probably easy to read. Each part has a well-defined purpose.

One disadvantage: If you do this too methodically, you may end up with subtasks that are <u>almost</u> the same, or tasks that can <u>almost</u> be accomplished with a library or built-in operator. E.g. you might have subtasks "sort a vector of doubles" and "sort a vector of strings", which might be almost identical.

It's really better to keep in mind where you are going, and look ahead in your task decomposition.
In spite of disadvantages, it is the best overall approach to programming.

The only wrinkle with TD development in C++ is that functions have to be defined before the functions that use them. So if you read the program from top to bottom, you will see the subtasks before the tasks that use them. What this means is that it's best to start at the end of a C++ program (the main function), and read backwards (function by function).

Bottom-up Development:

You start with what have got (PL and libraries) and put them together to make useful (higher level, more powerful) libraries.
And put these together to make even more powerful and useful libraries.
In electronics this is like assembling components into useful circuit boards, and these boards into useful equipment or modules, etc.
How do you know what you're assembling is going to be useful? You have to have some idea what are the useful subtasks to solve.
So the C++ standard libraries have been designed to provide facilities that experience with programming have shown to be useful.

In practice, it's usually useful to combine TD and BU development.

Incremental Development:

Based a simple idea: It's easier to take a working program and make it better, than to make the best program (your goal) from scratch.

We start with a simple program that works, but doesn't necessarily do everything we want it to do.
Then we make small (incremental) changes, making it closer to what we want, but testing at each stage.
If you ever add a piece and the program stops working, then you know the new piece is the problem. Either it has an error in itself, or it is interacting incorrectly with the previous parts.

There are only two kinds of programmers: beginners and experts. (Just ask them!)

Scaffolding:

Stuff we put in the programs to help us construct them correctly.
We intend to take it out later.
Prime example: extra output statements. Show where you are in your program, what the values of key variables are, etc.
You can put these in wherever you think there is a problem, or if you are doing incremental development, in the new part.
Sometimes you can put in input statements.
When things seem to be going right, you can delete them.
Or better, just comment them out. Then when you are really sure your program is correct, you can delete them.

Some of the scaffolding can be left in, but normally turned off unless it is turned on by a "debug switch" (set from a preference pane or a command-line option).

Modular Development and Unit Testing:

An important idea from engineering, and one of the main justifications for object-oriented programming (as in C++).

You break up your program into modules that can be tested independently (unit testing). Modules have a definite purpose and clearly defined inputs and outputs.

Then it is much more likely that the whole system will work when we put its modules together (but not guaranteed; there can unanticipated interactions).

Modular development is the best way to have a team developing software, since different sub-teams or individual programmers can work on different modules at the same time and independently.

Boolean Values:

Named for George Boole, a 19th-century English logician and mathematician that developed a lot of the ideas of formal logic. He wrote a book called The Laws of Thought. You have probably used Boolean searches in the library catalogs or the like (e.g., google).

What it's all about is true, false, and operators such as and, or, and not.

In C++ Boolean values are "first-class citizens" (i.e., they have all the "rights and responsibilities" of other data types).
What this means is that you can do the same things with bools as with ints, doubles, strings, etc.
For example, there are bool values (true = 1, false = 0), bool variables, bool-valued functions, bool parameters to functions, and bool operators.

When you say:

```
if (3 > 2) { …. }
```

It computes the bool value of 3>2 (true in this case), and the if statement looks at this value and decides what to do. (Do the then branch if it's true, and the else branch if it's false.)

The Boolean operators are && (and), || (or), ! (not).

x && y is true if and only if both x and y are true.
x || y is true if and only if x or y or both are true (inclusive or).
!x is true if and only if x is false.

There are rules for combining these operators, and for simplifying them.

DeMorgan's Laws:
```
! (x && y)  ==  (!x) || (!y)
! (x || y)    ==  (!x) && (!y)
```

Distributive Laws:
```
x && (y || z) == (x && y) || (x && z)
x || (y && z) == (x || y) && (x || z)
```

This is all part of Boolean algebra (the algebra of truth values rather than numbers). It's important in digital circuit design. Just as you simplify a formula in ordinary algebra, you can use Boolean algebra to simplify a digital circuit.

In programming bools are usually used as flags of one sort or another. The equivalent of check-boxes on a menu.