Conditional Statements:

They are basic mechanism for making decisions in a program.
C++ has an if-statement.

```
if ( <condition> ) {
  <statement-1>
  …
  <statement-n>
}
// this is where we continue if the condition is false
```

[Notice that we don't put a ";" after a "{}" around statements (for example around loop bodies, if-bodies or function-bodies).]

The if-statement (or conditional) evaluate the condition, to get true (1) or false (0), and if it gets a 1 (i.e., if the condition is true), then it executes the statements in the if-body (i.e., <statement-1> through <statement-n>).

We read the if-statement, "if <condition> then <statements>" so the <statements> are also called the then-part of the conditional.

Simple example: Print out the absolute value of a number that is typed in.

```
double X;
cin >> X;
if (X < 0) {
  X = -X;
}
```

```
cout << "The abs value is " << X << endl;
```

The above is a single-branch conditional.

A two-branch conditional looks like this:

```
if ( <condition> ) {
  <true-branch statements> // this is called the "then" branch
} else {
  <false-branch statements> // this is called the "else" branch
}
// you continue here after doing either the true- or false-branch
```

In all of the conditionals the <true branch> and the <false branch> can also contain conditionals. You can have nested if-statements.
All of these allow us to do different things depending on conditions.

N-branch conditional or chained if-statement or if-ladder:

```
if ( <cond 1> ) {
  <statements-1>
} else if ( <cond 2> ) {
  <statements-2>
} else if ( <cond 3> ) {
  <statements-3>
….
} else if ( <cond N> ) {
  <statement-N>
} else { /* this last "else part" can be omitted if there are no default statements,
but that's not such a good idea */
  <default statements>
}
// all the branches come back together here
```

A common logic error is to think you have covered all the bases, but you haven't.

```
cin >> X;
if (X > 0) {
  cout << log(X) << endl;
} else if (X < 0) {
  cout << log(-X) << endl;
}

/* Keep in mind that the negation of X>Y is X<=Y, and the negation of X<Y is
X>=Y. */

double abs (double X) {
  if (X < 0) {
    return -X;
  } else {
    return X;
  }
}

int main () {
  double number;
  cin >> number;
  cout << "log of abs value = " << abs(number) << endl;
  return 0;
}
```

This would work too:

```
double abs (double X) {
  if (X < 0) {
    X = -X;
  }
  return X;
}
```

The parameters to a function are local to a function. You can also declare local variables within a function.

To refine the analogy a little bit. When we call a function, it's kind of like setting up an office just for that task, and then when the task is done, clearing out the office so someone else can use it.

Every time you call a function in C++ it sets up a stack frame on the runtime stack.

Recursive Functions (functions that call themselves, directly or indirectly):
In effect this is putting something in your own InBox.

Example:

Suppose you are doing research project, and you run across a word you don't understand.
In effect, you call the DictionaryLookup task, by putting the word in its InBox.
So DL looks up the word, but in the process, maybe the definition contains another word that DL doesn't understand.
So DL calls itself with the second word, by setting up another DL office and putting word in its InBox.
And so on, until one of the DL tasks can be completed. Then each returns the explanation to its caller.

A recursive program to solve a problem can call itself to solve a "simpler" version of the same problem. If it keeps getting simpler, then eventually we will get the answer.

```
void CountDown (int N) {
  if (N == 0) { // base of the recursion.  Must have!
    cout << "Blastoff!\n";
  } else {
    cout << N << endl;
    // reduce to a simpler problem
    CountDown (N-1);
    cout << "Backing out of level " << N << endl;
  }
}
```