Structures:

Structures to group heterogeneous data together.

Sometimes it's helpful to think of these structures as values (abstractions, things we operate with, like numbers).
Other times it's helpful to think of these structures as objects (concrete things, which we can do things to, or ask to do things). This is really the essence of object-oriented programming.

Examples of objects: robot object (instance of the class of objects called scribblers), employee objects.

These are two different ways of thinking about data in programs, and sometimes one is more appropriate, sometimes the other, and sometimes it doesn't make much difference.

For example, normally we think of vectors as mathematical values, but we can also treat them as objects that we can modify.

For example, you can think of Times as values or objects:

```
struct Time {
    int hour, minute;
    double second;
};
```

Modules and Clean Interfaces:

Your program will be easier to design, debug, and understand if you break it up into well-defined modules:

(1) Each module should have a well-defined purpose, typically something you can state in a simple sentence.

(2) Each module should have clean interfaces with the other modules. This means it's easy to see what is going into the module and what is coming out of it. In other words, it's easy what it depends on and the effects it will have.

Example of modules with clean interfaces: pure functions.

In a pure function, the only inputs are the parameters to the function, and the output from the function is its return value.

```
Time addTime (const Time& t1, const Time& t2) {
    const double seconds = convertToSeconds(t1) + convertToSeconds(t2);
    return makeTime(seconds);
}
```

Impure functions have hidden inputs and/or outputs, which are often called side-effects. Of course, they're not completely hidden, since if you look at the program carefully, you can see them, but they're not obvious. Normally, side-effects are unintended consequences of something; in this case they may be intentional, but they are not obvious.

Example: suppose I wanted to count the number of times some function was called.

```
int max_calls = 0;
...
double max (double x, double y) {
      max_calls++;
      if (x > y) {
            return x;
      } else {
            return y;
      }
}
….
cout << max_calls << endl;
double dist = max (0, length); // not obvious it changes max_calls
cout << max_calls << endl;
```

Excessive use of impure functions can make programs harder to read, understand, and debug. But that doesn't mean you should never use them.

There are some programming languages, called functional languages (e.g., pure LISP and Scheme), which allow only pure functions, because it's often easier to design and prove correct programs in these terms. In fact, anything that can be programmed in any programming language can be programmed with if-statements and pure functions (you can prove this mathematically).

Constant Reference Parameters:

Sometimes a function needs to modify one or more of its parameters. Technically these are impure function, but it's not too bad, because you can see from the function header that the parameter can be modified. This is because if you're going to modify the actual parameter (or argument), you have to pass a reference to it, so you know where it is in memory. (If you pass it by value, the function gets a fresh copy of the argument, and anything the function does to it will only affect the copy).

Example:

```
void increment (Time& time, double secs) {
      ...
}
```

You can see that this function modifies its first parameter.

As it turns out in C++ reference parameters are used for two different purposes. One is as output parameters, the other is for efficiency. Consider:

```
void printVector (vector<double> V) {
      for (int i = 0; i < V.size(); i++) {
            cout << V[i] << endl;
      }
}
…
vector<double> data (10000);
…
printVector (data); // print the big vector
```

This will copy the entire vector data into local storage for the printVector function. This wastes both time and space. So no C++ programmer would do this. They might do the following:

```
void printVector (vector<double>& V) {
      for (int i = 0; i < V.size(); i++) {
            cout << V[i] << endl;
      }
}
```

This passes V by reference, which means that all that is copied into the function is the address of data (4 bytes).
The only problem with this is that it suggests that V is an output from the printVector function, i.e., that printVector might modify its argument. So C++ has a solution: pass by constant reference, which means pass by reference, but I'm not allowed to change the parameter (it's a constant):

```
void printVector (const vector<double>& V) {
      for (int i = 0; i < V.size(); i++) {
            cout << V[i] << endl;
      }
}
```

This is extra protection; it prevents me from accidentally modifying the parameter. It also informs the reader that although the parameter is passed by reference, it's only an input (not an output).

Constant Variables:
C++ also allows "constant variables".

```
int N = 0;
```

This is an ordinary "variable variable" (in other words I can change its value by assignment statements).

```
const int N = 0;
```

This makes N = 0, but I cannot change it later; I cannot assign to it. This is a convenient way to gives names to quantities.

```
const int upb = V.size() - 1;
```

A lot of the time when we declare and initialize variables, it's just to give convenient names to things, and we never intend to change them. So they should be const variables (not too many C++ programmers do this, but it's clearer).