Exam I on Oct. 14.
Review Questions are available, with Key.
Review Session on Wed., 8 PM.

Modules and Clean Interfaces.

Large programs are divided into modules, which in C++ usually means functions, structs, and classes.

Modules are self-contained units or parts out of which something is made. Big modules usually have little modules inside them. We want to understand modules on their own, and we want to understand how they interconnect with each other. I.e., how they are arranged on the insides and the outsides.

General guidelines:

(1) A module should have a well-defined purpose. As a general rule you should be able to state it in a simple sentence.

(2) A module should have clear and well-defined interfaces. In other words, you should know what goes into it and where, and what comes out and where. Presumably what the module does depends on what goes into it, and if you don't know everything that goes into it, it's hard to know what it does. Also, if you don't know everything that comes out, you don't know its effects.

We want the modules to be easy and reliable to use. We ought to be able to look at the code and see what it does.

Pure vs. Impure Functions:

Pure functions are like mathematical functions (sin, cos, log). They have one or more input parameters, and they deliver an output. When I write y = sin x, x is the input and the value of sin x is the output (in effect, through the function name "sin"). The inputs and outputs are obvious. In C++:

```
double max (double x, double y) { … }
```

We can see the inputs (x, y) and the output (throughthe function name max). This is like a box with two inputs and one output. This is a pure function. Pure functions have an especially clear interface, and therefore are easy to understand.

What does an impure function look like? It has hidden (unobvious) inputs and/or outputs. This includes global variables and file I/O.

```
double x, y;
… many lines of code …

double Mystery (double a) {
  … many lines of code …
  y = x / 2;
  … many lines of code …
  return a*a / x;
}
```

This program has x as a hidden input, and we cannot understand easily what Mystery does, without knowing what x is. If x changes between calls of Mystery, it may give different results even if called with the same argument.

```
  cout << Mystery (10) << endl;
  x = x + 1;
  cout << Mystery (10) << endl; // we expect the same output, but we don't get it
```

Also, y is a hidden output, because it's not obvious that calling Mystery changes y.

```
  y = 1;
  cout << y << endl;
  cout << Mystery(10) << endl;
  cout << y << endl; // we expect the same output, but we don't get it
```

So impure functions have side-effects, which can hard to understand. It's not *impossible* to figure out, but it's *harder* to figure out.

That said, there are sometimes good reasons for using impure functions and side-effects. So be sensible.

Example, in debugging, we might put a cout<< in the function, which is technically speaking a side-effect. You might even want to do a cin>> in the function (e.g., to put in a correct value, where it's not computing it correctly).
Another example: you might want to count the number of times a function is called.

```
int abscount = 0;
```

```
double abs (double x) { // note: uses and changes abscount
  abscount++;
  if (x < 0) { return -x; }
  else { return x; }
}
```

Constant Reference Parameters:

We have seen that parameters passed by value are used as inputs,
and that parameters passed by reference can be used for output as well as input.

```
void F (int x) { … }   // x is an input
void F (int& x) { … } // x is an output (and possibly input)
```

But recall, reference parameter are more time-efficient than value parameters if the parameter is big. So many programmers pass inputs by reference for efficiency, not because they intend to use them as output. This is misleading.

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

The above passes the vector by value, which could be inefficient.

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

The above is more efficient, but looking at the first line, it suggests that V is an output, which it's not. So the better way to do it is to make it a "constant reference":

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

This says that the parameter is a constant, and cannot be assigned to (cannot be changed). So someone reading this declaration of printVector knows V is not an output parameter.

Constant "Variables":

```
// ordinary "variable variable":
int X = 1;
….
X = 2; // changes value of X

// a constant variable:
const int X = 1; // declares X a constant with value 1
….
X = 2; // illegal, can't change a constant
```

What is this good for? Often we use a variable just to give a meaningful name to a quantity, and we never intend to change it. E.g.,

```
const int length = V.size();
```

So if you are a declaring a variable just to give a name to something, it's better to declare it const, since then it's obvious to the reader that it will never be changed.

As much as I can tell the compiler what my assumptions are, it can check them, and make sure I'm not violating them.

Ways of Using Structures:

Consider the Time struc defined in your book:

```
struct Time {
  int hour, minute;
```

```
  double second;
};
```

There are two general styles for using a struct such as this:

(1) Use pure functions and treat Times as values.

(2) Use modifier functions and treat Times as objects

For example, to treat them as values, you might define an addTime function that takes two Times, adds them, and returns the Time that is their sum.

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

Time start = {2, 10, 0};
Time length = {0, 75, 0};
Time endOfClass = addTime (start, length);
```

In this case we think of Times more like abstract values. We operate on them to compute other abstract Time values.

To treat them as objects, you might have an addToTime function, that adds a Time to a given Time, modifying the latter.

```
void addToTime (const Time& t, Time& tDest) {
  const double seconds = convertToSeconds(t) + convertToSeconds(tDest);
  tDest = makeTime (seconds);
}

Time start = {2, 10, 0};
Time length = {0, 75, 0};
addToTime (length, start);
// now start is = length + start
```

In the second case, we think of "start" more like an object, which we can do things to (like add to its time).

Which is better? Sometimes things are naturally values (e.g., abstractions), sometimes they are naturally objects (e.g., physical things like scribblers or decks of cards), and sometimes it doesn't make much difference.