Exam II results: average = 30/60 (i.e., 50%), SD = 13.35.
Grades ranged from a high of 59/60 (congratulations!) to a low of 0.
Because the grades were exceptionally low, I will add 10% to everyone's grade (which will give several of you a grade over 100%). But you need to know this material for CS140 and later courses! I'm hear to explain and answer questions.

#5:

```
int x;
double gpa;
cin >> gpa >> x;
printf("[%4d,    %0.4f]\n", x, gpa);
```

(a) 2.4 271828
After reading, gpa == 2.4 and x == 271828.
%4d is not big enough for 271828, so it expands the field to accommodate it.
%0.4f puts 4 digits after the decimal. Since 0 is not big enough for anything, it expands the field enough to print 2.4 with four digits after the decimal:
[271828,   2.4000]

(b) 16.46312   -17
After reading, gpa == 16.46312 and x == -17. This time, %4d is big enough to hold -17, so it right-justifies -17 in a field of width 4. %0.4f uses a field wide enough to print 16.46312 with 4 digits after the decimal:
[-17,   16.4631]

(Formatting and printf is explained in LCR 7.)

Classes and Object-Oriented Programming (OOP):

In everyday language, a class is an open-ended collection or set of similar objects. For example, the objects Fido and Rover belong to the class dog. The idea is that all dogs have similar properties, for example, fur, four legs, bark, fetch things, etc. It's the same with objects and classes in programming: objects belonging to the same class have similar properties and behaviors (member variables and member functions).

C++ Aside:

In C++, classes and structs are almost identical. The only difference is that in structs by default all the members are public, whereas in classes by default all the members are private. In both cases you can explicitly declare which members you want to be public, and which you want to be private.

We have already defined structs (which are essentially classes) for Cards and Decks.

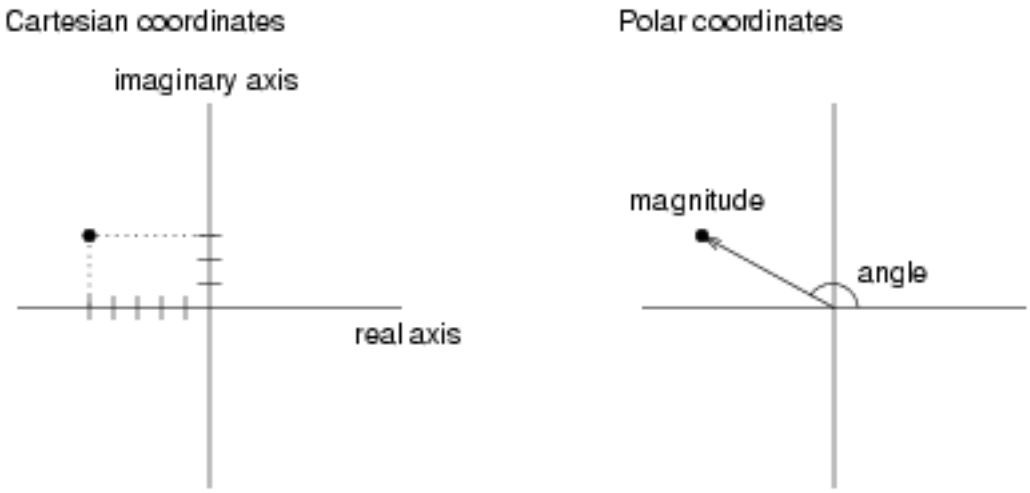
So all Cards have suit and rank instance (member) variables, but individual Card object may have different values for these variables.

Data Encapsulation:

Encapsulation is putting something inside some sort of controlled boundary. Functional encapsulation puts a bunch of instructions and local variables inside a function, but the interface to the function is just the parameters and the return value.

Data encapsulation supports the Information Hiding Principles. (You know what they are!)

The declaration of a class (or struct) defines the interface between the user and the implementer of a class. It creates a separation of concerns.



These are two ways of describing the same complex number.

It's simple to convert between them (just trig.).

Sometimes one form is more convenient, sometimes the other.

Addition and subtraction is more convenient in Cartesian form:

E.g., a = x + yi, and b = u + vi. Then:

a+b = (x + u) + (y + v)i.

a-b = (x - u) + (y - v)i.

"The real part of the sum is the sum of the real parts, and the imaginary part of the sum is the sum of the imaginary parts."

Multiplication and division is more convenient in polar form:

E.g., a = r e^{iu} and b = s e^{iv}.

(These are the mathematical expressions of complex numbers in polar form: r and s are the magnitudes of the numbers, u and v are their angles.)

Then:
$ab = rs\ e^{i(u+v)}$ and $a/b = (r/s)\ e^{i(u-v)}$ so it's very easy to multiply and divide complex numbers in polar form.

So which should you use?

If you are providing a class to implement complex numbers, you don't know if your users are going to be doing mostly addition and subtraction, or multiplication and division.

One solution is to use both representations, but then you have these problems:
(1) you have to keep the two representations consistent.
(2) You do not want to do everything twice, on both representations.

We use a kind of lazy evaluation. E.g., if we have the Cart. coords, we don't compute the polar until the user asks for them. And vice versa.