Exam I on Thursday

Errors in TCS chs. 10 and 11:

In ch. 10:
It says that over-subscripting a vector will give you a runtime error (10.1). This is not true. As we've discussed, it will just compute where the element would be and fetch or store something outside the vector, which might lead to a runtime error or more mysterious behavior. If you want subscript checking, use .at(). For example, count.at(i) rather than count[i].

Also, this chapter refers to a built-in random number generation called random(), but the standard version of C++ calls this rand().

In ch. 11: The text dealing with declaration and definition of member functions is confused and partly incorrect. This is now fixed in the .htm versions of TCs (but not yet in the pdf versions). This affects section 11.2.

Values and Objects

Generally speaking, we think of values as things we do operations on, such as numbers and strings.
We think of objects as things that have attributes and behaviors. Objects do things. They are active rather than passive (like values).
Some things have properties of both, but object-oriented programming focuses more on objects.

In programming terms the attributes and behaviors of an object are called member variables and member functions.

We have seen member variables, for example of Time objects:

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

How do we define member functions? We think of member functions (also called methods or behaviors) as things the object can do. So for example, one thing we might expect a Time object to do is to print itself.

We already have a function called printTime(t), which prints a time t. What we want to do is make print() a behavior of any Time object.

```
Time currentTime = { 9, 14, 30.0 };
Time now = { 14, 27, 0.0 };

currentTime.print();
now.print();
```

So we can think of print() as being part of the total behavioral repertoire of Time objects.

To do this we have to make print() part of the definition of Time.
We might also want to add other useful member functions.

Version 1:

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

  void print () {
  // print out my own hour, minute, second attributes with ":" between them.
  Time me = *this;
  cout << me.hour << ":" << me.minute << ":" << me.second << endl;
  }
};
```

We commonly qualify attributes by the objects that possess them. For example, in English, "Sally's age" corresponds in C++ to Sally.age. Likewise, to refer to "my age" I might say "me.age".

A different example, if I refer to "Sally's car" that's like Sally.car, or to "my car" that's like "me.car" or, more frequently I might just say "the car" (with the assumption it's mine). We can do a similar thing in C++:

```
  void print () {
  // print out my own hour, minute, second attributes with ":" between them.
  cout << hour << ":" << minute << ":" << second << endl;
  }
```

This is the more common (and convenient) way to do it in C++. An unqualified member variable refers to one's own member variable. This is called implicit access. ("this" is primarily used to pass all of the containing object to another function.)

As a first approximation, to change functions from standalone functions to member functions, we need to put them inside the curly braces of the structure.

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

```
  double second;

  // definitions of all the member functions of Time

};
```

Another example: make after() a member function of Times. E.g.,

```
Time t1, t2;
// initialize the times
if ( t1.after(t2) ) { … } //if t1 is after t2 do something or other
```
In effect I'm asking the t1 object to compare itself to t2.

Constructors for struct objects have a peculiarity: the name of the constructor is the same as the name of the struct.

All these functions (including constructors) obey the usual rules for overloading in C++: the compiler must be able to determine which definition to use from the types of the arguments.

It's important in OOP as we've seen to be specific about the interfaces to functions and other modules. So it's important to know what a object can do without necessarily knowing how it does it.