Exam II will be on Nov. 23.

Classes

Classes are the way we group similarly behaving objects in OOP. Just like classes of real objects, classes can be arranged in a hierarchy. For example, we might subdivide "animals" into "land animals," "birds," and "fish." Then we might divide land animals into "mammals," "reptiles," and "insects." And we might divide "mammals" into "dogs," "horses," etc.
We might divide "dogs" into "collies," "beagles," etc.
These are all classes and subclasses.
Belonging to these classes we also have individual objects. For example Fido might be a beagle, and Rover might be a collie.
This is essentially what we can do with classes and objects in OOP.

Suppose we were programming a video game. We might begin with a class:

displayObject: // anything that is going to be displayed
    position on screen
    change position
    show it
    hide it
    control size

movingObject (subclass of displayObject):
    set direction and speed
    move it from one place to another at some speed
    rotate

projectile (subclass of movingObject):

- collision detection
- explosion

animateObject (subclass of movingObject):

- limb motion
- die
- life meter

leggedObject (subclass of animateObject):

- control motion of legs (gait)

person (subclass of leggedObject);

- name
- avatar
- title
- ...

In general, in OOP:
A subclass inherits all of the properties (member variables and member functions) of its superclasses.
But there are exceptions. You can redefine a member function in a subclass to do something different. Or you can eliminate in a subclass a member that was defined in the superclass.

To redefine a function in a subclass, it has to be declared virtual in the superclass.

Let's look at the message class from TCS ch. 15.

Normally (you can do other things):
If a member is public, it is visible to everyone else, and it's public in the subclasses.
If a member is private, it is visible only in its "home class" and nowhere else.
If a member is protected, it is visible in subclasses of its home class, but nowhere else. In effect it's private to the "family" of its home class.

To create a message:

```
Message RM;
RM = Message ("Memory Manager", "reorganizing memory");
…
cout << RM.getMessage() << endl;
```

This will print:

    Memory Manager: reorganizing memory

We suppose that error messages (a subclass of messages) also have an error number that you can look up in a book (or online).

```
ErrorMessage OOM;
OOM = ErrorMessage ("999", "Memory Manager", "out of memory!");
….
cout << OOM.getMessage() << endl;
```

Prints:

    ERROR 999: Memory Manager: out of memory!

After I have overloaded << to work with Messages, I can write:

```
cout << OOM << endl; // will print the same stuff
```

Templates

In general, computers are good at doing the same thing over and over again; people are not so good at it (we get bored, tired, distracted, etc. and make mistakes). This is one reason we have computers: to automate mechanical, regular tasks.

Any time you find yourself doing the same thing over and over again, you should ask if there is some way you can get the computer to do it.

Abstraction Principle: Avoid doing the same thing repeatedly. Factor out the recurring pattern and give it a name.

This is the basis for functional abstraction (defining a function to do the same thing, but possibly with different parameters).

Something you learned early in this class was how to swap the contents of two variables. For example, to swap the contents of integers M and N:

```
int T;
T = M;
M = N; // think about it!
N = T;
```

The problem is, it is easy to make a mistake, especially if you are rushing. One solution: figure it out once and define a function.

```
void swapvars (int& X, int& Y) {
    int T = X;
    X = Y;
    Y = T;
}
```

Then I can write swapvars(M,N) and be sure it's correct.
But what if I want to swap two doubles?

```
double A, B;
swapvars(A,B);    // won't work unless I define a new swapvar for doubles
                  // effectively overloading swapvars.
```

Suppose I want to swap two Cards? Or two Messages? They would all have a common pattern:

```
void swapvars (SOMETYPE& X, SOMETYPE& Y) {
    SOMETYPE T = X;
    X = Y;
    Y = T;
}
```

I could even give you this, and tell you to copy it into your program and use a text editor to replace SOMETYPE by whatever type you want. This is what templates do for you.

```
// A templated version of swapvars:

template <class SOMETYPE>
```

```
void swapvars (SOMETYPE& X, SOMETYPE& Y) {
    SOMETYPE T = X;
    X = Y;
    Y = T;
}
```

In effect I have defined a generic version of swapvars that works for any type (class) SOMETYPE.