TOWARDS OBJECT ORIENTATION

Today

- Today we will at how we construct classes from other classes:
  - Composition
  - Derivation
- These are important ideas in object-oriented programming.
- We will also briefly cover *polymorphism* and *overloading*, and mention friend classes.
- This material is taken from Pohl, Chapter 5, mainly 5.7 and 5.10.

Composition and derivation

- Compsition and derivation are the cornerstones of onject-oriented programming.
- Composition:
  - Creating objects with other objects as members
- Derivation:
  - Defining classes by expanding other classes
- We can see examples of both of these ideas in the program `coloredTriangle.cpp` which you can get from the course website.

- Here's the class `triangle` that we met in a lab:

```
class triangle {

private:
  point a, b, c;

public:
  triangle(){};
  triangle(point, point, point);

  void print() const;
  void set( point, point, point);
  point getA();
  point getB();
  point getC();
};
```

- Here is the class `coloredTriangle` which adds an extra attribute to `triangle`:

```
class coloredTriangle: public triangle {

private:
  string color;

public:
  coloredTriangle(point, point, point);

};
```

- Like “extends” in java.

- "Base class" (`triangle`)
- "Derived class" (`coloredTriangle`)
- Derived class contains all the attributes of the base class.
- However, derived class can only access *public* members of base class

  See `coloredTriangle.cpp` for an example.
- The word `public` in the declaration:

  ```
  class coloredTriangle: public triangle {
  ```

  controls what is accessible in the base class. We will look more at this later in the semester.

Here is another example

- Taken from the program `robots.cpp` which you can get from the course website.

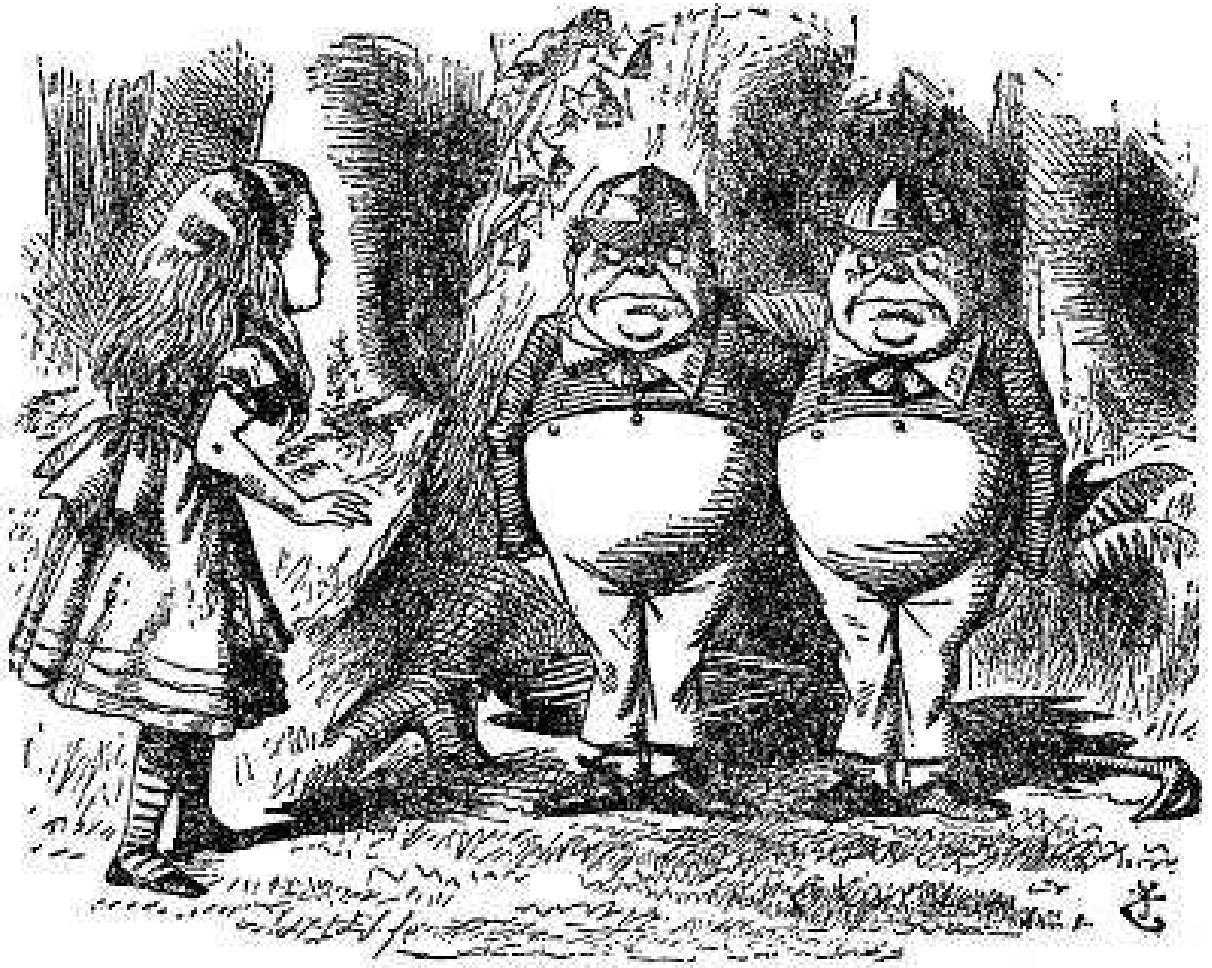
```
class roomba: public robot {
private:
  string type;

public:
  void setType(string s);
  void vacuum(double x, double y);
};
```

Polymorphism and overloading

- *polymorphism*—giving different meanings to the same function or operator, i.e., "having many forms". Lets us use different implementations of a single class
- *overloading*—creating new versions of functions with the same or different arguments
- When you overload a function, the name of the function is the same, but what is does is different from the default
- Operators can also be overloaded
- *signature matching* is what the compiler uses when there are multiple versions of a function (or operator) to determine which version should be invoked
- Textbook goes into a LOT of detail about this—we'll come back to it more later in the semester.

Friend classes

- Allows two or more classes to share private members and functions
  - e.g., container and iterator classes
- Friendship is not transitive.
- Since friendship violates the usual rules about hiding members, you need to use it with care.
- In fact you should try *not* to use `friend`.
  - When writing code from scratch you should be able to avoid it.
  - It tends to be used when quickly patching code.



```
class tweedledee {
...
  friend class tweedledum;

  int cheshire();
...
};
```

- This allows any instance of `tweedledum` to access any member of any instance of `tweedledee`.
- However no instance of `tweedledee` can access any private member of `tweedledum`.

Friend functions

- Friendship can also be at the individual function level.
- A non-member friend function can have access to the private components in a class.
- Extending the previous example:

```
void alice() {
...
}

class tweedledum {
...
  friend void alice()   // prototypes for friend functions
  friend int tweedledee::cheshire ();
...
};
```

- This allows `alice` and `cheshire` to access the data in instances of `tweedledum`.
- For concrete example see the program `robots.cpp`
- If this example seems contrived, that's because it is :-)
- `friend` is like that — unless you really need it, it seems rather superfluous.

Derivation and friendship

- Friendship is not the same as derivation!
- Example:
  - $b2$ is a friend of $b1$
  - $d1$ is derived from $b1$
  - $d2$ is derived from $b2$
- In this case:
  - $b2$ has special access to private members of $b1$, as a friend
  - But $d2$ does not inherit this special access
  - Nor does $b2$ get special access to $d1$ (derived from friend $b1$)
- `arrays.cpp` gives a more interesting example than `robots.cpp`, but you need to be comfortable with pointers.
- We'll talk about derivation more later in the course.

Summary

- This lecture has looked at:
  - Composition and derivation
  - Polymorphism, overloading
  - Friends
- For most of these topics, it has been a first look; we will come back to them over and over again through the semester.