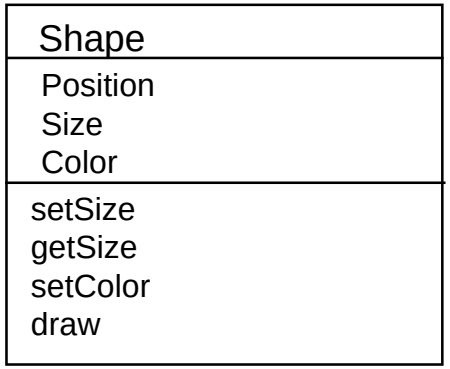
Translating UML Classes to C++

UML -> C++ Code




```
class Shape
{
public:
    Shape(Position p, Size s, Color c);     // nominal constructor
    Shape( );                               // default constructor
    Shape(Shape & orig);                    // copy constructor
    ~Shape( );                              // destructor
    void setPosition(Position p);           // setters
    void setSize(Size s);
    void setColor(Color c);
    Size getSize( );                        // getters
    void draw(Graphic g);                   // other actions
    Shape& operator=(Shape& original);      // assignment

private:
    Position position;
    Size size;
    Color color;
    ...
};
```

Guideline on Constructors vs. Setters

- If information is ***essential*** to the ***meaning*** of an object, it is better to pass it through the **constructor** than using a setter.
- Rationale: Using a setter, the object either
  - has **no meaning** while awaiting for setting to occur, or
  - the essential information has to have **default values** so that the object has meaning.
- Exception: Extremely large number of variables need to be passed, which would be confusing using the **positional notation** of a constructor.
- Exception: Two-way navigable associations; one object must be created before the other.

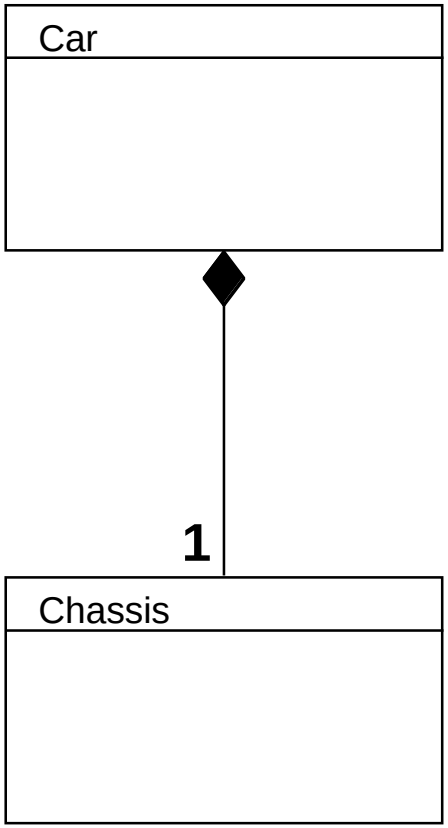
Guideline on Constructors vs. Setters

```
Shape s(p, s, c);        // use nominal constructor
```

preferred over:

```
Shape s;                 // use default constructor
 . . .
s.setPosition(p);
s.setSize(s);
s.setColor(c);
```

UML -> C++



See /cs/cs121/code/initializer Car.hh, Car.cc for details.

```
class Car
{
   . . .
private:

/** the make of this car **/

string make;

/** the Chassis of this car **/

Chassis chassis;

}; // class Car
```

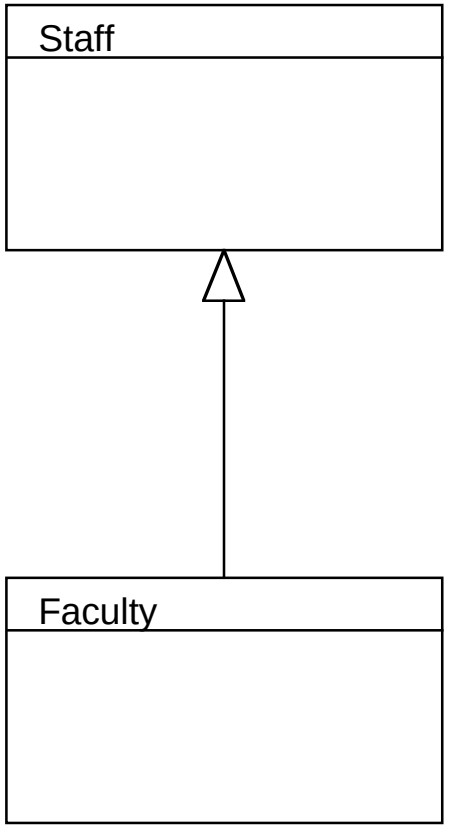
```
/** Create a car of a specific make. **/

Car::Car(string _make)
      : make(_make),
        chassis(getChassisMake(_make))
{}
```

***initializer* notation**

empty body

calls *Chassis* constructor

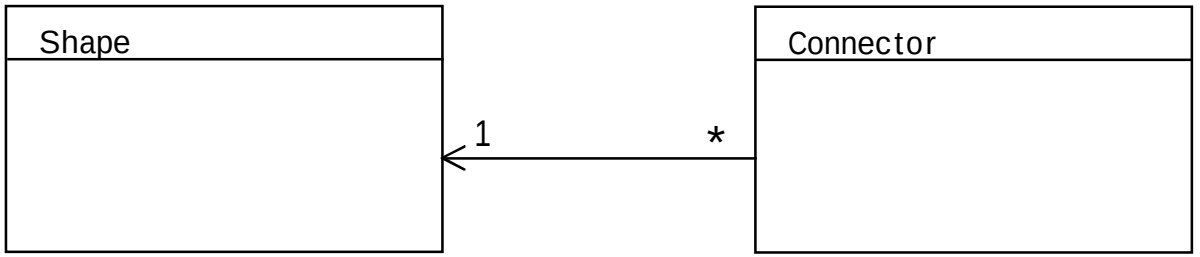
Inheritance/Generalization




```
class Staff
{
public:

Staff(string name);            // constructor

    ...

};


class Faculty : public Staff
{
public:

Faculty(string name);          // constructor
};


Faculty::Faculty(string name)
          : Staff(name)
  {
    ...
  }
```

***initializer* notation,**
calls constructor for *superclass*

UML -> C++:
One-way navigability



This UML says that one shape may have many connectors.

One can get from a Connector to its Shape, but not vice-versa.

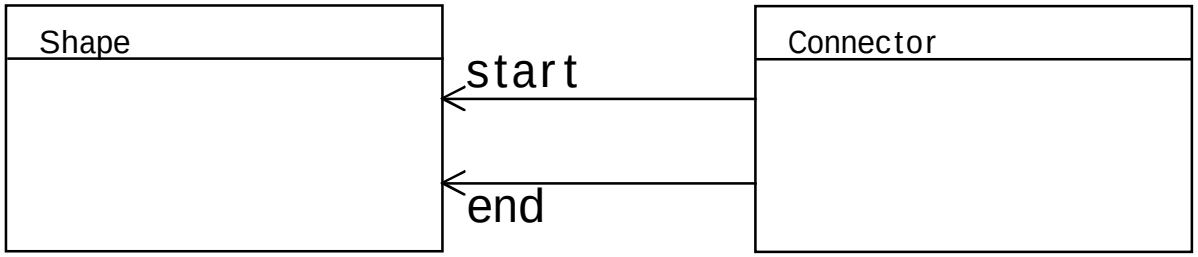
```
class Shape
{
public:
    ...
private:
    ...
};
```

```
class Connector
{
public:
    ...

private:
    ...
    Shape *shape;
};
```

Disclaimer

- The C++ code examples are samples of what ***can*** be done.
- They are generally not the ***only*** way a specific type of association can be implemented.
- A specific ***tool*** (Rose, Rhapsody, etc.) may generate a specific type of implementation; selection from a a menu of implementations might be possible.
- Use of the/a standard library, is possible and often advised for portability.

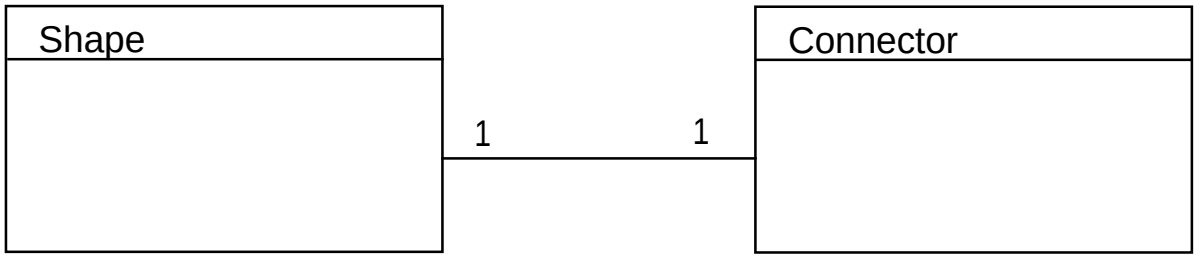
UML -> C++:
Multiple associations with different roles




```
class Shape
{
public:
    ...
private:
    ...
};
```

```
class Connector
{
public:
    ...
private:
    ...
    Shape *start;
    Shape *end;
};
```

UML -> C++:
1-1 association




```
class Shape
{
public:
    ...
    setConnector(Connector *connector);

private:
    ...
    Connector *connector;
};
```

```
class Connector
{
public:
    ...
setShape(Shape *shape);
private:
    ...
    Shape *shape;
};
```

Generally choose one, not both; use constructor for the other.
Each calling the other could be a problem.