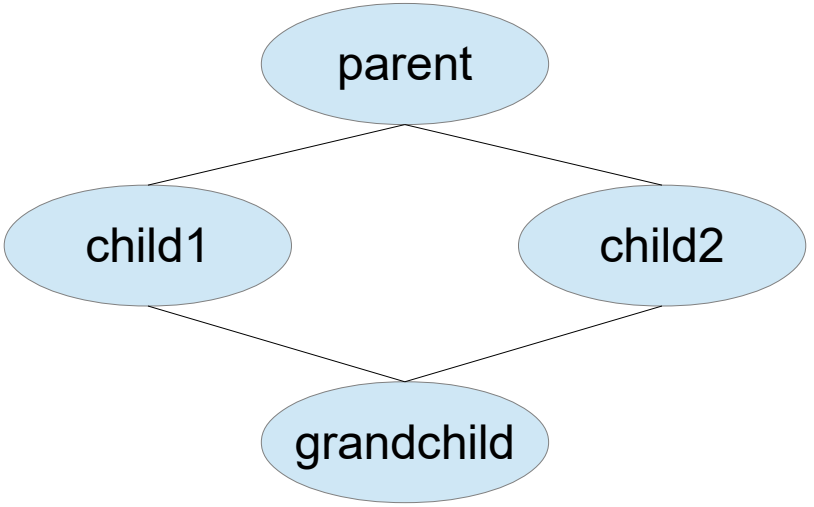
Multiple inheritance

- a class can be derived from multiple other classes
- it inherits all the fields and methods for each
- need to resolve name clashes
- need to address the “diamond problem”



by default, grandchild inherits everthing from parent twice:
once through child1,
once through child2

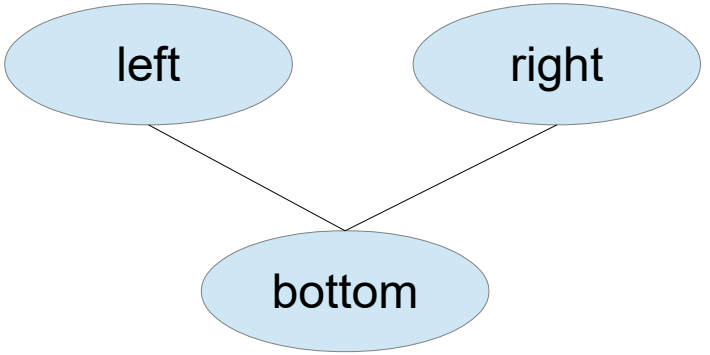
Declaration syntax

- specify a comma separated list of base classes, giving the inheritance mode for each (public, protected, private)

```
class left {
   ...
}

class right {
   ...
}

class bottom: public left, public right {
}
```



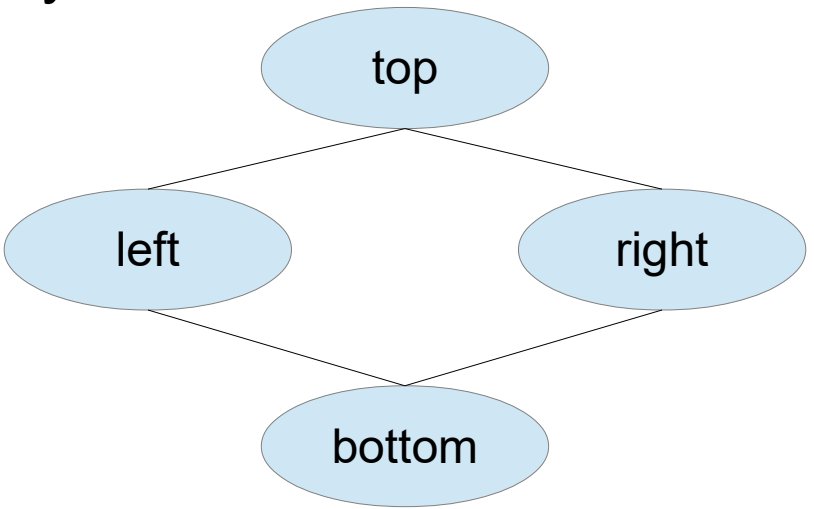

```
// constructor order: left, right, bottom (based on derivation order in  bottom's declaration)
// destructor order: bottom, right, left
```

Inherited fields/methods

- if all names are unique, the derived class can simply refer to the inherited fields/methods by name
- in the case of name clashes (ancestor classes have fields/methods of the same name):
  - access inherited method by classname::methodname syntax
- if an inherited method has a different parameter list than in the derived class (e.g. print() vs print(int x)) the inherited one is said to be *hidden*
  - is only assessible using classname::methodname

The diamond problem

- bottom gets all top's fields and methods twice, as left::___ and right::___
- thus two versions of each data field, possibly with different data over time
- this is usually not the behaviour we want ...



suppose top class has a field X,

then bottom gets a left::X and also a right::X

Constructors and diamond

- default constructors used unless otherwise specified
- bottom can't specify top constructor, needs to be done by left/right
- shown below with initializer lists

```
class top {
   protected:
      int xVal;
   public:
      top(int x): xVal(x) { }
};

class left: public top {
   protected:
       float yVal;
   public:
       left(int i, float y): top(i), yVal(y) { }
};
```

```
class right: public top {
    protected:
        string wVal;
    public:
        left(int i, string w): top(i), wVal(w) { }
};

class bottom: public left, public right {
   protected:
        char zVal;
   public:
      bottom(char z, int j, float k, string s):
            left(j, k), right(a, b), zVal(z) { }
};
```

Virtual base classes

- can derive virtually, essentially telling the compiler that we wish to share any inherited ancestors
- means there will only be a single top inherited by bottom

```
class left: virtual public top {
...
};

class right: virtual public top {
...
};

class bottom: virtual public left, virtual public right {
...
};
```

now bottom can refer to top's constructors, fields, and methods without ambiguity