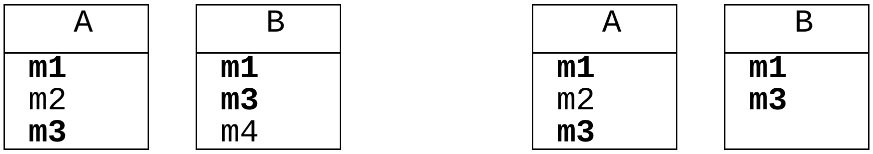
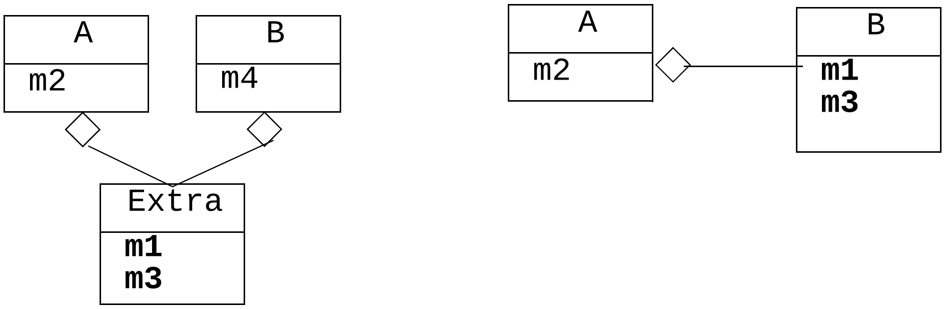
4 INHERITANCE

- ❍ Class has attributes, methods (and relations)
- ❍ If class A and B have
  - ❍ all different members
    - ● the classes are different
  - ❍ all the same members (including exact behavior)
    - ● they are one the same class
    - ● note: members can have the same names/interfaces yet be different if behavior differs
  - ❍ some members the same (the choice is design decision not a amust)
    - ● left different
    - ● aggregation/composition
    - ● generalization/specialization
    - ● different case if one class includes all members of the other (if non-inclusive then a third class is needed)

Example 4.1 Some mebers are shared in non-inclusive and all-inclusive fasion. Potential for generalization/specialization or aggregation/compistion.



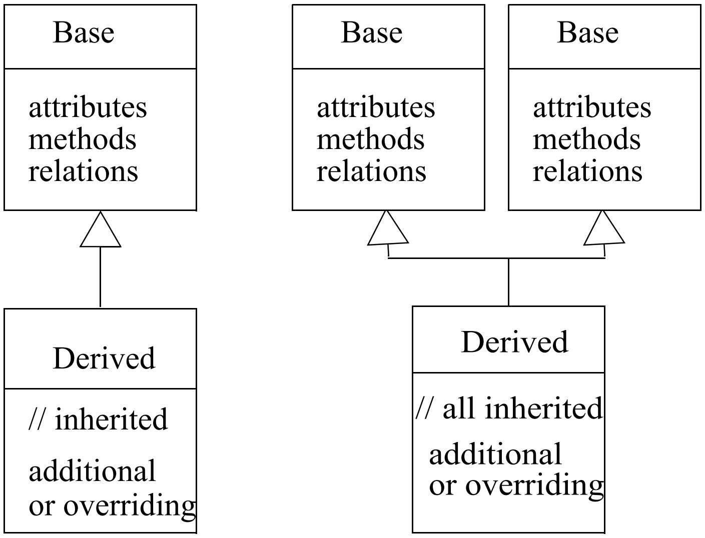
Example 4.2 Approach with aggregation/compostion.



4.1 Generalization/Specialization with Inheritance

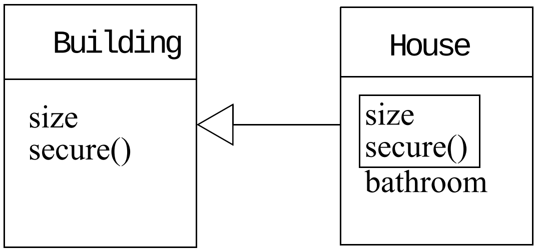
- One class is *Base* (*Superclass*), another is *Derived* (*Subclass*)
- The derived class *inherits* attributes, associations, and methods from base(s)
  - *single inheritance* when one base

Example 4.3 Single and multiple inheritance.

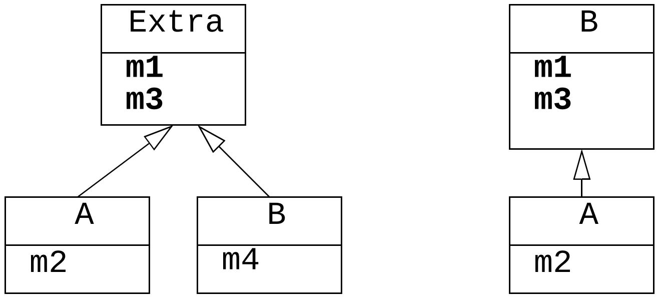


- Derived class will have all data and methods of the base
  - it will be more specialized
    - must have something different

Example 4.4 Inheritance. Derived `House` has everything base `Building` has.



Example 4.5 Approach with generalization/pecialization. Derived will also inherit from base so will also have `m1/m3`..



- ❍ Basic principles
  - ❍ no data/methods can be removed
  - ❍ data/methods can be hidden (`private`/`protected`)
  - ❍ methods can be overridden
  - ❍ no overloading between base and derived even if signature different
    - ● derived method of the same name always overrides base's
    - ● base's can still be called using `Base::method` notation
- ❍ Inheritance chain may continue
  - ❍ but objects *bloat*
- ❍ Inheritance kinds
  - ❍ `default is private`
  - ❍ `public`
    - ● `Base::private members` are not directly accessible in derived
    - ● `Base::protected members` are directly accessible in derived and remain protected
    - ● `Base::public members` are directly accessible in derived and remain public
  - ❍ `private/protected` inheritance
    - ● all members of base become at least `protected`/`private` in derived
    - ● used to inherit data members but remove public interfaces (to redo them)

```
class Derived : public/protected/private Base {
   // derived class definition
};
```

Example 4.6 Example inheritance: `Point3D` is a `Point` plus it is has a 3rd dimension.

```
class Point {
private:
   int x;
   int y;
public:
   Point (int xv=0, int yv=0);
   void setX(int);
   void setY(int);
   void show(void) const;
};

class Point3D : public Point {
private:
   int z;
   //...
};
```

4.2 Default Access and Adjusting Access

- Using `public` inheritance
  - `private` from Base
    - in derived: not directly accessible
    - in applications: not directly accessible
  - `public` from Base
    - in derived: directly accessible
    - in applications: directly accessible
  - `protected` from Base
    - in derived: directly accessible (and passed down the inheritance link)
      - makes it simpler to manipulate in derived classes
    - in applications: not directly accessible

Example 4.7 In Example 4.6, Point3D cannot access x and y directly.

```
Point3D::Point3D(void) {
   x=0;     // error
   setX(0); // ok, public
   // etc
}
```

- Access can be selectively restricted or relaxed but only on accessible members
  - using using declaration
  - but must have access to to start with
- Access can be restricted to all, using protected/public inheritance

Example 4.8 Here Point3D makes all public members of Point private.

```
class Point {
   private:
      int x;
      int y;
   public:
      void setX(int);
      // etc
   };

   class Point3D : private Point {
      //...
   };
// now somewhere
   Point3D p3;
   p3.setX(3); // error
```

Example 4.9 Here Point3D makes only Point::setX private.

```
class Point {
   private:
      int x;
      int y;
   public:
      void setX(int);
      void setY(int);
      // etc
   };

   class Point3D : public Point {
      private: using Point::setX;
      //...
   };
// now somewhere
   Point3D p3;
   p3.setY(3);
   p3.setX(2); // error
```

Exercise 4.1 We have movies in `Film` class. Then we have `DirectorCut` and `ForeignFilm`, which are also `Film`s, thus we have inheritance, with `Film` the base. When implementing `output` method of both derived, we can first call upon `output` of the base, to display the inherited part, then handle the rest.
Important: note the multiple inclusion mechanism used in base (`Film.h`).

4.3 Initialization in Basic Inheritance

- Basic rules
  - contructor of *Base* triggers before contructor of *Derived*
    - if no colon initialization give, the default constructor or error if none
    - otherwise colon initialization triggers
    - destructors work in reverse

Example 4.10 `Point3D` is a specialization of `Point`. An instance of `Point3D` would have 3 integers (`x,y,z`). But only `z` can be directly accessed (`x, y` are private).

```
class Point {
private:
   int x;
   int y;
public:
   Point (int xv=0, int yv=0);
   void setX(int);
   void setY(int);
   int X(void) const;
   int Y(void) const;
};

class Point3D : public Point {
private:
   int z;// Point3D has x, y, and z!!!
public:
   Point3D(int xv=0, int yv=0, int zv=0);
   void setZ(int);
   int Z(void) const;
};

// in some function
   Point3D p;
   p.setZ(1);     // ok
   p.setX(20);    // ok
```

- ❍ How to construct and initialize Point3D?
  - ❍ must initilize Point3D attributes
  - ❍ must thus initialize base attributes
    - ● by direct/method access

Example 4.11 Initialization by direct/method access

```
Point3D::Point3D(int xv=0, int yv=0, int zv=0) {
   z=zv;
   setX(xv); // x is private so cannot do x=xv
   setY(yv); // setX() and setY() are inherited by Point3D from base
}
```

- ● by constructors. This is better and must be used to initialize
  - ● non-static const data members
  - ● reference data members

```
DerivedClass::DerivedClass(arguments)
      : BaseConstructors(arguments) {
   // constructor body
}
```

Example 4.12 Initialization by constructors.

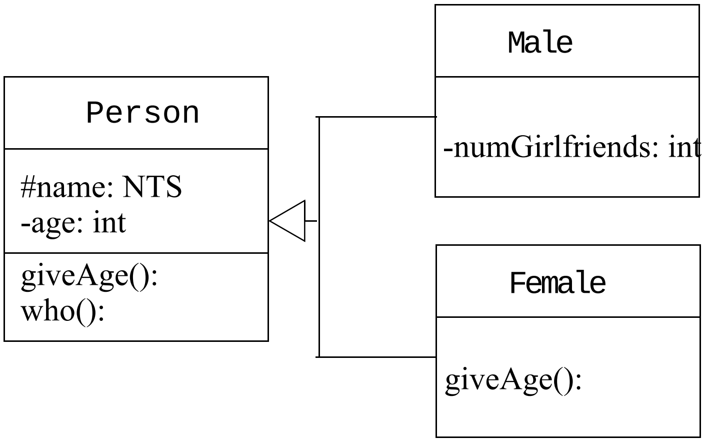
```
Point3D::Point3D(int xv=0, int yv=0, int zv=0) :
      Point(xv,yv){   // constructor to initialize x and y
   z=zv; // or setZ(zv);
}
```

Example 4.13 Another initialization by constructors.

```
Point3D::Point3D(int xv=0, int yv=0, int zv=0) :
      Point(xv,yv),// constructor to initialize x and y
      setZ(zv) {// or z(zv)
   // nothing to do now
}
```

Example 4.14 Person may be abstract - every person must be either male or female and thus only Male/Female objects will ever be created. This is discussed later.

Exercise 4.2 Person has name and age. Person is specialized into Male and Female. Female overrides giveAge(). Male has extra data. name is protected and thus can be accessed in derived classes.



Exercise 4.3 Sequence is specialized into Sorted Sequence. Not a complete program.

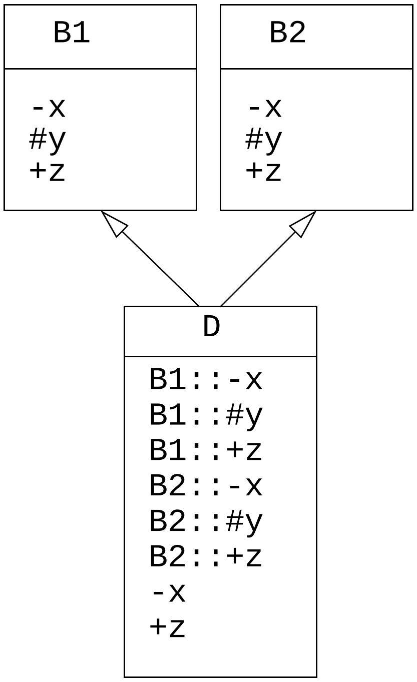
4.4 Multiple Inheritance

```
class Derived: public/private/protected Base1 [,...] {
   /inherits from all listed bases
};
```

- ❍ Single inheritance
  - one base, one or more derived
  - derived class is specialization
- ❍ Multiple inheritance
  - at least two bases
  - one or more derived
  - derived has a union of properties of the bases and thus not specialization
  - potantial problems with name conflicts

Example 4.15 We saw it before that iostream inherits from istream and ostream.

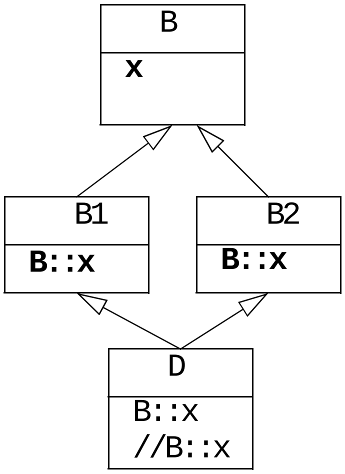
Example 4.16 Example with potential name conflicts. With multiple bases, the possible conflicts multiply. Which are valid in constructor to D?



```
D::D() {
B1::x=0;
B2::x=0;
x=0;

B1::y=0;
B2::y=0;
y=0;

B1::z=0;
B2::z=0;
z=0;
}

// in application, can do
D d;
d.z=1;
d.B1::z=1;
d.B2::z=1;
```

- Conflicts are sure if both parents inherit from the same base
  - called *diamond* inheritance
  - `virtual` inheritance prevents this

Example 4.17 Diamond inheritance.



```
class B1: virtual public B {
//
};
class B2: virtual public B {
//
};

class D: public B1, public B2 {
// now only one B::x
};
```