LECTURE 11

Multiple Inheritance

In the class hierarchies discussed so far, each class has inherited from a single parent. However, it is possible for a class to inherit from more than one base class. This is called multiple inheritance. An example is the class sleeper sofa, which is both a bed and a sofa:

```
class Bed
  {
    public:
Bed();                         //constructor
      void sleep();
      int weight;
  };
class Sofa
  {
    public:
      Sofa();                         //constructor
      void sit();
      int weight;
  };
class SleeperSofa:  public Bed, public Sofa
  {
    public:
      SleeperSofa();
      void foldOut();
  };
int main()
  {
    SleeperSofa ss;
      ss.sit();                    //Sofa::sit()
      ss.foldOut();                //SleeperSofa::foldOut()
      ss.sleep();                  //Bed::sleep()
    return 0;
  }
```

Notice the class `SleeperSofa` inherits from both `Bed` and `Sofa`. `SleeperSofa` inherits all members of both base classes. Thus `ss.sit()` and `ss.sleep();` are legal. You can use a `SleeperSofa` as a `Bed` or as a `Sofa`. Plus the class `SleeperSofa` can have members of its own such as `foldOut()`.

Inheritance Ambiguities

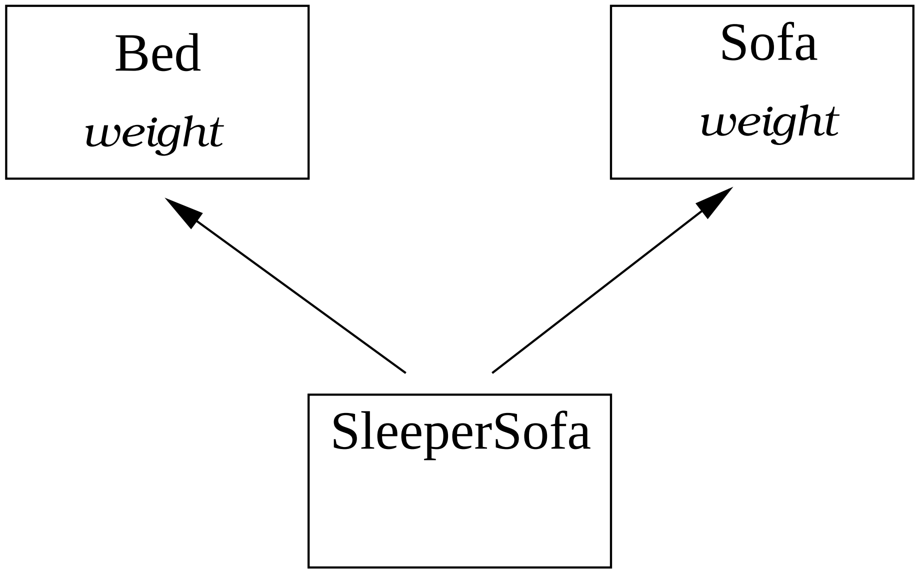
Although multiple inheritance is a powerful feature, it introduces some ambiguities. Consider the preceding example where both `Bed` and `Sofa` have a data member `weight`.

Which `weight` does `SleeperSofa` inherit? The answer is both. `SleeperSofa` has a member `Bed::weight` and another member `Sofa::weight`. Because they have the same name, unqualified references to `weight` are now ambiguous as in the following example:

```
#include <iostream.h>

void fn()
{
   SleeperSofa ss;
   cout << "weight = "
        << ss.weight     //illegal: which weight?
        << "\n";
}
```

The inheritance structure is shown in the following figure:



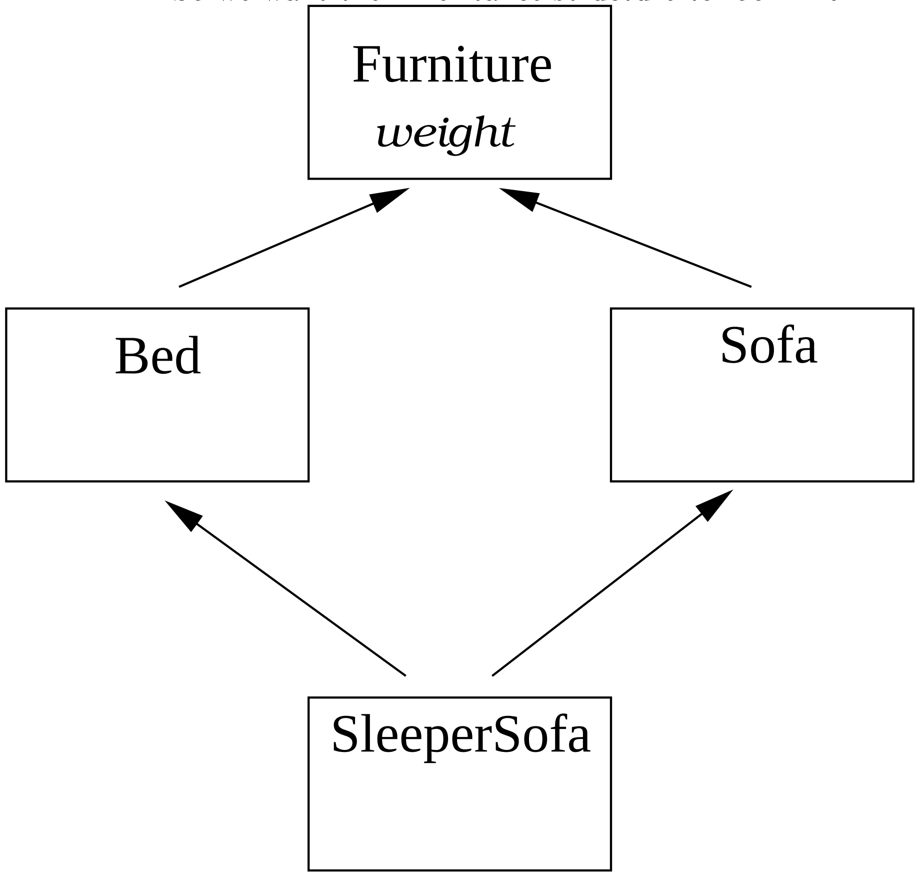
We could fix this by explicitly specifying the desired base class:

```
#include <iostream.h>

void fn()
{
   SleeperSofa ss;
   cout << "weight = "
        << ss.Sofa::weight  //ok but ugly
        << "\n";
}
```

This is ok but undesirable because it forces class information to leak outside the class into the application code.

So how do we fix name collisions? One way might be to make a more fundamental base class `Furniture` that has `weight` as a member. Then derive `Bed` and `Sofa` from `Furniture`. So we want the inheritance structure to look like:



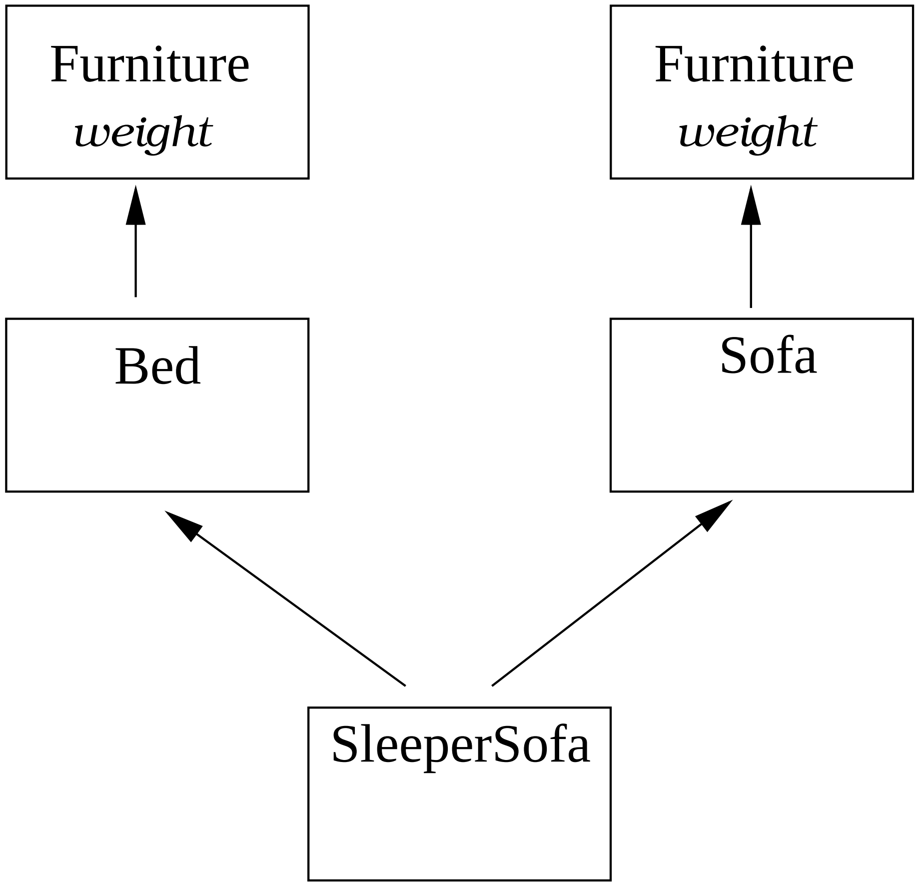

```
#include <iostream.h>

//Furniture - more fundamental concept; this class
//            has "weight" as a property
class Furniture
{
  public:
   Furniture()
   { }
   int weight;
};
class Bed : public Furniture
{
  public:
   Bed()
```

```
   { }
   sleep()
   { }
};
class Sofa : public Furniture
{
  public:
   Sofa()
   { }
   void watchTV()
   { }
};
class SleeperSofa : public Bed, public Sofa
{
  public:
   SleeperSofa()
   { }
   void foldOut()
   { }
};
void fn()
{
   SleeperSofa ss;
   cout << "weight = "
        << ss.weight     //problem solved; right?
        << "\n";
}

int main()
{
   fn();
   return 0;
}
```

This doesn't work because now `SleeperSofa` inherits 2 `Furniture` objects; one through `Bed` and one through `Sofa`. So when `weight` is called, it doesn't know which one to use. The inheritance structure actually looks like:



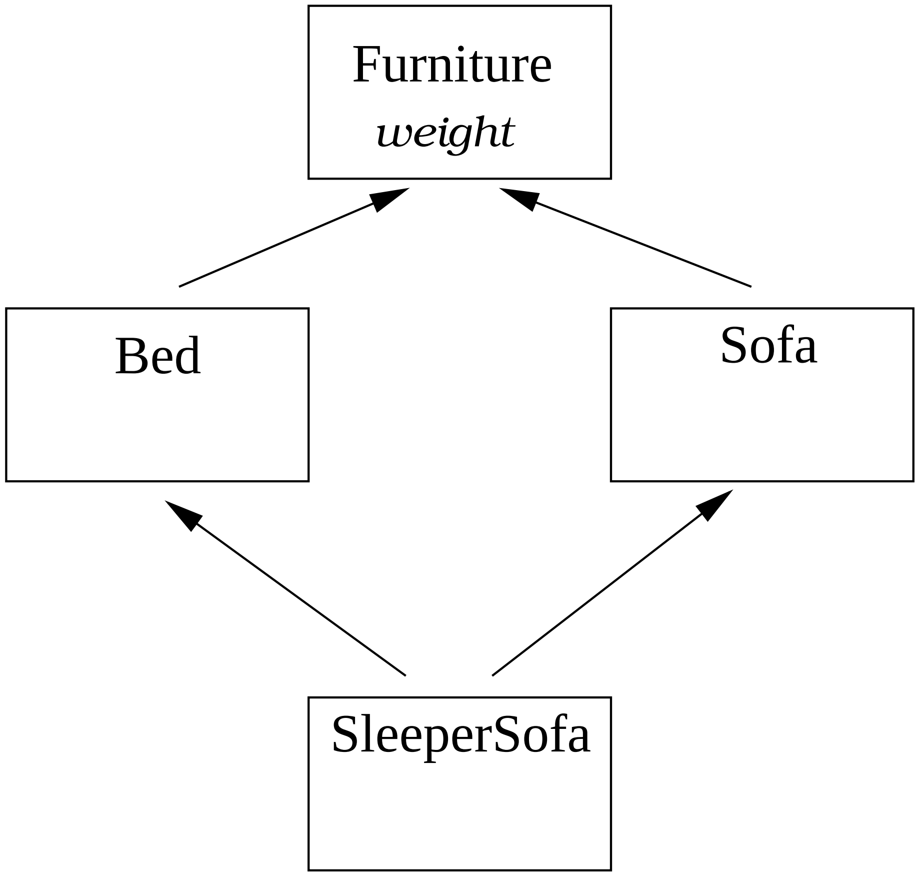
Obviously `SleeperSofa` only needs one `Furniture` object. To accomplish this, we use *virtual inheritance.*

```
#include <iostream.h>
class Furniture
{
  public:
   Furniture()
   { }
   int weight;
};
class Bed : virtual public Furniture
{
  public:
   Bed()
   { }
   void sleep()
   { }
};
class Sofa : virtual public Furniture
```

```
{
  public:
   Sofa()
   { }
   void watchTV()
   { }
};
class SleeperSofa : public Bed, public Sofa
{
  public:
   SleeperSofa() : Sofa(), Bed()
   { }
   void foldOut()
   { }
};
void fn()
{
   SleeperSofa ss;
   cout << "weight = "
        << ss.weight
        << "\n";
}

int main()
{
   fn();
   return 0;
}
```

Notice the addition of the keyword *virtual* in the inheritance of `Furniture` in `Bed` and `Sofa`. This says, "Give me a copy of `Furniture` unless you already have one somehow, in which case I'll just use that one." So `SleeperSofa` ends up inheriting only one `Furniture` object. Now the reference to `weight` in `fn` is no longer ambiguous and we have solved the problem of name collisions. Now the inheritance structure is the desired one and looks like:



If virtual inheritance solves this problem so nicely, why isn't it the norm or default? There are 2 reasons. First virtually inherited base classes are handled internally quite differently than normally inherited base classes, and these differences involve extra overhead. Second, you might want 2 copies of the base class, though this is unusual. As an example of the latter, consider a `TeachingAssistant` who is both a `Student` and a `Teacher`, both of which are subclasses of `Academician`. If the university gives its TA's 2 ID's–a student ID and a separate teacher ID–class `TeachingAssistant` will need 2 copies of class `Academician`.

Order of Construction

The rules for constructing objects need to be expanded to handle multiple inheritance. The constructors are invoked in the following order:

1. The constructors for any virtual base classes are called in the order in which the classes are inherited.

2. Then the constructors for any nonvirtual base classes are called in the order in which the classes are inherited.

3. Next the constructor for any member objects are called in the order in which the member objects appear in the class.

4. Finally, the constructor for the class itself is called

Notice that base classes are constructed in the order in which they are inherited and not in the order in which they appear on the constructor line.

Cautionary Note

I bring up the topic of virtual inheritance not because I want you to go out and use it right away, but because you may want to read someone else's code which uses it. Here are some reasons to be cautious about using multiple inheritance:

1. Multiple inheritance introduces more overhead and therefore is more inefficient.

2. It can introduce ambiguities like the name collisions we saw earlier. Another ambiguity arises when casting a pointer from a subclass to a base class; this often changes the value of the pointer in strange and mysterious ways. For example:

   ```
   #include <iostream.h>
   class Base1 {int mem};
   class Base2 {int mem};
   class SubClass: public Base1, public Base2 {};

   void fn(SubClass *pSC)
   {
     Base1 *pB1 = (Base1*)pSC;
     Base2 *pB2 = (Base2*)pSC;
   }
   ```

   `pB1` and `pB2` are not numerically equal even though they came from the same original value `pSC`. (Actually, if `fn` is passed a zero, they are equal. See how strange it gets?)