SESSION

24

Multiple Inheritance

Session Checklist

✔ Introducing multiple inheritance

✔ Avoiding ambiguities with multiple inheritance

✔ Avoiding ambiguities with virtual inheritance

✔ Reviewing the ordering rules for multiple constructors

30 Min. To Go

In the class hierarchies discussed so far, each class inherited from a single parent. This is the way things usually are in the real world. A microwave oven is a type of oven. One might argue that a microwave has things in common with a radar which also uses microwaves, but that's a real stretch.

Some classes, however, do represent the blending of two classes into one. An example of such a class is the sleeper sofa. As the name implies, it is a sofa and also a bed (although not a very comfortable bed). Thus, the sleeper sofa should be allowed to inherit bed-like properties as well as couch properties. To address this situation, C++ enables a derived class to inherit from more than one base class. This is called multiple inheritance.

How Does Multiple Inheritance Work?

Let's expand the sleeper sofa example to examine the principles of multiple inheritance. Figure 24-1 shows the inheritance graph for class `SleeperSofa`. Notice how this class inherits from class `Sofa` and from class `Bed`. In this way, it inherits the properties of both.

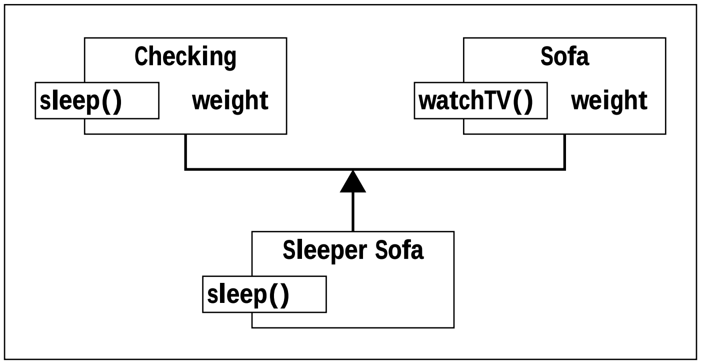

Figure 24-1
Class hierarchy of a sleeper sofa

The code to implement class `SleeperSofa` looks like the following:

```
// SleeperSofa - demonstrate how a sleeper sofa might work
#include <stdio.h>
#include <iostream.h>

class Bed
{
  public:
    Bed()
    {
        cout << "Building the bed part\n";
    }
    void sleep()
    {
        cout << "Trying to get some sleep over here!\n";
    }
    int weight;
};
```

```
class Sofa
{
  public:
    Sofa()
    {
        cout << “Building the sofa part\n”;
    }
    void watchTV()
    {
        cout << “Watching TV\n”;
    }
    int weight;
};

//SleeperSofa - is both a Bed and a Sofa
class SleeperSofa : public Bed, public Sofa
{
  public:
    // the constructor doesn’t need to do anything
    SleeperSofa()
    {
        cout << “Putting the two together\n”;
    }
    void foldOut()
    {
        cout << “Folding the bed out\n”;
    }
};

int main()
{
   SleeperSofa ss;
   //you can watch TV on a sleeper sofa...
   ss.watchTV();         //Sofa::watchTV()
   //...and then you can fold it out...
   ss.foldOut();          //SleeperSofa::foldOut()
   //...and sleep on it (sort of)
   ss.sleep();           //Bed::sleep()
   return 0;
}
```

The names of both classes — Bed and Sofa — appear after the name SleeperSofa, indicating that SleeperSofa inherits the members of both base classes. Thus, both of the calls ss.sleep() and ss.watchTV() are legal. You can use the class SleeperSofa as either a Bed or a Sofa. Plus the class SleeperSofa can have members of its own, such as foldOut().

Executing the program generates the following output:

```
Building the bed part
Building the sofa part
Putting the two together
Watching TV
Folding the bed out
Trying to get some sleep over here!
```

The bed portion of the sleeper sofa is constructed first cbecause the class Bed appears first in the list of classes from which SleeperSofa inherits (it does not depend upon the order in which the classes are defined). Next the Sofa portion of the SleeperSofa is constructed. Finally, the SleeperSofa gets a crack at it.

Once the SleeperSofa object has been created, main() accesses each of the member functions in turn — first watching the TV on the sofa, then folding the sofa out and finally sleeping on the bed. (Obviously the member functions could have been called in any order.)

Inheritance Ambiguities

Although multiple inheritance is a powerful feature, it introduces several possible problems to the programmer. One is apparent in the preceding example. Notice that both Bed and Sofa contain a member weight. This is logical because both have a measurable weight. The question is, which weight does SleeperSofa inherit?

The answer is both. SleeperSofa inherits a member Bed::weight and a separate member Sofa::weight. Because they have the same name, unqualified references to weight are now ambiguous. The following snippet demonstrates the principle:

```
int main()
{
   // output the weight of the sleepersofa
   SleeperSofa ss;
   cout << “sofa weight = “
```

```
          << ss.weight    //this doesn't work!
          << "\n";
    return 0;
}
```

The program must indicate one of the two weights by specifying the desired base class. The following code snippet is correct:

```
#include <iostream.h>
void fn()
{
    SleeperSofa ss;
    cout << "sofa weight = "
          << ss.Sofa::weight    //specify which weight
          << "\n";
}
```

Although this solution corrects the problem, specifying the base class in the application function isn't desirable because it forces class information to leak outside the class into application code. In this case, `fn()` has to know that `SleeperSofa` inherits from `Sofa`.

These types of so-called name collisions were not possible with single inheritance, but are a constant danger with multiple inheritance.

20 Min. To Go

Virtual Inheritance

In the case of `SleeperSofa`, the name collision on `weight` was more than a mere accident. A `SleeperSofa` doesn't have a bed weight separate from its sofa weight — it has only one weight. The collision occurred because this class hierarchy does not completely describe the real world. Specifically, the classes have not been completely factored.

Thinking about it a little more, it becomes clear that both beds and sofas are special cases of a more fundamental concept: furniture. (I suppose I could get even more fundamental and use something like `object_with_mass`, but furniture is fundamental enough.) Weight is a property of all furniture. This relationship is shown in Figure 24-2.

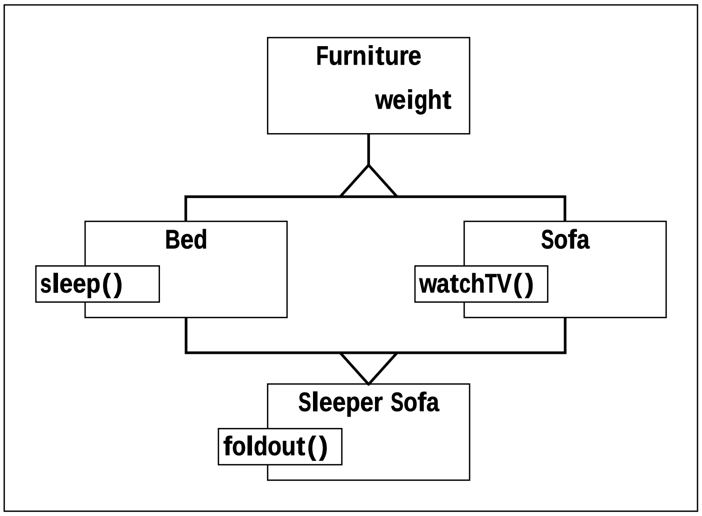

Figure 24-2
Further factoring of beds and sofas

Factoring out the class `Furniture` should relieve the name collision. With much relief and great anticipation of success, I generated the following C++ class hierarchy in the program AmbiguousInheritance:

```
// AmbiguousBaseClass- both Bed and Sofa can inherit from
//                     a common class Furniture
// This program does not compile!
#include <stdio.h>
#include <iostream.h>

class Furniture
{
  public:
    Furniture()
    {
        cout << “Creating the furniture concept”;
    }
    int weight;
};

class Bed : public Furniture
{
  public:
    Bed()
    {
        cout << “Building the bed part\n”;
    }
    void sleep()
    {
        cout << “Trying to get some sleep over here!\n”;
    }
};

class Sofa : public Furniture
{
  public:
    Sofa()
    {
        cout << “Building the sofa part\n”;
    }
    void watchTV()
    {
        cout << “Watching TV\n”;
    }
};

//SleeperSofa - is both a Bed and a Sofa
class SleeperSofa : public Bed, public Sofa
{
  public:
    // the constructor doesn’t need to do anything
    SleeperSofa()
    {
        cout << “Putting the two together\n”;
    }
    void foldOut()
    {
        cout << “Folding the bed out\n”;
    }
};

int main()
```

```
{
   // output the weight of the sleepersofa
   SleeperSofa ss;
   cout << “sofa weight = “
        << ss.weight    //this doesn’t work!
        << “\n”;
   return 0;
}
```

Unfortunately, this doesn’t help at all — `weight` is still ambiguous. “OK,” I say (not really understanding why weight is still ambiguous), “I’ll try casting `ss` to a `Furniture`.”

```
#include <iostream.h>
void fn()
{
   SleeperSofa ss;
   Furniture *pF;
   pF = (Furniture*)&ss; //use a Furniture pointer...
   cout << “weight = “   //...to get at the weight
        << pF->weight
        << “\n”;
};
```

Even this doesn’t work. Now I get an error message indicating that the cast of `SleeperSofa*` to `Furniture*` is ambiguous. What’s going on?

The explanation is straightforward. `SleeperSofa` doesn’t inherit from `Furniture` directly. Both `Bed` and `Sofa` inherit from `Furniture` and then `SleeperSofa` inherits from them. In memory, a `SleeperSofa` looks like Figure 24-3.

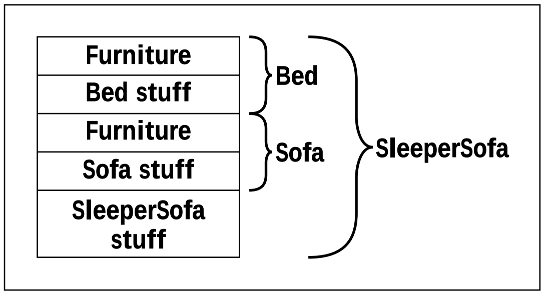

Figure 24-3
Memory layout of a SleeperSofa

You can see that a SleeperSofa consists of a complete Bed followed by a complete Sofa followed by some SleeperSofa unique stuff. Each of these subobjects in SleeperSofa has its own Furniture part, because each inherits from Furniture. Thus, a SleeperSofa contains two Furniture objects.

I haven't created the hierarchy shown in Figure 24-2 after all. The inheritance hierarchy I actually created is the one shown in Figure 24-4.

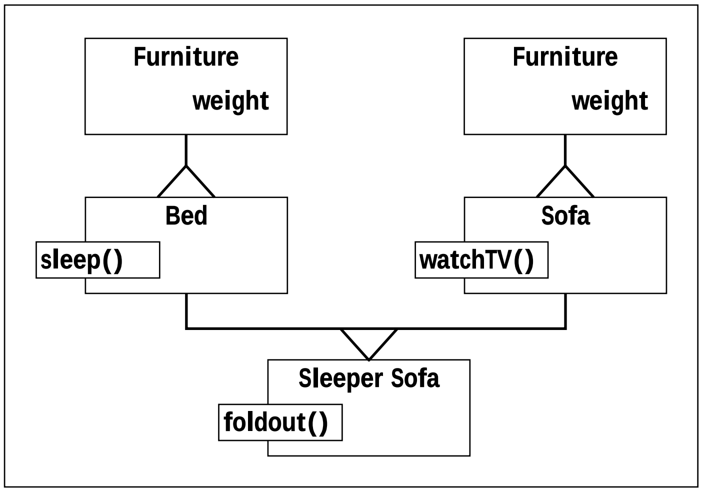

Figure 24-4
Actual result of my first attempt to factor out the Furniture class common to both Bed and Sofa

But this is nonsense. SleeperSofa needs only one copy of Furniture. I want SleeperSofa to inherit only one copy of Furniture, and I want Bed and Sofa to share that one copy.

C++ calls this *virtual inheritance* because it uses the virtual keyword.

Note

I hate this overloading of the term virtual because virtual inheritance has nothing to do with virtual functions.

I return to class SleeperSofa and implement it as follows:

```
// MultipleVirtual - base SleeperSofa on a single copy of
//                   Furniture
// This program does compile!
```

```
#include <stdio.h>
#include <iostream.h>

class Furniture
{
  public:
    Furniture()
    {
        cout << “Creating the furniture concept”;
    }
    int weight;
};

class Bed : virtual public Furniture
{
  public:
    Bed()
    {
        cout << “Building the bed part\n”;
    }
    void sleep()
    {
        cout << “Trying to get some sleep over here!\n”;
    }
};

class Sofa : virtual public Furniture
{
  public:
    Sofa()
    {
        cout << “Building the sofa part\n”;
    }
    void watchTV()
    {
        cout << “Watching TV\n”;
    }
};

//SleeperSofa - is both a Bed and a Sofa
```

```
class SleeperSofa : public Bed, public Sofa
{
  public:
    // the constructor doesn't need to do anything
    SleeperSofa()
    {
        cout << "Putting the two together\n";
    }
    void foldOut()
    {
        cout << "Folding the bed out\n";
    }
};

int main()
{
   // output the weight of the sleepersofa
   SleeperSofa ss;
   cout << "sofa weight = "
        << ss.weight    //this doesn't work!
//      << ss.Sofa::weight // this does work
        << "\n";
   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." A `SleeperSofa` ends up looking like Figure 24-5 in memory.

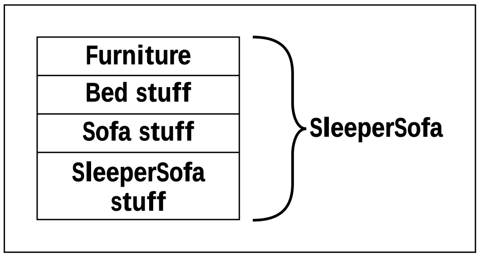

Figure 24-5
Memory layout of SleeperSofa with virtual inheritance

A SleeperSofa inherits Furniture, then Bed minus the Furniture part, followed by Sofa minus the Furniture part. Bringing up the rear are the members unique to SleeperSofa. (This may not be the order of the elements in memory, but that's not important for our purposes.)

The reference in main() to weight is no longer ambiguous because a SleeperSofa contains only one copy of Furniture. By inheriting Furniture virtually, you get the desired inheritance relationship as expressed in Figure 24-2.

If virtual inheritance solves this problem so nicely, why isn't it the norm? There are two reasons. First, virtually inherited base classes are handled internally much differently than normally inherited base classes, and these differences involve extra overhead. (Not that much extra overhead, but the makers of C++ were almost obsessively paranoid about overhead.) Second, sometimes you want two copies of the base class (although this is unusual).

Note

I think virtual inheritance should be the norm.

As an example of a case in which you might not want virtual inheritance, consider a TeacherAssistant who is both a Student and a Teacher, both of which are subclasses of Academician. If the university gives its teaching assistants two IDs — a student ID and a separate teacher ID — class TeacherAssistant will need to contain two copies of class Academician.

10 Min. To Go

Constructing the Objects of Multiple Inheritance

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

- First, the constructor for any virtual base classes is called in the order in which the classes are inherited.
- Then the constructor for any nonvirtual base class is called in the order in which the classes are inherited.
- Next, the constructor for any member objects is called in the order in which the member objects appear in the class.
- Finally, the constructor for the class itself is called.

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.

A Contrary Opinion

Not all object-oriented practitioners think that multiple inheritance is a good idea. In addition, many object-oriented languages don't support multiple inheritance. For example, Java does not support multiple inheritance — it is considered too dangerous and not really worth the trouble.

Multiple inheritance is not an easy thing for the language to implement. This is mostly the compiler's problem (or the compiler writer's problem) and not the programmer's problem. However, multiple inheritance opens the door to additional errors. First, there are the ambiguities mentioned in the section "Inheritance Ambiguities." Second, in the presence of multiple inheritance, casting a pointer from a subclass to a base class often involves changing the value of the pointer in sophisticated and mysterious ways, which can result in unexpected results. 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;
   if ((void*)pB1 == (void*)pB2)
   {
      cout << "Members numerically equal\n";
   }
}
int main()
{
   SubClass sc;
   fn(&sc);
   return 0;
}
```

Done!

`pB1` and `pB2` are not numerically equal even though they came from the same original value, `pSC`, and the message "Members numerically equal" doesn't appear. (Actually, `fn()` is passed a zero because C++ doesn't perform these transmigrations on `null`. See how strange it gets?)

Review

I suggest that you avoid using multiple inheritance until you are comfortable with C++. Single inheritance provides enough expressive power to get used to. Later, you can study the manuals until you're sure that you understand exactly what's going on when you use multiple inheritance. One exception is the use of commercial libraries such as Microsoft's Foundation Classes (MFC), which use multiple inheritance quite a bit. These classes have been checked out and are safe. (You are generally not even aware that you are using multiply inherited base classes when using libraries such as MFC.)

- A class can inherit from more than one class by stringing class names, separated by commas, after the `:`. Although the examples in this base class only used two base classes, there is no reasonable limitation to the number of base classes. Inheritance from more than two base classes is extremely unusual.
- Members that the base classes share are ambiguous to the subclass. That is, if both `BaseClass1` and `BaseClass2` contain a member function `f()`, then `f()` is ambiguous in `Subclass`.
- Ambiguities in the base classes can be resolved by using a class indicator, thus the subclass might refer to `BaseClass1::f()` and `BaseClass2::f()`.
- Having both base classes inherit from a common base class of their own in which common properties have been factored out can solve the problem if the classes inherit virtually.

Quiz Yourself

1. What might we use as the base classes for a class like `CombinationPrinterCopier`? (A printer-copier is a laser printer that can also serve as a copy machine.) (See the introduction section.)
2. Complete the following class description by replacing the question marks:

```
class Printer
{
  public:
    int nVoltage;
    // ....other stuff...
}
class Copier
```

```
{
  public:
    int nVolatage;
    // ....other stuff...
}
class CombinationPinterCopier ?????
{
    // ....other stuff...
}
```

3. What is the main problem that might arise in accessing the voltage of a `CombinationPrinterCopier` object? (See “Inheritance Ambiguities.”)
4. Given that both a `Printer` and a `Copier` are `ElectronicEquipment`, what could be done to solve the voltage problem? (See “Virtual Inheritance.”)
5. What are some of the reasons why multiple inheritance might not be a good thing? (See “A Contrary Opinion.”)