`private` Derivation in EMPMULT

The `manager` and `scientist` classes in EMPMULT are privately derived from the `employee` and `student` classes. There is no need to use public derivation because objects of `manager` and `scientist` never call routines in the `employee` and `student` base classes. However, the `laborer` class must be publicly derived from `employer`, since it has no member functions of its own and relies on those in `employee`.

Constructors in Multiple Inheritance

EMPMULT has no constructors. Let's look at an example that does use constructors, and see how they're handled in multiple inheritance.

Imagine that we're writing a program for building contractors, and that this program models lumber-supply items. It uses a class that represents a quantity of lumber of a certain type: 100 8-foot-long construction grade 2×4s, for example.

The class should store various kinds of data about each such lumber item. We need to know the length (3'–6", for example) and we need to store the number of such pieces of lumber and their unit cost.

We also need to store a description of the lumber we're talking about. This has two parts. The first is the nominal dimensions of the cross-section of the lumber. This is given in inches. For instance, lumber 2 inches by 4 inches (for you metric folks, about 5 cm by 10 cm) is called a *two-by-four*. This is usually written *2×4*. We also need to know the grade of lumber—rough-cut, construction grade, surfaced-four-sides, and so on. We find it convenient to create a `Type` class to hold this data. This class incorporates member data for the nominal dimensions and the grade of the lumber, both expressed as strings, such as *2×6* and *construction*. Member functions get this information from the user and display it.

We'll use the `Distance` class from previous examples to store the length. Finally we create a `Lumber` class that inherits both the `Type` and `Distance` classes. Here's the listing for ENGLMULT:

```
// englmult.cpp
// multiple inheritance with English Distances
#include <iostream>
#include <string>
using namespace std;
////////////////////////////////////////////////////////////////
class Type                          //type of lumber
   {
   private:
      string dimensions;
      string grade;
```

```
   public:                          //no-arg constructor
      Type() : dimensions("N/A"), grade("N/A")
         {  }
                                    //2-arg constructor
      Type(string di, string gr) : dimensions(di), grade(gr)
         {  }
      void gettype()                //get type from user
         {
         cout << "   Enter nominal dimensions (2x4 etc.): ";
         cin >> dimensions;
         cout << "   Enter grade (rough, const, etc.): ";
         cin >> grade;
         }
      void showtype() const         //display type
         {
         cout << "\n   Dimensions: " << dimensions;
         cout << "\n   Grade: " << grade;
         }
   };
////////////////////////////////////////////////////////////////
class Distance                      //English Distance class
   {
   private:
      int feet;
      float inches;
   public:                          //no-arg constructor
      Distance() : feet(0), inches(0.0)
         {  }                       //constructor (two args)
      Distance(int ft, float in) : feet(ft), inches(in)
         {  }
      void getdist()                //get length from user
         {
         cout << "   Enter feet: ";  cin >> feet;
         cout << "   Enter inches: ";  cin >> inches;
         }
      void showdist() const         //display distance
         { cout  << feet << "\'-" << inches << '\"'; }
   };
////////////////////////////////////////////////////////////////
class Lumber : public Type, public Distance
   {
   private:
      int quantity;                       //number of pieces
      double price;                       //price of each piece
   public:                                //constructor (no args)
      Lumber() : Type(), Distance(), quantity(0), price(0.0)
```

```
      {  }
                                        //constructor (6 args)
   Lumber( string di, string gr,        //args for Type
           int ft, float in,            //args for Distance
           int qu, float prc ) :        //args for our data
           Type(di, gr),                //call Type ctor
           Distance(ft, in),            //call Distance ctor
           quantity(qu), price(prc)     //initialize our data
       {  }
   void getlumber()
      {
      Type::gettype();
      Distance::getdist();
      cout << "   Enter quantity: "; cin >> quantity;
      cout << "   Enter price per piece: "; cin >> price;
      }
   void showlumber() const
      {
      Type::showtype();
      cout << "\n   Length: ";
      Distance::showdist();
      cout << "\n   Price for " << quantity
          << " pieces: $" << price * quantity;
      }
};
////////////////////////////////////////////////////////////////
int main()
   {
   Lumber siding;                   //constructor (no args)

   cout << "\nSiding data:\n";
   siding.getlumber();              //get siding from user

                                    //constructor (6 args)
   Lumber studs( "2x4", "const", 8, 0.0, 200, 4.45F );

                                    //display lumber data
   cout << "\nSiding";  siding.showlumber();
   cout << "\nStuds";     studs.showlumber();
   cout << endl;
   return 0;
   }
```

The major new feature in this program is the use of constructors in the derived class `Lumber`. These constructors call the appropriate constructors in `Type` and `Distance`.

No-Argument Constructor

The no-argument constructor in `Type` looks like this:

```
Type()
   { strcpy(dimensions, "N/A"); strcpy(grade, "N/A"); }
```

This constructor fills in "N/A" (not available) for the `dimensions` and `grade` variables so the user will be made aware if an attempt is made to display data for an uninitialized lumber object.

You're already familiar with the no-argument constructor in the `Distance` class:

```
Distance() : feet(0), inches(0.0)
   {  }
```

The no-argument constructor in `Lumber` calls both of these constructors.

```
Lumber() : Type(), Distance(), quantity(0), price(0.0)
         {  }
```

The names of the base-class constructors follow the colon and are separated by commas. When the `Lumber()` constructor is invoked, these base-class constructors—`Type()` and `Distance()`—will be executed. The `quantity` and `price` attributes are also initialized.

Multi-Argument Constructors

Here is the two-argument constructor for `Type`:

```
Type(string di, string gr) : dimensions(di), grade(gr)
         {  }
```

This constructor copies string arguments to the dimensions and grade member data items.

Here's the constructor for `Distance`, which is again familiar from previous programs:

```
Distance(int ft, float in) : feet(ft), inches(in)
   {  }
```

The constructor for `Lumber` calls both of these constructors, so it must supply values for their arguments. In addition it has two arguments of its own: the quantity of lumber and the unit price. Thus this constructor has six arguments. It makes two calls to the two constructors, each of which takes two arguments, and then initializes its own two data items. Here's what it looks like:

```
Lumber( string di, string gr,      //args for Type
        int ft, float in,          //args for Distance
        int qu, float prc ) :      //args for our data
        Type(di, gr),              //call Type ctor
        Distance(ft, in),          //call Distance ctor
        quantity(qu), price(prc)   //initialize our data
   {  }
```