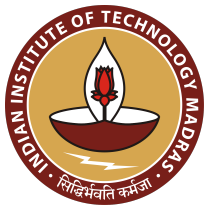
Class Hierarchies

Saurav Samantaray

Department of Mathematics

Indian Institute of Technology Madras

October 28, 2024



Class Hierarchies

- Until now inheritance has been used to add functionality to an existing class.
- let's look at an example where inheritance is used for a different purpose: as part of the original design of a program.
- The next example models a database of employees of a widget company.
- We've simplified the situation so that only three kinds of employees are represented.
- *Managers* manage, *scientists* perform research to develop better widgets, and *labourers* operate the dangerous widget-stamping presses.

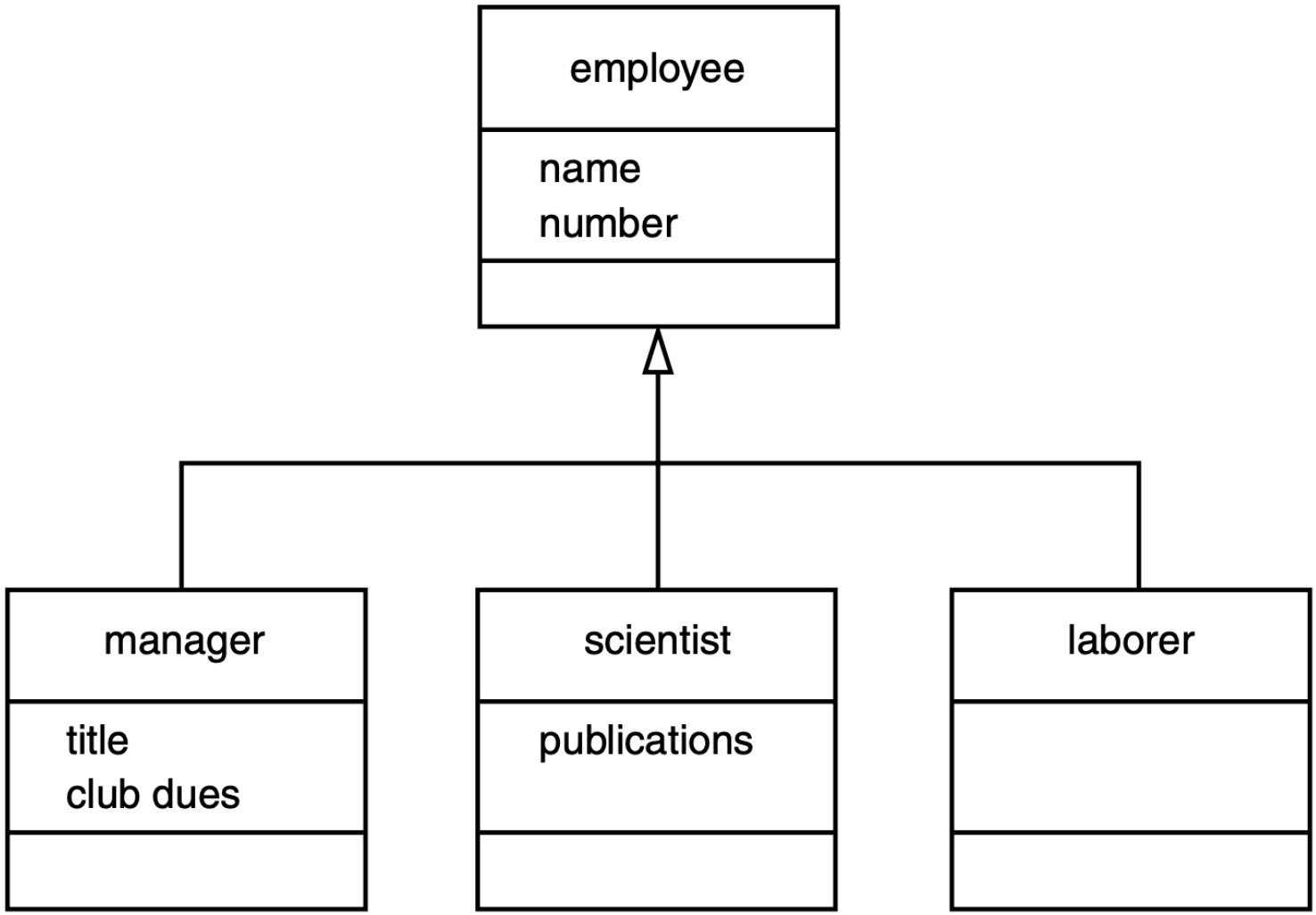
Employee Class

- The database stores a name and an employee identification number for all employees, no matter what their category.
- However, for managers, it also stores their titles and golf club dues
- For scientists, it stores the number of scholarly articles they have published.
- Labourers need no additional data beyond their names and identification numbers.

Employee Class

- The example program starts with a base class `employee`.
- This class handles the employee's last name and employee number.
- From this class three other classes are derived: `manager`, `scientist`, and `labourer`.
- The `manager` and `scientist` classes contain additional information about these categories of employee, and member functions to handle this information.
- the `labourer` class doesn't have any additional data to be introduced.
- See "employ.cpp"

UML class diagram for Employee Class



"Abstract" Base Class

- Notice that we don't define any objects of the base class employee.
- We use this as a general class whose sole purpose is to act as a base from which other classes are derived.
- The `labourer` class operates identically to the employee class, since it contains no additional data or functions.
- It may seem that the `labourer` class is unnecessary, but by making it a separate class we emphasise that all classes are descended from the same source, `employee`.
- Also, if in the future we decided to modify the `labourer` class, we would not need to change the declaration for `employee`.
- Classes used only for deriving other classes, as `employee` is in EMPLOY, are sometimes loosely called abstract classes,
- meaning that no actual instances (objects) of this class are ever created.

Constructors and Member Functions

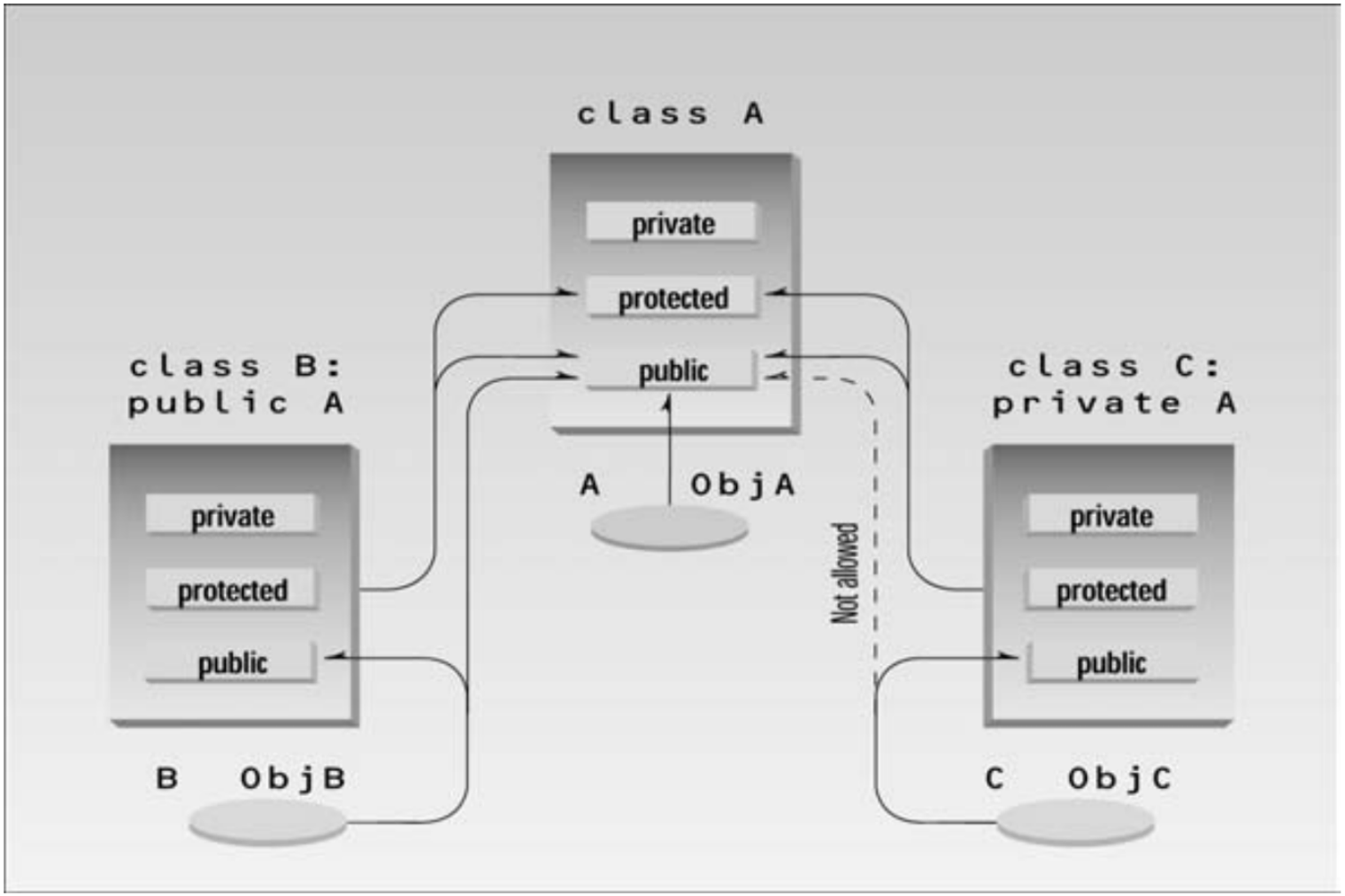
- There are no constructors in either the base or derived classes, so the compiler creates objects of the various classes automatically when it encounters definitions like
  `manager m1, m2;`
- using the default constructor for manager calling the default constructor for `employee`.
- The `getdata()` and `putdata()` functions in employee accept a name and number from the user and display a name and number.
- Functions also called `getdata()` and `putdata()` in the manager and `scientist` classes use the functions in `employee`, and also do their own work.
- In manager, the `getdata()` function asks the user for a title and the amount of golf club dues, and `putdata()` displays these values.
- In `scientist`, these functions handle the number of

Content

Public and Private Inheritance

- See ”pubpriv.cpp”
- The program specifies a base class, `A`, with private, protected, and public data items.
- Two classes, `B` and `C`, are derived from `A`.
- `B` is publicly derived and `C` is privately derived.
- functions in the derived classes can access protected and public data in the base class.
- Objects of the derived classes cannot access private or protected members of the base class.
- What’s new is the difference between publicly derived and privately derived classes.
- Objects of the publicly derived class `B` can access public members of the base class `A`,
- while objects of the privately derived class `C` cannot;
- they can only access the public members of their own derived class.

Public and Private Inheritance

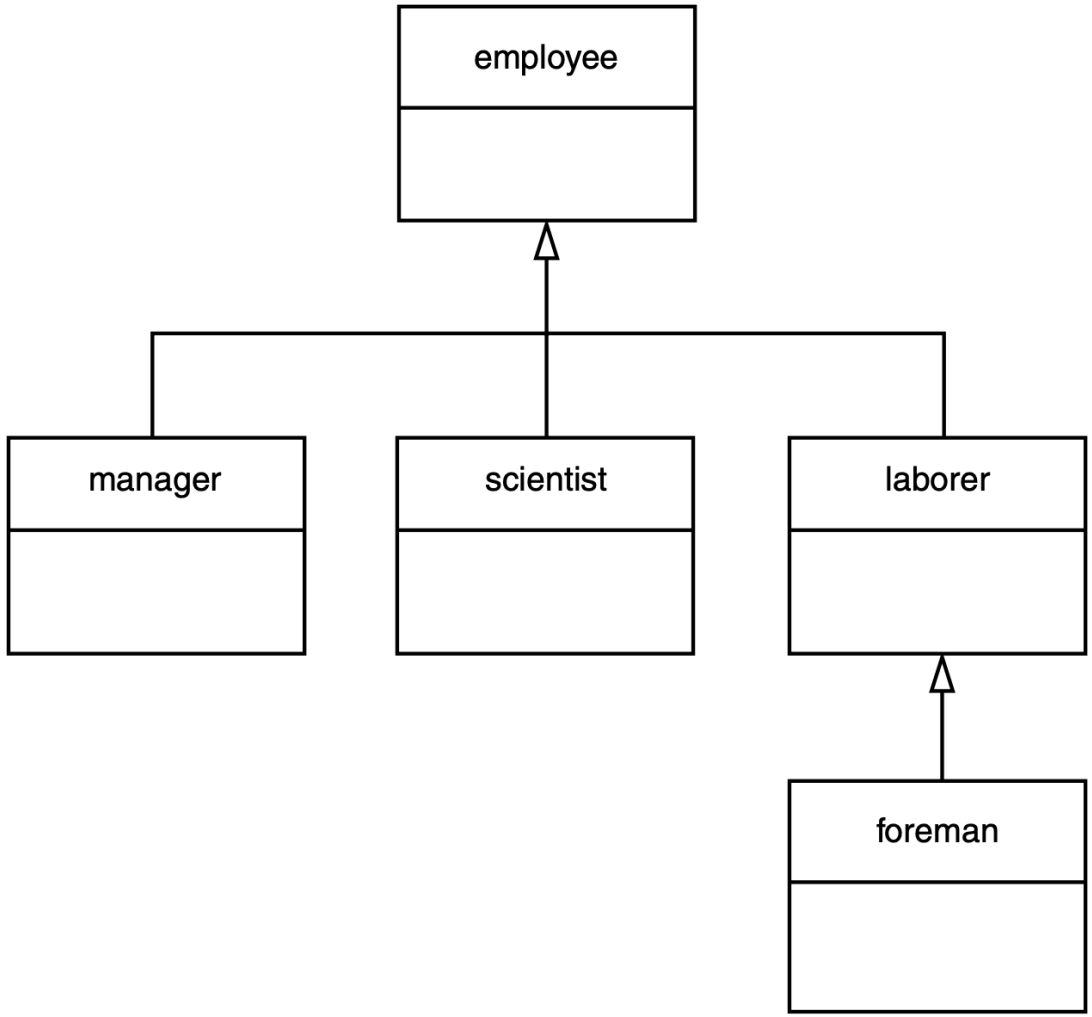


If we don't supply any access specifier when creating a class, private is assumed.

Levels of Inheritance

- Classes can be derived from classes that are themselves derived.
- ```
  class A
    { };
  class B : public A
    { };
  class C : public B
    { };
  ```
- Here `B` is derived from `A`, and `C` is derived from `B`.
- The process can be extended to an arbitrary number of levels—`D` could be derived from `C`, and so on.
- suppose that we decided to add a special kind of `labourer` called a foreman to the EMPLOY program.
- Since a `foreman` is a kind of `labourer`, the `foreman` class is derived from the `labourer` class
  ```

UML class diagram for New Employee



Content

Multiple Inheritance

- A class can be derived from more than one base class.
- This is called multiple inheritance.

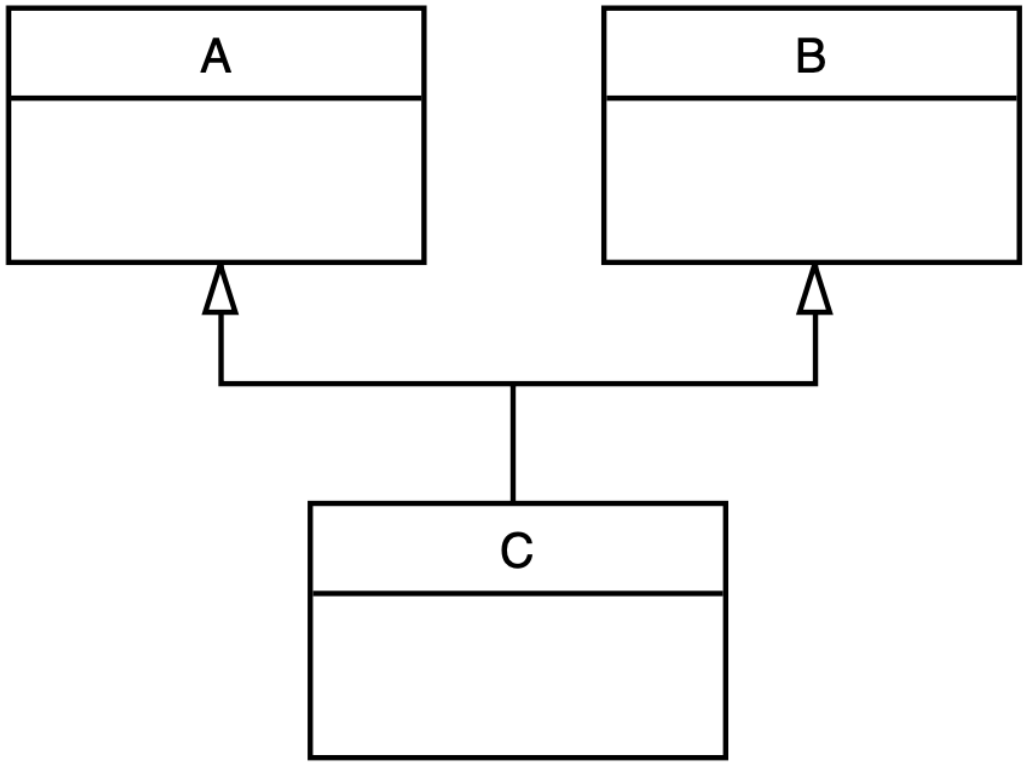


Figure 1: UML class diagram for multiple inheritance.

Multiple Inheritance

- The syntax for multiple inheritance is similar to that for single inheritance.
- the relationship is expressed like this:

```
class A          // base class A
{ };
 class B          // base class B
{ };
class C : public A, public B     // C is
derived from A and B
{ };
```

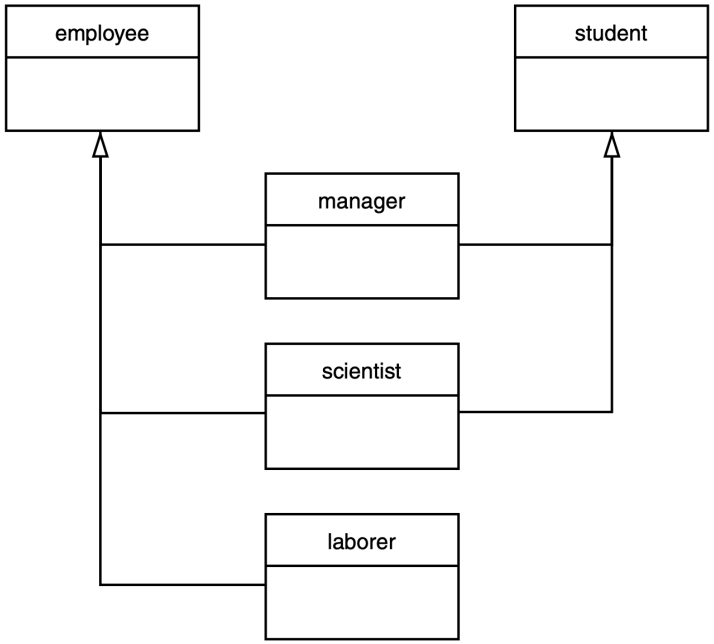
- The base classes from which `C` is derived are listed following the colon in `C`'s specification;
- they are separated by commas.

Member Functions in Multiple Inheritance

- As an example of multiple inheritance, suppose that we need to record the educational experience of some of the employees in the EMPLOY program.
- Let's also suppose that, perhaps in a different project, we've already developed a class called `student` that models students with different educational backgrounds.
- We decide that instead of modifying the employee class to incorporate educational data, we will add this data by multiple inheritance from the `student` class.
- The `student` class stores the name of the school or university last attended and the highest degree received.
- Two member functions, `getedu()` and `putedu()`, ask the user for this information and display it.
- Educational information may not be relevant to every class of employee.

Member Functions in Multiple Inheritance

- See ”empmult.cpp”.
- Let’s suppose, that we don’t need to record the educational experience of labourers;
- it’s only relevant for managers and scientists.
- We therefore modify `manager` and `scientist` so that they inherit from both the `employee` and `student` classes.



Member Functions in Multiple Inheritance

- The `getdata()` and `putdata()` functions in the `manager` and `scientist` classes incorporate calls to functions in the `student` class, such as
  `student::getedu();`
  and
  `student::putedu();`
- These routines are accessible in manager and `scientist` because these classes are descended from `student`.

Constructors 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: 108-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 is called a two-by-four. This is usually written 2×4.

Constructors in Multiple Inheritance

- 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.
- "englmult.cpp".

Constructors in Multiple Inheritance

- 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"); }
```

- and in `Distance` class looks like:

```
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

Constructors in Multiple Inheritance

Multi-Argument Constructors

- The two-argument constructor for `Type`:

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

- The constructor for `Distance`:

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

- The constructor for `Lumber` takes in values for their arguments.
- In addition it has two arguments of its own: the quantity of lumber and the unit price.

```
Lumber( string di, string gr,   //args for Type
    int ft, float in,     //args for Distance
    int qu, float prc ) :    //args for our data
```

Constructors in Multiple Inheritance

- The constructor for `Lumber` calls both of the constructors i.e. of `Type` and `Distance`.
- ```
  ....
  Type(di, gr),          //call Type ctor
  Distance(ft, in),      //call Distance ctor
  quantity(qu), price(prc)   //initlize our data
      { }
  ```
- It makes two calls to the two constructors;
- each of which takes two arguments,
- and then initializes its own two data items.
  ```

Content

Ambiguity in Multiple Inheritance

- Odd sorts of problems may surface in certain situations involving multiple inheritance.
- Here's a common one.
- Two base classes have functions with the same name, while a class derived from both base classes has no function with this name.
- How do objects of the derived class access the correct base class function?
- The name of the function alone is insufficient, since the compiler can't figure out which of the two functions is meant.
- See "ambigu.cpp".
- Both base classes `A` and `B` have a function
  `void show () {}`
  each.

Ambiguity in Multiple Inheritance

- Even though the body of each of the function is different, they have the same protype.
- If one uses the following command for an object `objc` of Class `C`:
  `objc.show();`
  it leaves the compiler dumbfound.
- The compiler is caught between the classes `A` and `B`, and there is no extra information provided, so that it can make a choice.
- In such a scenario the compiler will throw out an error.
- The problem is resolved using the scope-resolution operator to specify the class in which the function lies.
- `objC.A::show();` refers to the version of `show()` that's in the `A` class, and;
- `objC.B::show();` refers to the version of `show()` that's in the `B`

Ambiguity in Multiple Inheritance

- Another kind of ambiguity arises if we derive a class from two classes that are each derived from the same class.
- Classes `B` and `C` are both derived from class `A`, and class `D` is derived by multiple inheritance from both `B` and `C`.
- Trouble starts if we try to access a member function in class `A` from an object of class `D`.
- In this example `objD` tries to access `func()`.
- However, both `B` and `C` contain a copy of `func()`, inherited from `A`.
- The compiler can't decide which copy to use, and signals an error.
- The fact that such ambiguities can arise causes many experts to recommend avoiding multiple inheritance altogether.
- We should certainly not use it in serious programs unless we have considerable experience.

Ambiguity in Multiple Inheritance

```
//diamond.cpp
//investigates diamond-shaped multiple inheritance
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class A
{
   public:
      void func();
};
class B : public A  { };
class C : public A  { };
class D : public B, public C { };
////////////////////////////////////////////////////////////
int main()
{
   D objD;
   objD.func();    //ambiguous: won't compile
   return 0;
}
```