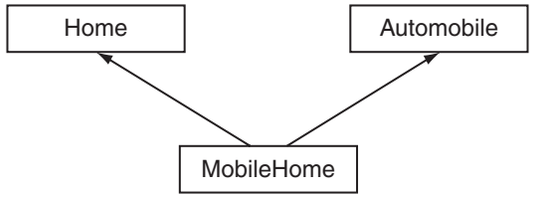
APPENDIX H

Multiple and Virtual Inheritance

Introduction

Like all object-oriented programming languages, C++ supports the concept of inheritance. Unlike most other such languages, however, C++ also supports multiple inheritance. *Multiple inheritance* is a type of inheritance in which a new class is simultaneously derived from two or more base classes. A programmer should consider using multiple inheritance when modeling an object that seems to simultaneously belong to more than one class. For example, one might have a class `Home` to model structures in which people live, and another class `Automobile` to model certain types of machines that humans use for purposes of transportation. One can then envision a class `MobileHome` whose objects are simultaneously instances of the `Home` and `Automobile` classes. This relationship is depicted in Figure H-1.

Figure H-1



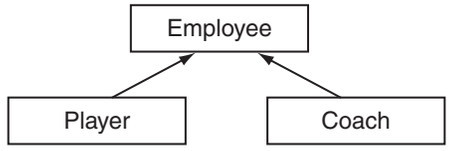
In this figure, the upward-pointing arrows depict an *is-a* relationship, illustrating the fact that a mobile home is at once both a home and an automobile. In C++, the relationship would be represented by the class declarations

```
class Automobile
{
   // Automobile members
};
class Home
{
  // Home members
};

class MobileHome : public Automobile, public Home
{
  // Additional members of MobileHome
};
```

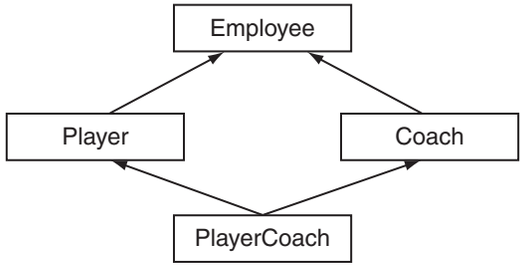
As another example of a situation in which the use of multiple inheritance might be appropriate, consider a professional sports organization that has various types of employees. Players and coaches are both employees, so the inheritance hierarchy shown in Figure H-2 is quite natural.

Figure H-2



Now consider an employee that is both a player and a coach. Such people have traditionally been called *player-coaches*. Because a player-coach is both a player and a coach, we get the inheritance diagram shown in Figure H-3.

Figure H-3



We will refer to the situation depicted in Figure H-3 as the *multiple inheritance diamond*. The upper part of the figure is the usual single inheritance structure because it shows that both `Player` and `Coach` have a single base class. The bottom part of the figure depicts multiple inheritance: it shows that `PlayerCoach` has two base classes.

C++ Implementation of Multiple Inheritance

In C++, the inheritance hierarchy shown in the upper part of Figure H-3 might be implemented with code as follows:

Contents of `PlayerCoach.h`

```
 1 #include <string>
 2 #include <iostream>
 3 using namespace std;
 4 class Employee
 5 {
 6    string name;
 7 public:
 8    string getName(){ return name; }
 9    Employee(string name){ this->name = name; }
10 };
11
```

```
12 class Player : public Employee
13 {
14    int salary;
15 public:
16    int getSalary(){ return salary; }
17    void play()
18    {
19        cout << getName() << " is playing.\n";
20    }
21     //Constructor
22    Player(string name, int salary):
23         // Constructor initialization
24         Employee(name), salary(salary)
25    {
26    }
27 };
28
29 class Coach : public Employee
30 {
31     int salary;
32 public :
33     int getSalary(){ return salary; }
34     void coach()
35     {
36        cout << getName() << " is coaching.\n";
37     }
38     //Constructor
39     Coach(string name, int salary):
40          // Constructor initialization
41          Employee(name), salary(salary)
42     {
43     }
44 };
```

Contents of PlayerCoach.cpp

```
1 #include "PlayerCoach.h"
2 int main()
3 {
4   // Create Player and Coach objects
5   Player phil("Phillip", 20000);
6   Coach carol("Carol", 30000);
7   // Call play and coach member functions
8   phil.play();
9   carol.coach();
10
11   return 0;
12 }
```

Program Output

```
Phillip is playing.
Carol is coaching.
```

Let us now consider the use of multiple inheritance to define a `PlayerCoach` class. Before we do so, however, we should mention that multiple inheritance is rarely used in C++, and indeed, it can be the cause of many problems in programs that use it. We will point out some of these difficulties as we work through our `PlayerCoach` example.

The simplest class that derives from both `Player` and `Coach` is

```
#include "playercoach.h"
class PlayerCoach : public Player, public Coach
{
};
```

Note that `PlayerCoach` has no members other than those it inherits from its two base classes. We deliberately do this to keep our example as simple as possible. In a more realistic situation a derived class would have additional members not found in its base classes. Unfortunately, this class will not compile. The reason is both of its base classes have non-default constructors that need to be passed arguments. We can try to solve that problem by equipping `PlayerCoach` with a constructor initialization list, as shown here.

Contents of `PlayerCoachMult.h`

```
 1 #include "playerCoach.h"
 2 using namespace std;
 3 class PlayerCoach : public Player, public Coach
 4 {
 5 public:
 6     PlayerCoach(string name, int playerSalary, int coachSalary):
 7     Player(name, playerSalary), Coach(name, coachSalary)
 8     {
 9     }
10 };
```

As you can see, the constructor for `PlayerCoach` takes three parameters: the name of the employee, the employee's salary as a player, and the employee's salary as a coach. The constructor then invokes the constructors of its base classes, passing each the appropriate sequence of parameters. But now, if we try to print the employee's salary,

```
cout << pc.getSalary();
```

We run into another problem. The `PlayerCoach` class has two different versions of the `getSalary()` member function, one inherited from each of its base classes, and the compiler cannot determine which of the two functions to call. To get the statement to compile, we must remove the ambiguity by using the name of a base class together with the scope-resolution operator `::` as shown in lines 6, 8, 10 and 12 of the following program.

Contents of `of MultInherit1.cpp`

```
 1 #include "PlayerCoachMult.h"
 2 int main()
 3 {
 4   PlayerCoach pc("Peter Collins", 40000, 50000);
 5   cout << "The name of the Employee is "
 6        << pc.Player::getName() << "\n";
 7   cout << "The name of the Employee is "
 8        << pc.Coach::getName() << "\n";
```

```
 9   cout << "Player salary is ";
10   cout << pc.Player::getSalary() << "\n";
11   cout << "Total salary is ";
12   cout << pc.Player::getSalary() + pc.Coach::getSalary() << endl;
13   return 0;
14 }
```

Program Output

```
The name of the Employee is Peter Collins
The name of the Employee is Peter Collins
Player salary is 40000
Total salary is 90000
```

It is not surprising that the `PlayerCoach` class has two versions of the `getSalary()` function because it inherits one from each of its base classes. In fact, `PlayerCoach` will have two copies of every member of its grandparent class `Employee`. This is because a copy of each member of `Employee` is separately inherited by both `Player` and `Coach`, and these different copies are then inherited by `PlayerCoach`. This is why lines 6 and 8 of `MultiInherit1.cpp` have to use the scope-resolution operator when accessing the `get-Name()` function inherited from `Employee`.

Whenever a class ***K*** is derived from two different classes that directly or indirectly share the same base class, there will be two distinct copies of the common base class object in the class ***K***. Having two copies of the base class is unnecessary and wasteful of memory. It forces the programmers using the derived class to resort to the use of the scope-resolution operator to remove the resulting ambiguity. Over time, this may lead to errors in the program due to inconsistencies in application of the scope-resolution operator. Although not possible with our simple example, one can imagine updating `pc.Player::name` and then later accessing `pc.Coach::name`. For these reasons, multiple inheritance should be avoided when possible, and used with great care when it cannot be.

Virtual Inheritance

The problem of inheriting multiple copies of a shared base class are addressed through the concepts of *virtual inheritance* and *virtual base classes*. A class being derived from another class can declare its base class `virtual` by prefixing the keyword virtual to the base class specification. For example, the `Player` and `Coach` classes can declare their base class virtual as follows:

```
class Player : virtual public Employee
{
  // Constructor for Player invokes constructor for Employee
};

class Coach: virtual public Employee
{
  // Constructor for Player invokes constructor for Employee
};
```

When processing a declaration of a class ***K*** that inherits from multiple classes with a common base class ***B*** the compiler will ensure that no more than a single copy of ***B*** is included in ***K*** if ***B*** has been declared virtual. While solving the problem of multiple copies of a

shared base class, virtual inheritance brings with it a few problems of its own. Look again at the constructor initialization list in Line 7 in the listing of the `PlayerCoach` class in the `PlayerCoachMult.h`:

```
Player(name, playerSalary), Coach(name, coachSalary)
```

Because `Employee` is a virtual base class, the `PlayerCoach` class has only one copy of it. The invocation of the `Player` and `Coach` constructors result in two invocations of the constructor for the single `Employee` object inside `PlayerCoach`. This fact will result in yet another compiler error.

To solve the problem of multiple initialization of a virtual base class, the C++ compiler will ignore invocations of constructors of a virtual base class in all intermediate classes along the various derivation chains. To make sure that the virtual base class gets initialized, C++ requires the *most derived class* (the class at the common end of the multiple derivation chains) to specify the initialization of the common virtual base class. As an example, look at Figure H-3. The inheritance diamond shown there has two derivation chains (the left and right sides of the diamond) and the intermediate classes along these two chains are respectively `Player` and `Coach`. Accordingly, the compiler will ignore the invocations `Employee(name)` in the constructors of those two classes. To ensure that the `Employee` object will be initialized, a call to its constructor must be included in the constructor initialization list of `PlayerCoach` which is of course the most derived class. This modification is shown in the following code listing.

Contents of `vPlayerCoach.h`

```
 1 #include <string>
 2 #include <iostream>
 3 using namespace std;
 4 class Employee
 5 {
 6    string name;
 7 public:
 8    string getName(){ return name; }
 9    Employee(string name){ this->name = name; }
10 };
11
12 class Player : virtual public Employee
13 {
14    int salary;
15 public:
16    int getSalary(){ return salary; }
17    void play()
18    {
19        cout << getName() << " is playing.\n";
20    }
21     //Constructor
22    Player(string name, int salary):
23         // Constructor initialization
24         Employee(name), salary(salary)
25    {
26    }
27 };
28
```

```
29 class Coach : virtual public Employee
30 {
31     int salary;
32 public :
33     int getSalary(){ return salary; }
34     void coach()
35     {
36        cout << getName() << " is coaching.\n";
37     }
38     //Constructor
39     Coach(string name, int salary):
40          // Constructor initialization
41          Employee(name), salary(salary)
42     {
43     }
44 };
```

Contents of vPlayerCoachMult.h

```
 1 #include "vplayerCoach.h"
 2 using namespace std;
 3 class PlayerCoach : public Player, public Coach
 4 {
 5 public:
 6     PlayerCoach(string name, int playerSalary, int coachSalary):
 7     Player(name, playerSalary), Coach(name, coachSalary),
 8     Employee(name)
 9     {
10     }
11 };
```

Note that the changes required to support virtual inheritance are not extensive. In our case, we have added the keyword virtual in Lines 12 and 29 of the `PlayerCoach.h` file, and a call to the `Employee` constructor in Line 8 of the last file shown. The program can be tested with the same main function used for the previous example.