PEARSON

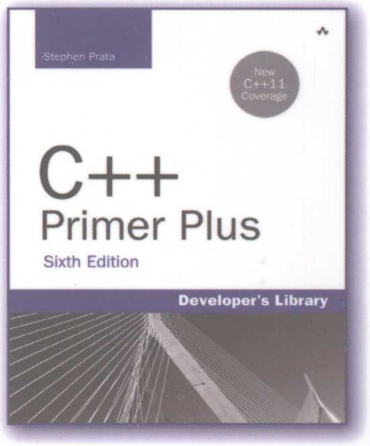
C和C++实务精选
品味岁月积淀，读享技术菁华

C++ Primer Plus

（第6版）英文版（下册）

[美] Stephen Prata 著

- 经久不衰的C++畅销经典教程
- 涵盖C++11新标准



人民邮电出版社
POSTS & TELECOM PRESS

PEARSON

C++ Primer Plus

（第6版）英文版（下册）

[美] Stephen Prata 著

人民邮电出版社
北京

Table of Contents

13 Class Inheritance 707
Beginning with a Simple Base Class 708
Inheritance: An *Is-a* Relationship 720
Polymorphic Public Inheritance 722
Static and Dynamic Binding 737
Access Control: `protected` 745
Abstract Base Classes 746
Inheritance and Dynamic Memory Allocation 757
Class Design Review 766
Summary 778
Chapter Review 779
Programming Exercises 780

14 Reusing Code in C++ 785
Classes with Object Members 786
Private Inheritance 797
Multiple Inheritance 808
Class Templates 830
Summary 866
Chapter Review 869
Programming Exercises 871

15 Friends, Exceptions, and More 877
Friends 877
Nested Classes 889
Exceptions 896
Runtime Type Identification 933
Type Cast Operators 943
Summary 947
Chapter Review 947
Programming Exercises 949

16 The `string` Class and the Standard Template Library 951
The `string` Class 952
Smart Pointer Template Classes 968
The Standard Template Library 978
Generic Programming 992
Function Objects (a.k.a. Functors) 1026
Algorithms 1035
Other Libraries 1045

Summary 1054
Chapter Review 1056
Programming Exercises 1057

17 Input, Output, and Files 1061
An Overview of C++ Input and Output 1062
Output with `cout` 1069
Input with `cin` 1093
File Input and Output 1114
Incore Formatting 1142
Summary 1145
Chapter Review 1146
Programming Exercises 1148

18 Visiting with the New C++ Standard 1153
C++11 Features Revisited 1153
Move Semantics and the Rvalue Reference 1164
New Class Features 1178
Lambda Functions 1184
Wrappers 1191
Variadic Templates 1197
More C++11 Features 1202
Language Change 1205
What Now? 1207
Summary 1208
Chapter Review 1209
Programming Exercises 1212

Appendixes

A Number Bases 1215

B C++ Reserved Words 1221

C The ASCII Character Set 1225

D Operator Precedence 1231

E Other Operators 1235

F The `string` Template Class 1249

G The Standard Template Library Methods and Functions 1271

H Selected Readings and Internet Resources 1323

I Converting to ISO Standard C++ 1327

J Answers to Chapter Reviews 1335

Index 1367

13

Class Inheritance

In this chapter you'll learn about the following:

- Inheritance as an *is-a* relationship
- How to publicly derive one class from another
- Protected access
- Constructor member initializer lists
- Upcasting and downcasting
- Virtual member functions
- Early (static) binding and late (dynamic) binding
- Abstract base classes
- Pure virtual functions
- When and how to use public inheritance

One of the main goals of object-oriented programming is to provide reusable code. When you develop a new project, particularly if the project is large, it's nice to be able to reuse proven code rather than to reinvent it. Employing old code saves time and because it has already been used and tested, can help suppress the introduction of bugs into a program. Also the less you have to concern yourself with details, the better you can concentrate on overall program strategy.

Traditional C function libraries provide reusability through predefined, precompiled functions, such as `strlen()` and `rand()`, that you can use in your programs. Many vendors furnish specialized C libraries that provide functions beyond those of the standard C library. For example, you can purchase libraries of database management functions and of screen control functions. However, function libraries have a limitation: Unless the vendor supplies the source code for its library functions (and often it doesn't), you can't extend or modify the functions to meet your particular needs. Instead, you have to shape your program to meet the workings of the library. Even if the vendor does supply the source code, you run the risk of unintentionally modifying how part of a function works or of altering the relationships among library functions as you add your changes.

C++ classes bring a higher level of reusability. Many vendors now offer class libraries, which consist of class declarations and implementations. Because a class combines data representation with class methods, it provides a more integrated package than does a function library. A single class, for example, may provide all the resources for managing a dialog box. Often class libraries are available in source code, which means you can modify them to meet your needs. But C++ has a better method than code modification for extending and modifying classes. This method, called *class inheritance*, lets you derive new classes from old ones, with the derived class inheriting the properties, including the methods, of the old class, called a *base class*. Just as inheriting a fortune is usually easier than earning one from scratch, deriving a class through inheritance is usually easier than designing a new one. Here are some things you can do with inheritance:

- You can add functionality to an existing class. For example, given a basic array class, you could add arithmetic operations.
- You can add to the data that a class represents. For example, given a basic string class, you could derive a class that adds a data member representing a color to be used when displaying the string.
- You can modify how a class method behaves. For example, given a `Passenger` class that represents the services provided to an airline passenger, you can derive a `FirstClassPassenger` class that provides a higher level of services.

Of course, you could accomplish the same aims by duplicating the original class code and modifying it, but the inheritance mechanism allows you to proceed by just providing the new features. You don't even need access to the source code to derive a class. Thus, if you purchase a class library that provides only the header files and the compiled code for class methods, you can still derive new classes based on the library classes. Conversely, you can distribute your own classes to others, keeping parts of your implementation secret, yet still giving your clients the option of adding features to your classes.

Inheritance is a splendid concept, and its basic implementation is quite simple. But managing inheritance so that it works properly in all situations requires some adjustments. This chapter looks at both the simple and the subtle aspects of inheritance.

Beginning with a Simple Base Class

When one class inherits from another, the original class is called a *base class*, and the inheriting class is called a *derived class*. So to illustrate inheritance, let's begin with a base class. The Webtown Social Club has decided to keep track of its members who play table tennis. As head programmer for the club, you have designed the simple `TableTennisPlayer` class defined in Listings 13.1 and 13.2.

Listing 13.1 **tabtenn0.h**

```
// tabtenn0.h -- a table-tennis base class
#ifndef TABTENN0_H_
#define TABTENN0_H_
#include <string>
```

```
using std::string;
// simple base class
class TableTennisPlayer
{
private:
    string firstname;
    string lastname;
    bool hasTable;
public:
    TableTennisPlayer (const string & fn = "none",
                       const string & ln = "none", bool ht = false);
    void Name() const;
    bool HasTable() const { return hasTable; };
    void ResetTable(bool v) { hasTable = v; };
};
#endif
```

Listing 13.2 **tabtenn0.cpp**

```
//tabtenn0.cpp -- simple base-class methods
#include "tabtenn0.h"
#include <iostream>

TableTennisPlayer::TableTennisPlayer (const string & fn,
    const string & ln, bool ht) : firstname(fn),
           lastname(ln), hasTable(ht) {}

void TableTennisPlayer::Name() const
{
    std::cout << lastname << ", " << firstname;
}
```

All the `TableTennisPlayer` class does is keep track of the players' names and whether they have tables. There are a couple of points to notice. First, the class uses the standard `string` class to hold the names. This is more convenient, flexible, and safer than using a character array. And it is rather more professional than the `String` class of Chapter 12, "Classes and Dynamic Memory Allocation." Second, the constructor uses the member initializer list syntax introduced in Chapter 12. You could also do this:

```
TableTennisPlayer::TableTennisPlayer (const string & fn,
                  const string & ln, bool ht)
{
    firstname = fn;
    lastname = ln;
    hasTable = ht;
}
```

However, this approach has the effect of first calling the default `string` constructor for `firstname` and then invoking the `string` assignment operator to reset `firstname` to `fn`. But the member initializer list syntax saves a step by just using the `string` copy constructor to initialize `firstname` to `fn`.

Listing 13.3 shows this modest class in action.

Listing 13.3 **usett0.cpp**

```
// usett0.cpp -- using a base class
#include <iostream>
#include "tabtenn0.h"

int main ( void )
{
    using std::cout;
    TableTennisPlayer player1("Chuck", "Blizzard", true);
    TableTennisPlayer player2("Tara", "Boomdea", false);
    player1.Name();
    if (player1.HasTable())
        cout << ": has a table.\n";
    else
        cout << ": hasn't a table.\n";
    player2.Name();
    if (player2.HasTable())
        cout << ": has a table";
    else
        cout << ": hasn't a table.\n";

    return 0;
}
```

And here's the output of the program in Listings 13.1, 13.2, and 13.3:

```
Blizzard, Chuck: has a table.
Boomdea, Tara: hasn't a table.
```

Note that the program uses constructors with C-style string arguments:

```
TableTennisPlayer player1("Chuck", "Blizzard", true);
TableTennisPlayer player2("Tara", "Boomdea", false);
```

But the formal parameters for the constructor were declared as type `const string &`. This is a type mismatch, but the `string` class, much like the `String` class of Chapter 12, has a constructor with a `const char *` parameter, and that constructor is used automatically to create a `string` object initialized by the C-style string. In short, you can use either a `string` object or a C-style string as an argument to the `TableTennisPlayer` constructor. The first invokes a `string` constructor with a `const string &` parameter, and the second invokes a `string` constructor with a `const char *` parameter.

Deriving a Class

Some members of the Webtown Social Club have played in local table tennis tournaments, and they demand a class that includes the point ratings they've earned through their play. Rather than start from scratch, you can derive a class from the `TableTennisPlayer` class. The first step is to have the `RatedPlayer` class declaration show that it derives from the `TableTennisPlayer` class:

```
// RatedPlayer derives from the TableTennisPlayer base class
class RatedPlayer : public TableTennisPlayer
{
...
};
```

The colon indicates that the `RatedPlayer` class is based on the `TableTennisPlayer` class. This particular heading indicates that `TableTennisPlayer` is a public base class; this is termed *public derivation*. An object of a derived class incorporates a base class object. With public derivation, the public members of the base class become public members of the derived class. The private portions of a base class become part of the derived class, but they can be accessed only through public and protected methods of the base class. (We'll get to protected members in a bit.)

What does this accomplish? If you declare a `RatedPlayer` object, it has the following special properties:

- An object of the derived type has stored within it the data members of the base type. (The derived class inherits the base-class implementation.)
- An object of the derived type can use the methods of the base type. (The derived class inherits the base-class interface.)

Thus, a `RatedPlayer` object can store the first name and last name of each player and whether the player has a table. Also a `RatedPlayer` object can use the `Name()`, `HasTable()`, and `ResetTable()` methods from the `TableTennisPlayer` class (see Figure 13.1 for another example).

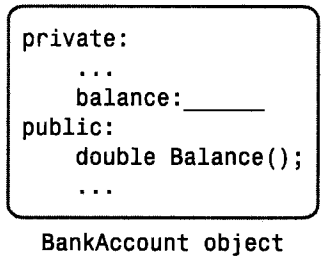
What needs to be added to these inherited features?

- A derived class needs its own constructors.
- A derived class can add additional data members and member functions as needed.

In this particular case, the class needs one more data member to hold the `ratings` value. It should also have a method for retrieving the rating and a method for resetting the rating. So the class declaration could look like this:

```
// simple derived class
class RatedPlayer : public TableTennisPlayer
{
private:
    unsigned int rating;     // add a data member
```

```
public:
    RatedPlayer (unsigned int r = 0, const string & fn = "none",
                 const string & ln = "none", bool ht = false);
    RatedPlayer(unsigned int r, const TableTennisPlayer & tp);
    unsigned int Rating() const { return rating; }   // add a method
    void ResetRating (unsigned int r) {rating = r;} // add a method
};
```




```
class Overdraft : public BankAccount {...};
```

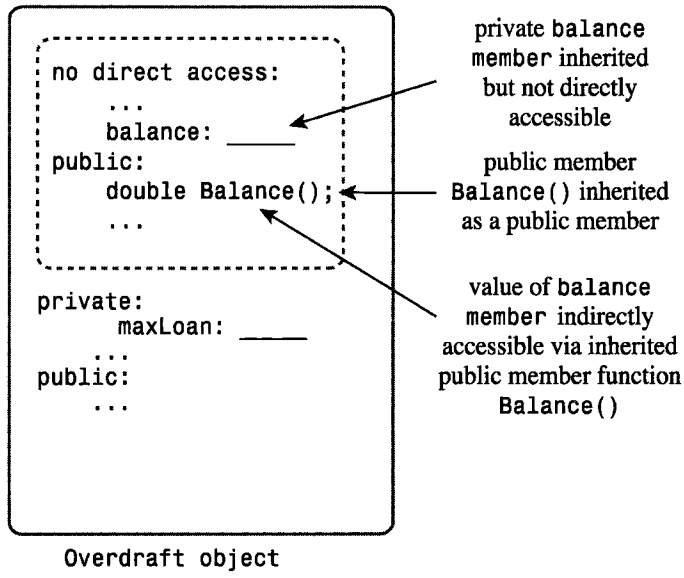


Figure 13.1 Base-class and derived-class objects.

The constructors have to provide data for the new members, if any, and for the inherited members. The first `RatedPlayer` constructor uses a separate formal parameter for each member, and the second `RatedPlayer` constructor uses a `TableTennisPlayer` parameter, which bundles three items (`firstname`, `lastname`, and `hasTable`) into a single unit.

Constructors: Access Considerations

A derived class does not have direct access to the private members of the base class; it has to work through the base-class methods. For example, the `RatedPlayer` constructors cannot directly set the inherited members (`firstname`, `lastname`, and `hasTable`). Instead, they have to use public base-class methods to access private base-class members. In particular, the derived-class constructors have to use the base-class constructors.

When a program constructs a derived-class object, it first constructs the base-class object. Conceptually, that means the base-class object should be constructed before the program enters the body of the derived-class constructor. C++ uses the member initializer list syntax to accomplish this. Here, for instance, is the code for the first `RatedPlayer` constructor:

```
RatedPlayer::RatedPlayer(unsigned int r, const string & fn,
     const string & ln, bool ht) : TableTennisPlayer(fn, ln, ht)
{
    rating = r;
}
```

The following part is the member initializer list:

```
: TableTennisPlayer(fn, ln, ht)
```

It's executable code, and it calls the `TableTennisPlayer` constructor. Suppose, for example, a program has the following declaration:

```
RatedPlayer rplayer1(1140, "Mallory", "Duck", true);
```

The `RatedPlayer` constructor assigns the actual arguments `"Mallory"`, `"Duck"`, and `true` to the formal parameters `fn`, `ln`, and `ht`. It then passes these parameters on as actual arguments to the `TableTennisPlayer` constructor. This constructor, in turn, creates the embedded `TableTennisPlayer` object and stores the data `"Mallory"`, `"Duck"`, and `true` in it. Then the program enters the body of the `RatedPlayer` constructor, completes the construction of the `RatedPlayer` object, and assigns the value of the parameter `r` (that is, `1140`) to the `rating` member (see Figure 13.2 for another example).

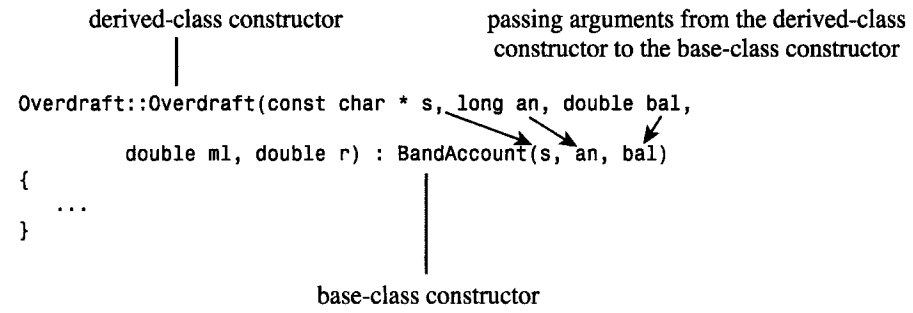


Figure 13.2 Passing arguments through to a base-class constructor.

What if you omit the member initializer list?

```
RatedPlayer::RatedPlayer(unsigned int r, const string & fn,
     const string & ln, bool ht) // what if no initializer list?
{
    rating = r;
}
```

The base-class object must be created first, so if you omit calling a base-class constructor, the program uses the default base-class constructor. Therefore, the previous code is the same as the following:

```
RatedPlayer::RatedPlayer(unsigned int r, const string & fn,
     const string & ln, bool ht) // : TableTennisPlayer()
{
    rating = r;
}
```

Unless you want the default constructor to be used, you should explicitly provide the correct base-class constructor call.

Now let's look at code for the second constructor:

```
RatedPlayer::RatedPlayer(unsigned int r, const TableTennisPlayer & tp)
    : TableTennisPlayer(tp)
{
    rating = r;
}
```

Again, the `TableTennisPlayer` information is passed on to a `TableTennisPlayer` constructor:

```
TableTennisPlayer(tp)
```

Because `tp` is type `const TableTennisPlayer &`, this call invokes the base-class copy constructor. The base class didn't define a copy constructor, but recall from Chapter 12 that the compiler automatically generates a copy constructor if one is needed and you haven't defined one already. In this case, the implicit copy constructor, which does memberwise copying, is fine because the class doesn't directly use dynamic memory allocation. (The `string` members do use dynamic memory allocation, but, recall, memberwise copying will use the `string` class copy constructors to copy the `string` members.)

You may, if you like, also use member initializer list syntax for members of the derived class. In this case, you use the member name instead of the class name in the list. Thus, the second constructor can also be written in this manner:

```
// alternative version
RatedPlayer::RatedPlayer(unsigned int r, const TableTennisPlayer & tp)
    : TableTennisPlayer(tp), rating(r)
{
}
```

These are the key points about constructors for derived classes:

- The base-class object is constructed first.
- The derived-class constructor should pass base-class information to a base-class constructor via a member initializer list.
- The derived-class constructor should initialize the data members that were added to the derived class.

This example doesn't provide explicit destructors, so the implicit destructors are used. Destroying an object occurs in the opposite order used to construct an object. That is, the body of the derived-class destructor is executed first, and then the base-class destructor is called automatically.

Note

When creating an object of a derived class, a program first calls the base-class constructor and then calls the derived-class constructor. The base-class constructor is responsible for initializing the inherited data members. The derived-class constructor is responsible for initializing any added data members. A derived-class constructor always calls a base-class constructor. You can use the initializer list syntax to indicate *which* base-class constructor to use. Otherwise, the default base-class constructor is used.

When an object of a derived class expires, the program first calls the derived-class destructor and then calls the base-class destructor.

Member Initializer Lists

A constructor for a derived class can use the initializer list mechanism to pass values along to a base-class constructor. Consider this example:

```
derived::derived(type1 x, type2 y) : base(x,y) // initializer list
{
    ...
}
```

Here, `derived` is the derived class, `base` is the base class, and `x` and `y` are variables used by the base-class constructor. If, say, the derived-class constructor receives the arguments `10` and `12`, this mechanism then passes `10` and `12` on to the base-class constructor defined as taking arguments of these types. Except for the case of virtual base classes (see Chapter 14, "Reusing Code in C++"), a class can pass values back only to its immediate base class. However, that class can use the same mechanism to pass back information to its immediate base class, and so on. If you don't supply a base-class constructor in a member initializer list, the program uses the default base-class constructor. The member initializer list can be used *only* in constructors.

Using a Derived Class

To use a derived class, a program needs access to the base-class declarations. Listing 13.4 places both class declarations in the same header file. You could give each class its own header file, but because the two classes are related, it makes more organizational sense to keep the class declarations together.

Listing 13.4 **tabtenn1.h**

```
// tabtenn1.h -- a table-tennis base class
#ifndef TABTENN1_H_
#define TABTENN1_H_
#include <string>
using std::string;
// simple base class
class TableTennisPlayer
{
private:
    string firstname;
    string lastname;
    bool hasTable;
public:
    TableTennisPlayer (const string & fn = "none",
                       const string & ln = "none", bool ht = false);
    void Name() const;
    bool HasTable() const { return hasTable; };
    void ResetTable(bool v) { hasTable = v; };
};

// simple derived class
class RatedPlayer : public TableTennisPlayer
{
private:
    unsigned int rating;
public:
    RatedPlayer (unsigned int r = 0, const string & fn = "none",
                 const string & ln = "none", bool ht = false);
    RatedPlayer(unsigned int r, const TableTennisPlayer & tp);
    unsigned int Rating() const { return rating; }
    void ResetRating (unsigned int r) {rating = r;}
};

#endif
```

Listing 13.5 provides the method definitions for both classes. Again, you could use separate files, but it's simpler to keep the definitions together.

Listing 13.5 **tabtenn1.cpp**

```
//tabtenn1.cpp -- simple base-class methods
#include "tabtenn1.h"
#include <iostream>

TableTennisPlayer::TableTennisPlayer (const string & fn,
    const string & ln, bool ht) : firstname(fn),
        lastname(ln), hasTable(ht) {}

void TableTennisPlayer::Name() const
{
    std::cout << lastname << ", " << firstname;
}

// RatedPlayer methods
RatedPlayer::RatedPlayer(unsigned int r, const string & fn,
     const string & ln, bool ht) : TableTennisPlayer(fn, ln, ht)
{
    rating = r;
}

RatedPlayer::RatedPlayer(unsigned int r, const TableTennisPlayer & tp)
    : TableTennisPlayer(tp), rating(r)
{
}
```

Listing 13.6 creates objects of both the `TableTennisPlayer` class and the `RatedPlayer` class. Notice that objects of both classes can use the `TableTennisPlayer` class `Name()` and `HasTable()` methods.

Listing 13.6 **usett1.cpp**

```
// usett1.cpp -- using base class and derived class
#include <iostream>
#include "tabtenn1.h"

int main ( void )
{
    using std::cout;
    using std::endl;
    TableTennisPlayer player1("Tara", "Boomdea", false);
    RatedPlayer rplayer1(1140, "Mallory", "Duck", true);
    rplayer1.Name();          // derived object uses base method
    if (rplayer1.HasTable())
        cout << ": has a table.\n";
    else
```

```
        cout << ": hasn't a table.\n";
    player1.Name();          // base object uses base method
    if (player1.HasTable())
        cout << ": has a table";
    else
        cout << ": hasn't a table.\n";
    cout << "Name: ";
    rplayer1.Name();
    cout << "; Rating: " << rplayer1.Rating() << endl;
// initialize RatedPlayer using TableTennisPlayer object
    RatedPlayer rplayer2(1212, player1);
    cout << "Name: ";
    rplayer2.Name();
    cout << "; Rating: " << rplayer2.Rating() << endl;

    return 0;
}
```

Here is the output of the program in Listings 13.4, 13.5, and 13.6:

```
Duck, Mallory: has a table.
Boomdea, Tara: hasn't a table.
Name: Duck, Mallory; Rating: 1140
Name: Boomdea, Tara; Rating: 1212
```

Special Relationships Between Derived and Base Classes

A derived class has some special relationships with the base class. One, which you've just seen, is that a derived-class object can use base-class methods, provided that the methods are not private:

```
RatedPlayer rplayer1(1140, "Mallory", "Duck", true);
rplayer1.Name();  // derived object uses base method
```

Two other important relationships are that a base-class pointer can point to a derived-class object without an explicit type cast and that a base-class reference can refer to a derived-class object without an explicit type cast:

```
RatedPlayer rplayer1(1140, "Mallory", "Duck", true);
TableTennisPlayer & rt = rplayer;
TableTennisPlayer * pt = &rplayer;
rt.Name();   // invoke Name() with reference
pt->Name();  // invoke Name() with pointer
```

However, a base-class pointer or reference can invoke just base-class methods, so you couldn't use `rt` or `pt` to invoke, say, the derived-class `ResetRanking()` method.

Ordinarily, C++ requires that references and pointer types match the assigned types, but this rule is relaxed for inheritance. However, the rule relaxation is just in one direction. You can't assign base-class objects and addresses to derived-class references and pointers:

```
TableTennisPlayer player("Betsy", "Bloop", true);
RatedPlayer & rr = player;      // NOT ALLOWED
RatedPlayer * pr = player;      // NOT ALLOWED
```

Both these sets of rules make sense. For example, consider the implications of having a base-class reference refer to a derived object. In this case, you can use the base-class reference to invoke base-class methods for the derived-class object. Because the derived class inherits the base-class methods and data members, this causes no problems. Now consider what would happen if you could assign a base-class object to a derived-class reference. The derived-class reference would be able to invoke derived-class methods for the base object, and that could cause problems. For example, applying the `RatedPlayer::Rating()` method to a `TableTennisPlayer` object makes no sense because the `TableTennisPlayer` object doesn't have a `rating` member.

The fact that base-class references and pointers can refer to derived-class objects has some interesting consequences. One is that functions defined with base-class reference or pointer arguments can be used with either base-class or derived-class objects. For instance, consider this function:

```
void Show(const TableTennisPlayer & rt)
{
    using std::cout;
    cout << "Name: ";
    rt.Name();
    cout << "\nTable: ";
    if (rt.HasTable())
        cout << "yes\n";
    else
        cout << "no\n";
}
```

The formal parameter `rt` is a reference to a base class, so it can refer to a base-class object or to a derived-class object. Thus, you can use `Show()` with either a `TableTennis` argument or a `RatedPlayer` argument:

```
TableTennisPlayer player1("Tara", "Boomdea", false);
RatedPlayer rplayer1(1140, "Mallory", "Duck", true);
Show(player1);   // works with TableTennisPlayer argument
Show(rplayer1);  // works with RatedPlayer argument
```

A similar relationship would hold for a function with a pointer-to-base-class formal parameter; it could be used with either the address of a base-class object or the address of a derived-class object as an actual argument: