第 12 章　实现继承

前一章介绍了一些面向对象关系，其中包括具体化/泛化。C++通过继承来实现这种关系。

本章介绍以下内容：

- 什么是继承？
- 如何使用继承从一个类派生出另一个类？
- 什么是保护访问权限（protected access）及如何使用它？
- 什么是虚方法？

12.1　什么是继承

什么是狗？观察宠物时，你看到了什么？我看到 4 条腿和一张嘴。生物学家看到的是相互作用的器官网，物理学家看到的是原子和力，而分类学家看到的是犬类动物的代表。

现在令人感兴趣的是最后一种看法，狗是犬类的一种，而犬类动物是哺乳动物的一种，依次类推。分类学家将动物分成门、纲、目、科、属、种。

这种泛化/具体化层次结构建立了一种 is-a 关系。人是一种灵长目动物。这种关系无处不在：客货两用车是一种汽车，而汽车是一种交通工具。圣代冰激凌是一种甜食，而后者是一种食品。

当我们说一种东西是另外一种东西时，意味着它是后者的特例。也就是说，汽车是一种特殊的交通工具。

12.1.1　继承和派生

狗继承（即自动获得）了哺乳动物的所有特点。狗是哺乳动物，因此我们知道它能够运动和呼吸。根据定义，所有哺乳动物都能够运动和呼吸。狗在哺乳动物的基础上增加了吠和摇尾巴等功能。

可以将狗分为工作犬、运动犬和猎狐梗。将运动犬分为拾猎犬、猎犬等。每种这样的狗还可进一步细分，例如，拾猎犬可细分为拉布拉多猎犬和加利福尼亚猎犬。

加利福尼亚猎犬是一种拾猎犬，后者是一种运动犬，运动犬是一种狗，狗是一种哺乳动物，哺乳动物是一种动物，动物是一种生物。图 12.1 说明了这种层次结构。

C++允许你定义从另一个类派生出另一个类来表示这种关系。派生是一种表示 is-a 关系的方式。你从类 Mammal 派生出新类 Dog。由于 Dog 类从 Mammal 类继承了运动功能，因此你不必显式说明狗能够运动。

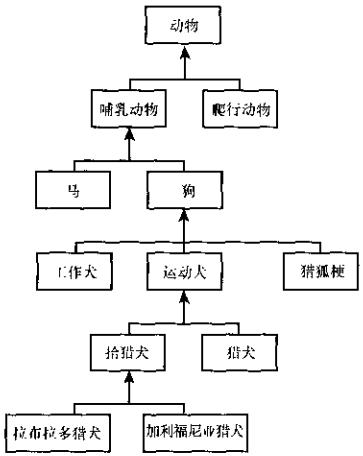

图 12.1　动物层次结构

在已有类的基础上添加了新功能的类被称为从原来的类派生而来。原来的类被称为新类的基类。

如果 Dog 类从 Mammal 类派生而来，则 Mammal 类是 Dog 的基类。派生类是其基类的超集。狗在哺乳动物的基础上增加一些新特征，而 Dog 类在 Mammal 类的基础上增加了一些方法或数据。

通常，基类有多个派生类。由于狗、猫和马都是哺乳动物，因此它们对应的类都是从 Mammal 类派生而来的。

12.1.2　动物世界

为便于讨论派生和继承，本章将重点放在一些表示动物的类之间的关系上。假设有人请你设计一款孩子玩的游戏：模拟农场。

此时你将开发一组农场动物，包括马、奶牛、狗、猫和绵羊等。你将为这些类创建一些方法，以便它们能按孩子期望的方式行动，但就现在而言，将每种方法简化为只包含一条打印语句。

简化函数意味着只需编写能显示函数被调用的代码即可，将细节留待以后有时间去完成。读者可扩展本章提供的哺乳动物范例代码，让这些动物更为逼真。

读者将发现，使用动物的范例很容易理解，也很容易将这些概念用于其他领域。例如，编写 ATM 程序时，可能需要涉及到支票账户，它是一种银行账户，而银行账户是一种账户。这类似于狗是哺乳动物，而哺乳动物是一种动物。

12.1.3　派生的语法

声明类时，可在类名后加上冒号（:）、派生类型（公有或其他）和基类名来指出它是从哪个类派生而来的，格式如下：

```
class derivedClass : accessType baseClass
```

例如，要创建一个名为 Dog 的从 Mammal 类派生而来的新类，可以这样做：

```
class Dog : public Mammal
```

派生类型（accessType）将在本章后面讨论。就现在而言，总是使用 public。从中派生的类必须已经声明，否则将出现编译错误。程序清单 12.1 演示了如何声明从 Mammal 类派生而来的 Dog 类。

程序清单 12.1　简单继承

```
 1: //Listing 12.1 Simple inheritance
 2: #include <iostream>
 3: using namespace std;
 4:
 5: enum BREED { GOLDEN, CAIRN, DANDIE, SHETLAND, DOBERMAN, LAB };
 6:
 7: class Mammal
 8: {
 9:   public:
10:     // constructors
11:     Mammal();
12:     ~Mammal();
13:
14:     //accessors
15:     int GetAge()    const;
16:     void SetAge(int);
17:     int GetWeight() const;
18:     void SetWeight();
19:
20:     //Other methods
21:     void Speak() const;
```

```
22:    void Sleep() const;
23:
24:
25:  protected:
26:    int itsAge;
27:    int itsWeight;
28: };
29:
30: class Dog : public Mammal
31: {
32:  public:
33:
34:    // Constructors
35:    Dog();
36:    ~Dog();
37:
38:    // Accessors
39:    BREED GetBreed() const;
40:    void SetBreed(BREED);
41:
42:    // Other methods
43:    WagTail();
44:    BegForFood();
45:
46:  protected:
47:    BREED itsBreed;
48: };
```

该程序没有输出，因为其中只有一组类声明，没有类的实现。虽然如此，还是有很多需要介绍的内容。

分析：

第7~28行声明了Mammal类。注意，在这个例子中，Mammal类没有继承任何类。但在现实生活中，哺乳动物确实是派生而来的，即哺乳动物是动物的一种。在C++程序中，只能表示有关物体的部分信息；实际情况非常复杂，不可能表示全部信息，因此每个C++层次结构都知道可用数据的有限表示。提供良好设计的技巧是，以相当忠实于实际情况的方式表示你关心的部分，而不增加不必要的复杂性。

层次结构必须从某个地方开始，这个程序从Mammal开始。由于这种决定，这个类中包含一些原本应该放在更高级基类中的成员变量。例如，所有动物都有年龄和体重，如果Mammal是从Animal派生而来的，将可以继承这些属性；然而，这些属性却出现在Mammal类中。

以后，如果添加了另一种也有这些属性的动物（如Insect），可以将这些属性放到Animal类中，并将Animal作为Mammal和Insect的基类。这就是类层次结构是如何随时间推移而演变的。

为确保程序简单和易于管理，Mammal类只包含6个方法：4个存取器方法以及Speak()和Sleep()。

如第30行所示，Dog类是自Mammal类派生而来的，这是因为类名（Dog）后面有冒号和基类名（Mammal）。

每个Dog对象将有3个成员变量：itsAge、itsWeight和itsBreed。注意，Dog的类声明中并没有成员变量itsAge和itsWeight，Dog对象从Mammal类继承了这些变量以及除复制构造函数、构造函数和析构函数外的所有方法。

12.2 私有和保护

读者可能注意到，程序清单12.1的第25和46行使用了一个新的关键字protected。以前，类数据都被声

明为私有的。然而，私有成员在当前类之外是不能直接访问的，在派生类中也如此。可以将变量 itsAge 和 itsWeight 声明为公有的，但这并不理想，因为你不希望其他类能够直接访问这些数据成员。

注意：有一种观点认为，应将所有成员数据声明为私有，而不是保护的；这是 C++创始人 Stroustrup 在其 *The Design and Evolution of C++*（ISBN 0-2-1-543330-3，Addison Wesley，1994）一书中指出的。受保护的方法通常不被认为是有问题的，而且可能很有用。

你想要的派生类型是：使这些数据对当前类及其派生类来说是可见的。这种派生类型为保护（protected）。保护型数据成员和函数对派生类来说是完全可见的，但对其他类来说是私有的。

总共有 3 种访问限定符：公有（public）、保护（protected）和私有（private）。如果在函数中声明了一个对象，则它可以访问所有的公有成员数据和函数。而成员函数可以访问其所属类的所有私有数据成员和函数，也可以访问任何从其所属类派生而来的类的保护数据成员和函数。

因此，函数 Dog::WagTail( )可以访问私有数据 itsBreed 以及 Mammal 类的保护数据 itsAge 和 itsWeight。

即使继承层次结构中，Mammal 类和 Dog 类之间有其他类（如 DomesticAnimals），Dog 类仍能够访问 Mammal 类的保护成员，条件是这些其他类都采用公有继承。私有继承将在第 16 章讨论。

程序清单 12.2 演示了如何创建 Dog 对象以及如何访问其数据和函数。

程序清单 12.2　使用派生对象

```
1:  //Listing 12.2 Using a derived object
2:  #include <iostream>
3:  using std::cout;
4:  using std::endl;
5:
6:  enum BREED { GOLDEN, CAIRN, DANDIE, SHETLAND, DOBERMAN, LAB };
7:
8:  class Mammal
9:  {
10:   public:
11:     // constructors
12:     Mammal():itsAge(2), itsWeight(5){}
13:     ~Mammal(){}
14:
15:     //accessors
16:     int GetAge() const   { return itsAge; }
17:     void SetAge(int age) { itsAge = age; }
18:     int GetWeight() const { return itsWeight; }
19:     void SetWeight(int weight) { itsWeight = weight; }
20:
21:     //Other methods
22:     void Speak()const { cout << "Mammal sound!\n"; }
23:     void Sleep()const { cout << "shhh. I'm sleeping.\n";}
24:
25:   protected:
26:     int itsAge;
27:     int itsWeight;
28: };
29:
30: class Dog : public Mammal
31: {
32:   public:
```

```
33:
34:     // Constructors
35:     Dog():itsBreed(GOLDEN){}
36:     ~Dog(){}
37:
38:     // Accessors
39:     BREED GetBreed() const { return itsBreed; }
40:     void SetBreed(BREED breed) { itsBreed = breed; }
41:
42:     // Other methods
43:     void WagTail() const { cout << "Tail wagging...\n"; }
44:     void BegForFood() const { cout << "Begging for food...\n"; }
45:
46:   private:
47:     BREED itsBreed;
48: };
49:
50: int main()
51: {
52:   Dog Fido;
53:   Fido.Speak();
54:   Fido.WagTail();
55:   cout << "Fido is " << Fido.GetAge() << " years old" << endl;
56:   return 0;
57: }
```

输出：

```
Mammal sound!
Tail wagging...
Fido is 2 years old
```

分析：

第 8～28 行声明了 Mammal 类（为节省篇幅，其所有函数都是内联的）。第 30～48 行将 Dog 类声明为 Mammal 的派生类。通过这些声明，所有 Dog 对象都有年龄、体重和品种。正如前面指出的，年龄和体重是从基类 Mammal 那里继承而来的。

第 52 行声明了一个 Dog 对象：Fido。Fido 继承了 Mammal 的所有属性，同时具有 Dog 的所有属性。因此，Fido 不但知道如何 WagTail()，还知道如何 Speak()和 Sleep()。第 55 行成功地调用了基类中的存取器方法 GetAge()。

12.3 构造函数和析构函数的继承性

Dog 对象是 Mammal 对象，这是 is-a 关系的本质。

创建 Fido 时，首先调用基类的构造函数，创建一个 Mammal 对象；然后调用 Dog 的构造函数，完成 Dog 对象的创建。由于创建 Fido 时没有提供参数，因此将调用 Mammal 和 Dog 的默认构造函数。在 Fido 创建好之前，它是不存在的，这就意味着必须创建其 Mammal 部分和 Dog 部分。因此，必须调用这两个类的构造函数。

Fido 被销毁时，首先调用 Dog 的析构函数，然后为 Fido 的 Mammal 部分调用析构函数。每个析构函数都提供了在其对应部分被销毁后执行清理工作的机会。别忘了，在 Dog 对象被销毁后进行清理！程序清单 12.3

演示了对构造函数和析构函数的调用。

程序清单 12.3　调用构造函数和析构函数

```
1:  //Listing 12.3 Constructors and destructors called.
2:  #include <iostream>
3:  using namespace std;
4:  enum BREED { GOLDEN, CAIRN, DANDIE, SHETLAND, DOBERMAN, LAB };
5:
6:  class Mammal
7:  {
8:    public:
9:      // constructors
10:     Mammal();
11:     ~Mammal();
12:
13:     //accessors
14:     int GetAge() const { return itsAge; }
15:     void SetAge(int age) { itsAge = age; }
16:     int GetWeight() const { return itsWeight; }
17:     void SetWeight(int weight) { itsWeight = weight; }
18:
19:     //Other methods
20:     void Speak() const { cout << "Mammal sound!\n"; }
21:     void Sleep() const { cout << "shhh. I'm sleeping.\n";}
22:
23:   protected:
24:     int itsAge;
25:     int itsWeight;
26: };
27:
28: class Dog : public Mammal
29: {
30:   public:
31:
32:     // Constructors
33:     Dog();
34:     ~Dog();
35:
36:     // Accessors
37:     BREED GetBreed() const { return itsBreed; }
38:     void SetBreed(BREED breed) { itsBreed = breed; }
39:
40:     // Other methods
41:     void WagTail() const { cout << "Tail wagging...\n"; }
42:     void BegForFood() const { cout << "Begging for food...\n"; }
43:
44:   private:
45:     BREED itsBreed;
46: };
47:
48: Mammal::Mammal():
49: itsAge(3),
```

```
50: itsWeight(5)
51: {
52:    std::cout << "Mammal constructor... " << endl;
53: }
54:
55: Mammal::~Mammal()
56: {
57:    std::cout << "Mammal destructor... " << endl;
58: }
59:
60: Dog::Dog():
61: itsBreed(GOLDEN)
62: {
63:    std::cout << "Dog constructor... " << endl;
64: }
65:
66: Dog::~Dog()
67: {
68:    std::cout << "Dog destructor... " << endl;
69: }
70: int main()
71: {
72:    Dog Fido;
73:    Fido.Speak();
74:    Fido.WagTail();
75:    std::cout << "Fido is " << Fido.GetAge() << " years old" << endl;
76:    return 0;
77: }
```

输出：

```
Mammal constructor...
Dog constructor...
Mammal sound!
Tail wagging...
Fido is 3 years old
Dog destructor...
Mammal destructor...
```

分析：

除第48～69行的构造函数和析构函数在被调用时将信息打印到屏幕上外，程序清单12.3和程序清单12.2相同。首先调用Mammal的构造函数，然后调用Dog的构造函数。至此，Dog对象被创建好，才可以调用其方法。

当Fido不再在作用域中时，Dog的析构函数首先被调用，然后Mammal的析构函数被调用。

向基类的构造函数传递参数

你可能想在基类构造函数中初始化某些值。例如，你可能想重载Mammal的构造函数，使其接受年龄作为参数，并重载Dog的构造函数，使其接受品种作为参数。如何确保年龄和体重参数被传递给正确的Mammal构造函数呢？如果Dog想初始化体重而Mammal不想该如何办呢？

通过在派生类的构造函数名后加上冒号、基类名和基类构造函数需要的参数，可以在初始化期间对基类进行初始化。程序清单12.4说明了这一点。

程序清单 12.4　重载派生类的构造函数

```
1:  //Listing 12.4 Overloading constructors in derived classes
2:  #include <iostream>
3:  using namespace std;
4:
5:  enum BREED { GOLDEN, CAIRN, DANDIE, SHETLAND, DOBERMAN, LAB };
6:
7:  class Mammal
8:  {
9:    public:
10:     // constructors
11:     Mammal();
12:     Mammal(int age);
13:     ~Mammal();
14:
15:     //accessors
16:     int GetAge() const { return itsAge; }
17:     void SetAge(int age) { itsAge = age; }
18:     int GetWeight() const { return itsWeight; }
19:     void SetWeight(int weight) { itsWeight = weight; }
20:
21:     //Other methods
22:     void Speak() const { cout << "Mammal sound!\n"; }
23:     void Sleep() const { cout << "shhh. I'm sleeping.\n"; }
24:
25:
26:   protected:
27:     int itsAge;
28:     int itsWeight;
29: };
30:
31: class Dog : public Mammal
32: {
33:   public:
34:
35:     // Constructors
36:     Dog();
37:     Dog(int age);
38:     Dog(int age, int weight);
39:     Dog(int age, BREED breed);
40:     Dog(int age, int weight, BREED breed);
41:     ~Dog();
42:
43:     // Accessors
44:     BREED GetBreed() const { return itsBreed; }
45:     void SetBreed(BREED breed) { itsBreed = breed; }
46:
47:     // Other methods
48:     void WagTail() const { cout << "Tail wagging...\n"; }
49:     void BegForFood() const { cout << "Begging for food...\n"; }
50:
```

```
51:   private:
52:     BREED itsBreed;
53: };
54:
55: Mammal::Mammal():
56: itsAge(1),
57: itsWeight(5)
58: {
59:    cout << "Mammal constructor..." << endl;
60: }
61:
62: Mammal::Mammal(int age):
63: itsAge(age),
64: itsWeight(5)
65: {
66:    cout << "Mammal(int) constructor..." << endl;
67: }
68:
69: Mammal::~Mammal()
70: {
71:    cout << "Mammal destructor..." << endl;
72: }
73:
74: Dog::Dog():
75: Mammal(),
76: itsBreed(GOLDEN)
77: {
78:    cout << "Dog constructor..." << endl;
79: }
80:
81: Dog::Dog(int age):
82: Mammal(age),
83: itsBreed(GOLDEN)
84: {
85:    cout << "Dog(int) constructor..." << endl;
86: }
87:
88: Dog::Dog(int age, int weight):
89: Mammal(age),
90: itsBreed(GOLDEN)
91: {
92:    itsWeight = weight;
93:    cout << "Dog(int, int) constructor..." << endl;
94: }
95:
96: Dog::Dog(int age, int weight, BREED breed):
97: Mammal(age),
98: itsBreed(breed)
99: {
100:    itsWeight = weight;
101:    cout << "Dog(int, int, BREED) constructor..." << endl;
102: }
```

```
103:
104: Dog::Dog(int age, BREED breed):
105: Mammal(age),
106: itsBreed(breed)
107: {
108:    cout << "Dog(int, BREED) constructor..." << endl;
109: }
110:
111: Dog::~Dog()
112: {
113:    cout << "Dog destructor..." << endl;
114: }
115: int main()
116: {
117:    Dog Fido;
118:    Dog rover(5);
119:    Dog buster(6,8);
120:    Dog yorkie (3,GOLDEN);
121:    Dog dobbie (4,20,DOBERMAN);
122:    Fido.Speak();
123:    rover.WagTail();
124:    cout << "Yorkie is " << yorkie.GetAge()
125:         << " years old" << endl;
126:    cout << "Dobbie weighs ";
127:    cout << dobbie.GetWeight() << " pounds" << endl;
128:    return 0;
129: }
```

注意：为便于分析时引用，给输出加上了行号。

输出：

```
 1: Mammal constructor...
 2: Dog constructor...
 3: Mammal (int) constructor...
 4: Dog (int) constructor...
 5: Mammal (int ) constructor...
 6: Dog (int, int ) constructor...
 7: Mammal (int) constructor...
 8: Dog(int, BREED) constructor...
 9: Mammal (int) constructor...
10: Dog(int, int, BREED) constructor...
11: Mammal sound!
12: Tail wagging...
13: Yorkie is 3 years old.
14: Dobbie weighs 20 pounds.
15: Dog destructor...
16: Mammal destructor...
17: Dog destructor...
18: Mammal destructor...
19: Dog destructor...
20: Mammal destructor...
21: Dog destructor...
```

```
22: Mammal destructor...
23: Dog destructor...
24: Mammal destructor...
```

分析：

在程序清单 12.4 中，第 12 行重载了 Mammal 的构造函数，使之接受一个 int 参数（Mammal 的年龄）。第 62～67 行为该构造函数的实现，它用传递给构造函数的值来初始化 itsAge 进行初始化，并将 itsWeight 初始化为 5。

第 36～40 行重载了 5 个 Dog 构造函数。第一个是默认构造函数。在第 37 行，第二个接受年龄作为参数，与 Mammal 的构造函数接受的参数相同。第三个构造函数接受年龄和体重作为参数，第四个构造函数接受年龄和品种作为参数，而第五个接受年龄、体重和品种作为参数。

从第 74 行开始，是 Dog 的默认构造函数的代码，其中有些新内容。从第 75 行可知，该构造函数被调用时，它将调用 Mammal 的默认构造函数。虽然并不是非得这样做，但它表明要调用不接受任何参数的基类构造函数。在任何情况下，都将调用基类构造函数，但通过显式地调用，可使你的意图更明确。

接受一个 int 参数的 Dog 构造函数的实现位于第 81～86 行。在其初始化阶段（第 82～83 行），Dog 使用传入的参数初始化基类，然后初始化品种。

另一个 Dog 构造函数位于第 88～94 行，它接受两个参数。同样，它在 89 行通过调用合适的构造函数来初始化基类，但这次也将体重参数赋给基类的 itsWeight 变量。注意，不能在初始化阶段给基类的 itsWeight 变量赋值，因为 Mammal 类没有接受这种参数的构造函数，因此必须在 Dog 的构造函数体中完成这项工作。

请读者分析其他的构造函数，确保自己理解了它们的工作原理。请注意哪些成员变量被初始化，哪些成员变量必须到构造函数体中进行赋值。

为方便在分析中引用，给输出加上了行号。前两行输出表明，Fido 是使用默认构造函数实例化的。

输出的第 3 和 4 行表示 rover 被创建；第 5 和 6 行表示 buster 被创建。注意，调用的是接受一个 int 参数的 Mammal 构造函数，但调用的 Dog 构造函数是接受两个 int 参数的构造函数。

所有对象被创建后，使用了这些对象，然后它们不再在作用域中。每个对象被销毁时，首先调用 Dog 的析构函数，然后调用 Mammal 的析构函数，每个析构函数都被调用了 5 次。

12.4 覆盖基类函数

Dog 对象可以访问 Mammal 类的所有成员函数，也可以访问 Dog 类新增的任何数据成员和函数，如 WagTail()。派生类还可以覆盖基类函数。覆盖基类函数意味着在派生类中修改其实现。

在派生类中创建一个返回值和特征标与基类成员函数相同但实现不同的函数时，被称为覆盖该函数。在这种情况下，创建一个派生类对象后，便可以调用正确的函数。

覆盖函数时，特征标必须与基类中被覆盖的函数相同。特征标指的是函数原型中除返回类型外的内容，即函数名、参数列表和可能用到的关键字 const。

程序清单 12.5 说明了在 Dog 类中覆盖 Mammal 的 Speak()方法后将发生的情况。为节省篇幅，删除了这些类的存取器函数。

程序清单 12.5　在派生类中覆盖基类方法

```
1: //Listing 12.5 Overriding a base class method in a derived class
2: #include <iostream>
3: using std::cout;
4:
5: enum BREED { GOLDEN, CAIRN, DANDIE, SHETLAND, DOBERMAN, LAB };
6:
7: class Mammal
```

```
 8: {
 9:   public:
10:     // constructors
11:     Mammal() { cout << "Mammal constructor...\n"; }
12:     ~Mammal() { cout << "Mammal destructor...\n"; }
13:
14:     //Other methods
15:     void Speak()const { cout << "Mammal sound!\n"; }
16:     void Sleep()const { cout << "shhh. I'm sleeping.\n"; }
17:
18:   protected:
19:     int itsAge;
20:     int itsWeight;
21: };
22:
23: class Dog : public Mammal
24: {
25:   public:
26:     // Constructors
27:     Dog(){ cout << "Dog constructor...\n"; }
28:     ~Dog(){ cout << "Dog destructor...\n"; }
29:
30:     // Other methods
31:     void WagTail() const  { cout << "Tail wagging...\n"; }
32:     void BegForFood() const  { cout << "Begging for food...\n"; }
33:     void Speak() const { cout << "Woof!\n"; }
34:
35:   private:
36:     BREED itsBreed;
37: };
38:
39: int main()
40: {
41:    Mammal bigAnimal;
42:    Dog Fido;
43:    bigAnimal.Speak();
44:    Fido.Speak();
45:    return 0;
46: }
```

输出：

```
Mammal constructor...
Mammal constructor...
Dog constructor...
Mammal sound!
Woof!
Dog destructor...
Mammal destructor...
Mammal destructor...
```

分析：

在 Mammal 类中，第 15 行定义了一个名为 Speak( )的方法。第 23～37 行声明了 Dog 类，它是从 Mammal

派生而来的（第 23 行），因此能够访问该 Speak( )方法。然而，在 33 行，Dog 类覆盖了 Speak( )方法，导致 Speak( )方法被调用时，Dog 对象打印“Woof!”。

在 main( )中，第 41 行创建了 Mammal 对象 bigAnimal，这导致 Mammal 的构造函数被调用，打印第一行输出。第 42 行创建了 Dog 对象 fido，这导致首先调用 Mammal 的构造函数，然后调用 Dog 的构造函数，它们打印接下来的两行输出。

第 43 行通过 Mammal 对象调用 Speak( )方法；然后第 44 行通过 Dog 对象调用 Speak( )方法。输出表明，调用了正确的方法：bigAnimal 发出哺乳动物叫声（打印“Mammal Sound!”），Fido 发出低叫声（打印“Woof!”）。最后，这两个对象不再在作用域中，因此析构函数被调用。

重载和覆盖

这两个术语类似，功能也相似。重载方法时，创建多个名称相同但特征标不同的多个方法；覆盖方法时，在派生类中创建一个名称和特征标与基类方法相同的方法。

12.4.1 隐藏基类方法

在前一个程序清单中，Dog 类的 Speak( )方法隐藏了基类的 Speak( )方法。这正是我们希望的，但可能有意想不到的结果。如果 Mammal 有一个被重载的方法 Move( )，而 Dog 覆盖了该方法，则 Dog 的 Move( )方法将隐藏 Mammal 中所有名称为 Move 的方法。

如果 Mammal 重载了三个 Move( )方法：第一个不接受任何参数，第二个接受一个 int 参数，第三个接受一个 int 参数和一个方向参数，且 Dog 只覆盖了不接受任何参数的 Move( )方法，将难以通过 Dog 对象访问其他两个 Move( )方法。程序清单 12.6 说明了这种问题。

程序清单 12.6 隐藏方法

```
1:  //Listing 12.6 Hiding methods
2:  #include <iostream>
3:  using std::cout;
4:
5:  class Mammal
6:  {
7:    public:
8:      void Move() const { cout << "Mammal move one step.\n"; }
9:      void Move(int distance) const
10:     {
11:       cout << "Mammal move ";
12:       cout << distance <<" steps.\n";
13:     }
14:   protected:
15:     int itsAge;
16:     int itsWeight;
17: };
18:
19: class Dog : public Mammal
20: {
21:   public:
22:     // You might receive a warning that you are hiding a function!
23:     void Move() const { cout << "Dog move 5 steps.\n"; }
24: };
25:
26: int main()
```

```
27: {
28:    Mammal bigAnimal;
29:    Dog Fido;
30:    bigAnimal.Move();
31:    bigAnimal.Move(2);
32:    Fido.Move();
33:    // Fido.Move(10);
34:    return 0;
35: }
```

输出：

```
Mammal move one step.
Mammal move 2 steps.
Dog move 5 steps.
```

分析：

所有多余的方法和数据都从这些类中删除了。在第 8 行和第 9 行，Mammal 类声明了重载的 Move( )方法。在第 23 行，Dog 覆盖了接受任何参数的 Move( )方法。第 30～32 行调用了这些方法，程序运行时输出反映了这一点。

然而，第 33 行将导致编译错误，因此被注释掉了。覆盖基类的某个方法后，就不能通过派生类对象使用任何同名的基类方法。如果 Dog 类没有覆盖不接受任何参数的 Move( )方法，它将可以调用 Move(int)方法，但既然覆盖了，如果要使用这两个版本的 move( )方法，必须将它们都覆盖。否则将隐藏没有覆盖的方法。与该规则类似的情形是，如果你提供了任何构造函数，编译器将不会提供默认构造函数。

规则是这样的：覆盖任一个重载方法后，该方法的其他所有版本都将被隐藏；如果不希望它们被隐藏，必须对其进行覆盖。

一种常见的错误是，在本想覆盖一个基类方法时，由于忘记包含关键字 const，而将其隐藏了。const 是特征标的一部分，省略它将改变特征标，导致方法被隐藏而不是被覆盖。

覆盖和隐藏

下一节将讨论虚方法。通过覆盖虚方法可支持多态，但隐藏虚方法将破坏多态。稍后将更详细地介绍这一点。

12.4.2　调用基类方法

覆盖基类方法后，仍可以通过限定方法名来调用它——在方法名前加上基类名和两个冒号：

```
baseClass::Method()
```

要调用 Mammal 类的 Move( )方法，可以这样做：

```
Mammal::Move().
```

可以像使用其他方法名一样使用这些被限定的方法名。可以像下面这样修改清单 12.6 中第 33 行，使其能够通过编译：

```
Fido.Mammal::Move(10);
```

这显式地调用 Mammal 的方法。程序清单 12.7 演示了这种概念。

程序清单 12.7　调用被覆盖的基类方法

```
1: //Listing 12.7 Calling a base method from a overridden method.
2: #include <iostream>
3: using namespace std;
4:
```

```
 5: class Mammal
 6: {
 7:   public:
 8:     void Move() const { cout << "Mammal move one step\n"; }
 9:     void Move(int distance) const
10:     {
11:       cout << "Mammal move " << distance;
12:       cout << " steps." << endl;
13:     }
14:
15:   protected:
16:     int itsAge;
17:     int itsWeight;
18: };
19:
20: class Dog : public Mammal
21: {
22:   public:
23:     void Move() const;
24: };
25:
26: void Dog::Move() const
27: {
28:   cout << "In dog move...\n";
29:   Mammal::Move(3);
30: }
31:
32: int main()
33: {
34:   Mammal bigAnimal;
35:   Dog Fido;
36:   bigAnimal.Move(2);
37:   Fido.Mammal::Move(6);
38:   return 0;
39: }
```

输出：

```
Mammal move 2 steps.
Mammal move 6 steps.
```

分析：

第 34 行创建了 Mammal 对象 bigAnimal，第 35 行创建了 Dog 对象 Fido。第 36 行 Mammal 类中接受一个参数的 Move( )方法。

程序员想对 Dog 对象调用方法 Move(int)，但存在一个问题。Dog 覆盖了不接受任何参数的 Move( )方法，但没有对其进行重载，使之接受一个 int 参数，即没有提供接受一个 int 参数的版本。第 37 行通过显式地调用基类的 Move(int)方法解决了这种问题。

提示：通过使用“::”来调用祖先的类方法时，如果在继承层次结构中、在祖先和后代之间插入了新的类，后代将跳过这些中间类，从而遗漏对中间类实现的重要功能的调用。

应该：

应通过派生来扩展经过测试的类的功能。

通过覆盖基类方法来改变派生类中某些函数的行为。

不应该：

不要通过修改函数特征标来隐藏基类函数。

别忘了 const 是特征标的组成部分。

别忘了返回类型不是特征标的组成部分。

12.5　虚　方　法

本章强调了这样一个事实：Dog 对象是一个 Mammal 对象。到目前为止，这只意味着 Dog 对象继承了其基类的属性（数据）和功能（方法）。然而，在 C++中，is-a 关系要比这深入得多。

C++扩展了其多态性，允许将派生类对象赋给指向基类的指针。因此，可以这样编写代码：

```
Mammal * pMammal = new Dog;
```

上述代码在堆中创建了一个新的 Dog 对象，并返回一个指向该对象的指针，然后将该指针赋给一个 Mammal 指针。之所以可以这样做，是因为狗是一种哺乳动物。

注意：这是多态的本质。例如，可创建很多类型的窗口，包括对话框、可滚动窗口和列表框，然后给每种窗口定义一个虚方法 draw( )。通过创建一个窗口指针，并将对话框和其他派生类对象赋给指针，就可以调用方法 draw( )，而不用不考虑运行时指针指向的实际对象类型。程序将调用正确的 draw( )方法。

然后可以使用这个指针来调用 Mammal 类的任何方法。你希望在调用被 Dog 覆盖的方法时，将调用正确的函数。虚函数让你能够做到这一点。要创建虚函数，可在函数声明前加上关键字 virtual。程序清单 12.8 演示了虚函数的工作原理以及使用非虚方法时将发生的情况。

程序清单 12.8　使用虚方法

```
1:  //Listing 12.8 Using virtual methods
2:  #include <iostream>
3:  using std::cout;
4:
5:  class Mammal
6:  {
7:    public:
8:       Mammal():itsAge(1) { cout << "Mammal constructor...\n"; }
9:       virtual ~Mammal() { cout << "Mammal destructor...\n"; }
10:      void Move() const { cout << "Mammal move one step\n"; }
11:      virtual void Speak() const { cout << "Mammal speak!\n"; }
12:
13:   protected:
14:      int itsAge;
15: };
16:
17: class Dog : public Mammal
18: {
19:   public:
20:     Dog() { cout << "Dog Constructor...\n"; }
21:     virtual ~Dog() { cout << "Dog destructor...\n"; }
22:     void WagTail() { cout << "Wagging Tail...\n"; }
23:     void Speak()const { cout << "Woof!\n"; }
```

```
24:     void Move()const { cout << "Dog moves 5 steps...\n"; }
25: };
26:
27: int main()
28: {
29:    Mammal *pDog = new Dog;
30:    pDog->Move();
31:    pDog->Speak();
32:
33:    return 0;
34: }
```

输出：

```
Mammal constructor...
Dog Constructor...
Mammal move one step
Woof!
```

分析：

第 11 行给 Mammal 类提供了一个虚方法：speak( )。这个类的设计者指出，他希望这个类最终会是另一个类的基类。派生类可能想覆盖这个函数。

第 29 行创建了一个 Mammal 指针（pDog），但将一个新 Dog 对象的地址赋给它。由于狗是哺乳动物，因此这种赋值是合法的。然后第 30 行使用这个指针来调用函数 Move( )。由于编译器知道 pDog 是一个 Mammal 指针，因此在 Mammal 类中查找 Move( ) 方法。从第 10 行可知，这是一个标准的、非虚方法，因此是该函数的 Mammal 版本。

第 31 行通过该指针调用 Speak( ) 函数。由于 Speak( ) 是一个虚方法（如第 11 行），因此调用 Dog 中的 Speak( ) 函数。

这几乎是在变魔术！对调用函数来说，它有一个 Mammal 指针，但这里却调用了 Dog 的方法。事实上，如果有一个 Mammal 指针数组，其中每个指针都指向不同的 Mammal 子类对象，则可以依次通过这些指针来调用虚方法，这将调用正确的函数。程序清单 12.9 说明了这一点。

程序清单 12.9　依次调用多个虚方法

```
1: //Listing 12.9 Multiple virtual functions called in turn
2: #include <iostream>
3: using namespace std;
4:
5: class Mammal
6: {
7:   public:
8:     Mammal():itsAge(1) { }
9:     virtual ~Mammal() { }
10:    virtual void Speak() const { cout << "Mammal speak!\n"; }
11:
12:  protected:
13:    int itsAge;
14: };
15:
16: class Dog : public Mammal
17: {
18:   public:
19:    void Speak()const { cout << "Woof!\n"; }
```

```
20: };
21:
22: class Cat : public Mammal
23: {
24:   public:
25:     void Speak()const { cout << "Meow!\n"; }
26: };
27:
28:
29: class Horse : public Mammal
30: {
31:   public:
32:     void Speak()const { cout << "Winnie!\n"; }
33: };
34:
35: class Pig : public Mammal
36: {
37:   public:
38:     void Speak()const { cout << "Oink!\n"; }
39: };
40:
41: int main()
42: {
43:    Mammal* theArray[5];
44:    Mammal* ptr;
45:    int choice, i;
46:    for ( i = 0; i<5; i++)
47:    {
48:       cout << "(1)dog (2)cat (3)horse (4)pig: ";
49:       cin >> choice;
50:       switch (choice)
51:       {
52:         case 1: ptr = new Dog;
53:                 break;
54:         case 2: ptr = new Cat;
55:                 break;
56:         case 3: ptr = new Horse;
57:                 break;
58:         case 4: ptr = new Pig;
59:                 break;
60:         default: ptr = new Mammal;
61:                  break;
62:       }
63:       theArray[i] = ptr;
64:    }
65:    for (i=0;i<5;i++)
66:       theArray[i]->Speak();
67:    return 0;
68: }
```

输出：

```
(1) dog (2) cat (3) horse (4) pig: 1
```

```
(1) dog (2) cat (3) horse (4) pig: 2
(1) dog (2) cat (3) horse (4) pig: 3
(1) dog (2) cat (3) horse (4) pig: 4
(1) dog (2) cat (3) horse (4) pig: 5
Woof!
Meow!
Whinny!
Oink!
Mammal speak!
```

分析：

这个简化的程序只提供了每个类最基本的功能，它以最简洁的形式演示了虚方法。声明了 4 个类：Dog、Cat、Horse 和 Pig，它们都是从 Mammal 类派生而来的。

第 10 行将 Mammal 的 Speak( )函数声明为虚方法。在第 19、25、32 和 38 行，4 个派生类覆盖了 Speak( )函数的实现。

在第 46～64 行，程序循环 5 次，每次都提示用户选择要创建哪种对象，然后第 50～62 行的 switch 语句将一个指向这种对象的指针加入到数组中。

在该程序编译时，无法知道将创建什么类型的对象，因此也无法知道将调用哪个 Speak( )方法。ptr 指向的对象是在运行阶段确定的，这被称为动态联编或运行阶段联编，与此相对的是静态联编或编译阶段联编。

在第 65 行和 66 行，程序再次遍历数组。这次，对数组中每个对象调用 Speak( )方法。由于在虚类中，Speak( )为虚方法，因此将根据对象的类型，调用相应的 Speak( )方法。如果选择不同的类型，将从输出中可知，确实调用了相应的方法。

FAQ

如果在基类中将一个成员方法标记为虚方法，还需要在派生类中将它标记为虚方法吗？

答：不需要。方法被声明为虚方法后，如果在派生类覆盖它，它仍是虚方法。在派生类中继续将方法标记为虚方法是个不错的主意（虽然不是必须这样做），这将使代码更容易理解。

12.5.1 虚函数的工作原理

创建派生对象（如 Dog 对象）时，首先调用基类的构造函数，然后调用派生类的构造函数。图 12.2 说明了 Dog 对象被创建后的情景。注意，Mammal 部分和 Dog 部分在内存中是相邻的。

在类中创建虚函数后，这个类的对象必须跟踪虚函数。很多编译器创建了虚函数表（v-table）。每个类都有一个虚函数表，每个类对象都有一个指向虚函数表的指针（vptr 或 v-pointer）。

虽然实现方式不同，但所有编译器都必须完成这项工作。

每个对象的 vptr 指向 v-table，对于每个虚函数，后者都包含一个指向它的指针（函数指针将在第 15 章深入讨论）。创建 Dog 的 Mammal 部分时，vptr 被初始化为指向 v-table 的正确部分，如图 12.3 所示。

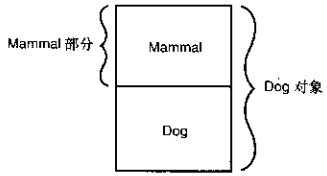

图 12.2 创建后的 Dog 对象

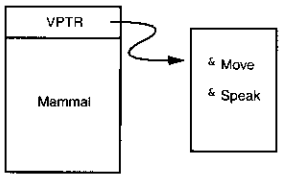

图 12.3 Mammal 的 v-table

当 Dog 的构造函数被调用时，添加对象的 Dog 部分，并调整 vptr 指针，使之指向 Dog 类中覆盖的虚函数（如果有的话），如图 12.4 所示。

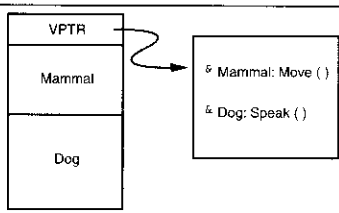

图 12.4　Dog 的 v-table

使用 Mammal 指针时，根据它指向的对象的实际类型，vptr 指向正确的函数。这样，调用函数 Speak( )时，将调用正确的函数。

12.5.2　通过基类指针访问派生类的方法

前面介绍过，通过基类指针可以访问派生类的虚方法。如果派生类有一个基类没有的方法，可以像访问虚方法那样通过基类指针来访问它吗？由于只有派生类有该方法，因此不存在名称冲突。

如果 Dog 有一个 WagTail( )方法，但 Mammal 没有，则不能通过 Mammal 指针来访问该方法。由于 WagTail( )不是虚方法，且 Mammal 没有这种方法，因此如果没有 Dog 对象或 Dog 指针，将不能访问它。

虽然可以将 Mammal 指针强制转换为 Dog 指针，但如果 Mammal 不是一个 Dog，这样做将是不安全的。虽然通过强制类型转换，可以将 Mammal 指针转换为 Dog 指针，但有一种更好、更安全的调用 WagTail( )函数的方法。C++不赞成强制类型转换，因为这样容易出错。这个主题将在第 15 章介绍多重继承时深入讨论，并在第 20 章介绍模板时再次进行讨论。

12.5.3　切除

仅当通过指针和引用进行调用时，才能发挥虚方法的魔力；按值传递对象时将不能发挥虚方法的魔力。程序清单 12.10 说明了这种问题。

程序清单 12.10　按值传递时的数据切除（slicing）

```
1:  //Listing 12.10 Data slicing with passing by value
2:  #include <iostream>
3:  using namespace std;
4:
5:  class Mammal
6:  {
7:    public:
8:      Mammal():itsAge(1) { }
9:      virtual ~Mammal() { }
10:     virtual void Speak() const { cout << "Mammal speak!\n"; }
11:
12:   protected:
13:     int itsAge;
14: };
15:
16: class Dog : public Mammal
17: {
18:   public:
19:     void Speak()const { cout << "Woof!\n"; }
20: };
21:
```

```
22: class Cat : public Mammal
23: {
24:   public:
25:     void Speak()const { cout << "Meow!\n"; }
26: };
27:
28: void ValueFunction (Mammal);
29: void PtrFunction (Mammal*);
30: void RefFunction (Mammal&);
31: int main()
32: {
33:    Mammal* ptr=0;
34:    int choice;
35:    while (1)
36:    {
37:       bool fQuit = false;
38:       cout << "(1)dog (2)cat (0)Quit: ";
39:       cin >> choice;
40:       switch (choice)
41:       {
42:          case 0: fQuit = true;
43:                  break;
44:          case 1: ptr = new Dog;
45:                  break;
46:          case 2: ptr = new Cat;
47:                  break;
48:          default: ptr = new Mammal;
49:                   break;
50:       }
51:       if (fQuit == true)
52:         break;
53:       PtrFunction(ptr);
54:       RefFunction(*ptr);
55:       ValueFunction(*ptr);
56:    }
57:    return 0;
58: }
59:
60: void ValueFunction (Mammal MammalValue)
61: {
62:    MammalValue.Speak();
63: }
64:
65: void PtrFunction (Mammal * pMammal)
66: {
67:    pMammal->Speak();
68: }
69:
70: void RefFunction (Mammal & rMammal)
71: {
72:    rMammal.Speak();
73: }
```

输出：

```
(1) dog (2) cat (0) Quit: 1
Woof
Woof
Mammal Speak!
(1) dog (2) cat (0) Quit: 2
Meow!
Meow!
Mammal Speak!
(1) dog (2) cat (0) Quit: 0
```

分析：

第 4～26 行声明了简化的 Mammal、Dog 和 Cat 类。声明了 3 个函数：PtrFunction( )、RefFunction( )和 ValueFunction( )，它们分别接受 Mammal 指针、Mammal 引用和 Mammal 对象作为参数。从第 60～73 行可知，这 3 个函数都做同一件事：调用 Speak( )方法。

程序提示用户选择 Dog 或 Cat，根据用户的选择，第 44 或 46 行创建一个指向相应类型的对象指针。

输出的第一行表明，用户选择了 Dog。第 44 行在自由存储区中创建一个 Dog 对象。然后通过指针（第 53 行）、引用（第 54 行）和值（第 55 行）将该 Dog 对象分别传递给 3 个函数。

接受指针和引用的函数都调用虚方法，因此调用成员函数 Dog->Speak( )，如第 2 和 3 行输出所示。

然而，第 55 行解除指针引用，并按值将结果传递给第 60～63 行定义的函数。函数希望接受一个 Mammal 对象，因此编译器将 Dog 对象切除到只余下 Mammal 部分。第 62 行调用 Speak( )方法时，只有 Mammal 的信息可用，Dog 部分没有了，第 4 行输出表明了这一点。这种效果被称为切除，因为转换为 Mammal（基类对象）时，对象的 Dog 部分（派生类部分）被切除了。

接下来，对 Cat 对象重复这样的过程，结果类似。

12.5.4　创建虚析构函数

在需要基类指针的地方使用指向派生类对象的指针是一种合法和常见的做法。当指向派生对象的指针被删除时将发生什么情况呢？如果析构函数是虚函数（应该如此），将执行正确的操作：调用派生类的析构函数。由于派生类的析构函数会自动调用基类的析构函数，因此整个对象将被正确地销毁。

经验规则是：如果类中任何一个函数是虚函数，析构函数也应该是虚函数。

注意：读者可能注意到了，本章的程序清单中都包含虚析构函数。现在读者知道其原因了！通常，总是将析构函数声明为虚函数是明智的选择。

12.5.5　虚复制构造函数

构造函数不能是虚函数，因此从技术上说，不存在虚复制构造函数。然而，有时候程序非常需要通过传递一个指向基类对象的指针，创建一个派生类对象的拷贝。对于这种问题，一种常见的解决方法是，在基类中创建一个 Clone( )方法，并将其设置为虚方法。Clone( )方法创建当前类对象的一个拷贝，并返回该拷贝。

由于每个派生类都覆盖了 Clone( )方法，因此它将创建派生类对象的一个拷贝。程序清单 12.11 演示了如何使用 Clone( )方法。

程序清单 12.11　虚复制构造函数

```
1:  //Listing 12.11 Virtual copy constructor
2:  #include <iostream>
3:  using namespace std;
4:
5:  class Mammal
```

```
 6: {
 7:   public:
 8:     Mammal():itsAge(1) { cout << "Mammal constructor...\n"; }
 9:     virtual ~Mammal() { cout << "Mammal destructor...\n"; }
10:     Mammal (const Mammal & rhs);
11:     virtual void Speak() const { cout << "Mammal speak!\n"; }
12:     virtual Mammal* Clone() { return new Mammal(*this); }
13:     int GetAge()const { return itsAge; }
14:   protected:
15:     int itsAge;
16: };
17:
18: Mammal::Mammal (const Mammal & rhs):itsAge(rhs.GetAge())
19: {
20:   cout << "Mammal Copy Constructor...\n";
21: }
22:
23: class Dog : public Mammal
24: {
25:   public:
26:     Dog() { cout << "Dog constructor...\n"; }
27:     virtual ~Dog() { cout << "Dog destructor...\n"; }
28:     Dog (const Dog & rhs);
29:     void Speak()const { cout << "Woof!\n"; }
30:     virtual Mammal* Clone() { return new Dog(*this); }
31: };
32:
33: Dog::Dog(const Dog & rhs):
34: Mammal(rhs)
35: {
36:   cout << "Dog copy constructor...\n";
37: }
38:
39: class Cat : public Mammal
40: {
41:   public:
42:     Cat() { cout << "Cat constructor...\n"; }
43:     ~Cat() { cout << "Cat destructor...\n"; }
44:     Cat (const Cat &);
45:     void Speak()const { cout << "Meow!\n"; }
46:     virtual Mammal* Clone() { return new Cat(*this); }
47: };
48:
49: Cat::Cat(const Cat & rhs):
50: Mammal(rhs)
51: {
52:   cout << "Cat copy constructor...\n";
53: }
54:
55: enum ANIMALS { MAMMAL, DOG, CAT};
56: const int NumAnimalTypes = 3;
57: int main()
```

```
58: {
59:    Mammal *theArray[NumAnimalTypes];
60:    Mammal* ptr;
61:    int choice, i;
62:    for ( i = 0; i<NumAnimalTypes; i++)
63:    {
64:       cout << "(1)dog (2)cat (3)Mammal: ";
65:       cin >> choice;
66:       switch (choice)
67:       {
68:         case DOG:  ptr = new Dog;
69:                    break;
70:         case CAT:  ptr = new Cat;
71:                    break;
72:         default:   ptr = new Mammal;
73:                    break;
74:       }
75:       theArray[i] = ptr;
76:    }
77:    Mammal *OtherArray[NumAnimalTypes];
78:    for (i=0;i<NumAnimalTypes;i++)
79:    {
80:       theArray[i]->Speak();
81:       OtherArray[i] = theArray[i]->Clone();
82:    }
83:    for (i=0;i<NumAnimalTypes;i++)
84:       OtherArray[i]->Speak();
85:    return 0;
86: }
```

输出：

```
 1: (1) dog (2) cat (3) Mammal: 1
 2: Mammal constructor...
 3: Dog constructor...
 4: (1) dog (2) cat (3) Mammal: 2
 5: Mammal constructor...
 6: Cat constructor...
 7: (1) dog (2) cat (3) Mammal: 3
 8: Mammal constructor...
 9: Woof!
10: Mammal Copy Constructor...
11: Dog copy constructor...
12: Meow!
13: Mammal Copy Constructor...
14: Cat copy constructor...
15: Mammal speak!
16: Mammal Copy Constructor...
17: Woof!
18: Meow!
19: Mammal speak!
```

分析：

除第 12 行给 Mammal 类新增了虚方法 Clone( )外，程序清单 12.11 与前两个程序清单非常相似。这个方

法通过调用复制构造函数，并将当前对象（*this）作为 const 引用传递给它，返回了一个指向新 Mammal 对象的指针。

Dog 和 Cat 都覆盖了 Clone( )方法，使之调用相应类的复制构造函数，并将当前对象传递给它。由于 Clone( )是虚方法，因此这相当于创建了一个虚复制构造函数。当第 81 行执行时，读者将明白这一点。

和前一个程序类似，该程序也提示用户选择创建狗、猫或哺乳动物，然后第 68～73 行创建用户指定的对象。第 75 行将指向对象的指针存放到数组中。

程序在第 78～82 行遍历数组，并通过每个指针调用 Speak( )和 Clone( )方法。第 81 行调用 Clone( )的结果为一个指向对象拷贝的指针，该指针被存储到另一个数组中。

第 1 行输出表明，用户被提示时选择了 1，即创建一个 Dog 对象，这将调用 Mammal 和 Dog 的构造函数。第 4 和第 8 行输出表明，对 Cat 和 Mammal 重复了这个过程。

第 9 行输出为对第一个对象（Dog 对象）调用 Speak( )的结果。由于 Speak( )是一个虚方法，因此将调用正确的 Speak( )版本。接下来调用 Clone( )函数，由于它也是一个虚方法，因此调用 Dog 的 Clone( )方法，这导致 Mammal 的构造函数和 Dog 的复制构造函数被调用。

第 12～14 行输出是对 Cat 重复上述过程的结果；第 15～16 行是对 Mammal 重复上述过程的结果。最后，第 83～84 行遍历新的数组，通过其中的每个指针调用 Speak( )方法，结果如第 17～19 行的输出所示。

12.5.6 使用虚方法的代价

由于包含虚方法的类必须维护一个 v-table，因此使用虚方法会带来一些开销。如果类很小，且不打算从它派生出其他类，则根本没有必要使用虚方法。

将任何方法声明为虚方法后，便付出了创建 v-table 的大部分代价（虽然每增加一个表项都会增加一些内存开销）。在这种情况下，应将析构函数声明为虚函数，并假设其他所有方法也可能是虚方法。应仔细琢磨那些非虚方法，确保理解它们为什么不是虚方法。

应该：

打算从一个类派生出其他类时，应在这个类中使用虚方法。

在任何一个方法为虚方法时，应将析构函数声明为虚函数。

不应该：

不要将构造函数声明为虚函数。

不要试图通过派生类对象访问基类的私有数据。

12.6 小　　结

本章介绍了派生类如何继承基类，讨论了公有继承和虚函数。派生类将继承其基类的所有公有和保护数据和函数。

对派生类来说，基类的所有保护成员都是公有的；而对其他类来说是私有的。即使是派生类，也不能访问其基类的私有数据和函数。

在构造函数体之前可以进行初始化。在初始化阶段，可以调用基类的构造函数，并给它传递参数。

在派生类中，可以覆盖基类的函数。如果函数是虚函数，且通过指针或引用来调用它时，将根据指针或引用在运行阶段指向的对象类型，调用相应类的该函数。

通过加上由基类名和两个冒号组成的前缀，可以调用基类的函数。例如，如果 Dog 是从 Mammal 派生而来的，则可以使用 Mammal::Walk( )来调用 Mammal 的 Walk( )方法。

在包含虚方法的类中，几乎总是应该将析构函数声明为虚函数。虚析构函数确保将 delete 用于指针时，指针指向的对象的派生部分得到释放。构造函数不能是虚函数。通过创建一个调用复制构造函数的虚函数，

相当于创建了一个虚复制构造函数。

12.7 问 与 答

问：继承而来的成员和函数能传给下一代吗？如果 Dog 从 Mammal 派生而来，而 Mammal 从 Animal 派生而来，Dog 将继承 Animal 的函数和数据吗？

答：可以。随着派生的进行，派生类将继承其所有基类的所有的函数和数据，但只能访问公有或保护函数和数据。

问：在上面的例子中，如果 Mammal 覆盖了 Animal 的一个函数，Dog 将继承原来的函数还是覆盖后的函数？

答：如果 Dog 从 Mammal 派生而来，将继承覆盖后的函数。

问：派生类可以把公有基类函数变成私有的吗？

答：可以。派生类可以覆盖该函数，使其变成私有的。这样，在接下来的派生类中，它将继续为私有的。然而，应尽可能避免这样做，因为用户期望类包含其祖先的所有方法。

问：为什么不将类函数都声明为虚函数？

答：创建 v-table 表的开销伴随第一个虚方法的创建而发生。在此之后，创建其他虚方法带来的开销将很小。很多 C++程序员认为，如果一个函数为虚函数，其他所有函数也应为虚函数；另一些程序员不同意这样观点，他们认为，无论做什么都应有充分的理由。

问：如果基类的一个函数（SomeFunc()）是虚函数，且被重载以便能接受一个或两个 int 参数，而派生类覆盖了接受一个 int 参数的版本，则通过指向派生类对象的指针调用接受两个 int 参数的函数时，将调用哪个函数？

答：正如本章正文中指出的，覆盖接受一个 int 参数的版本将隐藏基类中所有与此同名的函数，因此将出现编译错误，指出该函数只接受一个 int 参数。

12.8 作　　业

作业包括测验和练习，前者帮助加深读者对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 中的答案，继续学习下一章之前，请务必弄懂这些答案。

12.8.1 测验

1. 什么是 v-table？
2. 什么是虚析构函数？
3. 如何声明虚构造函数？
4. 如何创建一个虚复制构造函数？
5. 如何在派生类调用被覆盖的基类成员函数？
6. 如何在派生类调用未被覆盖的基类成员函数？
7. 如果基类将一个函数声明为虚函数，而派生类在覆盖这个函数没有使用关键字 virtual，则被第三代类继承时，该函数是否仍为虚函数？
8. 关键字 protected 的作用是什么？

12.8.2 练习

1. 声明一个接受一个 int 参数且返回 void 的虚函数。
2. 声明 Square 类，它从 Rectangle 派生而来，而 Rectangle 又从 Shape 派生而来。

3. 如果在练习 2 中，Shape 的构造函数不接受任何参数，Rectangle 的构造函数接受两个参数（长度和宽度），而 Square 的构造函数接受一个参数（长度），请写出 Square 的构造函数的初始化部分。

4. 为练习 3 中的 Square 类编写一个虚复制构造函数。

5. 查错：下面这段代码有什么错误？

```
void SomeFunction (Shape);
Shape * pRect = new Rectangle;
SomeFunction(*pRect);
```

6. 查错：下面这段代码有什么错误？

```
class Shape()
{
   public:
      Shape();
      virtual ~Shape();
      virtual Shape(const Shape&);
};
```