第 14 章　多　　态

第 12 章介绍了如何在派生类中编写虚函数，这是多态的基石：在运行阶段将派生类对象绑定到基类指针。本章介绍以下内容：

- 什么是多重继承及如何使用它？
- 什么是虚继承及何时使用它？
- 什么是抽象类及何时使用它们？
- 什么是纯虚函数？

14.1　单继承存在的问题

假设你使用动物类已有一段时间了，且将类层次结构分为了鸟类和哺乳动物。Bird 类包括成员函数 Fly()；从 Mammal 类派生出了包括 Horse 在内的很多哺乳动物类。Horse 类包括成员函数 Whinny()和 Gallop()。

突然你意识到你需要一个 Pegasus（飞马）对象：一种介于马和鸟之间的动物。Pegasus 类可以有成员函数 Fly()、Whinny()和 Gallop()。如果使用单继承，你将陷入困境。

使用单继承时，只能从一个类派生而来。可以从 Bird 派生出 Pegasus，但这样它将不会有成员函数 Whinny()和 Gallop()；也可从 Horse 派生出 Pegasus，但这样它将不会有成员函数 Fly()。

第一种解决方法是，从 Horse 类派生出 Pegasus，并将 Fly()方法复制到 Pegasus 类中。这是可行的，但代价是 Fly()方法位于两个地方（Bird 和 Pegasus 类中）。如果修改了其中一个，必须修改另一个。当然，数月或数年后负责维护代码的开发人员也必须知道需要修改这两个地方。

然而，很快又出现了新问题。你想创建一个 Horse 对象链表和一个 Bird 对象链表。你希望可以将 Pegasus 对象添加到任何一个链表中，但如果 Pegasus 是从 Horse 派生而来的，将不能把它添加到 Birds 链表中。

有两三种可能的解决方案。可以将 Horse 的方法 Gallop()重命名为 Move()，然后在 Pegasus 类中覆盖 Move()，使之完成 Fly()的工作。然后在其他从 Horse 派生而来的类中覆盖 Move()，使之完成 Gallop()的工作。也许 Pegasus 足够聪明，能够奔跑较短的距离和飞翔更长的距离。

```
Pegasus::Move(long distance)
{
    if (distance > veryFar)
        fly(distance);
    else
        gallop(distance);
}
```

这段代码有一定的局限性。也许有朝一日，Pegasus 想飞翔较短的距离或奔跑较长的距离。另一种解决方法是，将 Fly()方法移到 Horse 类中，如程序清单 14.1 所示。问题是大多数马都不能飞，因此必须使这个方

法什么也不做，除非其所属类为 Pegasus。

程序清单 14.1 如果马能飞

```
// Listing 14.1. If horses could fly...
// Percolating Fly() up into Horse

#include <iostream>
using namespace std;

class Horse
{
  public:
     void Gallop(){ cout << "Galloping...\n"; }
     virtual void Fly() { cout << "Horses can't fly.\n";}
  private:
     int itsAge;
};

class Pegasus : public Horse
{
  public:
     virtual void Fly()
          {cout<<"I can fly! I can fly! I can fly!\n";}
};

const int NumberHorses = 3;
int main()
{
   Horse* Ranch[NumberHorses];
   Horse* pHorse;
   int choice,i;
   for (i=0; i<NumberHorses; i++)
   {
     cout << "(1)Horse (2)Pegasus: ";
     cin >> choice;
     if (choice == 2)
       pHorse = new Pegasus;
     else
       pHorse = new Horse;
     Ranch[i] = pHorse;
   }
   cout << endl;
   for (i=0; i<NumberHorses; i++)
   {
     Ranch[i]->Fly();
     delete Ranch[i];
   }
   return 0;
}
```

输出：

```
(1)Horse (2)Pegasus: 1
```

```
(1)Horse (2)Pegasus: 2
(1)Horse (2)Pegasus: 1
(1)Horse (2)Pegasus: 2
(1)Horse (2)Pegasus: 1

Horses can't fly.
I can fly! I can fly! I can fly!
Horses can't fly.
I can fly! I can fly! I can fly!
Horses can't fly.
```

分析：

该程序肯定可行，虽然其代价是 Horse 类有一个 Fly( )方法。第 10 行为 Horse 类提供了 Fly( )方法。在现实世界的类中，你可能让它发出错误信号或平静地失败。在第 18 行，Pegasus 类覆盖了 Fly( )方法使其“完成正确的工作”，这里通过打印一条快乐消息来表示。

第 25 行名为 Ranch 的 Horse 指针数组用来演示，将根据运行阶段联编的 Horse 或 Pegasus 对象正确地调用 Fly( )方法。

第 28～37 行提示用户选择 Horse 或 Pegasus，然后根据用户的选择创建相应类型的对象，并将其放到数组 Ranch 中。

在第 39～43 行，程序通过循环来遍历数组 Ranch，对数组中每个对象调用 Fly( )方法。将根据对象是 Horse 还是 Pegasus 调用正确的 Fly( )方法，输出证明了这一点。由于程序不再使用数组 Ranch 中的对象，因此第 42 行调用 delete 来释放每个对象占用的内存。

注意：为突出要讨论的重点，对这些例子做了删节，只保留最基本的内容。为简化代码，删去了构造函数、虚析构函数等。在实际的程序中不推荐这样做。

14.1.1 提升

将所需的函数放到类层次结构中较高的位置，是一种常用的解决这种问题的方法，其结果是很多函数被提升（percolating up）到基类中。这样，基类很可能变成派生类可能使用的所有函数的全局名称空间。这将严重破坏 C++类的类型化（class typing），创建出庞杂的基类。

一般而言，将共有的功能沿层次结构向上提升，而不用迁移每个类的接口。这意味着如果两个类有相同的基类（如 Horse 和 Bird 的基类都是 Animal）且有一项相同的功能（如鸟和马都会吃），则应将该功能移到基类中，在其中创建一个虚函数。

然而需要避免的是，仅仅为了能够在某些派生类中调用它，而将函数（如 Fly( )）提升到不属于它的地方（不符合基类的含义）。

14.1.2 向下转换

采用单继承时，另一种办法是将把 Fly( )方法留在 Pegasus 中，且仅当指针指向 Pegasus 对象时才调用它。为此，需要知道指针实际指向的类型。这被称为运行阶段识别（Run Time Type Identification，RTTI）。

RTTI 是 C++规范中的一项新功能，并非所有的编译器都支持。如果编译器不支持 RTTI，可在每个类中放置一个返回枚举类型的方法来模拟 RTTI。然后可以在运行阶段检查类型，如果返回的是 Pegasus，则调用 Fly( )。

警告：在程序中应慎使 RTTI。需要使用 RTTI 可能意味着继承层次结构设计得很糟糕。应考虑使用虚函数、模板或多重继承来代替它。

在这个例子中，声明了 Horse 和 Pegasus 对象并将其存储到一个 Horse 对象数组中，每个对象都是作为 Horse 对象存储到数组中的。使用 RTTI，可检查每个对象看它实际上是 Horse 还是 Pegasus。

然而为调用 Fly( )，必须对指针进行类型转换以告诉指针，它指向的对象是 Pegasus 而不是 Horse。这被称为向下转换（casting down），因为你将 Horse 对象向下转换为派生类型。

虽然较勉强，现在 C++已经提供了新的 dynamic_cast 运算符，用于支持向下转换。其工作原理如下。

如果有一个指向基类（如 Horse）的指针，并将指向派生类（如 Pegasus）的指针赋给它，便可以以多态方式使用 Horse 指针。如果需要访问 Pegasus 对象，可创建一个 Pegasus 指针并使用 dynamic_cast 运算符来进行转换。

在运行阶段，将检查基类指针。如果转换正确，新的 Pegasus 指针也是正确的；如果转换不正确或根本没有 Pegasus 对象，新指针将为空。程序清单 14.2 演示了这一点。

程序清单 14.2 向下转换

```
0:  // Listing 14.2 Using dynamic_cast.
1:  // Using rtti
2:
3:  #include <iostream>
4:  using namespace std;
5:
6:  enum TYPE { HORSE, PEGASUS };
7:
8:  class Horse
9:  {
10:   public:
11:     virtual void Gallop(){ cout << "Galloping...\n"; }
12:
13:   private:
14:     int itsAge;
15: };
16:
17: class Pegasus : public Horse
18: {
19:   public:
20:     virtual void Fly()
21:         {cout<<"I can fly! I can fly! I can fly!\n";}
22: };
23:
24: const int NumberHorses = 5;
25: int main()
26: {
27:    Horse* Ranch[NumberHorses];
28:    Horse* pHorse;
29:    int choice,i;
30:    for (i=0; i<NumberHorses; i++)
31:    {
32:       cout << "(1)Horse (2)Pegasus: ";
33:       cin >> choice;
34:       if (choice == 2)
35:          pHorse = new Pegasus;
36:       else
37:          pHorse = new Horse;
38:       Ranch[i] = pHorse;
39:    }
40:    cout << endl;
```

```
41:    for (i=0; i<NumberHorses; i++)
42:    {
43:       Pegasus *pPeg = dynamic_cast< Pegasus *> (Ranch[i]);
44:       if (pPeg != NULL)
45:          pPeg->Fly();
46:       else
47:          cout << "Just a horse\n";
48:
49:       delete Ranch[i];
50:    }
51:    return 0;
52: }
```

输出：

```
(1)Horse (2)Pegasus: 1
(1)Horse (2)Pegasus: 2
(1)Horse (2)Pegasus: 1
(1)Horse (2)Pegasus: 2
(1)Horse (2)Pegasus: 1

Just a horse
I can fly! I can fly! I can fly!
Just a horse
I can fly! I can fly! I can fly!
Just a horse
```

FAQ

使用 Microsoft Visual C++进行编译时出现警告 "warning C5441: 'dynamic_cast' used on polymorphic type 'class Horse' with/GR-; unpredictable behavior may result." 我该怎么办？运行该程序时，出现如下消息："This application has requested the Runtime to terminate it in an unusual way. Please contact the application's support team for more information."

答：这是 MFC 最让人迷惑的错误消息之一。按以下步骤执行可解决这个问题：

1. 在工程中选择菜单 Project/Settings。
2. 打开 C++选项卡。
3. 从下拉式列表中选择 C++ Language。
4. 选中复选框 Enable Runtime Type Information(RTTI)。
5. 重新编译整个工程。

分析：

这种解决方案也可行，但不推荐这样做。

这取得预期的效果。Fly( )方法不在 Horse 中，没有对 Horse 对象调用它。然而，对 Pegasus 对象中调用 Fly( )时（第 45 行），必须进行显式转换（第 43 行）。由于 Horse 对象没有方法 Fly( )，因此必须在使用指针前告诉它，它指向的是 Pegasus 对象。

需要对 Pegasus 对象进行转换意味着设计可能有问题。该程序实际上破坏了虚函数的多态性，因为它依赖于将对象转换为运行阶段类型。

14.1.3 将对象添加到链表中

这些解决方案存在的另一个问题是，Pegasus 被声明为一种 Horse，所以不能将 Pegasus 对象添加到 Birds 链表中。你付出了将 Fly( )方法移到 Horse 中或对指针进行向下转换的代价，却没有获得所需的全部功能。

最后一种单继承解决方案是，将 Fly( )、Whinny( )和 Gallop( )都提升到 Bird 和 Horse 的基类 Animal 中。现在，可以使用一个统一的 Animals 链表，而不是分别使用一个 Birds 链表和一个 Horse 链表。这种解决方案也可行，但导致将派生类的所有特征都放在基类中。

另一种方法是，将这些方法留在原来的地方，但对 Horses、Birds 和 Pegasus 对象进行向下转换，这样做更糟。

应该：

务必将在概念上符合基类含义的功能沿继承层次向上提升。

务必避免执行基于对象的运行阶段类型的操作，而应使用虚函数、模板和多重继承。

不应该：

不要将为支持多态而本应在派生类中添加的功能放到基类中。

不要将基类指针向下转换为派生类指针。

14.2　多重继承

可以从多个基类派生出新类，这被称为多重继承。要从多个基类派生，在类声明中将每个基类用逗号分开，如下所示：

```
class DerivedClass: public BaseClass1, public BaseClass2 {}
```

除添加了基类 BaseClass2 外，这与声明单继承完全相同。

程序清单 14.3 说明了如何声明 Pegasus，使其从 Horse 和 Birds 派生而来。该程序然后将 Pegasus 对象添加到两种类型的链表中。

程序清单 14.3　多重继承

```
// Listing 14.3. Multiple inheritance.

#include <iostream>
using std::cout;
using std::cin;
using std::endl;

class Horse
{
  public:
    Horse() { cout << "Horse constructor... "; }
    virtual ~Horse() { cout << "Horse destructor... "; }
    virtual void Whinny() const { cout << "Whinny!... "; }
  private:
    int itsAge;
};

class Bird
{
  public:
    Bird() { cout << "Bird constructor... "; }
    virtual ~Bird() { cout << "Bird destructor... "; }
    virtual void Chirp() const { cout << "Chirp... "; }
    virtual void Fly() const
```

```
24:     {
25:       cout << "I can fly! I can fly! I can fly! ";
26:     }
27:   private:
28:     int itsWeight;
29: };
30:
31: class Pegasus : public Horse, public Bird
32: {
33:   public:
34:     void Chirp() const { Whinny(); }
35:     Pegasus() { cout << "Pegasus constructor... "; }
36:     ~Pegasus() { cout << "Pegasus destructor... "; }
37: };
38:
39: const int MagicNumber = 2;
40: int main()
41: {
42:   Horse* Ranch[MagicNumber];
43:   Bird* Aviary[MagicNumber];
44:   Horse * pHorse;
45:   Bird * pBird;
46:   int choice,i;
47:   for (i=0; i<MagicNumber; i++)
48:   {
49:     cout << "\n(1)Horse (2)Pegasus: ";
50:     cin >> choice;
51:     if (choice == 2)
52:       pHorse = new Pegasus;
53:     else
54:       pHorse = new Horse;
55:     Ranch[i] = pHorse;
56:   }
57:   for (i=0; i<MagicNumber; i++)
58:   {
59:     cout << "\n(1)Bird (2)Pegasus: ";
60:     cin >> choice;
61:     if (choice == 2)
62:       pBird = new Pegasus;
63:     else
64:       pBird = new Bird;
65:     Aviary[i] = pBird;
66:   }
67:
68:   cout << endl;
69:   for (i=0; i<MagicNumber; i++)
70:   {
71:     cout << "\nRanch[" << i << "]: " ;
72:     Ranch[i]->Whinny();
73:     delete Ranch[i];
74:   }
75:
```

```
76:    for (i=0; i<MagicNumber; i++)
77:    {
78:       cout << "\nAviary[" << i << "]: " ;
79:       Aviary[i]->Chirp();
80:       Aviary[i]->Fly();
81:       delete Aviary[i];
82:    }
83:    return 0;
84: }
```

输出:

```
(1)Horse (2)Pegasus: 1
Horse constructor...
(1)Horse (2)Pegasus: 2
Horse constructor... Bird constructor... Pegasus constructor...
(1)Bird (2)Pegasus: 1
Bird constructor...
(1)Bird (2)Pegasus: 2
Horse constructor... Bird constructor... Pegasus constructor...
Ranch[0]: Whinny!... Horse destructor...
Ranch[1]: Whinny!... Pegasus destructor... Bird destructor...
Horse destructor...
Aviary[0]: Chirp... I can fly! I can fly! I can fly! Bird destructor...
Aviary[1]: Whinny!... I can fly! I can fly! I can fly!
Pegasus destructor... Bird destructor... Horse destructor...
```

分析:

第 7~15 行声明了 Horse 类，其构造函数和析构函数分别打印一条消息，而 Whinny( )方法打印“Whinny!...”。

第 17~29 行声明了 Birds 类。除构造函数和析构函数外，这个类还有两个方法：Chirp( )和 Fly( )，它们都打印识别消息。在实际程序中，它们可能激活扬声器或生成动画图像。

最后，第 31～37 行是新的代码，使用多重继承声明了 Pegasus 类。从第 31 行可知，这个类是从 Horse 和 Bird 派生而来的。在第 34 行，Pegasus 类覆盖了 Chirp( )方法。Pegrsus 的 Chrip( )只是调用从 Horse 类继承的 Whinny( )方法。

在该程序的 main( )函数中创建了两个链表：第 42 行创建了一个名为 Ranch 的 Horse 指针数组，第 43 行创建了名为 Aviary 的 Bird 指针数组。第 47~56 行将 Horse 和 Pegasus 对象添加到数组 Ranch 中；第 57~66 行将 Bird 和 Pegasus 对象添加到数组 Aviary 中。

通过 Bird 指针和 Horse 指针调用虚方法时，都会为 Pegasus 对象执行正确的操作。例如，第 79 行使用数组 Aviary 的元素来对其指向的对象调用 Chirp( )。Bird 类将该方法声明为虚方法，因此对每个对象都调用了正确的函数。

输出表明，每次创建 Pegasus 对象时都创建了其 Bird 部分和 Horse 部分。当 Pegasus 对象被销毁时，其 Bird 和 Horse 部分也被销毁，这要归功于析构函数被声明为虚函数。

声明多重继承

要声明从多个类派生而来的类，在类名后加上一个冒号，再加上基类名即可。基类名之间用逗号分开。

范例 1:

```
class Pegasus : public Horse, public Bird
```

范例 2:

```
class Schnoodle : public Schnauzer, public Poodle
```

14.2.1 多重继承对象的组成部分

在内存中创建 Pegasus 对象时，两个基类都将成为 Pegasus 对象的组成部分，如图 14.1 所示。图中表示的是整个 Pegasus 对象，这包括 Pegasus 类新增的功能以及从基类那里继承而来的功能。

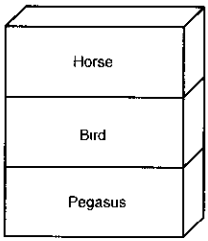

图 14.1 多重继承对象

有多个基类的对象存在一些问题。例如，如果碰巧两个基类有同名的虚函数或数据将如何呢？如何调用多个基类构造函数？如果多个基类都从同一个类派生而来又将如何呢？接下来的几节将回答这些问题并探讨如何使多重继承可行。

14.2.2 多重继承对象中的构造函数

如果 Pegasus 从 Horse 和 Bird 类派生而来且每个基类都有接受参数的构造函数，Pegasus 将依次调用这些构造函数。程序清单 14.4 说明了这是如何完成的。

程序清单 14.4 调用多个构造函数

```
// Listing 14.4
// Calling multiple constructors

#include <iostream>
using namespace std;

typedef int HANDS;
enum COLOR { Red, Green, Blue, Yellow, White, Black, Brown };

class Horse
{
  public:
    Horse(COLOR color, HANDS height);
    virtual ~Horse() { cout << "Horse destructor...\n"; }
    virtual void Whinny()const { cout << "Whinny!... "; }
    virtual HANDS GetHeight() const { return itsHeight; }
    virtual COLOR GetColor() const { return itsColor; }
  private:
    HANDS itsHeight;
    COLOR itsColor;
};

Horse::Horse(COLOR color, HANDS height):
itsColor(color),itsHeight(height)
{
  cout << "Horse constructor...\n";
}

class Bird
{
  public:
    Bird(COLOR color, bool migrates);
    virtual ~Bird() {cout << "Bird destructor...\n"; }
```

```
33:     virtual void Chirp()const { cout << "Chirp... "; }
34:     virtual void Fly()const
35:     {
36:       cout << "I can fly! I can fly! I can fly! ";
37:     }
38:     virtual COLOR GetColor()const { return itsColor; }
39:     virtual bool GetMigration() const { return itsMigration; }
40:
41:   private:
42:     COLOR itsColor;
43:     bool itsMigration;
44: };
45:
46: Bird::Bird(COLOR color, bool migrates):
47: itsColor(color), itsMigration(migrates)
48: {
49:   cout << "Bird constructor...\n";
50: }
51:
52: class Pegasus : public Horse, public Bird
53: {
54:   public:
55:     void Chirp()const { Whinny(); }
56:     Pegasus(COLOR, HANDS, bool,long);
57:     ~Pegasus() {cout << "Pegasus destructor...\n";}
58:     virtual long GetNumberBelievers() const
59:     {
60:       return itsNumberBelievers;
61:     }
62:
63:   private:
64:     long itsNumberBelievers;
65: };
66:
67: Pegasus::Pegasus(
68:    COLOR aColor,
69:    HANDS height,
70:    bool migrates,
71:    long NumBelieve):
72:    Horse(aColor, height),
73:    Bird(aColor, migrates),
74:    itsNumberBelievers(NumBelieve)
75: {
76:    cout << "Pegasus constructor...\n";
77: }
78:
79: int main()
80: {
81:    Pegasus *pPeg = new Pegasus(Red, 5, true, 10);
82:    pPeg->Fly();
83:    pPeg->Whinny();
84:    cout << "\nYour Pegasus is " << pPeg->GetHeight();
```

```
85:    cout << " hands tall and ";
86:    if (pPeg->GetMigration())
87:       cout << "it does migrate.";
88:    else
89:       cout << "it does not migrate.";
90:    cout << "\nA total of " << pPeg->GetNumberBelievers();
91:    cout << " people believe it exists." << endl;
92:    delete pPeg;
93:    return 0;
94: }
```

输出：

```
Horse constructor...
Bird constructor...
Pegasus constructor...
I can fly! I can fly! I can fly! Whinny!...
Your Pegasus is 7 hands tall and it does migrate.
A total of 10 people believe it exists.
Pegasus destructor...
Bird destructor...
Horse destructor...
```

分析：

第 9～20 行声明了 Horse 类，其构造函数接受两个参数：一个是第 7 行声明的颜色枚举类型，另一个是第 6 行声明的 typedef。第 22～26 行的构造函数实现只是初始化成员变量并打印一条消息。

第 28～44 行声明了 Bird 类，其构造函数的实现位于第 46～50 行。Bird 类的构造函数也接受两个参数。有趣的是，Horse 构造函数将颜色作为参数（以便你可以识别不同颜色的马），而 Bird 构造函数将羽毛颜色作为参数（同一种羽毛的鸟可友好相处）。当你想了解 Pegasus 的颜色时将引发一个问题，这将在下一个例子中看到。

Pegasus 类的声明位于第 52～65 行，其构造函数是在第 67～77 实现的。Pegasus 对象的初始化部分包括 3 条语句：首先，用颜色和高度作为参数来调用 Horse 构造函数（第 72 行）；然后，用颜色和指出是否迁徙的 Boolean 值作为参数来调用 Bird 构造函数（第 73 行）；最后，初始化 Pegasus 的成员变量 itsNumberBelievers。完成这些工作后，执行 Pegasus 的构造函数体。

在 main()函数中，第 81 行创建了一个 Pegasus 指针，然后使用它来访问从基类那里继承而来的成员函数。对这些方法的访问非常简单。

14.2.3 歧义解析

在程序清单 14.4 中，Horse 类和 Bird 类都有方法 GetColor()。读者可能注意到了，在程序清单 14.4 中没有调用这些方法！你可能需要让 Pegasus 对象返回其颜色，但将遇到问题：Pegasus 是从 Bird 和 Horse 类派生而来的，它们都有颜色，且获得颜色的方法的名称和特征标相同。这将给编译器带来歧义，你必须澄清。

如果编写下面这样的代码：

```
COLOR currentColor = pPeg->GetColor();
```

你出现这样的编译错误：

```
Member is ambiguous: 'Horse::GetColor' and 'Bird::GetColor'
```

显式地调用要调用的函数可解决这种歧义问题：

```
COLOR currentColor = pPeg->Horse::GetColor();
```

要指出从哪个类继承而来的成员函数或成员数据时，都可以通过在数据或函数前加上基类名来全限定

调用。

注意，如果 Pegasus 覆盖了这个函数，这个问题将挪到 Pegasus 成员函数中去（本该如此）：

```
virtual COLOR GetColor() const { return Horse::GetColor();}
```

这样就对 Pegasus 类的客户隐藏了问题，并在 Pegasus 中封装了它要从哪个基类继承颜色的知识。客户仍可通过编写下面的代码来解决这种问题：

```
COLOR currentColor = pPeg->Bird::GetColor();
```

14.2.4　从共同基类继承

如果 Bird 和 Horse 是从同一个基类（如 Animal）派生而来的，情况将如何呢？图 14.2 说明了这种情形。

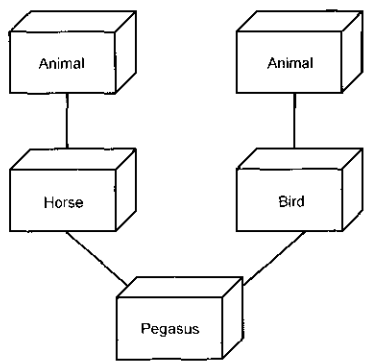

图 14.2　相同的基类

从图 14.2 可知，有两个基类对象。调用共同基类的函数或数据成员时，将出现另一种歧义。例如，如果 Animal 将 itsAge 声明为其成员变量，将 GetAge()声明为其成员函数，当你调用 pPeg->GetAge()时，指的是通过 Horse 还是 Bird 从 Animal 那里继承而来的 GetAge()呢？必须解决这种歧义问题，如程序清单 14.5 所示。

程序清单 14.5　共同基类

```
0:  // Listing 14.5
1:  // Common base classes
2:
3:  #include <iostream>
4:  using namespace std;
5:
6:  typedef int HANDS;
7:  enum COLOR { Red, Green, Blue, Yellow, White, Black, Brown };
8:
9:  class Animal       // common base to both horse and bird
10: {
11:   public:
12:     Animal(int);
13:     virtual ~Animal() { cout << "Animal destructor...\n"; }
14:     virtual int GetAge() const { return itsAge; }
15:     virtual void SetAge(int age) { itsAge = age; }
16:   private:
17:     int itsAge;
```

```
18: };
19:
20: Animal::Animal(int age):
21: itsAge(age)
22: {
23:    cout << "Animal constructor...\n";
24: }
25:
26: class Horse : public Anima_
27: {
28:   public:
29:     Horse(COLOR color, HANDS height, int age);
30:     virtual ~Horse() { cout << "Horse destructor...\n"; }
31:     virtual void Whinny()const { cout << "Whinny!... "; }
32:     virtual HANDS GetHeight() const { return itsHeight; }
33:     virtual COLOR GetColor() const { return itsColor; }
34:   protected:
35:     HANDS itsHeight;
36:     COLOR itsColor;
37: };
38:
39: Horse::Horse(COLOR color, HANDS height, int age):
40: Animal(age),
41: itsColor(color),itsHeight(height)
42: {
43:    cout << "Horse constructor...\n";
44: }
45:
46: class Bird : public Animal
47: {
48:   public:
49:     Bird(COLOR color, bool migrates, int age);
50:     virtual ~Bird() {cout << "Bird destructor...\n"; }
51:     virtual void Chirp()const { cout << "Chirp... "; }
52:     virtual void Fly()const
53:           { cout << "I can fly! I can fly! I can fly! "; }
54:     virtual COLOR GetColor()const { return itsColor; }
55:     virtual bool GetMigration() const { return itsMigration; }
56:   protected:
57:     COLOR itsColor;
58:     bool itsMigration;
59: };
60:
61: Bird::Bird(COLOR color, bool migrates, int age):
62: Animal(age),
63: itsColor(color), itsMigration(migrates)
64: {
65:    cout << "Bird constructor...\n";
66: }
67:
68: class Pegasus : public Horse, public Bird
69: {
```

```
70:   public:
71:     void Chirp()const { Whinny(); }
72:     Pegasus(COLOR, HANDS, bool, long, int);
73:     virtual ~Pegasus() {cout << "Pegasus destructor...\n";}
74:     virtual long GetNumberBelievers() const
75:     { return itsNumberBelievers; }
76:     virtual COLOR GetColor()const { return Horse::itsColor; }
77:     virtual int GetAge() const { return Horse::GetAge(); }
78:   private:
79:     long itsNumberBelievers;
80: };
81:
82: Pegasus::Pegasus(
83:    COLOR aColor,
84:    HANDS height,
85:    bool migrates,
86:    long NumBelieve,
87:    int age):
88:    Horse(aColor, height,age),
89:    Bird(aColor, migrates,age),
90:    itsNumberBelievers(NumBelieve)
91: {
92:    cout << "Pegasus constructor...\n";
93: }
94:
95: int main()
96: {
97:    Pegasus *pPeg = new Pegasus(Red, 5, true, 10, 2);
98:    int age = pPeg->GetAge();
99:    cout << "This pegasus is " << age << " years old.\n";
100:   delete pPeg;
101:   return 0;
102: }
```

输出：

```
Animal constructor...
Horse constructor...
Animal constructor...
Bird constructor...
Pegasus constructor...
This pegasus is 2 years old.
Pegasus destructor...
Bird destructor...
Animal destructor...
Horse destructor...
Animal destructor...
```

分析：

该程序清单有多个有趣的特性。Animal 类的声明位于第 9～18 行，它新增了一个成员变量（itsAge）和两个存取器函数（GetAge( )和 SetAge( )）。

第 26 行将 Horse 声明为从 Animal 类派生而来。现在 Horse 构造函数接受第三个参数（年龄），它将该参数传递给基类 Animal（第 40 行）。注意，Horse 类没有覆盖 GetAge( )，而只是继承它。

在第 46 行，Bird 类也被声明为从 Animal 派生而来，其构造函数也接受年龄作为参数并使用它去初始化基类 Animal（第 62 行）。它也只是继承 GetAge()而没有覆盖它。

在第 68 行，Pegasus 从 Bird 和 Animal 派生而来，因此在它的继承链中有两个 Animal 类。如果要对 Pegasus 对象调用 GetAge()方法，而 Pegasus 类没有覆盖这个方法，则必须明确或完全限定该方法。

第 77 行覆盖 GetAge()方法，使之什么也不做而不足调用基类的同名方法，从而解决了这种问题。

调用基类的同名方法的原因有两个：明确要调用哪个基类的方法（如本例所示）；或者先做一些工作，然后再让基类的函数做另外一些工作。有时候，可能想先做一些工作然后调用基类的同名方法；或者先调用基类的同名方法，并在该方法返回后再做一些工作。

从第 82 行开始的 Pegasus 构造函数接受 5 个参数：颜色、高度（类型为 HANDS）、是否迁徙、有多少人相信它和年龄。

在第 88 行，构造函数使用颜色、高度和年龄初始化 Pegasus 的 Horse 部分；第 89 行使用颜色、是否迁徙和年龄初始化 Bird 部分；最后，第 90 行初始化 itsNumberBelieves。

第 88 行对 Horse 构造函数的调用将调用第 39 行的实现。Horse 构造函数用年龄参数初始化 Pegasus 的 Horse 部分中的 Animal 部分，然后初始化 Horse 的两个成员变量：itsColor 和 itsHeight。

第 89 行对 Bird 构造函数的调用将调用第 61 行的实现。这里也使用年龄参数来初始化 Bird 的 Animal 部分。

注意，Pegasus 的颜色参数被用来初始化 Bird 和 Horse 的成员变量。另外，年龄被用来初始化 Horse 和 Bird 部分中的 Animal 的 itsAge。

警告：别忘了，每当你显式地指定祖先类时都将带来这样的风险：如果以后在当前类及其祖先类之间插入了一个新类，将导致调用新的祖先类，直接进入原来的祖先类。这可能导致意外的后果。

14.2.5 虚继承

在程序清单 14.5 中，Pegasus 尽力想明确使用哪个 Animal 基类。大多数时候，决定使用哪个是随意的，毕竟 Horse 和 Bird 有相同的基类。

可以告诉 C++，不想使用共同基类的两个拷贝，而只想要一个共同基类的拷贝，如图 14.3 所示。

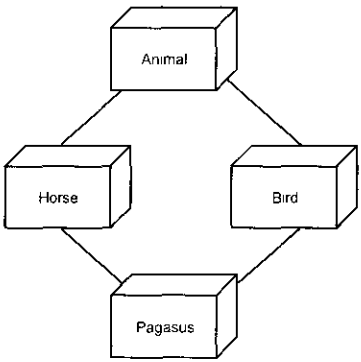

图 14.3 钻石形继承

为此，可让 Animal 成为 Horse 和 Bird 的虚基类。根本不用修改 Animal，对于 Horse 和 Bird 类，只需在其声明中使用关键字 virtual 即可。然而，对 Pegasus 类却要做重大修改。

通常，类的构造函数只初始化自己的变量及其基类，但虚继承的基类例外，它们由最后的派生类进行初始化。因此，Animal 不是由 Horse 和 Bird 初始化，而是由 Pegasus 初始化。Horse 和 Bird 必须在其构造函数

中初始化 Animal，但创建 Pegasus 对象时，这些初始化将被忽略。

程序清单 14.6 重写了程序清单 14.5 以利用虚继承。

程序清单 14.6 使用虚继承

```
// Listing 14.6
// Virtual inheritance
#include <iostream>
using namespace std;

typedef int HANDS;
enum COLOR { Red, Green, Blue, Yellow, White, Black, Brown };

class Animal       // common base to both horse and bird
{
  public:
    Animal(int);
    virtual ~Animal() { cout << "Animal destructor...\n"; }
    virtual int GetAge() const { return itsAge; }
    virtual void SetAge(int age) { itsAge = age; }
  private:
    int itsAge;
};

Animal::Animal(int age):
itsAge(age)
{
   cout << "Animal constructor...\n";
}

class Horse : virtual public Animal
{
  public:
    Horse(COLOR color, HANDS height, int age);
    virtual ~Horse() { cout << "Horse destructor...\n"; }
    virtual void Whinny()const { cout << "Whinny!... "; }
    virtual HANDS GetHeight() const { return itsHeight; }
    virtual COLOR GetColor() const { return itsColor; }
  protected:
    HANDS itsHeight;
    COLOR itsColor;
};

Horse::Horse(COLOR color, HANDS height, int age):
Animal(age),
itsColor(color),itsHeight(height)
{
   cout << "Horse constructor...\n";
}

class Bird : virtual public Animal
{
  public:
```

```
48:     Bird(COLOR color, bool migrates, int age);
49:     virtual ~Bird() {cout << "Bird destructor...\n"; }
50:     virtual void Chirp()const { cout << "Chirp... "; }
51:     virtual void Fly()const
52:           { cout << "I can fly! I can fly! I can fly! "; }
53:     virtual COLOR GetColor()const { return itsColor; }
54:     virtual bool GetMigration() const { return itsMigration; }
55:   protected:
56:     COLOR itsColor;
57:     bool itsMigration;
58: };
59:
60: Bird::Bird(COLOR color, bool migrates, int age):
61: Animal(age),
62: itsColor(color), itsMigration(migrates)
63: {
64:   cout << "Bird constructor...\n";
65: }
66:
67: class Pegasus : public Horse, public Bird
68: {
69:   public:
70:     void Chirp()const { Whinny(); }
71:     Pegasus(COLOR, HANDS, bool, long, int);
72:     virtual ~Pegasus() {cout << "Pegasus destructor...\n";}
73:     virtual long GetNumberBelievers() const
74:         { return itsNumberBelievers; }
75:     virtual COLOR GetColor()const { return Horse::itsColor; }
76:   private:
77:     long itsNumberBelievers;
78: };
79:
80: Pegasus::Pegasus(
81:    COLOR aColor,
82:    HANDS height,
83:    bool migrates,
84:    long NumBelieve,
85:    int age):
86:    Horse(aColor, height,age),
87:    Bird(aColor, migrates,age),
88:    Animal(age*2),
89:    itsNumberBelievers(NumBelieve)
90: {
91:    cout << "Pegasus constructor...\n";
92: }
93:
94: int main()
95: {
96:    Pegasus *pPeg = new Pegasus(Red, 5, true, 10, 2);
97:    int age = pPeg->GetAge();
98:    cout << "This pegasus is " << age << " years old.\n";
99:    delete pPeg;
```

```
100:    return 0;
101: }
```

输出：

```
Animal constructor...
Horse constructor...
Bird constructor...
Pegasus constructor...
This pegasus is 4 years old.
Pegasus destructor...
Bird destructor...
Horse destructor...
Animal destructor...
```

分析：

第 25 行将 Horse 声明为从 Animal 虚继承而来；第 45 行对 Bird 做了同样声明。注意 Bird 和 Animal 的构造函数仍初始化其 Animal 部分。

Pegasus 从 Bird 和 Animal 派生而来，作为 Animal 最后的派生类，它也初始化 Animal。然而，调用的是 Pegasus 的初始化部分，而忽略 Bird 和 Horse 中对 Animal 构造函数的调用。之所以知道这一点，是因为传递的值为 2，Bird 和 Horse 将其传给 Animal，但 Pegasus 将乘以 2。第 98 行的打印输出表明，结果为 4。

Pegasus 不用明确调用 GetAge( )引起的歧义，因为它只从 Animal 那里继承一个这样的函数。然而，Pegasus 仍必须明确对 GetColor( )的调用，因为这个函数位于其两个基类而不是 Animal 类中。

声明虚继承类

为确保派生类只有一个共同基类的实例，可将中间类声明为从共同基类虚继承。

范例 1：

```
class Horse : virtual public Animal
class Bird : virtual public Animal
class Pegasus : public Horse, public Bird
```

范例 2：

```
class Schnauzer : virtual public Dog
class Poodle : virtual public Dog
class Schnoodle : public Schnauzer, public Poodle
```

14.2.6　多重继承存在的问题

虽然多重继承与单继承相比有很多优点，但很多 C++程序员却并不愿意使用它。他们提出的问题是，多重继承增加调试难度，开发多重继承类层次结构比开发单继承类层次结构更困难且风险更大，几乎所有用多重继承能完成的工作不用它也能完成。基于这些原因，诸如 Java 和 C#等语言不支持多重继承。

这些担心是有道理的，应避免在程序中引入不必要的复杂性。有些调试器对多重继承问题无能为力，有些设计在没必要的情况下使用多重继承，给其带来了不必的复杂性。

应该：

当新类需要多个基类的函数和特性时应使用多重继承。

当最后的派生类只能有一个共同基类的实例时必须使用虚继承。

使用虚基类时务必在最后的派生类中初始化共同基类。

不应该：

在单继承可行的情况下不要使用多重继承。

14.2.7 混合（功能）类

在多重继承和单继承之间的一种折衷方案是使用混合类（mixin）。可以从 Animal 和 Displayable 类派生出 Horse 类，Displayable 只是添加一些将对象显示在屏幕上的方法。

混合（功能）类增加专用功能而不会增加大量方法或数据的类。

将功能类混合到派生类中的方法与其他类相同：将派生类声明为以公有方式继承功能类。功能类和其他类的惟一区别是：功能类没有或只有很少的数据。当然这种区分是不明确的，是一种表示只增加一些功能而不增加派生类复杂程度的方式。

对于某些调试器来说，这使得处理混合继承对象比处理复杂的多重继承对象更容易。另外，在访问其他基类的数据时出现歧义的可能性更低。

例如，如果 Horse 从 Animal 和 Displayable 派生而来，而 Displayable 没有数据，这样，Horse 的全部数据都来自 Animal，而 Horse 中的函数来自两者。

注意：混合继承的说法来自马萨诸塞州 Sommerville 的一家冰激凌店，该店将糖果和蛋糕混合到基本的冰激凌口味中。这好像是某些在那里度暑假的面向对象程序员，尤其是使用面向对象编程语言 SCOOPS 的程序员的比喻。

14.3 抽象数据类型

你经常需要创建类层次结构。例如，你可能创建 Shape 类，然后从其派生出 Rectangle 和 Circle。从 Rectangle 类，又可能派生出 Square 类，将其作为 Rectangle 的特例。

每个派生类都将覆盖 Draw( )、GetArea( )方法等。程序清单 14.7 是 Shape 及其派生类 Circle 和 Rectangle 的简单实现。

程序清单 14.7 Shape 类

```
0:  //Listing 14.7. Shape classes.
1:
2:  #include <iostream>
3:  using std::cout;
4:  using std::cin;
5:  using std::endl;
6:
7:  class Shape
8:  {
9:    public:
10:     Shape(){}
11:     virtual ~Shape(){}
12:     virtual long GetArea() { return -1; } // error
13:     virtual long GetPerim() { return -1; }
14:     virtual void Draw() {}
15:   private:
16: };
17:
18: class Circle : public Shape
19: {
20:   public:
21:     Circle(int radius):itsRadius(radius){}
```

```
22:     ~Circle(){}
23:     long GetArea() { return 3 * itsRadius * itsRadius; }
24:     long GetPerim() { return 6 * itsRadius; }
25:     void Draw();
26:   private:
27:     int itsRadius;
28:     int itsCircumference;
29: };
30:
31: void Circle::Draw()
32: {
33:   cout << "Circle drawing routine here!\n";
34: }
35:
36:
37: class Rectangle : public Shape
38: {
39:   public:
40:     Rectangle(int len, int width):
41:     itsLength(len), itsWidth(width){}
42:     virtual ~Rectangle(){}
43:     virtual long GetArea() { return itsLength * itsWidth; }
44:     virtual long GetPerim() {return 2*itsLength + 2*itsWidth; }
45:     virtual int GetLength() { return itsLength; }
46:     virtual int GetWidth() { return itsWidth; }
47:     virtual void Draw();
48:   private:
49:     int itsWidth;
50:     int itsLength;
51: };
52:
53: void Rectangle::Draw()
54: {
55:   for (int i = 0; i<itsLength; i++)
56:   {
57:     for (int j = 0; j<itsWidth; j++)
58:       cout << "x ";
59:
60:   cout << "\n";
61:   }
62: }
63:
64: class Square : public Rectangle
65: {
66:   public:
67:     Square(int len);
68:     Square(int len, int width);
69:     ~Square(){}
70:     long GetPerim() {return 4 * GetLength();}
71: };
72:
73: Square::Square(int len):
```

```
74: Rectangle(len,len)
75: {}
76:
77: Square::Square(int len, int width):
78: Rectangle(len,width)
79: {
80:    if (GetLength() != GetWidth())
81:    cout << "Error, not a square... a Rectangle??\n";
82: }
83:
84: int main()
85: {
86:    int choice;
87:    bool fQuit = false;
88:    Shape * sp;
89:
90:    while ( !fQuit )
91:    {
92:       cout << "(1)Circle (2)Rectangle (3)Square (0)Quit: ";
93:       cin >> choice;
94:
95:       switch (choice)
96:       {
97:        case 0:  fQuit = true;
98:          break;
99:        case 1: sp = new Circle(5);
100:         break;
101:       case 2: sp = new Rectangle(4,6);
102:         break;
103:       case 3: sp = new Square(5);
104:         break;
105:       default:
106:         cout <<"Please enter a number between 0 and 3"<<endl;
107:         continue;
108:         break;
109:      }
110:      if( !fQuit )
111:        sp->Draw();
112:      delete sp;
113:      sp = 0;
114:      cout << endl;
115:   }
116:   return 0;
117: }
```

输出：

```
(1)Circle (2)Rectangle (3)Square (0)Quit: 2
x x x x x x
x x x x x x
x x x x x x
x x x x x x
```

```
(1)Circle (2)Rectangle (3)Square (0)Quit:3
x x x x x
x x x x x
x x x x x
x x x x x
x x x x x

(1)Circle (2)Rectangle (3)Square (0)Quit:0
```

分析：

第 7～16 行声明了 Shape 类。GetArea( )和 GetPerim( )方法返回一个错误值，而 Draw( )不执行任何操作，绘制 Shape 究竟意味着什么呢？只有具体的形状（如圆、矩形等）才能绘制，Shape 作为一个抽象概念是不能绘制的。

在第 18～29 行，Circle 从 Shape 派生而来，并覆盖了 3 个虚方法。注意，没有理由使用关键字 virtual，因为这是其继承性的一部分；但这样做也没有害处，如 Rectangle 类声明中的第 43、44 和 47 行所示。作为提示（一种说明方式），加上关键字 virtual 是个不错的主意。

在第 64～71 行，Square 从 Rectangle 派生而来，它覆盖了 GetPerim( )方法，继承了 Rettangle 中定义的其他方法。

客户试图实例化 Shape 对象会很麻烦，也许应该使其不可能。毕竟，Shape 类只是为其派生类提供接口，因此它是一种抽象数据类型（ADT）。

在抽象类中，接口表示一种概念（如形状）而不是具体的对象（如圆）。在 C++中，抽象类只能用作其他类的基类，不能创建抽象类的实例。

14.3.1　纯虚函数

C++通过提供纯虚函数来支持创建抽象类。通过将虚函数初始化为 0 来将其声明为纯虚的，如下所示：

```
virtual void Draw() = 0;
```

在这个例子中，类有一个 Draw( )函数，但其实现为空，因此不能被调用。

任何包含一个或多个纯虚函数的类都是抽象类，因此不能对其实例化。事实上，试图对抽象类或从抽象类派生而来但没有实现其所有纯虚函数的类进行实例化都是非法的。试图这样做将导致编译错误。将纯虚函数放在类中向其客户指出了两点：

- 不要创建这个类的对象；而应从其派生。
- 务必覆盖从这个类继承的纯虚函数。

若从抽象类派生而来的类中，继承的纯虚函数仍是纯虚的，要实例化这种类的对象，必须覆盖每个纯虚函数。因此，如果 Rectangle 从 Shape 派生而来，而 Shape 有 3 个纯虚函数，Rectangle 必须覆盖这 3 个纯虚函数，否则它也将是抽象类。程序清单 14.8 重写了 Shape 类，使其成为一种抽象类；为节省篇幅，这里没有再次列出程序清单 14.7 中的其他代码。请用程序清单 14.8 中 Shape 的声明替换程序清单 14.7 程序中第 7～16 行的 Shape 声明，然后重新运行该程序。

程序清单 14.8　抽象类

```
0:  //Listing 14.8 Abstract Classes
1:
2:  class Shape
3:  {
4:    public:
5:      Shape(){}
6:      ~Shape(){}
7:      virtual long GetArea() = 0;
```

```
 8:      virtual long GetPerim()= 0;
 9:      virtual void Draw() = 0;
10:   private:
11: };
```

输出：

```
(1)Circle (2)Rectangle (3)Square (0)Quit: 2
x x x x x x
x x x x x x
x x x x x x
x x x x x x

(1)Circle (2)Rectangle (3)Square (0)Quit: 3
x x x x x
x x x x x
x x x x x
x x x x x
x x x x x

(1)Circle (2)Rectangle (3)Square (0)Quit: 0
```

分析：

正如读者看到的，程序的运行情况完全相同。惟一的区别是，现在不能创建 Shape 类对象了。

抽象数据类型

要将类声明为抽象类（也叫抽象数据类型），在类声明中包含一个或多个纯虚函数即可。要将函数声明为纯虚的，在声明函数后加上“=0”即可。

范例：

```
class Shape
{
    virtual void Draw() = 0;    // pure virtual
};
```

14.3.2 实现纯虚函数

通常，不实现抽象基类中的纯虚函数。由于不能创建抽象类的对象，因此没有理由提供实现。另外，抽象类用作从其派生而来的类的接口定义。

然而，可以给纯虚函数提供实现。这样，就可以通过从抽象类派生而来的对象调用该函数，该函数可能旨在给所有覆盖函数提供通用功能。程序清单 14.9 重写了程序清单 14.7，这次将 Shape 声明为抽象类并提供了纯虚函数 Draw( )的实现。Circle 类覆盖了 Draw( )方法（必须这样做），并调用了基类的 Draw( )方法以提供额外的功能。

在这个例子中，额外的功能只是打印一条消息，但可以想见，基类可提供通用的绘图机制，这也许是设置所有派生类都要用到的窗口。

程序清单 14.9 实现纯虚函数

```
0: //Listing 14.9 Implementing pure virtual functions
1:
2: #include <iostream>
3: using namespace std;
4:
```

```
 5: class Shape
 6: {
 7:   public:
 8:     Shape(){}
 9:     virtual ~Shape(){}
10:     virtual long GetArea() = 0;
11:     virtual long GetPerim()= 0;
12:     virtual void Draw() = 0;
13:   private:
14: };
15:
16: void Shape::Draw()
17: {
18:   cout << "Abstract drawing mechanism!\n";
19: }
20:
21: class Circle : public Shape
22: {
23:   public:
24:     Circle(int radius):itsRadius(radius){}
25:     virtual ~Circle(){}
26:     long GetArea() { return 3.14 * itsRadius * itsRadius; }
27:     long GetPerim() { return 2 * 3.14 * itsRadius; }
28:     void Draw();
29:   private:
30:     int itsRadius;
31:     int itsCircumference;
32: };
33:
34: void Circle::Draw()
35: {
36:   cout << "Circle drawing routine here!\n";
37:   Shape::Draw();
38: }
39:
40:
41: class Rectangle : public Shape
42: {
43:   public:
44:     Rectangle(int len, int width):
45:     itsLength(len), itsWidth(width){}
46:     virtual ~Rectangle(){}
47:     long GetArea() { return itsLength * itsWidth; }
48:     long GetPerim() {return 2*itsLength + 2*itsWidth; }
49:     virtual int GetLength() { return itsLength; }
50:     virtual int GetWidth() { return itsWidth; }
51:     void Draw();
52:   private:
53:     int itsWidth;
54:     int itsLength;
55: };
56:
```

```
57: void Rectangle::Draw()
58: {
59:    for (int i = 0; i<itsLength; i++)
60:    {
61:       for (int j = 0; j<itsWidth; j++)
62:          cout << "x ";
63:
64:    cout << "\n";
65:    }
66:    Shape::Draw();
67: }
68:
69:
70: class Square : public Rectangle
71: {
72:   public:
73:     Square(int len);
74:     Square(int len, int width);
75:     virtual ~Square(){}
76:     long GetPerim() {return 4 * GetLength();}
77: };
78:
79: Square::Square(int len):
80: Rectangle(len,len)
81: {}
82:
83: Square::Square(int len, int width):
84: Rectangle(len,width)
85:
86: {
87:    if (GetLength() != GetWidth())
88:    cout << "Error, not a square... a Rectangle??\n";
89: }
90:
91: int main()
92: {
93:    int choice;
94:    bool fQuit = false;
95:    Shape * sp;
96:
97:    while (fQuit == false)
98:    {
99:       cout << "(1)Circle (2)Rectangle (3)Square (0)Quit: ";
100:      cin >> choice;
101:
102:      switch (choice)
103:      {
104:       case 1: sp = new Circle(5);
105:            break;
106:       case 2: sp = new Rectangle(4,6);
107:            break;
108:       case 3: sp = new Square (5);
```

```
109:            break;
110:       default: fQuit = true;
111:            break;
112:       }
113:       if (fQuit == false)
114:       {   lib:          sp->Draw();
116:          delete sp;
117:          cout << endl;
118:       }
119:    }
120:    return 0;
121: }
```

输出：

```
(1)Circle (2)Rectangle (3)Square (0)Quit: 2
x x x x x x
x x x x x x
x x x x x x
x x x x x x
Abstract drawing mechanism!
(1)Circle (2)Rectangle (3)Square (0)Quit: 3
x x x x x
x x x x x
x x x x x
x x x x x
x x x x x
Abstract drawing mechanism!

(1)Circle (2)Rectangle (3)Square (0)Quit: 0
```

分析：

第 5～14 行声明了抽象类 Shape，其 3 个存取器函数都被声明为纯虚函数。注意这不是必须的，但是一种不错的做法。只要任何一个函数被声明为纯虚函数，这个类就是抽象类。

GetArea( )和 GetPerim( )方法没有实现，但第 16～19 行实现了 Draw( )方法。Circle 和 Rectangle 都覆盖了 Draw( )，它们都调用了同名的基类方法，以利用基类提供的通用功能。

14.3.3　复杂的抽象层次结构

有时候，会从抽象类派生出其他抽象类。你可能想将一些继承而来的纯虚函数变成非纯虚函数，而保留其他的不变。

如果创建 Animal 类，可将 Eat( )、Sleep( )、Move( )和 Reproduce( )都声明为纯虚函数。你可能从 Animal 类派生出 Mammal 和 Fish 类。

经过考察后，你发现每种哺乳动物都以相同的方式进行繁殖，因此决定将 Mammal::Reproduce( )变成非纯虚函数，但保留 Eat( )、Sleep( )和 Move( )为纯虚函数。

你从 Mammal 派生出 Dog，Dog 类必须覆盖和实现其他 3 个纯虚函数，以使能够创建 Dog 类对象。

作为类设计人员，你刚才的意思是，Animals 和 Mammals 不能实例化，但所有的哺乳动物类都可继承提供的 Reproduce( )方法而不覆盖它。

程序清单 14.10 通过这些类的简化实现说明了这种技巧。

程序清单 14.10　从抽象类派生出其他抽象类

```
0:  // Listing 14.10
```

```
// Deriving Abstract Classes from other Abstract Classes
#include <iostream>
using namespace std;

enum COLOR { Red, Green, Blue, Yellow, White, Black, Brown } ;

class Animal       // common base to both Mammal and Fish
{
  public:
    Animal(int);
    virtual ~Animal() { cout << "Animal destructor...\n"; }
    virtual int GetAge() const { return itsAge; }
    virtual void SetAge(int age) { itsAge = age; }
    virtual void Sleep() const = 0;
    virtual void Eat() const = 0;
    virtual void Reproduce() const = 0;
    virtual void Move() const = 0;
    virtual void Speak() const = 0;
  private:
    int itsAge;
};

Animal::Animal(int age):
itsAge(age)
{
   cout << "Animal constructor...\n";
}

class Mammal : public Animal
{
  public:
    Mammal(int age):Animal(age)
      { cout << "Mammal constructor...\n";}
    virtual ~Mammal() { cout << "Mammal destructor...\n";}
    virtual void Reproduce() const
      { cout << "Mammal reproduction depicted...\n"; }
};

class Fish : public Animal
{
  public:
    Fish(int age):Animal(age)
      { cout << "Fish constructor...\n";}
    virtual ~Fish() {cout << "Fish destructor...\n"; }
    virtual void Sleep() const { cout << "fish snoring...\n"; }
    virtual void Eat() const { cout << "fish feeding...\n"; }
    virtual void Reproduce() const
      { cout << "fish laying eggs...\n"; }
    virtual void Move() const
      { cout << "fish swimming...\n";  }
    virtual void Speak() const { }
};
```

```
53:
54: class Horse : public Mammal
55: {
56:   public:
57:     Horse(int age, COLOR color ):
58:     Mammal(age), itsColor(color)
59:       { cout << "Horse constructor...\n"; }
60:     virtual ~Horse() { cout << "Horse destructor...\n"; }
61:     virtual void Speak()const { cout << "Whinny!... \n"; }
62:     virtual COLOR GetItsColor() const { return itsColor; }
63:     virtual void Sleep() const
64:       { cout << "Horse snoring...\n"; }
65:     virtual void Eat() const { cout << "Horse feeding...\n"; }
66:     virtual void Move() const { cout << "Horse running...\n";}
67:
68:   protected:
69:     COLOR itsColor;
70: };
71:
72: class Dog : public Mammal
73: {
74:   public:
75:     Dog(int age, COLOR color ):
76:     Mammal(age), itsColor(color)
77:        { cout << "Dog constructor...\n"; }
78:     virtual ~Dog() { cout << "Dog destructor...\n"; }
79:     virtual void Speak()const { cout << "Woof!... \n"; }
80:     virtual void Sleep() const { cout << "Dog snoring...\n"; }
81:     virtual void Eat() const { cout << "Dog eating...\n"; }
82:     virtual void Move() const { cout << "Dog running...\n"; }
83:     virtual void Reproduce() const
84:       { cout << "Dogs reproducing...\n"; }
85:
86:   protected:
87:     COLOR itsColor;
88: };
89:
90: int main()
91: {
92:    Animal *pAnimal=0;
93:    int choice;
94:    bool fQuit = false;
95:
96:    while (fQuit == false)
97:    {
98:       cout << "(1)Dog (2)Horse (3)Fish (0)Quit: ";
99:       cin >> choice;
100:
101:      switch (choice)
102:      {
103:        case 1: pAnimal = new Dog(5,Brown);
104:                break;
105:        case 2: pAnimal = new Horse(4,Black);
```

```
106:                break;
107:         case 3: pAnimal = new Fish (5);
108:                break;
109:        default: fQuit = true;
110:                break;
111:       }
112:       if (fQuit == false)
113:       {
114:         pAnimal->Speak();
115:         pAnimal->Eat();
116:         pAnimal->Reproduce();
117:         pAnimal->Move();
118:         pAnimal->Sleep();
119:         delete pAnimal;
120:         cout << endl;
121:       }
122:    }
123:    return 0;
124: }
```

输出：

```
(1)Dog (2)Horse (3)Bird (0)Quit: 1
Animal constructor...
Mammal constructor...
Dog constructor...
Whoof!...
Dog eating...
Dog reproducing....
Dog running...
Dog snoring...
Dog destructor...
Mammal destructor...
Animal destructor...

(1)Dog (2)Horse (3)Bird (0)Quit: 0
```

分析：

第 7～21 行声明了抽象类 Animal。Animal 有一个用于 itsAge 的非纯虚存取器函数，它们由所有动物类共享；还有 5 个纯虚函数：Sleep( )、Eat( )、Reproduce( )、Move( )和 Speak( )。

在第 29～37 行，Mammal 被声明为从 Animal 派生而来，且没有新增任何数据；但覆盖了 Reproduce( )方法，为所有哺乳动物提供了通用的繁殖方式。Fish 必须覆盖 Reproduce( )，因为它直接从 Animal 类派生而来，不能采用哺乳动物的繁殖方式，这是在第 47～48 行实现的。

现在，哺乳动物类不必覆盖 Reproduce( )方法，但也可以选择这样做，就像 Dog 类声明中的第 83 行那样。Fish、Horse 和 Dog 都覆盖了其他的纯虚函数，这样便可以实例化相应类型的对象了。

在该程序的 main( )函数中，将一个 Animal 指针依次指向各种派生类对象。调用虚函数时，将根据指针在运行阶段指向的对象，调用正确的派生类方法。

试图实例化 Animal 或 Mammal 将导致编译错误，因为它们都是抽象类。

14.3.4 哪些类是抽象的

在一个程序中，Animal 类是抽象的，而在另一个程序可能不是。什么因素决定应将类声明为抽象的呢？

这个问题的答案不取决于现实世界的固有因素，而是由程序如何做合理而决定的。假设你要编写一个描述农场或动物园的程序，可能将 Animal 声明为抽象类，但希望能实例化 Dog 类对象。

另一方面，如果你要描述各种狗，可能将 Dog 声明为抽象类，只实例化具体类型的狗：拾物狗、猎犬等。抽象层次取决于需要如何细分类型。

应该：

务必使用抽象类给系列相关类通用的功能提供描述。

务必将所有必须覆盖的函数声明为纯虚的。

不应该：

不要试图实例化抽象类对象。

14.4　小　　结

本章介绍了如何克服单继承的某些局限性。读者知道了将函数沿继承层次结构向上提升以及沿继承层次结构向下转换的风险。还学习了如何使用多重继承、多重继承会带来哪些问题以及如何使用虚继承来解决这些问题。

本章还介绍了什么是抽象类以及如何使用纯虚函数来创建抽象类。学习了如何实现纯虚函数以及何时和为什么要这样做。

14.5　问　与　答

问：何为 v-ptr？

答：v-ptr（虚函数指针）是虚函数的一个实现细节。每个有虚函数的类的对象都有一个 v-ptr，它指向这个类的虚函数表。编译器需要确定在特定情形下应调用哪个函数时，将查询虚函数表。

问：提升函数总是件好事吗？

答：如果提升的是共有函数，答案是肯定的；如果提升的是接口，答案是否定的。也就是说，如果所有派生类都不能使用该方法，那么将其移到共同基类中是错误的。如果这样做，必须启用对象的运行阶段类型识别功能，才能决定是否可以调用该函数。

问：为什么判断对象的运行阶段类型是糟糕的？

答：因为这表明没有合理的构建继承层次结构，最好回过头去修复设计方案，而不是使用这种规避方式。

问：为什么向下转换是糟糕的？

答：如果以类型安全（type-safe）的方式进行，向下转换并没有什么不好。然而，向下转换可能破坏 C++ 的强类型检查，这是需要避免的。如果启用运行阶段类型识别，并对指针进行向下转换，这可能表明设计有问题。

另外，函数应使用参数和成员变量的声明类型，而不依赖于通过某种隐式约定来获悉调用程序将提供什么。如果假定是错误的，将导致怪异和不可预测的问题。

问：为什么不将所有函数都声明为虚函数？

答：虚函数由虚函数表支持，后者会带来运行阶段开销，这种开销表现在程序的大小和性能方面。如果类非常小且预期不会有子类，则不应将其任何函数声明为虚函数。然而，当这种假设不成立时，应回过头去将相关类的函数声明为虚函数，否则将导致意料外的问题。

问：什么时候应将析构函数声明为虚函数？

答：在你认为类将被用作基类且基类指针将被用来访问派生类对象时。一种经验规则是，如果已经将类中的任何函数声明为虚函数，一定要将析构函数也声明为虚函数。

问：为什么要将类声明为抽象的？为什么不将它声明为非抽象类并避免创建这种类的对象呢？

答：C++中的很多约定旨在让编译器能够发现错误，避免将代码提供给客户后发生运行阶段错误。将类声明为抽象的（给它提供纯虚函数）可让编译器将任何使用抽象类创建对象的代码视为错误。

14.6 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。

14.6.1 测验

1. 什么是向下转换？
2. 将功能向上提升是什么意思？
3. 如果圆角矩形有直边和圆角，且 RoundRect 类从 Rectangle 和 Circle 派生而来，而 Rectangle 和 Circle 类都是从 Shape 派生而来，则创建一个 RoundRect 对象时将创建多少个 Shapes？
4. 如果 Horse 和 Bird 都采用公有虚继承从 Animal 派生而来，它们的构造函数会调用 Animal 的构造函数吗？如果 Pegasus 从 Horse 和 Bird 派生而来，它将如何调用 Animal 的构造函数？
5. 声明一个名为 Vehicle 的抽象类。
6. 如果基类为抽象类，它有 3 个纯虚函数，则其派生类需要覆盖其中的多少个函数？

14.6.2 练习

1. 声明 Jetplane 类，它从 Rocket 和 Airplane 派生而来。
2. 编写 Seven47 类的声明，它从练习 1 中的 Jetplane 类派生而来。
3. 编写类 Car 和 Bus 的声明，它们都从 Vehicle 类派生而来。将 Vehicle 类声明为包含有两个纯虚函数的抽象类。将 Car 和 Bus 声明为非抽象类。
4. 修改练习 3 编写的代码，将 Car 声明为抽象类并从 Car 派生出 SportsCar 和 Coupe。在 Car 类中，为 Vehicle 的一个纯虚函数提供实现，使其成为非纯虚函数。