第16章 继 承

因为继承可以创建层次分类，所以它是面向对象程序设计（OOP）的基础之一。人们可以利用继承创建一个普通的类并在该类中定义一些特性，这些特性是一组相关的项所共同具有的。因此，这个类可以被其他更特殊的类所继承，这些类只将那些各自独具的内容添加进来。

为了与标准C++术语保持一致，被继承的类称为基类，通过继承而来的类称为派生类。再进一步，某个派生类可以用做另一个派生类的基类。多继承就是这样实现的。

C++对继承的支持既丰富又灵活，本书在第11章中已经有过介绍，我们在这里更加详细地予以分析。

16.1 基类访问控制

如果一个类继承了另一个类，那么基类的成员将成为派生类的成员。类继承采用下面的通用形式：

```
class derived-class-name : access base-class-name {
 // body of class
};
```

派生类中基类成员的访问状态是由 access 决定的。基类访问限定符必须是 public，private 或protected。倘若没有提供访问限定符，那么如果派生类是class，则访问限定符默认为private；如果派生类是struct，那么在没有明确说明访问限定符的情况下，其默认值为public。下面分析一下 public 和 private 的区别（protected 限定符在下一节讲解）。

当某个基类的访问限定符为public时，基类的所有公有成员将成为派生类的公有成员，基类的所有保护成员将成为派生类的保护成员。然而，基类的私有元素将总是为基类专有，不能被派生类访问。例如，在下面的程序中，类型 derived 的对象可以直接访问 base 的公有成员：

```
#include <iostream>
using namespace std;

class base {
  int i, j;
public:
  void set(int a, int b) { i=a; j=b; }
  void show() { cout << i << " " << j << "\n"; }
};

class derived : public base {
  int k;
public:
  derived(int x) { k=x; }
  void showk() { cout << k << "\n"; }
};
```

```
int main()
{
  derived ob(3);

  ob.set(1, 2); // access member of base
  ob.show(); // access member of base

  ob.showk(); // uses member of derived class

  return 0;
}
```

当利用private访问限定符继承基类时，基类的所有公有成员和保护成员将变成派生类的私有成员。例如，由于set( )和show( )现在是derived的私有成员，下面的程序甚至不能编译：

```
// This program won't compile.
#include <iostream>
using namespace std;

class base {
  int i, j;
public:
  void set(int a, int b) { i=a; j=b; }
  void show() { cout << i << " " << j << "\n";}
};

// Public elements of base are private in derived.
class derived : private base {
  int k;

public:
  derived(int x) { k=x; }
  void showk() { cout << k << "\n"; }
};

int main()
{
  derived ob(3);

  ob.set(1, 2); // error, can't access set()
  ob.show(); // error, can't access show()

  return 0;
}
```

注意：当某个基类的访问限定符为private时，该基类的公有成员和保护成员将变成派生类的私有成员。也就是说，它们仍然可以被派生类的成员访问，但却不能被用户程序中既不是基类、也不是派生类的成员访问。

16.2 继承和保护成员

C++中的关键字protected给继承机制提供了更大的灵活性。当某个类成员声明为protected时，该成员将不能被程序中其他的非成员元素所访问。但是有一个例外，访问保护成员与访问私有成员相同——只能被该类的其他成员访问，惟一的例外发生在保护成员被继承时。在这种

情况下，保护成员与私有成员具有本质的不同。

正如前一节讲到的那样，基类的私有成员不能被程序中包括所有派生类的其他部分访问。然而，保护成员则不同。如果基类被继承为public，那么基类的保护成员将成为派生类的保护成员，因而可以被派生类访问。可以利用protected创建类成员，这些类成员是其类所私有的，但仍然可以被派生类继承和访问，参见下面的例子：

```
#include <iostream>
using namespace std;

class base {
protected:
  int i, j; // private to base, but accessible by derived
public:
  void set(int a, int b) { i=a; j=b; }
  void show() { cout << i << " " << j << "\n"; }
};

class derived : public base {
  int k;
public:
  // derived may access base's i and j
  void setk() { k=i*j; }

  void showk() { cout << k << "\n"; }
};

int main()
{
  derived ob;

  ob.set(2, 3); // OK, known to derived
  ob.show(); // OK, known to derived

  ob.setk();
  ob.showk();

  return 0;
}
```

在这个例子中，因为base被derived以public方式继承，而且i和j被声明为protected，所以可以被derived的函数setk( )访问。如果i和j已经被base声明为private，那么将不能被derived访问，面且程序也不能编译。

当一个派生类被用做另一个派生类的基类时，最初被第一个派生类以公有方式(public)继承的基类中的所有保护成员也可以再次被第二个派生类以保护方式(protected)继承。例如，下面的程序是正确的，而且derived2的确可以访问i和j。

```
#include <iostream>
using namespace std;

class base {
protected:
  int i, j;
public:
```

```
  void set(int a, int b) { i=a; j=b; }
  void show() { cout << i << " " << j << "\n"; }
};

// i and j inherited as protected.
class derived1 : public base {
  int k;
public:
  void setk() { k = i*j; } // legal
  void showk() { cout << k << "\n"; }
};

// i and j inherited indirectly through derived1.
class derived2 : public derived1 {
  int m;
public:
  void setm() { m = i-j; } // legal
  void showm() { cout << m << "\n"; }
};

int main()
{
  derived1 ob1;
  derived2 ob2;

  ob1.set(2, 3);
  ob1.show();
  ob1.setk();
  ob1.showk();

  ob2.set(3, 4);
  ob2.show();
  ob2.setk();
  ob2.setm();
  ob2.showk();
  ob2.showm();

  return 0;
}
```

然而，如果base被继承为private，那么base的所有成员都将成为derived1的私有成员，也就是说，这些成员不能被derived2访问（然而，i和j仍然可以被derived1访问）。下面的程序说明了这种情况，该程序是错误的，因而不能编译。其中的注释部分对每个错误进行了说明：

```
// This program won't compile.
#include <iostream>
using namespace std;

class base {
protected:
  int i, j;
public:
  void set(int a, int b) { i=a; j=b; }
  void show() { cout << i << " " << j << "\n"; }
};
```

```
// Now, all elements of base are private in derived1.
class derived1 : private base {
  int k;
public:
  // this is legal because i and j are private to derived1
  void setk() { k = i*j; } // OK
  void showk() { cout << k << "\n"; }
};

// Access to i, j, set(), and show() not inherited.
class derived2 : public derived1 {
  int m;
public:
  // illegal because i and j are private to derived1
  void setm() { m = i-j; } // Error
  void showm() { cout << m << "\n"; }
};

int main()
{
  derived1 ob1;
  derived2 ob2;

  ob1.set(1, 2); // error, can't use set()
  ob1.show(); // error, can't use show()

  ob2.set(3, 4); // error, can't use set()
  ob2.show(); // error, can't use show()

  return 0;
}
```

注意： 即使base被derived1继承为private，derived1仍然可以访问base的public和protected元素，然而不能传递这种特权。

16.2.1 受保护基类的继承

将基类继承为保护类是可行的。如果是这样，该基类的所有公有成员和保护成员将成为派生类的保护成员。范例如下：

```
#include <iostream>
using namespace std;

class base {
protected:
 int i, j; // private to base, but accessible by derived
public:
  void setij(int a, int b) { i=a; j=b; }
  void showij() { cout << i << " " << j << "\n"; }
};

// Inherit base as protected.
class derived : protected base{
  int k;
```

```
public:
  // derived may access base's i and j and setij().
  void setk() { setij(10, 12); k = i*j; }

  // may access showij() here
  void showall() { cout << k << " "; showij(); }
};

int main()
{
  derived ob;

// ob.setij(2, 3); // illegal, setij() is
//                    protected member of derived

  ob.setk(); // OK, public member of derived
  ob.showall(); // OK, public member of derived

// ob.showij(); // illegal, showij() is protected
//                 member of derived

  return 0;
}
```

通过阅读注释可知，即使 setij( )和 showij( ) 是基类的公有成员，但是当利用访问限定符 protected 继承该基类时，它们将变成 derived 的保护成员，也就是说，不能在 main( )的内部访问这些成员。

16.3 继承多个基类

一个派生类有可能从两个或更多的基类继承。例如，在下面这个简短的例子中，derived既继承 base1，又继承 base2。

```
// An example of multiple base classes.

#include <iostream>
using namespace std;

class base1 {
protected:
  int x;
public:
  void showx() { cout << x << "\n"; }
};

class base2 {
protected:
  int y;
public:
  void showy() { cout << y << "\n";}
};

// Inherit multiple base classes.
class derived: public base1, public base2 {
public:
```

```
  void set(int i, int j) { x=i; y=j; }
};

int main()
{
  derived ob;

  ob.set(10, 20); // provided by derived
  ob.showx(); // from base1
  ob.showy(); // from base2

  return 0;
}
```

从上面的例子可见，为了继承多个基类，需要使用以逗号分隔的列表。还有，要确保对每一个被继承的基类使用一个访问限定符。

16.4 构造函数、析构函数和继承

涉及到继承时会遇到两个与构造函数和析构函数有关的问题。第一，什么时候调用基类和派生类构造函数和析构函数？第二，怎样将参数传递到基类构造函数？本节主要探讨这两个主要问题。

16.4.1 什么时候执行构造函数和析构函数

基类、派生类或者这两个类都有可能包含构造函数和/或析构函数，重要的是要理解当一个派生类的对象创立和消亡时这些函数的执行顺序。我们先看一看下面这个简短的程序：

```
#include <iostream>
using namespace std;
class base {
public:
  base() { cout << "Constructing base\n"; }
  ~base() { cout << "Destructing base\n"; }
};

class derived: public base {
public:
  derived() { cout << "Constructing derived\n"; }
  ~derived() { cout << "Destructing derived\n"; }
};

int main()
{
  derived ob;

  // do nothing but construct and destruct ob

  return 0;
}
```

正如main( )中的注释语句说明的那样，这个程序只是构造和销毁derived类的一个称为ob的对象。当执行时，该程序显示：

```
Constructing base
Constructing derived
Destructing derived
Destructing base
```

可以看到，第一个base的构造函数执行完之后，跟着执行derived的构造函数，接下来调用derived的析构函数（因为ob在这个程序中立即被销毁），然后调用base的析构函数。

由前面程序的结果可以归纳出：当创建一个派生类对象时，首先调用的是基类的构造函数，然后才调用派生类的构造函数；当销毁一个派生类对象时，首先调用的是它的析构函数，然后才调用基类的析构函数。二者的不同在于：构造函数按照其派生顺序执行，而析构函数按照与派生顺序相反的顺序执行。

如果仔细想一想，就不难理解为什么构造函数会按照派生顺序执行。这是因为基类并不知道派生类的情况，基类执行的初始化独立于派生类的初始化，并且有可能是派生类初始化的先决条件。因此，基类必须首先被执行。

同样，析构函数的执行顺序与派生顺序相反也十分明智。因为基类是派生类的基础，析构基类对象就意味着析构派生类对象，因此，派生类析构函数必须在对象完全销毁之前被调用。

在多重继承情况下（也就是说，一个派生类是另一个派生类的基类），一般的规则是：构造函数按照派生顺序被调用，析构函数按照与派生顺序相反的顺序被调用，例如，下面的程序：

```
#include <iostream>
using namespace std;

class base {
public:
  base() { cout << "Constructing base\n"; }
  ~base() { cout << "Destructing base\n"; }
};

class derived1 : public base {
public:
  derived1() { cout << "Constructing derived1\n"; }
  ~derived1() { cout << "Destructing derived1\n"; }
};

class derived2: public derived1 {
public:
  derived2() { cout << "Constructing derived2\n"; }
  ~derived2() { cout << "Destructing derived2\n"; }
};

int main()
{
  derived2 ob;

  // construct and destruct ob

  return 0;
}
```

显示的结果如下：

```
Constructing base
Constructing derived1
Constructing derived2
Destructing derived2
Destructing derived1
Destructing base
```

同样的规则适用于多个基类的情况。例如，下面的程序：

```
#include <iostream>
using namespace std;

class base1 {
public:
  base1() { cout << "Constructing base1\n"; }
  ~base1() { cout << "Destructing base1\n"; }
};

class base2 {
public:
  base2() { cout << "Constructing base2\n"; }
  ~base2() { cout << "Destructing base2\n"; }
};

class derived: public base1, public base2 {
public:
  derived() { cout << "Constructing derived\n"; }
  ~derived() { cout << "Destructing derived\n"; }
};

int main()
{
  derived ob;

  // construct and destruct ob

  return 0;
}
```

生成下面的输出：

```
Constructing base1
Constructing base2
Constructing derived
Destructing derived
Destructing base2
Destructing base1
```

可以看到，构造函数根据derived继承列表中指定的类从左到右按照派生顺序被调用，析构函数从右到左按照相反的顺序被调用。也就是说，在derived列表中，先指定base2，后指定base1，如下所示：

```
class derived: public base2, public base1 {
```

所以这个程序的输入应该是：

```
Constructing base2
Constructing base1
Constructing derived
Destructing derived
Destructing base1
Destructing base2
```

16.4.2 把参数传递给基类构造函数

到目前为止，前面例子包括的构造函数都不需要参数。如果只有派生类的构造函数需要一个或多个参数，则简单地使用标准的参数化构造函数语法（参见第12章）。然而，怎样给基类中的构造函数传递参数呢？答案是：利用派生类构造函数声明的扩展形式把参数传递给一个或多个基类构造函数。该扩展的派生类构造函数声明的一般形式如下：

```
derived-constructor(arg-list) : base1(arg-list),
                                base2(arg-list),
                                //...
                                baseN(arg-list)
{
 // body of derived constructor
}
```

其中，base1 到 baseN 是被派生类继承的基类名。可以看到，派生类构造函数声明和基类声明用一个冒号(:)分开，在多个基类的情况下，各基类之间用逗号(,)分开。看一看下面的程序：

```
#include <iostream>
using namespace std;

class base {
protected:
  int i;
public:
  base(int x) { i=x; cout << "Constructing base\n"; }
  ~base() { cout << "Destructing base\n"; }
};

class derived: public base {
  int j;
public:
  // derived uses x; y is passed along to base.
  derived(int x, int y): base(y)
    { j=x; cout << "Constructing derived\n"; }

  ~derived() { cout << "Destructing derived\n"; }
  void show() { cout << i << " " << j << "\n"; }
};

int main()
{
  derived ob(3, 4);

  ob.show(); // displays 4 3
```

```
  return 0;
}
```

其中，derived的构造函数被声明为带有两个参数x和y，然而derived( )只使用x, y被传给了base( )。一般来说，派生类构造函数不但要声明自己需要的参数，而且要声明基类需要的参数。正如上例所示的，基类需要的参数通过在冒号后面指定的基类参数列表传递给它。

下面的例子使用了多个基类：

```
#include <iostream>
using namespace std;

class base1 {
protected:
  int i;
public:
  base1(int x) { i=x; cout << "Constructing base1\n"; }
  ~base1() { cout << "Destructing base1\n"; }
};

class base2 {
protected:
  int k;
public:
  base2(int x) { k=x; cout << "Constructing base2\n"; }
  ~base2() { cout << "Destructing base1\n"; }
};

class derived: public base1, public base2 {
  int j;
public:
  derived(int x, int y, int z): base1(y), base2(z)
    { j=x; cout << "Constructing derived\n"; }

  ~derived() { cout << "Destructing derived\n"; }
  void show() { cout << i << " " << j << " " << k << "\n"; }
};

int main()
{
  derived ob(3, 4, 5);

  ob.show(); // displays 4 3 5

  return 0;
}
```

基类构造函数的参数是通过派生类构造函数的参数传递的，理解这一点非常重要。因此，即使某个派生类构造函数不需要使用参数，但如果基类需要，该构造函数也必须声明相应的参数。在这种情况下，传递给派生类的参数只是简单地传递给基类。例如，在下面的程序中，派生类构造函数不需要参数，但是base1( )和base2( )需要：

```
#include <iostream>
using namespace std;

class base1 {
```

```
protected:
  int i;
public:
  base1(int x) { i=x; cout << "Constructing base1\n"; }
  ~base1() { cout << "Destructing base1\n"; }
};

class base2 {
protected:
  int k;
public:
  base2(int x) { k=x; cout << "Constructing base2\n"; }
  ~base2() { cout << "Destructing base2\n"; }
};

class derived: public base1, public base2 {
public:
  /* Derived constructor uses no parameter,
     but still must be declared as taking them to
     pass them along to base classes.
  */

  derived(int x, int y): base1(x), base2(y)
    { cout << "Constructing derived\n"; }

  ~derived() { cout << "Destructing derived\n"; }
  void show() { cout << i << " " << k << "\n"; }
};

int main()
{
  derived ob(3, 4);

  ob.show(); // displays 3 4

  return 0;
}
```

派生类构造函数可以随意使用它所声明的部分或全部参数,即使其中的一个或多个参数被传递给基类;换句话说,把某个参数传递给基类并不能阻止派生类使用该参数。例如,下面的程序段是完全正确的:

```
class derived: public base {
  int j;
public:
  // derived uses both x and y and then passes them to base.
  derived(int x, int y): base(x, y)
    { j = x*y; cout << "Constructing derived\n"; }
```

最后,当给基类构造函数传递参数时要记住:参数可以由任何有效的表达式组成,其中包括函数调用和变量。这与C++允许动态初始化是一致的。

16.5 准许访问

当基类以 private 方式被继承时，基类中所有公有成员和保护成员都成为派生类的私有成员。然而在某些情况下，可以把一个或多个继承的成员恢复为其最初的访问声明。例如，即使某个基类以private方式被继承，你可能也想在派生类中准许某些成员保持其在基类中的公有成员状态。在标准C++中有两种方法可以达此目的。第一：可以使用using语句，这是首选方法。using语句最初是被设计来支持名字空间的，本书将在第23章对此进行讨论。恢复一个继承成员访问声明的第二种方法是在派生类内部使用一个访问声明。虽然访问声明现在被标准C++支持，但并不提倡这种方法。也就是说，编写新代码时不应该采用这种方法。由于目前仍有许多现有程序采用访问声明，所以我们将在这里对其进行剖析。

访问声明的一般形式如下：

base-class::member;

访问声明放在派生类声明中适当的访问标题（access heading）下面。注意，访问声明中不需要（也可以说不允许）类型声明。要想了解访问声明的工作原理，首先看一看下面这段小程序：

```
class base {
public:
  int j; // public in base
};

// Inherit base as private.
class derived: private base {
public:

  // here is access declaration
  base::j; // make j public again
  .
  .
  .
};
```

因为 base 被 derived 以 private 方式继承，所以公有成员 j 成为 derived 的私有成员。然而，通过在 derived 的 public 标题下进行如下访问声明，可以把 j 恢复为公有状态：

```
base::j;
```

可以利用访问声明恢复公有成员和保护成员的访问权限，但是却不能利用访问声明提高或降低某成员的访问状态。例如，在基类中声明为私有的成员不能被派生类改为公有成员（如果C++允许这种事情发生，就会破坏它的封装机制）。

下面的程序说明了访问声明的使用方法，从中可以看到它是怎样利用访问声明把 j, seti( ) 和 geti( ) 恢复为 public 状态的。

```
#include <iostream>
using namespace std;

class base {
  int i; // private to base
```

```
public:
  int j, k;
  void seti(int x) { i = x; }
  int geti() { return i; }
};

// Inherit base as private.
class derived: private base {
public:
  /* The next three statements override
     base's inheritance as private and restore j,
     seti(), and geti() to public access. */
  base::j; // make j public again - but not k
  base::seti; // make seti() public
  base::geti; // make geti() public

// base::i; // illegal, you cannot elevate access

  int a; // public
};

int main()
{
  derived ob;

//ob.i = 10; // illegal because i is private in derived

  ob.j = 20; // legal because j is made public in derived
//ob.k = 30; // illegal because k is private in derived

  ob.a = 40; // legal because a is public in derived
  ob.seti(10);

  cout << ob.geti() << " " << ob.j << " " << ob.a;

  return 0;
}
```

C++支持访问声明以满足以下情况：即继承类的大多数成员是私有的，而少数成员继续保持其公有或保护状态。

注意： 虽然标准C++仍然支持访问声明，但并不提倡这种做法。因为现在虽然允许使用访问声明，但它们在将来或许就不被支持了。相反，标准建议通过使用关键字 using 达到同样的效果。

16.6 虚基类

当继承多个基类时，可以把一个具有二义性的元素引入到C++程序中。例如，看一看下面这个错误程序：

```
// This program contains an error and will not compile.
#include <iostream>
using namespace std;

class base {
public:
```

```
  int i;
};

// derived1 inherits base.
class derived1 : public base {
public:
  int j;
};

// derived2 inherits base.
class derived2 : public base {
public:
  int k;
};

/* derived3 inherits both derived1 and derived2.
   This means that there are two copies of base
   in derived3! */
class derived3 : public derived1, public derived2 {
public:
  int sum;
};

int main()
{
  derived3 ob;

  ob.i = 10; // this is ambiguous, which i???
  ob.j = 20;
  ob.k = 30;

  // i ambiguous here, too
  ob.sum = ob.i + ob.j + ob.k;

  // also ambiguous, which i?
  cout << ob.i << " ";

  cout << ob.j << " " << ob.k << " ";
  cout << ob.sum;

  return 0;
}
```

正如该程序中注释语句说明的那样，derived1 和 derived2 都继承 base，而 derived3 既继承 derived1，也继承 derived2。也就是说，在 derived3 类型的对象中有两个 base 的副本，那么表达式 ob.i = 10 中的 i 引用的是 derived1 中的呢？还是 derived2 中的呢？因为在对象 ob 中有两个 base 的副本，所以也有两个 ob.i 可见，这个语句具有二义性。

上述程序有两种补救方法。第一种方法是对 i 使用范围运算符并手工选择一个 i。例如，下面的程序将如预期的那样编译和运行：

```
// This program uses explicit scope resolution to select i.
#include <iostream>
using namespace std;

class base {
```

```
public:
  int i;
};

// derived1 inherits base.
class derived1 : public base {
public:
  int j;
};

// derived2 inherits base.
class derived2 : public base {
public:
  int k;
};

/* derived3 inherits both derived1 and derived2.
   This means that there are two copies of base
   in derived3! */
class derived3 : public derived1, public derived2 {
public:
  int sum;
};

int main()
{
  derived3 ob;

  ob.derived1::i = 10; // scope resolved, use derived1's i
  ob.j = 20;
  ob.k = 30;

  // scope resolved
  ob.sum = ob.derived1::i + ob.j + ob.k;

  // also resolved here
  cout << ob.derived1::i << " ";

  cout << ob.j << " " << ob.k << " ";
  cout << ob.sum;

  return 0;
}
```

可以看到，因为使用了::，该程序手工选择了base的derived1版本。然而，这种方法提出了一个更深入的问题：如果实际上只需要一个base的副本将怎么样呢？有没有某种方法可以防止在derived3中包括两个副本呢？或许你已经猜到了，答案是有。这种解决方案可通过使用虚基类实现。

当两个或多个对象源自一个公共基类时，可以通过在继承基类时把该基类声明为virtual来避免在一个源自这些对象的对象中出现多个基类副本。要达到这个目的，可以在继承基类时在该基类的前面加上关键字virtual。例如，下面是上述范例程序的另一个版本，其中的derived3只包含base的一个副本：

```
// This program uses virtual base classes.
#include <iostream>
using namespace std;

class base {
public:
  int i;
};

// derived1 inherits base as virtual.
class derived1 : virtual public base {
public:
  int j;
};

// derived2 inherits base as virtual.
class derived2 : virtual public base {
public:
  int k;
};

/* derived3 inherits both derived1 and derived2.
   This time, there is only one copy of base class. */
class derived3 : public derived1, public derived2 {
public:
  int sum;
};

int main()
{
  derived3 ob;

  ob.i = 10; // now unambiguous
  ob.j = 20;
  ob.k = 30;

  // unambiguous
  ob.sum = ob.i + ob.j + ob.k;

  // unambiguous
  cout << ob.i << " ";

  cout << ob.j << " " << ob.k << " ";
  cout << ob.sum;

  return 0;
}
```

可以看到，关键字 virtual 位于其他继承的类声明之前。既然 derived1 和 derived2 都将 base 继承为 virtual，那么任何涉及到它们的多重继承将只产生 base 的一个副本。因此，在 derived3 中，只有 base 的一个副本，语句 ob.i = 10 即正确又明确。

还要记住一点：即使 derived1 和 derived2 都将 base 指定为 virtual，base 仍然在每一种类型的对象中出现。例如，下面的程序段是完全正确的：

```
// define a class of type derived1
derived1 myclass;

myclass.i = 88;
```

普通基类和虚基类之间的惟一不同在于：当一个对象多次继承基类时将出现何种情况。如果使用虚基类，则在该对象中只出现一个基类，否则，将出现多个基类副本。