第19章

组合与继承

程序代码的再利用（reuse）是程序开发的重要方法。以面向对象(object-oriented)的方式设计程序时，类（class）的再利用有组合（composition）和继承（inheritance）两种重要的方式。

19.1 既有类的再利用

优良的程序设计方式是渐进、累加的，而不是一切重新撰写。本章的目的在于介绍如何利用已开发好且功能经过确认的类 (class)，以创造新的类。

要正确地利用既有的类，其关键在于不对它做任何的更动，而将它原原本本地当做新类的基础。最直接的做法有两种：

1. 在新类内使用既有类所定义的对象 (object)。
   以 18.5 节的电梯类为例，我们可以在定义大厦 (building) 类时，将电梯作为新类内的成员对象。这种做法称为组合（composition）。

2. 承接既有类的结构，并将其中的成员进行修改或加入新的成员。
   再以 18.5 节的电梯类为例，我们可以在定义新型式的电梯类时，继承原有的电梯类，并以加入紧急暂停，取消语音说明的方式产生新的类。这种做法称为继承（inheritance）。

继承为面向对象程序设计（OOP）的重要基础，我们在下一章中将继续探讨它所派生的问题和解决之道。

19.2 组合(composition)

对象的组合方式由于其“不可或缺”的程度不同，可以分为以下两种：

- 组件—零件（assembly-parts）：例如，汽车与引擎之间的关系。
- 聚集（aggregation，collection）：例如，班级与同学之间的关系。

在“组件—零件”的组合方式之中，如果少了一个零件，就无法构成完整的组件。而由“聚集”构成的集合，即使少了几个成员，也对集合本身的功能没有影响。本节所述的组合，主要是关系比较紧密的“组件—零件”隶属方式。

事实上，我们在第 18 章介绍类时，已用到组合的语法。只是我们之前用来构造类成员的是 C++ 的内置数据类型而不是自定的类，例如 `int`， `char` 等。我们将发现，使用自定的类在结构上并无不同。

首先考虑一个叫做 **Component** 的类 (本节类的成员函数都非常简单，只涉及数据成员的存取，因此成员函数的定义都直接写在类的声明里面，成为 inline 成员函数)：

```
class Component
{
  private:
    int I;
  public:
     // 构造函数使用“构造函数初始列”的语法
     Component (): I(1)        {}  // 默认构造函数
     Component (int N): I(N) {}  // 构造函数
     ~Component()              {}  // 析构函数
     int Get() const   {return I;}
     void Double()     {I*=2;}
};
```

构造函数初始列

在定义构造函数 (constructor) 时，可以同时完成初始值设定的动作。C++ 提供了一个很简便的语法，称为构造函数初始列 (constructor initializer list)。在构造函数的参数列之后加上冒号“:”，接着各个子对象的构造函数调用式，然后才是构造函数的本体。例如在上述类 Component 的声明中，构造函数的定义：

```
Component (): I(1)       {}
Component (int N): I(N) {}
```

接着，我们将使用两个由类 Component 定义的对象：C1 和 C2，作为新定义的类 Host 的 private 子对象。此外，我们也定义了一个叫做 C3 的 Component 对象作为 public 子对象。在使用组合的方式定义新类时，可以同时对数据成员和子对象进行初始值设定的动作。例如，在类 Host 的声明中，构造函数的定义可以写成：

```
Host() : k(1), C1(1), C2(1), C3(1)  {}
Host(int L, int M, int N, int P)
       : k(L), C1(M), C2(N), C3(P)  {}
```

若是实现部分独立在类的声明之外、则写成：

```
Host::Host() : k(1), C1(1), C2(1), C3(1)  {}
Host::Host(int L, int M, int N, int P)
              : k(L), C1(M), C2(N), C3(P)  {}
```

以下是类 Host 的完整声明：

```
//------ 声明类 Host ------------------
class Host
{
  private:
    int k;
    Component C1, C2;  // 子对象 C1, C2
  public:
    Component C3;      // 子对象 C3
    // 类 Host 的构造函数，使用构造函数初始列的语法
    Host() : k(1), C1(1), C2(1), C3(1)  {}
    Host(int L, int M, int N, int P)
           : k(L), C1(M), C2(N), C3(P)  {}
    ~Host()          {}   // 类 Host 的析构函数
    int  Get() const  {return k;}
    void Double()     {k*=2;}
    void DoubleComp()
      {C1.Double(); C2.Double(); C3.Double();}
    int  GetC1()      {return C1.Get();}
    int  GetC2()      {return C2.Get();}
};
```

这三个嵌入在类 Host 内的对象 C1，C2 和 C3 称为子对象 （sub-objects），由于 C1 和 C2 被指定为 private，要取用它们的成员函数时，必须使用标准的成员运算符号(member operator) “.”。 例如 GetC1()的定义语句：

```
int GetC1() { return C1.Get();}
```

如果我们使用类 Host 定义一个叫做 H 的对象：

```
Host H;
```

因为 C3 是 public 子对象，因此可以直接通过 C3 取用它的成员函数：

```
H.C3.Get();
```

我们把上述类 **Component** 和类 **Host** 的声明置于文件 CompClass.h 中，并以主程序 CompMain.cpp 使用类 **Host** 定义对象 H，以执行下述接口：

H.Get()	取用 H 的 private 数据成员 k
H.GetC1()	取用 H 的 private 子对象 C1 的 private 数据成员 I
H.GetC2()	取用 H 的 private 子对象 C2 的 private 数据成员 I
H.C3.Get();	取用 H 的 public 子对象 C3 的 private 数据成员 I
H.Double()	H 的接口，用来将 H 的 private 数据成员 k 乘以 2
H.DoubleComp()	H 的接口，用来调用各个子对象的接口 C1.Double()，C2.Double()，和 C3.Double()

以下是完整的程序及类 Host 的使用方式：

范例程序　文件 CompMain.cpp

```
// CompMain.cpp
#include "CompClass.h"
// ------ 主程序 --------------------
int main()
{
    Host H(1,2,3,4);
    cout << "执行 "Host H(1,2,3,4)" 之后: " << endl;
    cout << "H.Get()    = "   << H.Get()     << endl;
    cout << "H.GetC1() = "   << H.GetC1()   << endl;
    cout << "H.GetC2() = " << H.GetC2()     << endl;
```

```
    cout << "H.C3.Get()= " << H.C3.Get()    << endl;
    H.Double();
    cout << "执行 "H.Double()"     之后: "    << endl;
    cout << "H.Get()   = "   << H.Get()      << endl;
    cout << "H.GetC1() = "   << H.GetC1()    << endl;
    cout << "H.GetC2() = "   << H.GetC2()    << endl;
    cout << "H.C3.Get()= "   << H.C3.Get()   << endl;
    H.DoubleComp();
    cout << "执行 "H.DoubleComp()" 之后: "   << endl;
    cout << "H.Get()    = " << H.Get()      << endl;
    cout << "H.GetC1()  = " << H.GetC1()    << endl;
    cout << "H.GetC2()  = " << H.GetC2()    << endl;
    cout << "H.C3.Get() = " << H.C3.Get()   << endl;
  return 0;
}
```

程序执行结果

```
执行 "Host H(1,2,3,4)" 之后:
H.Get()   = 1
H.GetC1() = 2
H.GetC2() = 3
H.C3.Get()= 4
执行 "H.Double()"     之后:
H.Get()   = 2
H.GetC1() = 2
H.GetC2() = 3
H.C3.Get()= 4
执行 "H.DoubleComp()" 之后:
H.Get()   = 2
H.GetC1() = 4
H.GetC2() = 6
H.C3.Get()= 8
```

19.3　组合对象的构造函数和析构函数

当一个对象初定义时，不只是定义这个对象的构造函数被调用，它所包含的子对象之构造函数也会自动被调用.

为了追踪各个构造函数和析构函数被调用的次序，我们改写 CompClass.h，成为文件 CompClass2.h，让它们被调用时都会通过 cout 显示出来：

范例程序　文件 CompClass2.h

```
// CompClass2.h
#ifndef COMPCLASS2_H
#define COMPCLASS2_H
#include <iostream>
using std::cin;
using std::cout;
using std::endl;
using std::cerr;

//---- 声明类 Component --------
class Component
{
  private:
    int I;
  public:
    // 类 Component 的构造函数
    Component (): I(1)
     {cout << "调用 Component 默认构造函数" << endl; }
    Component (int N): I(N)
     {cout << "调用 Component 构造函数" << endl;}
    // 类 Component 的析构函数
    ~Component()
     {cout << "调用 Component 析构函数" << endl;}
    int Get() const   {return I;}
    void Double()     {I*=2;}
```

```
};

// ---- 声明类 Host ---------------------------
class Host
{
  private:
    int k;
    Component C1, C2;
  public:
    Component C3;
    // 类 Host 的构造函数
    Host() : k(1), C1(1), C2(1), C3(1)
      {cout << "调用 Host 默认构造函数" << endl; }
    Host(int L, int M, int N, int P)
          : k(L), C1(M), C2(N), C3(P)
      {cout << "调用 Host 构造函数" << endl; }
    // 类 Host 的析构函数
    ~Host() {cout << "调用 Host 析构函数" << endl;}
    int  Get() const  {return k;}
    void Double()     {k*=2;}
    void DoubleComp()
      {C1.Double(); C2.Double(); C3.Double();}
    int  GetC1()      {return C1.Get();}
    int  GetC2()      {return C2.Get();}
};

#endif
```

同时简化 19.2 节的主程序 CompMain.cpp 成为文件 CompMain2.cpp：

范例程序　文件 CompMain2.cpp

```
// CompMain2.cpp
#include "CompClass2.h"
int main()
{
  Host H(1,2,3,4);
  cout << "执行 sizeof(H)的结果是: "
       << sizeof(H) / sizeof(int)
       << " 个 int 变量." << endl;
  return 0;
}
```

程序执行结果

```
调用 Component 构造函数
调用 Component 构造函数
调用 Component 构造函数
调用 Host 构造函数
执行 sizeof(H)的结果是: 4 个 int 变量.
调用 Host 析构函数
调用 Component 析构函数
调用 Component 析构函数
调用 Component 析构函数
```

可见三个子对象的构造函数先被调用，其次才是对象本身的构造函数。而析构函数被调用的次序则和构造函数被调用的次序颠倒。

此外，使用 sizeof() 函数，我们知道每个由 Host 定义的对象都含有 k，以及 C1，C2 和 C3 所各自包含的一个名叫 I 的 int 变量，总共 4 个 int 变量。

19.4 继承 (inheritance)

使用组合的语法非常直接，只是把类当成数据类型来使用即可，而继承的使用则必须使用新的语法。

当我们说某一个新类继承某一既有类时，表示这个新类具有既有类的所有成员，同时对既有类的成员作了修改，或是增添了新的成员。我们称这个既有的类为基类（base class）或超类（super-class），而继承下来的新类为派生类（derived class）或子类（subclass）。

例如，“哺乳类”是“猫科”的基类，而“猫科”是“哺乳类”的派生类。这个继承的概念基本上和我们在中学时所学的林奈的分类法，将生物分成界、门、纲、目、科、属、种七个阶层的想法是一致的。

使用面向对象的类继承图（class inheritance diagram），可以把上述关系表示成图 19.4.1 的样子：

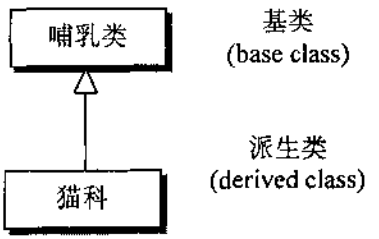


图 19.4.1 最简单的类继承图

图 19.4.2 中空白箭头的方向从派生类指向基类。这个方向代表的是一般化（generalization）的趋势，而其相反方向则为特殊化（specialization）的趋势。

有时候继承的关系可以很复杂。譬如蝙蝠就同时继承了哺乳类和飞行动物的特性：

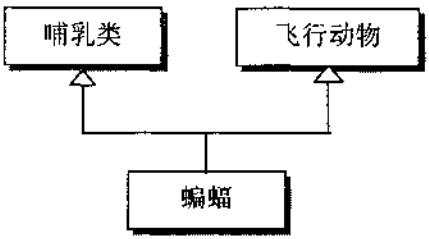


图 19.4.2　多重继承

当一个派生类具有不只一个基类时，称为多重继承（multiple inheritance）。

假设有一个名叫 Base 的基类：

```
class Base
{
 private:
   int i;
 public:
   Base(): i(3)          {}
   Base(int N): i(N) {}
   ~Base()               {}
   void Set(int N)   {i=N;}
   int  Get() const  {return i;}
   void Double()     {i*=2;}
   void Triple()     {i*=3;}
};
```

派生类 Derived 在声明时，必须在类名称和类的本体之间加上冒号“:”、 public 标记，以及基类名称。例如：

```
class Derived : public Base
{
 private:
   int i;
 public:
```

```
    Derived(): i(5)   {}
    Derived(int M, int N) : Base(M), i(N)     {}
    ~Derived()            {}
    void Set(int N)       {i=N;}
    void SetBase(int N)   {Base::Set(N);}
    void Double()         {i*=2;}
    int  Get()      const {return i;}
    int  GetBase() const  {return Base::Get();}
};
```

由于在类的语法里，默认的关系是 private，因此必须使用 public 来标记派生类和基类之间的关系，否则原来在基类内的 public 成员也会变为 private，无法在派生类中取用。

成员函数的重新定义（redefinition of member functions）

为了以下的讨论方便，我们先将基类 Base 和派生类 Derived 两个类中的成员列表如下：

	基　类	派生类
private 成员	int i	int i
public 成员	Base()	
	Base(int)	
	void **Set**(int)	void **Set**(int)
	int **Get**()	int **Get**()
	void **Double**()	void **Double**()
	void Triple()	
		Derived()
		Derived(int, int)
		void SetBase(int)
		int GetBase(int)
	void Multiply ()	
	void **Multiply**(int)	void **Multiply**(int)

在上表中，我们将基类和派生类间同名的成员函数放在同一列，左右对齐．这是因为我们在派生类中将这些成员函数重新定义．然而，这种覆盖式的重新定义法并不适用于数据成员．这一点可以使用标准函数 sizeof()检查基类和派生类的大小得知．

假如我们使用类 Derived 来定义一个名叫 X 的对象：

```
Derived X;
```

则

```
X.Set(2);
X.Get();
X.Double();
```

所调用的都是类 Derived 中新定义的成员函数。由于 Triple() 没有被重新定义，因此

```
X.Triple();
```

所调用的是类 Base 中原有的成员函数。

重载成员函数的重新定义 (redefining an overloaded member function)

在类 Base 中，下列两个成员函数

```
void Multiply();
void Multiply(int);
```

具有重载（overloading）的关系。如果我们在派生类中重新定义被重载的成员函数，例如：

```
int Multiply(int);
```

即使这时候返回值不一样（原来是 void，新的成员函数是 int），基类中的所有同名成员函数也都同时被覆盖，无法从派生类所定义的对象调用。例如：

```
cout << X.Multiply(5);
```

调用的是派生类中的

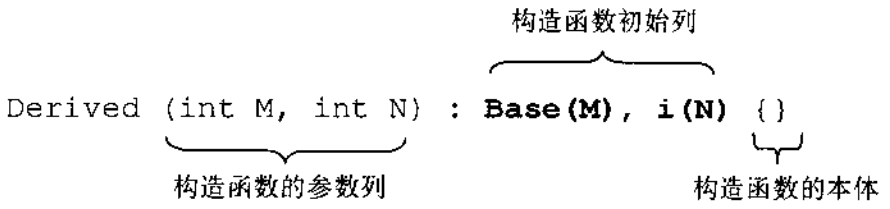
```
int Multiply(int)
```

而原有的基类中两个函数 Multiply() 再也无法从 X 调用。这种情况称为名称隐藏（name hiding）。

构造函数初始列 (constructor initializer list)

和 19.2 节使用组合（composition）的语法非常相似，我们可以在派生类的构造函数中给定派生类自己的数据成员和基类的数据成员初始值。

派生类的构造函数初始列也是介于构造函数的参数列和构造函数的本体之间．例如：



在构造函数初始列中，派生类自己的数据成员可以直接在名称之后加上小括号及其初始值，例如上式中的:

```
i(N)
```

而基类的数据成员的初始值则以调用基类的构造函数来给定初值，例如上式中的：

```
Base(M)
```

此外，我们也可以将派生类自己数据成员初始化的动作放在构造函数的本体内：

```
Derived(int M, int N) : Base(M) { i=N; }
```

在派生类的成员函数中调用基类的成员函数

在本例中，Get() 和 Set() 两个在基类中的成员函数被派生类 Derived 重新定义，因此我们在派生类中另外写了 GetBase()和 SetBase()两个成员函数去取用在类 Base 中的成员函数。

这时候，我们在定义 GetBase()和 SetBase()时，必须在函数名称之前加上基类的名称和范围确认运算符号（scope resolution operator）“::”。例如：

```
void SetBase(int N) { Base::Set(N);}
int GetBase() const { return Base::Get();}
```

我们把上述类 **Base** 和类 **Derived** 的声明置于文件 B&D.h 中，并在主程序 B&D.cpp 中使用类 **Derived** 定义对象 X，Y，以执行下述接口：

X.GetBase()	调用 Base::Get(), 以取用基类的 private 数据成员 i
X.Get()	取用派生类的 private 数据成员 i
X.Triple()	将派生类的 private 数据成员 i 乘以 3
Y.GetBase()	调用 Base::Get(), 以取用基类的 private 数据成员 i
Y.Get()	取用派生类的 private 数据成员 i

以下是完整的程序及类 **Derived** 的使用方式：

范例程序　文件 B&D.cpp

```
// B&D.cpp
#include "B&D.h"
main()
{
  cout << "sizeof(Base)/sizeof(int)   : "
       << sizeof(Base)/sizeof(int)       << endl;
  cout << "sizeof(Derived)/sizeof(int): "
       << sizeof(Derived)/sizeof(int)        << endl;
  cout << "执行 "Derived X" 之后: "          << endl;
  Derived X;
  cout << "X.GetBase():   " << X.GetBase()   << endl;
  cout << "X.Get()    :   " << X.Get()       << endl;
  X.SetBase(7);
  X.Set(9);
  Cout << "执行 "X.SetBase(7)" 和 "X.Set(9)" 之后: "
       << endl;
  cout << "X.GetBase():   " << X.GetBase()   << endl;
  cout << "X.Get()    :   " << X.Get()       << endl;
  X.Double();
  Cout << "执行 "X.Double()" 之后: " << endl;
```

```
    Cout << "X.GetBase():   " << X.GetBase()   << endl;
    Cout << "X.Get()    :   " << X.Get()       << endl;
    X.Triple();
    Cout << "执行 "X.Triple()" 之后: " << endl;
    cout << "X.GetBase():   " << X.GetBase()   << endl;
    cout << "X.Get()    :   " << X.Get()       << endl;
    Derived Y(12,25);
    cout << "执行 "Derived Y(12,25)" 之后: "   << endl;
    cout << "Y.GetBase():   " << Y.GetBase()   << endl;
    cout << "Y.Get()    :   " << Y.Get()       << endl;
}
```

程序执行结果

```
sizeof(Base)/sizeof(int)   : 1
sizeof(Derived)/sizeof(int): 2
执行 "Derived X" 之后:
X.GetBase():   3
X.Get()    :   5
执行 "X.SetBase(7)" 和 "X.Set(9)" 之后:
X.GetBase():   7
X.Get()    :   9
执行 "X.Double()" 之后:
X.GetBase():   7
X.Get()    :   18
执行 "X.Triple()" 之后:
X.GetBase():   21
X.Get()    :   18
执行 "Derived Y(12,25)" 之后:
Y.GetBase():   12
Y.Get()    :   25
```

19.5　protected 成员

到目前为止，类内成员的存取开放程度分为 **private**(私用的) 和 **public**(公用的) 两类。被设定为 private 的成员只能被同一个类内的成员函数取用，不对外开放。而 public 成员则完全开放给程序内的所有函数和语句。除了 private 和 public 外，C++ 还有第三个存取设定关键词（access specifier）：**protected**(受保护的)。“protected”的作用在一般情况下和 private 相同，但对于派生类却是开放的。也就是说，被设定为 protected 的成员只对自己所属的类，以及其派生类中的成员函数开放。

以本章 Base 和 Derived 两个类为例，由于我们没有直接使用 Base 去定义对象，所以 Base 中的 public 改成 protected 不会影响这个程序的执行结果。有趣的是，把类 Derived 中的 private 改成 protected 也不会改变 Base 和 Derived 这两个类间的存取限制。

19.6　派生类所定义的对象的构造和析构次序

当我们使用派生类定义对象时，不只是派生类的构造函数被调用，基类之构造函数也会自动被调用。

为了追踪各个构造函数和析构函数被调用的次序，我们改写 19.4 节的 B&D.h，成为文件 B&D2.h，让它们被调用时都会通过 cout 显示出来：

范例程序　文件 B&D2.h

```
// B&D2.h
#ifndef B&D2_H
#define B&D2_H
#include <iostream>
using std::cin;
using std::cout;
using std::endl;
using std::cerr;

//---- 声明类 Base --------
```