第13章　继承和组合

C++最重要的性能之一是代码重用。但是，为了具有可进化性，我们应当能够做比拷贝代码更多的工作。

在C的方法中，这个问题未能得到很好的解决。而用 C++，可以用类的方法解决，通过创建新类重用代码，而不是从头创建它们，这样，我们可以使用其他人已经创建并调试过的类。

关键是使用类而不是更改已存在的代码。这一章将介绍两种完成这件事的方法。第一种方法是很直接的：简单地创建一个包含已存在的类对象的新类，这称为组合，因为这个新类是由已存在类的对象组合的。

第二种方法更巧妙，创建一个新类作为一个已存在类的类型，采取这个已存在类的形式，对它增加代码，但不修改它。这个有趣的活动被称为继承，其中大量的工作由编译器完成。继承是面向对象程序设计的基石，并且还有另外的含义，将在下一章中探讨。

对于组合和继承（感觉上，它们都是由已存在的类型产生新类型的方法），它们在语法上和行为上是类似的。这一章中，读者将学习这些代码重用机制。

13.1　组合语法

实际上，我们一直都在用组合创建类，只不过我们是在用内部数据类型组合新类。其实使用用户定义类型组合新类同样很容易。

考虑下面这个在某种意义上很有价值的类：

```
//: USEFUL.H -- A class to reuse
#ifndef USEFUL_H_
#define USEFUL_H_

class X {
  int i;
  enum { factor = 11 };
public:
  X() { i = 0; }
  void set(int I) { i = I; }
  int read() const { return i; }
  int permute() { return i = i * factor; }
};
#endif // USEFUL_H_
```

在这个类中，数值成员是私有的，所以对于将类型 X 的一个对象作为公共对象嵌入到一个新类内部，是绝对安全的。

```
//: COMPOSE.CPP -- Reuse code with composition
#include "..\12\useful.h"

class Y {
```

```
  int i;
public:
  X x; // Embedded object
  Y() { i = 0; }
  void f(int I) { i = I; }
  int g() const { return i; }
};

main() {
  Y y;
  y.f(47);
  y.x.set(37); // Access the embedded object
}
```

访问嵌入对象（称为子对象）的成员函数只须再一次选择成员。

如果嵌入的对象是private的，可能更具一般性，这样，它们就变成了内部实现的一部分（这意味着，如果我们愿意，我们可以改变这个实现）。而对于新类的public接口函数，包含对嵌入对象的使用，但不必模仿这个嵌入对象的接口。

```
//: COMPOSE2.CPP -- Private embedded objects
#include "..\12\useful.h"
class Y {
  int i;
  X x; // Embedded object
public:
  Y() { i = 0; }
  void f(int I) { i = I; x.set(I); }
  int g() const { return i * x.read(); }
  void permute() { x.permute(); }
};

main() {
  Y y;
  y.f(47);
  y.permute();
}
```

这里，permute()函数的执行调用了X的接口，而X的其他成员函数也在Y的成员函数中被调用。

13.2 继承语法

组合的语法是明显的，而完成继承，则有新的不同的形式。

继承也就是说“这个新的类像那个老的类”。通过给出这个类的名字，在代码中声明继承，在这个类体的开括号前面，加一冒号和基类名（或加多个类，对于多重继承）。这样做，就自动地得到了基类中的所有数据成员和成员函数。下面是一个例子：

在Y中，我们可以看到继承，它意味着 Y 将包含 X 中的所有数据成员和成员函数。实际

```
//: INHERIT.CPP -- Simple inheritance
#include "..\12\useful.h"
#include <iostream.h>
class Y : public X {
  int i; // Different from X's i
public:
  Y() { i = 0; }
  int change() {
    i = permute(); // Different name call
    return i;
  }
  void set(int I) {
    i = I;
    X::set(I); // Same-name function call
  }
};

main() {
  cout << "sizeof(X) = " << sizeof(X) << endl;
  cout << "sizeof(Y) = "
       << sizeof(Y) << endl;
  Y D;
  D.change();
  // X function interface comes through:
  D.read();
  D.permute();
  // Redefined functions hide base versions:
  D.set(12);
}
```

上，Y包含了 X 的一个子对象，就像在 Y 中创建 X 的一个成员对象，而不从 X 继承一样。无论成员对象还是基类存储，都被认为是子对象。在main()，可以看到这些数据成员已被加入了，因为sizeof(Y）是sizeof(X)的 2 倍大。

我们将注意到，基类由public处理，否则，在继承中，所有被继承的东西都是 private，也就是说在基类中的所有public成员在派生类中都是private。这当然不是所希望的，希望的结果是保持基类中的public成员在派生类中也是public。这可在继承期间用关键字public做到。

在change()中，基类permute()函数被调用，派生类对所有的public基类函数都有直接的访问权。

在派生类中的set()函数重定义了基类中的set()函数。这就是，如果对于类型 Y 的对象调用函数read()和permute()，得到的是这些函数的基类版本（可以在 main()中看到表现）。但如果对于对象 Y 调用 set()，得到的是重定义的版本。这意味着，如果我们不喜欢在继承中得到某个函数的基类版本，可以改变它（还能够增加全新的函数，例如 change()）。

然而，当重定义函数时，我们可能希望仍然保留基类版本。如果简单地调用 set()，得到的是这个函数的本地版本（一个递归函数调用）。为了调用基类版本，我们必须用准确名，即使

用基类名和范围分解运算符。

13.3 构造函数的初始化表达式表

我们已经看到，在C++中保证合适的初始化表达式多么重要，在组合和继承中，也是一样。当创建一个对象时，必须保证编译器调用所有子对象的构造函数。到目前为止，例子中的所有子对象都有缺省的构造函数，编译器可以自动调用它们。但是，如果子对象没有缺省构造函数或如果我们想改变某个构造函数的缺省参数，情况会怎么样呢？这是一个问题，因为这个新类的构造函数不能保证访问它的子对象的private数据成员，所以不能直接地对它们初始化。

解决办法很简单：对于子对象调用构造函数，C++为此提供了专门的语法，即构造函数的初始化表达式表。构造函数的初始化表达式表的形式模仿继承活动。对于继承，我们在冒号之后和这个类体的左括号之前放置基类。而在构造函数的初始化表达式表中，我们可以将对子对象构造函数的调用语句放在构造函数参数表和冒号之后，在函数体的开括号之前。对于从 bar 继承来的类 foo，如果 bar 有一个取单个int参数的构造函数，则表示为

```
foo::foo(int i) : bar(i) { //...
```

13.3.1 成员对象初始化

当使用组合时，对于成员对象初始化使用相同的语法是不可行的。对于组合，给出对象的名字而不是类名。如果在初始化表达式表中有多于一个构造函数调用，应当用逗号隔开：

```
foo2:foo2(int I) : bar(i), memb(i+1) { // ...
```

这是类 foo2 构造函数的开头，它是从 bar 继承来的，包含称为 memb 的成员对象。注意，当我们在这个构造函数的初始化表达式表中能看到基类的类型时，只能看到成员对象的标识符。

13.3.2 在初始化表达式表中的内置类型

构造函数的初始化表达式表允许我们显式地调用成员对象的构造函数。事实上，这里没有其它方法调用那些构造函数。主要思想是，在进入新类的构造函数体之前调用所有的构造函数。这样，对子对象的成员函数所做的任何调用都已经转到了这个被初始化的对象中。没有对所有的成员对象和基类对象的构造函数调用，就没有办法进入这个构造函数的左括号，即便是编译器也必须对缺省构造函数做隐藏调用。这是 C++进一步的强制，以保证没有调用它的构造函数就没有对象（或对象的部分）能进入第一道门。

所有的成员对象在构造函数的左括号之前被初始化的思想是编程时一个方便的辅助方法。一旦遇到左括号，我们能假设所有的子对象已被适当地初始化了，并集中精力在希望该构造函数完成的特殊任务上。然而，这里还有一个问题：内置类型的嵌入对象如何？它没有构造函数吗？

为了让语法一致，允许对待内置类型就像对待有单个构造函数的对象一样，它取单个参数：这个参数与我们正在初始化的变量类型相同。这样，我们就可以写：

```
class X {
  int i;
  float f;
  char c;
  char* s;
public:
```

```
  X() : i(7), f(1.4), c('x'), s("howdy") {}
  // ...
```

这些“伪构造函数调用”是为了完成简单的赋值。它是传统的技术和好的编码风格，所以常常能看到它被使用。

甚至在类之外创建这种类型的变量时，我们也可能用伪构造函数语法：

```
int i(100);
```

这使得内置类型的效果更像对象。

记住，这些并不真的是构造函数，特别是，如果没有显式地进行伪构造函数调用，就不会进行初始化。

13.4 组合和继承的联合

当然，我们还可以把两者放在一起使用。下面的例子中这个更复杂类的创建就使用了继承和组合两种方法。

```
//: COMBINED.CPP -- Inheritance & composition

class A {
  int i;
public:
  A(int I) { i = I; }
  ~A() {}
  void f() const {}
};

class B {
  int i;
public:
  B(int I) { i = I; }
  ~B() {}
  void f() const {}
};

class C : public B {
  A a;
public:
  C(int I) : B(I), a(I) {}
  ~C() {} // Calls ~A() and ~B()
  void f() const {  // Redefinition
    a.f();
    B::f();
  }
};

main() {
  C c(47);
}
```

C 继承 B 并且有一个成员对象（这是类 A的对象）。我们可以看到，构造函数的初始化表达式表中调用了基类构造函数和成员对象构造函数。

函数C::f() 重定义了它所继承的B::f()，并且还调用基类版本。另外，它还调用了a.f()。注意，只能在继承期间重定义函数。通过成员对象，只能操作这个对象的公共接口，而不能重定义它。另外，如果C::f() 还没有被定义，则对类型C 的一个对象调用 f() 不会调用 a.f()，而是调用 B::f()。

• 自动析构函数调用

虽然常常需要在初始化表达式表中做显式构造函数调用，但我们决不需要做显式的析构函数调用，因为对于任何类型只有一个析构函数，并且它并不取任何参数。然而，编译器仍保证所有的析构函数被调用，这意味着，在整个层次中的所有析构函数，从最底层的析构函数开始调用，一直到根层。

构造函数和析构函数与众不同之处在于每一层函数都被调用，这是值得强调的。然而对于一般的成员函数，只是这个函数被调用，而不是任意基类版本被调用。如果还想调用一般成员函数的基类版本，必须显式地调用。

13.4.1 构造函数和析构函数的次序

当一个对象有许多子对象时，知道构造函数和析构函数的调用次序是有趣的。下面的例子明显表明它如何工作：

```
//: ORDER.CPP -- Constructor/destructor order
#include <fstream.h>
ofstream out("order.out");

#define CLASS(ID) class ID { \
public: \
  ID(int) { out << #ID " constructor\n"; } \
  ~ID() { out << #ID " destructor\n"; } \
};

CLASS(base1);
CLASS(member1);
CLASS(member2);
CLASS(member3);
CLASS(member4);

class derived1 : public base1 {
  member1 m1;
  member2 m2;
public:
  derived1(int) : m2(1), m1(2), base1(3) {
    out << "derived1 constructor\n";
  }
  ~derived1() {
    out << "derived1 destructor\n";
  }
};
```

```
class derived2 : public derived1 {
  member3 m3;
  member4 m4;
public:
  derived2() : m3(1), derived1(2), m4(3) {
    out << "derived2 constructor\n";
  }
  ~derived2() {
    out << "derived2 destructor\n";
  }
};

main() { derived2 d2; }
```

首先，创建 ofstream 对象，以发送所有输出到一个文件中。为了在书中少敲一些字符也为了演示一种宏技术（这个技术将在第 18章中被一个更好的技术代替），这里使用了宏以建立一些类（这些类将被用于继承和组合）。每个构造函数和析构函数向这个跟踪文件报告它们自己的行动。注意，这些是构造函数，而不是缺省构造函数，它们每一个都有一整型参数。这个参数本身没有标识符，它的唯一的任务就是强迫在初始化表达式表中显式调用这些构造函数。（消除标识符防止编译器警告信息）这个程序的输出是：

```
base1 constructor
member1 constructor
member2 constructor
derived1 constructor
member3 constructor
member4 constructor
derived2 constructor
derived2 destructor
member4 destructor
member3 destructor
derived1 destructor
member2 destructor
member1 destructor
base1 destructor
```

可以看出，构造在类层次的最根处开始，而在每一层，首先调用基类构造函数，然后调用成员对象构造函数。调用析构函数则严格按照构造函数相反的次序 —— 这是很重要的，因为要考虑潜在的相关性。另一有趣的是，对于成员对象，构造函数调用的次序完全不受在构造函数的初始化表达式表中次序的影响。该次序是由成员对象在类中声明的次序所决定的。如果能通过构造函数的初始化表达式表改变构造函数调用次序，那么就会对两个不同的构造函数有二种不同的调用顺序。而析构函数不可能知道如何为析构函数相应地反转调用次序，这就引起了相关性问题。

13.4.2 名字隐藏

如果在基类中有一个函数名被重载几次，在派生类中重定义这个函数名会掩盖所有基类版

本，这也就是说，它们在派生类中变得不再可用。

```
//: HIDE.CPP -- Name hiding during inheritance

class homer {
public:
  int doh(int) const { return 1; }
  char doh(char) const { return 'd';}
  float doh(float) const { return 1.0; }
};

class bart : public homer {
public:
  class milhouse {};
  void doh(milhouse) const {}
};

main() {
  bart b;
//! b.doh(1); // Error
//! b.doh('x'); // Error
//! b.doh(1.0); // Error
}
```

因为 bart 重定义了 doh()，这些基类版本中没有一个是对于 bart 对象可调用的。这时,编译器试图变换参数成为一个 milhouse 对象，并报告出错，因为它不能找到这样的变换。正如在下面章节中会看到的，更普遍的方法是用与在基类中严格相同的符号重定义函数并且返回类型也与基类中的相同。

13.4.3 非自动继承的函数

不是所有的函数都能自动地从基类继承到派生类中的。构造函数和析构函数是用来处理对象的创建和析构的，它们只知道对在它们的特殊层次的对象做什么。所以，在整个层次中的所有的构造函数和析构函数都必须被调用，也就是说，构造函数和析构函数不能被继承。

另外，operator= 也不能被继承，因为它完成类似于构造函数的活动。这就是说，尽管我们知道如何由等号右边的对象初始化左边的对象的所有成员，但这并不意味着这个初始化在继承后仍有意义。在继承过程中，如果我们不亲自创建这些函数，编译器就综合它们。（通过构造函数，我们不能对缺省构造函数和被自动创建的拷贝构造函数创建任何构造函数）这在第 11 章中已经简要地讲过了。被综合的构造函数使用成员方式的初始化，而被综合的 operator= 使用成员方式的赋值。这是由编译器创建而不是继承的函数的例子。

```
//: NINHERIT.CPP -- Non-inherited functions
#include <fstream.h>
ofstream out("ninherit.out");

class root {
public:
```

```
  root() { out << "root()\n"; }
  root(root&) { out << "root(root&)\n"; }
  root(int) { out << "root(int)\n"; }
  root& operator=(const root&) {
    out << "root::operator=()\n";
    return *this;
  }
  class other {};
  operator other() const {
    out << "root::operator other()\n";
    return other();
  }
  ~root() { out << "~root()\n"; }
};

class derived : public root {};

void f(root::other) {}

main() {
  derived d1;  // Default constructor
  derived d2 = d1; // Copy-constructor
//! derived d3(1); // Error: no int constructor
  d1 = d2; // Operator= not inherited
  f(d1); // Type-conversion IS inherited
}
```

所有的构造函数和 operator= 都自我宣布，所以我们能知道编译器何时使用它们。另外，operator other() 完成自动类型变换，从 root 对象到被嵌入的类 other 的对象。类 derived 直接从 root 继承，并没有创建函数（观察编译器如何反应）。函数 f() 取一个 other 对象以测试这个自动类型变换函数。

在 main() 中，创建缺省构造函数和拷贝构造函数，调用 root 版本作为构造函数调用继承的一部分，尽管这看上去像是继承，但新的构造函数实际上是创建的。正如我们所预料的，自动创建带参数的构造函数是不可能的，因为这样太依赖编译器的直觉。

在 derived 中，operator=( ) 也被综合为新函数，使用成员函数赋值，因为这个函数在新类中不显式地写出。

关于处理对象创建的重写函数的所有这些原则，我们也许会觉得奇怪，为什么自动类型变换运算也能被继承。但其实这不足为奇——如果在 root 中有足够的块建立一个 other 对象，那么从 root 派生出的任何东西中，这些块仍在原地，类型变换当然也就仍然有效。（尽管实际上我们可能想重定义它）

13.5 组合与继承的选择

无论组合还是继承都能把子对象放在新类型中。两者都使用构造函数的初始化表达式表去构造这些子对象。现在我们可能会奇怪，这两者之间到底有什么不同？该如何选择？

组合通常在希望新类内部有已存在类性能时使用，而不希望已存在类作为它的接口。这就是说，嵌入一个计划用于实现新类性能的对象，而新类的用户看到的是新定义的接口而不是来自老类的接口。为此，在新类的内部嵌入已存在类的 private 对象。

有时，允许类用户直接访问新类的组合是有意义的，这就让成员对象是 public。成员函数隐藏它们自己的实现，所以，当用户知道我们正在装配一组零件并且使得接口对他们来说更容易理解时，这样会安全的。Car 对象是一个很好的例子：

```
//: CAR.CPP -- Public composition

class engine {
public:
  void start() const {}
  void rev() const {}
  void stop() const {}
};

class wheel {
public:
  void inflate(int psi) const {}
};

class window {
public:
  void rollup() const {}
  void rolldown() const {}
};

class door {
public:
  window Window;
  void open() const {}
  void close() const {}
};

class car {
public:
  engine Engine;
  wheel Wheel[4];
  door left, right; // 2-door
};

main() {
  car Car;
  Car.left.Window.rollup();
  Car.Wheel[0].inflate(72);
}
```

因为小汽车的组合是分析这个问题的一部分（不是基本设计的部分），所以让成员是公共的有助于客户程序员理解如何使用这个类，而且能使类的创建者有更小的代码复杂性。

稍加思考就会看到，用车辆对象组合小汽车是无意义的——小汽车不能包含车辆，它本身就是一种车辆。这种is-a关系用继承表达，而 has-a 关系用组合表达。

13.5.1 子类型设置

现在假设想创建包含 ifstream 对象的一个类，它不仅打开一个文件，而且还保存文件名。这时我们可以使用组合并把 ifstream 及 strstream 都嵌入这个新类中：

```
//: FNAME1.CPP -- An fstream with a file name
#include <fstream.h>
#include <strstrea.h>
#include "..\allege.h"

class fname1 {
  ifstream File;
  enum { bsize = 100 };
  char buf[bsize];
  ostrstream Name;
  int nameset;
public:
  fname1() : Name(buf, bsize), nameset(0) {}
  fname1(const char* filename)
    : File(filename), Name(buf, bsize) {
      allegefile(File);
      Name << filename << ends;
      nameset = 1;
  }
  const char* name() const { return buf; }
  void name(const char* newname) {
    if(nameset) return; // Don't overwrite
    Name << newname << ends;
    nameset = 1;
  }
  operator ifstream&() { return File; }
};
main() {
  fname1 file("fname1.cpp");
  cout << file.name() << endl;
  // Error: rdbuf() not a member:
//!  cout << file.rdbuf() << endl;
}
```

然而这里存在一个这样的问题：我们也许想通过包含一个从 fname1 到ifstream &的自动类型转换运算，在任何使用 ifstream 的地方都使用 fname1 对象，但在 main 中，

```
cout<<file.rdbuf()<<endl;
```

这一行不能编译，因为自动类型转换只发生在函数调用中，而不在成员选择期间。所以，此时这个方法不行。

第二个方法是对 fname1 增加 rdbuf() 定义：

```
filebuf * rdbuf()  {return File.rdbuf();}
```

如果只有很少的函数从 ifsream 类中拿来，这是可行的。在这种情况下，我们只是使用了这个类的一部分，并且组合是适用的。

但是，如果我们希望这个类的东西都进来，应该做什么呢？这称为子类型设置，因为正在由一个已存在的类做一个新类，并且希望这个新类与已存在的类有严格相同的接口（希望增加任何我们想要加入的其他成员函数），所以能在已经用过这个已存在类的任何地方使用这个新类，这就是必须使用继承的地方。我们可以看到，子类型设置很好地解决了先前例子中的问题。

```
//: FNAME2.CPP -- Subtyping solves the problem
#include <fstream.h>
#include <strstrea.h>
#include "..\allege.h"
class fname2 : public ifstream {
  enum { bsize = 100 };
  char buf[bsize];
  ostrstream Name;
  int nameset;
public:
  fname2() : Name(buf, bsize), nameset(0) {}
  fname2(const char* filename)
    : ifstream(filename), Name(buf, bsize) {
      Name << filename << ends;
      nameset = 1;
  }
  const char* name() const { return buf; }
  void name(const char* newname) {
    if(nameset) return; // Don't overwrite
    Name << newname << ends;
    nameset = 1;
  }
};

main() {
  fname2 file("fname2.cpp");
  allegefile(file);
  cout << "name: " << file.name() << endl;
  const bsize = 100;
  char buf[bsize];
  file.getline(buf, bsize); // This works too!
  file.seekg(-200, ios::end);
  cout << file.rdbuf() << endl;
}
```

现在，能与ofstream 对象一起工作的任何成员函数也能与 fname2 对象一起工作。这是因为，fname2 是ofstream的一个类型。不是简单地包含。这是非常重要的问题，将在本章最后和在第14章中讨论。

13.5.2 专门化

继承也就是取一个已存在的类，并制作它的一个专门的版本。通常，这意味着取一个一般目的的类并为了特殊的需要对它专门化。

例如，考虑前面章中的 stack 类，与这个类有关的问题之一是必须每次完成计算，从包容器中引出一个指针。这不仅乏味，而且不安全——我们能让这个指针指向所希望的任何地方。

较好的方法是使用继承来专门化这个一般的 stack 类。这里有一个例子，它使用来自前一章的类。

```
//: INHSTAK.CPP -- Specializing the stack class
#include "..\11\stack11.h"
#include "..\11\strings.h"
#include <fstream.h>
#include "..\allege.h"

class Stringlist : public stack {
public:
  void push(String* str) {
    stack::push(str);
  }
  String* peek() const {
    return (String*)stack::peek();
  }
  String* pop() {
    return (String*)stack::pop();
  }
};

main() {
  ifstream file("inhlist.cpp");
  allegefile(file);
  const bufsize = 100;
  char buf[bufsize];
  Stringlist textlines;
  while(file.getline(buf,bufsize))
    textlines.push(String::make(buf));
  String* s;
  while((s = textlines.pop()) != 0) // No cast!
    cout << *s << endl;
}
```

两个头文件STRINGS.H（第12.2.1节）和 STACK11.H （第12.2.3章）都从第12章引来

（STACK11.OBJ 文件也必须连进来。）

stringlist专门化stack ，所以push()只接受string指针。在此之前，stack会接受void指针，所以用户没有类型检查以确保插入合适的指针。另外， peek()和pop()现在返回string指针，而不返回void指针，所以使用这个指针时不必映射。

令人惊奇的是，额外的类型安全性检查是无需代价的！编译器给出额外的类型信息，这些只在编译时使用，而这些函数是内联的，并不产生另外的代码。

不幸的是，继承并不能解决所有这些与包容器类有关的问题，析构函数仍然引起麻烦。我们也许会记得，在第12章中，stack::~stack()析构函数遍历整个表，并对所有的指针调用 delete。问题是，对于void指针调用delete，它只释放这块内存而不调用析构函数（因为 void没有类型信息）。可以创建一个stringlist::~stringlist()析构函数，它能遍历这个表并对所有在表中的 string指针调用delete，这样，问题就解决了。条件是：

1) stack数据成员被定义为protected，使得这个stringlist析构函数能访问它们。（protected在本章稍后介绍。）

2) 移去stack基类析构函数，使得这块内存不会两次被释放。

3) 不执行更多的继承，否则会再次面临相同的两难境地：要么调用多重析构函数，要么进行不正确的析构函数调用（可能包含string对象而不是从stringlist派生出的类）。

这个问题将在下一章重述，但直到第15章介绍了模板后才能得到完全解决。

13.5.3 私有继承

通过在基类表中去掉 public 或者通过显式地声明private，可以私有地继承基类（后者可能是更好的策略，因为可以让用户明白它的含义）。当私有继承时，创建的新类有基类的所有数据和功能，但这些功能是隐藏的，所以它只是内部实现部分。该类的用户访问不到这些内部功能，并且一个对象不被看作这个基类的成员（如在第 13.5.1节中的FNAME2.CPP中的）。

我们可能奇怪， private 继承的目的是什么，因为在这个新类中选择创建一个 private对象似乎更合适。将private继承包含在该语言中只是为了语言的完整性。但是，如果没有其他理由，则应当减少混淆，所以通常建议用 private成员而不是private继承。然而，这里可能偶然有这种情况，即可能想产生像基类接口一样的接口，而不允许处理该对象像处理基类对象一样。private继承提供了这个功能。

• 对私有继承成员公有化

当私有继承时，基类的所有 public成员都变成了private。如果希望它们中的任何一个是可视的，只要用派生类的public选项声明它们的名字即可。

```
//: PRIVINH.CPP -- Private inheritance

class base1 {
public:
  char f() const { return 'a'; }
  int g() const { return 2; }
  float h() const { return 3.0; }
};

class derived : base1 { // Private inheritance
public:
```

```
  base1::f; // Name publicizes member
  base1::h;
};

main() {
  derived d;
  d.f();
  d.h();
//! d.g(); // Error -- private function
}
```

这样，如果想要隐藏这个类的基类部分的功能，则private继承是有用的。

在我们使用private继承取代对象成员之前，应当注意到，当与运行类型标识相连时，私有继承有特定的复杂性。（第18章内容。）

13.6 保护

关键字protected对于继承有特殊的意义。在理想世界中，private成员总是严格私有的，但在实际项目中，有时希望某些东西隐藏起来，但仍允许其派生类的成员访问。于是关键字protected派上了用场。它的意思是：“就这个类的用户而言，它是private的，但它可被从这个类继承来的任何类使用。”

数据成员最好是private，因为我们应该保留改变内部实现的权利。然后我们才能通过保护成员函数控制对该类的继承者的访问。

```
//: PROTECT.CPP -- The protected keyword
#include <fstream.h>

class base {
  int i;
protected:
  int read() const { return i; }
  void set(int I) { i = I; }
public:
  base(int I = 0) : i(I) {}
  int value(int m) const { return m*i; }
};

class derived : public base {
  int j;
public:
  derived(int J = 0) : j(J) {}
  void change(int x) { set(x); }
};

main() {}
```

在附录C中的SSHAPE例子中，我们可以看到需要protected的很好的例子。

被保护的继承

继承时，基类缺省为private，这意味着所有public成员函数对于新类的用户是private的。通常我们都会让继承public，从而使得基类的接口也是派生类的接口。然而在继承期间，也可以使用protected关键字。

被保护的派生意味着对其他类来“照此实现”，但对派生类和友元是“is-a”。它是不常用的，它的存在只是为了语言的完整性。

13.7 多重继承

既然我们已可以从一个类继承，那么我们也就应该能同时从多个类继承。实际上这是可以做到的，但是否它象设计部分一样有意义仍是一个有争议的话题。不过有一点是可以肯定的：直到我们已经很好地学会程序设计并完全理解这门语言时，我们才能试着用它。这时，我们大概会认识到，不管我们如何认为我们必须用多重继承，我们总是能通过单重继承来完成。

起初，多重继承似乎很简单，在继承期间，只需在基类表中增加多个类，用逗号隔开。然而，多重继承有很多含糊的可能性，这就是为什么第16章要讨论这一主题的原因。

13.8 渐增式开发

继承的优点之一是它支持渐增式开发，它允许我们在已存在的代码中引进新代码，而不会给原代码带来错误，即使产生了错误，这个错误也只与新代码有关。也就是说当我们继承已存在的功能类并对其增加数据成员和成员函数（并重定义已存在的成员函数）时，已存在类的代码并不会被改变，更不会产生错误。

如果错误出现，我们就会知道它肯定是在我们的新派生代码中。相对于修改已存在代码体的做法来说，这些新代码很短也很容易读。

相当奇怪的是，这些类如何清楚地被隔离。为了重用这些代码，甚至不需要这些成员函数的源代码，只需要表示类的头文件和目标文件或带有已编译成员函数的库文件。（对于继承和组合都是这样。）

认识到程序开发是一个渐增过程，就象人的学习过程一样，这是很重要的。我们能做尽可能多的分析，但当开始一个项目时，我们仍不可能知道所有的答案。如果开始把项目作为一个有机的、可进化的生物来“培养”，而不是完全一次性的构造它，使之像一个玻璃盒子式的摩天大楼，那么我们就会获得更大的成功和更直接的反馈。

虽然继承对于实验是有用的技术，但在事情稳定之后，我们需要用新眼光重新审视一下我们的类层次，把它看成可感知的结构。记住，继承首先表示一种关系，其意为：“新类是老类的一个类型。”我们的程序不应当关心怎样摆布比特位，而应当关心如何创建和处理各类型的对象，以便用问题的术语表示模型。

13.9 向上映射

在这一章的前面，我们已经看到了由ofstream派生而来的类的对象如何有ofstream对象所有的特性和行为。在13.5.1节中FNAME2.CPP中，任何ofstream成员函数应当能被fname2对象调用。

继承的最重要的方面不是它为新类提供了成员函数，而在于它是基类与新类之间的关系描述：“新类是已存在类的一个类型”。

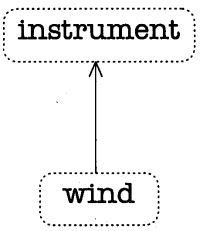
这个描述不仅仅是一种解释继承的方法——它直接由编译器支持。例如，考虑称为

instrument的基类（它表示乐器）和派生类wind，因为继承意味着在基类中的所有函数在派生类中也是可行的，可以发送给基类的消息也可以发送给这个派生类，所以，如果 instrument类有play()成员函数，那么wind 也有。这意味着，我们可以确切地说， wind是instrument 的一个类型。下面的例子表明编译器是如何支持这个概念的。

```
//: WIND.CPP -- Inheritance & upcasting
enum note { middleC, Csharp, Cflat }; // Etc.

class instrument {
public:
  void play(note) const {}
};

// Wind objects are instruments
// because they have the same interface:
class wind : public instrument {};

void tune(instrument& i) {
  // ...
  i.play(middleC);
}

main() {
  wind flute;
  tune(flute); // Upcasting
}
```

在这个例子中，有趣的是tune()函数，它接受instrument参数。然而，在main()中，tune()函数的调用却被传递了一个wind参数 。我们可能会感到奇怪， C++对于类型检查应该是非常严格的，而接受某个类型的函数为什么会这么容易地接受另一个类型。直到人们认识到 wind对象也是一个instrument对象，tune()函数能对instrument 调用，也能对wind调用时，才会恍然大悟。在tune()中，这些代码对instrument和从instrument派生来的任何类型都有效，这种将 wind的对象、引用或指针转变成instrument对象、引用或指针的活动称为向上映射。

13.9.1 为什么“向上映射”



这个术语引入的是有其历史原因的，而且它也与类继承图的传统画法有关：在顶部是根，向下长（当然我们可以用任何我们认为方便的方法画我们的图）。对于WIND.CCP的继承如右图

从派生类到基类的映射，在继承图中是上升的，所以一般称为向上映射。向上映射总是安全的。因为是从更专门的类型到更一般的类型——对于这个类接口可能出现的唯一的情况是它失去成员函数，不会增加成员函数。这就是编译器允许向上映射不需要显式地说明或做其他标记的原因。

• 向下映射

当然我们也可以实现向上映射的反转，称为向下映射，但是，这涉及到一个两难问题，这是第17章中讨论的主题。

13.9.2 组合与继承

确定应当用组合还是用继承，最清楚的方法之一是询问是否需要新类向上映射。在本章的前面，stack类通过继承被专门化，然而，stringlist对象仅作为string包容器，不需向上映射，所以更合适的方法可能是组合：

```
//: INHSTAK2.CPP -- Composition vs inheritance
#include "..\11\stack11.h"
#include "..\11\strings.h"
#include <fstream.h>
#include "..\allege.h"

class Stringlist {
  stack Stack; // Embed instead of inherit
public:
  void push(String* str) {
    Stack.push(str);
  }
  String* peek() const {
    return (String*)Stack.peek();
  }
  String* pop() {
    return (String*)Stack.pop();
  }
};

main() {
  ifstream file("inhlst2.cpp");
  allegefile(file);
  const bufsize = 100;
  char buf[bufsize];
  Stringlist textlines;
  while(file.getline(buf,bufsize))
    textlines.push(String::make(buf));
  String* s;
  while((s = textlines.pop()) != 0) // No cast!
    cout << *s << endl;
}
```

这个文件与INHSTACK.CPP(13.5.2节中)是一样的，只不过stack对象被嵌入在stringlist内，还有被嵌入对象调用的一些函数也被嵌入在 stringlist内。这里没有时间和空间的开销，因为其子类占用相同量的空间，而且所有另外的类型检查都发生在编译时。

我们也可以用private继承以表示“照此实现”。用以创建stringlist类的方法在这种情况下不是重要的——因为各种方法都能解决问题。然而，当可能存在多重继承时就要注意了，多重继承时可能被警告。在这种情况下，如果能发现一个类组合可以使用，那就不要用继承，因为这样可以消除对多重继承的需要。

13.9.3 指针和引用的向上映射

在WIND.CPP(13.9)中，向上映射发生在函数调用期间——在函数外的wind对象被引用并且变成一个在这个函数内的instrument的引用。

向上映射还能出现在对指针或引用简单赋值期间：

```
wind w;
instrument* ip = &w; // upcast
instrument& ir = w; // upcast
```

和函数调用一样，这两个例子都不要求显式地映射。

13.9.4 危机

当然，任何向上映射都会损失对象的类型信息，如果说

```
wind w;
instrument * ip = &w;
```

编译器只能把ip作为一个instrument指针处理。这就是说，它不知道ip实际上指向wind的对象。所以，当调用play()成员函数时，如果使用

```
ip->play(middleC);
```

编译器只知道它正在对于一个instrument指针调用play()，并调用instrument::play()的基本版本，而不是它应该做的调用wind::play()。这样将会得到不正确的结果。

这是一个重要的问题，将在下一章通过介绍面向对象编程的第三块基石：多态性（在C++中用virtual函数实现）来解决。

13.10 小结

继承和组合都允许由已存在的类型创建新类型，两者都是在新类型中嵌入已存在的类型的子对象。然而，当我们想重用原类型作为新类型的内部实现的话，我们最好用组合，如果我们不仅想重用这个内部实现而且还想重用原来接口的话那就用继承。如果派生类有基类的接口，它就能向上映射到这个基类，这一点对多态性很重要，这将在下一章中讲到。

虽然通过组合和继承进行代码重用对于快速项目开发有帮助，但通常我们会希望在允许其他程序员依据它开发程序之前重新设计类层次。

我们的类层次必须有这样的特性：它的每个类有专门的用途，不能太大（包含太多的功能不利于重用），也不能太小（太小如不对它本身增加功能就不能使用）。而且这些类应当容易重用。

13.11 练习

1. 修改CAR.CPP，使得它也从被称为vehicle的类继承，在vehicle中放置合适的成员函数（也就是说，补充一些成员函数）。对vehicle增加一个非缺省的构造函数，在car的构造函数内部必须调用它。

2. 创建两个类，A和B，带有能宣布自己的缺省构造函数。从A继承出一个新类，称为C，并且在C中创建B的一个成员对象，而不对C创建构造函数。创建类C的一个对象，观察结果。

3. 使用继承，专门化在第12章（PSTASH.H & PSTASH.CPP）中的pstash类，使得它接受和返回String指针。修改PSTEST.CPP并测试它。改变这个类使得pstash是一个成员对象。

4. 使用private和protected继承从基类创建两个新类。然后尝试向上映射这个派生类的对象成为基类。解释所发生的事情。