第16章 多重继承

多重继承MI（multiple inheritance)的基本概念听起来非常简单。

可以通过继承多个基类来生成一个新类。这种语法正如料想的那样，只要继承图简单，MI也不复杂。然而MI会引入许多二义性和奇异的情况，它贯穿于本章。现在首先给出关于本主题概述。

16.1 概述

在C++以前，最为成功的面向对象语言是Smalltalk。Smalltalk从开始建立就成为面向对象的语言。就面向对象来说，Smalltalk是“纯种的”,而C++则是“混血的”，这是因为C++是在C语言上建立的。Smalltalk的一个设计原则是：所有的类应从一个单一层次上进行派生，它们根植于一个基类（称之为Object，它是基于对象层次的模式），在Smalltalk中不从一个已生成类中进行继承就不能生成一个新类，所以在创建新类之前必须学习 Smalltalk类库，这就是为什么得花相当长的时间才能成为Smalltalk熟手的原因。所以Smalltalk class 类层次总是一棵简单的单一树。

Smalltalk的类有相当部分是共同的，例如其中Object的行为和特性必然是相同的，所以几乎不会碰上需要继承多个基类的情况。然而只要我们想一想， C++可以创建任意个继承树，因而对C++语言的逻辑完备性来说，它必须支持多个类的结合，由此要求有多重继承性。

然而，这并不是多重继承必须存在的令人信服的理由。过去有，现在仍然有关于 MI是C++精华的否定意见。AT&T 的cfront2.0 版首先把MI加入语言并对语言做了明显修改。从那以后，许多改变我们编程方法的其他特性（特别如模板）都被加入进去并且降低了 MI的重要性。我们可以把MI当作语言的“次要”特性。

一个最为紧迫的问题是如何驾驭涉及 MI的包容器。假若我们打算创建一个使每个用户都容易使用的包容器，在包容器内使用 void*是一个方法，如pstash 和 stack。而Smalltalk的方法是创建一个包含 Objects的包容器（记住Object是整个Smalltalk层次的基类型），由于Smalltalk的所有东西都是由Object派生的，所以任何含有Objects的包容器都可以包容任何东西，该方法是不错的。

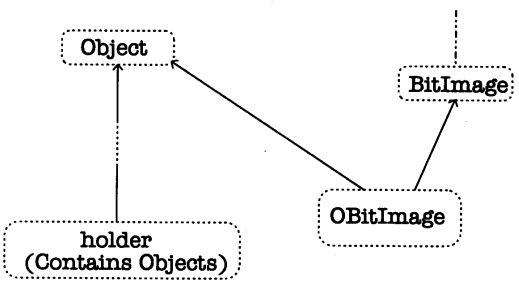

图 16-1

现在考虑C++的情况。假若供应商A创建了一个基于对象的层次结构，该层次包含了一组有用的包容器，其中含有一个打算使用的称之为 holder的包容器。现在我们又发现了供应商 B的类层次，其中有一些重要的其他类，如BitImage类，它可以含有位图。创建一个位图包容器仅有的方法是对BitImage和holder进行继承以创建一个新类，它可拥有BitImage和holder。

对于MI来说，这是一个重要的理由，许多类库都是以这种模式构造的。然而，正如在第15章所看到的，模板的增加已经改变了包容器的生成方法，所以上述情况并不是 MI的有力论

点。

其他需要MI的理由是和设计相关的逻辑上的。和以上情况不同，这里有对基类的控制，同时我们为了使设计更灵活，更有用而采用 MI。这种情况可以以最初的输入输出流的类库设计为例：

输入流和输出流本身都是有用的类，但是通过对它们的继承可将两者的特性和行为加以结合而形成一个新类。

不管使用MI是出于什么样的动机，在处理过程中都会出现许多问题，必须先理解它们而后再使用它们。

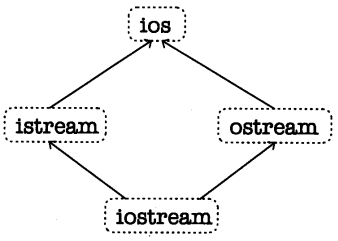

图 16-2

16.2 子对象重叠

当继承基类时，在派生类中就获得了基类所有数据成员的副本，该副本称为子对象。假若对类d1和类d2进行多重继承而形成类 mi，类mi会包含d1的子对象和d2的子对象，所以mi对象看上去如：

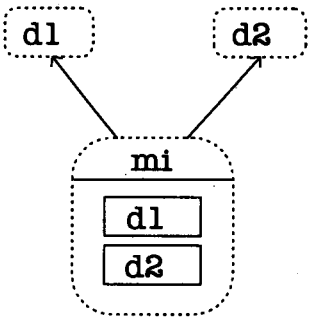

图 16-3

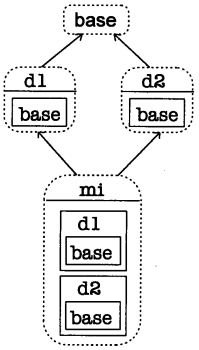

图 16-4

现在考虑如果d1和d2都是从相同基类派生的，该基类称为base，那么会发生什么呢？

在上面的图中，d1和d2都包含base的子对象，所以mi包含基的两个子对象。从继承图形状上看，有时该继承层次结构称为“菱形”。没有菱形情况时，多重继承相当简单，但是只要菱形一出现，由于新类中存在重叠的子对象，麻烦就开始了。重叠的子对象增加了存储空间，这种额外开销是否成为一个问题取决于我们的设计，但它同时又引入了二义性。

16.3 向上映射的二义性

在上图中，假若把一个指向 mi的指针映射给一个指向 base的指针时，将会发生什么呢？base有两个子对象，因此会映射成哪一个的地址呢?这里用代码来揭示上图的问题：

```
//: MULTIPL1.CPP -- MI & ambiguity
#include <iostream.h>
#include "..\14\tstash.h"

class base {
public:
  virtual char* vf() const = 0;
```

```
};

class d1 : public base {
public:
  char* vf() const { return "d1"; }
};

class d2 : public base {
public:
  char* vf() const { return "d2"; }
};

// Causes error: ambiguous override of vf():
//! class mi : public d1, public d2 {};

main() {
  tstash<base> b;
  b.add(new d1);
  b.add(new d2);
  // Cannot upcast: which subobject?:
//!  b.add(new mi);
  for(int i = 0; i < b.count(); i++)
    cout << b[i]->vf() << endl;
}
```

这里存在两个问题。首先，由于d1和d2分别对vf()定义这会导致一个冲突，所以不能生成mi类。其次，在对b[]的数组定义中试图创建一个new mi 并将类型转化为base*，由于没有办法搞清我们打算使用d1子对象的base还是d2子对象的base作为结果地址，所以编译器将不会受理。

16.4 虚基类

为了解决第一个问题，必须对类mi中的函数vf()进行重新定义以消除二义性。

对于第二个问题的解决应着眼于语言扩展，这就是对 virtual赋予新的含义。假若以virtual的方式继承一个基类，则仅仅会出现一个基类子对象。虚基类由编译器的指针法术（ pointer magic）来实现，该方法使人想起普通虚函数的实现。

由于在多重继承期间仅有一个虚基类子对象，所以在地址回溯中不会产生二义性。下面是一个例子：

```
//: MULTIPL2.CPP -- Virtual Base Classes
#include <iostream.h>
#include "..\14\tstash.h"

class base {
public:
```

```
  virtual char* vf() const = 0;
};

class d1 : virtual public base {
public:
  char* vf() const { return "d1"; }
};

class d2 : virtual public base {
public:
  char* vf() const { return "d2"; }
};
// MUST explicitly disambiguate vf():
class mi : public d1, public d2 {
public:
  char* vf() const { return d1::vf();}
};

main() {
  tstash<base> b;
  b.add(new d1);
  b.add(new d2);
  b.add(new mi); // OK
  for(int i = 0; i < b.count(); i++)
    cout << b[i]->vf() << endl;
}
```

现在编译器可以接受地址向上映射了，但是依然要对 mi中的vf()消除二义性，否则，编译器分辨不出需要使用哪一个版本。

16.4.1 “最晚辈派生”类和虚基初始化

虚基类的使用并不如此简单。以上的例子中使用的是编译器生成的缺省构造函数，如果虚基类存在一个构造函数，情况就有些不同了。为了便于理解，引入一个新术语：最晚辈派生类(most-derived)。

最晚辈派生类是当前所在的类，当考虑构造函数时它尤其重要。在前面的例子中，基构造函数里的最晚辈派生类是base；在d1构造函数中，d1是最晚辈派生类；在mi构造函数中，mi是最晚辈派生类。

打算使用一个虚基类时，最晚辈派生类的构造函数的职责是对虚基类进行初始化。这意味着该类不管离虚基类多远，都有责任对虚基类进行初始化。这里有一个初始化的例子：

```
//: MULTIPL3.CPP -- Virtual base initialization
// Virtual base classes must always be
// Intialized by the "most-derived" class
#include <iostream.h>
#include "..\14\tstash.h"
```

```
class base {
public:
  base(int) {}
  virtual char* vf() const = 0;
};

class d1 : virtual public base {
public:
  d1() : base(1) {}
  char* vf() const { return "d1"; }
};

class d2 : virtual public base {
public:
  d2() : base(2) {}
  char* vf() const { return "d2"; }
};

class mi : public d1, public d2 {
public:
  mi() : base(3) {}
  char* vf() const {
    return d1::vf(); // MUST disambiguate
  }
};

class x : public mi {
public:
  // You must ALWAYS init the virtual base:
  x() : base(4) {}
};

main() {
  tstash<base> b;
  b.add(new d1);
  b.add(new d2);
  b.add(new mi); // OK
  b.add(new x);
  for(int i = 0; i < b.count(); i++)
    cout << b[i]->vf() << endl;
}
```

d1和d2在它们的构造函数中都必须对base初始化，这和我们想象的一样。但mi和x也都如此，尽管它们和base相隔好几个层次。这是因为每一个都能成为最晚辈派生类。编译器是无法知道使用d1初始化base还是使用d2的，因而我们总是被迫在最晚辈派生类中具体指出。注意，仅仅这个被选中的虚基构造函数被调用。

16.4.2 使用缺省构造函数向虚基“警告”

为了促进最晚辈派生类初始化一个虚基类，最好通过创建一个虚基类的缺省构造函数，将虚基视为黑箱,如下面的例子。这是由于虚基可能被深深地埋藏在类层次中，它看上去繁琐而令人困惑：

```
//: MULTIPL4.CPP -- "Tying off" virtual bases
// so you don't have to worry about them
// in derived classes
#include <iostream.h>
#include "..\14\tstash.h"
class base {
public:
 // Default constructor removes responsibility:
  base(int = 0) {}
  virtual char* vf() const = 0;
};

class d1 : virtual public base {
public:
  d1() : base(1) {}
  char* vf() const { return "d1"; }
};

class d2 : virtual public base {
public:
  d2() : base(2) {}
  char* vf() const { return "d2"; }
};

class mi : public d1, public d2 {
public:
  mi() {} // Calls default constructor for base
  char* vf() const {
    return d1::vf(); // MUST disambiguate
  }
};

class x : public mi {
public:
  x() {} // Calls default constructor for base
};

main() {
  tstash<base> b;
  b.add(new d1);
```

```
  b.add(new d2);
  b.add(new mi); // OK
  b.add(new x);
  for(int i = 0; i < b.count(); i++)
    cout << b[i]->vf() << endl;
}
```

假若我们总能为虚基类安排缺省构造函数，这可使别人继承我们的类变得非常容易。

16.5 开销

术语“指针法术（pointer magic）”用于描述虚继承的实现。下面的程序可观察到虚继承的物理开销。

```
//: OVERHEAD.CPP -- Virtual base class overhead
#include <fstream.h>
ofstream out("overhead.out");

class base {
public:
  virtual void f() const {};
};

class nonvirtual_inheritance
  : public base {};

class virtual_inheritance
  : virtual public base {};

class virtual_inheritance2
  : virtual public base {};

class mi
  : public virtual_inheritance,
    public virtual_inheritance2 {};
#define WRITE(arg) \
out << #arg << " = " << arg << endl;

main() {
  base b;
  WRITE(sizeof(b));
  nonvirtual_inheritance nonv_inheritance;
  WRITE(sizeof(nonv_inheritance));
  virtual_inheritance v_inheritance;
  WRITE(sizeof(v_inheritance));
  mi MI;
  WRITE(sizeof(MI));
}
```

这些都包含一个单字节，该字节是“内核长度（core size）”。由于所有类都包含虚函数，由于有一个指针，因此对象长度会比内核长度大（起码编译器为了校正定位会把额外的字节添入对象中），然而结果却有点让人吃惊。（该结果来自特定的编译器，不同版本可能有所不同。）

```
sizeof(b)=2
sizeof(nonv_inheritance)=2
sizeof (v_inheritance)=6
sizeof(MI)=12
```

正如所料，b和nonv_inheritance包含有额外指针。但当虚继承时，显示出VPTR和两个额外指针被加了进去。到了多重继承执行时，对象显示出它拥有五个额外指针（然而，指针之一可能是针对第二个多重继承子对象的VPTR。）

这种奇怪的情况可以通过探究我们的特殊实现和考查成员选择的汇编语言，以准确地确定这些额外字节是为什么而设计的以及用多重继承成员选择的开销有多大[1]。总之，虚拟多重继承不过是权益之计，在重视效率的情况下，应该保守地（或避免）使用它。

16.6 向上映射

无论是通过创建成员对象还是通过继承的方式，当我们把一个类的子对象嵌入一个新类中时，编译器会把每一个子对象置于新对象中。当然，每一个子对象都有自己的 this指针，在处理成员对象的时候可以万事俱简。但是只要引入多重继承，一个有趣的现象就会出现：由于对象在向上映射期间出现多个类，因而对象存在多个this指针。下面就是这种情况的例子：

```
//: MITHIS.CPP -- MI and the "this" pointer
#include <fstream.h>
ofstream out("mithis.out");

class base1 {
  char c[0x10];
public:
  void printthis1() {
    out << "base1 this = " << this << endl;
  }
};

class base2 {
  char c[0x10];
public:
  void printthis2() {
    out << "base2 this = " << this << endl;
  }
};

class member1 {
  char c[0x10];
public:
```

[1] 看Jan Gray C++Under the Hood, a chapter in Black Belt C++(edited by Bruce Eckel, M& T press, 1995)。

```
  void printthism1() {
    out << "member1 this = " << this << endl;
  }
};

class member2 {
  char c[0x10];
public:
  void printthism2() {
    out << "member2 this = " << this << endl;
  }
};

class mi : public base1, public base2 {
  member1 m1;
  member2 m2;
public:
  void printthis() {
    out << "mi this = " << this << endl;
    printthis1();
    printthis2();
    m1.printthism1();
    m2.printthism2();
  }
};

main() {
  mi MI;
  out << "sizeof(mi) = "
    << hex << sizeof(mi) << " hex" << endl;
  MI.printthis();
  // A second demonstration:
  base1* b1 = &MI; // Upcast
  base2* b2 = &MI; // Upcast
  out << "base 1 pointer = " << b1 << endl;
  out << "base 2 pointer = " << b2 << endl;
}
```

例子中由于每个类的数组字节数都是用十六进制长度来创建的，所以用十六进制数打出的输出地址是容易阅读的。每个类都有一个打印 this指针的函数，这些类通过多重继承和组合而被装配成类mi，它打印自己和其他所有子对象的地址，由主程序调用这些打印功能。可以清楚地看到，能在一个相同的对象中获得两个不同的 this指针。下面是mi及向上映射到两个不同类的地址输出：

```
sizeof(mi)=40 hex
mi this=0x223e
base1 this=0x223e
```

```
base2 this=0x224e
member1 this=0x225e
member2 this=0x226e
base 1 pointer=0x223e
base 2 pointer=0x224e
```

虽然上述输出根据编译器的不同而有所不同，并且在标准 C++中亦未做详细说明，但它仍具有相当的典型性。派生对象的起始地址和它的基类列表中的第一个类的地址是一致的，第二个基类的地址随后，接着根据声明的次序安排成员对象的地址。

当向base1和base2进行映射时，产生的指针表面上是指向同一个对象，而实际上则必然有不同 this指针，只有这样，固有的起始地址能够被传给相应子对象的成员函数。假若当我们为多重继承的子对象调用一个成员函数时，隐式向上映射就会产生，而只有上述方法才能使程序正确工作。

持久性

由于打算调用与多重继承对象的子对象相关的成员函数，持久性通常并不是一个问题。然而，假若成员函数需要知晓对象的真实起始地址，多重继承就会出现问题。反而言之，多重继承有用的一种情况是拥有持久性。

局部对象的生命周期由其定义确定范围，全局对象的生命周期就是程序的生命周期，持久对象则存在于程序请求之间，通常它被认为存在于磁盘上而非存储器中。面向对象数据库的定义就是“ 一个持久对象集 ”。

为了实现持久性，必须把一个持久对象从磁盘中移入存储器以便为其调用函数，稍后在程序任务完成前还必须把它存入磁盘。把对象存入磁盘涉及四个方面的内容：

1) 必须把对象在存储器中的表示转化成磁盘上的字节序列。

2) 由于存储器中的指针值在下一次程序启用时已毫无意义，所以必须把它们转化成有意义的东西。

3) 指针指向的东西也必须被存储和取回。

4) 当从磁盘到存储器上重组一个对象时，必须考虑对象中的虚指针。

将对象从存储器中转换到磁盘上的过程（把对象写入磁盘）称为串行化（ serialization）；而把对象恢复重组到存储器中的过程称为反串行化（ deserialization）。尽管这些过程十分方便，但由于处理的开销过大而使语言不能直接支持它。类库常常根据增加特定的成员函数以及在新类上设置该需求而支持串行化和反串行化。（通常每个新类都有特定的serialize() 函数）持久性的实现一般不是自动完成的，通常必须对对象进行显式读写。

1. 基于MI的持久性

现在可以考虑跨越指针问题，考虑创建一个使用多重继承把持久性装入简单对象的类。通过继承这个persisence类连同我们的新类，我们可以自动地创建能读写磁盘的类。这听起来很好，但是使用多重继承会引入下例所示的一个缺陷。

```
//: PERSIST1.CPP -- Simple persistence with MI
#include <fstream.h>

class persistent {
  int objSize; // Size of stored object
public:
```

```
  persistent(int sz) : objSize(sz) {}
  void write(ostream& out) const {
    out.write((char*)this, objSize);
  }
  void read(istream& in) {
    in.read((char*)this, objSize);
  }
};
class data {
  float f[3];
public:
  data(float f0 = 0.0, float f1 = 0.0,
    float f2 = 0.0) {
    f[0] = f0;
    f[1] = f1;
    f[2] = f2;
  }
  void print(const char* msg = "") const {
    if(*msg) cout << msg << "   ";
    for(int i = 0; i < 3; i++)
      cout << "f[" << i << "] = "
           << f[i] << endl;
  }
};

class wdata1 : public persistent, public data {
public:
  wdata1(float f0 = 0.0, float f1 = 0.0,
    float f2 = 0.0) : data(f0, f1, f2),
    persistent(sizeof(wdata1)) {}
};

class wdata2 : public data, public persistent {
public:
  wdata2(float f0 = 0.0, float f1 = 0.0,
    float f2 = 0.0) : data(f0, f1, f2),
    persistent(sizeof(wdata2)) {}
};

main() {
  {
    ofstream f1("f1.dat"), f2("f2.dat");
    wdata1 d1(1.1, 2.2, 3.3);
    wdata2 d2(4.4, 5.5, 6.6);
    d1.print("d1 before storage");
```

```
    d2.print("d2 before storage");
    d1.write(f1);
    d2.write(f2);
  } // Closes files
  ifstream f1("f1.dat"), f2("f2.dat");
  wdata1 d1;
  wdata2 d2;
  d1.read(f1);
  d2.read(f2);
  d1.print("d1 after storage");
  d2.print("d2 after storage");
}
```

在上面的简单版本中，persistent::read() 和persistent::write()函数可获取this指针并调用输入输出流的read() 和 write() 函数（注意，任意类型的IO流都可使用）。一个更为复杂的持久性类可能为每个子对象安排虚write()函数。

到目前为止，本书涉及的语言特性尚不能使持久性类知道对象的字节长度，所以在构造函数中插入了一个长度参数。（在第18章中，“运行时类型识别”会揭示怎样找到仅由一个基指针所给定对象的确切类型，一旦获得确切类型，就可以使用 sizeof运算符而求得正确的长度。）

类data不含有指针或VPTR，所以向磁盘写及从磁盘读运算是安全的。类 wdata1在main()中向F1.DAT写入，稍后把数据从文件中取出，没有什么问题。然而当把 persistent置于类wdata2的继承列表中的第二项时，persistent的this指针指向对象的末端，所以读写运算的内容会超出对象的尾部。这样从磁盘文件中读取的内容毫无价值而且会对安排在该对象之后的存储内容造成破坏。

在多重继承中，这种问题是在类必需从子对象的this指针产生实际对象的this指针时发生的。当然，我们知道编辑器是根据继承列表中的类声明次序而安排对象，所以应把重要的类放置在列表的首部（假定只有一个重要类），然而该类可能存在于其他类的继承层次中，这可能会使我们无意识地把该类置于错误的位置上。幸运的是，即使使用了多重继承，在第 18章中介绍的“运行时类型识别”技术也会产生指向实际对象的正确指针。

2. 改良的持久性

下面是一个更具实践化、经常使用的持久性方法的例子。该例子在基类中创建了具有读写功能的虚函数，当类不断派生时，它要求每个新类的创建者能重载这些虚函数。函数的参数是可读出或写入的流对象 [1]。作为类的创建者，我们知道怎样对新的部分进行读写，有责任创建正确的函数调用。该例子没有前面例子的精巧的质量，它要求部分用户有更多的知识和参加更多的编码工作，但该方法并不会由于提交指针而出错。

```
//: PERSIST2.CPP -- Improved MI persistence
#include <fstream.h>
#include <string.h>

class persistent {
public:
```

[1] 有时流只有一个函数，参数包括打算读还是写的信息。

```
  virtual void write(ostream& out) const = 0;
  virtual void read(istream& in) = 0;
};
class data {
protected:
  float f[3];
public:
  data(float f0 = 0.0, float f1 = 0.0,
    float f2 = 0.0) {
    f[0] = f0;
    f[1] = f1;
    f[2] = f2;
  }
  void print(const char* msg = "") const {
    if(*msg) cout << msg << endl;
    for(int i = 0; i < 3; i++)
      cout << "f[" << i << "] = "
           << f[i] << endl;
  }
};

class wdata1 : public persistent, public data {
public:
  wdata1(float f0 = 0.0, float f1 = 0.0,
    float f2 = 0.0) : data(f0, f1, f2) {}
  void write(ostream& out) const {
    out << f[0] << " " << f[1] << " " << f[2];
  }
  void read(istream& in) {
    in >> f[0] >> f[1] >> f[2];
  }
};

class wdata2 : public data, public persistent {
public:
  wdata2(float f0 = 0.0, float f1 = 0.0,
    float f2 = 0.0) : data(f0, f1, f2) {}
  void write(ostream& out) const {
    out << f[0] << " " << f[1] << " " << f[2];
  }
  void read(istream& in) {
    in >> f[0] >> f[1] >> f[2];
  }
};

class conglomerate : public data,
```

```
  public persistent {
    char* name; // Contains a pointer
    wdata1 d1;
    wdata2 d2;
  public:
    conglomerate(const char* nm = "",
      float f0 = 0.0, float f1 = 0.0,
      float f2 = 0.0, float f3 = 0.0,
      float f4 = 0.0, float f5 = 0.0,
      float f6 = 0.0, float f7 = 0.0,
      float f8= 0.0) : data(f0, f1, f2),
      d1(f3, f4, f5), d2(f6, f7, f8) {
      name = new char[strlen(nm) + 1];
      strcpy(name, nm);
    }
    void write(ostream& out) const {
      int i = strlen(name) + 1;
      out << i << " "; // Store size of string
      out << name << endl;
      d1.write(out);
      d2.write(out);
      out << f[0] << " " << f[1] << " " << f[2];
    }
    // Must read in reverse order as write:
    void read(istream& in) {
      delete []name; // Remove old storage
      int i;
      in >> i >> ws; // Get int, strip whitespace
      name = new char[i];
      in.getline(name, i);
    d1.read(in);
    d2.read(in);
    in >> f[0] >> f[1] >> f[2];
  }
  void print() const {
    data::print(name);
    d1.print();
    d2.print();
  }
};

main() {
  {
    ofstream data("data.dat");
    conglomerate C("This is conglomerate C",
      1.1, 2.2, 3.3, 4.4, 5.5,
```

```
      6.6, 7.7, 8.8, 9.9);
    cout << "C before storage" << endl;
    C.print();
    C.write(data);
  } // Closes file
  ifstream data("data.dat");
  conglomerate C;
  C.read(data);
  cout << "after storage: " << endl;
  C.print();
}
```

基类persistent的纯虚函数在派生类中必须被重载以执行正确的读写运算。假若我们已经知道data是持久的，可以直接继承它并在那里重载虚函数，因而可不必使用多重继承。本例的思想是在于我们不拥有data的代码，这些代码在别处已经创建了，它可能是其他类层次的一部分（我们不能控制它的继承）。然而，为了使这个方案能正确地工作，我们必须对下层实现能访问，使得它能被存放，所以我们使用了protected。

类wdata1和wdata2使用了常见的IO流插入器和取出器以便向流对象存储和从流对象取回data的保护性数据。在write()中，我们可以看到每个浮点数后都加了一个空格，这对读入数据的分解是必要的。类conglomerate不仅继承了data，而且拥有两个wdata1和wdata2类型的成员对象及一个字符串指针。另外，由persistent派生的所有类也包含一个VPTR，所以该例子揭示了使用持久性所遇到的一些问题。

当创建write()和read()函数对时，read()函数必须对发生在write()期间的东西进行准确镜像，所以可通过read()抽取磁盘上由write()安置的比特流。这里的第一个问题是char*，它指向一个任意长的字符串。对字符串进行计算并在磁盘上存储其长度，这可使read()函数能正确地分配存储容量。

当拥有的子对象具有read()和write()成员函数时，我们所需要做的是在新的函数read()和write()中调用基类的成员函数。

它的后面跟着基类直接存储的成员函数。

人们在自动持久性方面已竭尽全力。例如，定义类时创建了修改过的预处理器以支持“持久性”主题。你可以想出一个实现持久性更好的方法，但上述方法的优点是它在C++实现下工作，无需特别的语言扩展，相对较健壮。

16.7 避免MI

在PERSIST2.CPP中对多重继承的使用有一点人为的因素，它考虑到在项目中一些类代码不受程序员控制的情况。对以上的例子进行细查，可以看到，通过使用data类型的成员对象以及把虚read()和write()成员放入data或wdata1和wdata2中而不是置于一个独立的类中，这样MI是可以避免使用的。语言会包含一些不常用的特性，这种特殊性只有在其他方法困难或者不可能处理时才使用。当出现是否使用多重继承的问题时，我们可以先问自己两个问题：

1) 我们有必要同时使用两个类的公共接口吗，是否可在一个类中用成员函数包含这些接口呢？

2) 我们需要向上映射到两个基类上吗？（当然，在我们有两个以上的基类被应用。）

如果我们对以上两个问题都能以“不”来回答，那么就可以避免使用 MI。

需要注意，当类仅仅需要作为一个成员参数被向上回溯的情况。在这种情况下，该类可以被嵌入，同时可由新类的自动类型转化运算符产生一个针对被嵌入对象的引用。当将新类的对象作为参数传给希望以嵌入对象为参数的函数时，都将发生类型转换。然而类型转换不能用于通常的成员选择，这时需要继承。

16.8 修复接口

使用多重继承的最好的理由之一是使用控制之外的代码。假定已经拥有了一个由头文件和已经编译的成员函数组成的库，但是没有成员函数的源代码。该库是具有虚函数的类层次，它含有一些使用库中基类指针的全局函数，这就是说它多态地使用库对象。现在，假定我们围绕着该库创建了一个应用程序并且利用基类的多态方式编写了自己的代码。

在随后的项目开发或维护期间，我们发现基类接口和供应商所提供的不兼容：我们所需要的是某虚函数而可能提供的却是非虚的，或者对于解决我们的问题的基本虚函数在接口中根本不存在。假若有源代码则可以返回去修改，但是我们没有，我们有大量的依赖最初接口的已生成的代码，这时多重继承则是极好的解决方法。

下面的例子是所获得的库的头文件：

```
//: VENDOR.H -- Vendor-suppplied class header
// You only get this & the compiled VENDOR.OBJ
#ifndef VENDOR_H_
#define VENDOR_H_

class vendor {
public:
  virtual void v() const;
  void f() const;
  ~vendor();
};

class vendor1 : public vendor {
public:
  void v() const;
  void f() const;
  ~vendor1();
};

void A(const vendor&);
void B(const vendor&);
// Etc.
#endif // VENDOR_H_
```

假定库很大并且有更多的派生类和更大的接口。注意，它包含函数 A()和B()，以基类指针为参数。下面是库的实现文件：

```
//: VENDOR.CPP -- Implementation of VENDOR.H
// This is compiled and unavailable to you
```

```
#include <fstream.h>
#include "..\15\vendor.h"

extern ofstream out; // For trace info

void vendor::v() const {
  out << "vendor::v()\n";
}

void vendor::f() const {
  out << "vendor::f()\n";
}

vendor::~vendor() {
  out << "~vendor()\n";
}

void vendor1::v() const {
  out << "vendor1::v()\n";
}
void vendor1::f() const {
  out << "vendor1::f()\n";
}

vendor1::~vendor1() {
  out << "~vendor1()\n";
}

void A(const vendor& V) {
  // ...
  V.v();
  V.f();
  //...
}

void B(const vendor& V) {
  // ...
  V.v();
  V.f();
  //...
}
```

在我们的项目中，这些源代码是不能得到的，而我们得到的是已编译过的 VENDOR.OBJ或VENDOR.LIB文件（或系统中相应的等价物）。

使用该库会产生问题。首先析构函数不是虚的，这实际上是创建者的一个设计错误。另外，

f()也不是虚的，这可能是库的创建者认为没有必要。但是我们会发现基类接口失去了解决前述问题的必要能力。假若我们已经使用已存在的接口（不包含函数A()和B()，因为它们不受控制）编制了大量代码，而且并不打算改变它。

为了补救该问题，我们可以创建自己的类接口以及从我们的接口和已存在的类中进行多重继承，以便生成一批新类：

```
//: PASTE.CPP -- Fixing a mess with MI
#include <fstream.h>
#include "..\15\vendor.h"

ofstream out("paste.out");

class mybase { // Repair vendor interface
public:
  virtual void v() const = 0;
  virtual void f() const = 0;
  // New interface function:
  virtual void g() const = 0;
  virtual ~mybase() { out << "~mybase()\n"; }
};

class paste1 : public mybase, public vendor1 {
public:
  void v() const {
    out << "paste1::v()\n";
    vendor1::v();
  }
  void f() const {
    out << "paste1::f()\n";
    vendor1::f();
  }
  void g() const {
    out << "paste1::g()\n";
  }
  ~paste1() { out << "~paste1()\n"; }
};

main() {
  paste1& p1p = *new paste1;
mybase& mp = p1p; // Upcast
out << "calling f()\n";
mp.f();  // Right behavior
out << "calling g()\n";
mp.g(); // New behavior
out << "calling A(p1p)\n";
A(p1p); // Same old behavior
```

```
  out << "calling B(p1p)\n";
  B(p1p);  // Same old behavior
  out << "delete mp\n";
  // Deleting a reference to a heap object:
  delete &mp; // Right behavior
}
```

在mybase(它不使用MI)中 ,f()和析构函数都改成虚的，并在接口中增加了新的虚函数 g()。现在，每一个原来的派生类都必须重新创建，采用 MI使其搀入到一个新接口中。函数paste1::v() 和paste1::f()仅需要调用该函数原先基类的版本。但是，如果现在在 main()中对mybase进行向上映射：

```
mybase * mp=plp;// upcast
```

这样，所有函数的调用包括delete都通过多态的mp来完成，同样对新接口函数g()的调用也通过mp。下面是程序的输出：

```
calling f()
paste1::f()
vendor1::f()
calling g()
paste1::g()
calling A(p1p)
paste1::v()
vendor1::v()
vendor::f()
calling B(p1p)
paste1::v()
vendor1::v()
vendor::f()
delete mp
~paste1()
~vendor1()
~vendor()
~mybase()
```

原先的库函数A()和B()仍然可以工作（若新的v()调用它的基类版本）。析构函数现在是虚的并表现了正确的行为。

虽然这是一个散乱的例子，但它确实可以在实际中出现，同时很好地说明了多重继承在何处是必要的：我们必须能够向上映射到两个基类。

16.9 小结

除C++之外，其他OOP语言都不支持多重继承，这是由于针对OOP来说C++是一个“混血”版，它不能像Smalltalk那样把类强制安排成一个单一完整的类层次。 C++支持许多不同形式的继承树，有时需要结合两个或更多的继承树接口形成一个新类。

假若在类层次中没有出现“菱形”形状， MI是相当简单的，尽管必须解决在基类中相同的函数标识。假若出现“菱形”形状，我们必须处理由于引入虚基类导致的子对象重叠问题。

这不仅增加了混乱而且使底层变得更为复杂和缺乏效率。多重继承被 Zack Urlocker称为“九十年代的goto”，因为它确实像goto。在通常编程开发时，应避免使用多重继承，只是在某一时候它才变得非常有用。MI是C++中“次要的”但更为高级的特性，它被设计用于处理特定的情况。如果我们发现经常使用它，应该调查一下使用它的原因，应基于 Occam所提出的简单完美性原则（Occam’s Razor）“我必须向上映射到所有基类吗”，如果我们的回答是否定的，则用嵌入所有基类实例的方法会更容易，而不必使用向上映射。

16.10 练习

1. 本练习会使我们一步一步地穿过 MI陷阱。创建一个含有单个 int参数的构造函数和返回为void型的无参数成员函数f()的基类X。从X派生出Y和Z，为Y和Z各创建一个单个int参数的构造函数。通过多重继承从Y和Z中派生出A。生成一个类A的对象并为对象调用f()。以明显无二义性的方式解决这个问题。

2. 创建一个指向X的指针px，将类型A的对象的地址在它被创建前赋予 px。注意虚基类的使用问题。现在修改X，使得这时不必再在A中为X调用构造函数。

3. 移去f()的明显无二义性说明，观察能否通过px调用f()。对其跟踪以便观察哪个函数被调用。注意这个问题以在一个类层次中调用正确的函数。