第14章　多态和虚函数

多态性（在C++中用虚函数实现）是面向对象程序设计语言继数据抽象和继承之后的第三个基本特征。

它提供了与具体实现相隔离的另一类接口，即把“ what ”从“ how ”分离开来。多态性提高了代码的组织性和可读性，同时也可使得程序具有可生长性，这个生长性不仅指在项目的最初创建期可以“ 生长 ”，而且希望项目具有新的性能时也能“ 生长 ”。

封装是通过特性和行为的组合来创建新数据类型的，通过让细节 private来使得接口与具体实现相隔离。这类机构对于有过程程序设计背景的人来说是非常有意义的。而虚函数则根据类型的不同来进行不同的隔离。上一章，我们已经看到，继承如何允许把对象作为它自己的类型或它的基类类型处理。这个能力很重要，因为它允许很多类型（从同一个基类派生的）被等价地看待就象它们是一个类型，允许同一段代码同样地工作在所有这些不同类型上。虚函数反映了一个类型与另一个类似类型之间的区别，只要这两个类型都是从同一个基类派生的。这种区别是通过其在基类中调用的函数的表现不同来反映的。

在这一章中，我们将从最基本的内容开始学习虚函数，为了简单起见，本章所用的例子经过简化，只保留了程序的虚拟性质。

• C++程序员的进步

C程序员似乎可以用三步进入C++：

第一步：简单地把C++作为一个“ 更好的C ”，因为C++在使用任何函数之前必须声明它，并且对于如何使用变量有更苛刻的要求。简单地用C++编译器编译C程序常常会发现错误。

第二步：进入“ 面向对象 ”的C++。这意味着，很容易看到将数据结构和在它上面活动的函数捆绑在一起的代码组织，看到构造函数和析构函数的价值，也许还会看到一些简单的继承，这是有好处的。许多用过C的程序员很快就知道这是有用的，因为无论何时，创建库时，这些都是要做的。然而在C++中，由编译器来帮我们完成这些工作。

在基于对象层上，我们可能受骗，因为无须花费太多精力就能得到很多好处。它也很容易使我们感到正在创建数据类型——制造类和对象，向这些对象发送消息，一切漂亮优美。

但是，不要犯傻，如果我们停留在这里，我们就失去了这个语言的最重要的部分。这个最重要的部分才真正是向面向对象程序设计的飞跃。要做到这一点，只有靠第三步。

第三步：使用虚函数。虚函数加强类型概念，而不是只在结构内和墙后封装代码，所以毫无疑问，对于新C++程序员，它们是最困难的概念。然而，它们也是理解面向对象程序设计的转折点。如果不用虚函数，就等于还不懂得OOP。

因为虚函数是与类型概念紧密联系的，而类型是面向对象的程序设计的核心，所以在传统的过程语言中没有类似于虚函数的东西。作为一个过程程序员，没有以往的参考可以帮助他思考虚函数，因为接触的是这个语言的其他特征。过程语言中的特征可以在算法层上理解，而虚函数只能用设计的观点理解。

14.1　向上映射

在上一章中，我们已经看到对象如何作为它自己的类型或它的基类的对象使用。另外，它

还能通过基类的地址被操作。取一个对象的地址（或指针或引用），并看作基类的地址，这被称为向上映射，因为继承树是以基类为顶点的。

我们看到会出现一个问题，这表现在下面的代码中：

```
//: WIND2.CPP -- Inheritance & upcasting
#include <iostream.h>
enum note { middleC, Csharp, Cflat }; // Etc.

class instrument {
public:
  void play(note) const {
    cout << "instrument::play" << endl;
  }
};

// Wind objects are instruments
// because they have the same interface:
class wind : public instrument {
public:
  // Redefine interface function:
  void play(note) const {
    cout << "wind::play" << endl;
  }
};

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

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

函数tune()（通过引用）接受一个instrument，但也不拒绝任何从instrument派生的类。在main()中，可以看到，无需映射，就能将wind对象传给tune()。这是可接受的，在instrument 中的接口必然存在于wind中，因为wind是公共的从instrument继承而来的。wind到instrument的向上映射会使wind的接口“变窄”，但它不能变得比instrument的整个接口还小。

这对于处理指针的情况也是正确的，唯一的不同是用户必须显式地取对象的地址，传给这个函数。

14.2 问题

WIND2.CPP的问题可以通过运行这个程序看到，输出是 instrument::play。显然，这不是所

希望的输出，因为我们知道这个对象实际上是 wind而不只是一个instrument。这个调用应当输出wind::play。为此，由instrument派生的任何对象应当使它的play版本被使用。

然而，当对函数用C方法时，WIND2.CPP的表现并不奇怪。为了理解这个问题，需要知道捆绑的概念。

函数调用捆绑

把函数体与函数调用相联系称为捆绑（binding）。当捆绑在程序运行之前（由编译器和连接器）完成时，称为早捆绑。我们可能没有听到过这个术语，因为在过程语言中是不会有的：C编译只有一种函数调用，就是早捆绑。

上面程序中的问题是早捆绑引起的，因为编译器在只有 instrument地址时它不知道正确的调用函数。

解决方法被称为晚捆绑，这意味着捆绑在运行时发生，基于对象的类型。晚捆绑又称为动态捆绑或运行时捆绑。当一个语言实现晚捆绑时，必须有一种机制在运行时确定对象的类型和合适的调用函数。这就是，编译器还不知道实际的对象类型，但它插入能找到和调用正确函数体的代码。晚捆绑机制因语言而异，但可以想象，一些种类的类型信息必须装在对象自身中。稍后将会看到它是如何工作的。

14.3 虚函数

对于特定的函数，为了引起晚捆绑，C++要求在基类中声明这个函数时使用 virtual关键字。晚捆绑只对 virtual起作用，而且只发生在我们使用一个基类的地址时，并且这个基类中有virtual函数，尽管它们也可以在更早的基类中定义。

为了创建一个virtual成员函数，可以简单地在这个函数声明的前面加上关键字 virtual。对于这个函数的定义不要重复，在任何派生类函数重定义中都不要重复它（虽然这样做无害）。如果一个函数在基类中被声明为 virtual，那么在所有的派生类中它都是 virtual的。在派生类中virtual函数的重定义通常称为越位。

为了从WIND2.CPP中得到所希望的结果，只需简单地在基类中的 play()之前增加virtual关键字：

```
//: WIND3.CPP -- Late binding with virtual
#include <iostream.h>
enum note { middleC, Csharp, Cflat }; // Etc.

class instrument {
public:
  virtual void play(note) const {
    cout << "instrument::play" << endl;
  }
};

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

```
  // Redefine interface function:
  void play(note) const {
    cout << "wind::play" << endl;
  }
};

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

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

这个文件除了增加了virtual关键字之外，一切与WIND2.CPP相同，但结果明显不一样。现在的输出是wind::play。

扩展性

通过将play( )在基类中定义为virtual，不用改变tune()函数就可以在系统中随意增加新函数。在一个设计好的 OOP程序中，大多数或所有的函数都沿用 tune()模型，只与基类接口通信。这样的程序是可扩展的，因为可以通过从公共基类继承新数据类型而增加新功能。操作基类接口的函数完全不需要改变就可以适合于这些新类。

现在，instrument例子有更多的虚函数和一些新类，它们都能与老的版本一起正确工作，不用改变tune()函数：

```
//: WIND4.CPP -- Extensibility in OOP
#include <iostream.h>
enum note { middleC, Csharp, Cflat }; // Etc.

class instrument {
public:
  virtual void play(note) const {
    cout << "instrument::play" << endl;
  }
  virtual char* what() const {
    return "instrument";
  }
  // Assume this will modify the object:
  virtual void adjust(int) {}
};

class wind : public instrument {
public:
  void play(note) const {
```

```
    cout << "wind::play" << endl;
  }
  char* what() const { return "wind"; }
  void adjust(int) {}
};

class percussion : public instrument {
public:
  void play(note) const {
    cout << "percussion::play" << endl;
  }
  char* what() const { return "percussion"; }
  void adjust(int) {}
};
class string : public instrument {
public:
  void play(note) const {
    cout << "string::play" << endl;
  }
  char* what() const { return "string"; }
  void adjust(int) {}
};

class brass : public wind {
public:
  void play(note) const {
    cout << "brass::play" << endl;
  }
  char* what() const { return "brass"; }
};

class woodwind : public wind {
public:
  void play(note) const {
    cout << "woodwind::play" << endl;
  }
  char* what() const { return "woodwind"; }
};

// Identical function from before:
void tune(instrument& i) {
  // ...
  i.play(middleC);
}

// New function:
```

```
void f(instrument& i) { i.adjust(1); }

// Upcasting during array initialization:
instrument* A[] = {
  new wind,
  new percussion,
  new string,
  new brass
};

main() {
  wind flute;
  percussion drum;
  string violin;
  brass flugelhorn;
  woodwind recorder;
  tune(flute);
  tune(drum);
  tune(violin);
  tune(flugelhorn);
  tune(recorder);
  f(flugelhorn);
}
```

可以看到，这个例子已在wind之下增加了另外的继承层，但virtual机制正确工作，不管这里有多少层。adjust()函数不对于brass和woodwind重定义。当出现这种情况时，自动使用先前的定义——编译器保证虚函数总是有定义的，所以，决不会最终出现调用不与函数体捆绑的情况。（这种情况将意味着灾难。）

数组A[]存放指向基类instrument 的指针，所以在数组初始化过程中发生向上映射。这个数组和函数f()将在稍后的讨论中用到。

在对tune()的调用中，向上映射在对象的每一个不同的类型上完成。期望的结果总是能得到。这可以被描述为“发送消息给一对象和让这个对象考虑用它来做什么”。virtual函数是在试图分析项目时使用的透镜：基类应当出现在哪里？应当如何扩展这个程序？然而，在程序最初创建时，即便我们没有发现合适的基类接口和虚函数，在稍后甚至更晚，当我们决定扩展或维护这个程序时，我们也常常会发现它们。这不是分析或设计错误，它只意味着一开始我们还没有所有的信息。由于C++严格的模块化，这并不是大问题。因为当我们对系统的一部分作了修改时，往往不会象C那样波及系统的其他部分。

14.4 C++如何实现晚捆绑

晚捆绑如何发生？所有的工作都由编译器在幕后完成。当我们告诉它去晚捆绑时（用创建虚函数告诉它），编译器安装必要的晚捆绑机制。因为程序员常常从理解 C++虚函数机制中受益，所以这一节将详细阐述编译器实现这一机制的方法。

关键字virtual告诉编译器它不应当完成早捆绑，相反，它应当自动安装实现晚捆绑所必须的所有机制。这意味着，如果我们对brass对象通过基类instrument地址调用play()，我们将得到

恰当的函数。

为了完成这件事，编译器对每个包含虚函数的类创建一个表（称为 VTABLE）。在VTABLE中，编译器放置特定类的虚函数地址。在每个带有虚函数的类中，编译器秘密地置一指针，称为vpointer（缩写为VPTR），指向这个对象的VTABLE。通过基类指针做虚函数调用时（也就是做多态调用时），编译器静态地插入取得这个VPTR，并在VTABLE表中查找函数地址的代码，这样就能调用正确的函数使晚捆绑发生。

为每个类设置VTABLE、初始化VPTR、为虚函数调用插入代码，所有这些都是自动发生的，所以我们不必担心这些。利用虚函数，这个对象的合适的函数就能被调用，哪怕在编译器还不知道这个对象的特定类型的情况下。

下面几节将对此做更详细地阐述。

14.4.1 存放类型信息

可以看到，在任何类中，不存在显式的类型信息。而先前的例子和简单的逻辑告诉我们，必须有一些类型信息放在对象中，否则，类型不能在运行时建立。实际上，类型信息被隐藏了。为了看到它，这里有一个例子，可以测试使用虚函数的类的长度，并与没有虚函数的类比较。

```
//: SIZES.CPP -- Object sizes vs. virtual funcs
#include <iostream.h>

class no_virtual {
  int a;
public:
  void x() const {}
  int i() const { return 1; }
};

class one_virtual {
  int a;
public:
  virtual void x() const {}
  int i() const { return 1; }
};
class two_virtuals {
  int a;
public:
  virtual void x() const {}
  virtual int i() const { return 1; }
};

main() {
  cout << "int: " << sizeof(int) << endl;
  cout << "no_virtual: "
       << sizeof(no_virtual) << endl;
  cout << "void* : " << sizeof(void*) << endl;
  cout << "one_virtual: "
```

```
        << sizeof(one_virtual) << endl;
    cout << "two_virtuals: "
        << sizeof(two_virtuals) << endl;
}
```

不带虚函数，对象的长度恰好就是所期望的：单个 int的长度。而带有单个虚函数的one_virtual，对象的长度是no_virtual的长度加上一个void指针的长度。它反映出，如果有一个或多个虚函数，编译器在这个结构中插入一个指针（VPTR）。在one_virtual 和 two_virtuals之间没有区别。这是因为VPTR指向一个存放地址的表，只需要一个指针，因为所有虚函数地址都包含在这个表中。

这个例子至少要求一个数据成员。如果没有数据成员， C++编译器会强制这个对象是非零长度，因为每个对象必须有一个互相区别的地址。如果我们想象在一个零长度对象的数组中索引，我们就能理解这一点。一个“哑”成员被插入到对象中，否则这个对象就有零长度。当virtual关键字插入类型信息时，这个“哑”成员的位置就被占用。在上面例子中，用注释符号将所有类的int a去掉，我们就会看到这种情况。

14.4.2 对虚函数作图

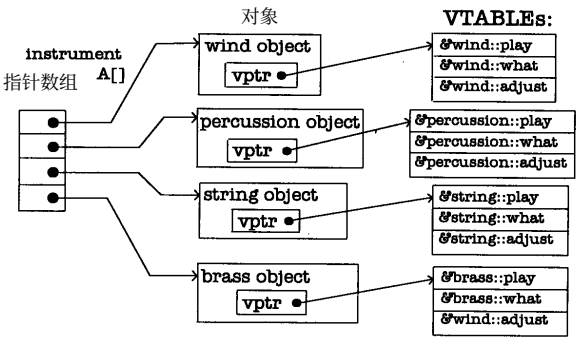

图 14-1

为了准确地理解使用虚函数时编译器做了些什么，使屏风之后进行的活动看得见是有帮助的。这里画的是在 14.3节WIND4.CPP中的指针数组A[]。

这个instrument指针数组没有特殊类型信息，它的每一个元素指向一个类型为instrument 的对象。wind、percussion、string和brass都适合这个范围，因为它们都是从instrument派生来的（并且和instrument有相同的接口和响应相同的消息），因此，它们的地址也自然能放进这个数组里。然而，编译器并不知道它们比instrument对象更多的东西，所以，留给它们自己处理，而通常调用所有函数的基类版本。但在这里，所有这些函数都被用 virtual声明，所以出现了不同的情况。每当创建一个包含有虚函数的类或从包含有虚函数的类派生一个类时，编译器就为这个类创建一个VTABLE，如这个图的右面所示。在这个表中，编译器放置了在这个类中或在它的基类中所有已声明为virtual的函数的地址。如果在这个派生类中没有对在基类中声明为 virtual的函数进行重新定义，编译器就使用基类的这个虚函数地址。（在brass的VTABLE中，adjust的入口就是这种情况。）然后编译器在这个类中放置 VPTR（可在SIZES.CPP中发现）。当使用简单继承时，对于每个对象只有一个VPTR。VPTR必须被初始化为指向相应的VTABLE。（这在构造函数中发生，在稍后会看得更清楚。）

一旦VPTR被初始化为指向相应的VTABLE，对象就“知道”它自己是什么类型。但只有当虚函数被调用时这种自我知识才有用。

通过基类地址调用一个虚函数时（这时编译器没有能完成早捆绑的足够的信息），要特殊处理。它不是实现典型的函数调用，对特定地址的简单的汇编语言 CALL，而是编译器为完成这个函数调用产生不同的代码。下面看到的是通过 instrument指针对于brass调用adjust()。

instrument引用产生如下结果：

编译器从这个instrument指针开始，这个指针指向这个对象的起始地址。所有的 instrument对象或由 instrument派生的对象都有它们的VPTR，它在对象的相同的位置（常常在对象的开头），所以编译器能够取出这个对象的VPTR。VPTR指向VTABLE的开始地址。所有的VTABLE有相同的顺序，不管何种类型的对象。play()是第一个，what()是第二个，adjust()是第三个。所以编译器知道adjust()函数必在VPTR+2处。这样，不是“以instrument::adjust地址调用这个函数”（这是早捆绑，是错误活动），而是产生代码，“在VPTR+2处调用这个函数”。因为VPTR的效果和实际函数地址的确定发生在运行时，所以这样就得到了所希望的晚捆绑。向这个对象发送消息，这个对象能断定它应当做什么。

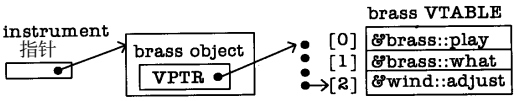

图 14-2

14.4.3 撩开面纱

如果能看到由虚函数调用而产生的汇编语言代码，这将是很有帮助的，这样可以看到后捆绑实际上是如何发生的。下面是在函数f（instrument&i）中调用

```
i.adjust(1);
```

某个编译器所产生的输出：

```
push   1
push   si
mov    bx,word ptr [si]
call   word ptr [bx+4]
add    sp,4
```

C++函数调用的参数与C函数调用一样，是从右向左进栈的（这个顺序是为了支持C的变量参数表），所以参数1首先压栈。在这个函数的这个地方，寄存器si（intel x86处理器的一部分）存放i的首地址。因为它是被选中的对象的首地址，它也被压进栈。记住，这个首地址对应于this的值，正因为调用每个成员函数时this都必须作为参数压进栈，所以成员函数知道它工作在哪个特殊对象上。这样，我们总能看到，在成员函数调用之前压栈的次数等于参数个数加一（除了static成员函数，它没有this）。

现在，必须实现实际的虚函数调用。首先，必须产生VPTR，使得能找到VTABLE。对于这个编译器，VPTR在对象的开头，所以this的内容对应于VPTR。下面这一行

```
mov bx，word ptr[si]
```

取出si（即this）所指的字，它就是VPTR。将这个VPTR放入寄存器bx中。

放在bx中的这个VPTR指向这个VTABLE的首地址，但被调用的函数在VTABLE中不是第0个位置，而是第二个位置（因为它是这个表中的第三个函数）。对于这种内存模式，每个函数指针是两个字节长，所以编译器对VPTR加四，计算相应的函数地址所在的地方，注意，这是编译时建立的常值。所以我们只要保证在第二个位置上的指针恰好指向adjust()。幸好编译器仔细处理，并保证在VTABLE中的所有函数指针都以相同的次序出现。

一旦在VTABLE中相应函数指针的地址被计算出来，就调用这个函数。所以取出这个地址并马上在这个句子中调用：

```
call word ptr [bx+4]
```

最后，栈指针移回去，以清除在调用之前压入栈的参数。在 C和C++汇编代码中，我们将常常看到调用者清除这些参数，但这依处理器和编译器的实现而有所变化。

14.4.4 安装vpointer

因为VPTR决定了对象的虚函数的行为，所以我们会看到 VPTR总是指向相应的VTABLE是多么重要。在VPTR适当初始化之前，我们绝对不能对虚函数调用。当然，能保证初始化的地点是在构造函数中，但是，在WIND例子中没有一个是有构造函数的。

这样，缺省构造函数的创建是很关键的。在 WIND例子中，编译器创建了一个缺省构造函数，它只做初始化 VPTR的工作。在能用任何 instrument对象做任何事情之前，对于任何instrument对象自动调用这个构造函数。所以，调用虚函数是安全的。

在构造函数中，自动初始化VPTR的含义在下一节讨论。

14.4.5 对象是不同的

认识到向上映射仅处理地址，这是重要的。如果编译器有一个它知道确切类型的对象，那么（在C++中）对任何函数的调用将不再用晚捆绑，或至少编译器不必须用晚捆绑。因为编译器知道对象的类型，为了提高效率，当调用这些对象的虚函数时，很多编译器使用早捆绑。下面是一个例子：

```
//: EARLY.CPP -- Early binding & virtuals
#include <iostream.h>

class base {
public:
  virtual int f() const { return 1; }
};

class derived : public base {
public:
  int f() const { return 2; }
};
main() {
  derived d;
  base* b1 = &d;
  base& b2 = d;
  base b3;
  // Late binding for both:
  cout << "b1->f() = " << b1->f() << endl;
  cout << "b2.f() = " << b2.f() << endl;
  // Early binding (probably):
  cout << "b3.f() = " << b3.f() << endl;
}
```

在b1->f()和b2.f()中，使用地址，就意味着信息不完全：b1和b2可能表示base的地址也可能表示其派生对象的地址，所以必须用虚函数。而当调用 b3.f()时不存在含糊，编译器知道确切的类型和知道它是一个对象，所以它不可能是由 base派生的对象，而确切的只是一个 base。这

样，可以用早捆绑。但是，如果不希望编译器的工作如此复杂，仍可以用晚捆绑，并且有相同的结果。

14.5 为什么需要虚函数

在这个问题上，我们可能会问：“如果这个技术如此重要，并且能使得任何时候都能调用‘正确’的函数。那么为什么它是可选的呢？为什么我还需要知道它呢？”

问得好。回答关系到C++的基本哲学：“因为它不是相当高效率的”。从前面的汇编语言输出可以看出，它并不是对于绝对地址的一个简单的CALL，而是为设置虚函数调用需要多于两条复杂的汇编指令。这既需要代码空间，又需要执行时间。一些面向对象的语言已经接受了这种概念，即晚捆绑对于面向对象程序设计是性质所决定的，所以应当总是出现，它应当是不可选的，而且用户不应当必须知道它。这是由创造语言时的设计决定，而这种特殊的方法对于许多语言是合适的[1]。C++来自C传统，效率是重要的。创造C完全是为了代替汇编语言以实现操作系统（从而改写操作系统——Unix——使得比它的先驱更轻便）。希望有C++的主要理由之一是让C程序员效率更高[2]。C程序员遇到C++时提出的第一个问题是“我将得到什么样的规模和速度效果？”如果回答是“除了函数调用时需要有一点额外的开销外，一切皆好”，那么许多人就会仍使用C，而不会改变到C++。另外，内联函数是不可能的，因为虚函数必须有地址放在VTABLE中。所以虚函数是可选的，而且该语言的缺省是非虚拟的，这是最快的配置。Stroustrup声明他的方针是“如果我们不用它，我们就不会为它花费”。

因此，virtual关键字可以改变程序的效率。然而，设计我们的类时，我们不应当为效率问题而担心。如果我们想使用多态，就在每处使用虚函数。当我们试图加速我们的代码时，我们只需寻找能让它非虚的函数（在其他方面通常有更大的好处）。

有些证据表明，进入C++的规模和速度改进是在C的规模和速度的10%之内，并且常常更接近。能够得到更小的规模和更高速度的原因是因为C++可以有比用C更快的方法设计程序，而且设计的程序更小。

14.6 抽象基类和纯虚函数

在所有的instrument的例子中，基类instrument中的函数总是“假”函数。如果调用这些函数，就会指出已经做错了什么事。这是因为，instrument的目的是对所有从它派生来的类创建公共接口，如在下面的图中看到的：

虚线表示类（一个类只是一个描述，而不是一个物理实体——虚线代表了它的非物理的“性质”）。从派生类到基类的箭头表示继承关系。

建立公共接口的唯一的理由是使得它能对于每个不同的子类有不同的表示。它建立一个基本的格式，由此可以知道什么是对于所有派生类公共的。注意，另外一种表达方法是称instrument为抽象基类（或简称为抽象类），当希望通过公共接口

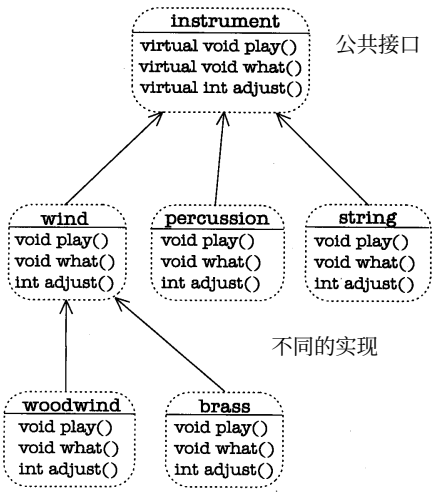

图 14-3

[1] 例如，Smalltalk用这种方法获得了很大的成功。

[2] 发明C++的贝尔实验室就是利用高效率为公司节约了大笔的费用。

操作一组类时就创建抽象类。

注意，只需在基类中声明函数为 virtual。与这个基类声明相匹配的所有派生类函数都将按照虚机制调用。当然我们也可以在派生类声明中使用 virtual 关键字（有些人为了清楚而这样做），但这是多余的。

如果我们有一个真实的抽象类（就像 instrument），这个类的对象几乎总是没有意义的。也就是说，instrument的含义只表示接口，不表示特例实现。所以创建一个 instrument对象没有意义。我们也许想防止用户这样做。这能通过让 instrument的所有虚函数打印出错信息而完成，但这种方法到运行时才能获得出错信息，并且要求用户可靠而详尽地测试。所以最好是在编译时就能发现这个问题。

C++对此提供了一种机制，称为纯虚函数。下面是它的声明语法：

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

这样做，等于告诉编译器在VTABLE中为函数保留一个间隔，但在这个特定间隔中不放地址。只要有一个函数在类中被声明为纯虚函数，则 VTABLE就是不完全的。包含有纯虚函数的类称为纯抽象基类。

如果一个类的VTABLE是不完全的，当某人试图创建这个类的对象时，编译器做什么呢？由于它不能安全地创建一个纯抽象类的对象，所以如果我们试图制造一个纯抽象类的对象，编译器就发出一个出错信息。这样，编译器就保证了抽象类的纯洁性，我们就不用担心误用它了。

这是修改后的WIND4.CPP 14.3节，它使用了纯虚函数：

```
//: WIND5.CPP -- Pure abstract base classes
#include <iostream.h>
enum note { middleC, Csharp, Cflat }; // Etc.

class instrument {
public:
  // Pure virtual functions:
  virtual void play(note) const = 0;
  virtual char* what() const = 0;
  // Assume this will modify the object:
  virtual void adjust(int) = 0;
};
// Rest of the file is the same ...

class wind : public instrument {
public:
  void play(note) const {
    cout << "wind::play" << endl;
  }
  char* what() const { return "wind"; }
  void adjust(int) {}
};

class percussion : public instrument {
```

```
public:
  void play(note) const {
    cout << "percussion::play" << endl;
  }
  char* what() const { return "percussion"; }
  void adjust(int) {}
};

class string : public instrument {
public:
  void play(note) const {
    cout << "string::play" << endl;
  }
  char* what() const { return "string"; }
  void adjust(int) {}
};

class brass : public wind {
public:
  void play(note) const {
    cout << "brass::play" << endl;
  }
  char* what() const { return "brass"; }
};

class woodwind : public wind {
public:
  void play(note) const {
    cout << "woodwind::play" << endl;
  }
  char* what() const { return "woodwind"; }
};

// Identical function from before:
void tune(instrument& i) {
  // ...
  i.play(middleC);
}

// New function:
void f(instrument& i) { i.adjust(1); }

main() {
  wind flute;
  percussion drum;
  string violin;
```

```
  brass flugelhorn;
  woodwind recorder;
  tune(flute);
  tune(drum);
  tune(violin);
  tune(flugelhorn);
  tune(recorder);
  f(flugelhorn);
}
```

纯虚函数是非常有用的，因为它们使得类有明显的抽象性，并告诉用户和编译器希望如何使用。

注意，纯虚函数防止对纯抽象类的函数以传值方式调用。这样，它也是防止对象意外使用值向上映射的一种方法。这样就能保证在向上映射期间总是使用指针或引用。

纯虚函数防止产生VTABLE，但这并不意味着我们不希望对其他函数产生函数体。我们常常希望调用一个函数的基类版本，即便它是虚拟的。把公共代码放在尽可能靠近我们的类层次根的地方，这是很好的想法。这不仅节省了代码空间，而且能允许使改变的传播变得容易。

纯虚定义

在基类中，对纯虚函数提供定义是可能的。我们仍然告诉编译器不要允许纯抽象基类的对象，而且纯虚函数在派生类中必须定义，以便于创建对象。然而，我们可能希望一块代码对于一些或所有派生类定义能共同使用，不希望在每个函数中重复这段代码，如下所示：

```
//: PVDEF.CPP -- Pure virtual base definition
#include <iostream.h>

class base {
public:
  virtual void v() const = 0;
  // In situ:
  virtual void f() const = 0 {
    cout << "base::f()\n";
  }
};

void base::v() const { cout << "base::v()\n";}

class d : public base {
public:
  // Use the common base code:
  void v() const { base::v(); }
  void f() const { base::f(); }
};

main() {
```

```
  d D;
  D.v();
  D.f();
}
```

在base VTABLE中的间隔仍然空着，但在这个派生类中刚好有一个函数，可以通过名字调用它。

这个特征的另外的好处是，它允许使用一个纯虚函数而不打乱已存在的代码。（这是一个处理没有重定义虚函数类的方法。）

14.7 继承和VTABLE

可以想象，当实现继承和定义一些虚函数时，会发生什么事情？编译器对新类创建一个新VTABLE表，并且插入新函数的地址，对于没有重定义的虚函数使用基类函数的地址。无论如何，在VTABLE中总有全体函数的地址，所以绝对不会对不在其中的地址调用。（否则损失惨重。）

但当在派生类中增加新的虚函数时会发生什么呢？这里有一个例子：

```
//: ADDV.CPP -- Adding virtuals in derivation
#include <iostream.h>

class base {
  int i;
public:
  base(int I) : i(I) {}
  virtual int value() const { return i; }
};
class derived : public base {
public:
  derived(int I) : base(I) {}
  int value() const {
    return base::value() * 2;
  }
  // New virtual function in the derived class:
  virtual int shift(int x) const {
    return base::value() << x;
  }
};

main() {
  base* B[] = { new base(7), new derived(7) };
  cout << "B[0]->value() = "
       << B[0]->value() << endl;
  cout << "B[1]->value() = "
       << B[1]->value() << endl;
//! cout << "B[1]->shift(3) = "
//!      << B[1]->shift(3) << endl; // Illegal
}
```

类base包含单个虚函数value()，而类derived增加了第二个称为shift()的虚函数，并重定义了value的含义。下图有助于显示发生的事情，其中有编译器为 base和derived创建的两个VTABLE。

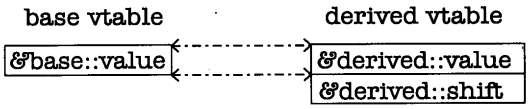

图 14-4

注意，编译器映射 derived VTABLE中的value地址位置等于在base VTABLE中的位置。类似的，如果一个类从derived继承而来，它的shift版本在它的VTABLE中的位置应当等于在derived中的位置。这是因为（正如通过汇编语言例子看到的）编译器产生的代码只是简单地在 VTABLE中用偏移选择虚函数。不论对象属于哪个特殊的类，它的VTABLE是以同样的方法设置的，所以对虚函数的调用将总是用同样的方法。

这样，编译器只对指向基类对象的指针工作。而这个基类只有 value函数，所以它就是编译器允许调用的唯一的函数。那么，如果只有指向基类对象的指针，那么编译器怎么可能知道自己正在对derived对象工作呢？这个指针可能指向其他一些没有shift函数的类。在VTABLE中，可能有，也可能没有一些其他函数的地址，但无论何种情况，对这个VTABLE地址做虚函数调用都不是我们想要的。所以编译器防止对只在派生类中存在的函数做虚函数调用，这是幸运的，合乎逻辑的。

有一些很少见的情况：可能我们知道指针实际上指向哪一种特殊子类的对象。这时如果想调用只存在于这个子类中的函数，则必须映射这个指针。下面的语句可以纠正由前面程序产生的错误：

```
((derived*)B[1])->shift(3)
```

在这里我们碰巧知道B[1]指向derived对象，但这种情况很少见。如果我们的程序确定我们必须知道所有对象的准确的类型，那么我们应当重新考虑它，因为我们可能在进行不正确的虚函数调用。然而对于有些情况如果知道保存在一般包容器中的所有对象的准确类型，会使我们的设计工作在最佳状态（或没有选择）。这就是运行时类型辨认问题（简称RTTI）。

运行时类型辨认是有关映射基类指针向下到派生类指针的问题。（“向上”和“向下”是相对典型类图而言的，典型类图以基类为顶点。）向上映射是自动发生的，不需强制，因为它是绝对安全的。向下映射是不安全的，因为这里没有关于实际类型的编译信息，所以必须准确地知道这个类实际上是什么类型。如果把它映射成错误的类型，就会出现麻烦。

第18章将描述C++提供运行时类型信息的方法。

• 对象切片

当多态地处理对象时，传地址与传值有明显的不同。所有在这里已经看到的例子和将会看到的例子都是传地址的，而不是传值的。这是因为地址都有相同的长度 [1]，传派生类型（它通常稍大一些）对象的地址和传基类（它通常小一点）对象的地址是相同的。如前面解释的，使用多态的目的是让对基类对象操作的代码也能操作派生类对象。

如果使用对象而不是使用地址或引用进行向上映射，发生的事情会使我们吃惊：这个对象被“切片”，直到所剩下来的是适合于目的的子对象。在下面例子中可以看到通过检查这个对象的长度切片剩下来的部分。

```
//: SLICE.CPP -- Object slicing
#include <iostream.h>
```

[1] 实际上，并不是所有机器上的指针都是同样大小的。但就本书讨论的范围而言，它们可被认为是同样大小的。

```
class base {
  int i;
public:
  base(int I = 0) : i(I) {}
  virtual int sum() const { return i; }
};

class derived : public base {
  int j;
public:
  derived(int I = 0, int J = 0)
    : base(I), j(J) {}
  int sum() const { return base::sum() + j; }
};

void call(base b) {
  cout << "sum = " << b.sum() << endl;
}

main() {
  base b(10);
  derived d(10, 47);
  call(b);
  call(d);
}
```

函数call()通过传值传递一个类型为base的对象。然后对于这个base对象调用虚函数sum()。我们可能希望第一次调用产生 10，第二次调用产生57。实际上，两次都产生10。

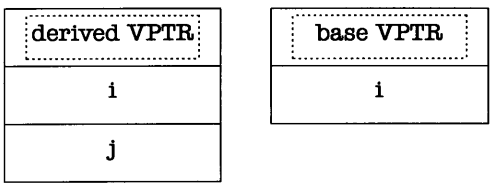

图 14-5

在这个程序中，有两件事情发生了。第一，call()接受的只是一个base对象，所以所有在这个函数体内的代码都将只操作与 base相关的数。对call()的任何调用都将引起一个与 base大小相同的对象压栈并在调用后清除。这意味着，如果一个由base派生来类对象被传给call，编译器接受它，但只拷贝这个对象对应于base的部分，切除这个对象的派生部分，如图：

现在，我们可能对这个虚函数调用感到奇怪：这里，这个虚函数既使用了base（它仍存在），又使用了derived的部分（derived不再存在了，因为它被切片）。

其实我们已经从灾难中被解救出来，这个对象正安全地以值传递。因为这时编译器认为它知道这个对象的确切的类型（这个对象的额外特征有用的任何信息都已经失去）。另外，用值传递时，它对base对象使用拷贝构造函数，该构造函数初始化 VPTR指向base VTABLE，并且只拷贝这个对象的base部分。这里没有显式的拷贝构造函数，所以编译器自动地为我们合成一个。由于上述诸原因，这个对象在切片期间变成了一个base对象。

对象切片实际上是去掉了对象的一部分，而不是象使用指针或引用那样简单地改变地址的

内容。因此，对象向上映射不常做，事实上，通常要提防或防止这种操作。我们可以通过在基类中放置纯虚函数来防止对象切片。这时如果进行对象切片就将引起编译时的出错信息。

14.8 虚函数和构造函数

当创建一个包含有虚函数的对象时，必须初始化它的 VPTR以指向相应的VTABLE。这必须在有关虚函数的任何调用之前完成。正如我们可能猜到的，因为构造函数有使对象成为存在物的工作，所以它也有设置 VPTR的工作。编译器在构造函数的开头部分秘密地插入能初始化VPTR的代码。事实上，即使我们没有对一个类创建构造函数，编译器也会为我们创建一个带有相应VPTR初始化代码的构造函数（如果有虚函数）。这有几个含意。

首先这涉及效率。内联（inline）函数的理由是对小函数减少调用代价。如果C++不提供内联（inline）函数，预处理器就可能被用以创建这些“宏”。然而，预处理器没有通道或类的概念，因此不能被用以创建成员函数宏。另外，有了由编译器插入隐藏代码的构造函数，预处理宏根本不能工作。

当寻找效率漏洞时，我们必须明白，编译器正在插入隐藏代码到我们的构造函数中。这些隐藏代码不仅必须初始化 VPTR，而且还必须检查this的值（万一operator new返回零）和调用基类构造函数。放在一起，这些代码能影响我们认为是一个小内联函数的调用。特别是，构造函数的规模会抵消减少函数调用节省的费用。如果做大量的内联构造函数调用，我们的代码长度就会增长，而在速度上没有任何好处。

当然，我们也许并不会立即把这些小构造函数都变成非内联，因为它们更容易做为内联的来写。但是，当我们正在调整我们的代码时，记住，务必去掉这些构造函数的内联性。

14.8.1 构造函数调用次序

构造函数和虚函数的第二个有趣的方面涉及构造函数的调用顺序和在构造函数中虚函数调用的方法。

所有基类构造函数总是在继承类构造函数中被调用。这是有意义的，因为构造函数有一项专门的工作：确保对象被正确的建立。派生类只访问它自己的成员，而不访问基类的成员，只有基类构造函数能恰当地初始化它自己的成员。因此，确保所有的构造函数被调用是很关键的，否则整个对象不会适当地被构造。这就是为什么编译器强制构造函数对派生类的每个部分调用。如果不在构造函数初始化表达式表中显式地调用基类构造函数，它就调用缺省构造函数。如果没有缺省构造函数，编译器将报告出错。（在这个例子中， class x没有构造函数，所以编译器能自动创建一个缺省构造函数。）

构造函数调用的顺序是重要的。当继承时，我们必须完全知道基类和能访问基类的任何public 和protected成员。这也就是说，当我们在派生类中时，必须能肯定基类的所有成员都是有效的。在通常的成员函数中，构造已经发生，所以这个对象的所有部分的成员都已经建立。然而，在构造函数内，必须想办法保证所有我们的成员都已经建立。保证它的唯一方法是让基类构造函数首先被调用。这样，当我们在派生类构造函数中时，在基类中我们能访问的所有成员都已经被初始化了。在构造函数中，“必须知道所有成员对象是有效的”也是下面做法的理由：只要可能，我们应当在这个构造函数初始化表达式表中初始化所有的成员对象（放在合成的类中的对象）。只要我们遵从这个做法，我们就能保证所有基类成员和当前对象的成员对象已经被初始化。

14.8.2 虚函数在构造函数中的行为

构造函数调用层次会导致一个有趣的两难选择。试想；如果我们正在构造函数中并且调用虚函数，那么会发生什么现象呢？对于普通的成员函数，虚函数的调用是在运行时决定的，这是因为编译时并不能知道这个对象是属于这个成员函数所在的那个类，还是属于由它派生出来的类。于是，我们也许会认为在构造函数中也会发生同样的事情。

然而，情况并非如此。对于在构造函数中调用一个虚函数的情况，被调用的只是这个函数的本地版本。也就是说，虚机制在构造函数中不工作。

这个行为有两个理由。在概念上，构造函数的工作是把对象变成存在物。在任何构造函数中，对象可能只是部分被形成——我们只能知道基类已被初始化了，但不知道哪个类是从这个基类继承来的。然而，虚函数是“向前”和“向外”进行调用。它能调用在派生类中的函数。如果我们在构造函数中也这样做，那么我们所调用的函数可能操作还没有被初始化的成员，这将导致灾难的发生。

第二个理由是机械的。当一个构造函数被调用时，它做的首要的事情之一是初始化它的VPTR。因此，它只能知道它是“当前”类的，而完全忽视这个对象后面是否还有继承者。当编译器为这个构造函数产生代码时，它是为这个类的构造函数产生代码 --既不是为基类，也不是为它的派生类（因为类不知道谁继承它）。所以它使用的VPTR必须是对于这个类的VTABLE。而且，只要它是最后的构造函数调用，那么在这个对象的生命期内， VPTR将保持被初始化为指向这个VTABLE。但如果接着还有一个更晚派生的构造函数被调用，这个构造函数又将设置VPTR指向它的VTABLE，等等，直到最后的构造函数结束。VPTR的状态是由被最后调用的构造函数确定的。这就是为什么构造函数调用是从基类到更加派生类顺序的另一个理由。

但是，当这一系列构造函数调用正发生时，每个构造函数都已经设置 VPTR指向它自己的VTABLE。如果函数调用使用虚机制，它将只产生通过它自己的 VTABLE的调用，而不是最后的VTABLE（所有构造函数被调用后才会有最后的 VTABLE）。另外，许多编译器认识到，如果在构造函数中进行虚函数调用，应该使用早捆绑，因为它们知道晚捆绑将只对本地函数产生调用。无论哪种情况，在构造函数中调用虚函数都没有结果。

14.9 析构函数和虚拟析构函数

构造函数不能是虚的（在附录B中的技术只类似于虚构造函数）。但析构函数能够且常常必须是虚的。

构造函数有其特殊的工作。它首先调用基本构造函数，然后调用在继承顺序中的更晚派生的构造函数，如此一块一块地把对象拼起来。类似的，析构函数也有一个特殊的工作——它必须拆卸可能属于某类层次的对象。为了做这些工作，它必须按照构造函数调用相反的顺序，调用所有的析构函数。这就是，析构函数自最晚派生的类开始，并向上到基类。这是安全且合理的：当前的析构函数能知道基类成员仍是有效的，因为它知道它是从哪一个派生而来的，但不知道从它派生出哪些。

应当记住，构造函数和析构函数是必须遵守调用层次唯一的地方。在所有其他函数中，只是某个函数被调用，而无论它是虚的还是非虚的。同一个函数的基类版本在通常的函数中被调用（无论虚否）的唯一的方法是直接地调用这个函数。

通常，析构函数的活动是很正常的。但是，如果我们想通过指向某个对象的基类的指针操纵这个对象（这就是，通过它的一般接口操纵这个对象），会发生什么现象呢？这在面向对象

的程序设计中确实很重要。当我们想delete在栈中已经用new创建的类的对象的指针时，就会出现这个问题。如果这个指针是指向基类的，编译器只能知道在 delete期间调用这个析构函数的基类版本。我们已经知道，虚函数被创建恰恰是为了解决同样的问题。幸好，析构函数可以是虚函数，于是一切问题就迎刃而解了。

虽然析构函数象构造函数一样，是“ 例外 ”函数，但析构函数可以是虚的，这是因为这个对象已经知道它是什么类型（而在构造期间则不然）。一旦对象已被构造，它的VPTR就已被初始化了，所以虚函数调用能发生。

如果我们创建一个纯虚析构函数，我们就必须提供函数体，因为（不像普通函数）在类层次中所有析构函数都总是被调用。这样，这个纯虚析构函数的函数体以调用结束。下面是例子：

```
//: PVDEST.CPP -- Pure virtual destructors
// require a function body.
#include <iostream.h>

class base {
public:
  virtual ~base() = 0 {
    cout << "~base()" << endl;
  }
};

class derived : public base {
public:
  ~derived() {
    cout << "~derived()" << endl;
  }
};

main() {
  base* bp = new derived; // Upcast
  delete bp; // Virtual destructor call
}
```

尽管在基类中的析构函数的纯虚性有强制继承者重定义这个析构函数的作用，但这个基类体仍作为析构函数的一部分被调用。

作为准则，任何时候在类中有虚函数，我们就应当直接增加虚析构函数（即便它什么事也不做）。这样，能保证以后不发生意外。

在析构函数中的虚机制

在析构期间，有一些我们可能不希望马上发生的情况。如果我们正在一个普通的成员函数中，并且调用一个虚函数，则这个函数被使用晚捆绑机制调用。而对于析构函数，这样不行，不论是虚的还是非虚的。在析构函数中，只有成员函数的本地版本被调用，虚机制被忽略。

为什么是这样呢？假设虚机制在析构函数中使用，那么调用下面这样的虚函数是可能的：这个函数是在继承层次中比当前的析构函数“ 更靠外 ”（更晚派生的）。但是，有一点我们要注意，析构函数从“ 外层 ”被调用（从最晚派生析构函数向基本析构函数）。所以，实际上被调

用的函数就可能操作在已被删除的对象上。因此，编译器决定在编译时只调用这个函数的“本地”版本。注意，对于构造函数也是如此（这在前面已讲到）。但在构造函数的情况下，这样做是因为信息还不可用，在析构函数中，信息（也就是VPTR）虽存在，但不可靠。

14.10 小结

多态性在C++中用虚函数实现，它有不同的形式。在面向对象的程序设计中，我们有相同的表面（在基类中的公共接口）和使用这个表面的不同的形式：虚函数的不同版本。

在这一章中，我们已经看到，理解甚至创建一个多态的例子，不用数据抽象和继承是不可能的。多态是不能隔离看待的特性（例如像const和switch语句），它必须同抽象与继承一起工作，它是类关系的一个重要方面。人们常常被C++的其他非面向对象的特性所混淆，例如重载和缺省参数，它们有时被作为面向对象的特性描述。不要犯傻，如果它不是晚捆绑它就不是多态。

为了在我们的程序中有效的使用多态等面向对象的技术，我们不能只知道让我们的程序包含单个类的成员和消息，而且还应当知道类的共性和它们之间的关系。虽然这需要很大的努力，但这是值得的，因为我们将更快地开发程序和更好地组织代码，得到可扩充的程序和更容易维护的代码。

多态完善了这个语言的面向对象特性，但在C++中，有两个更重要的特性：模板（第15章）和异常处理（第17章）。这些特性使我们的程序设计能力有很大的提高，就像面向对象的其他特性：抽象数据类型、继承和多态一样。

14.11 练习

1. 创建一个非常简单的“shape”层次：基类称为shape，派生类称为circle、square和triangle。在基类中定义一个虚函数draw()，再在这些派生类中重定义它。创建指向我们在堆中创建的shape对象的指针数组（这样就形成了指针向上映射）。并且通过基类指针调用draw()，检验这个虚函数的行为。如果我们的调试器支持，就用单步执行这个例子。

2. 修改练习1，使得draw()是纯虚函数。尝试创建一个类型为shape的对象。尝试在构造函数内调用这个纯虚函数，结果如何。给出draw()的一个定义。

3. 写出一个小程序以显示在普通成员函数中调用虚函数和在构造函数中调用虚函数的不同。这个程序应当证明两种调用产生不同的结果。

4. 在EARLY.CPP中，我们如何能知道编译器是用早捆绑还是晚捆绑进行调用？根据我们自己的编译器来确定。

5. （中级）创建一个不带成员和构造函数而只有一个虚函数的基类class X，创建一个从X继承的类class Y，它没有显式的构造函数。产生汇编代码并检验它，以确定X的构造函数是否被创建和调用，如果是的，这些代码做什么？解释我们的发现。X没有缺省构造函数，但是为什么编译器不报告出错？

6. （中级）修改练习5，让每个构造函数调用一个虚函数。产生汇编代码。确定在每个构造函数内VPTR在何处被赋值。在构造函数内编译器使用虚函数机制吗？确定为什么这些函数的本地版本仍被调用。

7. （高级）参数为传值方式传递的对象的函数调用如果不用早捆绑，则虚调用可能会侵入不存在的部分。这可能吗？写一些代码强制虚调用，看是否会引起冲突。解释这个现象，检验当对象以传值方式传递时会发生什么现象。

8. （高级）通过我们的处理器的汇编语言信息或者其他技术，找出简单调用所需的时间数及虚函数调用的时间数，从而得出虚函数调用需要多用多少时间。