第17章　虚函数与多态性

无论在编译时还是在运行时，C++都支持多态性。我们在前面章节讨论过：编译时多态性是通过重载函数和运算符实现的，运行时多态性是通过使用继承和虚函数实现的。本章将介绍这些主题。

17.1　虚函数

虚函数是一个成员函数，该成员函数在基类内部声明并且被派生类重新定义。为了创建虚函数，应在基类中该函数声明的前面加上关键字virtual。当继承包含虚函数的类时，派生类将重新定义该虚函数以符合自身的需要。从本质上讲，虚函数实现了“一个接口，多种方法”的理念，而这种理念是多态性的基础。基类内部的虚函数定义了该函数的接口形式。当明确涉及到该派生类时，派生类对虚函数的重新定义执行相应的操作，即，该重新定义创建一个具体的方法。

“正常”访问时，虚函数就像所有其他类型的类成员函数一样。然而，虚函数之所以重要，之所以能够支持运行时的多态性，就在于当通过指针对其进行访问时虚函数的表现。正如第13章中讨论的那样，一个基类指针可以用于指向该基类的任何派生类的对象。当基类指针指向包含虚函数的派生对象时，C++会根据该指针所指的对象类型决定调用的虚函数版本。这一决定是在运行时做出的，因此，当指针指向不同的对象时，就执行该虚函数的不同版本。这同样适用于基类引用。

我们首先分析下面这个简短的例子：

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

class base {
public:
  virtual void vfunc() {
    cout << "This is base's vfunc().\n";
  }
};

class derived1 : public base {
public:
  void vfunc() {
    cout << "This is derived1's vfunc().\n";
  }
};

class derived2 : public base {
public:
  void vfunc() {
```

```
    cout << "This is derived2's vfunc().\n";
  }
};

int main()
{
  base *p, b;
  derived1 d1;
  derived2 d2;

  // point to base
  p = &b;
  p->vfunc(); // access base's vfunc()

  // point to derived1
  p = &d1;
  p->vfunc(); // access derived1's vfunc()

  // point to derived2
  p = &d2;
  p->vfunc(); // access derived2's vfunc()

  return 0;
}
```

这个程序显示下面的输出信息：

```
This is base's vfunc().
This is derived1's vfunc().
This is derived2's vfunc().
```

如程序所示，在 base 内部声明了虚函数 vfunc( )。注意，关键字 virtual 位于函数声明的前面，当 derived1 和 derived2 重新定义 vfunc( )时，不再需要关键字 virtual（然而，在派生类内部当重新定义虚函数时包括了它也不算错误，只是不需要它而已）。

在这个程序中，base 被 derived1 和 derived2 继承。在每个类定义内部，vfunc( )被重新定义。在 main( )里，声明了 4 个变量：

变量名	变量类型
p	基类指针
b	基类对象
d1	derived1 的对象
d2	derived2 的对象

接下来，将 b 的地址赋给 p 并通过 p 调用 vfunc( )。因为此时 p 指向一个 base 类型的对象，所以应执行相应版本的 vfunc( )。然后，将 p 设置为 d1 的地址，并通过 p 再一次调用 vfunc( )。这一次 p 指向一个 derived1 类型的对象，这导致执行 derived1::vfunc( )。最后，将 d2 的地址赋给 p，且 p->vfunc( )将执行在 derived2 内重新定义的 vfunc( )版本。这里的关键是：p 所指的对象类型决定执行哪一个版本的 vfunc( )。此外，这个决定是在运行时做出的，而且这一过程构成了运行时多态性的基础。

虽然可以通过使用对象名和点运算符以一般的方式调用虚函数，但只有通过基类指针访问时才能实现运行时的多态性。例如，假定仍采用前面的例子，下面的语句在语法上是正确的：

```
d2.vfunc(); // calls derived2's vfunc()
```

以这种方式调用虚函数并没有错，只是没有利用vfunc( )的虚特性。

乍看上去，利用派生类对虚函数进行的重新定义类似于函数重载，然而实际上却并非如此。由于二者存在若干差异，所以不能将“重载”这个术语应用于对虚函数的重新定义上。或许最重要的是：重新定义的虚函数原型必须完全符合基类中指定的原型。这不同于重载一个普通函数，在普通函数里，返回类型以及参数的个数和类型都不同（事实上，当重载一个函数时，不是参数个数不同，就是参数类型不同，二者之中必须有一个不同！C++正是通过这些差异才能够选出正确的重载函数版本）。然而，当重新定义虚函数时，其原型的所有方面必须完全相同。如果在重新定义虚函数时改变了它的原型，那么该函数只能被认为是由C++编译器重载的，其虚特性也将丧失。另一个重要的限制是：虚函数必须是其所属类的非静态成员而不能是友元（friend）。最后，构造函数不能是虚函数，但析构函数可以是。

因为这些限制以及函数重载与虚函数重新定义之间的许多差异，所以我们用术语覆盖（overriding）来描述派生类对虚函数所做的重新定义。

17.1.1 通过基类引用调用虚函数

虽然在前面的例子中，虚函数是通过基类指针调用的，但是通过基类引用调用虚函数时，其多态特性也是可用的。正如第13章介绍的那样，一个引用是一个隐含的指针，所以一个基类引用可以用来访问该基类的某个对象或任何从该基类派生的对象。当通过基类引用调用虚函数时，所执行的函数版本由调用时访问的对象决定。

最常见的通过基类引用调用虚函数的情况是：引用是一个函数参数。例如，看一看下面的程序，该程序是前面程序的变体：

```
/* Here, a base class reference is used to access
   a virtual function. */
#include <iostream>
using namespace std;

class base {
public:
  virtual void vfunc() {
    cout << "This is base's vfunc().\n";
  }
};

class derived1 : public base {
public:
  void vfunc() {
    cout << "This is derived1's vfunc().\n";
  }
};

class derived2 : public base {
public:
  void vfunc() {
    cout << "This is derived2's vfunc().\n";
  }
};
```

```
// Use a base class reference parameter.
void f(base &r) {
  r.vfunc();
}

int main()
{
  base b;
  derived1 d1;
  derived2 d2;

  f(b); // pass a base object to f()
  f(d1); // pass a derived1 object to f()
  f(d2); // pass a derived2 object to f()

  return 0;
}
```

这个程序的输出与前面的程序相同。在这个例子中，函数 f( )定义了一个 base 类型的引用参数。在 main( )内，该函数通过 base，derived1 和 derived2 类型的对象被调用。在 f( )内，被调用的 vfunc( )函数版本由调用该函数时引用的对象类型决定。

为了简单起见，本章其余的例子将通过基类指针调用虚函数，其效果与采用基类引用时的效果相同。

17.2　继承虚属性

当继承虚函数时，其虚属性也被继承下来。这意味着，当一个继承了虚函数的派生类本身用做另一个派生类的基类时，该虚函数仍然可以被覆盖。换句话说，无论虚函数被继承多少次，它仍然是虚函数。例如，看一看下面的程序：

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

class base {
public:
  virtual void vfunc() {
    cout << "This is base's vfunc().\n";
  }
};

class derived1 : public base {
public:
  void vfunc() {
    cout << "This is derived1's vfunc().\n";
  }
};

/* derived2 inherits virtual function vfunc()
   from derived1. */
class derived2 : public derived1 {
public:
  // vfunc() is still virtual
  void vfunc() {
```

```
    cout << "This is derived2's vfunc().\n";
  }
};

int main()
{
  base *p, b;
  derived1 d1;
  derived2 d2;

  // point to base
  p = &b;
  p->vfunc(); // access base's vfunc()

  // point to derived1
  p = &d1;
  p->vfunc(); // access derived1's vfunc()

  // point to derived2
  p = &d2;
  p->vfunc(); // access derived2's vfunc()

  return 0;
}
```

正如我们所期望的，上面的程序显示如下输出：

```
This is base's vfunc().
This is derived1's vfunc().
This is derived2's vfunc().
```

在这个例子中，derived2 继承 derived1，而不是继承 base，但是 vfunc( )仍然是虚函数。

17.3 虚函数是分层的

正如上面所述，当一个函数被基类声明为 virtual 时，该函数可以被一个派生类覆盖，然而该函数并非一定被覆盖。当派生类未能覆盖虚函数时，如果该派生类的对象访问这个函数，那么将使用基类定义的函数。例如，看一看下面的程序，其中，derived2 没有覆盖 vfunc( )：

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

class base {
public:
  virtual void vfunc() {
  cout << "This is base's vfunc().\n";
  }
};

class derived1 : public base {
public:
  void vfunc() {
    cout << "This is derived1's vfunc().\n";
  }
};
```

```
class derived2 : public base {
public:
// vfunc() not overridden by derived2, base's is used
};

int main()
{
  base *p, b;
  derived1 d1;
  derived2 d2;

  // point to base
  p = &b;
  p->vfunc(); // access base's vfunc()

  // point to derived1
  p = &d1;
  p->vfunc(); // access derived1's vfunc()

  // point to derived2
  p = &d2;
  p->vfunc(); // use base's vfunc()

  return 0;
}
```

这个程序的输出是：

```
This is base's vfunc().
This is derived1's vfunc().
This is base's vfunc().
```

因为 derived2 没有覆盖 vfunc( )，当针对类型 derived2 的对象引用 vfunc( )时，使用的是由 base 定义的函数。

上面的程序说明的情况阐明了一个更一般规则的特殊情况。因为在 C++ 中继承是分层的，所以虚函数也是分层的。也就是说，当某个派生类不能覆盖虚函数时，使用按照派生顺序的逆序找到的第一个重新定义的函数。例如，在下面的程序中，derived2 由 derived1 派生，而 derived1 由 base 派生，但是 derived2 没有覆盖 vfunc( )。这意味着，对于 derived2 而言，最近的 vfunc( ) 版本在 derived1 中，因此当 derived2 的对象试图调用 vfunc( )时，应该使用 derived1::vfunc( )。

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

class base {
public:
  virtual void vfunc() {
    cout << "This is base's vfunc().\n";
  }
};

class derived1 : public base {
public:
  void vfunc() {
    cout << "This is derived1's vfunc().\n";
```

```
  }
};

class derived2 : public derived1 {
public:
/* vfunc() not overridden by derived2.
   In this case, since derived2 is derived from
   derived1, derived1's vfunc() is used.
*/
};

int main()
{
  base *p, b;
  derived1 d1;
  derived2 d2;

  // point to base
  p = &b;
  p->vfunc(); // access base's vfunc()

  // point to derived1
  p = &d1;
  p->vfunc(); // access derived1's vfunc()

  // point to derived2
  p = &d2;
  p->vfunc(); // use derived1's vfunc()

  return 0;
}
```

这个程序的输出如下：

```
This is base's vfunc().
This is derived1's vfunc().
This is derived1's vfunc().
```

17.4 纯虚函数

正如前一节通过例子说明的那样，当虚函数没有被派生类重新定义时，将使用基类中定义的虚函数版本。然而，在许多情况下，基类中没有有意义的虚函数定义。例如，某个基类或许不能够充分定义一个对象，以允许创建一个基类虚函数。此外，在某些情况下，你可能想使某个虚函数被所有派生类覆盖。为了处理上述两种情况，C++对纯虚函数提供支持。

纯虚函数是没有在基类中定义的虚函数。要声明一个纯虚函数，应采用下面的形式：

virtual *type func-name(parameter-list)* = 0;

当一个虚函数变为纯虚函数时，任何派生类必须给出自己的定义。如果派生类未能覆盖该纯虚函数，则将导致编译错误。

下面的程序包含一个简单的纯虚函数范例。其中，基类number包含一个整型变量val、函数setval( )和纯虚函数show( )，派生类hextype，dectype，octtype继承了number并重新定义

了 show( )，从而分别以相应的数基（即十六进制、十进制或八进制）输出 val 的值。

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

class number {
protected:
  int val;
public:
  void setval(int i) { val = i; }

  // show() is a pure virtual function
  virtual void show() = 0;
};

class hextype : public number {
public:
  void show() {
    cout << hex << val << "\n";
  }
};

class dectype : public number {
public:
  void show() {
    cout << val << "\n";
  }
};

class octtype : public number {
public:
  void show() {
    cout << oct << val << "\n";
  }
};

int main()
{
  dectype d;
  hextype h;
  octtype o;

  d.setval(20);
  d.show(); // displays 20 - decimal

  h.setval(20);
  h.show(); // displays 14 - hexadecimal

  o.setval(20);
  o.show(); // displays 24 - octal

  return 0;
}
```

虽然这个例子非常简单，但却说明了为什么基类或许不能定义有意义的虚函数。在这个例子中，number 只提供了派生类所用的通用接口。因为数基不确定，所以没有理由在 number 中

定义 show( )。当然，也可以总是创建一个虚函数的占位定义，然而，把 show( )处理为纯虚函数可以确保所有派生类真正对其进行重新定义以满足各自的需要。

记住，当虚函数被声明为纯虚函数时，所有派生类都必须覆盖该函数。如果某个派生类未能覆盖它，则将产生编译错误。

17.4.1 抽象类

至少包含一个纯虚函数的类称为抽象类。因为抽象类包含一个或多个没有定义的函数（即纯虚函数），所以不能创建抽象类的对象。相反，抽象类构成了一个不完全的类，这种类可以用做派生类的基础。

尽管不能创建抽象类的对象，但却可以创建指向抽象类的指针和引用。这使得抽象类能够支持运行时多态性，从而可以依靠基类指针和引用选择适当的虚函数。

17.5 使用虚函数

面向对象编程的核心内容之一是遵循“一个接口，多个方法”的原则。也就是说，可以定义一个一般的功能类，该类的接口是固定的，每一个派生类可以定义各自特有的操作。用C++的术语来说，可以用基类定义一般类的接口特性，当涉及到派生类所用的数据类型时，每一个派生类将执行其特有的操作。

实现“一个接口，多个方法”原则的最有效和最灵活的方法之一是利用虚函数、抽象类以及运行时多态性。利用这些特性，可以创建一个从一般到特殊（从基类到派生类）的类层次结构。按照这种理念，可以在一个基类中定义所有的通用功能和接口。在一些操作只能通过派生类实现的情况下，应该使用虚函数。从本质上讲，可以在基类中创建和定义与一般情况有关的所有事物，派生类则详细填充具体的内容。

下面列举了一个简单的例子，可以说明“一个接口，多个方法”理念的价值所在。我们可以创建一个类的分层结构，从而把一种单位体系转换为另一种（例如，把公升转换为加仑）。例子中的基类 convert 声明了两个变量：val1 和 val2，这两个变量分别存放初始值和转换后的值。此外，该基类还定义了函数 getinit( )和 getconv( )，这两个函数分别返回初始值和转换后的值。convert 的这些元素是固定的，可以被所有从 convert 继承而来的派生类使用。然而，实际上完成转换功能的函数compute( )是一个纯虚函数，该函数必须由从convert继承而来的派生类定义。compute( )的具体特性将由当时执行的转换类型决定。

```
// Virtual function practical example.
#include <iostream>
using namespace std;

class convert {
protected:
  double val1; // initial value
  double val2; // converted value
public:
  convert(double i) {
    val1 = i;
  }
```

```
  double getconv() { return val2; }
  double getinit() { return val1; }

  virtual void compute() = 0;
};

// Liters to gallons.
class l_to_g : public convert {
public:
  l_to_g(double i) : convert(i) { }
  void compute() {
    val2 = val1 / 3.7854;
  }
};

// Fahrenheit to Celsius
class f_to_c : public convert {
public:
  f_to_c(double i) : convert(i) { }
  void compute() {
    val2 = (val1-32) / 1.8;
  }
};

int main()
{
  convert *p; // pointer to base class

  l_to_g lgob(4);
  f_to_c fcob(70);

  // use virtual function mechanism to convert
  p = &lgob;
  cout << p->getinit() << " liters is ";
  p->compute();
  cout << p->getconv() << " gallons\n"; // l_to_g

  p = &fcob;
  cout << p->getinit() << " in Fahrenheit is ";
  p->compute();
  cout << p->getconv() << " Celsius\n"; // f_to_c

  return 0;
}
```

前面的程序创建了两个从convert继承而来的派生类，它们是l_to_g和f_to_c。这两个类分别把公升转换为加仑和把华氏温度（Fahrenheit）转换为摄氏温度（Celsius）。然而，即使l_to_g和f_to_c实际的转换操作（即方法）不同，但接口保持不变。

使用派生类和虚函数的好处之一是能够轻而易举地处理新的情况。例如，以上述程序为例，可以通过包含下面的类增加一种从英尺到米的转换功能：

```
// Feet to meters
class f_to_m : public convert {
public:
```

```
    f_to_m(double i) : convert(i) { }
    void compute() {
      val2 = val1 / 3.28;
    }
};
```

抽象类和虚函数的一个重要用途是在类库中。你可以创建一个通用的供其他程序员使用的可扩展类库。其他程序员可以继承你定义的一般类（该类定义接口及其从它派生的所有派生类共用的元素），并把具体函数添加到相应的派生类中。通过创建类库，可以创建一般类的接口，从而让其他程序员根据其特殊需要使用。

最后，基类convert是一个抽象类的例子。由于没有提供有意义的定义，所以没有在convert中定义虚函数compute( )。convert类没有包含定义compute( )的足够信息，只有当convert被某个派生类继承时才创建完整的类型。

17.6 早期绑定与后期绑定

在结束本章关于虚函数和运行时多态性的讨论之前，我们还需要解释两个术语，因为在讨论C++和面向对象的编程时，我们会频繁用到它们。这两个术语就是早期绑定和后期绑定。

早期绑定是指编译时发生的事件。从本质上讲，如果编译时知道调用某个函数需要的所有信息，就会发生早期绑定（换句话讲，早期绑定意味着在编译过程中对象和函数调用绑定在一起）。早期绑定的例子包括正常的函数调用（包括标准库函数）、重载函数调用以及重载运算符。早期绑定的主要优点是高效。因为调用一个函数需要的所有信息都在编译时确定，所以这种类型的函数调用速度非常快。

与早期绑定相反的是后期绑定。在谈到C++时，后期绑定是指在运行时才确定的函数调用。我们可以利用虚函数来实现后期绑定。我们知道，当通过基类指针和引用访问虚函数时，被调用的虚函数实际上是由指针所指的对象类型决定的。因为在一般情况下，被调用的虚函数在编译时还无法确定，相应的对象和函数直到运行时才被连接。后期绑定的主要优点是比较灵活。与早期绑定不同的是：后期绑定使编程人员创建的程序可以对程序运行时发生的事件做出响应，而且不必编写大量处理偶发事件的代码。记住，因为后期绑定的函数调用在运行时才能确定，所以执行速度变慢。