第20章

多态与虚函数

多态（polymorphism）是面向对象程序设计（object-oriented programming）的第三个特色，它可以让程序有更好的结构及可读性，减少使用继承以定义新类的困难，因而能够加速大型程序的开发，并简化后续的程序维护工作。在 C++ 的语法中，多态需要使用虚函数来实现。

20.1 多态的基本概念

通过类 (class) 的声明，我们可以定义全新的数据类型，把对象的属性（attributes）和行为（behaviors）以数据成员和成员函数的方式封装起来，并通过 private，public 和 protected 三种存取设定关键词（access specifier）来管制成员和外界的存取关系。

在上述面向对象程序设计的协助下，我们写作程序时关心的是类的创造，以及如何以传递信息的方式与对象交互。使用继承（inheritance）和组合（composition），我们可以在不更动既有类的情况下，建立新的类以保留原有的功能或接口，进一步利用到类间的相似性及相对关系。

没有封装（encapsulation）和数据抽象化（data abstraction）就谈不上继承。同样地，如果没有了继承也就没有多态。在讨论多态之前，我们先要介绍什么叫 **upcast**（向上转换数据类型）。

Cast（数据类型转换）

“Cast”原本是“铸造”的意思。把呈流体状的材料倒到模子里，让它冷却成型就是 **cast**。例如，铜制雕像的铸造和家里“布丁”的制作都是。在程序设计的领域里，我们借用这个概念来描述数据类型间的转换。譬如说，有一个名叫 N 的 int 数字：

```
int N = 5;
```

则

```
float(N)
int (&N)
```

分别将变量 N 的值，以及 N 的地址转换为数据类型 float 和数据类型 int 。(&N 预设的表达方式为十六进制，不是一般的数据类型 int)。上述两个表示式也可以写成

```
(float) N
(int)&N
```

效果完全一样。

Upcast（向上转换数据类型）

Upcast 只发生在有继承关系的类之间。假设有一个名叫 Circle 的派生类继承自名叫

Shape 的基类，如图 20.1.1 所示：

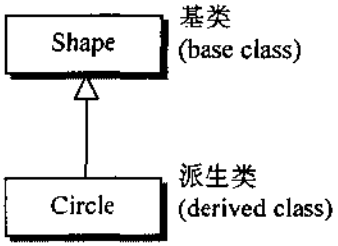

图 20.1.1　Shape 和 Circle 之间的类继承关系

我们定义一个名叫 C1 的 Circle 对象：

```
Circle C1;
```

则名叫 pS 的 Shape 指针和名叫 S1 的引用分别可以与 C1 的地址和 C1 本身有以下的赋值（assignment）关系：

```
Shape *pS = &C1;      //  Shape 指针指向 C1
Shape &S1 = C1;       //  定义 C1 的引用 S1
```

这两个运算就是典型的“upcast”。我们将它们的关系以“up”来描述，原因是在类继承图上，基类通常画在派生类的上面，而这些运算将派生对象（例如 C1）的地址传给了基类的指针（例如 pS）或引用（例如 S1），是一种“向上转型”的动作。

Upcast 是可以理解的，因为“继承”原来的用意就在于沿用既有的接口。所以基类的指针和引用分别可以接受派生对象的地址和对象本身。

提示

内置的数据类型间并没有继承的关系，因此，即使 int 和 float 都使用相同的地址记录方式，定义一个名叫 N 的 int 变量：

```
int N = 5;            //  定义一个名叫 N 的 int 变量
```

下列两个语句都无法通过编译：

```
float *pF = &N;      //  错误!
float &F1 = N;       //  错误!
```

这是因为 C++ 对于数据类型的检查非常严格的缘故.

有了 upcast 的概念，以下我们来看看多态可以带来什么样的便利. 延续上述的例子，假设有两个类，Circle (圆形) 继承自 Shape (图形)，而且各类内部都有各自不同版本的成员函数 Rotate()和 Erase()，如下面的类继承图所示（见图 20.1.2）：

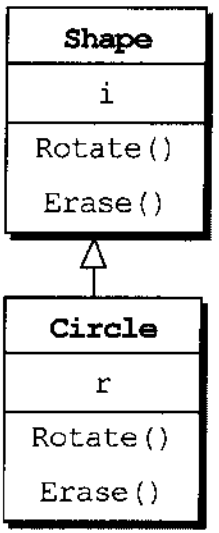

图 20.1.2　Circle 和 Shape 两个类之间的继承关系

由于能够进行 upcast 的处理，我们可以另外设计两个函数 Turn()和 Remove()，以一致的方式，使用基础对象的引用 (S1) 或指针 (pS) 来调用派生对象的成员函数 Rotate()和 Erase()：

```
void Turn(Shape &S1)     {S1.Rotate();}
void Remove(Shape *pS)   {pS->Erase();}
```

在上式中“pS->Erase();”也可以写成“(*pS).Erase();”.

也就是说，我们希望能够在定义了如下的派生对象 C1 及其指针 pC1 之后：

```
Circle C1;
Circle *pC1 = &C1;
```

可以直接以下述的方式调用到正确版本的 Rotate()和 Erase()：

```
Turn(C1);          希望调用   Circle::Rotate()
Remove(pC1);       希望调用   Circle::Erase()
```

不管是正方形、三角形，还是圆形，它们都是图形；如果可以用一致的方式处理，则不仅表达方式简明，而且可以避免错误．以计算机辅助设计 (CAD) 的应用为例，每个绘图单元都是图形对象，但不同的形状有其不同版本的 Rotate()和 Erase()，可以对不同的图形对象个别进行旋转和清除的工作．对使用者而言，我们在窗口下点选“旋转”和“清除”的图标 (icon) 时，不需要再指明图形的种类就可直接完成工作。

提示

对于上述问题而言，使用函数的重载（overloading）可以达到同样的效果，但是必须为每一个派生类写出一套同样名为 Turn()和 Remove()的函数．如果能够使用多态，我们只要写出一套共享的函数即可。

应用 upcast 的原理，我们可以试着写出下列可以通过编译的完整程序 Upcast.cpp：

范例程序 文件 Upcast.cpp

```
// Upcast.cpp
#include <iostream>
using std::cin;
using std::cout;
using std::endl;
//-------- 声明类 Shape ----------------
class Shape
{
  int i;
public:
  Shape(): i(7)  {}
  ~Shape()       {}
  void Rotate()  {cout<<"将图旋转 \n";}
  void Erase()   {cout<<"将图清除\n";}
};

//-------- 声明类 Circle-----------------
class Circle : public Shape
{
  int r;     //r是 private 变量
public:
  Circle(): r(5)          {}
  Circle(int N): r(N)     {}
  ~Circle()               {}
  void Rotate()  {cout<<"将圆形旋转 \n";}
  void Erase()   {cout<<"把圆形清除\n";}
};
void Turn  (Shape &S1) { S1.Rotate();}
void Remove(Shape *pS) { pS->Erase();}
// 也可写成 void Remove(Shape *pS) {(*pS).Erase();}

// ------- 主程序 -------------------------------
main()
{
```

```
    Circle C1;
    Circle *pC1 = &C1;
    cout << "执行 “Turn(C1)” 之后: " << endl;
    Turn(C1);
    cout << "执行 “Remove(pC1)” 之后: " << endl;
    Remove(pC1);
    cout << "C1 有 "
         << sizeof(C1)/sizeof(int)
         << " 个 int." << endl;
}
```

程序执行结果

```
执行 “Turn(C1)” 之后:
将图旋转
执行 “Remove(pC1)” 之后:
将图清除
C1 有 2 个 int.
```

讨 论

虽然程序可以执行，但是 Turn()和 Remove()分别都只调用到基类的成员函数：

```
Shape::Rotate()
Shape::Erase()
```

而不是

```
Circle::Rotate()
Circle::Erase()
```

要解决这个问题必须使用下一节即将介绍的虚函数（virtual function）。

20.2 后期绑定与虚函数

将函数的调用与函数本体间建立起关系叫做绑定（binding）。到目前为止，这个绑定的动作在编译阶段就已完成，称为前期绑定（early binding）。在 20.1 节中，以 upcast 得到的对象地址无法找到正确的派生类中的成员函数的原因，就在于绑定太早完成，以致于无法改变。

要改变这个过早的绑定，也就是希望在执行的时候才临时进行绑定，就必须将基类中的相关成员函数声明为虚函数（virtual function），这个改变只要在这些成员函数的返回数据类型（return data type）前加上关键词 `virtual` 即可。例如：

```
virtual void Rotate(){cout<<"将图旋转\n";}
virtual void Erase() {cout<<"将图清除\n";}
```

执行时才进行的绑定称为后期绑定（late binding）、执行期绑定（runtime binding）或动态绑定（dynamic binding）。

在以下即将介绍的程序 Poly.cpp 中，我们扩展了上节的继承关系，不仅在类 Shape 下继承了三个派生类 Square、Triangle 和 Circle，又在类 Circle 下继承了一个派生类 Cylinder，如图 20.2.1 所示。此外，在类 Shape 中，两个成员函数 `Rotate()` 和 `Erase()` 都声明为虚函数（virtual function）。

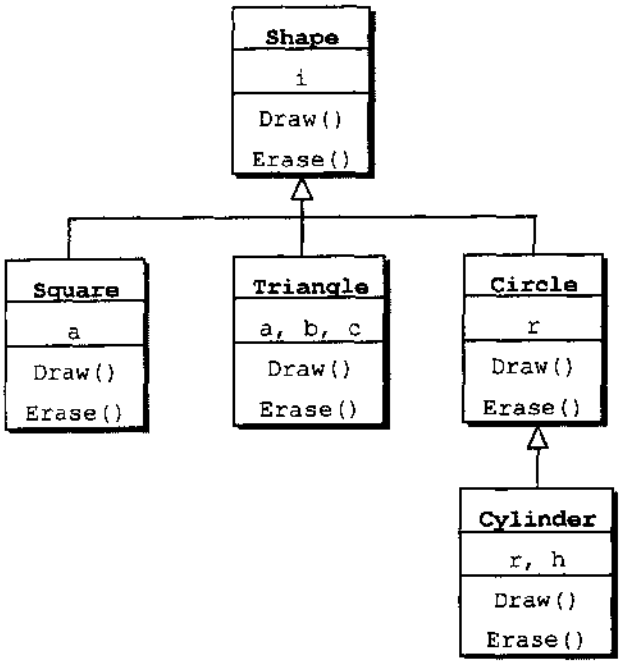

图 20.1.3　五个类之间的继承关系

范例程序　文件 Poly.cpp

```
// Poly.cpp
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

//----- 声明类 Shape ------------------------------
class Shape
```

```
{
  private:
   int i;
  public:
   Shape(): i(7){}
   ~Shape(){}
   virtual void Rotate() {cout<<"将图旋转\n";}
   virtual void Erase()  {cout<<"将图清除\n";}
};

//----- 声明类 Circle ----------------------------
class Circle : public Shape
{
  private:
   int r;
  public:
   Circle(): r(5)        {}
   Circle(int N): r(N)   {}
   ~Circle()             {}
   void Rotate(){cout<<"将圆形旋转\n";}
   void Erase() {cout<<"把圆形清除\n";}
};

//---- 声明类 Square ----------------------------
class Square : public Shape
{
private:
  int a;
public:
  Square(): a(2)      {}
  Square(int N): a(N) {}
  ~Square()           {}
  void Rotate() {cout<<"将正方形旋转\n";}
  void Erase()  {cout<<"把正方形清除\n";}
};

//---- 声明类 Triangle --------------------------
```

```
class Triangle : public Shape
{
private:
  int a, b, c;
public:
  Triangle(): a(1), b(1), c(1)      {}
  Triangle(int L, int M, int N): a(L), b(M), c(N) {}
  ~Triangle()           {}
  void Rotate()  {cout<<"将三角形旋转\n";}
  void Erase()   {cout<<"把三角形清除\n";}
};

//---- 声明类 Cylinder ---------------------------
class Cylinder : public Circle
{
private:
  int r, h;
public:
  Cylinder(): r(5), h(1)   {}
  Cylinder(int M, int N): r(M), h(N) {}
  ~Cylinder()                 {}
  void Rotate()  {cout    <<"将圆柱形旋转\n";}
  void Erase()   {cout    <<"把圆柱形清除\n";}
};

void Turn(Shape &S1) {S1.Rotate();}
void Remove(Shape *pS) {pS->Erase();}

// ---- 主程序 ------------------------------------
main()
{
  Circle   C1;
  Cylinder CyL;
  Triangle T2;
  Square   Sq3;
  cout   << "T2 有 "
         << sizeof(T2)/sizeof(int)
```

```
        << " 个 int." << endl;
    cout  << "CyL 有 "
        << sizeof(CyL)/sizeof(int)
        << " 个 int." << endl;
    cout  << " "Turn(C1))"  : ";
    Turn(C1);
    cout << " "Turn(CyL))" : ";
    Turn(CyL);
    cout << " "Turn(T2))"  : ";
    Turn(T2);
    cout << " "Turn(Sq3))" : ";
    Turn(Sq3);
    cout << "执行 "pS=&C1"  之后: " << endl;
    Shape *pS=&C1;
    cout << " "Remove(pS)" : ";
    Remove(pS);
    cout << "执行 "pS=&CyL"  之后: " << endl;
    pS=&CyL;
    cout << " "Remove(pS)" : ";
    Remove(pS);
    cout << "执行 "pS=&T2"  之后: "  << endl;
    pS=&T2;
    cout << " "Remove(pS)" : ";
    Remove(pS);
    cout << "执行 "pS=&Sq3"  之后: " << endl;
    pS=&Sq3;
    cout << " "Remove(pS)" : ";
    Remove(pS);
}
```

程序执行结果

```
T2 有 5 个 int.
CyL 有 5 个 int.
"Turn(C1))" : 将圆形旋转
"Turn(CyL))" : 将圆柱形旋转
"Turn(T2))" : 将三角形旋转
"Turn(Sq3))" : 将正方形旋转
执行 "pS=&C1"  之后:
"Remove(pS)" : 把圆形清除
执行 "pS=&CyL"  之后:
"Remove(pS)" : 把圆柱形清除
执行 "pS=&T2"  之后:
"Remove(pS)" : 把三角形清除
执行 "pS=&Sq3"  之后:
"Remove(pS)" : 把正方形清除
```

我们可以从执行结果知道，将基类 Shape 中的成员函数声明为虚函数 (virtual function) 后，程序就可以按照原先的预期，调用到正确版本的成员函数 `Rotate()`和 `Erase()`了。而且这个效果可以扩展到以下的所有派生类之中，包括类 Shape 的三个派生类 Square、Triangle 和 Circle 以及在类 Circle 下的派生类 Cylinder。这就是“多态”(polymorphism) 所带来的效果。

至于有了虚函数后，T2 和 CyL 以函数 `sizeof()` 检查的结果比预期的还多了 32 bits 的原因，要等到 20.3 节才能了解。

对象分割（object slicing）

在程序 Poly.cpp 中，`Turn()`和 `Remove()`的参数分别为引用（reference）和指针。为了利用 upcast，以达到多态的效果，这是必要的两种做法。

如果我们把这两个函数改为传对象的值：

```
void Turn(Shape S1){ S1.Rotate();}
void Remove(Shape S1){ S1.Erase();}
```

则程序仍然能够通过编译，但执行的结果变成没有多态的效果。这是因为 upcast 时虽然可以

接受传值的做法，但原来的位置只容得下基类的成员，在派生类中定义的成员都必须舍弃。这种现象叫做对象分割（object slicing），因为对象以传值的方式作为自变量时，有部分成员被分割，遗失不见了。

■ 重新定义（redefinition）与超越（overriding）

在派生类中定义与基类同名的成员函数称为重新定义（redefinition），可以改变成员函数的功能。如果重新定义的对象是虚函数的话，则称为超越（overriding）。

20.3 VPTR 和 VTABLE

类和对象的大小是其中所有数据成员大小的总和，在程序 Poly.cpp 中，各类中含有的数据成员分别为：

```
Shape:      int i
Circle:     int r
Cylinder:   int r, h
Triangle:   int a, b, c;
```

但以函数 sizeof() 检查，Triangle 和 Cylinder 所定义的对象 T2 和 CyL 大小都是 5 个 int，而不是 4 个。这是因为如果基类中有虚函数时，编译器会自动为每一个由该基类及其派生类所定义的对象加上一个叫做 v-pointer 的指针，简称 **VPTR**。这个指针就是对象大小增加的原因。

事实上，编译器还为每个类加上了一个叫做 v-table 的表，简称 **VTABLE**。VPTR 指向 VTABLE 开头的地方，如图 20.3.1 所示。在图 20.3.1 中，对象 C1 的 VPTR 指向类 Circle 的 VTABLE，而对象 CyL 的 VPTR 指向类 Cylinder 的 VTABLE。

一开始，由于指令

```
Shape *pS=&C1;
```

指针 pS 指向对象 C1 的 VPTR (图中的①)，而随后的赋值语句

```
pS = &CyL;
```

指针 pS 又改指向对象 CyL 的 VPTR (图中的②).

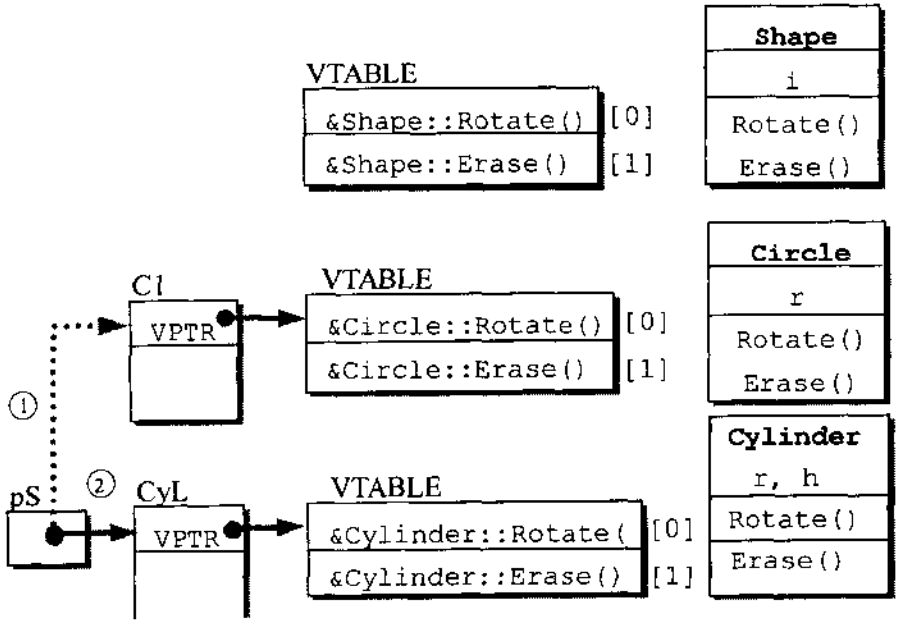

图 20.3.1　VPTR 和 VTABLE 之间的关系

在各 VTABLE 中记录的是各个类里面，所有在该类内或其基类内声明为 virtual 的成员函数地址 (成员函数的程序代码所在的地址).

如果在派生类中没有超越基类的成员函数，则在 VTABLE 中记录的是基类成员函数的地址．如果在派生类中有超越基类的成员函数，则在 VTABLE 中记录的是派生类成员函数的地址．这些地址都在 VTABLE 内按照它们在基础函数内的次序排列．

例如，假设在类 Cylinder 中没有定义 Erase()，则 Cylinder 的 VTABLE 内容变成：

```
&Cylinder::Rotate()
```

```
&Circle::Erase()
```

而不是

```
&Cylinder::Rotate()
&Cylinder::Erase()
```

一个对象只有一个 VPTR，通常 VPTR 位于对象内的开头位置，在对象被建立时就会被初始化成指向正确 VTABLE 开头的位置。

以程序 Poly.cpp 为例，当我们将 pS 存入 CyL 的地址时：

```
pS = &CyL;
```

则语句

```
Remove(pS);
```

所绑定的函数位于类 Cylinder 的 VTABLE 内的 VPTR+1 处 （因为 Erase()位于每个 VTABLE 的第二个字段处），亦即下标为 [1] 的地方，储存的内容是

```
&Cylinder::Erase()
```

所以可以调用到正确的成员函数版本（参考图 20.3.1）。 如果没有使用虚函数，则由于早期绑定 (early binding) 的关系，只能调用到：

```
Shape::Erase()
```

这就是多态的基础：后期绑定（late binding）内部运作的原理。

关于运算效率的考虑

当基类含有虚函数（virtual function）时，编译器会自动进行以下的动作：

1. 为基类和每个派生类都设定一个 VTABLE。
2. 对基类和它的所有派生类都加入一个 VPTR，并对所有由这些类所定义对象内的 VPTR 进行初始化的动作：将各个 VPTR 指向正确 VTABLE 的第一个字段。
3. 对每个 upcast 的虚函数调用都加入专用的程序代码，以动态设定基类指针，指向正确的 VPTR。

这些额外的设定当然会影响程序的执行效率，这也是多态所需要付出的代价。所幸这

些动作都是由编译器自动完成的，而且如果没有使用多态，所需要的其它做法往往更没有效率。

此外，为了使范例程序简洁，本章的所有成员函数都写成内联函数（inline functions）。(注意，即使没有使用关键词 inline，在类声明内定义的成员函数都是 inline 函数)。原本 inline 函数是对于语句很少的成员函数才适用，否则大量的程序代码直接加入程序中，将严重拖累执行效率。但是由以上的讨论知道，即使原本成员函数很简短，加上 virtual 的设定后，实际的程序代码将会大量增加。因此，在考虑效率的前提下，所有的虚函数都不应该以 inline 函数的方式存在。

20.4 纯虚函数 (pure virtual function) 与抽象类 (abstract class)

一般的基类都有完整的成员函数，因此也可以使用它来声明新的对象。但是，有时候我们想要避免使用基类来声明新的对象。以本章的例子而言，我们固然可以声明各种大小的圆形和圆柱形，但是如果我们只单纯的要定义一个“形状”是没有意义的。而且当这种情况发生时，往往代表在逻辑上出了问题或是发生了对象分割（object slicing）的现象，应该要加以避免。

为了阻止使用基类来定义对象，并能在编译的阶段就能够找出这个现象，可以将 virtual 函数改为 **pure virtual** 函数（pure virtual function，纯虚函数）。pure virtual 函数的语法非常简单，只要将 inline 形式的 virtual 函数本体部分简化为“= 0;”即可。例如：

```
virtual void Rotate()  = 0;
virtual void Erase() = 0;
```

含有纯虚函数的类称为抽象类（abstract class）。如果类内的所有虚函数都是纯虚函数，则此基类称为纯抽象类（pure abstract class）。

不论是抽象类还是纯抽象类都无法用来定义实例 (instance)。类的抽象化是避免错误的重要手段。这时候，如果我们想要使用基类声明对象，就会在编译时收到：“无法使用抽象类创造实例（cannot create instance of abstract class）”的错误信息。

为了具有代表性，可以完整说明上述语法，同时又为了避免程序过于冗长，我们在程序