用 C++ 中的子类型实现软件再用之探讨

蓝雯飞 （中南民族学院计算机科学系 武汉 430074）

摘要：本文讨论了 C++ 中通过公有继承实现的子类型关系。阐述了通过对类型子类型化提高了程序的灵活性和软件的重用率，并结合例子说明了子类型化在程序设计中的应用。

关键词：继承 子类型 多态性 虚函数 动态束定

公有继承实现抽象之间的一般和特殊的关系，这个关系从语言的角度而言，被称为子类型关系。这是继承在面向对象程序设计中具有重要地位的主要原因。本文以 C++ 语言为例着重对继承的这一应用进行讨论。

1. 什么是子类型

一个特定的类型 S，当该类型 S 至少提供了类型 T 的行为，就称类型 S 是类型 T 的子类型。这样，类型 S 的对象可以象类型 T 的对象那样被使用。

C++ 提供的公有继承机制能够实现子类型。因为类是类型(的一种实现)，通过从类型 T 公有继承实现类型 S，类型 S 自动具备类型 T 中的操作，类型 T 中的操作可以被用于类型 S 的对象。当我们考察一个操作类型 T 的对象的程序段时，我们可以得出这样的结论：如果将类型 S 的对象交给这段程序处理，丝毫不会影响到这个程序段的功能；因而可实现软件再用。

2. 多态的形式

一段程序能够处理多种类型的对象的能力被称为多态性。在 C++ 中，这种多态性可以通过几种形式来实现：

(1)强制多态。例如，在需要一个实数操作数的地方可以使用一个整数操作数，通过类型强制，使这段程序呈现多态性。

(2)重载多态。例如，通过定义一个能对两个实数进行某种操作的函数和定义一个能对一个实数和一个整数进行该操作的函数，使这段程序呈现多态性。

(3)参数化多态。通过对一个程序段所操作的对象的类型参数化，使这段程序呈现多态性。

(4)包含多态。通过子类型化，一个程序段既能够处理类型 T 的对象，也能够处理类型 T 的子类型 S 的对象。本文着重对这种形式的多态进行讨论。

3. 包含多态

在 C++ 语言中，将类型 T 的子类型 S 的对象交给处理类型 T 的程序段进行处理即是基类型(指针或引用所指向或引用对象的类型)为 S 的指针或引用被用于基类型是 T 的指针或引用可以使用的场合，即：

- S* 或 S& 分别适应 T* 或 T&；
- S* 或 S& 分别适应 const T* 或 const T&；
- const S* 或 const S& 分别适应 const T* 或 const T&。

考察下面的例子：

```
#include <iostream.h>
#include <math.h>
//point.h
class point //点类
{
  double x,y;
  public:
   point(double x1,double y1){x=x1;y=y1;}
   void showarea(){cout<<"area of point is:"<<
0.0<<endl;}
   double getx(){return x;}
   double gety(){return y;}
};
//circle.h
class circle:public point //圆类
{ double r;
  public:
  circle(double x1,double y1,double r1):point(x1,
y1){r=r1;}
   void showarea()
   {cout<<"area of circle with "<<r<<" is:"
<<3.14*r*r<<endl;}
};
double distance(const point& r1,const point& r2) //
计算两个形状之间的距离
{ double x=s1.getx()-s2.getx();
  double y=s1.gety()-s2.gety();
```

```
    return sqrt(x * x + y * y); }
  void main()
  { point pt1(1,1), pt2(7,8);
    circle c1(3,3,7), c2(4,4,5);
    cout<<"pt1<--->pt2"<<distance(pt1, pt2)
<<endl;
    cout<<"pt1<--->c1"<<distance(pt1, c1)<
<endl;
    cout<<"c2<--->pt2"<<distance(c2, pt2)<
<endl;
    cout<<"c1<--->c2"<<distance(c1, c2)<<
endl; }
```

在这个例子中，如果没有公有继承所建立的子类型关系，我们必须针对计算各种形状的距离的各种可能组合，重载定义操作不同参数类型的 distance 函数，这将是一个很大的负担。通过这个例子，我们可以理解子类型关系给程序设计所带来的益处：它使我们能够重复引用某个程序段来处理多种类型的对象。这种再用与以前通过函数调用进行代码再用有本质的不同，这种再用使程序有更大的灵活性。因此，公有继承结构的建立是 C++ 程序设计的重要内容，这种继承结构所建立的类型之间的关系也被称为 IS-A(是一个)关系。

4. 动态束定与虚函数

派生类继承了基类的所有的操作，或者说，基类的操作能被用于操作派生类的对象。但经常的情况是，我们在派生类中还声明有数据成员，当要求基类中声明的操作能够适应派生类中的这种情况时，简单的继承就不能满足我们的需要，所以，经常的情况是，我们在派生类中给出这个操作的新的实现，例如 void circle::showarea()。

下面考察将 circle 类的对象作为 point 类的对象处理的情况。

```
  #include <iostream.h>
  #include "point.h"
  #include "circle.h"
  void disparea(const point * p)
  { p->showarea(); }
  void main()
  { circle c1(1,1,1);
    disparea(&c1); }
```

程序的运行结果为 0.0(我们要的正确结果应为 3.14)。出什么问题了呢？

继承结构为程序提供了一定程度的灵活性(多态性)，这种灵活性使程序在被编译时无法知道参数 p 将指向什么类型的对象。编译器总假定参数 p 所指向的对象是在参数声明中所指定的静态类型的对象，以便使表达式：

```
p->showarea()
```

中的函数调用能够在编译时被束定到函数体上。由于 p 的静态类型是 point 类，所以，静态束定的结果使这个表达式中的函数调用执行函数：

```
void point::showarea(){cout<<"area of point is:"<
<0.0<<endl;}
```

因此，程序的输出结果为 0.0。

当程序员在实现一个子类型时变动了基类中的操作的实现时，C++ 提供虚函数机制可将这种变动告诉编译器，即将关键字 virtual 放在类 point 中该函数的函数说明之前(virtual void showarea())，程序其他部分保持不变(circle::showarea()自动地成为虚函数)，编译器就不会对函数调用 p->showarea()进行静态束定(在编译/连接时进行的束定)，而产生有关的代码，使函数调用与它所应执行的代码的束定工作在程序运行时进行。在程序运行时进行的束定被称为动态束定。

利用虚函数，可以在基类和派生类中使用相同的函数名，而定义函数的不同实现，从而实现"一个接口，多种方法"。当用基类指针或引用对虚函数进行访问时，软件系统将根据运行时指针或引用所指向或引用的实际对象来确定调用对象所在类的成员函数版本。当指针或引用指向或引用不同的对象时，执行的是虚函数的不同版本。在动态束定的多态性中，消息的实现代码是在运行时选定的；当一个程序正在运行时，软件系统为给定的对象选择合适的实现代码；消息被发送到一个指向对象的指针或对对象的引用，而不是直接发送到对象本身。

5. 结束语

由于静态束定是在程序被编译时进行的，所以，这种束定没有运行时开销。动态束定是在程序运行时进行的，为进行束定，需要一些运行时间开销，因此，相对于动态束定，静态束定不影响程序的运行效率。但 C++ 对动态束定的实现是很高效的。因此，在面向对象的程序设计中有效地利用子类型化，可以提高程序的可扩充性、灵活性及软件的重用率。

参考文献

[1] 蓝雯飞.C++ 中的多态性及其应用.计算机时代，1998,7.

[2] 张国峰.C++ 语言及其程序设计实用教程.北京：电子工业出版社，1997

[3] 谭浩强.C++ 程序设计教程.北京：中国科学技术出版社，1994

(来稿时间：1999 年 1 月)