DIVISION OF COMPUTER SCIENCE

Overloading and Polymorphism in the interpretation of Inheritance in C++

Mary Buchanan

Technical Report No.202

July 1994

Contents

1 Introduction

The work described in this report was motivated by a desire to understand the implementation of inheritance in C++. In particular a paper by W. Harris entitled "Contravariance for the rest of us" [1] had raised issues concerning the interpretation of overloading in inheritance. We wished to establish whether overloading in C++ is used to maintain the subtype relationship in the way that Harris suggests.

Other work on contravariance [2] and [3] has addressed the contravariance-covariance debate and C++ is interesting in that it permits neither contravariance nor covariance. This 'no-variance' policy was thought to be worth investigating in relation to subtyping.

Before exploring C++, we explain the way in which we use the terms overloading and polymorphism and introduce the C++ concept of virtual functions. Our experiments are concerned with virtual functions since it is these functions which enable inheritance polymorphism to be realised and hence give C++ its object-oriented flavour.

1.1 Overloading and Polymorphism

A function is said to be overloaded if, within the same scope, there exists more than one function with the same name but different parameters. A function call will be resolved by the parameter type such that different code will be invoked for arguments of different types. Strachey identified universal and ad hoc polymorphism. Overloading has been classified as *ad hoc* polymorphism by Cardelli and Wegner [4] in their seminal paper "On Understanding Types, Data Abstraction, and Polymorphism". Overloading is ad hoc because the different functions called may behave in semantically different ways.

Another form of polymorphism is *inclusion polymorphism* [4]. An example of this is subtyping; a value of a subtype may be used in any context which requires a value of the supertype. In object-oriented programming, (inclusion) polymorphism can be defined as the ability to have more than one dynamic type [5]. Entities may become attached to different objects at run time.

In this report we use the term overloading rather than overloading polymorphism and polymorpism rather than inclusion polymorphism. Although it may appear somewhat eccentric, this terminology is widely used in object-oriented texts.

Polymorphism in object-oriented languages refers to the ability to define an interface in a base class such that objects can be manipulated uniformly through this interface, regardless of whether they are base class or derived class objects. The function actually called via the interface will depend on the class of the object on which the function is invoked. Although different functions are called, the functions are expected to have the same semantic behaviour.

1.2 Virtual functions

In C++ functions are statically bound at compile time unless they are declared as virtual functions. Non-virtual function calls are completely resolved at compile time. In order to effect the virtual mechanism, each **object** of a class with virtual functions has a pointer to a table of pointers. The pointers in the table address the virtual function definitions [6]. A table of pointers exists for every class that defines, or redefines, virtual functions. A call to a virtual function is transformed by the compiler into an indirect call such that the address of the appropriate function is obtained from the table, the address dereferenced and the resulting function parameterised with the original parameters. An example of this process is given in Ellis and Stroustrup [6, page 228].

Virtual functions are dynamically bound at run time. If a base class contains a virtual function, then a derived class can "override" the function by defining a function with the same name and signature. A call of the function by an object of the derived class will invoke the derived function and this will happen regardless of whether the object is being accessed via a pointer or reference to the base class or to the derived class.

If the derived class function differs in argument type from the base class function of the same name, then the derived function is considered a different function and there will be no dynamic

link between the functions.

In C++ function calls can be made via objects or via pointers or references to those objects. If a call is made via an object, then the exact type of the object is known at compile time and the call can be resolved directly without resort to the virtual mechanism. When a virtual function is called through a pointer or reference, the actual type of the object is not necessarily known and the virtual mechanism must be used.

A base pointer may be assigned the address of a derived object; the pointer then has static type base but dynamic type derived. The interpretation of a call of a nonvirtual member function depends only on the type of the pointer denoting that object, that is the static type of the pointer. Hence a call to a nonvirtual member function for a pointer of base class will invoke the base class function regardless of the class of the object actually pointed to by the pointer. Static binding to the base pointer is effected. The interpretation of a call of a virtual member function depends on the type of the specific object being pointed to; to achieve this dynamic binding is necessary.

2 Redeclaration and redefinition of virtual functions

In this report the definitions of the terms redefinition and redeclaration are the same as those used in [7]. They are taken from Böszörmenyi [8].

Redefinition is defined as changing the implementation of a function while keeping the signature and specification the same. Redeclaration means changing the signature, specification and the implementation.

2.1 Redeclaration of a virtual function

In this section we examine the effect of redeclaring a function declared as virtual in a base class. A class hierarchy describing geometric points is used to illustrate the results of such function redeclaration.

The class Point2D defines a two dimensional point with coordinates x and y. The function *move* takes two integer arguments which are used to move a point object such that the first argument increments the x coordinate and the second, the y coordinate. The function *show* displays the x and y values of a Point2D object. The class implementation is shown below.

```
class Point2D {
    int x,y;
  public:
    Point2D(int xx,int yy) { x=xx; y=yy;}
    int X(void) { return x; }
    int Y(void) { return y; }
    virtual void move(int a, int b) {
        cout << ''Point2D::move called\n'';
        x=x+a; y=y+b;
    }
    virtual void show(void) {
        cout << ''Point2D('' << x << '','' << y << '')'';
    }
  };
```

Both the functions *move* and *show* are declared virtual.

The class Point3D defines a three dimensional point with coordinates x, y and z. The class inherits publically from Point2D but redeclares the *move* function in order that all three coordinates are moved. The implementation is shown overleaf.

```
class Point3D : public Point2D {
    int z;
  public:
    Point3D(int xx,int yy, int zz):Point2D(xx,yy){ z=zz;}
    int Z(void) { return z; }
    void move(int a,int b, int c) {
        cout << ''Point3D::move called\n'';
        Point2D::move(a,b);
        z=z+c;
    }
    void show(void) {
        cout << ''Point3D('' << X()
                << '','' << Y()
                    << '','' << z << '')'';
    }
};
```

Class Point3D inherits the x and y coordinates from class Point2D. The function *show* is redefined to display all three coordinates. In the redeclared *move* function, x and y cannot be accessed directly since they are private in the base class. Consequently, it is necessary to call the Point2D:: move function in order to increment x and y. This illustrates the fact that a publically derived class can only access private members (and private member functions) defined in a base class via the public member functions of that base class.

The redeclaration of *move* such that its parameters are different from those in Point2D::move means that Point3D::move will not be invoked using the virtual mechanism. Instead the function will be statically bound at compile time. If the move function had been redefined, rather than redeclared, in class Point3D then the function would have been treated as virtual even if the keyword virtual did not precede it. If the redeclared function is declared virtual, then this would enable it to be redefined in classes derived from Point3D but it would still not be invoked dynamically via the Point2D move since the two functions are regarded as different.

What happens to the *move* function inherited from class Point2D? It has been suggested [1] that *move* will be overloaded in class Point3D such that both the inherited and the redeclared versions are applicable to instances of the class.

Tests were conducted to show what happens when *move* is invoked by (references to) point objects. The results of the tests using the gnu compiler, g++, are shown below.

The following declarations of objects and pointers to objects were used in the tests:

```
Point2D p2(22,22),*pp2;
Point3D p3(3,3,3),*pp3;
```

`Test1`

Move a Point2D with 2 integers

```
cout << ''move a 2D with 2 integers\n'';
pp2 = &p2;                          // Pointer pp2 addresses object p2
cout << ''2D point= ''; pp2->show(); cout << ''\n'';
cout << ''2 integer arguments = 10,10 ''; cout << ''\n'';
pp2->move(10,10);
pp2->show();
cout << ''\n\n'';
```

```
Result 1

    move a 2D with 2 integers
    2D point= Point2D(22,22)
    2 integer arguments = 10,10
    Point2D::move called
    Point2D(32,32)
```

The Point2D object addressed by pp2 responds as expected to *move* called with two integers. Since move is a virtual function, dynamic binding will have been invoked. (At least in theory, in practice the compiler may have been able to resolve the call to use static binding.)

```
Test 2
```

Move a Point3D with 3 integers

```
    cout << ''move a 3D with  3 integers\n'';
    pp3 = &p3;
    cout << ''3D point= ''; pp3->show(); cout << ''\n'';
    cout << ''3 integer arguments = 2,2,2''; cout << ''\n'';
    pp3->move(2,2,2);
    pp3->show();
    cout << ''\n\n'';

Result 2

    move a 3D with 3 integers
    3D point= Point3D(3,3,3)
    3 integer arguments = 2,2,2
    Point3D::move called
    Point2D::move called
    Point3D(5,5,5)
```

The object referenced by pp3 is moved as expected but in this case the *move* function will have been statically bound.

```
Test 3
```

Test whether *move* is overloaded in Point3D

```
Try to  move a Point3D with  2 integers

    cout << ''move a 3D with  2 integers\n'';
    pp3 = &p3;
    cout << ''3D point= ''; pp3->show(); cout << ''\n'';
    cout << ''2 integer arguments to move 10,10''; cout << ''\n'';
    pp3->move(10,10);
    pp3->show();
    cout << ''\n\n'';
```

This test produces a compile time error: too few arguments for method 'move'. The error suggests that *move* is resolved before the parameter type is evaluated which would not be the case in overloading.

The move function is not overloaded in Point3D. Two functions with the same name but different arguments can be overloaded if they are in the same scope. However, a function member of a derived class is not in the same scope as a function member of the same name in a base class [6]. The redeclaration of move in Point3D hides the Point2D move. In order to access the Point2D move for a Point3D object, it is necessary to use the scope resolution operator (::) as shown in the next test.

```
Test 4
```

Move a Point3D with 2 integers by coercing move to parent's move function

```
cout << ''move a 3D with 2 integers, coerced\n'';
pp3 = &p3;
cout << ''3D point= ''; pp3->show(); cout << ''\n'';
cout << ''2 integer arguments to move 10,10''; cout << ''\n'';
pp3->Point2D::move(10,10);
pp3->show();
cout << ''\n\n'';
```

```
Result 4

move a 3D with 2 integers, coerced
3D point= Point3D(5,5,5)
2 integer arguments to move 10,10
Point2D::move called
Point3D(15,15,5)
```

By specifically invoking the Point2D move function, it is possible to move a Point3D object with 2 integers. Obviously, this results in the z coordinate being unchanged. Explicit qualification with the scope resolution operator suppresses the virtual mechanism.

A direct result of the move function not being overloaded in Point3D is that Point3D is not a subtype of Point2D since the Point2D interface is not contained in the Point3D interface. This is discussed further in section 5.4.

Suppose the move functions are called through a base pointer referring to a derived class as shown below:

```
Test 5

Move a Point3D in a Point2D with  3 integers
```

```
cout << ''move a 3D in a 2D with  3 integers\n'';
p3 = Point3D(3,3,3);
pp2 = &p3; // Point2D pointer addresses a Point3D object
cout << ''3D point(in 2D*)= ''; pp2->show(); cout << ''\n'';
cout << ''3 integer argument 2,2,2''; cout << ''\n'';
pp2->move(2,2,2);
pp2->show();
cout << ''\n\n'';
```

This test will not compile. The error message refers to 'move' having too many arguments. A base class pointer can only respond to functions in the base class interface. The function *move* with 3 integers is not in the interface of class Point2D.

Since pp2 is of class Point2D and since *move* in Point2D is virtual, *move* invoked via pp2 will be dynamically bound. At run time pp2 is dynamically a Point3D reference so the search for a *move* function will begin in the scope of Point3D. Point3D does not contain a **redefined** *move* method so the *move* invoked will default to that defined in class Point2D which expects two integers as arguments. The *move* function **redeclared** with 3 integers in class Point3D is treated as a completely different function from the *move* defined in class Point2D thereby effectively disabling inheritance polymorphism for this function. Since the 3 integer *move* is first defined in class Point3D, it cannot be accessed via a call through a Point2D pointer.

If a move with 2 integers is invoked via a Point2D pointer which dynamically refers to a Point3D object, then since Point3D does not redefine move with 2 integers, the Point2D *move* is invoked:

```
Test 6
```

Move a Point3D in a Point2D with 2 integers

```
    cout << ''move a 3D in a 2D with 2 integers\n'';
    p3 = Point3D(3,3,3);
    pp2 = &p3;
    cout << ''3D point(in 2D*)= ''; pp2->show(); cout << ''\n'';
    cout << ''2 integer arguments 10,10''; cout << ''\n'';
    pp2->move(10,10);
    pp2->show();
    cout << ''\n\n'';
```

```
Result 6

    move a 3D in a 2D with 2 integers
    3D point(in 2D*)= Point3D(3,3,3)
    2 integer arguments 10,10
    Point2D::move called
    Point3D(13,13,3)
```

2.2 A virtual function neither redefined nor redeclared

If a virtual function is not redefined and not redeclared in a derived class then the base class function will be used in all calls.

We redefine class Point3D such that a *move* function is not included:

```
class Point3D : public Point2D {
    int z;
  public:
    Point3D(int xx,int yy, int zz):Point2D(xx,yy){ z=zz;}
    int Z(void) { return z; }
    void show(void) {
        cout << ''Point3D('' << X()
                << '','' << Y()
                    << '','' << z << '')'';
    }
};
```

Test 3 as above is run to establish what happens when a call to *move* with 2 integers is made for a Point3D object.

```
Result

move a 3D with a 2 integers
3D point= Point3D(3,3,3)
2 integer arguments to move 10,10
Point2D::move called
Point3D(13,13,3)
```

The Point2D *move* function is called dynamically via the virtual mechanism.

2.3 Redefinition of a virtual function

In this section we examine the effect of the *move* 2 integer argument function being declared virtual in class Point2D, being redeclared with an extra parameter in class Point3D and being redefined from Point2D with 2 integers in a class Point3DExt which inherits publically from class Point3D. Classes Point2D and Point3D are the same as those in section 2.1 above. They are reproduced here for convenience together with the definition of class Point3DExt.

```
class Point2D {
    int x,y;
  public:
    Point2D(int xx,int yy) { x=xx; y=yy;}
    int X(void) { return x; }
    int Y(void) { return y; }
    virtual void move(int a, int b) {
        cout << ''Point2D::move called\n'';
        x=x+a; y=y+b;
    }
    virtual void show(void) {
        cout << ''Point2D('' << x << '','' << y << '')'';
    }
};

class Point3D : public Point2D {
    int z;
  public:
    Point3D(int xx,int yy, int zz):Point2D(xx,yy){ z=zz;}
    int Z(void) { return z; }
    void move(int a,int b, int c) {
        cout << ''Point3D::move called\n'';
        Point2D::move(a,b);
        z=z+c;
    }
    void show(void) {
        cout << ''Point3D('' << X()
                << '','' << Y()
                    << '','' << z << '')'';
    }
};
```

```
class Point3DExt : public Point3D {
    public:
  Point3DExt(int xx, int yy, int zz):Point3D(xx,yy,zz){}
  void move(int a, int b){
    cout << ''Point3DExt::move called\n'';
  }
};
```

The *move* function in class Point3DExt has been implemented such that it does no more than merely output that it has been called. This is adequate for the purposes of our tests.

The following declarations of objects and pointers were used in the tests:

```
Point2D p2(22,22), *pp2;
Point3D p3(3,3,3), *pp3;
Point3DExt p33(4,4,4), *pp33;
```

```
Test 7
```

Move a Point3DExt with 2 integers

```
cout << ''move a 3DExt with 2 integers\n'';
pp33 = &p33;
pp33->move(5,5);
cout << ''\n\n'';
```

```
Result 7
```

```
move a 3DExt with 2 integers
Point3DExt::move called
```

The pointer object pp3 is both statically and dynamically of class Point3DExt. As expected, the Point3DExt::move is called and it will have been invoked dynamically. The next test invokes *move* with 2 integers on a Point3DExt object denoted by a base class pointer of class Point2D.

```
Test 8
```

Move a Point3DExt referred to by a Point2D pointer. Arguments of 2 integers.

```
cout << ''move a 3DExt in a 2D with 2 integers\n'';
pp2 = &p33;
pp2->move(6,6);
cout << ''\n\n'';
```

```
Result 8
```

```
move a 3DExt in a 2D with 2 integers
Point3DExt::move called
```

The *move* function will have been invoked dynamically such that the search for the function will have begun in the scope of the Point3DExt class. The redefined *move* will have been found and invoked.

Thus a function which is declared virtual in a base class and which has been redeclared in a derived class can still be redefined lower down the class hierarchy. Only the functions which redefine the original virtual function can respond dynamically. Any redeclared functions are regarded as entirely different functions.

The next test is designed to show what happens to the 3 integer *move* which is ostensibly inherited from Point3D by Point3DExt. Will the *move* operation be overloaded in Point3DExt or, as we might expect from Test 3 in section 2.1, will the 3 integer *move* be hidden?

`Test9`

Move a Point3DExt with 3 integers

```
cout << ''move a 3DExt  with 3 integers\n'';
pp33 = &p33;
pp33->move(1,2,3);
cout << ''\n\n'';
```

This test will not compile. The error message refers to too many arguments for method 'move'. The Point3D::move for 3 integers is hidden by the redefinition of the 2 integer move in Point3DExt. Functions with the same name but different parameters are not overloaded in inheritance. It is possible to move a Point3DExt with 3 integers by explicit invocation of the Point3D::move as shown below:

`Test 10`

```
cout << ''move a 3DExt  with 3 integers with
             explicit invocation of 3Dmove\n\'';
pp33 = &p33;
pp33->Point3D::move(1,2,3);
cout << ''\n\n'';
```

`Result 10`

```
move a 3DExt  with 3 integers with explicit invocation of 3Dmove
Point3D::move called
Point2D::move called
```

Similarly, it is possible to move a Point3DExt object with 3 integers if the object is addressed by a pointer to the Point3D base class:

`Test 11`

```
cout << ''move a 3DExt in a 3D with 3 integers\n'';
pp3 = &p33;
pp3->move(1,2,3);
cout << ''\n\n'';
```

```
Result 11

    move a 3DExt in a 3D with 3 integers
    Point3D::move called
    Point2D::move called
```

The Point3D::move will have been invoked statically, since the 3 integer move in Point3D has not been declared as virtual.

It is not possible to move with 2 integers a Point3DExt referenced by a Point3D pointer:

```
Test 12

    cout << ''move a 3DExt in a 3D with 2 integers\n'';
    pp3 = &p33;
    pp3->move(6,6);
    cout << ''\n\n'';
```

This test will not compile due to too few arguments for method 'move'. The function will be statically bound to the move declared for 3 integers and hence is unable to respond to a 2 integer argument.

2.4 Redefinition of a redeclared function and redefinition of a base class function

If the redeclared *move* function in class Point3D is declared as virtual it can be redefined in classes derived from class Point3D. In this experiment we redefine the 2D::move and the 3D::move in class Point3DExt. As before, these redefinitions are written merely for the experiment, they serve no other practical purpose.

```
class Point3D : public Point2D {
    int z;
  public:
    Point3D(int xx,int yy, int zz):Point2D(xx,yy){ z=zz;}
    int Z(void) { return z; }
    virtual void move(int a,int b, int c) {      // virtual declaration
        cout << ''Point3D::move called\n'';
        Point2D::move(a,b);
        z=z+c;
    }
    void show(void) {
        cout << ''Point3D('' << X()
                << '','' << Y()
                    << '','' << z << '')'';
    }
};

class Point3DExt : public Point3D {
    public:
  Point3DExt(int xx, int yy, int zz):Point3D(xx,yy,zz){}
  void move(int a, int b){  // redefine 2 integer move
    cout << ''Point3DExt::2intmove called\n'';
```

```
  }
  void move(int a, int b, int c){  // redefine 3 integer move
    cout << ''Point3DExt::3intmove called\n'';
  }
};
```

Tests 13 and 14 show that it is now possible to move a Point3DExt with either 2 integers or 3 integers, in other words *move* is overloaded in Point3DExt.

```
Test 13
```

Move a Point3DExt with 2 integers

```
    cout << ''move a 3DExt with 2 integers\n'';
    pp33 = &p33;
    pp33->move(5,5);
    cout << ''\n\n'';
```

```
Result 13

    move a 3DExt with 2 integers
    Point3DExt::2intmove called
```

```
Test 14
```

Move a Point3DExt in a 2D with 2 integers

```
    cout << ''move a 3DExt  with 3 integers\n'';
    pp33 = &p33;
    pp33->move(1,2,3);
    cout << ''\n\n'';
```

```
Result 14

    move a 3DExt  with 3 integers
    Point3DExt::3intmove called
```

Tests have confirmed that, as would be expected, it is possible to move a Point3DExt denoted by a Point2D pointer with 2 integers and to move a Point3DExt denoted by a Point3D pointer with 3 integers. It is not possible to move a Point3DExt denoted by a Point3D pointer with 2 integers since the Point3D interface does not contain a 2 integer move.

2.5 A parent class function redeclared in a child class but neither redefined nor redeclared in a grandchild class

In this version of Point3DExt, the *move* function is neither redefined nor redeclared. The tests already undertaken would suggest that the virtual 3 integer *move* in class Point3D will be inherited but that the virtual 2 integer *move* in class Point2D, which was hidden in class Point3D, will remain hidden. The following tests show that this is indeed the case.

```
class Point3DExt : public Point3D {
    public:
  Point3DExt(int xx, int yy, int zz):Point3D(xx,yy,zz){}
};
```

```
Test 15
```

Move a Point3DExt with 2 integers

```
    cout << ''move a 3DExt with 2 integers\n'';
    pp33 = &p33;
    pp33->move(5,5);
    cout << ''\n\n'';
```

This test will not compile due to too few arguments for method 'move'. The 2 integer move from Point2D is hidden by the redeclaration of *move* in Point3D.

```
Test 16
```

Move a Point3DExt with 3 integers

```
    cout << ''move a 3DExt  with 3 integers\n'';
    pp33 = &p33;
    pp33->move(1,2,3);
    cout << ''\n\n'';
```

```
Result 16

move a 3DExt  with 3 integers
Point3D::move called
Point2D::move called
```

Point3DExt only inherits the 3 integer *move* from Point3D.

3 Functions taking class arguments

We shall now consider what happens when the arguments to the *move* functions are represented not by integers but by objects of class Point2D and class Point3D.

These tests were carried out with the AT&T C++ compiler, CC, as well as the gnu compiler. The results were the same but the CC compiler was found to give more helpful error messages.

3.1 Redeclaration of a virtual function

In this example, the *move* function is redeclared in class Point3D such that it takes a pointer to Point3D argument rather than a pointer to Point2D argument. The class definitions are as follows:

```
class Point2D {
    int x,y;
  public:
    Point2D(int xx,int yy) { x=xx; y=yy;}
    int X(void) { return x; }
    int Y(void) { return y; }
    virtual void move(Point2D *d) {
        cout << ''Point2D::move called\n'';
        x=x+d->x; y=y+d->y;
    }
    virtual void show(void) {
        cout << ''Point2D('' << x << '','' << y << '')'';
    }
  };

class Point3D : public Point2D {
    int z;
  public:
    Point3D(int xx,int yy, int zz):Point2D(xx,yy){ z=zz;}
    int Z(void) { return z; }
    void move(Point3D *d) {
        cout << ''Point3D::move called\n'';
        Point2D::move(new Point2D(d->X(),d->Y()));
        z=z+d->z;
    }
    void show(void) {
        cout << ''Point3D('' << X()
                << '','' << Y()
                    << '','' << z << '')'';
    }
};
```

The CC compiler gives a useful warning: Point3D::move() hides virtual Point2D::move().

The object and pointer declarations used in the tests are shown below:

```
Point2D p2(22,22),delta2(11,11),*pp2;
Point3D p3(3,3,3),delta3(1,1,1),*pp3;
```

Test 17

Move a Point2D object with a Point2D argument.

```
cout << ''move a 2D with 2D\n'';
pp2 = &p2;
cout << ''2D point= ''; pp2->show(); cout << ''\n'';
cout << ''2D delta= ''; delta2.show(); cout << ''\n'';
pp2->move(&delta2);
pp2->show();
cout << ''\n\n'';
```

```
Result 17

    move a 2D with 2D
    2D point= Point2D(22,22)
    2D delta= Point2D(11,11)
    Point2D::move called
    Point2D(33,33)
```

```
Test 18
```

Move a Point3D object with a Point3D argument.

```
    cout << ''move a 3D with a 3D\n'';
    pp3 = &p3;
    cout << ''3D point= ''; pp3->show(); cout << ''\n'';
    cout << ''3D delta= ''; delta3.show(); cout << ''\n'';
    pp3->move(&delta3);
    pp3->show();
    cout << ''\n\n'';
```

```
Result 18


    move a 3D with a 3D
    3D point= Point3D(3,3,3)
    3D delta= Point3D(1,1,1)
    Point3D::move called
    Point2D::move called
    Point3D(4,4,4)
```

Tests 17 and 18 give the 'same' results as tests 1 and 2 for integer arguments.

```
Test 19
```

Move a Point2D object with a Point3D argument.

```
    cout << ''move a 2D with 3D\n'';
    pp2 = &p2;
    cout << ''2D point= ''; pp2->show(); cout << ''\n'';
    cout << ''3D delta= ''; delta3.show(); cout << ''\n'';
    pp2->move(&delta3);
    pp2->show();
    cout << ''\n\n'';
```

```
Result 19

    move a 2D with 3D
    2D point= Point2D(33,33)
    3D delta= Point3D(1,1,1)
    Point2D::move called
    Point2D(34,34)
```

Test 19 differs from any of the tests with integer arguments. With integer arguments it is not possible to move a Point2D object with 3 integers. It is possible to move a Point2D object with a Point3D argument because a Point3D is derived from, and hence conforms to a Point2D. Only the Point2D part of the Point3D argument is used, that is the x and y coordinates of delta3 are used to increment pp2.

It is not possible to move a Point3D object with a Point2D argument since a Point2D does not conform to a Point3D:

```
Test 20
```

Move a Point3D object with a Point2D argument.

```
cout << ''move a 3D with a 2D\n'';
pp3 = &p3;
cout << ''3D point= ''; pp3->show(); cout << ''\n'';
cout << ''2D delta= ''; delta2.show(); cout << ''\n'';
pp3->move(&delta2);
pp3->show();
cout << ''\n\n'';
```

This test will not compile: bad argument 1 for function 'void Point3D::move(class Point3D *)' (type was class Point2D *). The error message confirms that a class Point2D argument may not replace a class Point3D argument. Similarly for the CC compiler, the error message is "no standard conversion of Point2D* to Point3D*".

However, a Point3D object can be moved with a Point2D argument provided the Point2D::move is called specifically. Thus test 21 corresponds to test 4 for integers.

```
Test 21
```

Cooerce the move of a Point3D object with a Point2D argument

```
cout << ''move a 3D with a 2D, coerced\n'';
pp3 = &p3;
cout << ''3D point= ''; pp3->show(); cout << ''\n'';
cout << ''2D delta= ''; delta2.show(); cout << ''\n'';
pp3->Point2D::move(&delta2);
pp3->show();
cout << ''\n\n'';
```

```
Result 21

    move a 3D with a 2D, coerced
    3D point= Point3D(4,4,4)
    2D delta= Point2D(11,11)
    Point2D::move called
    Point3D(15,15,4)
```

The following test corresponds to test 5 for integers but unlike test 5, the test compiles due to the fact that a Point3D conforms to a Point2D.

```
Test 22
```

Move a Point3D object referenced by a Point2D pointer with a Point3D argument.

```
cout << ''move a 3D in a 2D with a 3D\n'';
p3 = Point3D(3,3,3);
pp2 = &p3;
cout << ''3D point(in 2D*)= ''; pp2->show(); cout << ''\n'';
cout << ''3D delta= ''; delta3.show(); cout << ''\n'';
pp2->move(&delta3);
pp2->show();
cout << ''\n\n'';
```

```
Result 22
```

```
move a 3D in a 2D with a 3D
3D point(in 2D*)= Point3D(3,3,3)
3D delta= Point3D(1,1,1)
Point2D::move called
Point3D(4,4,3)
```

As expected, it is possible to move a Point3D object referenced by a Point2D object with a Point2D argument. This test corresponds to test 6 for integers.

```
Test 23
```

Move a Point3D in a Point2D with a Point2D

```
cout << ''move a 3D in a 2D with a 2D\n'';
p3 = Point3D(3,3,3);
pp2 = &p3;
cout << ''3D point(in 2D*)= ''; pp2->show(); cout << ''\n'';
cout << ''2D delta= ''; delta2.show(); cout << ''\n'';
pp2->move(&delta2);
pp2->show();
cout << ''\n\n'';
```

```
Result 23
```

```
move a 3D in a 2D with a 2D
3D point(in 2D*)= Point3D(3,3,3)
2D delta= Point2D(11,11)
Point2D::move called
Point3D(14,14,3)
```

We can conclude that the differences (shown by tests 19 and 22) which arise when integer arguments are changed to class arguments are due solely to the fact that a derived class argument can be used in place of a base class argument.

3.2 Overloaded redefinitions

If we redefine both the Point2D::move and the Point3D::move in the class Point3DExt will we get overloaded *move* functions as in section 2.4 above? Since a Point3D conforms to a Point2D we

might expect such overloading to be disallowed on the grounds of ambiguity. Argument matching of overloaded functions in C++ is defined by Ellis and Stroustrup [6, page 312]. "A call of a given function name chooses, from among all functions by that name that are in scope and for which a set of conversions exist so that the function could possibly be called, a function that best matches the actual arguments." Sets of functions that best match on each argument are produced. If the intersection of the sets contains one member then this is chosen as the best-matching function; otherwise the call is illegal. Furthermore, the function selected must be a strictly better match for at least one argument than every other possible function.

```
   class Point3DExt : public Point3D {
    public:
      Point3DExt(int xx, int yy, int zz):Point3D(xx,yy,zz){}
   void move(Point2D *d){
    cout << ''Point3DExt::2Dmove called\n'';
    }
  void move(Point3D *d){
    cout << ''Point3DExt::3Dmove called\n'';
    }
};
```

```
Test 24
```

Move a Point3DExt object with a Point2D argument.

```
    cout << ''move a 3DExt with a 2D\n'';
    pp33 = &p33;
    pp33->move(&delta2);
    cout << ''\n\n'';
```

```
Result 24

    move a 3DExt with a 2D
    Point3DExt::2Dmove called
```

```
Test 25
```

Move a Point3DExt object with a Point3D argument.

```
    cout << ''move a 3DExt with a 3D\n'';
    pp33 = &p33;
    pp33->move(&delta3);
    cout << ''\n\n'';
```

```
Result  25

    move a 3DExt with a 3D
    Point3DExt::3Dmove called
```

When a call to *move* is made with a Point3D argument, either *move* method could be called since a Point3D can be used in place of a Point2D. Since the *move* function can be overloaded, we

conclude that the possible ambiguity is resolved because a Point3D argument is a strictly better match for the *move* declared with a Point3D argument than for *move* declared with a Point2D argument.

3.3 Using type casts to redefine within the virtual mechanism

In order to maintain dynamic binding of polymorphic types, functions which are redefined in derived classes must keep the same signature as the base class functions. In this experiment we use type casts such that the *move* function in class Point3D can be redefined to effect a three coordinate move while still being invoked dynamically through a Point2D interface. The use of type casts short-circuits the type checking mechanism and consequently should only be used as a last resort (if at all). Once casts have been used, the type checker can verify that the programmer has declared the operations invoked for the class that the parameter has been cast to but the type checker cannot prevent inadvertent use of a base class parameter which will have ill-defined results [9, page 511].

```
class Point2D {
    int x,y;
  public:
    Point2D(int xx,int yy) { x=xx; y=yy;}
    int X(void) { return x; }
    int Y(void) { return y; }
    virtual void move(Point2D *d) {
        cout << ''Point2D::move called\n'';
        x=x+d->x; y=y+d->y;
    }
    virtual void show(void) {
       cout << ''Point2D('' << x << '','' << y << '')'';
    }
};
class Point3D : public Point2D {
    int z;
  public:
    Point3D(int xx,int yy, int zz):Point2D(xx,yy){ z=zz;}
    int Z(void) { return z; }
    void move(Point2D *d) {
        cout << ''Point3D::move called\n'';
        Point3D *p;
        p = (Point3D*) d;
        Point2D::move(new Point2D(p->X(),p->Y()));
        z=z+p->z;
    }
    void show(void) {
        cout << ''Point3D('' << X()
                << '','' << Y()
                    << '','' << z << '')'';
    }
};
```

The function *move* in Point3D takes an argument of type pointer to Point2D but then uses a local pointer, p, of type Point3D to convert the argument to a Point3D. The z coordinate can thereby be moved as required for a Point3D.

Test 26

Move a Point3D object with a Point3D argument.

```
cout << ''move a 3D with a 3D\n'';
pp3 = &p3;
cout << ''3D point= ''; pp3->show(); cout << ''\n'';
cout << ''3D delta= ''; delta3.show(); cout << ''\n'';
pp3->move(&delta3);
pp3->show();
cout << ''\n\n'';
```

Result 26

```
move a 3D with a 3D
3D point= Point3D(3,3,3)
3D delta= Point3D(1,1,1)
Point3D::move called
Point2D::move called
Point3D(4,4,4)
```

Since a Point3D is derived from a Point2D, it is possible to have a Point3D argument for the *move* defined for a Point2D argument and thereby to effect a 3 coordinate move as required.

Test 27

Move a Point3D object with a Point2D argument.

```
cout << ''move a 3D with a 2D\n'';
pp3 = &p3;
cout << ''3D point= ''; pp3->show(); cout << ''\n'';
cout << ''2D delta= ''; delta2.show(); cout << ''\n'';
pp3->move(&delta2);
pp3->show();
cout << ''\n\n'';
```

Result 27

```
move a 3D with a 2D
3D point= Point3D(4,4,4)
2D delta= Point2D(11,11)
Point3D::move called
Point2D::move called
Point3D(15,15,38)
```

When a Point2D argument is used, it is converted to a Point3D argument and an attempt is made to access a non-existent z coordinate. As a result, the z coordinate is incremented by an arbitrary amount which in this test is 34. The intention is that the Point3D::move should never be called with a Point2D argument, but this is not enforced.

Test 28

Move a Point3D object referenced by a Point2D pointer with a Point3D argument.

```
cout << ''move a 3D in a 2D with a 3D\n'';
p3 = Point3D(3,3,3);
pp2 = &p3;
cout << ''3D point(in 2D*)= ''; pp2->show(); cout << ''\n'';
cout << ''3D delta= ''; delta3.show(); cout << ''\n'';
pp2->move(&delta3);
pp2->show();
cout << ''\n\n'';
```

Result 28

```
move a 3D in a 2D with a 3D
3D point(in 2D*)= Point3D(3,3,3)
3D delta= Point3D(1,1,1)
Point3D::move called
Point2D::move called
Point3D(4,4,4)
```

The virtual mechanism dynamically binds the Point3D object to the Point3D::move.

Test 29

Move a Point3D object referenced by a Point2D pointer with a Point2D argument.

```
cout << ''move a 3D in a 2D with a 2D\n'';
p3 = Point3D(3,3,3);
pp2 = &p3;
cout << ''3D point(in 2D*)= ''; pp2->show(); cout << ''\n'';
cout << ''2D delta= ''; delta2.show(); cout << ''\n'';
pp2->move(&delta2);
pp2->show();
cout << ''\n\n'';
```

Result 29

```
move a 3D in a 2D with a 2D
3D point(in 2D*)= Point3D(3,3,3)
2D delta= Point2D(11,11)
Point3D::move called
Point2D::move called
Point3D(14,14,37)
```

Test 29 shows that the Point3D::move has been invoked dynamically and as in test 27 the z coordinate is moved by an arbitrary amount.

The tests show that when a Point3D object is moved with a Point3D argument, the desired behaviour is obtained. However, there is nothing to prevent a Point3D object being moved with a

Point2D pointer argument. When this happens, the conversion of the argument, d, to a Point3D argument by the type casting means that the interpretation of the contents and size of memory addressed by d will be changed to that appropriate for a Point3D. Hence, the value for the 'z' coordinate will be whatever is addressed by the next memory location. When a Point3D is moved by a Point2D the value of the 'z' coordinate will be incremented by an arbitrary amount.

If the pointer passed to the *move* function is indeed of class Point3D then all is well. Such a use of type casts to enable the desired move function for three coordinates to be implemented places an onus on the programmer to ensure that a class Point3D is always passed to the function and not a class Point2D pointer.

According to The Annotated C++ Reference Manual [6], casting from a base class to a derived class is illegal. Although few implementations check for this, it would be legal for a compiler to disallow such a cast in which case the 'solution' proposed here for redefining the move function would not be allowed.

4 Private Inheritance

When a derived class inherits a base class privately, all public members of the base class become private members of the derived class. Objects of the derived class are not able to respond to base class functions. However, the base class functions are available within the derived class. We wished to ascertain whether overloading might occur *within* a derived class as seems to be implied by Harris [1].

```
class Point2D {
    int x,y;
  public:
    Point2D(int xx,int yy) { x=xx; y=yy;}
    int X(void) { return x; }
    int Y(void) { return y; }
    virtual void move(int a, int b) {
        cout << ''Point2D::move called\n'';
        x=x+a; y=y+b;
    }
    virtual void show(void) {
         cout << ''Point2D('' << x << '','' << y << '')'';
    }
};

class Point3D : private Point2D {
    int z;
  public:
    Point3D(int xx,int yy, int zz):Point2D(xx,yy){ z=zz;}
    int Z(void) { return z; }
    void move(int a,int b, int c) {
        cout << ''Point3D::move called\n'';
        Point2D:: move(a,b);
        z=z+c;
    }
    void show(void) {
        cout << ''Point3D('' << X()
               << '','' << Y()
                    << '','' << z << '')'';
    }
```

```
    void test(void) {
         this->show();
         this->move(20,20,20);
         this->show();
     }
};
```

Within the function move in the class Point3D definition, an attempt to invoke move(a,b) directly rather than via the scope resolution operator resulted in a compilation error (too few arguments). Similarly, in the function *test*, a call to this→move(2,2) would not compile.

It has been suggested that within a privately inherited class, a function such as *move* would be overloaded [1]. If this were so, both move(int, int) and move(int, int, int) would be in scope. However, the fact that move(a,b) was not permissible in the *move* function and that move(2,2) would not compile for the *test* function in class Point3D indicates that for the gnu compiler at least, overloading does not arise. The tests were repeated on the AT&T compiler, CC, and again overloading did not occur.

We must not lose sight of the fact that if *move* is not redeclared in class point3D, then it will be possible to call this→ *move*(2,2) in Point3D.

A derived class object may not be assigned to its private base class unless the assignment is within the scope of the derived class. For example, outside the scope of class Point3D the assignment pp2 = &p3 is illegal. On the gnu compiler it results in the error message: type 'Point3D' is derived from private 'Point2D'. Objects of class Point3D will not be able to respond to the interface of class Point2D, only to the interface of class Point3D. In other words, Point3D is not a subtype of Point2D.

It is interesting to note that with the gnu compiler it is possible to access Point2D::move via a pointer to a Point3D object in the main program but that with the CC compiler, this is an error. The error message refers to:- main() cannot access move: Point2D is a private base class.

In order to ascertain whether it might be possible to *externally* overload the two integer *move* for class Point3D, the Point2D *move* was re-exported from class Point3D as shown below:

```
class Point3D : private Point2D {
    int z;
  public:
    Point3D(int xx,int yy, int zz):Point2D(xx,yy){ z=zz;}
    int Z(void) { return z; }
    void move(int a,int b, int c) {
        cout << ''Point3D::move called\n'';
        Point2D:: move(a,b);
        z=z+c;
    }

    Point2D::move;      // Re-export move from class Point2D

    void show(void) {
        cout << ''Point3D('' << Point2D::X()
                << '','' << Point2D::Y()
                    << '','' << z << '')'';
    }

    void test(void) {
        this->show();
        this->move(20,20,20);
```

```
        this->show();
    }
};
```

However, attempts to move a Point3D object with a two integer argument failed to compile due to too few arguments for method 'move'. This suggests that the same mechanism works for private inheritance as for public inheritance, namely that the redeclaration of *move* to a three argument *move* hides the two integer *move* inherited from Point2D. Overloading does not occur.

As with public inheritance, overloading can be only be achieved by manually adding a two integer *move* to the class Point3D; such a *move* can call the Point2D method.

5 Discussion

5.1 Class scope in inheritance

In order to understand how inheritance is implemented in C++, it is necessary to understand how scope is affected by class derivation. Each class defines an independent scope even if it is involved in inheritance [10]. When a derived class is defined, only the members explicitly defined for the derived class are in the scope of the derived class. The members inherited by the derived class are in the scope of the base class and are *not* in the scope of the derived class, despite the fact that an object of the derived class will have inherited members as an integral part of it.

A derived class encloses the scope of its base class. Hence the search for a member of a derived class starts first in the scope of the derived class and only if the member is not found is the base class scope searched. Thus when a derived class redefines a member function, the function is not really overridden, rather it remains in the scope of the base class and so is not found because the derived class member is in the first scope searched. It is in this sense that the base class member function is said to be hidden.

Consider test 23 in section 3.1 in which a Point3D object referenced by a class Point2D pointer is moved with a Point2D argument. Since the move is effected via a class Point2D pointer, and since the move function is declared virtual in class Point2D, the move function is dynamically bound and the virtual mechanism invoked. The search for a move(Point2D) begins in class Point3D (the dynamic class of the Point2D pointer), finding no such function since move(Point2D) is not redefined in class Point3D, the search continues in class Point2D where the appropriate *move* is found. Recall that the move function defined for a Point3D argument in Point3D is a different function and is not involved in the virtual mechanism of the move function with parameter Point2D.

5.2 Overloading

When inheritance is involved, functions are resolved by the name of the function in the scope of the class. To overload a function in a derived class it is necessary to define, or redefine, both the functions in the class as in section 2.4 above. Otherwise one version will be hidden.

For example, in order to overload *move* in class Point3D it would be necessary to redefine the 2 coordinate *move* as well as redeclaring the 3 coordinate *move*. The redefinition of the 2 coordinate *move* could redirect the call to the Point2D::move.

5.3 No-variance and Covariance

The unwary may be tempted to think that the coexistence of Point2D::move(Point2D) and Point3D::move(Point3D) indicates that covariant redefinition has occurred. However, this is not so; the no-variant policy causes the Point3D::move(Point3D) to be treated as a *different* function. It may be in this sense that Harris [1] is considering that the move function is "overloaded".

In Eiffel [5], covariant redefinition is permitted. When a function is redefined in a derived class, the base class function is not available to the derived class, not even with a scope resolution

operator. If the base class method is required for the derived class, then either the new method added in the derived class must be given a different name or the inherited method must be renamed.

Consider the following code, assuming Eiffel classes have been defined to correspond to Point2D and Point3D as in section 3.1.

```
pp1, pp2 : Point2D;
pp3 : Point3D;

......

pp1 := pp3;
pp1.move(pp2);              (1)
```

Eiffel adopts reference semantics so pp1, pp2 and pp3 are pointers. The call pp1.move(pp2) compiles because statically pp1 is a Point2D and the call is correct since Point2D::move is defined for a Point2D argument. Unfortunately, at run time the dynamic class of pp1 is a Point3D and the Point3D move function has been redefined to take a Point3D parameter. There is no Point3D move function that takes a Point2D parameter and the Point2D move is inaccessible since it has been redefined and subsequently overridden. As a result there will be a run time failure.

This illustrates the way in which type safety is compromised by the covariant policy in Eiffel; for a detailed discussion of this see Cook [11]. To overcome this problem Meyer has proposed system-level validity checking [5, pages 287 and 362]. A call such as (1) above would be flagged as invalid because an instance of Point3D such as pp3 may also be attached to pp1 in this particular system.

Covariance enables subtypes to be specialised in a conceptually useful manner but can result in type checking problems. The no-variance approach of C++ is type safe but can impede the modelling of specialisation.

5.4 The subtype relationship and the virtual mechanism

In C++ a base class defines an interface specified by the public member functions it declares. If the public member functions are declared virtual in the base class, then the functions may be redefined in derived classes such that the interface of the base class is contained in the derived class. Objects of the derived class can be manipulated via the base class interface using the redefined derived class functions.

A direct result of the move function not being overloaded in Point3D is that Point3D is not a subtype of Point2D since the Point2D interface is not contained in the Point3D interface. A Point3D object does not respond to a 2 coordinate move. In addition, a direct result of the no-variance policy is that the redeclaration of *move* in Point3D has invalidated the virtual mechanism - a Point3D object manipulated via a Point2D pointer can only respond to the base class Point2D *move* and not to the derived class Point3D *move*.

The much cited paper by Cook et al [12] clearly shows that inheritance is not just subtyping. Simons and Cowling [2] advocate that "where choices exist in how to provide a package of functionality in a given set of terminal classes" then software reuse should be achieved by *composition* and inheritance should be reserved to denote strict subtyping relationships. Taking the Simons point of view, it would have been better to have implemented the Point3D class using composition rather than inheritance such that the Point3D class contained an attribute of class Point2D. Alternatively, the subtype relationship could have been enforced by one of the methods discussed below.

5.4.1 The use of type casts

The use of type casts to maintain the virtual mechanism and the subtype relationship was illustrated in section 3.3. The undermining of the type checking mechanism was discussed. We could put a tag attribute into each class so that each object can be accessed at run time to determine what class it is. If the actual parameter passed to a function which takes a Point2D formal parameter but requires a Point3D for the function to work properly, is not actually a Point3D then an error would be signalled.

The difficulty with this is that the error will occur at run time which raises problems concerning error recovery at run time.

It could be concluded that classes must be designed such that virtual functions in derived classes will not be required to take arguments which are specialisations of the base class. This does not seem very satisfactory. Other ways of preserving the subtype relationship are indicated below.

5.4.2 The redefinition of the class hierarchy

We could redesign the inheritance hierarchy such that the base class is Point containing x, y. Point2D and Point3D would inherit from Point independently. Point2D would add *move* for a 2 coordinate move and *show* to display x and y. Point3D would add z, *move* for three coordinates and *show* for all three coordinates. The subtype relationship would hold between Point and Point2D and between Point and Point3D. There would be no relationship between Point2D and Point3D.

However, this solution is not appropriate if the class Point2D already exists, in a library class for instance. In this case it would not be feasible to redefine the hierarchy.

5.4.3 The redefinition of the move operation

We could decompose the move function in class Point2D such that there were separate move functions for the x and y coordinates. Class Point3D would add another function to move the z coordinate.

Again this approach would not be feasible if class Point2D was already in use and it is not only a cumbersome solution but it is also conceptually unsatisfactory to move a point one coordinate at a time.

6 Conclusion

If a virtual function in a base class is publically inherited, redeclaration of the function does not result in the function being overloaded in the derived class. The redeclaration hides the base function. The redeclared function is treated as a separate function which is not invoked via the virtual mechanism of the base class function. However, the base class function can be accessed by a derived class object. Statically this can be achieved by using the scope resolution operator. Dynamically, it can be effected by invoking the base class function for a pointer which is statically of type base class but which actually points to a derived class object.

The no-variance policy of C++ requires that for inheritance polymorphism to be realised, a parent class function redefined in child classes must not have its signature changed. Further research is required to establish the implications of no-variance for the implementation of inheritance classifications in software.

References

[1] William Harris. Contravariance for the rest of us. *Journal of Object-Oriented Programming*, 4(7), November/December 1991.

[2] A.J.H.Simons and A.J.Cowling. A Proposal for Harmonising Types, Inheritance and Polymorphism for Object-Oriented Programming. Research Report CS-92-13, The University of Sheffield, Department of Computer Science, Regent Court, University of Sheffield, 211 Portobello St., Sheffield S1 4DP, 1992.

[3] K.M.Buchanan and R.G.Dickerson. An Investigation of Types leading to an Examination of some aspects of F-bounded Interfaces and the Type Classes of Haskell. Technical Report 154, Division of Computer Science, University of Hertfordshire, College Lane, Hatfield, Herts AL10 9AB, 1993.

[4] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys*, 17(4), December 1985.

[5] B. Meyer. *Eiffel: The Language.* Prentice Hall, Hemel Hempstead, 1992.

[6] Margaret A.Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, Massachusetts, second edition, 1990.

[7] Audrey Mayes and Mary Buchanan. A Comparison of Eiffel, C++ and Oberon-2 . Technical report 191, The University of Hertfordshire, College Lane, Hatfield, Herts AL10 9AB, 1994.

[8] Laszlo Boszormenyi. A Comparison of Modula-3 and Oberon-2. *Structured Programming*, 14, 1993.

[9] Stanley B. Lippman. *C++ Primer.* Addison-Wesley, Reading, Massachusetts, 1991.

[10] Russel Winder. *Developing C++ Software.* John Wiley and Sons Ltd., Chichester, West Sussex, 1991.

[11] W.R.Cook. A Proposal for Making Eiffel Type-safe. *The Computer Journal*, 32(4), 1989.

[12] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is Not Subtyping. *Proceedings of the 17th ACM Symposium on Principles of Programming Languages*, January 1990.