ELSEVIER

Information and Software Technology 39 (1997) 403–415

INFORMATION AND SOFTWARE TECHNOLOGY

Enhanced reusability via polymorphic additive virtual methods in C++

R. Chattamvelli, H. Saiedian

Department of Computer Science, University of Nebraska at Omaha, Omaha, NE 68182-0500, USA

Received 1 March 1996; accepted 29 November 1996

Abstract

The virtual method is a useful concept in polymorphic behavior of object-oriented programs. By making a method virtual in a class, all classes derived from that class are allowed to modify or enhance the definition of the method (while retaining its original signature) providing one kind of polymorphism. In this article, we explore the virtues of virtual methods and introduce different ways of implementing *additive* virtual methods in C++. The concepts presented can find applications in shared software libraries, integrating software applications, and distributed computing. © 1997 Elsevier Science B.V.

Keywords: Additive methods; Assistant functions; Late binding; Multiple inheritance; Virtual tables

1. Introduction

The polymorphic behavior of object-oriented programs could be achieved by different means, one of which is by *virtual methods* (or *virtual member functions* in C++ jargon). A member function can become virtual if it is explicitly specified using the keyword virtual in all classes in the inheritance hierarchy where the method must be visible.[1] Derived classes that do not redefine a virtual method use the virtual method defined in the nearest base class as the default in all virtual calls using the dominance rule ([1], pp. 205–209). The signature of a virtual method is determined uniquely using the return type, method name, and an ordered specification of the parameters. Methods with a variable number of arguments, global functions and static functions cannot be virtual in C++. All virtual methods with the same signature up in the inheritance lattice are automatically virtual methods in the derived classes if a path exists between the derived and base classes in the inheritance diagram. A virtual method can be overridden by a function in the derived class. All such overridden methods are unrelated independent functions, which simply hides the methods of the same name at a higher level in the inheritance hierarchy.

When a method or member function[2] is declared as virtual, an entry for it gets created in a virtual table `vtbl`, which is class specific (each class in which a virtual method is visible has its own copy of `vtbl`, while different instantiations of the same class simply set a pointer to the classes `vtbl`). All subsequent redefinitions of the function in derived classes set a pointer to their own table entry [1]. The derived classes are allowed to modify the implementation of the virtual functions (called *overloading*) according to the complexity of the object they have to deal with, provided they do not conflict with the unique signature. The virtual method that is to be invoked is determined dynamically at run time; this is known as *late binding*.

In a multiple inheritance hierarchy, an object can contain more than one `vptr`. This is necessary because an object in a derived class may contain all combinations of objects in the base classes. Hence, when an object `S` higher up in the inheritance hierarchy gets a pointer, it must be able to uniquely distinguish if it is an `S` or an `S` part of another object. This is most easily handled by storing a separate `vtbl` for alike classes higher in the directed acyclic graph (DAG) representing the inheritance relationship.

Sometimes the virtual methods contribute incrementally to the method defined in base classes so that the derived class could implement its methods by first invoking the base methods, instead of code duplicating. This form of virtual method refinement will be called *additive virtual methods*. For example, if a base class draws rectangles using the known boundary point and other parameters like width, length and orientation, a derived class could make use of the base method to draw shaded rectangles or other

[1] Since the class hierarchy is designed with maximum reuse in mind, most designers make even the last level methods virtual.

[2] We use the terms 'method' and 'member function' interchangeably in this article. Furthermore, for convenience, we may simply use the term 'function' as we will only be dealing with member functions in this article.

geometric figures in which rectangles (or other geometric figures defined in the base classes) are used as a component.

If a base class is inherited as private, all members of the base class are visible in the immediate derived class *only*, unless the friend mechanism is used. However, if the derivation is public or protected, the methods higher up in the inheritance hierarchy could still be additively invoked. In Section 2, we look at various ways to implement additive virtual methods in C++. An application is discussed in Section 3.

2. Virtual method invocation

The natural way to invoke the base class method is by using an explicit address resolution operator, in which the fully qualified name of the base class method is used at the point of message passing. If the base method is called `func`, all derived classes could explicitly invoke `func` as `base::func()` in their methods. However, this technique uses static binding, in principle, because the virtual method to invoke is completely known at compile time.

A more objective way of invoking the base method is by defining a pointer (or a reference) to the base class and using this pointer in all derived class methods for invoking base class methods, as shown in Scheme 1. Note that a pointer to the base class can be used in a derived class to access only those members that are inherited from the base class. However, members that are not inherited could still be accessed through casting of pointers to derived classes or using explicit pointers to derived classes. To simplify the code, we abbreviate the implementation of a method to `cout <<"class::method" << endl;`.

In Scheme 1, the class `B` uses a pointer to the base class `A` to first invoke the method higher up in the hierarchy. Also, the class `C` has a pointer to the `B` object (which may also be initialized using a pointer to the `A` object) where standard conversion takes place. Because a constructor for the `B`

C++ Listing 1

```
class A {
public:
    // constructors and other members
    virtual void f() { cout << "A::f" << endl; }
};

class B: public A {
public:
    // class B's constructor gets a ptr to base class as an argument
    B (A *ptr) : a(ptr) {}; // initialize local pointer a

    ~B () {delete a;};       // destructor
    A *a;                    // pointer to base class

    virtual void f() {
        a -> f();
        cout << "B::f" << endl; // incrementally implemented
    }
};

class C: public B {
public:
    C (A *ptr) : b(ptr) {}; // standard conversion
    ~C() {delete b;}

    B *b;
    virtual void f() {
        b -> f();
        cout << "C::f" << endl;
    }
};
```

Scheme 1.

object is automatically called when a C object is instantiated, a default B constructor must be provided as shown in Scheme 1. This rule applies to all classes in the inheritance DAG. When a pointer to a base class is used to invoke a virtual function, the specific function called depends on the type of the object rather than the type of the pointer ([2]). Scheme 2 shows the sequence of steps involved in the invocation.

This technique has the disadvantage that the constructor is used to initialize the pointer to the virtual methods. The applicability of this technique is hence limited, since the constructor is called on a per object basis at the time of object creation. A solution is to define another member function to dynamically initialize the pointers, as shown in Scheme 3.

Because the pointers are now initialized using ordinary member functions, all the pointers used in calling virtual methods in their respective base classes must be properly initialized before the virtual methods may be additively invoked as shown in Scheme 3. For instance, if a C object needs to invoke one of the virtual methods in class B as well as A, the Init function for both B and C must be called properly so that when the C object sends a message to the B object, it knows exactly where the next message needs to be sent to additively invoke the method in one of its base classes. Since each class with virtual functions uses additional pointers which require memory indirection, this technique will be slower than standard function invocation. If the specific virtual method(s) to be invoked from the base classes are predetermined, one could use a pointer to the base class method itself, instead of a more general pointer to the base class, as shown in Scheme 4.

However, there is a problem with the code in Scheme 4. The pointer which is passed as an argument to the constructor of B and C is a general function pointer which does not bind itself to a specific object. As discussed in Ref. [1] (p. 157), there are two simple ways to invoke a method using a pointer – either using the (object.*ptr)() syntax which is consistent with C syntax, or using a pointer to the object as (cptr - > *mptr)() where cptr is a pointer to the object and mptr is a pointer to the method to be invoked. If the second approach is followed, the code in Scheme 4 must be modified as shown in Scheme 5. The application of proper type casting allows selective invocation of methods additively as in the previous case (see Scheme 6).

Another way of invoking base class methods is using a global non-member function. We could also use specialized member functions to return the address of virtual member functions, as shown in Scheme 7.

Note that we have used typedef within the class Y to simplify the code. If the typedef is defined in the base class as xptr, all derived classes should type-cast the return function names as (xptr) funcname. The *address* of a virtual function is defined relative to the class in which it belongs. Owing to a lack of proper terminology, we call the method that returns the address of (other) member functions an *assistant function*. To invoke a virtual method from a class derived from the class Y, we simply get the address of the method by invoking the assistant and then supply the proper parameters according to the signature of the virtual method to be invoked. If the class contains several virtual methods, we can pass the name of the virtual method to be invoked to the assistant which could search through the *visible* methods and return the correct address. For instance,

C++ Listing 1a

```
A animal;
A *aptr = &animal;

B bloodhound(aptr);
B *bptr = &bloodhound;
bptr -> f();  // or equivalently bloodhound.f();

C cat(bptr);
C *cptr = &cat;

// The following invokes C::f(), which in turn calls B::f(), A::f()
cptr -> f();  // or  cat.f();

C bigcat( (B*) aptr); // pointer to animal is type casted
cptr = &bigcat;

// The following virtual call invokes C::f() and A::f() only
cptr -> f();
```

Scheme 2.

C++ Listing 1b

```
class A {
public:
    // Constructors and other members
    virtual void f() { cout << "A::f "; }
};

class B: public A {
public:
    B(){};
    A *a;
    ~B(){delete a;};

    void Init (A *aptr) { a = aptr;} // set a and other data members
    virtual void f() { a -> f(); cout << "B::f"; }
};

class C: public B {
public:
    C(){};
    ~C(){delete b;}

    B *b;
    virtual void f() { b -> f (); cout << "C::f "; }
    void Init (B *bptr) { b = bptr; }
};

void main(){
    A animal;
    A *aptr = &animal;

    B bloodhound;
    B *bptr = &bloodhound;

    C cat;
    C *cptr = &cat;
    cptr -> Init( (B *)aptr );
    cptr -> f();            // calls C::f() which in turn calls A::f()

    bptr -> Init(aptr);
    cptr -> Init(bptr);
    cptr -> f();            // calls C::f() which in turn calls B::f(), A::f()
}
```

Scheme 3.

pmf in Scheme 8 is declared to be a pointer to the member function of class Y and assigns the address of the virtual method 'f' to it using a call to the assistant. It can be used to invoke the virtual method for an instantiated object (say y), as shown in Scheme 8.

This technique is efficient only for small depths of inheritance with limited virtual function invocations from higher hierarchies. Note that the assistant function need return the addresses of only those methods which are to be reinvoked.

Although constructors cannot be virtual in C++, destructors can. When the classes in an inheritance hierarchy contain dynamically allocated objects, it is the usual practice to declare the destructors as virtual. Owing to an abnormal program condition or to a failed assertion, the destructor

C++ Listing 2

```
class A {
public:
    A(){ cout << "Constructor of A " <<  endl; }
    ~A(){};

    virtual void f() { cout <<  "A::f" << endl; }
};

class B: public A {
public:

    // A pointer to base member fn is passed as argument to B's constructor
    B (const void (A::*ptr)()) : a(ptr) {
        cout <<  "B's constructor" << endl;
    }
    ~B(){};

    const void (A::*a)(); // define a ptr to A's methods
    virtual void f() {
        (*a)();
        cout << "B::f" << endl;
    }
};

class C: public B {
public:
    C (const void (A::*ptr)()) :  b(ptr) {
        cout <<  "C's constructor" <<  endl;
    }

    const void (B::*b)(); // define a ptr to B's method

    virtual void f() {
        (*b)();
        cout <<  "C::f" << endl;
    }
};
```

Scheme 4.

of a derived class may have to call some or all of the destructors of its base classes. This could be achieved by many means, one of which is explored in Scheme 9.

In some applications, need may arise to invoke several virtual functions of the base class(es) collectively. This is most easily implemented by storing the addresses of the virtual methods in an array of pointers (see Scheme 9) and storing the names of these functions in an associated tag-array. Because the constructors of all base classes are automatically invoked in reverse order when a derived class object is instantiated, one could define a public method in the top level base class to be automatically invoked by the constructors to set up the pointer arrays.

3. Practical application

In this section we demonstrate the effectiveness of the proposed method over the ordinary subclassing with virtual method inheritance. We consider the well-known problem of a planar drawing of a n-dimensional hypercube for small values of n up to 4. We assume that, in addition to the intrinsic geometric aspect of the planar drawing, each node of the hypercube also has some data associated with it because a planar drawing could be accomplished even without the class concept.

A two-dimensional hypercube is a square and we assume that it is drawn with one pair of sides parallel to the X-axis

C++ Listing 3

```
class A {
public:
    ~A(){};
    A() { cout << "Constructor of A " << endl; }
    virtual void f() { cout << "A::f" << endl; }
};

class B: public A {
public:
    // A pointer to base member fn is passed as argument to B's constructor
    B (void (A::*ptr)(), A *cptr) : a(ptr), aa(cptr) {
        cout << "Constructor for B" << endl;
    };

    void (A::*a)(); // define a ptr to A's methods
    A *aa;          // define a ptr to the class itself

    ~B (){};
    virtual void f() {
        (aa -> *a)();
        cout << "B::f" << endl;
    }
};

class C: public B {
public:
    C (const void (A::*ptr)(), A *cptr) : b(ptr), bb((B *)cptr) {
        cout << "Constructor for C" << endl;
    };

    const void (B::*b)(); // define a ptr to A's methods
    B *bb;                // define ptr to the class itself

    virtual void f() {
        (bb -> *b)();
        cout << "C::f" << endl;
    }
};
```

Scheme 5.

and the other pair parallel to the *Y*-axis. A three-dimensional hypercube is drawn in two dimensions with pairs of opposite edges parallel to the *X*-axis and the other pairs parallel to the *Y*-axis. The two-dimensional hypercube is drawn by additively invoking the one-dimensional hypercube program with the direction reversed orthogonally as shown in the code (listed in). For drawing the three-dimensional hypercube, we invoke the two-dimensional program, which in turn invokes the one-dimensional program additively. The three-dimensional program then draws the slanted sides. We use the arcs of an ellipse to represent the node connections in a four-dimensional hypercube (called a *tesseract*), which is drawn using two three-dimensional hypercubes.

In a situation like the above, if the method does not utilize data other than its own parameters, we could very well implement it as ordinary C functions. If the method does not access the private data members, another possibility is to implement it using ordinary subclassing with virtual method inheritance. However, maximum code reuse is achieved by structuring the class hierarchies and invoking the base class methods additively. Notice that we have not used the pointers to base classes or pointers to base class methods

C++ Listing 3a

```
A animal;
A *aptr = &animal;
void (A::*af)()= animal.f;

B bloodhound (af, aptr);
B *bptr = &bloodhound;
bptr -> f();

C cat(af, bptr);
cat.f();          // Calls C::f() which in turn calls B::f(), A::f()

void (B::*bf)() = bloodhound.f;
C bigcat (bf, (B *)aptr );
C *cc = &bigcat;
cc -> f();        // Calls C::f() and A::f() only
```

Scheme 6.

in the example because each of the methods in the DAG of the class hierarchy is invoked in succession.

In essence, additive invocation of methods defined in the higher level classes in the inheritance DAG simplifies software development and significantly improves reuse of previously integrated class hierarchies. For example, if a software system needs to be developed at different expertise levels (like a junior version, a professional version and an advanced user version) we could hierarchically build up the class library by moving down more sophisticated functionalities towards the lower level classes, which then additively invoke the less sophisticated methods defined at higher level

C++ Listing 4

```
class X {
public:
    // constructors and other members
    virtual void f() { cout <<  "Base::f" << endl; }
    friend void fg(char *);
};

class Y: public X {
public:
    // constructors and other members
    typedef void (Y::*vptr)(); // pointer to member function
    virtual void f() { cout << "Derived f" << endl; }
    virtual void g() { cout << "Derived g" << endl; }

    vptr fg (char *method_name) {
        switch (*method_name) {
            case 'f': return f; // return address of method f
            case 'g': return g; // return address of method g
            default : cout << "Illegal function name " << *method_name << endl;
                      return f; // return address of default method
        }
    }
};
```

Scheme 7.

C++ Listing 5

```
Y y;
void (Y::*pmf)();
pmf = y.fg("f"); // assistant returns the address of specified method
(y.*pmf)();      // call the virtual method
```

Scheme 8.

C++ Listing 6

```
class X {
public:
    typedef void (X::*xptr)();

    xptr fptrs [10]; // pointer array to store address of virtual functions
    int iptr;        // index to the array

    X():iptr(0) { fptrs [iptr++] = f; }

    virtual void f() { cout << "Base::f" << endl; }
};

class Y: public X {
public:
    // typedef void (Y::*yptr)();
    Y() { fptrs [iptr++] = (xptr)f; }

    virtual void f() { cout << "Derived f" << endl; }
    virtual void g() { cout << "Derived g" << endl; }
};
```

Scheme 9.

classes. The ordinary subclassing with virtual method inheritance is clearly inadequate and produces cumbersome code, compared to the concise code achieved with the aid of additive virtual methods, clearly demonstrating the usefulness of our approach. The proposed methods are also useful in designing many software applications where hierarchically specialized shared class libraries will be heavily used.

Although the suggested methods in Section 2 at first seem rather complicated, the stated objectives are best achieved using our method. The interested reader is referred to Refs. [3,4] for related issues.

4. Conclusion

Object-oriented software development techniques have already been found to be immensely useful in several disciplines and continue to find many more applications. A class hierarchy with hundreds of virtual methods is not uncommon in practice. Designing the hierarchy with maximum reusability and minimum code duplication yields fruitful results in the long run. Our discussion in this article has focused on simple ways of invoking virtual methods additively; however, these are by no means the only ways of implementing the problems discussed. In recent years, there has been an increasing trend towards applying object orientation to distributed computing, client server computing and the like, in which improved productivity, better reuse of existing code and easy maintainability of large software systems are of prime concern. It is hoped that the discussion in this article will find applications in shared software libraries, in integrating software applications and in other emerging technologies as well.

```
// A program to draw the hypercube up to 4 dimensions using
// additive virtual methods. Higher dimensional hypercubes
// can be drawn recursively using lower dimensional methods.
//
// This program was implemented in Microsoft C++ version 7.0.

#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <graph.h>
#include <conio.h>
#include <windows.h>

#define regular_side 5
#define slanted_side 8
enum dir {left, right, up, down};
float theta = 3.142/4;

class dim_1 {
public:
  POINT u;
  dim_1(int i, int j) {u.x = i; u.y = j; }
  ~dim_1() {}
  void Setxy(int p, int q) { u.x = p; u.y = q; }
  void draw(int direction) {
     switch(direction) {
       case left:
            _lineto(u.x - regular_side, u.y);
            u.x -=  regular_side;
            break;
       case right:
            _lineto(u.x + regular_side, u.y);
            u.x +=  regular_side;
            break;
       case up:
            _lineto(u.x, u.y + regular_side);
            u.y += regular_side;
            break;
       case down:
```

```
                _lineto(u.x, u.y - regular_side);
                u.y -= regular_side;
            }
        }
};

class dim_2: public dim_1 {
public:
    dim_2(int p, int q):dim_1(p, q) {
    }
    virtual void draw() {
        dim_1::draw(right);
        //  change orientation and draw again
        dim_1::draw(up);
        dim_1::draw(left);
        dim_1::draw(down);
    } // draw
}; // dim_2

class dim_3 : public dim_2 {
public:
    dim_3(int p, int q):dim_2(p, q) {
    };
    void draw_slant() {
        int p = u.x, q = u.y, r, s;
        r = p + slanted_side * acos(theta);
        s = q + slanted_side * asin(theta);
        _lineto(r, s);
        p += regular_side;
        Setxy(p, q);
        r = p + slanted_side * acos(theta);
        _lineto(r, s);
        q += regular_side;
        Setxy(p, q);
        s = q + slanted_side * asin(theta);
        _lineto(r, s);
        p -= regular_side;
        Setxy(p, q);
        r = p + slanted_side * acos(theta);
        _lineto(r, s);
        q -= regular_side;
        Setxy(p, q);
```

```
            s = q + slanted_side * asin(theta);
            _lineto(r, s);
        } // draw_slant

        virtual void draw() {
            dim_2::draw();
            draw_slant();
            int p = dim_1::u.x + slanted_side * acos(theta);
            int q = dim_1::u.y + slanted_side * asin(theta);
            dim_1::Setxy(p, q);
            //  change orientation and draw again
            dim_2::draw();
            // Now draw the slanted lines
        } // draw
    }; // dim_3

    class dim_4 : public dim_3 {
    public:
        dim_4(int p, int q):dim_3(p, q) {};
        void draw_arcs(int p, int q, int r, int s) {
            int p1 = p, q1 = q, r1 = r, s1 = s;
            _arc(r, s, p, q-2, r, s, p, q);
            r += regular_side;
            p += regular_side;
            _arc(r, s, p, q-2, r, s, p, q);
            r += slanted_side * acos(theta);
            s += slanted_side * asin(theta);
            p += slanted_side * acos(theta);
            q += slanted_side * asin(theta);
            _arc(r, s, p, q-2, r, s, p, q);
            r -= regular_side;
            p -= regular_side;
            _arc(r, s, p, q-2, r, s, p, q);
            q1 += regular_side;
            s1 += regular_side;
            _arc (r1, s1, p1, q1-2, r1, s1, p1, q1);
            r1 += regular_side;
            p1 += regular_side;
            _arc(r1, s1, p1, q1-2, r1, s1, p1, q1);
            r1 += slanted_side * acos(theta);
            s1 += slanted_side * asin(theta);
            p1 += slanted_side * acos(theta);
            q1 += slanted_side * asin(theta);
            _arc(r1, s1, p1, q1-2, r1, s1, p1, q1);
            r1 -= regular_side;
            p1 -= regular_side;
            _arc(r1, s1, p1, q1-2, r1, s1, p1, q1);
        } // draw_arcs

        virtual void draw() {
            dim_3::draw();
            int p = dim_1::u.x+2 * regular_side;
            int q = dim_1::u.y;
            int r = dim_1::u.x;
            int s = dim_1::u.y;
            int p1, q1, r1, s1;
            p1 = p;
            q1 = q;
            r1 = r;
            s1 = s;
            dim_1::Setxy(p, q);
            dim_3::draw();
            dim_1::Setxy(r, s);
            draw_arcs(r1, s1, p1, q1);
        } // draw
    }; // dim_4

    void main (void) {
        dim_4 tesseract(12,30);
        tesseract.draw();
    }
```

References

[1] Ellis, M., and Stroustrup, B., (1990), The Annotated C++ Reference Manual (Addison Wesley, Reading).

[2] Koenig, A. (1989), How virtual functions work, Journal of Object-Oriented Programming, January/February, pp. 73–74.

[3] Eckel, B. (1994), Polymorphism and virtual function in C++, Embedded Systems Programming, 7(10), 42–47.

[4] Stroustrup, B. (1986), The C++ Programming Language (Addison Wesley, Reading).