Overriding Return Types of Virtual Functions

29 May 1992
X3J16/92-0064
W621/N0141

John Bruns
Chicago Research and Trading
E-mail: jwb@cup.portal.com

Dmitry Lenkov
Hewlett-Packard Company
E-mail: dmitry@cup.hp.com

The experience of C++ developers shows that in many applications it is advantageous to have different return types for a virtual function in the base and derived classes. The widely accepted view of subtyping in object-oriented languages permits the return type of such virtual function to be restricted in the derived class if the restricted return type can be legally converted to the original one [6], [7]. Restriction of the return type of a virtual member function in a derived class was illegal in the original definition of C++. Consider writing a clone function. It is natural to define this function as returning a pointer to a new copy of the original object.

Example 1:

```
class A {
public:
   ......
   virtual A * clone();
}
class B : public A {
public:
   ......
   B * clone();
}
```

According to the original rule [1, section 10.2] the declaration of function *clone()* in class **B** is an error. This rule says:

> *It is an error for a derived class function to differ from a base class' virtual function in the return type only.*

The extension adopted by the C++ committee at its March of 1992 meeting, relaxes this rule if involved return types are pointers or references to classes where a class used in the base class is, in turn, a base class of the class used in the derived class. Required conversions of such pointers and references are legal and type safe. This extension was originally proposed by Alan Snyder [9], and later analyzed in [10] and [11]. The new rule states:

> *It is an error for a derived class function to differ from a base class' virtual function in the return type only unless the return types are either both pointers to classes or both references to classes, and the class in the base class function's return type is an accessible base class of the class in the derived class function's return type.*

The new rule makes Example 1 legal.

During discussions about this extension several work-arounds were suggested. All of them require, in one way or another, a type cast to recover the type information. In the case of return types being pointers (or references) to classes in an inheritance hierarchy, the type cast usually becomes a cast (downcast) of a pointer to a base class to a pointer to its descendent. Downcasts are type unsafe, and therefore dangerous in general. They force the programmer to follow very strict discipline in creating and extending his or her code. In addition, multiple inheritance and virtual base classes make the code resulting from these work-arounds difficult to follow and extend.

1. Single Inheritance

Having a return type differ as you move down a class hierarchy is natural in an object oriented system. It allows static type information to be preserved and safely used in the appropriate context. The simplest case is where the object returns a pointer or reference to an instance of its own class. For example, it is often useful to have operations defined for list classes to return objects of these classes.

Example 2:

```
class List {
public:
   ......
   virtual List* tail();
};
class SortedList: public List {
public:
   ......
   SortedList* tail();
   ListElem* find( Key);
};
```

The virtual member function *tail()* returns the list tail which is a list of the same kind. The actual type (statically defined) of the returned list is defined in the context of types of pointers, references, or actual objects for which this function is invoked. Consider the following set of uses of *tail()*.

Example 3:

```
List l;
SortedList s;
List& l_r = s;
   ......
List* l_p1 = l.tail();                    // 1
List* l_p2 = l_r.tail();                  // 2
SortedList* s_p = s.tail();               // 3
ListElem* e_p = (s_p->tail())->find( k);  // 4
```

In statement 1, *tail()* of class **List** is called because it is invoked on object l of this class. In statement 2, **l_r** is a reference to an object of class **SortedList**. Consequently, *tail()* of class **SortedList** is called. However, it returns a pointer of type **List*** because it is invoked through reference **l_r** of type **List&**, i.e. it is invoked in the context of class **List**. In statement 3, *tail()* of class **SortedList** is invoked in the context of class **SortedList**. Therefore, it returns a pointer of type **SortedList***. Similarly, in statement 4 *tail()* returns a pointer of type **SortedList***. Static type information is preserved, and the call to function *find()* does not require any explicit casts.

The next example demonstrates the case where the object returns a pointer or reference to an instance of a class in a parallel hierarchy. Consider the following classes:

```
class Window {
public:
   ......
   virtual void display();
 };
```

```
class ScrollingWindow : public Window {
public:
   ......
   void display();
   virtual void scrollUp();
   virtual void scrollDown();
};
```

It is evident that the derived class implements behavior (scrolling) that is not available to the base class. If both classes are under control of the same developer, we could move the new behavior into the base class. Often, however, this base class comes from a library under external control, in many cases without access to source code. So the new functionality must be added later. Now consider the companion classes:

Example 4:

```
class WindowController {
public:
   WindowController( Window*);
   ......
   virtual Window* window();
};
class ScrollingWindowController : public WindowController {
public:
   ScrollingWindowController( ScrollingWindow*);
   ......
   ScrollingWindow* window();                    //  NOTE redefined return type
   virtual void upButton() { window()->scrollUp();} ;
   virtual void downButton() { window()->scrollDown();};
};
```

Prior to the ability to redefine return types in virtuals, each of these functions would require a cast to implement necessary behavior since *window()* could only return a **Window***. For example:

```
virtual void upButton() {
   ((Scrolling Window *) window())->scrollUp(); };
```

In this case, a primary benefit is cleaning up the syntax. If the only use of the virtual functions involved were internal to the class, the class designer could cope with the syntax. However, such constructs are often used by application frameworks that assume a more naive programmer as the ultimate user. It is dangerous when programmers get accustomed to indiscriminate casting. Allowing the class to incorporate all the type checking and export that information through its interface provides safer programming.

In the trivial case of single inheritance, the extension requires no additional code generation since all pointers point to the same address within the object, regardless of their actual type. The compiler, however, needs to check return types and make sure that virtual functions in derived classes return objects of appropriate types.

2. Multiple Inheritance

Multiple inheritance presents the case where the extension provides an opportunity to simplify user code and make it more efficient. The extension makes the use and definition of virtual functions with overridden return types for classes with multiple inheritance as straight forward as for classes with single inheritance.

Example 5:

```
class A {
public:
   ......
   virtual A * clone();
}
class B : public A {
public:
   ......
   B * clone();
}


class C {
public:
   ......
   virtual C * clone();
}
class D: public B,  public C {
public:
   ......
   D * clone();
}
```

An attempt to write this code without the change allowing redefinition of return types leads to messy and error-prone code (see [4]). Note that the behavior of *clone()* is virtual and therefore the correct function will always be called. The dynamic type returned will always be the same as the dynamic type of the object calling it. The static type will also be exactly the same as the object calling it.

Example 6:

```
A a;
D d;
A* p_a = a.clone();     // make a new A object, type returned is A*.
D* p_d = d.clone();     // make a new D object, type returned is D*.
C* p_c1 = &d;
C* p_c2 = p_c1->clone();   // make a new D object, type returned is C*.
```

In the last case the function definition the compiler looks up is *C::clone()* (**p_c1** is of **C***) whose return type is **C***. Since *clone()* is a virtual function, the function actually executed depends on the run-time type of the object pointed to by **p_c1**, in this case **D**. The return type of *D::clone()* is **D***, but the compiler must insure **p_c1->clone()** is properly statically typed and provide the mechanism to convert the **D*** type returned by *D::clone()* into **C***.

This complication is due to the implementation details involved in multiple inheritance. Without MI, the value of all objects pointers referring to the same object are the same and no conversion must be done. The system must still keep track of static types for program analysis.

3. Multiple Inheritance with virtual base classes

Multiple inheritance with virtual base classes, presents the case where the extension provides even greater opportunities in simplifying user code and making it more efficient. The extension makes the use and definition of virtual functions with overridden return types in the case of virtual base classes as straight forward as for classes with single inheritance. Virtual base classes also add no more complexity than usual. Consider the following hierarchy (diamond) of classes.

Example 7:

```
class List {... virtual List* tail(); ...};
class SortedList: public virtual List
          {... SortedList* tail(); ListElem* find( Key); ...};
class LenList: public virtual List
          {... LenList* tail(); int length(); ...};
class LenSortedList: public SortedList, public LenList
          {... LenSortedList* tail(); };


   ......
List l;
LenSortedList ls;
List& l_r = ls;
   ......
List* l_p1 = l.tail();                          // 5
List* l_p2 = l_r.tail();                        // 6
LenSortedList* ls_p = ls.tail();                // 7
ListElem* e_p = (ls_p->tail())->find( k);       // 8
```

Statements 5-8 have the same semantics as statements 1-4 in Example 3.

It is an interesting fact that implementation of the extension for multiple inheritance, with or without virtual base classes, is not significantly more difficult than for single inheritance. Of course, it inherits the complexity of multiple inheritance implementation. But, conceptually, the only difference is that in each context distinguished by a particular implementation for a class hierarchy with multiple inheritance the pointer returned by a virtual function with overridden return type needs to be converted according to this context. There are several ways how this conversion can be applied to the return value. The choice how to do it is the choice between efficiency for space and efficiency for execution speed.

4. Work-arounds

As mentioned before, several work-arounds [4] which would eliminate the need for the new extension were discussed in the committee. All of them have the disadvantage of weakening the type checking. Let us consider two work-arounds. For this purpose we assume the old rule. We also assume that function *tail()* returns a pointer of type **List*** in both classes defined in Example 2.

4.1. Explicit use of casts

In this case the piece of code presented in Example 2 needs to be rewritten in the following way.

Example 8:

```
   ......
List* l_p1 = l.tail();                          // 1 - fine
List* l_p2 = l_r.tail();                        // 2 - fine
SortedList* s_p = (SortedList*)s.tail();              // 3 - explicit cast
ListElem* e_p = ((SortedList*)(s_p->tail()))->find( k); // 4 - explicit cast
```

This puts the burden of doing the cast and insuring that it is valid on the user. If circumstances change, it leaves a number of potentially unsafe casts in the code. In addition, this work-around does not work for virtual base classes. There is no way how we could apply this work-around to statements 7-8 in Example 7.

4.2. Hidden use of virtual functions

This work-around requires to change the definition of classes used in the previous example.

Example 9:

```
class List {
public:
   ......
   virtual List* hidden_tail();
   List* tail() { return hidden_tail(); }
};
class SortedList: public List {
public:
   ......
   virtual List* hidden_tail();
   SortedList* tail() { return (SortedList*)hidden_tail(); }
   ListElem* find( Key);
};
```

With this definition of classes, the piece of code in the Example 3 is valid and achieves the same result. However, the burden of doing the cast and insuring that it is valid is still on the user. The work-around is not intuitive. It is easy to make a mistake and turn the code around. The system is still not type safe. It is perfectly type safe at the user level but relies on the uncheckable guarantee that all future redefinitions will use correct casts.

While this work-around looks reasonably simple for the single inheritance case, it becomes complex and messy for multiple inheritance [4].

5. Why we cannot allow polymorphic overriding of non-pointer return types

When we relaxed the return type of virtuals, we were very selective about it. In particular we relaxed it only for those return types that were pointers or references to classes. Why didn't we go further? What, we were trying to do is to remove a restriction that was probably too strict to start with. Pointers and references to classes are the only language entities in C++ which exhibit polymorphic behavior. Assuming an object is an instance of some descendant class, it can always be referenced by a pointer (reference) to any of its accessible base classes. We are not changing the object in any way, merely allowing it to be statically typed to an appropriate class.

Now consider what would happen if we allowed a function that returned an object by value to be redefined to return a derived object by value. In the context of the base object, the returned object would be truncated to the smaller size. We already allow this behavior in assignment and it is almost never correct. To open this trap in the language would be wrong.

What about user defined conversions? Once again, we were closing a hole in the type system that was too restrictive. We were not trying to put new powerful constructs into the language. Implementing different return types for the same object is almost always, at most, an adjustment of the pointer. User defined conversions are a problem of a different magnitude. The cost of implementing them far outweighs any utility gained. It adds major language feature, hard to implement, hard to understand.

6. Why we cannot allow polymorphic overriding of virtual function parameters

6.1. What is often wanted is covarient paramenters

Example 10:

```
class Container {
   ......
   virtual Container* merge( Container* anotherContainer);
};
class Set: public Container {
   ......
   Set* merge( Set* anotherSet);
};
```

This is what is wanted but it is not safe. Consider the following:

```
Set s;
Container c;
Container* p_c = &s;
p_c->merge(c);    // OOPS Set::merge is expecting another Set!!!
```

The C++ virtual function mechanism provides no way to insure that the derived class functions are called with the proper arguments.

6.2. What is safe is contravarient parameters

Example 11:

```
class Container {
   ......
   virtual Container* merge( Set* anotherSet);
};
class Set: public Container {
   ......
   Set * merge( Container* anotherContainer);
};
```

This will never cause a type error. Unfortunately this is not generally useful. Furthermore it would add confusion to programmers in conjunction with the already complex overloading rules.

6.3. Overriding function parameters collides with overloading

Currently a function is defined by its signature. This includes its name and the types of all parameters. It excludes the return type. This allows us to override the return type of a virtual function without impacting the signature in any way. If we allow changes to the parameter types, we must add some syntax to the language to tell the compiler that this is actually a redefinition of a previously defined virtual function and not a completely new virtual (or non-virtual) function. Adding such syntax is a much more significant change than removing the restriction on return types.

6.4. Adding run-time type identification can help

Most of the time when we need to write code such as in Example 10, we are sure that it is an error if the code is called in the wrong context. What we need is some way to insure that the covarient variable is the proper type. Assuming a dynamic cast facility [5] with syntax (**?T***) that returns 0 on failure:

Example 12:

```
class Container {
   ......
   virtual Container* merge( Container* anotherContainer);
};
class Set: public Container {
   ......
   Set* merge( Container* anotherContainer);
};

Set* Set::merge( Container* anotherContainer) {
   if( Set* anotherSet = (?Set*) anotherContainer) {
      // OK the argument was a Set
   }
   else {
      // throw exception or do something smart here;
   }
}
```

7. We also do not allow changes in variable types - only function returns

Consider our window and controller classes again:

Example 13:

```
class Window {....};
class ScrollingWindow : public Window {....};

class WindowController {
   ......
   Window* window;
};
class ScrollingWindowController : public WindowController {
   ......
   ScrollingWindow* window;
};
```

If the above code compiles, functions in the base class will be referring to the wrong window. It is unlikely that this will work as the programmer intended. In addition, in the case of multiple inheritance it leads to wrong pointer manipulations. A good work-around instead is to make the variable private in a base class and refer the variable only through access functions. If the access function is a virtual function with the type redefined, the result is what was intended above. A cast might be required but it can be guaranteed safe either through some run-time mechanism or more likely by controlling the access to the private variable. Note such variables are often set only in object constructors. This is always the case with reference variables.

Example 14:

```
class WindowController {
   ......
   Window* myWindow;
public:
   virtual Window* window() { return myWindow; };
   ......
};
class ScrollingWindowController : public WindowController {
   ......
public:
   ScrollingWindow * window() {return (ScrollingWindow *) myWindow;} ;
   ......
};
```

8. Pointers to members and new feature

Example 15:

```
class X { };
class Y : public X {...int i;...};

class A {
   ......
   virtual X *f();
   X *g();
};
class B : public A {
   ......
   Y *f();
};

.........
X *(A::*pma)();
Y *(B::*pmb)();

X x;
X *A::g() { return &x; }
pma = &A::g;
pmb = pma;          // error
B b;
(b.*pmb)()->i = 1;   // result of the error above

pmb = &B::f;        // legal
....
pma = &A::f;        // legal
A* p_a = new B;
X* p_x = (p_a ->* pma)(); // O'K
```

We could add another extension:

Example 16:

```
class T;

class X {...};
class Y: X {...};

class A {
   ........
   virtual T *(Y::*f())();
};
class B : public A {
   ........
   T *(X::*f())();
};
```

It is safe but useless.

9. Conclusion

The proposal extends C++ subtyping rules for classes with virtual member functions. It is consistent with widely accepted subtyping rules for object-oriented programming languages. At the same time, it makes code using this extension easy to write, document, and understand. It adds no new syntax to the language. It breaks no existing code.

10. References

[1] M. A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.

[2] Alan Snyder, *A Proposal to Revise C++ Subtyping Rules*, X3J16/90-0049, 1990

[3] Martin J. O'Riordan, *Polymorphic Over-Riding of Function Return Types (An Examination)*, X3J16/91-0051, 1991

[4] John Bruns and Dmitry Lenkov, *Extending C++ to Allow Restricted Return Types on Virtual Functions*, WG21/N0082=X3J16/92-0004, 1992

[5] B. Stroustrup, D. Lenkov, *Run-Time Type Identification for C++*, The C++ Report, Vol.4, #2

[6] L. Cardelli, *A semantics of multiple inheritance*, In *Semantics of Data Typer*, LNCS 173, Springer-Verlag, 1984

[7] W. R. Cook, *A Proposal for Making Eifell Type-safe*, Proceedings of ECOOP'89, Cambridge University Press, 1989