X3J16 / 92–0004
WG21 / N0082

Extending C++ to allow restricted return types on virtual functions

John Bruns and Dmitry Lenkov

jwb@cup.portal.com
dmitry@hpcldnl.hp.com

Jan 31, 1992

MOTIVATION & DESCRIPTION

It is a type safe operation for a virtual function to generate a more specific result than that of its base class. Since C++ makes it an error for virtual functions to differ in return type, the result must be cast to the base return type regardless of its calling context. The only way to recover this knowledge in C++ is with a type cast. If the base return type is a virtual base, even a cast will not recover the type information. This causes the programmer to have to take some heroic measures to deal with a relatively simple concept. The resulting code is difficult to follow and extend, has inherent safety problems and is not efficient. By moving this simple concept into the compiler, these problems are resolved, the compiler can catch any typing errors at compile time.

This extension was initially sent to the committee in a paper by Alan Snyder [1]. The specific request discussed in this paper is to relax the restriction that a virtual function must return the same type in all derived classes as it does in its base class. The proposal was examined in a paper by Martin O'Riordan [2] who determined that there were no technical reasons that the proposal could not be implemented. The Extensions WG requested that we provide a more detailed analysis.

Currently the draft reads:

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

We propose that the change read something like:

> "If a derived class function differs from a base class function by only the return type, it is an error unless both return types are either pointers or references to classes and there is an implicit, unambiguous and accessible cast from the new class to the original class.

The change requires NO changes to the grammar. A return type of an expression would be determined using the same look up rules that non–virtual functions use now. The compiler would be required to check that each new return type for the virtual function is allowable based on all previous return types of the base class(es). The simplest example is:

```
struct A {
      virtual A * clone() {  new A(*this); };
...};

struct B : public A {
      virtual B * clone() { return new B(*this); };
            // NOTE CURRENTLY clone() could only be a non-virtual
            // function that HIDES A::clone()
            // (the keyword virtual would be illegal here)
            // UNDER THE PROPOSAL clone() would be the virtual
            // function that overrides A::clone()
... };

A * pa = &a;
A * pa2 = &b;
B * pb = &b;
pa->clone();    // case 1. executes A::clone(), evaluates to A *;
pa2->clone();   // case 2. executes B::clone(), evaluates to A *;
pb->clone();    // case 3. executes B::clone(), evaluates to B *;
pa = pb->clone();  // case 4 same as cases 3.  executes B::clone
            // which evaluates to B * which is then cast to an A*
            // for the assignment to pa;
pb->A::clone(); // case 5. executes A::clone(), evaluates to A *;
```

Under this proposal, the return type of an expression using any function would be determined exactly the same way that a non–virtual function is determined now. In the example above, if we omitted the keyword virtual in B::clone(), it would compile today with the following changed results:

```
pa2->clone();   // case 2. executes A::clone(), evaluates to A *;
                // there is no virtual behavior.
```

Snyder [1] suggested that the context in case 4 be A::clone since this function returns an A* and the assignment operator is looking for an A* type. This would be equivalent of overloading on return type of functions and is not supported in the language. However, proposal already guarantees that the new return types have an implicit, unambiguous cast to the base class type. There is no reason why a smart compiler could not optimize these two casts away by using the A::clone virtual address.

In case 5, the context of the clone functions was deliberately qualified to A:: even though the pointer was of type B. The resulting behavior is due to the fact that explicit qualification suppresses the virtual mechanism. (Note this is one area in which the work around differs from the expected implementation). What the proposal requires is that the compiler develop a mechanism to handle the casting of the return type into the type required for the expression it appears in. With the current language restriction, this is not necessary since their return type is always the same..

NOTE: The simplest, most intuitive function that would benefit from this extension is the clone operation. When we clone an object, we want an exact copy of the object. What should its type be especially in the absence of a single root class? Can we define it for MI? What about virtual base classes? The extension eliminates all these questions and allows for a simple expressive syntax. For simplicity, the clone example is used throughout this paper. The extension is not limited to classes that return their own type, nor must each level of the inheritance tree have a redefinition of the virtual function. For example the following would be legal under the proposal.

```
struct XX {
...};

struct YY : public XX {
...};

struct A  {
      virtual XX & func();
...}
struct B : public A {
      // inherits A::func()
...};
struct C : public B {
      virtual YY func();       // THIS WOULD FAIL NOW
...};
```

WORK AROUND

The extension can be written with a work around in C++. The work around is extremely messy, and is safe only if you follow the rules exactly (with no compiler checking) It also introduces inefficiencies that could easily be eliminated by the compiler directly.

The work around relies on the fact that non virtual functions CAN differ only by return type. It uses the non virtual function to transfer to a tandem virtual function to provide the dynamic behavior required.

Note that the intent of this work around is to simulate a virtual function (called clone). This virtual function would have all the properties expected of any virtual function and in addition have a different return type for each override of clone. The "helper functions" are implementation details that would not be used outside of the abstraction. In order to force the execution of a particular version of clone (inhibit the virtual mechanism) you must be aware of the actual name of the helper virtual function called and externally cast to the proper type. This is not considered of any consequence since it is a relatively rare construct.

SIMPLE SINGLE INHERITANCE

```
struct A {
      A * clone() { return  A_clone();};
      virtual A * A_clone() { return new A(*this); };
...};

struct B : public A {
      B * clone() { return (B *) A_clone(); };
      virtual A * A_clone() { return new B(*this); };
... };
```

Note that proposal refers to the static type of the function. given

```
A * pa;
B * pb;
B b;
pa = pb = &b;
pa->clone();  // type is A *
pb->clone();  // type is B *
```

Note that the actual function used in coding would always be "clone()". To get the virtual behavior, we must always transfer to a virtual function. To get the restriction of type behavior, we must force a cast to the new type. The work must be done in the virtual function so that it will be done correctly

regardless of the context. The clone function itself is the one called in outside code. This function must call the virtual helper function to give the virtual behavior. Since all the casting is manual, the compiler will not help us if we make a mistake.

MULTIPLE INHERITANCE

```
ASSUME A & B from above.

struct C {
      C * clone() { return C_clone(); };
      virtual C * C_clone { return new C(*this); };
...};

struct D : public B, public C {
      D * clone() { return (D *) A_clone();
      A * A_clone() { return  new D(*this); };
      C * C_clone() { return (C *) clone(): };
...};
```

When you use MI, the pseudo virtual function (clone) will be implemented by multiple virtual functions. The easiest way to handle this is to do the actual work in only one virtual function (here A_clone) and use casting to transfer to the other legs (here C_clone). In subsequent descendant classes, only the one virtual function need be implemented. Note that this works but causes the additional overhead of a second virtual call if called in the context of the class whose virtual function was discontinued.

Note that this shows an interesting case in the merging of two different trees having the same virtual function that are semantic equivalents. While the A based tree could continue to use A * returns, it is unlikely that a C based tree would implement an operation returning an A *.

VIRTUAL INHERITANCE

```
ASSUME A from above.

struct C  : public virtual A{
      C * clone() { return C_clone(); };
      virtual C * C_clone { return new C(*this); };
      virtual A * A_clone { return C_clone(); };
...};

struct D : public C , public virtual A {
      D * clone() { return (D *) C_clone(); };
      virtual C * C_clone { return new D(*this); };
...};
```

Note with virtual inheritance, you can no longer use the trick of casting up from the base class. Instead you must add a new virtual function in the class that first inherits the virtual base class. This allows you to recover the type information. Note that this causes an additional virtual function to be called when accessed from the context of the virtual base.

WORK AROUND 2

The second work around is based on another extension to be examined this conference. Run time type identification is not yet available in C++ but it has been discussed in the committee and is generally agreed upon as a feature that will eventually be required. This allows a simplification of the

above coding style. In particular, it eliminates the requirement for an additional virtual function to be added when adding a virtual base class. We will assume the syntax of a run time cast is identical to that of a static cast as per the paper by Bjarne Stroustrup and Dmitry Lenkov [3]. We will point these out with comments.

Note that runtime type identification would allow us to simply cast the result of the operation (clone) where required, but this is not our goal. What we are attempting to implement is a function that provides proper typing of the operation wherever it is used without the user doing typecasts, dynamic or static.

SIMPLE SINGLE AND MULTIPLE INHERITANCE

The availability of dynamic casting does nothing to change the cases of single and non–virtual multiple inheritance.

VIRTUAL INHERITANCE

```
ASSUME A from above.

struct C  : public virtual A{
      C * clone() { return (C *) A_clone(); };
                  // (C *) is a run time cast operation -
                  // this might fail - see note #.
      virtual A * A_clone { return new C(this); };
...};

struct D : public C , public virtual A {
      D * clone() { return (D *) C_clone(); };
                  // (D *) is a run time cast - see note #
      virtual A * A_clone { return new D(*this); };
...};
```

Note that dynamic casting allows us to eliminate the additional virtual function to enable us to cast up from an A * to a C * and A * to D * in this example. If the class is coded correctly the cast should always succeed, but if there is an ambiguous cast, or the work around precepts are violated (see below) this cast will return 0. Once a virtual base class is used, all subsequent casts will be dynamic.

WORK AROUND PROBLEMS

Consider the following example

```
struct A {
      A * clone() { return  A_clone();};
      virtual A * A_clone() { return new A(*this); };
...};

struct B : public A {
      B * clone() { return (B *) A_clone(); };
      virtual A * A_clone() { return new B(*this); };
... };

struct C : public B {
      virtual A * A_clone() { return A(this); };
```

Although someone implement clone is unlikely to make such a mistake, it could easily happen in more complicated cases. Note that although the definition for C::A_clone type checks correctly, we have the following bad behavior:

```
A * pa;
B * pb;
C c
;
pa = pb = &c;
pa->clone();  //  OK type is A *
pb->clone();  //  ERROR illegal cast of A* to B * in B::clone
```

In general the work around contains a number of casts that assume the programmer is keeping strict control of the actual types of objects. The breach of type safety could happen easily and not in a local that would be easy to check

IMPLEMENTATION

We are describing one possible method of implementing the proposed extens[illegible]. For another [illegible]he original paper by Snyder [1]. The plan is to implement the first work around directly by the compiler. It assumes each new definition of the virtual function is shadowed by a non virtual function. The unique naming of the real virtual functions and the creation of new ones where required are left to the compiler.

SIMPLE SINGLE INHERITANCE

```
struct A {
      virtual A * clone() {  new A(*this); };
...};

struct B : public A {
      B * clone() { return ne- 3(*this); };
... };
struct C : public B {
      C * clone() { return ne- C(*this); };
... };
```

In the trivial case of single inheritance this requires no additional code gene[illegible]on. Since an A *, a B *, and a C * are the s[illegible]for an obje[illegible]es A, B and C. Note that the c[illegible]er m[illegible]still check [illegible]rn types and dete[illegible]rors such [illegible]clone would attempt to retu[illegible]*;

ULTIPLE IN[illegible]ITANC

```
ASSUME A & B from above..

struct C {
      virtual C * clone() {    w C(*this); };
...};

struct D : public B, public C
      D * clone() { new D(*th[illegible]); };
...};
```

In the case of MI, only one of the base classes points to the same place as the entire object and the other points to an enclosed object. "clone" is now actually two different virtual functions A::clone and C::clone. As in the work around, there is only a single implementation. There is no work in implementing the A::clone path as in the single inheritance case, it points to the entire object. A compiler generated pseudo function could adjust the return pointer whenever clone is called from a C context using the vtable slot for C::clone. Note that the the compiler can keep track of this information for subsequent classes and perform any adjustments once rather than encoding this information in multi-

ple virtual functions. Also note that the compiler already has two distinct vtbl slots to help it keep track of this information.

VIRTUAL INHERITANCE

```
ASSUME A from above.

struct C : public virtual A{
      C * clone() { return new C(*this); };
...};
struct D : public virtual A{
      D * clone() { return new D(*this); };
...};
```

Once again the compiler is armed with information that the programmer is not. While the compiler can perform the same trick of adding a new virtual function, it might also be able to handle the casts through access to the vtable of the returned variable, perhaps using the same mechanism as run time type identification. The compiler would then know for sure that it is safe to perform the cast from the original return types base class.

CONSEQUENCES

As can be seen from the above examples, the code with the extension is considerably easier to write, look at and understand. Without the change you must either adopt the tortured style of the work around, or use casts as required throughout the code. As stated before, the change does not have any impact on syntax. It also will not affect any working program, since the current behavior is a subset of the requested behavior.

By making the change in the language, it would allow the compiler to perform optimizations on these types of programming constructs. For example, when a virtual function is called in a static context, that is one in which the member variable is not either a pointer nor a reference, the compiler can bind the function directly to the specific subroutine avoiding the vtbl look up. This is not possible when using a transfer function. This change would have no effect on any code not using the feature.

Static checking is enhanced by this feature. It eliminates a number of situations where casting would otherwise be required to recover lost type information. Sets of functions with covariant return types are relatively common. By having the compiler understand the semantics, it eliminates the potential of errors such as the example in the problems section This would be impossible to catch in the work around case since it is masked by type casting operations, but easily caught by the compiler. The compiler would be able to flag any problems with access or ambiguity at the same time.

We would certainly find this change easy to explain to both novices and experts. Indeed, it is hard to explain why you can't write a clone function in C++. The work around is relatively hard to explain and document. It also opens add on classes to potential problems.

Martin [2] raised a question about the effect on pointers to member functions. We would assume that given A and B above:

```
A * ( A::*pa) () = &A::clone;
A * ( B::*pb) () = pa;  // legal
B * ( B::*pb) () = pa;  // ERROR casting doesn't know about
                        // anything but the function address
```

Note that the original pointer pa is assigned to a function with class A as a context. Even though B::clone has return type B *, the function address captured in pa is A::clone with return A *and remains so when promoted. Note that this would be the same for the work around or the extension and goes along with treating a virtual function identically with non–virtual functions.

Another issue is that the original proposal was tied to allowing contravariant argument types. In fact, what people often want is covariant arguments which are inherently type unsafe. The need for covariant return types (this proposal) far exceeds the need for contravariant arguments.

SUMMARY

The proposal removes a restriction on return types of virtual functions that exceeds the requirements for strict type safety and can compromise type safety. It makes code using the feature easy to write, document and understand. It eliminates the need for type casts in situations where the type information is lost due to the virtual return type restriction. It gives the compiler more opportunity to perform optimizations by directly stating the programmers intention rather than hiding it in layers of transfer functions. It has no impact on the syntax or on existing code. The only negative is the cost of compiler complexity to implement it. Judging from the relatively mechanical nature of the work around, this should not be a major obstacle.

---

[1] Alan Snyder, A Proposal to Revise C++ Subtyping Rules, (X3J16/90-0049)

[2] Martin J O"Riordan, Polymorphic Over-Riding of Function Return Types, (X3J16/91-0051)

[3] Bjarne Stroustrup, Dmitry Lenkov, Run_Time Type Identification for C++, (X3J16/92-????=WG21/N????)