X3J16/92-0037
WG21/N0114

The effect of construction order on virtual function overriding

Ernest Choi
IBM Canada Ltd.
844 Don Mills Rd.
North York, Ontario
Canada
M3C 1V7
(ernestc@torolab6.iinus1.ibm.com)

March 3, 1992

Abstract

The ANSI C++ standard does not state how virtual functions are affected by the order of construction of base classes. Four alternatives are suggested: undefined, implementation defined, overrides based on original definition of constructing class, overrides based on construction order of most derived class. It is proposed that the standard explicitly state that the third alternative (overrides based on original definition of constructing class) is chosen.

1 Terminology

- Constructing class

  The type of the object being constructed is considered the constructing class. The constructing class is also known as the most derived class.

- Construction order

  The order by which the base classes of a class are constructed.

- Construction phase

  A particular step in a class's construction order. A phase is associated with one base class of the class object being constructed. Although the most derived class is not

105

the most derived class's members are constructed. Hence, the last construction
phase is always that of the most derived class.

2 Construction vs. Destruction

This paper discusses the issue with regards to object construction only. However this issue applies equally to object destruction. Loosely speaking, the words "construction", "constructing" and "constructor" can be replaced with the words "destruction", "destructing" and "destructor" without violating the arguments presented here. The only other consideration to remember when making these mental substitutions is to note that destruction order is the reverse of construction order. Hence, for example, phrases such as "have been constructed" can be replaced with "will be destructed".

3 Introduction

With regard to construction order, the ANSI C++ standard merely states that base classes are constructed in declaration order with virtual bases being constructed first (§12.6.2). It also states that virtual functions may be called from a constructor and that

> (§12.7) The function called will be the one defined in the constructor's (or destructor's) own class or its bases, but *not* any function overriding it in a derived class. This ensures that unconstructed objects will not be accessed during construction or destruction.

The above implies that the mentioned "derived class" is a class derived from the constructor's own class (and not derived from a base class).

The above also states that the only virtual functions that can be called will be those overridden by the constructing class and by its bases. However, this ignores the fact that, in a multiple inheritance hierarchy with virtual bases, other base classes that were constructed before the constructing base class may have overridden some virtual functions as well.

For example, consider the declarations in Figure 1.

The class hierarchy diagram for Right is:

```
Top { virtual void a(); }
 |
Right {}
```

When a Right object is constructed, its base class Top is constructed first and then Right is constructed. When Right::Right() calls the function a() it will resolve to a call to Top::a() since Right did not override a().

105

```
class Top
{
public:
   virtual void a();
};

class Left : public virtual Top
{
public:
   void a();
   Left(){ a(); }
};

class Right : public virtual Top
{
public:
   Right(){ a(); }
};

class Bottom : public Left, public Right
{
public:
   Bottom(){ a(); }
};
```

Figure 1:

105

```
                    Top { virtual void a(); }
                    / \
Left {void a();}   Right {};
                    \ /
                  Bottom {};
```

When a Bottom object is constructed the construction order is Top, Left, Right and then Bottom. Now after Left is constructed the virtual function a() would've been overridden by Left::a(). (As far as Left is concerned, it was the one that overrode a() last.) When Right::Right() calls a(), which a() will be called: Top::a() or Left::a()?

If the answer is Top::a(), then it implies it is no longer true that Left has overridden a(). This is very strange considering that Left::a() will reappear when we enter the construction phase of Bottom. When Bottom constructs, and suppose its constructor calls a(), it will resolve to a call to Left::a() (since it is most dominant). Why then, during the construction of Right, should a call to a() resolve to Top::a() instead?

If the answer is Left::a(), then it implies that, for a class derived from a virtual base, one cannot assume that a call from a constructor to a virtual function introduced by that virtual base will always resolve to the same virtual function. Whether or not this poses a problem for a class designer is not clear.

In short, the question is:

> What is the expected behaviour of calling a virtual function, that was introduced by a virtual base class, from a constructor of a multiply inherited base class derived from the virtual base?

This paper will present four alternatives to this question.

1. Undefined behaviour.

   A base class X that was constructed in an earlier construction phase may have overridden a virtual function in a virtual base. If X is a base class of the current construction phase, say class Y, all calls from Y's constructors to virtual functions overridden by X will resolve to X's overriding functions. (This is the normal case.) But if X is not a base class of Y, the resolution of the virtual call may reach X's overriding functions or the overriding functions of Y (or some base class of Y). The exact resolution is undefined.

2. Implementation defined.

   The situation is the same as (1) but the virtual function call resolution from Y's constructor if X is not a base class of Y is to be specified by the implementation. Hence the implementation must choose between (3) and (4) below.

105

3. Static overrides.

   Virtual function overrides are based on the original definition of the constructing class.

   The situation is the same as (1) but the virtual function call resolution from Y's constructor if X is not a base class of Y will resolve only to the overriding functions of Y or some base class of Y. The overrides contributed by X are ignored while Y is constructing.

4. Dynamic overrides.

   Virtual function overrides are based on the construction order of the most derived class.

   The situation is the same as (1) but the virtual function call resolution from Y's constructor if X is not a base class of Y will resolve to the overriding functions of X, Y or some base class of Y. The overrides contributed by X are evident unless they were overridden by some construction phase that occurred after X and before (and including) class Y's phase.

4 Observations

4.1 An Example

The example code shown in Figure 2 will be used in the observations below.

Suppose we had a window class that can perform special actions activated by key presses from the keyboard. Furthermore, these keys and associated actions can be specified by a derived class during construction.

In the example, the member function Window::AddKey() is used to inform the Window class that a given key is associated with the given function. If the key is pressed, the function will be called. The key's name will not be displayed in the window. (One can imagine that it will be displayed in the window if the user presses a "help" key but this is not important in the discussions below.)

A MenuedWindow object will display a menu if the escape key is pressed. The window is informed of this special "show menu" key when MenuedWindow constructs.

The BorderedWindow class will show all the keys that the window supports by displaying icons of those keys in the border of the window. This class will override AddKey() so that the icon is added to the border before the Window::AddKey() is called.

The BorderedMenuedWindow class tries to combine the behaviour of the BorderedWindow and MenuedWindow. A BorderedMenuedWindow should have a border containing a list of special keys, one of which is a key to display a menu. It relies on the construction order of the base classes to set up the virtual function table properly so that the constructor of MenuedWindow will call BorderedWindow::AddKey(). In effect, the

105

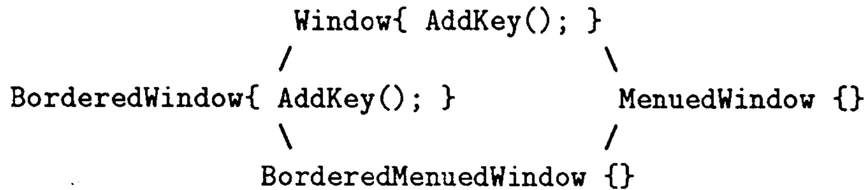
```
typedef void (*Func)();

struct Window {
   virtual void AddKey( int key, char * name, Func f )
   {
      // save 'key' as the function key .
   }
};

struct BorderedWindow : virtual Window {
   void AddKey( int key, char * name, Func f )
   {
      // ... display key icon in border ...

      // perform usual action
      Window::AddKey(key, name, f);
   }
};

extern void menuUp();

struct MenuedWindow : virtual Window {
   enum Keys { EscKey = 27 /* Ascii */ };
   MenuedWindow(char*name)
   {
      // will border be updated too?
      AddKey(EscKey, "Show Menu", menuUp);
   }
};

struct BorderedMenuedWindow : BorderedWindow, MenuedWindow {
};
```

Figure 2:

105

BorderedMenuedWindow is trying to "intercept" the call to AddKey (from MenuedWindow) by placing BorderedWindow ahead of MenuedWindow in its base class list.

Note that the class hierarchy for BorderMenuedWindow is



The remainder of this section will summarize the effect of each choice on this example and on virtual function calls in general.

Author's note: This example is a hybrid of two published examples. The original examples can be found in [Strous 90, pages 233–236] and [Gorlen 90, pages 316–318]. The class names have been changed to make it look like a real application. The construction order dependency idea is from [Gorlen 90].

4.2 Undefined Behaviour

During the construction of MenuedWindow, one cannot be guaranteed that the overriding function BorderedWindow::AddKey() is called. Furthermore, neither can one be guaranteed that Window::AddKey() will be called instead!

With this choice, one can no longer trust a virtual function call in the presence of virtual bases in a given construction phase to resolve to a particular overriding function.

4.3 Implementation Defined

Although the behaviour can now be determined by the programmer, the code may break when it is ported to a different platform.

Note that "implementation defined" features are normally reserved for features that are hardware specific or features that depend upon the particulars of a host environment. The issue under discussion is not any of those.

4.4 Static Overrides

When a BorderedMenuedWindow object is constructed the construction order is Window, BorderedWindow, MenuedWindow and then BorderedMenuedWindow. Now after BorderedWindow is constructed the virtual function AddKey() would've been overridden by BorderedWindow::AddKey(). (As far as BorderedWindow is concerned, it was the one that overrode AddKey() last.)

constructing class then when MenuedWindow::MenuedWindow() calls AddKey(), Window::AddKey() will be reached. (Thus, the key will be added but no icon will be present for that key.)

This implies that it is no longer true that BorderedWindow has overridden AddKey(). This is very strange considering that BorderedWindow::AddKey() will reappear when we enter the construction phase of BorderedMenuedWindow. (Suppose the constructor of BorderedMenuedWindow trys to add a key using AddKey(). It will be successful in adding the key and displaying the icon that represents that key. The unsuspecting user will wonder why the icon for this key appears while menu key icon did not!)

To cope with this behaviour, the class BorderedMenuedWindow must be rewritten so that AddKey() is called after the constructor of MenuedWindow is called. This implies that the construction of MenuedWindow is split into two parts: the constructor and some initialization function. The constructor of MenuedWindow will initialize as much as it can without calling any functions[1].

The second initialization function will complete the initialization of MenuedWindow by calling AddKey()[2].

Hence, in general, a class derived from a virtual base with virtual functions should not call any functions from its constructor. Instead, it should split its construction in two as mentioned above.

This choice is simple to implement. All implementations must determine the set of virtual functions that are most dominant for a given class. If one thinks of each class as having a set of virtual functions that are most dominant then, to determine the set of virtual functions in the most derived class, one simply needs to form the union of all the sets in the immediate base classes (of the most derived class) and then replace those functions that are overridden (by the most derived class). In short, one only needs to examine the most immediate base classes to form the virtual function tables for a given derived class.

---

[1] MenuedWindow constructor must avoid calling any functions because the called function may call AddKey() and thus reach the wrong function. In general, the constructors of all classes derived from a virtual base class with virtual functions must avoid calling functions.

[2] The second initialization function should be called by the routine that actually declares or allocates an object of BorderedMenuedWindow (or an object of some class derived from it). This is because only when the object is completely constructed will all the virtual function overrides from the base classes are present.

Note that one cannot be sure that the object is completely constructed until one exits the constructor of BorderedMenuedWindow. This is because the BorderedMenuedWindow can itself be a base class and hence, just like MenuedWindow, its constructor may resolve incorrectly to a virtual function that is supposed to be overridden by another base class.

```
struct Window {
   virtual void AddKey();
   virtual void SetName();
};

struct BorderedWindow : virtual Window {
   void AddKey();
   BorderedWindow()
   {
      SetName();
   }
};

struct MenuedWindow : virtual Window {
   void SetName();
   MenuedWindow()
   {
      AddKey();
   }
};

struct BorderedMenuedWindow : BorderedWindow, MenuedWindow {
};
```

Figure 3:

4.5 Dynamic Overrides

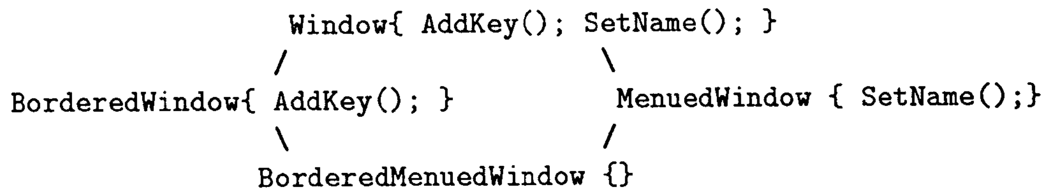
When a BorderedMenuedWindow object is constructed the construction order is Window, BorderedWindow, MenuedWindow and then BorderedMenuedWindow. Now after BorderedWindow is constructed the virtual function AddKey() would've been overridden by BorderedWindow::AddKey().

When MenuedWindow is constructed, the overrides contributed by BorderedWindow will not be changed (unless MenuedWindow explicitly overrides them).

Hence, this behaviour will support the example in Figure 2.

Unfortunately, this choice cannot support the situation where two "sibling" base classes override functions in the virtual base in an effort to intercept calls. Consider the code in Figure 3.

In short, BorderedWindow is trying to intercept all calls to AddKey() and MenuedWindow is trying to intercept all calls to SetName(). The class hierarchy for this version



With the above definitions, BorderedWindow is constructed first. SetName() would not have been overridden by MenuedWindow when BorderedWindow::BorderedWindow() is called. Hence MenuedWindow will fail to intercept the call to SetName().

If the declaration of BorderedMenuedWindow is changed so that the base class list is reversed, MenuedWindow will be constructed first. In this case AddKey() would not have been overridden by BorderedWindow so when MenuedWindow::MenuedWindow() calls AddKey, it will reach Window::AddKey(). Hence, in this case, BorderedWindow will fail to intercept the call to AddKey().

In short, there is no way to order the base classes so that both base classes are able to intercept the other's virtual function call. The solution to the original hierarchy is to split the construction of MenuedWindow in the manner described in the previous choice so that the creator of the object can complete the initialization of the MenuedWindow.

However, since a class should not assume the position it will take in a base class list, *both* classes should split its construction into two parts.

Hence, in general, a class derived from a virtual base with virtual functions should not call any functions from its constructor. Furthermore, it should split its construction in two as mentioned above.

The implementation of this choice involves going through each construction phase and determining which functions are dominant during that phase. This means that it is not sufficient to obtain the overriding functions from the immediate base classes. One must traverse the class hierarchy in the same order as the construction order so that one can keep track of the overriding virtual functions at each phase. Furthermore, one only needs to detect changes from the previous construction phase to the current construction phase.

5 Evaluation

5.1 Undefined behaviour

Leaving the behaviour undefined is unsatisfactory since one can no longer trust a virtual function call to resolve to a overriding function consistently. It is important that this behaviour is defined so that one can predict the outcome of a virtual function call (direct or indirect) from a constructor.

The only reason this choice was presented is to provide a "default" choice to fall back on if all the other choices are found to be less desirable than an undefined behaviour.

105

5.2 Implementation defined

This is satisfactory if one is writing for one compiler on one platform. But for portable code, this solution is not desired. It will not guarantee consistent behaviour during object construction for all platforms.

Furthermore, this issue is not one that should be considered "implementation defined" since it is not a hardware or environmental issue.

(Author's note: Perhaps ANSI can invent another category, such as "implementation specified" for behaviours that are to be determined by the implementation?)

5.3 Static overrides

This choice will allow one to predict which virtual function will be called when calling from a constructor or destructor of some intermediate base class. Each constructing phase behaves as if it was the most derived class. That is, the virtual function overrides are determined by the declaration of the constructing phase and ignores the possbility that it can be a base class of another class. However, as noted in the example in the introduction, it may lead to contradictory behaviour between two construction phases.

For instance, in example 2, if BorderedWindow calls AddKey() during its construction phase it will resolve to a call to BorderedWindow::AddKey(). However when MenuedWindow constructs, a call to AddKey() will resolve to a call to Window::AddKey(). Why doesn't it resolve to the overridden function BorderedWindow::AddKey() instead? Shouldn't the virtual function override from the previous construction phases be present in the later construction phases? Another strange event occurs is when BorderedMenuedWindow constructs: a call to AddKey() will resolve to BorderedWindow::AddKey(). In effect, BorderedWindow will override AddKey() again without re-entering its construction phase!

If a class's identity is defined by its function members as well as its data members, then one does not expect to see its function members removed after it has been constructed (just as one does not expect its data members to be removed). However, that is exactly what this choice will do as the construction phases are traversed.

For the sake of users that expect "Dynamic Overrides", it would be wise for an implementation to issue a warning at a virtual function call (to a virtual function introduced by a virtual base) from a constructor or destructor. The warning would inform them that the actual virtual function reached will always be the most dominant function in the constructor's class.

One advantage that this choice has over the following choice is that it is simple to implement.

105

This choice will allow consistent behaviour between construction phases. In effect, each construction phase will respect the possibility that it can be a base class of another class. Hence, the virtual function overrides are not limited to those that are most dominant in the constructions phase's class declaration.

If one regards a class's identity to be defined by its function members as well as its data members, then this choice preserves class identity better than Static Overrides. Once a base class overrides a virtual function it stays overridden until it is overridden again by a more dominant class. This behaviour is documented in [Gorlen 90] as the expected behaviour during construction.

This is not a surprising expectation since this is the behaviour for classes without virtual bases. One does not expect the presence of virtual bases to change this behaviour. And nowhere in the standard does it state that this behaviour should change due to the presence of virtual bases.

One may ask: how can a class designer predict the behaviour of a constructor if it is possible to call a user's overriding function instead of an overriding function in his own class library? Note that an overriding function must behave in a manner such that the routines that call the virtual function does not realize that it was overridden. If it doesn't, then the overriding function is in error. This is a fundamental property for all overriding virtual functions. As a result the class designer will be able to predict the behaviour of the constructor even though the overriding function may be different.

For the sake of users that expect "Static Overrides", it would be wise for an implementation to issue a warning at a virtual function call (to a virtual function introduced by a virtual base) from a constructor or destructor. The warning would inform them that the actual virtual function reached may not be the most dominant function in the constructor's class.

The implementation of this choice may seem to be expensive, but it is performed only once per class that has a virtual base with virtual functions. The number of tables generated may actually be less than the number of tables produced by the Static Overrides case if the virtual functions are not overridden by all "arms" of a class hierarchy. (The example in the introduction is one such hierarchy. The class Right can use the virtual function table used by Left since Right does not override any virtual functions.)

6 Conclusions

The "Undefined Behaviour" choice is not desired since it is better to have this behaviour specified.

The "Implementation Defined" choice is not desired since it leads to nonportable code.

The "Static Overrides" choice leads to inconsistent virtual function call resolution between two construction phases.

105

The "Dynamic Overrides" choice gives a consistent behaviour to virtual function call resolution and preserves object identity between construction phases. However, it is more expensive to implement than the "Static Overrides" choice.

We only need to decide between "Static Overrides" and "Dynamic Overrides" because the behaviour must be specified and be consistent across platforms.

It is difficult to judge "correctness" of either behaviour. Whether or not a class's overriding virtual functions should remain to be "visible" to the next construction phase is unclear (although it seems natural to expect the overrides to remain). It is the author's belief that debating the correctness of either behaviour will degenerate into a religious argument. There is nothing in the standard that tells us which view is the "correct" view.

As far as the practical aspects of class design is concerned, there is no compelling reason to choose "Static Overrides" over "Dynamic Overrides" (or vice versa). Regardless of the choice, the class designer should structure class initialization so that functions are not called until all the phases of construction have completed.

Since one copes with the shortcomings of the two choices in the same manner, and it is not clear what the correct behaviour should be, the deciding factors will be practical ones. In particular, the deciding factors are:

- Is it easier to understand and explain?
- Will constructors be easier to debug and maintain?
- Is it easier to implement?
- Do other implementations support it?
- Can current debugging technology support it?

"Static Overrides" wins on all points.

7 Recommendations

ANSI should clarify how the virtual functions from a virtual base will be overridden as each construction phase of the most derived class is traversed. In particular, the statement

> (§12.7) The function called will be the one defined in the constructor's (or destructor's) own class or its bases, but *not* any function overriding it in a derived class.

should be rewritten as

> (§12.7) The function called will be the one defined in the constructor's (or destructor's) own class or its bases, but *not* any function overriding it in a derived class or a sibling class introduced by multiple inheritance.

Two base classes are siblings if one is not derived from the other, and both are derived from the same virtual base class and both are inherited by the same derived class.

8 Summary

The ANSI C++ draft does not say whether or not the construction order of a class affects the set of virtual functions from a virtual base class. Although is seems natural to expect virtual function overrides to remain overridden as the construction phases are traversed, the practical aspects of using classes prefer a simpler, if flawed, behaviour. The ANSI C++ standard should explicitly state that the virtual function overrides from sibling classes will not be called from a constructor or destructor.

9 Appendix

9.1 Existing practice

- Cfront:

  Both 2.1 and 3.0 are inconsistent. The example in Figure 1 will exhibit behaviour as if Dynamic Overrides was implemented. However, if Right overrides a function that is not overridden by Left (as shown in Figure 4), then Cfront will compile the code as if Static Overrides was implemented.

- Zortech: V3.0 will behave just like Cfront. That is, the example in Figure 1 will behave as if Dynamic Overrides is implemented while the example in Figure 4 will behave as if Static Overrides is implemented.

- Borland: Version 3.00 compiles both Figure 1 and Figure 4 as if Static Overrides is implemented.

- IBM XL/ C++: This compiler implements Static Overrides.

References

[Strous 90] B. Stroustrup and M. Ellis, *The Annotated C++ Reference Manual*, Addison-Wesley, Reading, Massachusetts, 1990.

[Gorlen 90] K. E. Gorlen, S. M. Orlow, and P. S. Plexico, *Data Abstraction and Object-Oriented Programming in C++*, John Wiley & Sons Ltd, Chichester, West Sussex, England, 1990

105

```
class Top
{
public:
   virtual void a();
   virtual void b();
};

class Left : public virtual Top
{
public:
   void a();
   Left(){ a(); b(); }  // Left::a(), Top::b()
};

class Right : public virtual Top
{
public:
   void b();
   Right(){ a(); b(); } // Top::a(), Right::b()
};

class Bottom : public Left, public Right
{
public:
   Bottom(){ a(); b(); } // Left::a(), Right::b()
};
```

Figure 4:

105