Doc No:SC22/WG21/N3163 = PL22.16/10-0153

Date: 2010-10-10

Project: JTC1.22.32

Reply to: Herb Sutter
Microsoft Corp.
1 Microsoft Way
Redmond WA USA 98052
Email: hsutter@microsoft.com

Override Control Using Contextual Keywords

In Rapperswil, we considered three options for the virtual control attributes in C++0x.

Option 1 (Status quo): Use [[attributes]].

The FCD syntax (following the attribute placement style of the examples in the FCD):

```
class A [[base_check]] : public B {
  virtual void f [[override]] () { … }
  virtual void g [[hiding]] () { … }
  virtual void h [[final]] () { … }
};

class C [[final]] { };
```

The Rapperswil straw polls viewed this option as least desirable, I believe correctly, for a number of reasons including that attributes should not be keywords in disguise.

Option 2: Use reserved keywords.

We can instead use reserved keywords. Strawman example:

```
class A base_check : public B {
  virtual void f override_func () { … }
  virtual void g hiding_func () { … }
  virtual void h final_func () { … }
};

class C final_class { };
```

The main drawback to this approach is that it will take identifiers away from the user namespace, and so would force us to choose intentionally ugly names to minimize code breakage.

Option 3: Use context-sensitive keywords.

C++/CLI [1] already supports most of these as contextual keywords, or identifiers that are:

- not reserved keywords; and
- have special meaning only when they appear in a grammar position where no user identifier can legally appear.

C++/CLI intentionally uses this technique in order to simultaneously achieve:

- **full backward compatibility**, because the identifier is only special in that grammar position and so cannot conflict with existing code which might already use those words as identifiers; and
- **nice names**, because since there's no conflict with user identifiers, we can use whatever nice name we want to.

Here's an example of how it would look. The following code is a variation of legal C++/CLI code that has been used in the field since 2005, only with the identifiers renamed to match the C++0x FCD preferred words (e.g., in C++/CLI, "final" is called "sealed" but has the same meaning) and base_check added:

```
class A base_check : public B {              // 1
  virtual void f() override { … }           // 2
  virtual void g() hiding { … }             // 3
  virtual void h() final { … }              // 4
};

class C final {                              // 5
  int final, override, hiding, base_check;   // 6: ok, still legal identifier names
};
```

In lines 1 and 5, "base_check" and "final" appear where no user identifier (of those or any other name) can legally appear, namely after the class name and the following : or { token.

In lines 2, 3, and 4, "override," "hiding" and "final" appear where no user identifier (of those or any other name) can legally appear, namely between the end of the parameter list and the following -> or { or ; token.

Line 6 illustrates that the names can continue to be freely used as user identifiers without restriction.

This approach has been implemented in a major shipping compiler since 2005, and used successfully by hundreds of thousands of developers without any known source code compatibility issues. Note this includes not only new code, but also customers recompiling very large quantities of existing C++ code without change and not encountering any problem.

The observation was made in Rapperswil that this approach can make error recovery and syntax highlighting more difficult. For example, syntax highlighting is a bit harder because instead of globally highlighting the keyword you need to do parsing to know whether the identifier is in the location where it has special meaning and should be highlighted. But this is not exceptionally difficult, particularly not in comparison to other far more difficult things we already have to do in C++ compared to other languages.

These are minor inconveniences for a few compiler writers for a week, but are seamless for users. This is the right tradeoff. Otherwise, having ugly globally reserved names (Option 2) or inappropriate and visibly bolted-on attributes (Option 1) make things easier for a few compiler writers for a week, but are things millions of users will have to live with forever.

Details From C++/CLI Specification

For reference in drafting proposed wording should this proposal be acceptable to WG21, here are the relevant portions of the C++/CLI specification which was written as a "diff" from C++03. I've removed most of what isn't relevant for this paper, but there are still some references to other related features that C++/CLI allows (e.g., some examples show the feature of renaming a virtual function while overriding it); please just ignore those for the purpose of this paper.

Note that, as shown in the first quoted section, this "contextual keyword" technique was used widely for many keywords, with great success and no known user code breakage.

9.1.1 Identifiers

Certain places in the Standard C++ grammar do not allow identifiers. However, C++/CLI allows a defined set of identifiers to exist in those places, with these identifiers having special meaning. [*Note:* Such identifiers are colloquially referred to as context-sensitive keywords; nonetheless, they are identifiers. *end note*] The identifiers that carry special meaning in certain contexts are:

```
abstract    delegate    event       finally     generic     in
initonly    internal    literal     override    property    sealed
where
```

When referred to in the grammar, these identifiers are used explicitly rather than using the *identifier* grammar production. Ensuring that the identifier is meaningful is a semantic check rather than a syntax check. An identifier is considered a keyword in a given context if and only if there is no valid parse if the token is taken as an identifier. That is, if it can be an identifier, it *is* an identifier.

[…]

8.8.10.1 Function overriding

In Standard C++, given a derived class with a function having the same name, parameter-type-list, and cv-qualification as a virtual function in a base class, the derived class function always overrides the one in the base class, even if the derived class function is not declared `virtual`.

```
struct B {
      virtual void f();
      virtual void g();
};
struct D : B {
      virtual void f();        // D::f overrides B::f
      void g();                // D::g overrides B::g
};
```

We refer to this as ***implicit overriding***. (As the `virtual` specifier on `D::f` is optional, the presence of `virtual` there really isn't an indication of explicit overriding.) Since implicit overriding gets in the way of versioning (§8.13), implicit overriding must be diagnosed by a C++/CLI compiler.

C++/CLI supports two virtual function-overriding features not available in Standard C++. These features are available in ref class types. They are explicit overriding and named overriding.

Explicit overriding: In C++/CLI, it is possible to state that

1. A derived class function explicitly overrides a base class virtual function having the same name, parameter-type-list, and cv-qualification, by using the function modifier `override`, with the program being ill-formed if no such base class virtual function exists; and
2. A derived class function explicitly does not override a base class virtual function having the same name, parameter-type-list, and cv-qualification, by using the function modifier `new`.

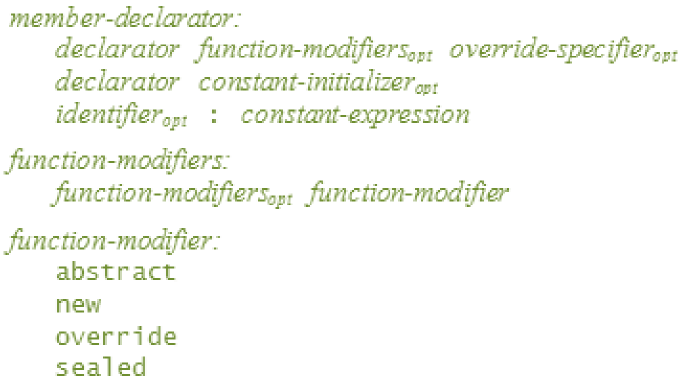
```
ref struct B {
      virtual void F() {}
      virtual void G() {}
};

ref struct D : B {
      virtual void F() override {}  // D::F overrides B::F
      virtual void G() new {} // D::G doesn’t override B::G, it hides
it
};
```

`D::F` must be virtual, and must be marked as such. On the other hand, `D::G` doesn't have to be virtual, and if it isn't, it shouldn't be marked as such.

[…]

The addition of overriding specifiers and function modifiers requires augmentations to the Standard C++ grammar for *function-definition* and to one of the productions of *member-declarator*. [*Note:* The two new optional syntax productions, *function-modifier* and *override-specifier*, appear in that order, after *exception-specification,* but before *function-body* or *function-try-block*. *end note*]

To allow attributes, function modifiers, and an override specifier on a function declaration that is not a definition, one of the productions for the Standard C++ grammar for *member-declarator* (§9.2) is augmented, as follows:



member-declarator:
    declarator function-modifiers$_{opt}$ *override-specifier*$_{opt}$
    declarator constant-initializer$_{opt}$
    identifier$_{opt}$ `:` *constant-expression*

function-modifiers:
    function-modifiers$_{opt}$ *function-modifier*

function-modifier:
    `abstract`
    `new`
    `override`
    `sealed`

[…]

A member function declaration containing any of the *function-modifier*s `abstract`, `override`, or `sealed`, or an *override-specifier*, shall explicitly be declared `virtual`. [*Rationale:* A major goal of this new syntax is to let the programmer state his intent, by making overriding more explicit, and by reducing silent overriding. The `virtual` keyword is required on all virtual functions, except in the one case where backwards compatibility with Standard C++ allows the `virtual` keyword to be optional. *end rationale*]

If a function contains both `abstract` and `sealed` modifiers, or it contains both `new` and `override` modifiers, it is ill-formed.

An out-of-class member function definition shall not contain a *function-modifier* or an *override-specifier*.

If a destructor or finalizer (§19.13) contains an *override-specifier*, or a `new` or `sealed` *function-modifier*, the program is ill-formed.

19.4.1 Override functions

The Standard C++ grammar for *direct-declarator* is augmented to allow the function modifier `override` as well as override specifiers.

override-specifier:
  = *overridden-name-list*
  pure-specifier

overridden-name-list:
  id-expression
  overridden-name-list , *id-expression*

In Standard C++, given a derived class with a function that has the same name, parameter-type-list, and cv-qualification of a virtual function in a base class, the derived class function always overrides the one in the base class, even if the derived class function is not declared `virtual`. This is known as ***implicit overriding***. A program containing an implicitly overridden function in ref classes and value classes is ill-formed. [*Note:* A programmer can eliminate the diagnostic by using explicit or named overriding, as described below. *end note*]

With the addition of the function modifier `override` and override specifiers, C++/CLI provides the ability to indicate ***explicit overriding*** and ***named overriding***, respectively.

If either the *function-modifier* `override` or an *override-specifier* is present in the derived class function declaration, no implicit overriding takes place. [*Example:*

```
ref struct B {
      virtual void F() {}
      virtual void F(int i) {}
};
ref struct D1: B {
      virtual void F() override {}        // explicitly overrides
B::F()
};
ref struct D2: B {
      virtual void F() override {}        // explicitly overrides
B::F()
      virtual void G(int i) = B::F {}     // named override of
B::F(int)
};
ref struct D3: B {
      virtual void F() new = B::F {}            // named override of
B::F()
};
```

end example]

[*Note:* A member function declaration containing the *function-modifier* `override` or an *override-specifier* shall explicitly be declared `virtual` (§19.2.4**Error! Reference source not found.**). *end note*]

An *override-specifier* contains a comma-separated list of names designating the virtual functions from one or more direct or indirect base classes that are to be overridden.

An *id-expression* that designates an overridden name shall designate a single function to be overridden. Lookup for the name given in the *id-expression* starts in the containing class. [*Note*: If the *id-expression* is an unqualified name, and the containing class has a function by the same name the program is ill-formed. It is not possible to override a function within the same class. *end note*] Further qualification is necessary if the base class name is ambiguous. That function shall have the same parameter-type-list and cv-qualification as the overriding function, and the return types of the two functions shall be the same.

[*Example:*

```
interface class I {
      void F();
};

ref struct B {
      virtual void F() { … }
};

ref struct D : B, I {
      virtual void G() = B::F, I::F { … } // override B::F and I::F
};
```

Both `B::F` and `I::F` must be listed separately. If the named override used just `F`, two names are found. Named overrides must designate a single function. *end example*]

[*Note:* The same overriding behavior can sometimes be achieved in different ways. For example, given a base class `A` with a virtual function `f`, an overriding function might have an *override-specifier* of `A::f`, have no *override specifier* or `override` *function modifier*, have the *function-modifier* `override`, or a combination of the two, as in `override = A::f`. All override `A::f`. *end note*]

The name of the overriding function need not be the same as that being overridden.

A derived class shall not override the same virtual function more than once. If an implicit or explicit override does the same thing as a named override, the program is ill-formed. [*Example:*

```
interface struct I {
      void F();
};

ref struct B {
      virtual void F() { … }
      virtual void G() { … }
};

ref struct D : B, I {
      virtual void G() = B::F { … }
      virtual void F() {} // error, would override B::F and I::F, but
                          // B::f is already overridden by G.
};
```

end example]

A class is ill-formed if it has multiple functions with the same name, parameter-type-list, and cv-qualification even if they override different inherited virtual functions. [*Example:*

```
ref struct D : B, I {
      virtual void F() = B::F { … }       // ok
      virtual void F() = I::F { … }       // error, duplicate
declaration
};
```

end example]

A function can both hide and override at the same time: [*Example:*

```
interface struct I {
   void F();
};
```

```
ref struct B {
   virtual void F() { … }
};
ref struct D : B, I {
   virtual void F() new = I::F { … }
};
```

The presence of the `new` function modifier (§19.4.4) indicates that `D::F` does not override any method `F` from its base class or interface. The named override then goes on to say that `D::F` actually overrides just one function, `I::F`. *end example*]

[*Note*: An override-specifier does not introduce that name into the class. *end note*][*Example:*

```
interface struct I {
      virtual void V();
};
ref struct R {
      virtual void W() {}
};
ref struct S : R, I {
      virtual void F() = I::V, R::W {}
};
ref struct T : S {
      virtual void G() = I::V {}
      virtual void H() = R::W {}
};
void Test(S^ s) { // s could refer to an S, T, or something else
      s->W();     // ok, virtual call
      s->R::W();  // nonvirtual call to R::W
      s->S::W();  // nonvirtual call to R::W
      s->S::F();  // ok (classes derived from S might need to do this,
                  //  and there’s no ambiguity in this case)
}
int main() {
      Test(gcnew S);
      Test(gcnew T);
}
```

end example]

[…]

19.4.2 Sealed function modifier

A virtual member function marked with the *function-modifier* `sealed` cannot be overridden in a derived class. [*Example:*

```
ref struct B {
      virtual int f() sealed;
      virtual int g() sealed;
};
ref struct D : B {
      virtual int f();     // error: cannot override a sealed function
      virtual int g() new; // okay: does not override B::g
};
```

end example]

[*Note:* A member function declaration containing the *function-modifier* `sealed` shall explicitly be declared `virtual`. *end note*] If there is no `virtual` function to implicitly override in the base class, the derived class introduces the virtual function and seals it.

Whether or not any member functions of a class are sealed has no effect on whether or not that class itself is sealed.

An implicit, explicit, or (in a CLI class type, a) named override can succeed as long as there is a non-sealed virtual function in at least one of the bases. [*Example:* Consider the case in which `A::f` is sealed, but `B::f` is not. If `C` inherits from `A` and `B`, and tries to implement `f`, it will succeed, but will only override `B::f`. *end example*]

[…]

19.4.3 Abstract function modifier

Standard C++ permits virtual member functions to be declared abstract by using a *pure-specifier*. C++/CLI provides an alternate approach via the *function-modifier* `abstract`. The two approaches are equivalent; using both together is well-formed, but redundant. [*Example:* A class `shape` can declare an abstract function `draw` in any of the following ways:

```
virtual void draw() = 0;                  // Standard C++ style
virtual void draw() abstract;       // function-modifier style
virtual void draw() abstract = 0;   // okay, but redundant
```

end example]

[*Note:* A member function declaration containing the *function-modifier* `abstract` shall be declared `virtual`. *end note*]

[…]

19.4.4 New function modifier

A function need not be declared `virtual` to have the `new` function modifier. If a function is declared `virtual` and has the `new` function modifier, that function does not override another function. However, for CLI class types, it can override another function with a named override. A function that is not declared `virtual` and is marked with the `new` function modifier does not become virtual and does not implicitly override any function.

[*Example:*

```
ref struct B {
   virtual void F() { Console::WriteLine("B::F"); }
   virtual void G() { Console::WriteLine("B::G"); }
};
ref struct D : B {
   virtual void F() new { Console::WriteLine("D::F"); }
};
int main() {
   B^ b = gcnew D;
   b->F();
   b->G();
}
```

The output produced is

```
B::F
B::G
```

In the following example, hiding and overriding occur together:

```
ref struct B {
   virtual void F() {}
};
interface class I {
   void F();
};
ref struct D : B, I {
   virtual void F() new = I::F {}
};
```

The presence of the `new` function modifier indicates that `D::F` does not override any method `F` from its base classes. The named override then goes on to say that `D::F` actually overrides just one function, `I::F`. The net result is that `I::F` is overridden, but `B::F` is not.

end example]

Static functions can use the `new` modifier to hide an inherited member. [*Example*:

```
ref class B {
public:
      virtual void F() { … }
};
ref class D : B {
public:
      static void F() new { … }
};
```

end example]

[…]

References

C++/CLI: http://www.ecma-international.org/publications/standards/Ecma-372.htm