Remove explicit from class-head

Date: 2010-02-20
Version: N3234=11-0004
Authors: Ville Voutilainen
<ville.voutilainen@gmail.com>

Abstract

Multiple National Bodies requested in their CD ballot responses that a virtual override control attribute should be added. The attributes were since turned into context-sensitive keywords. It seems that turning the attributes into keywords brings forth problems with grammatical placement of the keyword "new" and the proposed solution is to allow "new" to appertain for functions only. That solution creates a two-fold problem: having "explicit" in the class-head sets in stone what "explicit" can check for, so it can't later be made to check the lack of "new" for non-functions without breaking existing code. This paper proposes removing "explicit" from the class-head so that the lack of "new" and "override" annotations is not diagnosed, so that the evolution path for such checking is kept open until a better solution for such checking of lack of annotations is designed.

Background

The reason for having "explicit" in the class head is to diagnose the lack of "new" and "override" annotations, protecting against accidental hiding or overriding, as follows:

```
struct B {
   virtual void f();
};

struct D explicit : B {
   void f(); // ill-formed, no override annotation
   void f(int); // ill-formed, no new annotation
};
```

This checking for functions arguably provides a useful facility. The problem with having it for functions only is that types hiding types would not be checked for hiding, thus:

```
struct X {/*...*/};
struct Y {/*...*/};

class B {
   typedef X value_type;
};

class D explicit : public B {
   typedef Y value_type; // well-formed if "new" can only appertain to functions
};
```

In this case programs may use D::value_type without any protection against mistakes, as the hiding of value_type goes undiagnosed. The core reflector message 18661 provides another illustrative example, repeated below:

```
#include <iostream>

struct B
{
  void f() { std::cout << "B::f()" << std::endl;}
};

struct D explicit : B
{
  struct foo { void operator()() { std::cout << "D::foo::operator()()"
                                           << std::endl; }};
  foo f;
};

struct B2
{
  static void f() { std::cout << "B2::f()" << std::endl; };
};

struct D2 explicit : B2
{
  struct f { f() { std::cout << "E::f::f()" << std::endl;} };
};

int main(void)
{
  B b;
  b.f();
  D d;
  d.f();
  B2::f();
  D2::f();
}
```

In this example, a data member hides a non-static member function, and a type hides a static member function. Such hiding cases are not diagnosed if "new" appertains to functions only, so there's no protection against such cases, any "explicit" annotation in the class-head wouldn't help and there would be no way to express the intent that such hiding is what was intended.

Solution and wording

The conservative solution seems to be to remove "explicit" from class-head, and reconsider such a facility in a later standard revision.

Change in 9 [class] paragraph 1:

> *class-key attribute-specifier-seq*$_{\text{opt}}$ *class-head-name* ~~*class-virt-specifier-seq*$_{\text{opt}}$~~ **final**$_{\textbf{opt}}$ *base-clause*$_{\text{opt}}$

Remove in 9 [class] paragraph 1:

> ~~*class-virt-specifier-seq*: *class-virt-specifier class-virt-specifier-seq class-virt-specifier*~~
>
> ~~*class-virt-specifier*: final explicit~~

Change in 9 [class] paragraph 1:

> ~~A *class-virt-specifier-seq* shall contain at most one of each class-virt-specifier.~~ A *class-specifier* whose *class-head* omits the *class-head-name* defines an unnamed class. [*Note:* an unnamed class thus can’t be final ~~or explicit~~. — *end note*]

Change in 9 [class] paragraph 3:

> If a class is marked ~~with the *class-virt-specifier*~~ **final** and it appears as a *base-type-specifier* in a base-clause (Clause 10), the program is ill-formed.

Remove in 10 [class.derived]:

Strike paragraph 9 completely.

> ~~In a class definition marked with the *class-virt-specifier* explicit, if a virtual member function that is neither implicitly declared nor a destructor overrides (10.3) a member function of a base class and it is not marked with the *virt-specifier* override, the program is ill-formed. Similarly, in such a class definition...~~

Change in A.8 [gram.class]:

> *class-key attribute-specifier-seq*$_{\text{opt}}$ *class-head-name* ~~*class-virt-specifier-seq*$_{\text{opt}}$~~ **final**$_{\textbf{opt}}$ *base-clause*$_{\text{opt}}$

Remove in A.8 [gram.class]:

~~*class-virt-specifier-seq*: *class-virt-specifier class-virt-specifier-seq*~~
~~*class-virt-specifier*~~

~~*class-virt-specifier*: final explicit~~