Accredited Standards Committee X3
Information Processing Systems
Operating under the procedures of
American National Standards Institute

Doc No: X3J16/95-0056 WG21/N0656
Date: March 3, 1995 Page 1 of 4
Project: Programming Language C++
Ref Doc:
Reply to: Josee Lajoie
(josee@vnet.ibm.com)

The class copy operators and volatile

Should the form of the copy constructor and assignment operator be:
1. X(const X&), operator=(const X&)
or
2. X(const volatile X&), operator=(const volatile X&)
?

Solution 1.

John Skaller has been arguing strongly in favor of 1. He states that only users should be able to declare a constructor or an assignment operator accepting a volatile reference parameter and further argues that a constructor accepting a volatile reference parameter is not a copy constructor and an assignment operator accepting a volatile reference parameter is not a copy assignment operator.

> [John Skaller in core-5214]:
>
> ```
> T volatile vt;
> T t = T(vt);
> ```
>
> this code is ill-formed unless either
>
> ```
> T(T volatile&)
> ```
> or
> ```
> T(T const volatile&)
> ```
>
> has been defined by the user or unless T is not a class type. The code
>
> ```
> T t = vt;
> ```
>
> is well defined if, and only if, one of the above constructors exists and the copy constructor for T exists:
>
> ```
> T(const T&)
> ```
>
> That is,
>
> ```
> T(T volatile const&);
> T(T volatile &);
> ```
>
> are implicit conversions (unless marked "explicit") but they are NOT copy constructors and the presence of their declaration in the class member list does not prevent the implementation from providing a copy constructor for T.

This seems to satisfy the requirement made in some earlier core messages requesting that the standard only describe in limited ways the semantics of volatile. The proposal above requires that volatile class objects be copied only by specially user written code. That is, volatile is _programmer_ defined, not implementer defined. Without this, hardware programming, threading, concurrent programming, and the like, becomes non-portable across compilers for the same platform.

Note that John is suggesting that volatile scalar objects always be copyable. Only classes cannot be copied unless the user provides a

I can see two ways not to break C compatibility:

1. Have a special rule for objects of POD class types.
   That is, an implicitly-declared copy operator for a POD class
   type is always of the form:

     X(const volatile X&), operator=(const volatile X&)

   while the one for non-POD class type is of the form:

     X(const X&), operator=(const X&)
   or
     X(X&), operator=(X&)

2. Adopt solution 2 for all classes.

Solution 2.

The copy operator may have anyone of the following forms:
  a. X(const volatile X&), operator=(const volatile X&)
  b. X(const X&), operator=(const X&)
  c. X(volatile X&), operator=(volatile X&)
  c. X(X&), operator=(X&)

If a class does not have a user-declared copy constructor (copy assignment operator), one is implicitly declared.

What is the form of an implicitly-declared copy operator?

2.1 Only one kind of implicitly-declared copy operator

  If all bases and members have copy operators that can accept a const volatile parameter, the implicitly-declared copy operators have the form:

    X(const volatile X&), operator=(const volatile X&)

  otherwise, the implicitly-declared copy operators have the form:

    X(X&), operator=(X&)

2.2. Two kinds of implicitly-declared copy operators are provided

  [Fergus Henderson, core-5301]:

    The problem with option 2.1 is efficiency. It could be very inefficient to use volatile semantics all the time for the copy operators.

    A possible solution is for the implementation to provide two implicitly-declared copy operators, one that supports volatile semantics, and one that does not:

X(const volatile X&), operator=(const volatile X&)
X(const X&), operator=(const X&)

This solution assumes that the proper copy operator is selected using the overload resolution rules.

This solution makes it a bit more tricky to describe the form of the implicitly-declared copy operators if a class has bases and members with user-declared copy operators.

What if a class T has a base with copy operators of the form:

X(const X&), operator=(const X&)

and a member with copy operators of the form:

X(volatile X&), operator=(volatile X&)

which copy-operators are implicitly declared?

T(T&), operator=(T&) ??

[Gavin Koch, core-5311]:

In cases where the following copy constructors are implicitly-declared (when the class has no user-declared copy constructors):

X::X(const X&) or X::X(X&)

the volatile version of these constructors are also implicitly declared:

X::X(volatile const X&) or X::X(volatile X&)

[something similar need to be done for assignment operators]

When these functions need to be implicitly-defined, it is an error if they can’t be (for the same kinds of reasons the "const" versions might not be implicitly-define-able).

This means that for volatile objects, volatile semantics are use, and for non-volatile objects non-volatile semantics are used. If the copy constructor is user-declared, then the user decides whether to handle volatile or not.