Document: N2600=08-0110
Date: 2008-03-26
Reply to: Alisdair Meredith <alisdair.meredith@codegear.com>

noncopyable utility class

Background

For a long time C++ developers have had to solve the problem that a class will implicitly declare special member functions for copying, whether appropriate for a given class or not.

Initially the solution was been to declare the copy constructor and copy assignment operator as private, and choose not to define them. However, this solution is generally confusing to newcomers learning the language, and easily overlooked by the experienced developer.

A popular library idiom has since evolved using a base class with private copy constructor and assignment operators. With the name of this class appearing in the base list at the top of the class definition, the (lack of) copy semantics are more visible and clearly documented. Once added to a suitable library this idiom is much easier to teach to novices. Boost has provided the `noncopyable` class for this purpose for many years.

The evolution group was sufficiently motivated to work through a series of papers focussing on the issue, culminating in paper n1717. However this direction was eventually rejected as the working group did not like the notion of adding a second kind of class to the core language when a library solution was available. Ultimately an alternative syntax was introduced to make suppressing functions look less obscure and intimidating: n2346.

New Language Facilities

The addition and interaction of several new language facilities now makes the library-based solution even more appealing. First let us propose an appropriate definition for this class:

```
struct noncopyable {
    noncopyable() = default;
    noncopyable(noncopyable const &) = delete;
    noncopyable & operator=(noncopyable const &) = delete;
};
```

Note that the default constructor is declared as default to maintain triviality. Likewise, the deleted copy constructor and copy assignment operator are also deemed trivial. In addition struct `noncopyable` is an empty *standard layout* class. This means it is also a POD. Under the revised rules for trivial types, standard layout types and PODs, the well-known "empty base optimization" is now **required** for an empty standard-layout class – see n2342 for details.

The result of these language changes is a guarantee of no per-instance space or runtime overhead associated with this solution, compared to simply declaring the members deleted in the desired class.

With the update to the implicitly declared special functions wording in the upcoming concepts paper (n2501), derived types would implicitly have deleted copy constructor/assignment operators, which should give a clearer diagnostic than simply being ill-formed on use, as per C++98/03.

C++0x or a future TR?

The class is the implementation of a simple, well-understood idiom. It is a low risk, reasonable value addition to the library that that exploits new language features. As such it would be reasonable to include it in C++0x.

This class is certainly easy enough for end-users to implement themselves. The chief reason for providing it in the standard library is that a class so widely re-invented truly should be a part of the regular toolbox, rather than any common vocabulary issue. As such, it might safely be deferred to a future TR.

The biggest irritant in deferring to TR2 is that additions to existing standard headers through TRs can be painful to implement. Therefore, the author leans if favor of adopting directly for C++0x.

Acknowledgements

Thanks to Dave Abrahams for supplying and documenting the original idiom for Boost. Beman Dawes and Lawrence Crowl's determination to clean up awkward corners of the language led directly to this paper. Francis Glassborow and Lois Goldthwaite's support for the 'incidental developer' are the direct inspiration.

Proposed Standard Wording

Add to header <utility> synopsis in **20.2 [utility]**

```
// 20.2.x support classes
class noncopyable;
```

Append a new section to 20.2

20.2.x Support Classes[utility.support]

The following classes are provided to simplify implementation of common idioms.

20.2.x.1 class noncopyable [utility.noncopyable]

```
struct noncopyable {
    noncopyable() = default;
    noncopyable(noncopyable const &) = delete;
    noncopyable & operator=(noncopyable const &) = delete;
};
```

Class noncopyable is provided so simply creation of classes that inhibit copy semantics.

[example:

```
template< typename T >
class resource_manager : private noncopyable {
public:
    resource_manager() : t( new T() ) {}
    ~resource_manager() { delete t; }
private:
    T *t;
};
```

]