Missing "undefined behavior" for const

Randy Meyers

Note: this paper is a revision of a paper distributed privately during the Nashua meeting. The issue raised herein is listed as one of the US national body issues.

The working paper is missing a statement about undefined behavior involving **`const`**. The working paper does make the following statement about **`const`** (Subclause 7.1.5.1):

> Except that any class member declared **`mutable`** (7.1.1) can be modified, any attempt to modify a **`const`** object during its lifetime (3.8) results in undefined behavior.

The purpose of the undefined behavior is to allow implementations to optimize accesses to const objects. An implementation is free to assume that the value of a const object does not change, and may take advantage of the above rule to avoid reloading the value of a const object from memory just because an assignment was made through a pointer since the const object was first loaded into a register. However, this rule only holds during the object's lifetime.

Subclause 3.8 defines the lifetime of an object as beginning after its storage is obtained and its non-trivial constructor (if it has one) has completed. Its lifetime ends when its non-trivial destructor (if it has one) begins to execute or its storage is reused or released.

When taken together, the effect of these two rules are that you are not prohibited from modifying a **`const`** object outside its lifetime. This is good and consistent with the rest of the draft since constructors and destructors are allowed to modify **`const`** objects (Subclause 12.1):

> A constructor can be invoked for a **`const`**, **`volatile`**, or **`const volatile`** object. A constructor shall not be declared **`const`**, **`volatile`**, or **`const volatile`** (9.3.2). **`const`** and **`volatile`** semantics (7.1.5.1) are not applied on an object under construction. Such semantics only come into effect once the constructor for the most derived object (1.7) ends.

Similar words are in 12.4 for destructors.

This opens a hole in the ability of implementations to optimize const objects. Consider the following program:

```
class C;
void no_opt(C *);

class C {
public:
	int c;
	C() : c(0) {no_opt(this);}
};

const C cobj;

void no_opt(C *cptr)
{
	int i = cobj.c * 100;
	cptr->c = 1;
	cout << cobj.c * 100 << '\n';
}
```

An implementation would like to assume that it can optimize the function **no_opt()** by recognizing the expression **cobj.c * 100** as a common subexpression (after all, **cobj** is a **const** object whose value cannot change). However, since **no_opt()** is called during construction, the **const** semantics and the undefined behavior rule from 7.1.5.1 do not apply. So, the implementation is obliged to honor the modification of a **const** object through a pointer to a non-**const** object. The traditional optimizations on **const** objects cannot be done unless the implementation can prove that the code in question will never be executed during construction or destruction of the **const** object.

This is not a desirable state of affairs: it is not easy to determine that a function will not be called during construction or destruction of a **const** object. Therefore, implementations will have to assume that **const** objects with constructors or destructors might be modified. C++ programs will not benefit from optimizations that programmers expect just in case a very obscure and unreasonable coding practice might be used.

The best solution is to permit access to an object during the object's construction or destruction only through the **this** pointer or through a pointer value or reference that was formed using the **this** pointer. In the example above, the program would have undefined behavior because the function **no_opt()** accesses **cobj** directly using its name while **cobj** is undergoing construction. The access to **cobj** using **cptr** is permitted since **cptr** was initialized with the value of the **this** pointer.

I am not sure what wording to use to accomplish the intent of the above paragraph.

During the Nashua meeting, several people discussed making it undefined behavior to reference a const object by its "name" while it is undergoing construction or destruction. The problem with this approach is that name is not a term in the standard, and for this purpose, name would have to mean more than a simple identifier. For example, if **A** is an array whose elements are const, referring to **A[5]** using **A[5]** while **A[5]** is undergoing construction needs to be undefined behavior. Likewise, if a class **C** has a const member **M** undergoing construction, the reference **C.M** needs to be undefined behavior. Such cases make "name" a very complex concept. The **this** pointer approach is probably more fruitful.