Document Number: **P0277R0**
Date: 2016-02-13
Audience: Evolution Working Group
Reply To: **Alan Talbot**
dpp@alantalbot.com

const Inheritance

Abstract

Proposal for **const** inheritance, a form of inheritance that provides a derived class read-only access to its base class.

The Problem

In at least one practical scenario, I have found it necessary to protect a base class from modification by its derived classes. In this scenario, the base class **t_format** models a user-specified report format, while the derived class **t_report** models a report in that format. A report object supports many operations, such as adding data to itself, displaying itself to a screen, or saving itself a file. None of its operations should change its user-specified format. This translates to a requirement that a **t_report** object should never modify is base **t_format** object. It would be useful to enforce this guarantee at compile time.

The existing **public**, **protected** and **private** forms of inheritance are of no help to protect the base object. Under any of these forms of inheritance, the derived class is free to read and modify public or protected members of its base class, or to call non-**const** methods of the base class within its own non-**const** methods.

In the above example, one might argue that the proper relationship between **t_report** and **t_format** is a has-a relationship, not an is-a relationship as implemented. This suggests the solution of changing **t_format** from a base class to a **const** member of **t_report**. There are practical issues with this solution:

- In the above example, the derived class bears an arguable has-a relationship to its base class. In other situations, the derived class may have a more obvious is-a relationship to its base class, in which case changing the base class to a **const** member of the derived class is inconsistent with the relationship. The **const** member solution is not a one-size-fits-all solution.
- Even if the class relationship is arguably has-a, changing a base class to a **const** member in an existing design can involve considerable work. Constructors must change. Base class accesses within derived classed methods must be changed to member accesses. Derived class interfaces inherited from the base class must be explicitly implemented to access the member.

The Solution

In order to protect a base class from modification by a derived class, I propose a **const** inheritance mechanism.

- **const** inheritance is specified by including the **const** keyword among the base class specifiers (**public**, **protected**, **private**, **virtual**) in the derived class declaration.
- The effect of **const** inheritance on derived classes is as follows:
  - There is no effect on visibility of base class members or methods within derived class constructors. Derived class constructors are free to construct the base class instance and modify the base class instance within the constructor body as necessary.
  - Non-**mutable** members and non-**const** methods of the base class are not visible within derived class methods other than constructors. This prevents modification of the base class instance via derived class methods post construction.
  - Non-**mutable** members and non-**const** methods of the base class are not visible in the **public** or **protected** interface of the derived class. This prevents modification of the base class instance via external calls to derived class methods inherited from the base class.
- **const** inheritance propagates recursively downward to derived classes.
- Violations of **const** inheritance result in compile-time errors.

Examples

```
// Foo can modify any of its members
class Foo
{
public:

	Foo(int value) : m_value(value), m_count(0) { }

	int get_value() const		{ return m_value; }
	int get_count() const		{ return m_count; }

	const int& value() const	{ return m_value; }
	int& value()			{ return m_value; }
	int& count() const		{ return m_count; }

	void set_value(int value)	{ m_value = value; }
	void set_count(int count) const	{ m_count = count; }

	int		m_value;
	mutable int	m_count;
};

// Bar cannot modify its Foo base class instance because of const inheritance
class Bar : public const Foo
{
public:

	Bar(int value) : Foo(value)	{ }			// Good

	int get_value1() const		{ return m_value; }	// Good
```

```
	int get_value2() const			{ return get_value(); }	// Good
	int get_value3() const			{ return value(); }	// Good

	int get_count1() const			{ return m_count; }	// Good
	int get_count2() const			{ return get_count(); }	// Good
	int get_count3() const			{ return value(); }	// Good

	void set_value1(int value)		{ m_value = value; }	// Bad
	void set_value2(int value)		{ set_value(value); }	// Bad
	void set_value3(int value)		{ Foo::value() = value; }	// Bad

	void set_count1(int count) const	{ m_count = count; }	// Good
	void set_count2(int count) const	{ set_count(count); }	// Good
	void set_count3(int count) const	{ Foo::count() = count; }	// Good
};

// Awk cannot modify its Foo base class instance, because Bar cannot
class Awk : public Bar
{
public:
	Awk(int value = 0) : Bar(value)	{ }
};

// Joe can modify its Foo base class instance
class Joe : public virtual Foo
{
public:
	Joe(int value = 0) : Foo(value)	{ }
};

// Jim cannot modify its Foo base class instance, because of const inheritance
class Jim : public virtual const Foo
{
public:
	Jim(int value = 0) : Foo(value)	{ }
};

// Jon cannot modify its Foo base class instance, because Jim cannot
class Jon : public Joe, public Jim
{
public:
	Jon(int value = 0) : Foo(value)	{ }
};

int main()
{
	// Foo can modify any of its members
	Foo foo(123);			// Good
	int foo_value = foo.get_value();	// Good
	int foo_count = foo.get_count();	// Good
	foo_value = foo.value();		// Good
	foo_count = foo.count();		// Good
	foo.value() = foo_value;		// Good
	foo.count() = foo_count;		// Good
	foo.set_value(foo_value);		// Good
	foo.set_count(foo_count);		// Good
	foo.m_value = foo_value;		// Good
	foo.m_count = foo_count;		// Good
```

```
        // Bar cannot modify its Foo base class instance
        Bar bar(123);                       // Good
        int bar_value = bar.get_value();    // Good
        int bar_count = bar.get_count();    // Good
        bar_value = bar.value();            // Good
        bar_count = bar.count();            // Good
        bar.value() = bar_value;            // Bad
        bar.count() = bar_count;            // Good
        bar.set_value(bar_value);           // Bad
        bar.set_count(bar_count);           // Good
        bar.m_value = bar_value;            // Bad
        bar.m_count = bar_count;            // Good

        // Awk acts like Bar
        // Joe acts like Foo
        // Jim acts like Bar
        // Jon acts like Bar
}
```