Document Number: SC22/WG21/N1615=04-0055
Date: 2004-04-09
Author: Lois Goldthwaite
Email: Lois@LoisGoldthwaite.com

C++ Properties -- a Library Solution

N1600 is a summary of the "property" language extension that is proposed for C++/CLI. I can't help feeling that this is an effort to impose an alien idiom on C++. There is too much of "the compiler will perform black magic behind your back" in it. There are too many ways in which the programmer must be aware that some [pseudo-]data members must be handled in different ways from [real] data members.

The C++/CLI draft spec (N1608)says in 8.8.4 and 18.4 that "A property is a member that behaves as if it were a field." I think "behaves" is absolutely the wrong word to use here. A property is behavior that pretends to be state. From an OO design point of view, I think this is A Bad Idea. Behaviour should be visible; state should not.

Further on, the document continues, "the X and Y properties can be read and written as though they were fields. In the example above, the properties are used to implement data hiding within the class itself." The subtle advantage of "hiding" a private data member by treating it as a public data member escapes me.

Properties are just syntactic saccharine for getter and setter member functions for individual fields. C++ experts have spent years trying to educate people NOT to use public data, to use member functions instead, and now we propose to introduce something that looks just like public data? Do we really want to try to teach people that public fields are still bad, but properties are good, even though they look just the same from outside the class? Moreover, these public fields have side effects!

There is a danger that too many novice programmers will prototype their designs using public fields, with the intent of changing them to properties later, if and when it proves necessary, and the general quality of code will suffer because of it. Of course, in the quest for good design, replacing a public data member with simple get/set functions for that data member does not achieve a large increase in abstraction. A function-call syntax which does not overtly refer to manipulating data members may lead to a cleaner interface. It allows the possibility that some "data members" may be computed at runtime rather than stored. An overwhelming majority of books on OO design recommend thinking in terms of objects' behaviour, while hiding their state. Let us encourage good habits, not bad ones.

One UK panel member with almost a decade's daily experience using the Borland VCL properties, most of that via their C++ binding, had these comments:

> Gut reaction is: 'They are nice, but they are not C++ and I think I'm happier that way.'
>
> Properties work fantastically well in RAD development where you want interactive tools beyond a simple source code editor. For all they appear glorified get/set syntax, they make the life of the component writer much simpler. There are no strange coding

conventions to follow so that things magically work, and a lot of boilerplate code vanishes. From this perspective they are most definitely A Good Thing™.

Of course the property concept goes way beyond simply supporting GUI tools, and that is where the slippery slope begins...

If functional programming is 'programming without side effects', property oriented programming is the other extreme. Everything relies on side effects that occur as you update state. For example: you have a 'Left' property in your GUI editor to determine position of a control. How would you move this control at runtime? Traditionally we might write some kind of Move() function, but now we can set the 'Left' property instead and that will somehow magically move the control on the form as a side-effect, and maybe trigger other events with callbacks into user code as a consequence.

Experience shows that people prefer to update the Left property rather than look for some other function. After all, that is how you would perform the same task at design/compile time. Generally, code becomes manipulating properties (and expecting the side effects) rather than making explicit function calls. Again, this is the 'minimum interface' principle so that there is one and only one simple way of achieving a given result. Typically, the Move function() is never written, and this reinforces the programming style.

As we move beyond GUI-tools arena, I find the property syntax becomes more and more confusing. I can no longer know by simple inspection of a function implementation if I can take the address of everything used as a variable, or even pass them by reference. This only gets worse when templates enter the mix.

Then there is the danger of other developers becoming 'clever'. I still have a hard time persuading our team leader that 'Write only properties' are a bad idea [values you set to achieve the side effect, but can never query]. I guess we all know there is no protecting people from themselves if they are determined to do harm (e.g. dereferencing a null pointer to initialise a reference) but there is no reason to hand them a loaded gun either.

Properties have made our GUI-production up to an order of magnitude simpler. Less code, faster to produce, intuitive to design for, and generally this all leads to fewer bugs. In context they are a great tool. [And there is no reason to assume GUI is the only valid context. It is simply the only one to work well for us]

Outside the GUI code properties always made our code more obscure and difficult to manage. It is much harder for someone new to the code to pick it up and work out what is going on (or even be aware that there IS something going on). I would not like to see them generally supported outside compatibility layers.

The one-sentence summary of his experience is this: **"Properties made our code easier to write, but immensely more difficult to maintain."**

Other members with experience using properties in Delphi consider them "very elegant and usable," but add that the C++ interfaces may be rather less elegant. Perhaps there is just a cognitive mismatch with the general mindset of the language. All things considered, the integration with RAD tools was applauded -- it makes writing components much simpler and easier and increases programmer productivity because boilerplate code is automatically generated. If this can be achieved in line with traditional C++ paradigms it is all the more to be applauded.

Properties fall into a kind of twilight zone as a C++ extension -- you can't take their address, although you assign to them with operator = you can't chain assignments to properties, you can't use combined operators like += with them, they can't be passed by non-const reference, and they fail with bizarre errors when used for template deduction (I hope I am correct in saying these things -- as properties don't exist in standard C++, and I have little or no experience with Borland C++ Builder, I am repeating comments from others here).

My impression is that the main benefit of properties is not their syntactic sleight-of-hand but (1) their ability to be discovered through introspection/reflection and manipulated non-programmatically by some sort of RAD tool, and (2) their ability to be stored (I think "pickled" is the term used with Java Beans) in this configured state, so that they can be loaded at runtime, already configured. However, discussions during the TG5 meeting the week before the Sydney meeting of WG21 indicated that support for such a RAD tool was not a motivating factor in adding this feature to C++/CLI.

Because RAD tools rely on metadata for much of their functionality, CIL metadata distinguishes between fields, methods, properties, and events. CLI reflection distinguishes between fields and properties (there are distinct methods Type.GetFields and Type.GetProperties). More specifically, when working with the System.ComponentModel namespace it is possible to manipulate how types display properties at runtime, and even provide proxies for properties. This is not true for fields. In addition an accessor can be declared virtual, sealed, or abstract, and it can override non-accessor functions.

Properties are a common feature of GUI frameworks and RAD environments, but they are almost always combined with some form of reflection, introspection, and serialisation. See, for example, John Wiegley's paper "PME: Properties, Methods, and Events" at http://anubis.dkuug.dk/jtc1/sc22/wg21/docs/papers/2002/n1384.pdf which takes for granted that these features come as a package. These are very important topics in their own right, but are orthogonal to the issue of whether C++ needs special magic syntax to support properties.

Perhaps the most persuasive argument in favour of properties, at least in the CLI environment, is interoperability with other languages such as C# and VB, and with the system-level classes which expose properties (for example, System.Array has properties such as Length, Rank, IsReadOnly).

It is a foregone conclusion that TG5's deliverable will include support for properties as a feature of the core language. But I want to explore how far a portable C++ library implementation can go in providing the perceived advantages of properties, without requiring any change in the compiler. If this approach proves fruitful, it could lead to a coding style usable in both dialects of C++. And perhaps compiler vendors can find a way to generate the necessary metadata for CLI from these template classes, without taking the larger step of adding properties to standard C++ as a language feature.

The hastily-scribbled code at the end of this paper defines some utility template classes which may be used as class members. Briefly, these are the names of the suggested classes:

```
// a read-write property with data store and
// automatically generated get/set functions.
// this is what C++/CLI calls a trivial scalar property
template <class T>
class Property;
```

```
// a read-only property calling a user-defined getter
template <
               class T,
               class Object,
               typename T (Object::*real_getter)()
         >
class ROProperty;

// a write-only property calling a user-defined setter
template <
           class T,
           class Object,
           typename T (Object::*real_setter)( T const & )
         >
class WOProperty;

// a read-write property which invokes user-defined functions
template <
          class T,
          class Object,
          typename T (Object::*real_getter)(),
          typename T (Object::*real_setter)( T const & )
         >
class RWProperty;

// a read-write named property with indexed access to own data store
template <
              class Key,
              class T,
              class Compare = less<Key>,
              class Allocator = allocator<pair<const Key, T> >
         >
class IndexedProperty;
```

This technique, described in its basic form in C++ Report back in 1995, seems at least as convenient, and produces as economical source code, as most uses of properties. The basic assignment and return functions can be inlined, and therefore should be efficient. In order to support chaining, the setter functions return their new value, but for complete compatibility with C++/CLI they should have a void return type.

For programmer convenience these classes offer three redundant syntaxes for accessing their data members (or apparent data members -- properties need not have real storage behind them):

- function call syntax
- get/set functions
- operator = (T) and operator T() for assignment to and from properties

The read-only and write-only property classes implement only the accessors appropriate to their semantics, but for compatibility with C++/CLI they do reserve (unimplemented) the unnecessary get or set identifier.

The utility templates as outlined in this paper do have an address, they have a type for template deduction, and they can be used in a chain of assignments. They can also be used with today's compilers. If someone brings forward a RAD tool to read and manipulate properties through

metadata, then a "property" modifier or some other marker for the tool could be added to their declaration.

One objection to these that has been raised is that CLI properties allegedly take up no space inside a class, whereas a C++ subobject cannot have a size of 0 (ignoring certain clever optimizations). On the other hand, I expect that most useful properties will need some way to preserve state between set and get, and that has to take up space somewhere. The size of Property<T> takes up as much space as a single object of its template parameter, while the other three hold only a single pointer to an implementation object. Note that the Object and member function template parameters do not have to refer to the containing object, though that is likely to be the most common usage. They could delegate the processing to an external or nested helper class. The choice of implementation object (though not its type or member functions) can even be changed dynamically, and several objects could share a single implementation object.

Apart from the goal of supporting a common coding style for portable C++ and C++/CLI, another possible use for the Property<T> class template would be in migrating legacy code away from using public data members and towards better encapsulation. The class definition could be changed to include members of the Property<T> type, but client code need not be rewritten, only recompiled.

The biggest inconvenience to these classes that I perceive is that all but the simplest Property<T> instances need to be initialized at runtime with the address of their implementation object (usually the containing class). A smaller inconvenience is that using them on the right hand side of an assignment invokes a user-defined conversion, which could affect overload resolution and a sequence of conversions.

One of the features of C++/CLI properties is that they can be declared as virtual or pure virtual, for polymorphic overriding by derived classes. This obviously is not possible with data member declarations. But if the implementation object (which usually means the containing object) is itself polymorphic and the designated member functions are virtual, then the property will behave in a polymorphic fashion.

The C++/CLI feature of default (nameless) indexed properties can be simulated with a member operator [] (except that in C++/CLI indexes can take multiple arguments, but there is a separate proposal for C++ to allow comma-separated arguments inside square brackets).

In summary, tool support for RAD is very desirable. But it is an issue orthogonal to features of the language. These class templates may help code written for the CLI environment to be more portable to non-CLI ones, and they may have other utility in migrating code to a style using encapsulation instead of public data members.

This paper incorporates some remarks from Alisdair Meredith, James Slaughter, James Dennett, Ian Cooper, John Washington, and Alan Lenton, but they have not reviewed it. I am responsible for the code and the stated opinions.

```
// Some utility templates for emulating properties --
// preferring a library solution to a new language feature

// Each property has three sets of redundant acccessors:
// 1. function call syntax
// 2. get() and set() functions
// 3. overloaded operator =


// a read-write property with data store and
// automatically generated get/set functions.
// this is what C++/CLI calls a trivial scalar property
template <class T>
class Property
{
    T data;
public:
    // access with function call syntax
    Property() : data() { }
    T operator()() const
    {
       return data;
    }
    T operator()( T const & value)
    {
       data = value;
       return data;
    }

    // access with get()/set() syntax
    T get() const
    {
       return data;
    }
    T set( T const & value )
    {
       data = value;
       return data;
    }

    // access with '=' sign
    // in an industrial-strength library,
    // specializations for appropriate types might choose to
    // add combined operators like +=, etc.
    operator T() const
    {
        return data;
    }
    T operator = ( T const & value )
    {
        data = value;
        return data;
    }

    typedef T value_type;  // might be useful for template deductions
};
```

```
// a read-only property calling a user-defined getter
template <class T, class Object, typename T (Object::*real_getter)()>
class ROProperty
{
    Object * my_object;
 public:
    // this function must be called by the containing class, normally in a
    // constructor, to initialize the ROProperty so it knows where its
    // real implementation code can be found. obj is usually the containing
    // class, but need not be; it could be a special implementation object.
    void operator () ( Object * obj )
    {
       my_object = obj;
    }

    // function call syntax
    T operator()() const
    {
        return (my_object->*real_getter)();
    }

    // get/set syntax
    T get() const
    {
        return (my_object->*real_getter)();
    }
    void set( T const & value ); // reserved but not implemented, per C++/CLI

    // use on rhs of '='
    operator T() const
    {
        return (my_object->*real_getter)();
    }

    typedef T value_type;  // might be useful for template deductions
};
```

```
// a write-only property calling a user-defined setter
template <
          class T,
          class Object,
          typename T (Object::*real_setter)( T const & )
         >
class WOProperty
{
    Object * my_object;
 public:
    // this function must be called by the containing class, normally in
    // a constructor, to initialize the WOProperty so it knows where its
    // real implementation code can be found
    void operator () ( Object * obj )
    {
       my_object = obj;
    }

    // function call syntax
    T operator()( T const & value )
    {
        return (my_object->*real_setter)( value );
    }

    // get/set syntax
    T get() const;      // name reserved but not implemented per C++/CLI
    T set( T const & value )
    {
        return (my_object->*real_setter)( value );
    }

    // access with '=' sign
    T operator = ( T const & value )
    {
        return (my_object->*real_setter)( value );
    }

    typedef T value_type;  // might be useful for template deductions
};
```

```
// a read-write property which invokes user-defined functions
template <
          class T,
          class Object,
          typename T (Object::*real_getter)(),
          typename T (Object::*real_setter)( T const & )
         >
class RWProperty
{
    Object * my_object;
 public:
    // this function must be called by the containing class, normally in a
    // constructor, to initialize the ROProperty so it knows where its
    // real implementation code can be found
    void operator () ( Object * obj )
    {
       my_object = obj;
    }

    // function call syntax
    T operator()() const
    {
        return (my_object->*real_getter)();
    }
    T operator()( T const & value )
    {
        return (my_object->*real_setter)( value );
    }

    // get/set syntax
    T get() const
    {
        return (my_object->*real_getter)();
    }
    T set( T const & value )
    {
        return (my_object->*real_setter)( value );
    }

    // access with '=' sign
    operator T() const
    {
        return (my_object->*real_getter)();
    }

    T operator = ( T const & value )
    {
        return (my_object->*real_setter)( value );
    }

    typedef T value_type;  // might be useful for template deductions
};


// a read/write property providing indexed access.
// this class simply encapsulates a std::map and changes its interface
// to functions consistent with the other property<> classes.
// note that the interface combines certain limitations of std::map with
// some others from indexed properties as I understand them.
// an example of the first is that operator[] on a map will insert a
```

```
// key/value pair if it isn't already there. A consequence of this is that
// it can't be a const member function (and therefore you cannot access
// a const map using operator [].)
// an example of the second is that indexed properties do not appear
// to have any facility for erasing key/value pairs from the container.
// C++/CLI properties can have multi-dimensional indexes: prop[2,3]. This is
// not allowed by the current rules of standard C++

#include <map>

#ifndef USING_OLD_COMPILER
template <class Key, class T, class Compare = less<Key>,
        class Allocator = allocator<pair<const Key, T> > >
#else
template <class Key, class T>
#endif
class IndexedProperty
{
#ifndef USING_OLD_COMPILER
    std::map< Key, T, Compare, Allocator > data;
    typedef std::map< Key, T, Compare, Allocator >::iterator map_iterator;
#else
    std::map< Key, T > data;
    typedef std::map< Key, T >::iterator map_iterator;
#endif

  public:
    // function call syntax
    T operator()( Key const & key )
    {
        std::pair< map_iterator, bool > result;
        result =  data.insert(std::make_pair(key, T() ) );
        return (*result.first).second;
    }
    T operator()( Key const & key, T const & t )
    {
        std::pair< map_iterator, bool > result;
        result =  data.insert( std::make_pair( key, t ) );
        return (*result.first).second;
    }

    // get/set syntax
    T get_Item( Key const & key )
    {
        std::pair< map_iterator, bool > result;
        result =  data.insert(std::make_pair(key, T() ) );
        return (*result.first).second;
    }
    T set_Item( Key const & key, T const & t )
    {
        std::pair< map_iterator, bool > result;
        result =  data.insert( std::make_pair( key, t ) );
        return (*result.first).second;
    }

    // operator [] syntax
    T& operator[]( Key const & key )
    {
        return (*((data.insert(make_pair(key, T()))).first)).second;
    }
};
```

```
// ========================================================================
// and this shows how Properties are accessed:

#include <string>
#include <iostream>

class myClass {
  private:
    Property<std::string> secretkey_;

    // ----------- user-defined implementation functions ------------------
    // in order to use these as parameters, the compiler needs to see them
    // before they are used as template arguments. It is possible to get rid
    // of this order dependency by writing the templates with slight
    // differences, but then the program must initialize them with the function
    // addresses at run time. ask me for details if you prefer this approach.

    // myKids is the real get function supporting NumberOfChildren property
    int myKids()
    {
        return 42;
    }

    // addWeight is the real set function supporting WeightedValue property
    float addWeight( float const & value )
    {
        std::cout << "WO function myClass::addWeight called with value "
                  << value
                  << std::endl;
        return value;
    }

    // setSecretkey and getSecretkey support the Secretkey property
    std::string setSecretkey( const std::string& key )
    {
        // extra processing steps here
        return secretkey_( key );
    }

    std::string getSecretkey()
    {
        // extra processing steps here
        return secretkey_();
    }

 public:
    // Name and ID are read-write properties with automatic data store
    Property<std::string> Name ;
    Property<long> ID  ;

    // Number_of_children is a read-only property
    ROProperty< int, myClass, &myClass::myKids > NumberOfChildren;

    // WeightedValue is a write-only property
    WOProperty< float, myClass, &myClass::addWeight > WeightedValue;
```

```
    // Secretkey is a read-write property calling user-defined functions
    RWProperty< std::string,
                myClass,
                &myClass::getSecretkey,
                &myClass::setSecretkey
              > Secretkey;

    IndexedProperty< std::string, std::string > Assignments;

    // constructor for this myClass object must notify member properties
    // what object they belong to
    myClass()
    {
        NumberOfChildren( this );
        WeightedValue( this );
        Secretkey( this );
    }
};

int main()
{
    myClass thing;

    // Property<> members:
    thing.Name( "Pinkie Platypus" );
    std::string s1 = thing.Name();
    std::cout << "Name = " << s1 << std::endl;

    thing.Name.set( "Kuddly Koala" );
    s1 = thing.Name.get();
    std::cout << "Name = " << s1 << std::endl;

    thing.Name = "Willie Wombat";
    s1 = thing.Name;
    std::cout << "Name = " << s1 << std::endl;
    std::cout << std::endl;

    thing.ID( 12345678 );
    long id = thing.ID();
    std::cout << "ID = " << id << std::endl;

    thing.ID.set( 9999 );
    id = thing.ID.get();
    std::cout << "ID = " << id << std::endl;

    thing.ID = 42;
    id = thing.ID;
    std::cout << "ID = " << id << std::endl;
    std::cout << std::endl;

    // ROProperty<> member
    int brats = thing.NumberOfChildren();
    std::cout << "Children = " << brats << std::endl;

    brats = thing.NumberOfChildren.get();
    std::cout << "Children = " << brats << std::endl;

    brats = thing.NumberOfChildren;
    std::cout << "Children = " << brats << std::endl;
    std::cout << std::endl;
```

```
    // WOProperty<> member
    thing.WeightedValue( 2.71828 );

    thing.WeightedValue.set( 3.14159 );

    thing.WeightedValue = 1.618034;
    std::cout << std::endl;

    // RWProperty<> member
    thing.Secretkey( "******" );
    std::string key = thing.Secretkey();
    std::cout << "Secretkey = " << key << std::endl;

    thing.Secretkey.set( "!!!!!!" );
    key = thing.Secretkey.get();
    std::cout << "Secretkey = " << key << std::endl;

    thing.Secretkey = "?????";
    key = thing.Secretkey;
    std::cout << "Secretkey = " << key << std::endl;
    std::cout << std::endl;

    // IndexedProperty<> member
    thing.Assignments( "Convenor", "Herb");
    std::string job = thing.Assignments( "Convenor" );
    std::cout << "Convenor = " << job << std::endl;

    thing.Assignments.set_Item( "Minutes", "Daveed");
    job = thing.Assignments.get_Item( "Minutes" );
    std::cout << "Minutes = " << job << std::endl;

    thing.Assignments[ "Coffee" ] = "Francis";
    job = thing.Assignments[ "Coffee" ];
    std::cout << "Coffee = " << job << std::endl;
    std::cout << std::endl;

    return 0;
}
```

```
/* output expected :
Name = Pinkie Platypus
Name = Kuddly Koala
Name = Willie Wombat

ID = 12345678
ID = 9999
ID = 42

Children = 42
Children = 42
Children = 42

WO function myClass::addWeight called with value 2.71828
WO function myClass::addWeight called with value 3.14159
WO function myClass::addWeight called with value 1.61803

Secretkey = *****
Secretkey = !!!!!
Secretkey = ?????

Convenor = Herb
Minutes = Daveed
Coffee = Francis
*/
```