Accredited Standards Committee
X3, INFORMATION PROCESSING SYSTEMS*

Doc No: X3J16/92-0048
WG21/N0125
Date: May, 1992
Project: 738-D (PL C++)
Reply To: Oren Ben-Kiki
oren@wisdom.weizmann.ac.il

Overloading the '.' Operator and Overloading Lvalues/Rvalues

Oren Ben-Kiki
P.O.B. 14050, Tel-Aviv, Israel
Tel. 927-3-5614293

1 Description

The suggestion consists of two extensions to the language which are closely related, although each may be considered on its own.

1.1 Overloading the '.' operator

Section 13.4 would be modified to include the '.' operator in the list of overloadable operators. When overloaded, the function-id of the '.' operator would contain the identifier of the "pseudo member" defined immediately after the '.'. Thus the operator-function-id would be "operator . <identifier>".

If declared inside a class definition, the operator would take no arguments. Otherwise, it would take a single argument, which will be a class object or a reference to one.

Pseudo members share the same name space as data members and member functions of the class.

For example:

```
class X {
public:
        int     operator . member1 () { ... }
};
```

---

* *Operating under the procedures of the American National Standards Institute (ANSI) Standards Secretariat: CBEMA, 1250 Eye St. NW, Suite 200, Washington, DC 20005*

```
int     operator . member2 ( X& x ) { ... }

class Y : public X {
        int     operator . member1 () { ... }       // Overrides X's
};

class Z : public X {
        int     member1;                // Error; member1 ambiguous
        int     member2();              // Error; member2 ambiguous
        double operator . member1 ();   // Error; different return type
};

X       x;
Y       y;
int     a = x.member1;                  // Calls X::operator.member1
int     b = y.member1;                  // Calls Y::operator.member1
int     c = x.member2;                  // Calls operator.member2(x)
int     d = y.member2;                  // Calls operator.member2((X)y)
```

1.2 Overloading lvalues/rvalues

Section 13.4 would be further modified to allow specifying different behavior for lvalue and rvalue usage of the '.' and '[]' operators. The function-id of the rvalue operator would remain unchanged, as would its behavior, i.e. the function-id would remain "operator []" or "operator . <identifier>", it would take the same number of arguments, etc.

The function-id of the lvalue operator would be the function-id of the rvalue operator followed by a modification operator, i.e. the function-id would be "operator [] <mod-op-name>" or "operator . <identifier> <mod-op-name>". Valid <mod-op-name>s are:

```
=   +=   -=   *=   /=   %=   ^=   &=   |=   <<=   >>=   ++   --
```

When declared inside a class definition, the lvalue operator will take the arguments of the rvalue operator followed by the arguments of the modification operator. When declared outside of a class definition, the argument list will be preceded by a single argument, which will be a class object or a reference to one.

If a modification operator is used on the pseudo member/element, and a matching lvalue operator exists, the lvalue operator is invoked. Otherwise, the rvalue operator is invoked. Thus, in order to prevent any undesirable modification operators from being applied to a pseudo member/element, the return value of the rvalue operator should not be a modifiable lvalue.

For example:

```
class X {
public:
        int     operator [] ( int i ) { ... }
        int     operator [] = ( int i, int v ) { ... }
        int     operator [] = ( int i, double v ) { ... }
        int&    operator . member0 () { ... }
        int     operator . member1 () { ... }
        int     operator . member1 += ( int i) { ... }
        int&    operator . member3() { ... }
        int     operator . member3 ++ () { ... }
};

X       x;
x[2] = 3;                   // Calls X::operator[]=(int,int)
x[3] = 3.14;                // Calls X::operator[]=(int,double)
int     a = x.member0;      // Calls X::operator.member0
x.member0 = 7;              // Calls X::operator.member0
x.member1 = 2;              // Error: X::operator.member1 is not
                            // a modifiable lvalue
x.member1 += 2;             // Calls X::operator.member1 +=
x.member3++;                // Calls X::operator.member3++
x.member3 = 7;              // Calls X::operator.member3
```

2 Motivation

2.1 Overloading the '.' operator

Every conceptual object has a large number of attributes associated with it. Some of these attributes can be computed from others; thus, the set of attributes that is sufficient to completely determine the object's state is not unique and a change in one attribute causes changes in others.

Any implementation of an object separates the attributes into two groups, "stored" and "computed". An attribute is stored if it is defined as a class data member; otherwise, it is computable from the stored ones per access.

Stored attributes are further divided to "exposed" and "hidden". An attribute is exposed if it is public or protected, i.e. it is possible to manipulate the attribute from outside the class by directly manipulating the class data member associated with it.

Why should a stored attribute be hidden? here are a few compelling reasons:

(a) Some operations on the attribute are not allowed. For example, the attribute may be read-only, or may only be added to, etc.

(b) The protection for some operations is not the same; some are private, some are protected and some are public. For example, the attribute may be read-only for the public, may only be added to by derived classes, and may be freely modified privately or by friends.

(c) Not all values of the attribute are valid; operations that attempt to set the value to an illegal one should be prevented. For example, the attribute is a number that may be either 0 or an odd number.

(d) The attribute is not completely independent of all other stored attributes; sometimes it can be completely derived from some other stored attributes, but is stored for speeding up access. Changing it should therefore effect other attributes as well, or be simply prevented. For example, a vector might cache its norm and even allow scaling through manipulation of it.

(e) The class being defined is intended to be used as a base class, and any of points (a) - (d) may be raised in some conceivable derived class. For example, a 'Bag' abstract base class may be implemented by a derived linked-list implementation, which does not cache its size; a smarter one that would; or an array that would allow manipulating it in some ways, such as increasing it.

Given that there are many reasons to hide an attribute, what are the reasons for exposing it? It has in fact been suggested that exposed data members are a bad idea. Smalltalk, as an example, does not support this notion at all. However, there is a strong reason to use such members: Currently, C++ provides the following interface to exposed attributes:

```
value = object.member;
object.member = value;
object.member += value;
...
```

While hidden and computed attributes must be accessed through a function call interface:

```
value = object.get_member();
object.set_member(value);
object.add_to_member(value);
// or even worse:
object.set_member(object.get_member() + value);
...
```

It is obvious that there is a great incentive to expose attributes, in order to avoid the cumbersome function call interface.

However, exposing any attribute exposes fundamental design choices, these that deal with the internal representation of the class. This contradicts good OOP practice and prevents the full use of OOP power.

For example, consider the creation of a derived class which introduces a new stored attribute which is not independent of some attribute(s) exposed in the base class. Obviously, accessing an object of the new class through a pointer to the base class will cause inconsistencies. The programmer is forced to hide the previously exposed attribute(s) in the base class and provide virtual access routines for them. This results in a change of interface - all users of the base class and any derived class thereof have to change their code.

Naturally, the programmer may always give all attributes a virtual function call interface, insuring that derived classes would not require changes to it. Since this interface is cumbersome, programmers resort to this technique for abstract base classes only; however, the scenario described above is not limited to abstract base classes. In fact, it is when designing concrete classes that less thought is given for all possible derived classes; hence the set of exposed attributes is chosen according to points (a) - (d) only.

The first suggested extension (overloading the '.' operator) would solve these problems in a clean and elegant way through the introduction of pseudo-members.

2.2 Overloading lvalues/rvalues

Allowing the use of operators instead of functions where it makes sense is an important feature of C++. The ability to replace code such as:

```
add_objects(object1, object2, object3);
```

With:

```
object1 = object2 + object3;
```

Has allowed the programmers to write code that is much more readable and thus easier to maintain. It is natural to expect that code such as:

```
object.set_attribute(new_value);
object.increase_element(i, by_value);
```

Will be similarly simplified to:

```
object.attribute = new_value;
object[i] += by_value;
```

For exactly the same reasons. Such code is very common, as a change in one attribute often effects others. Array elements, as a specific type of attribute, are not exempt from such dependence. For example, a samples array might maintain its average and standard deviation, a matrix might cache its determinant and rank, etc.

It is also natural to expect that the simplified code would provide the same power that is currently available through the use of member functions. In particular, it is sometimes desirable to provide more modification operators for a pseudo member/element in a derived class then are available in the base class, or to change the visibility of such operators through private or protected derivation.

And most important, simplifying the class interface code should not be done at the expense of excessive additional complexity to the internal class code.

The second suggested extension (overloading the lvalues/rvalues) would satisfy all of the above in a clean and elegant way through the explicit overloading of lvalue behavior.

3. Considerations

3.1 Backward Compatibility

The proposed extension would not break any existing code. Any code not using the extension would continue to compile and run as usual. However, if the extensions are accepted, current code may look clumsy when compared to the code made possible by them. This will be a strong incentive to modify old code.

3.2 Programming Styles

The extension promotes good OOP practices such as encapsulation and using abstract base classes, as well as the writing of a more readable code. While the definition of "good" style using the extension may be different than the current one, the extension is fully backward compatible and does not preclude any programming style currently available.

3.3 Expressive power

By allowing the programmer to control the application of operators to class members, the extension makes it possible to express complex operations using a much more natural syntax then currently available. For example, the following code:

```
shape1.nw = shape2.se - point(1, 1);
shape3.center = point(3, 7);
shape3.area += 3;
...
```

Would reposition 'shape1' such that its north-west 'corner' would be at a position 1 unit below and 1 unit to the left of the south-east 'corner' of 'shape2', and move 'shape3' such that its 'center' would be at the point (3, 7). It then scales shape3 such that its total area increases by 3 units. All these attributes may be computed, e.g. if shape1 and shape2 are circles and shape3 is a rectangle.

3.4 Compilation

While the extension does have an effect on compiler implementation, it is a minor one. Currently, the compiler keeps a list of data members and member functions for each class. It would now be required to also keep a list of 'pseudo' members and their associated assignment operators. When parsing a member expression, the compiler would determine whether it is a 'pseudo' member; if it is, it will look at the operator applied on it and decide whether it is overloaded in the class. Compilation time should, if anything, be reduced (when compared to parsing a new class definition per attribute, and constructing/destructing the accessor objects).

3.5 Run-time

As the pseudo members are implemented via normal member functions, which can be inline or virtual as usual, there is no effect on the run-time support required or the efficiency of the code.

3.6 A simpler alternative

The first extension may be provided by simply not requiring the '()' when invoking a member function without arguments:

```
class X {
public:
        int     attr();
};

X       x;
int     a = x.attr();           // OK
int     b = x.attr;             // Also OK
```

However, omitting the '()' would be allowed for member functions only, otherwise the following code would break:

```
extern int      f();
int     (*a)();

a = f;          // 'f' is evaluated unexpectedly
```

Thus this alternative, while somewhat simpler to implement then overloading the '.' operator, introduces inconsistencies to the language syntax (sometimes the '()' are required, sometimes not). When combined with the second extension (overloading lvalues/rvalues) it introduces another inconsistency:

```
class X {
public:
        int     operator[](int index);
        int     operator[]=(int index, int value);
        int     attr();
        int     attr=(int value);       // Actually defining an operator=;
                                        // Inconsistent with normal operator
                                        // overloading syntax
};
```

3.7 Using accessors instead of lvalue/rvalue overloading

It is possible, through the use of accessors, to use a more natural interface to non-exposed attributes, at the cost of additional source code complexity. This technique was originally suggested for overloading the '[]' operator, but is trivially extended to pseudo members:

```
value = object.member();
object.member() = value;
object.member() += value;
...
```

When combined with the first suggested extension (overloading the '.' operator) or with the simpler alternative suggested above, this yields an interface which is identical to the interface to exposed members, solving the problems described in 2.1.

However, doing so would cause a new problems to appear, as described in 2.2:

(1) Since it is impossible to change the return type of a member function in a derived class (except for to a derived class of the original return value), it is very difficult to control the behavior of accessors under such complications as private/protected inheritance, multiple inheritance, etc. At best it means that a parallel inheritance tree has to be maintained. At worse, some constructs would be impossible, or the accessors inheritance tree would be only similar to the class inheritance tree, complicating the code to no end.

(2) Further, while operator '[]' requires a single accessor class, pseudo members require an accessor class per attribute. Each operator has to be specified in both the original class and in the accessor class.

Taken together, using accessors causes such an increase in coding effort that many programmers do not bother with it even for operator '[]'. Using templates and pre-processor is not much help in reducing this coding effort.

3.8 Providing pseudo members using current language tools

It is possible, in principle, to (almost) provide pseudo members using current language constructs. In fact, I have done so and used it in one system. (My code was slightly different from the one given below; however, the idea is the same). The idea was to define an unnamed union of all pseudo members (at the cost of 4 wasted bytes). Each member of the union was an object of an accessor class. The code depended on a kludge for obtaining the 'this' pointer to the real object from the 'this' pointer of the accessor:

```
#define     MdfyThis(C) ((C *)((char *)this - int(&((C *)0)->_marker)))
#define     CnstThis(C) ((const C *)((const char *)this - int(&((C *)0)->_marker)))

class V;    // Any type

// Define a class with a pseudo member 'attr', which is read-only
// publicly and modifiable privately.

class C {
public:
        V get_attr() const;
private:
        V set_attr(const V& v);
        V add_attr(const V& v);
public:
        friend class ATTR;
        union {
                char   _marker;
                class ATTR {
                friend class C:
                public:
                        operator V() const
                                { return(CnstThis(C)->get_attr()); }
                private:
                        V operator=(cont V& v)
                                { return(MdfyThis(C)->set_attr(v)); }
                        V operator+=(cont V& v)
                                { return(MdfyThis(C)->add_attr(v));}
                } attr;
        };
};
```

```
};
```

This mechanism suffers from some severe faults:

(1) It costs 4 bytes (for any number of pseudo members) per object. If a derived class wants to add more members, it has to pay another 4 bytes.

(2) It suffers from all the problems of using accessors, as described above.

(3) Using the offset to switch between the accessor's 'this' pointer and the real object's 'this' pointer is abominable.

3.9 Completeness

The extension is not quite "complete". Consider the case where an attribute's value is not an atomic type, e.g., it is a point with 'x' and 'y' coordinates. Assuming that the programmer provided:

```
class A {
public:
        Point& operator.location() { ...; return(Point(x, y)); }
        Point& operator.location=(Point& p) { ...; return(p); }
        ...
};
```

It would be valid to write:

```
        A a;
        a.location.x = 7;
```

However, the location of 'a' would not change, as the '.x' would be performed for a temporary 'Point' object. This is another reason why the rvalue operator should not return a modifiable lvalue. If the 'const' keyword was used in the above code, as follows:

```
class A {
public:
        const Point& operator.location() const { ... return(Point(x, y)); }
        const Point& operator.location=(const Point& p) { ...; return(p); }
        ...
};
```

Then the compiler would flag both:

```
        a.location.x = 7;
```

And:

```
(a.location = Point(3, 7)).x = 7;
```

As errors.

It is possible, in principle, for the compiler to translate:

```
a.location.x = 7;
```

To:

```
tmp p = a.location; p.x = 7; a.location = p;
```

Which is the expected behavior. However, that would require an extensive change to the compiler. It would also make the semantics of the code that much further from the visible syntax; even if the compiler could do it without cost it would not be a good idea. Note that a similar option existed for the '++' and '--' operators: one could require the compiler to convert:

```
b = a++ * 7;
```

To:

```
b = a * 7; a.operator++();
```

I assume this alternative was rejected for similar reasons.

3.10 Consistency with operator '->'

The suggested extension is very different from the currently provided overloading of the '->' operator. This may be viewed as an inconsistency.

However, the '->' operator and the '.' operator are NOT analogous; the '->' operator, in its original form, is equivalent to a dereferencing step followed by a member access step. The '.' operator provides a member access step only.

Currently, the programmer can control the dereferencing step by overloading the '->' operator. This is almost equivalent to controlling the unary '*' operator. Thus:

```
a->b
```

Gets converted to:

```
(a.operator->()).b
```

Which is functionally the same as:

```
(*a).b
```

Therefore, operator '->' only provides control on half the process only. The term "smart pointers" given to using the '->' operator emphasizes the point.

Overloading the '.' operator would allow the programmer to control the second step - the member access. Thus, the proposed suggestion is not inconsistent with operator '->'; rather, it is a natural complementary extension.

4. Experience

There are similar features in various languages; for example, in Turbo Pascal, there is no syntactic difference between accessing a data member of a class and invoking a member function with no arguments (this is similar to the simpler alternative described above). However, Turbo Pascal does not allow, to the best of my knowledge, the definition of modifiable computed attributes.

I have implemented pseudo-members in some classes I used for a 10K lines project in C++; however, I did not use them extensively as the compiler (g++ version 1.39) was able to inline the functions only through use of a slightly more convoluted code then the one depicted above. Once defined, however, using the pseudo members proved natural and elegant.

5. Summary

This paper proposes the extension of C++ with the overloading of the '.' operator and explicitly overloading lvalues/rvalues. The advantages are clearer source code, better encapsulation and stronger expressive power. The implementation cost is minor and there are no consequences for existing code.

Oren Ben-Kiki
oren@wisdom.weizmann.ac.il