WG21/N0831
X3J16/96-0013

Operator->* Proposal

David Dodgson
dsd@tr.unisys.com
UNISYS

In Monterey we accepted a change to iterator that required the -> operator be supported. This allows iterators to reference members directly. During deliberations of that change the library subgroup discussing iterators was asked to consider the ->* operator. This became an open issue for Clause 24. The changes required to implement the -> operator are relatively straightforward. They follow directly from an application of the `operator->` function. The changes needed for the ->* operator are not so clear cut. The difference is in the differing treatments of the overloading operator functions, `operator->` and `operator->*`. For the purposes of overloading, `operator->` in the expression `x->m` is treated as a unary operator. In the expression `x->*m`, `operator->*` is treated as a binary operator. The effect of this difference is shown below.

Consider the following program:

```
class X {
public:
   int  i;
   bool b;
};
X *p = new X;
int  X::* pmi = &X::i;
bool X::* pmb = &X::b;

p ->  i   = 3;       // fine
p ->* pmb = true;    // also fine
```

Now let's attempt a simple smart pointer approach to get debugging information.:

```
X * X::operator->() {
    cout << "X Ref";
    return this;
 };

 p ->  i   = 3;       // Says "X Ref"
 p ->* pmb = true;    // Says nothing
```

If we want to do the same thing for ->* we can't. `operator->*` is strictly a binary op. We would have to use:

```
 int X::operator->*( int X::* p1 ) {
    cout << "X Ref";
    return (*this).*p1;
 };
// and similarly for bool, etc.
 bool X::operator->*( bool X::* p1 ) {
    cout << "X Ref";
    return (*this).*p1;
 };
```

We can reduce the effort somewhat by using a template:

```
template< class T >
   T X::operator->*( T X::* p ) {
      cout << "X Ref";   return (*this).*p; }
```

Unfortunately, this does not work well for pointer-to-member functions.

This mechanism seems to be overly complex for what we would hope to be a relatively simple smart pointer or iterator class. Because the function behind the ->* operation is so similar to what is needed for the -> operation it makes sense to compare the two. The ->* operation could be treated similarly to the -> operation. Examining clause 13.5.6 (Class member access) we see that x->m is defined as `(x.operator->())->m`. If we redefine the x->*m operation to be `(x.operator->*())->*m`, the use of this operation would be much simpler.

Here is an example from clause 20 with the proposed change:

```
template<class X> class auto_ptr {
    public:
    // _lib.auto.ptr.cons_ construct/copy/destroy:
      explicit auto_ptr(X* p =0);
      template<class Y> auto_ptr(auto_ptr<Y>&);
      template<class Y> auto_ptr& operator=(auto_ptr<Y>&);
     ~auto_ptr();
    // _lib.auto.ptr.members_ members:
      X& operator*() const;
      X* operator->() const;
      X* operator->*() const;     // *** new member function ***
      X* get() const;
      X* release();
      void reset(X p =0);
    };
template<class X> X* operator->*() const {
    return operator->(); };
```

Most instances of `operator->*` in smart pointers or iterators with this proposed change would be to return `operator->()`. This is obviously much simpler than trying to define a member template to handle this operation. It is also potentially very useful. It is quite likely that a smart pointer class could be used in conjuntion with pointers to members.

The disadvantage to this proposal is that it removes the binary `operator->*` from overloading considerations. There may be current code which uses this feature. However, I believe that this usage is probably minimal and the advantage of using the redefined operator in smart pointers and iterators is substantial.

Proposed Changes

Add a new section after 13.5.6 “Class member access” for “Pointer to member operators”.

> `operator->` shall be a non-static member function taking no parameters.
> It implements pointer to member access using ->*
>
> *pm-expression ->* cast-expression*
>
> An expression `x->*m` is interpreted as `(x.operator->*())->*m` for a class object `x` of type `T` if `T::operator->*()` exists and if the operator is selected as the best match function by the overload resolution mechanism

Remove section 13.6 paragraph 12 discussing binary `operator->*`. [Note: `operator->` is not discussed in 13.6]

References

This issue is discussed in messages c++std-ext-3468 through 3471