X3J16 / 93-0137
WG21 / N0344

09/24/93
John Bruns
NationsBanc-CRT

Expressions of Class Type

A previous paper by Tom Plum and Scott Turner (N0293 / 93-0086) discussed the lvalue-ness of temporaries. The original paper noted a number of consequences of deciding that these expressions are lvalues, const lvalues, or non-lvalues (assuming the C model). The only real decision made was that they were not completely lvalues. The result was a vote by the committee that these constucts were not "lvalues" but in some sense refered to objects. (N0306 / 93-0099).

Since then much discussion has been seen on the core reflector. It is evident that there are two schools of thougth. Some want such objects to be first class objects in all sense of the word, and behave much more like lvalues. I will refer to this school of thought as like lvalues or LLV. Others want these objects to behave much more like C built in types and be totally value oriented. This school I will refer to as the abstract data types, or ADT. I took some info from the net and some in conversations from Kent Budge, but this is a synthesis of how I think the ADT camp wants things. Furthermore, I will shamelessly steal from the original paper in format and analysis and call such constructor or function expressions Cl expr.

```
class C {
public:   C();
          int i;
          const int c;
          void mf();
};

extern C f();                     // returns C
class Derived : public C {};      // derived from C

class S {
public:   int i;
          C c;                    // member of type C
};

extern S sf();
```

1. Result of function-call/explicit-conversion is not a an lvalue.

```
&f()      //  error both
&C()      //  error both
```

AGREED This is the essence of the decision that these object are not lvalues.

2. Non-lvalueness preserved when selecting a member.

```
&(sf().i) // error both
```

AGREED Neither camp wants this behavior

3. Non-const member function may be called.

```
f().mf()        // error ADT, OK LLV
C().mf()        // error ADT, OK LLV
```

DIFFER Here we have a disagreement. LLV sees these as real object, ADT as rvalues only. Both ADT and LLV support binding these values to constant member functions.

4. Initialization by [Cl expr] does not use copy constructor.

```
C c1 = C();     // Both - interpret as C c1();
C c2 = f();     //  copy constructor must be accessible , can be optimized away
return C();     //  default constructor used.
{ C c; return c; } // copy constructor must be accessible, can be optimized away
```

AGREED Here we both agree, but there is some area of controversy that compared to how things work now that I will go into later.

5. Initialization of reference converts [Cl expr] to referenced type.

```
const C &cref = Derived();
```

AGREED // both refer to the C in temporary of type Derived().

6. Overloading

```
f()       //  ADT value of type const C,  LLV - value of type C
C()       //  ditto.
```

DIFFER In the ADT case these expresion could be bound to a const C & argument or used directly (copy constructor optimized away) by a argument of type C. It would not overload to an argument of type C &. For LLV these would bind to C&, const C &, directly but would require a copy to supply a argument of type C.

7. Binding to a non-const reference allowed.

```
C &cref = f(); // error ADT, OK LLV.
C &cref = C();    // error ADT, OK LLV.
C &cref = Derived();  // error ADT, OK LLV entire Derived
```

DIFFER Again the difference is again the LLV wants these items to bind to non-const references and ADT wants only constant references.

The State of the Art

How do current compilers handle the current constructs. I do not have an exhaustive set, I tested these constructs on Borland 3.1, Cfront 2.1, Cfront 3.0 and GNU 2.2 (NeXTStep release, I know the current GNU compiler does better, but not what it does).

Case #3 All bind to mf() without any problems or warnings.

Case #6 All overloaded to a type C (ie choses C& not const C&) but all but GNU complained that it was an anachronism. Cfront 3.1

Case #7 All allowed the construct, all but GNU complained about it.

The conclusion I reach is that the support leans to the LLV school of thought. In particular, that is the way most compilers work today. However, there a twist. Although all compilers allow the return value of a function to be cv-quailfied, Cfront gives some meaning to this. In particular, const and non-const value are used to select overloaded member functions.

Extending overloading of class values using cv-qualified returns

```
class C {
public:
          C();
          int i;
          void mf();
          void mf() const;
          void mf2();
          void mf3() const;

};

extern C f();                      // returns C
extern const C g();           // returns a const C
```

Consider items 3,6, and 7 in this context.

3. Member function may be called.

```
f().mf()       // OK calls non-const function  (current behavior - all)
g().mf()       // OK calls to const function    (Cfront behavior)
f().mf2() // OK (current behavior)
```

```
g().mf2() // ERROR non-const member function T::mf2() called for const object
```

Now g() behaves like ADT wants it to, and f() behaves like LLV wants it to

6. Overloading

```
f()        //  would match equivalent to "C &" type (currently does but
           //   with error message)
g()        //  would match "const C &" type only
```

Once again g() would behave like ADT wants, and f() behaves like LLV wants.

7. Binding to a reference allowed.

```
C &cref = f();           // OK  f() is a non-const value
const C &cref = f();     // OK
C &cref = g();           // ERROR g() is const
const C &cref = g();     // OK
```

Same as 6.

Proposed Solution

Adopt the strategy described in the above examples

First, declare that CV qualifiers on return types are meaningful. The CV qualifier takes effect after the value has been created (from the functions return expression). This is similar to the effect of CV qualifiers for function arguments. They affect the value after it has been copied or constructed.

Fix overloading for member functions. Define the bahavior to work similar to how Cfront does now. That is: A temporary of class type const T binds only to const member functions. A termporary of type T binds to either const or non-const member functions but prefers non-const.

For function argument overloading, use the same selection mechanism. Non-qualified returns of class type would prefer binding to non const references. Const qualified returns would only bind to const references. Note that matching for value type arguments are dependent only on the constructors accessible (see section below).

The effect of these changes are that the objects returned by functions returning class objects and constructor expressions, are in all ways lvalues except fot the explicit taking of address (built in &). Examples have shown this is already the case, since any function that binds to a class type object can take an address.

Note on compiler generated temporaries.

```
class T { ... };
```

```
class S { ... S(const T &); };
T f();

g( T &);
h( S &):

g(f());         // ok
h(f());         // error  compiler generated temp is always const
h(S(f()));      // ok
```

The solution proposed above would allow the g(f()) call without any diagnostics, since the value returned is non-const. In the case of h(f()), the compiler (as opposed to the programmer) inserted the temporary. This is almost always an error and should be given a error message. The explict convresion is legal.

THE BEAUTY of this proposal is how it simplifies the overloading system.

```
C f();
const C g();
C & fr();
const C & gr();
```

In all cases, f() and fr() would overload to the same function, g() and gr() would likewise overload to the same function. This would happen for member selection and for other argument selection. This should be easy to teach and understand.

This solution also **breaks NO code.** Only people who declared the return types for their functions to be const , are not using Cfront, and are binding these to non-const functions. It seems they will get a belated warning.

Const (value only) Classes

Another solution suggested is to delcare all non-lvalue objects as inherently const and apply the above rules. Unfortunately, this has two negatives, it would break lots of existing code. Dag Bruck has mentioned that that is the case at his site. It is also the case at mine.

And second it loses information. Consider the following code, mentioned explicitly by Keith Gorlen's, Sanford Orlow"s and Perry Plexico's book "Data Abstraction and Object_Oriend Programming in C++" on page 68:

```
SubString String::operator() (unsigned pos, unsigned lgt);
const SubString String::operator() ( unsigned pos, unsigned lgt) const;
```

The idea here is that the constness of the String reflects in the constness of the SubString.
This allows the compiler to keep track of this information in a simple way. If all such SubStrings are "const" (or non-const for that matter). The class designer must either design two classes, or keep track

internally. All at higher programming cost.

If you reject that all returns of class type are const , the ADT people would like to have a way to declare that **all temporaries** of a class have a constant type.

```
class C const {     // all returns of type C are const C
     };

class D { //  returns are const or not as indicated in declaration
     };
```

It has been argued that knowing all values of a class are const could lead to compiler optimizations. This extension would tell the compiler (and the reader) that this is the case. It prevents someone externally subverting the issue by creating a function.

```
C copy (const C & x) { return x; };
```

with the above definition. copy() will still return a const.

I don't believe this extension is absolutely necessary, **BUT** it is simple and doesn't break existing code. Considering the passion that the supporters of ADT have put into making **ALL** temporaries const, it seems easily justified.

PS. We could explain builtin types behave as "const" classes.
PPS. If you hate overloading const - lets pick another word :-)

Constructors, Returns and Function Arguments

I mentioned these other issues. They are not as major, but they came up in the discussions.

```
class X {
     public:  X(int);
     private: X(const X &);
};

X x1(1);       // ARM -  call X(int)  - NEW - same
X x2 = 2;      // ARM  call X(int) then use copy constructor - NEW - call X(int)
X x3 = X(1);   // ARM  call X(int) then use copy constructor - NEW - call X(int)
```

Note that in the above cases, making the copy constructor private is not effective. Someone getting the odd error message will simply switch to the first form. Almost every compiler gets rid of the copy anyway, so lets declare that these are all the same thind

```
C c = f();     //  copy constructor must be accessible , can be optimized away
```

Now hiding the copy constructor could be useful, the class designer must not want a copy made and kept. If he hid all the constuctors, he certainly doesn't . Allowing the compiler to perfom the build without an accessible copy constructor subverts the intent of the designer.
A good compiler, however, could optimize it away.

Consider return expressions

```
C f() {return C(); };    //  default constructor used.
X  fx() { return 1; }; //  X(int)
{ C c; return c; } // copy constructor must be accessible, can be optimized away
```

I recommend that for functions returning class objects, the argument of the return expression be considered a constuctor expression. This means that the copy constructor is only required it the return expression is of that class type (and not an explicitly called constructor). The constructor called must be accessible.

```
X f1() {return X(1); };  //  OK X(int)
X f2() { return 1; };         //  OK X(int)
X f3() { X x(1); return x; }; // error - copy constructor must be accessible
```

Note This makes the rules for creating class type **return** the same as the rules for class type **argument** generation and **explicit initialization**. In all three cases:

1) Only a constructor can build an object of class type. Each of these expressions much search for a constructor that matches their arguments. If the constuctor matching is a copy constructor, and the source object is no longer needed, the compiler can optimize away the copy.

2) The constructor used must be accessible in the context it is used. For function arguments and return types, this is the function context. For explicit initialization, it is the context the statment appears in.

3) Cv-qualifiers take effect after the constructor has completed. Cv-qualifiers on the final type have no effect in selecting the constructor being called. Note that the constructors may affect the allowable type of the constructor argument, in particular, it is possible to define a copy constructor that takes only non-const references.

Loose Ends

CONSTRUTORS There is still no way to designate const returns from constructors expressions. If the "class X const" proposal is accepted, this would be included under the blanket of all class temporaries. If not, another suggestion was to allow designating const on individual constructors.

```
class S {
     public:  const S(int);   // return value constand
               S(const S &);       //  normal return value
};
```

I don't see this extension as anywhere near as useful and do not propose it.

VOLATILE types. I'm sorry but I don't feel comfortable about volatile. In any event, the recommendations present a very consistent framework for const, non-const. Perhaps it will help when fitting in volatile. In any event a consistent framework would seem better than an inconsistent one.

CONCLUSION

I recommend the following resolutions of core issues, these are seperable proposals:

1) CV-Qualified return types: Allow them. Returns of class type handled symetrically with reference type returns for purpose of member selection, function overloading and explicit binding to references. Note that these are similar to lvalues, but not exactly see #1,#2, #4, and #5 in first section. See Proposed Solution.

2) We add the minor extension of "const" classes. All value type temps of "const" classes are always "const". See Const (value only) Classes.

3) Unify the handling of constructors, arguments, and return values. Constructors expressions are simplified to mean the same thing regardless of the syntax chosen. Copy constructors need not be accessible if the named constructor is. Return statements of functions returning class values are handled the same as constructor expressions. The copy constructor must be accessible if the initializer or return expression is of class type (or class reference). Such copy constructors can be optimized away. See Constructors, Returns and Function Arguments.