Document: P3271r0
Date: May 20, 2024
Reply-to: Lisa Lippincott <lisa.e.lippincott@gmail.com>
Audience: SG21 (Contracts)

Function Usage Types (Contracts for Function Pointers)

Lisa Lippincott

Abstract

This paper introduces *function usage types,* novel function types which have associated contracts. Function usage types are named types that are not the type of any individual function. Instead, they are used as the basis of function pointer and reference types, and describe the generalized expectations indirect callers have of the functions pointed or referred to.

1 Introduction: a tale of function usage

As an introduction to function usage types, let's consider a scenario in which an existing use of function pointers is improved by adding contracts.

1.1 The background

A function `fancy_calculation` is parameterized on an operation, which it will invoke. It uses the operation in a way that expects these properties:

1. A single argument of type `int` will be passed to the operation.
2. A result of type `int` is expected from the operation.
3. The argument passed will be nonnegative; the operation may rely upon this.
4. The result must be nonnegative, and no greater than the argument passed.

Some existing functions, such as `identity` and `halve`, are suited to this usage:

```
int identity( const int x )    { return x;   }
int halve   ( const int x )    { return x/2; }
```

Some other functions with the same type, such as `twice`, are not suited to this usage:

```
int twice   ( const int x )    { return 2*x; }
```

Users of `fancy_calculation` nominate various operations to be used, indicating them with function pointers and references:

```
void print_fancy_results( const int s )
  {
   const std::array ops = { &identity,    // good choice
                            &halve,       // good choice
                            &twice    };  // bad choice!

   std::cout << s << ":";
   for ( auto op: ops )
     std::cout << " " << fancy_calculation( *op, s );
   std::cout << std::endl;
  }
```

These operations are passed to `fancy_calculation`, which invokes them:

```
int fancy_calculation( int(&op)(int), const int start )
  {
   int current = start;
   while( true )
     {
      int next = op( current );
      if ( current == next )
         break;
      current = next;
     }
   return current;
  }
```

Helpfully, the authors of `identity`, `halve`, and `twice` have provided assertions governing the use of their functions:

```
int identity( const int x )                          post( r: r == x   );
int halve   ( const int x )                          post( r: r == x/2 );
int twice   ( const int x ) pre( can_multiply(2,x) ) post( r: r == 2*x );
```

1.2 The problem

We would like to provide assertions governing the use of `fancy_calculation`. We can begin with some pre- and postconditions:

```
int fancy_calculation( int(&op)(int), const int start )
   pre( start >= 0 )
   post( r: r >= 0 )
   post( r: r <= start )
   post( r: r == op(r) );
```

But these assertions don't specify the restrictions on the function `fancy_calculation` calls indirectly. The closest we can come without some new mechanism is to add assertions to the body of `fancy_calculation`, surrounding each invocation of the operation:

```
int fancy_calculation( int(&op)(int), int start )
  {
   int current = start;
   while( true )
     {
      contract_assert( current >= 0 );     // our responsibility
      int next = op( current );
      contract_assert( next >= 0 );        // Not our responsibility: these
      contract_assert( next <= current ); //    fail if op was poorly chosen.

      if ( current == next )
         break;
      current = next;
     }
   return current;
  }
```

Putting these assertions in the function body is an imperfect solution. It becomes tedious and error-prone when there are many indirect function calls. It's invasive: the surrounding code may need to be restructured to make values available to these assertions. And finally, while these assertions are intended to inform the choice of function, they are hidden from the code making that choice.

1.3 A mitigation of the problem

To some extent, the problems listed above can be mitigated by introducing an object that wraps the function pointer:

```
class fancy_wrapper
   {
    private:
       int(*function)(int);

    public:
       explicit fancy_wrapper( int(&f)(int) )
         : function(&f)
         {}

       int operator()( const int x ) const
         pre( x >= 0 )
         post( r: r >= 0 )
         post( r: r <= x )
         {
          return (*function)(x);
         }
   };
```

A wrapper like this has some drawbacks. It introduces a shim function `operator()` that may incur some overhead, as parameters must be moved or copied before invoking the pointed-to function. The pre- and postconditions also no longer apply to the parameters or result of the pointed-to function; they instead apply to those of `operator()`. And finally, while the wrapper definition — and thus the contract it contains — may be made available to the caller of `fancy_calculation`, the contained contract is overwhelmed by the volume of boilerplate text surrounding it. The code above is also considerably shortened by providing a mixture of pointer-like and reference-like semantics; the boilerplate increases if we try to provide separate pointer-like and reference-like types.

Nevertheless, this wrapper points to a solution. We will introduce new contract-bearing function types that may be referred to by either pointers or references, but eliminate the unnecessary overhead, both syntactic and at run-time.

1.4 The plan

The solution proposed here is in several steps:

1. Gather the conditions imposed on the operation by fancy_calculation together under a name.

   ```
   int fancy_op( const int x ) usage
      pre( x >= 0 )
      post( r: r >= 0 )
      post( r: r <= x );
   ```

   This defines a *function usage type,* which is a new kind of function type. As with an abstract class type, there are no functions of this type. Instead, this type describes the manner in which a function of type `int(int)` will be used by a particular sort of indirect caller.

2. At the point where a particular function is chosen for use with `fancy_calculation`, use syntax that expressly indicates that the chosen function is intended to suit the `fancy_op` usage.

   ```
   void print_fancy_results( int start )
     {
      const std::array ops = { &fancy_op{identity},      // good choice
                               &fancy_op{halve},         // good choice
                               &fancy_op{twice}     };   // bad choice!
      // ...
     }
   ```

   The explicit use of `fancy_op` here designates a function as suitable for a particular usage. The expression `fancy_op{identity}` is an lvalue of type `fancy_op` referring to the `identity` function.

   At this point in compilation, both `fancy_calculation`'s usage contract and the chosen function's contract will be available to tools attempting to prove compatibility across all inputs. But we do not propose here to prove, or even require, compatibility-in-general; we only propose to check compatibility-as-used, in the same manner as other aspects of the contracts proposal.

3. Use the type system to create a chain of custody from the point where the function is chosen to the point where it is called, expressing at each step that the function is intended to suit the usage.

   ```
   int fancy_calculation( fancy_op& op, int start );
     // halve or twice cannot be bound to the fancy_op& parameter,
     // but fancy_op{halve} or fancy_op{twice} can be bound to it.
   ```

   Pointers and references to `fancy_op` are distinct from pointers and references to `int(int)`, and also distinct from pointers and references to other function usage types, even if those other usage types have identical pre- and postconditions. Programs evolve, and usage scenarios that have similar contracts today may have dissimilar contracts tomorrow.

   In the anticipated implementation, pointers and references to `fancy_op` have the same execution-time representations as pointers and references to `int(int)`. The distinction in type is purely a compile- and link-time matter.

4. When the indirect call is made, check compliance with the conditions in `fancy_op` as they apply to the particular invocation. These conditions do not replace the pre- and postconditions of the function referred to; they are additional conditions imposed by the caller.

   ```
   int fancy_calculation( fancy_op& op, int start )
     {
      // ...
         int next = op( current );
           // Executes, in this order:
           //    the preconditions of fancy_op
           //    the preconditions of the function referred to
           //    the body of the function referred to
           //    the postconditions of the function referred to
           //    the postconditions of fancy_op
      // ...
     }
   ```

   The poor decision to use `twice` can be detected here if a particular invocation violates a precondition of `twice` or a postcondition of `fancy_op`. (A precondition of `fancy_op` is the responsibility of the caller; a postcondition of the called function is the called function's responsibility.)

In this example, when `start` is zero, `twice` will comply with the usage conditions of `fancy_op`. When `start` is positive but sufficiently small, a postcondition of `fancy_op` will be violated. Very large values of `start` will violate a precondition of `twice`.

1.5 A look back

The key idea here is that the contract on a function pointer should express the needs of a caller, rather than just being a contract shared between several functions. A third party — often at a higher level than either the caller or callee — is typically responsible for choosing functions that meet the needs of the caller.

Once contracts are in heavy use, I expect most function contracts to be unique within a program. If two different functions may be called under identical circumstances and make identical guarantees about their results, how would one choose between them? Situations will arise, for example during refactoring or testing, where two functions will share a contract. Or functions may differ in ways, such as complexity, that we can't yet express in contracts. But I expect these situations to be rare.

Some people have suggested that we should change the type of a function pointer `&f` when `f` has a nonempty contract. To me, that seems to be a very disruptive change for very little benefit. If two functions almost never have the same contract, what would we gain? Contracts on function pointers need to be open to wide range of implementations, while contracts on functions may allow little variance. Conversely, contracts on function pointers may be very specific to a particular caller, while contracts on functions are open to a wide range of callers. The two should be specified separately.

2 Hazards to avoid

There are a number of approaches to using contracts in conjunction with function pointers. But to produce a workable solution, there are also a number of hazards we must avoid. Many approaches founder on the rocks of one of three hazards: allowing contracts to alter the types of functions, incorporating contract details into the type system, or requiring a function to explicitly opt in to a contract for indirect use. The present proposal is designed to avoid these hazards.

2.1 Contracts must not alter the types of functions

If upgrading to a new version of C++ changes the types of functions in a program, it creates a compatibility nightmare: libraries compiled with one version of the language can become incompatible with the next. Therefore, at the very least, the types of functions that have no contract annotations must remain the same.

But what of functions with contract annotations? If adding a contract annotation to a formerly-empty-contract function changes the type of the function, adding contracts to a program becomes a fraught process with wide-ranging repercussions. Empty-contract functions would also become distinguishable in the type system, going against our slogan "concepts don't see contracts."

To avoid this hazard, the type of a function must be unchanged both by the introduction of contracts into the language, and by the introduction of a contract to the function itself.

The present proposal does not alter the types of functions in any way.

2.2 Contract details must not become part of the type system

Some approaches rely on a form of duck-typing, wherein type equality is determined by a textual comparison of contracts. In doing so, they make a canonicalized expression of the contract part of a function pointer's type. Such an approach injects the canonicalized contracts into mangled linkage names, creating a strong disincentive to modifying either an individual contract or the contract system as a whole. In addition, any alteration of a contract that alters its canonicalized expression becomes visible to the type system, going against the slogan "concepts don't see contracts."

A similar hazard applies when the rules governing type conversions take the details of contracts into account. The type system, through SFINAE and concepts, is sensitive to the well-formedness of conversions. Allowing a change in contract details to affect conversions goes against "concepts don't see contracts."

While the present proposal does introduce new types, type equality is determined by name. The use of named types insulates the type system from the contents of contracts. Alteration of a contract affects neither type equality nor the well-formedness of conversions.

2.3 Function pointer contracts should not be invasive

Some approaches, modeled on the relationship between base and derived classes, require a function's definition to opt in to a function pointer type's contract. This invasiveness creates a pressure to continually modify otherwise-stable functions to opt in to newly-invented function pointer contacts. Conversely, it creates a disincentive to the introduction of contracts on function pointers, as many functions may need to be modified or wrapped to opt in to the new contract.

This problem may be mitigated by the use of shim functions that opt in to a contract, but encouraging the use of such shim functions goes against our zero-overhead principle.

The present proposal avoids invasiveness by recognizing three roles: a function that expresses its implementation contract, a caller that expresses its usage contract, and a third party that designates the function as usable by the caller. The designation is a type conversion that may be implemented without a change in representation, thereby imposing no overhead.

3 The details

3.1 The function usage types

Function usage types are novel function types, distinct from the function types in C++23. Function usage types have names with linkage, unlike the C++23 function types. For this reason, we can refer to the C++23 function types as *anonymous function types.*

A function usage type is a definable item [basic.def.odr].

Alternative syntax: It has been pointed out that the syntax shown here for defining of a function usage type looks very like the declaration of a function. To a large extent, I think this is desirable: both constructs must express a name, parameter list, result type, exception specification, and contract. To introduce a radically different syntax including these same items would be a disservice to the user.

On the other hand, there is much to be said for introducing new features with a keyword. Function usage types aren't so common that they need a super compact syntax. We could introduce function usage types with a rather long keyword, making them more visibly distinct from functions:

```
function_usage fancy_op( int x ) -> int
   pre( x >= 0 )
   post( r: r >= 0 )
   post( r: r <= x );
```

Or, taking it a step further:

```
function_usage fancy_op( int x ) -> int r
   pre( x >= 0 )
   post( r >= 0 )
   post( r <= x );
```

3.2 Function usage types may be incomplete

It is useful to declare a function usage type without defining it, leaving the type incomplete. As with class types, this allows pointers and references be passed through contexts that don't have or need the details of the type — specifically, contexts that neither bind the usage type to a function nor call a function through the usage type.

Ideally, forward declaration of a function usage type should reveal only enough information to make pointers to the type complete, (which may simply be the size of the pointers). But the syntax for C++ function types works against this goal, pushing us to specify a return type and parameter list. Trying to navigate these shoals, I propose this syntax that looks like a usage definition, but avoids all details:

```
auto fancy_op(...) usage;
```

In a function usage declaration that is not a definition, the result type is `auto` and there is no trailing return type. The presence of a return type distinguishes function usage definitions from mere declarations. In a declaration that is not a definition, the parameter list must be `(...)` and no pre- or postconditions may be specified. Such a declaration indicates only that `fancy_op` is a function usage type, revealing no further details.

As usual, pointers and references to incomplete function usage types are complete types. They may therefore be used in function and object declarations without completing the function usage type.

Alternative syntax: A real keyword makes things easier, even if it is as long as "`function_usage`":

```
function_usage fancy_op;
```

Open Question: Must a declaration of an incomplete usage type specify the language linkage of the function? Specifically, may language linkage affect the size or alignment of pointers or references to functions?

3.3 Function usage types may be templated

To express the needs of function or class templates, we need function usage type templates:

```
template < std::integral I >
I fancy_integral_op( const I x ) usage
   pre( x >= I{} )
   post( r: r >=  I{} )
   post( r: r <= x );

template < std::integral I >
I fancy_integral_calculation( fancy_op<I>& op, I start );
```

Function usage type templates may be explicitly specialized. The result and parameter types of the specialization must exactly match the template declaration (array-to-pointer or const-removal adjustments are not performed.) The pre- and postconditions may differ.

```
template <>
char fancy_integral_op< char >( const char x ) usage   // OK
   pre( x >= '0' )
   pre( x <= '9' )
   post( r: r >= '0' )
   post( r: r <= x );

template <>
int fancy_integral_op< short >( const int x ) usage;      // ill-formed
```

We can also allow partial specialization of these types, subject to a rule: a program is ill-formed if it instantiates a function usage type template specialization whose parameter types or result type do not match the base template.

3.4 Attaching usage to a function

The syntax used in this paper for attaching a usage type to a function, *e.g.* `fancy_op{halve}`, is a functional-notation explicit type conversion with a braced initializer list, resulting in direct initialization.[1] It is a syntax we use in other contexts to indicate a *nice* conversion, and I want to encourage that connotation here.

Niceness in this context involves an intention that the contract of the function is compatible with the usage contract: perhaps compatible for all use, and at least compatible for the intended use. To allow tools

[1]Earlier versions of this paper proposed dedicated punctuation, such as **`fancy_op:->halve`**. That seemed only to encourage bikeshedding, and drew attention away from the mechanics of the proposal.

to help enforce this compatibility, both contracts should be made available locally. Therefore, for a function usage type `u` and a an expression `f` of function type, I propose that the expression `u{f}` is only well-formed when

- The usage type `u` is complete, and
- The expression `f` either has complete function usage type or is a constant expression.

Both overload resolution and template argument deduction may be applied to the function expression, and template argument deduction may also be applied to the function usage type. These should be applied as if in an analogous function call expression. In the simplest case, the usage type `u1` is not a template specialization, and we imagine an analogous attachment function `attach_u1`:

```
usage_result u1( usage_func_params ) usage;

u1& attach_u1( usage_result (&f)( usage_func_params ) );
```

In this case, the expression `u1{f}` performs template argument deduction and overload resolution in the same way as a call `attach_u1(f)`.

For a template `u2`, the attachment function is similarly templated:

```
template < usage_tmp_params >
usage_result u2( usage_func_params ) usage;

template < usage_tmp_params >
u2< usage_tmp_params >&
attach_u2( usage_result (&f)( usage_func_params ) );
```

Explicit template arguments present on either subexpression are applied to the analogous function call: the expression `u2<usage_args> :-> f<func_args>` performs template argument deduction and overload resolution like `attach_u2<usage_args>(f<func_args>)`.

TODO: See if this analogy can be rewritten in terms of class template argument deduction; it would probably be more clear.

3.4.1 Other non-cast explicit conversions to usage types

Because the syntax is so similar, I propose that we restrict function style implicit conversion using parentheses, *e.g.,* `fancy_op(halve)` in the same way as `fancy_op{halve}`. I'm less concerned with the C-style cast `(fancy_op)halve`; it can go ahead and do a static or reinterpret cast.

3.5 Null pointer conversions

A null pointer constant is convertible to a pointer to a function usage type; no change to the wording of the standard seems to be necessary to allow this.

3.6 Implicit function usage conversions

Every function usage type is based on an anonymous function type, its *anonymized type,* which is determined as follows:

- The parameter types of the anonymized type are arrived at by applying the rules of [dcl.fct]¶5 to the parameter types of the function usage type (arrays become pointers, top-level const is removed).
- The result type of the anonymized type is the result type of the function usage type.
- The anonymized type is noexcept if the function usage type is noexcept.

For uniformity, the anonymized type of an anonymous function type is itself. We should provide a type trait that anonymizes a function type.

I propose to treat pointers and references to anonymous function types as indicating functions with unknown usage contracts. This treatment suggests that a pointer to a function usage type may be implicitly converted to a pointer to the anonymized type of the usage type. This should be a standard conversion [conv.general] sequenced before the existing function pointer conversion, so that in addition, an implicit conversion sequence may drop `noexcept`.

There are no implicit conversions between function usage types, or from anonymous function types to function usage types.

3.7 Static cast

A static cast may reverse an implicit conversion sequence that contains a function usage conversion. Thus a static cast may produce a pointer to a function usage type from a pointer to the corresponding anonymous function type. Because an incomplete function usage type does not reveal its anonymized type, such a cast is only well-formed when the usage type is complete. (Dependency on completeness is also a property of base-to-derived static casts.)

3.8 Reinterpret cast

Reinterpret casts to or from function usage types poses no problems for the anticipated implementation, as the usage contract is part of the type, not part of the execution-time data. I don't think any change is needed to [expr.reinterpret.cast], with the exception that note 4 in ¶6, which is already too strongly phrased, will become even less accurate.

3.9 Reference binding

The function usage conversion above makes the anonymized type of a function usage type reference compatible ([dcl.init.ref]¶4) with the usage type. Therefore, lvalue references to the anonymized type may be bound to lvalues of the usage type.

3.10 Calling the function

A call to a function through an expression of incomplete function usage type is ill-formed.

The restriction in [expr.call]¶6 on calling a function through a different type must be relaxed to allow calling a function through a usage type whenever it could be called through the anonymized type of the usage type.[2]

In the anticipated implementation, the code for checking a usage contract may be understood as using a class with three inline member functions: a constructor, a destructor, and a postcondition checker. An object of the type is constructed immediately before control is transferred to the called function; postconditions are checked and the object is destroyed immediately after control returns.

```
template < assertion_semantic_parameters >
struct __checker_for_fancy_op
   {
    references_to_parameters      members;
    data_saved_for_postconditions more_members;

    inline __checker_for_fancy_op( parameter_refs );
           // Called just before transfer of control.
           // Stores references to the parameters,
           //    checks preconditions, and
```

[2]It has been suggested that we weaken this restriction further, providing a feature similar to covariant return types of virtual functions. I don't know a good way to implement such a feature, and even if I did, it would be beyond the purview of this paper, and perhaps of SG21.

```
            //    saves data for postconditions.

    inline void check_postconditions( result_ref );
            // Called just after control returns.
            // Checks postconditions.

    inline ~__checker_for_fancy_op();
            // Called just after checking postconditions,
            //     or during stack unwinding.
            // Destroys data saved for postconditions.
   };
```

The code to check the assertions may either be inlined into the call site or gathered together under three linkage names. (I don't know a universally available technique that combines the three parts into a single function call, leaves room for saved postcondition data, and neither copies nor moves the parameters or result of the invoked function. If someone does, that would be an improvement.)

3.11 Member function usage types

Member functions also need usage types. To keep things simple: a usage type definition that has an explicit object parameter declaration defines a member function usage type.

```
int fancy_member_op( this C& self, const int x ) usage
   pre( x >= 0 )
   post( r: r >= 0 )
   post( r: r <= x );
```

Given this declaration, the type `fancy_member_op` is a "usage type for a member function of `C`", and so `fancy_member_op*` is "pointer to member function of `C` with usage `fancy_member_op`."

TODO: Can we forward declare them without naming the object type? Perhaps:

```
auto fancy_member_op( this auto&&, ... ) usage;
```

4 Addendum: reflections on other aspects of contracts

This addendum is not part of the proposal; you may stop reading here. It's just a collection of thoughts about other aspects of the contracts proposal, and about features that may be considered in future papers.

Stepping back, the central idea here — that a function call may be subject to two sets of preconditions and postconditions — colors my thinking about other aspects of the contracts proposal. We can look at every function call as being governed by two contracts:

An implementer's contract is a collection of terms offered by the function implementation.

A caller's contract is a collection of terms required by the function caller.

In an ideal function call, these two contracts are *compatible*, by which I mean:

- If the caller's preconditions are satisfied on entry to the function, it follows that implementer's preconditions will also be satisfied on entry to the function.
- If *in addition* the implementer's postconditions are satisfied on exit from the function, it follows that caller's postconditions will be satisfied on exit from the function.

A well-written contract is compatible with itself, but this is not a trivial property. Both side effects and unspecified behaviors may render a contract incompatible with itself.

4.1 Direct function calls

Direct function calls allow us to take shortcuts that hide this complexity. Often, a caller will select an existing function that satisfies the caller's needs, and, rather than codify those needs, simply accept the implementer's contract as its own. Or sometimes the process is reversed: the needs of a particular caller are known to the implementer, who codifies those needs and implements a function to satisfy them. In either of these scenarios, the caller's contract is identical to the implementer's contract. If we assume that the duplicated contract is self-compatible, it seems as if there is only one contract, because only the first execution of the common pre- or postconditions may fail.

This is one reason I argue for two (possibly ignored) executions of the pre- and postconditions of a direct function call. The outer repetitions verify compliance with the caller's contract, while the inner repetitions verify compliance with the implementer's contract. **Two is not an arbitrary number here; it is the number of parties to a function call.**

The apparent doubling goes away in indirect calls. When the caller's contract is not a copy of the implementer's contract, both contracts are executed, but neither contract appears to be doubled.

4.2 Virtual functions

A virtual function call is analogous to a call through a function pointer: we may think of it as an indirection through a function pointer stored in an object's vtable. In this analogy, a class defining a virtual function is doing two things: it is providing a means for indirect usage, and it is providing a function implementation that can be used either indirectly via those means or directly via a qualified function call.

The member function declaration can thus be associated with two contracts. The pointer in the vtable, like other function pointers, should have a usage contract, suitable for callers that that wish to call whatever function overrides the virtual function. If the function is implemented, it should have an implementer's contract. There are then three types of calls to the function:

Direct calls Qualified calls (`s.X::f()`) are direct calls; as in other direct calls, the caller adopts the implementer's contract as its own.

In a direct call, we expect the caller's contract to be compatible with the implementer's contract because they are identical and self-compatible.

Singly-indirect virtual calls Calls to a virtual function by unqualified name (`s.f()`, where `f` is virtual) are singly-indirect calls. The caller adopts the usage contract associated with the pointer stored in the vtable.

In a singly-indirect virtual call, we expect the caller's contract to be compatible with the implementer's contract because the implementer has designed their contract to be compatible with the virtual usage of the function being overriden.

Doubly-indirect calls Virtual calls through pointers-to-member-function (`(s.*mp)()`, where `mp` points to a virtual function) are doubly-indirect. The member pointer `mp` may be a pointer to a member function usage type, in which case the caller's contract is the contract of that member function usage type. Otherwise, the pointer points to an anonymous member function type, and the caller's contract is empty.

In a doubly-indirect call, we expect the caller's contract to be compatible with the implementer's contract transitively: the pointer's usage type was bound to a virtual function with compatible usage, and the virtual function was overridden by a function with compatible implementation. The intermediate contract (usage of the virtual function) plays a part in our reasoning, but not in execution; no code relies upon its terms. Only the caller's and implementer's contracts need be executed.[3]

[3] I am not against executing the intermediate contract on theoretical grounds, but the anticipated implementation here leaves no room for an intermediate contract to be executed. Contract checking thunks have been suggested, but in a future where we save data at precondition time for use by postconditions, I don't know where the thunk gets the stack space it needs. Neither the caller nor the implementation could coordinate with the thunk to provide stack space, because to achieve zero overhead, they must be unaware of the thunk.

This is why I argue for a syntax that allows a class to specify two contracts for its virtual functions. One is the implementer's contract, and the other is the usage contract for the pointer in the vtable. Often the two contracts are identical, and we should make that case simple. But we know of cases where the two differ (usually with the implementation having stronger postconditions), and we should make room for that.

If we settle on the function usage syntax above, we can extend it to express a second "virtual usage" contract while remaining compatible with the current contracts proposal.

```
struct foo
  {
   // A virtual function with one contract for direct calls,
   // and a different virtual usage contract:
   virtual int bar( const int x )
         post( r: r == x )    // for the implementation and direct usage
      virtual usage
         pre( x >= 0 )        // for singly-indirect usage
         post( r: r >= 0 )
         post( r: r <= x );

   // A virtual function where the direct call contract is the same
   // as the virtual usage contract
   virtual int baz( const int x )
      pre( x >= 0 )        // for the implementation and direct usage,
      post( r: r >= 0 )    // and also for singly-indirect usage.
      post( r: r <= x );

   // A member function usage type that that can be used to form
   // pointers to either of the member functions above:
   // &barlike{foo::bar} or &barlike{foo::baz}.
   int barlike( this foo& self, const int x ) usage
      pre( x >= 0 )        // for doubly-indirect usage
      post( r: r >= 0 )    // (or indirect to a nonvirtual member)
      post( r: r <= x );
  };
```