Document: P3271r1
Date: October 14, 2024
Reply-to: Lisa Lippincott <lisa.e.lippincott@gmail.com>
Audience: SG21 (Contracts)

Function Types with Usage (Contracts for Function Pointers)

Lisa Lippincott

Abstract

This paper introduces *function types with usage,* novel function types that have associated contracts. Function types with usage are named function 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 About the paper

The present revision, r1, is intended not as a finished product, but as a progress report and further exploration of the design space. I'm putting it in the mailing to make it generally available, but there is no need for SG21 discussion in Wrocław. This is not a paper targeting C++26.

1.1 History

Revision 1 is an interim progress report, with some revisions.

- The addendum about virtual functions became moot, and was removed.
- The terminology is improved. The term "function usage type" is replaced by "function type with usage." This allows more elegant and consistent terms for related types, e.g, "function type without usage," "pointer to function with usage," and "usage-erased reference to function."
- Revision 1 embraces the keyword-based "alternate syntax" from revision 0. The contextual keyword based syntax proposed in revision 0 made declarators more complex, and provided little benefit.
- Revision 1 adopts a strategy of expanding the definition of "similar" ([conv.qual]) to include function types. The similarity relationship ignores differences in usage.
- An exploration of two related features "declarations of appropriate usage" was added.
- An exploration of library features based on this feature was added.
- Speculation on how function types with usage may improve coroutines has been added.

Revision 0 was presented to SG21 in St. Louis in the afternoon of 2024-06-26. Two polls were taken:

- *We want to spend more time considering P3271R0 "Function usage types", in the MVP timeframe.* Consensus was against. (SF 0; F 1; N 9; A 8; SA 2)
- *We want to spend more time considering P3271R0 "Function usage types", as a post-MVP extension.* Consensus was in favor. (SF 12; F 8; N 2; A 0; SA 0)

Revision 0 contained an addendum outlining how contracts for virtual functions could be understood and implemented consistently with this proposal. A proposal (D3097r1) along those lines was accepted by SG21 and EWG in St. Louis.

2 Introduction: a tale of function usage

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

2.1 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 );
```

2.2 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.

2.3 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.

Please don't be confused: the class `fancy_wrapper` only illustrates a stage in our thinking. It's a data type, while function types with usage, introduced below, are function types. The class `fancy_wrapper` is analogous to a "pointer to function with usage," which is a kind of pointer type.

2.4 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.

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

   This defines a *function type with usage,* 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 types with usage, even if those other 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-time matter. For pointers, identical representation is essentially required: the type `fancy_op*` is similar (in the sense of [conv.qual]) to `int(*)(int)`.

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`.

2.5 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 nearly 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 a range of implementations, while contracts on individual 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.

3 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.

3.1 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.

3.2 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.

3.3 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.

4 The details

4.1 Types with and without usage

Function types with usage are novel function types, distinct from the kind of function types in C++23. When a distinction is necessary, the previously-existing kind of function types, are referred to as *function types without usage*.

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

Each function type with usage is based on a function type without usage. The latter type is referred to as the *usage-erased type* of the former.

```
function_usage fancy_op:( const int x )->( int r )
   pre( x >= 0 )       // The usage-erased type is int(int)
   post( r >= 0 )
   post( r <= x );
```

Specifically, the usage-erased type is arrived at by applying these rules:

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

The qualifiers "with usage," "without usage," and "usage-erased" may also be used with pointers and references:

- `fancy_op*` is a *pointer to function with usage* or *function pointer with usage*.
- `int(*)(int)` is a *pointer to function without usage* or *function pointer without usage*.
- `fancy_op&` is a *reference to function with usage* or *function reference with usage*.
- `int(&)(int)` is a *reference to function without usage* or *function reference without usage*.
- `int(*)(int)` is the *usage-erased type* of `fancy_op*`.
- `int(&)(int)` is the *usage-erased type* of `fancy_op&`.

For convenience, if T is a function, pointer, or reference type without usage, the usage-erased type of T is T itself.

4.1.1 Type equality

Type equality for function types with usage is determined by comparing fully qualified names and template arguments, much as it is for class types. This is in contrast to function types without usage, where type equality is determined by comparing parameter types, result types, and noexceptness. A function type with usage is not the same type as any function type without usage.

4.1.2 Type similarity

A notion of *type similarity* is defined in [conv.qual]¶2 and given force largely by [basic.lval]¶11. In brief, when two types S and T are similar, objects of type S may be accessed through expressions of type T, and vice-versa.

I propose to extend this notion to function types, decreeing that two function types are similar when they have the same usage-erased type, with the effect that a function of one type may be called through an expression of any similar type.

Similarity already extends recursively to types generated by adding cv-qualifiers, "pointer to," and "array of" (when the arrays don't have different known bounds). This property would be preserved, so that when F and G have the same usage-erased type, "array of three const pointers to F" will be similar to "array of three pointers to G."

4.2 Function types with usage may be incomplete

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

Ideally, forward declaration of a function type with usage should reveal only enough information to make pointers to the type complete (which may simply be the size and alignment of the pointers). Using a keyword to introduce the type declaration makes this easy:

```
function_usage fancy_op;
```

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

4.3 Function types with usage may be produced by templates

To express the needs of function or class templates, we need templates for function types with usage:

```
template < std::integral T >
function_usage fancy_integral_op:( const T x )->(T r)
   pre( x >= T{} )
   post( r >=  T{} )
   post( r <= x );

template < std::integral T >
T fancy_integral_calculation( fancy_integral_op<T>& op, T start );
```

Templates for function types with usage 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 <>
function_usage fancy_integral_op< char >:( const char x )->(char r) // OK
   pre( x >= '0' )
   pre( x <= '9' )
   post( r >= '0' )
   post( r <= x );

template <>
function_usage fancy_integral_op< short >:
     ( const int x )->( short r )   // ill-formed
```

[TODO] The restriction on result and parameter types is poorly thought out. It seems to suggest instantiating the base template in order to compare it to the specialization. Why would we do that?

4.4 Attaching usage to a function

The syntax used in this paper for attaching usage 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.

[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.

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

- The type with usage `u` is complete, and
- Either the expression `f` is a constant expression or the type of `f` is a complete function type with usage.

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 type with usage. These should be applied as if in an analogous function call expression. In the simplest case, the type with usage `u1` is not a template specialization, and we imagine an analogous attachment function `attach_u1`:

```
function_usage u1:( usage_func_params )->( usage_result );

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 function call expression `attach_u1(f)`.

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

```
template < usage_tmp_params >
function_usage u2:( usage_func_params )->( usage_result );

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. And because a function type with usage is a type, its template arguments should be more closely analogous to class template arguments than function template arguments.

4.4.1 Other non-cast explicit conversions to types with usage

Because the syntax is so similar, I propose that a function style implicit conversion using parentheses, *e.g.,* `fancy_op(halve)`, should only be well-formed when `fancy_op{halve}` is well-formed. A C-style cast `(fancy_op)halve` may be allowed in the wider contexts where a static or reinterpret cast would be allowed. All of these operations have the same result: an expression of type `fancy_op` referring to the function `halve`.

4.5 Null pointer conversions

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

4.6 Implicit function usage conversions

I propose to treat pointers and references to function types without usage as indicating functions with unknown usage contracts. This treatment suggests that a function pointer with usage may be implicitly converted to its usage-erased type. This should be a standard conversion [conv.general] sequenced before the existing function pointer conversion, so that in addition to erasing usage, an implicit conversion sequence may drop `noexcept`.

There are no standard conversions between types with usage, or from types without usage to types with usage.

4.7 Static cast

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

4.8 Reinterpret cast

A reinterpret cast should be able to add, erase, or alter usage, producing a callable result wherever the result type is similar to either the original function type or the original function type with noexcept removed. Reinterpret casts may be performed even when the function usage type(s) are incomplete.

Reinterpret casts to or from types with usage pose 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.

4.9 Reference binding

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

4.10 Calling the function

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

The restriction in [expr.call]¶6 on calling a function through an expression of different type must be relaxed to allow calling a function through a type with usage whenever it could be called through the usage-erased 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 that asserts preconditions, an `operator()` that asserts postconditions, and a destructor. 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 __asserter_for_fancy_op
   {
    references_to_parameters      members;
    data_saved_for_postconditions more_members;

    inline __asserter_for_fancy_op( parameter_refs );
            // Called just before transfer of control.
            // Stores references to the parameters,
            //    asserts preconditions, and
            //    saves data for postconditions.
```

[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.

```
    inline void operator()( result_ref );
            // Called just after control returns.
            // Asserts postconditions.

    inline ~__asserter_for_fancy_op();
            // Called just after asserter 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.)

4.11 Member function types with usage

Member functions also need types with usage, so that we can form pointers-to-member-functions with usage. Incomplete member function types need to be distinguishable from nonmember function types, because pointers to these types have different layout characteristics. Therefore I suggest using a different keyword to introduce the declaration of a member function types with usage.

```
member_function_usage fancy_member_op;

member_function_usage fancy_member_op: C::( const int x )->( int r )
   pre( x >= 0 )
   post( r >= 0 )
   post( r <= x );
```

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

5 Declarations of acceptable usage (experimental)

One criticism of the early proposal was that using function usage types involves too much explicit type conversion. One may desire a way to declare that "it's OK to use `halve` as a `fancy_op`," or that "it's OK to use an `even_fancier_op` in place of a `fancy_op`."

Generally, these situations arise when one has examined two function contracts and found that the converted-from contract (that of `halve` or `even_fancier_op`) places stronger restrictions on the called function and the converted-to contract (`fancy_op`) places stronger restrictions on the caller. Verifying this compatibility, while not a job for a C++ compiler, is a reasonable target for static analysis.

In addition to providing a programmer-friendly convenience, we can ease the burden of static analysis if we can associate these compatibility checks with a particular translation unit — that is, if many translation units can rely upon a declaration of compatibility, while only a single translation unit takes responsibility for the compatibility by providing a definition.

At first glance, treating compatibility as something that is separately declared and defined would seem to increase overhead: some additional function must be called or some new entity stored. Inlining would remove the overhead, but would defeat the purpose of separating the declarations from definitions. But in this case, there is a way to thread the needle, getting zero overhead despite not inlining the definition. *We will introduce a form of declaration that has only one possible non-deleted definition.*

5.1 Usage redeclarations

Let's reuse our `identity` and `halve` functions for another purpose: discounting prices. Without usage, we might write this code:

```
int identity( const int x )    post( r: r == x   );
int halve   ( const int x )    post( r: r == x/2 );

const std::array< int(*)(int), 7 > half_off_weekdays =
   { identity, halve, halve, halve, halve, halve, identity };
```

Switching to a function type with usage makes the code more expressive, but the array becomes wordier:

```
function_usage discount:( const int list_price )->( int charged_price )
   pre( list_price >= 0 )
   post( charged_price >= 0 )
   post( charged_price <= list_price );

const std::array< discount*, 7 > half_off_weekdays =
   {
    discount{identity},
    discount{halve}, discount{halve}, discount{halve},
                     discount{halve}, discount{halve},
    discount{identity}
   };
```

In the declaration of the array, the repetition of the type name dominates the text, but contributes little to our understanding. The sequence of function names has been obscured. It would be better to move the usage attachments outside of the array initializer, so that only two attachments are necessary. That can be done by declaring references.

```
discount& identity_discount = discount{identity};
discount& halve_discount    = discount{halve};

const std::array< discount*, 7 > half_off_weekdays =
   {
    identity_discount,
    halve_discount, halve_discount, halve_discount,
                    halve_discount, halve_discount,
    identity_discount
   };
```

That's better, but the `_discount` wart we added to our identifiers still clutters the text. The extra declaration also introduces an opportunity for a subtle error: a clumsy edit to the source might leave `identity_discount` not referring to the `identity` function.

Fundamentally, we don't want a new identifier. We want to allow the same identifier to refer to the same function, just with an additional type. We can allow this: let a function to be redeclared under the same qualified name but with a different type, as long as the types are similar.

```
discount identity;   // identity may be used as a discount
discount halve;      // halve may also be used as a discount

const std::array< discount*, 7 > half_off_weekdays =
   { identity, halve, halve, halve, halve, halve, identity };
```

The language mechanism here is simple overloading. The declarations introduce new names that overload the original names, in the same namespace. The new name always refers to the function with the same name in the same namespace with the same signature. (The usage type must therefore be complete at the point of declaration.)

To avoid introducing ambiguities, these new names are not viable when calling a function, but only when taking the address of an overload set, as in [over.over], and only when there is a target type.

As the new name does not refer to a new function, no new linkage name need be created. Despite only seeing a declaration, the compiler knows that the name refers to the original function.

For the sake of localizing analysis, I think we should allow definitions for these declarations, so we can pick a translation unit responsible for mating the contracts. We can require the definition to be defaulted.

```
discount identity = default;   // no other non-deleted definition is allowed
discount halve    = default;
```

5.2 Usage conversions

One usage may impose more stringent restrictions on a function than another. For example, a `discount` may be more stringent than a `price_adjustment`. In these cases, it may be useful to specify that one may sort of function may be used as the other.

```
function_usage price_adjustment:( const int list_price )->( int charged_price )
   pre( list_price >= 0 )
   post( charged_price >= 0 );

function_usage discount:( const int list_price )->( int charged_price )
   pre( list_price >= 0 )
   post( charged_price >= 0 )
   post( charged_price <= list_price );

usage_conversion discount->price_adjustment;
```

Such a conversion would allow `discount*` to convert to `price_adjustment*`, and allow `price_adjustment&` references to bind to `discount` lvalues. These would not be standard conversions.

As with usage redeclarations, a definition can be added for the benefit of localizing analysis.

```
usage_conversion discount->price_adjustment = default;
```

5.3 Usage declarations and conversions defined as deleted

It may also be possible to ban certain usages or usage conversions by defining them as deleted. More investigation is needed.

```
discount twice = delete;
usage_conversion price_adjustment->discount = delete;

discount *x = &discount{twice};   // ill-formed: the best match is deleted

price_adjustment& y = price_adjustment{identity};
discount& z = discount{y};        // ill-formed: deleted conversion
```

6 Function types with usage in the standard library (speculative)

A handful of extensions to the standard library can make function types with usage more widely useful. Each of the features listed here is likely to require compiler support, rather than being implementable as a pure library feature.

6.1 Type traits and concepts

A type trait should distinguish types with usage from those without, and a type trait should erase usage from a type.

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

static_assert( is_type_with_usage_v< fancy_op > );
static_assert( is_type_with_usage_v< fancy_op* > );
static_assert( is_type_with_usage_v< fancy_op& > );

static_assert( !is_type_with_usage_v< int()(int) > );
static_assert( !is_type_with_usage_v< int(*)(int) > );
static_assert( !is_type_with_usage_v< int(&)(int) > );

static_assert( is_same_v< int()(int),  usage_erased_t< fancy_op > );
static_assert( is_same_v< int(*)(int), usage_erased_t< fancy_op* > );
static_assert( is_same_v< int(&)(int), usage_erased_t< fancy_op& > );
```

The concepts `function_with_usage` and `function_without_usage` are handy below, and I suspect in general.

```
template < typename T >
concept function_with_usage    = is_function_v<T> &&  is_type_with_usage_v<T>;

template < typename T >
concept function_without_usage = is_function_v<T> && !is_type_with_usage_v<T>;
```

6.2 function_with_usage_matching

A number of people have expressed the desire to copy the contract from a function into a function type with usage, in order to provide a match for refactoring or testing. A library template can provide this capability.

```
template < function_without_usage& f >
function_usage function_usage_matching;
```

The type produced by this template, `function_usage_matching<f>`, has the same parameter list, result type, noexceptness, and contract as the function `f`. Different values for the argument `f` will, of course, produce different function types with usage, even if the argument functions have textually or semantically indistinguishable contracts.

6.3 Contract asserters

One may sometimes desire to wrap a contract around some operation that cannot be isolated into a function, *e.g.*, a `co_await` expression. This desire can be addressed by making available in the library a class similar to the one described as an implementation detail in section 4.10.

```
template < function_with_usage T >
class contract_asserter
   {
    public:
       contract_asserter( parameter_refs... );
       ~contract_asserter();

       void operator()( result_ref ) const;
   };
```

Unlike most library classes, this one should be documented to use contract assertions, as the entire purpose of the class is to invoke the assertion machinery. The constructor asserts the preconditions of `T`, and saves and captured data, as well as pointers to parameters used by the postconditions. The postconditions of `T` are asserted by `operator()`, using the saved data.

The implementation of the contract_asserter type must be careful not to expose details of the contract in its examinable properties. Specifically, its size must not vary with the amount of data captured for postconditions. One way to implement a class like this is as a wrapper around a coroutine; the captured data can be stored in objects local to the coroutine. After asserting preconditions and capturing data, the coroutine suspends itself awaiting a result reference. (For this reason, I have referred to this sort of class as a "cocontract" in some discussions. But the name `contract_asserter` is more descriptive.)

In conjunction with a feature that lets postconditions capture data, contract asserters allow one contract to be inserted into another:

```
int my_fancy_op( const int i )
   post( [ fancy_post = contract_asserter<fancy_op>(i) ]
         ( r: fancy_post(r) );

int replacement_for_f( const int i )
   post( [ f_post = contract_asserter<function_usage_matching<f>>(i) ]
         ( r: f_post(r) );
```

To facilitate the nesting of contracts, the captures should be performed bottom to top, in the reverse order of the postconditions.

7 Using types with usage to improve coroutines (speculative)

The execution of a coroutine brings together several separately-written pieces of code in a complex interaction. The purpose of function contracts is to specify the interfaces between separately-written pieces of code, but we haven't yet addressed the full complexity of interactions in a coroutine.

By my count, there are seven kinds of separately-written code involved with a coroutine.

- The caller, which invokes the "ramp function," written by a user;
- resumers, which invoke the `resume` member of a coroutine handle, written by users;
- the `coroutine_handle`, part of the standard library, templated on the promise type;
- the coroutine itself, written, mostly, in [dcl.fct.def.coroutine]¶5, with behavior governed by the promise type and parameters;
- the coroutine body, written by a user; and
- a complex of operations involved in an await-expression, described in [expr.await], and governed by the awaiter type;
- a complex of operations involved in a yield-expression, described in [expr.yield], and governed by the promise type.

As things stand, we have a way to specify a caller's contract on a coroutine. Because function types with usage allow us to specify a contract divorced from a function, we can use them to specify conditions to be asserted around the other portions of coroutine execution.

Specifically, type names looked up in the promise type could designate three contracts:

- A resumer contract, specifying conditions for resumption, either by `resume` or by coroutine chaining. This contract applies as if an entire chain of resumed coroutines were a function. The chain begins with the `await_resume` resuming the coroutine, and ends with the `await_suspend` method of a void-returning or boolean-returning awaiter.

  This contract can be copied to the declaration of `coroutine_handle<P>::resume()` for resumer-side checking when that `resume` function is used to resume the coroutine. The resume function in `coroutine_handle<void>` has no promise type, and so cannot expose this contract to the caller.

Resumption preconditions can be checked in the coroutine as well, and by one coroutine when chaining to another.

The resumption postcondition of the final coroutine in a chain can be checked within that coroutine, but, absent heroic implementation effort, the resumption postconditions of earlier coroutines in a chain cannot be checked.

- A body contract, specifying conditions for entry into and exit from the coroutine body (after the initial awaiter and before the final awaiter). This contract applies as if the body were a function.
- A yield contract that acts as if `yield` were a function. Preconditions are checked before `yield_value`, and postconditions at the end of the yield-expression, after the internal await-expression.

One more contract may be specified by awaiter types:

- An await contract that acts as if `await` were a member function of the awaiter. Preconditions are checked after the awaiter is determined (by the "*o* expression") and postconditions at the end of the await-expression, after `await_resume`.