Document: P0465R0
Date: 2016-10-16
Reply-to: Lisa Lippincott <lisa.e.lippincott@gmail.com>
Audience: Evolution Working Group

# Procedural function interfaces

Lisa Lippincott

**Abstract**

In order to reason clearly about the behavior of functions in C++, we need to specify function interfaces precisely. Here I will introduce a notation for function interfaces, with the goal of allowing local reasoning about functions and local testing of that reasoning. This approach differs from many others in focusing on the needs of procedural languages: instead of describing platonic relationships between values, these interfaces describe causal relationships between function invocation events.

By *local reasoning*, I mean that we will cover the execution paths of a program with a set of overlapping *neighborhoods*, and reason about each neighborhood independently. We can then, by noting that neighborhoods must agree in the overlapping regions, prove that certain aspects of a program are globally correct.

By *local testing*, I mean that conformance to interfaces may be tested by instrumentation added to an individual neighborhood, without altering the execution of the remainder of a program.

## 1 Overview

In my estimation, the difficulty of proving the correctness of everyday code does not stem from mathematical complexity: our programs rarely rely on sophisticated mathematics. Nor does it stem from the complexity of our programs: we strive to keep the reasoning behind our programs simple and easily apparent to every reader.

Instead, I've come to understand the difficulty as largely linguistic: the language in which we write a mathematical proof is so very different from the language in which we write a procedural program that joining the two together is a formidable task. In short, mathematics presents procedural programmers with a poor user interface.

One response to this mismatch is to find a way of programming that more closely matches mathematical language. That's been tried, with limited success. But I'm taking the opposite approach, trying to find a way to express the mathematical proof of a program's correctness in a procedural way, so that it fits harmoniously into procedural programs.

In the end, my goal is to have a single text that can be read either as a procedural program or as a mathematical statement of that program's correctness. This does, of course, involve changes to the programming language, but I think many programmers will recognize these changes as coming from familiar techniques or concepts. The remainder of this document describes the necessary concepts in detail:

**Assertions** are experiments that can be proposed in a program that mark limitations on correct behavior or illuminate the reasoning at a point in a function. When read mathematically, assertions are quantifiers over execution paths.

**Function implementations** allow complex operations to be presented to function callers as atomic. When read mathematically, a function implementation is a proof, or, more precisely, an attempted proof.

**Function interfaces** are inline functions wrapped around function implementations, containing assertions delimiting the conditions under which a function may correctly be called, and under which it may correctly return. When read mathematically, an interface is the statement of the theorem to be proved by the implementation.

**Function neighborhoods** are comprised of a function implementation, its interface, and, recursively, each interface called by the function. A function's neighborhood is the portion of the program we must understand in order to follow a function's reasoning. Mathematically, the function neighborhoods form a cover of the execution paths of the program, with neighborhoods overlapping in interfaces. This cover defines the notion of *local*.

**Capabilities** are assertions that in some neighborhoods are viewed as atomic abstractions. Within those neighborhoods, these locally atomic conditions may be asserted in interfaces, marking the causal connections between function implementations. Mathematically, capabilities are locally conserved quantities arising from symmetries at the function's level of abstraction. The conservation laws allow locally correct usage of a capability to be defined without reference to the capability's implementation. The local correctness can be tested by keeping tallies.

**The basic interface** is a simple function interface that is derived directly from a function declaration. It expresses the usual expectations placed on a function with that declaration, and provides a starting point for defining other interfaces. This is a convenience both programmatically and mathematically, reducing the complexity of writing interfaces.

**Propriety** is an assertion applied to an lvalue, expressing that it meets the usual expectations for its cv-qualified type. A large part of the basic interface is made up of assertions of propriety. For class types, propriety may include assertions from the class definition, providing class authors

with some influence on the basic interface. In some contexts, propriety is a locally atomic assertion, allowing the object to be treated abstractly.

**Non-basic function types** allow the type system to discriminate between functions with different interfaces, and allow reasoning about functions called indirectly.

In addition to all of those changes, we will need interfaces for fundamental operations such as scalar assignment and lvalue-to-rvalue conversion, connecting those operations to capabilities such as writability and readability. Likewise, we will need interfaces for standard library functions, ensuring that they are used correctly.

This road is long, and this roadmap is incomplete. There are questions still unanswered, and doubtless many still to be asked. But I think this is an important direction to take C++.

## 2 Interface and Implementation

In this scheme, the definition of a function is composed of two separate parts, an *interface definition* and an *implementation definition*. An interface definition is an inline function definition — it can be thought of as defining the inline portion of the complete function. It may contain a statement "`implementation;`" once within its outermost block, indicating the point at which the implementation of the function is executed. The implementation statement divides the interface into two regions, the *prologue* and the *epilogue*.

```
inline void foo()
  {
   // prologue

   implementation;

   // epilogue
  }
```

When the function is called, the prologue is executed, and if control reaches the implementation statement, the implementation body, defined in an implementation definition, is executed:

```
void implementation foo()
  {
   // implementation body
  }
```

Implementation definitions are distinguished by names prefixed with the keyword `implementation`. Implementations are matched to interfaces by name, namespace, signature, and template arguments. Even if declared inline, an implementation is not an interface.

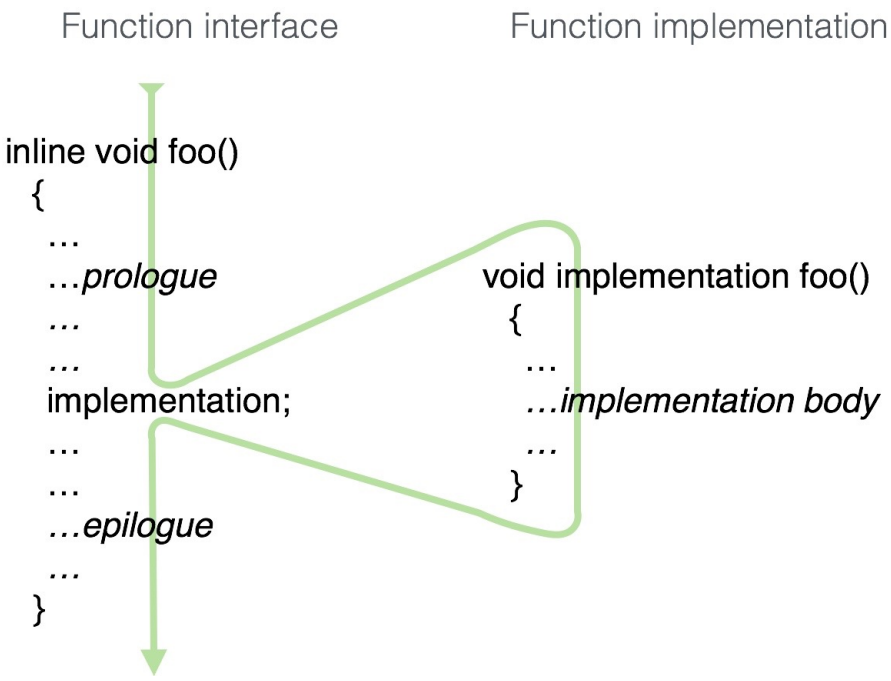


Figure 1: The execution path through an interface and its implementation

If the implementation returns, control passes to the epilogue of the interface. In a value-returning function, the identifier "`result`" is defined in the epilogue, referring to the result returned by the implementation. If control reaches the end of the epilogue, or if the epilogue executes a *return-statement*, the object returned by the implementation becomes the result of the function.

## 3 Neighborhoods

To reason locally about a program, we cover a program's execution paths with overlapping neighborhoods. Most neighborhoods are associated with the execution of a function implementation, but there are also neighborhoods associated with the initialization and destruction of static objects and with the destruction of exception objects.

A function implementation's neighborhood begins with entry into the function's interface, includes the function's prologue, implementation body, and epilogue, and ends with exit from the interface. It also includes, recursively, the interfaces — but not implementations — of functions called from the prologue, implementation body, or epilogue. Thus, when a function is called, its interface is in both the calling and implementation neighborhoods, but its implementation is excluded from the calling neighborhood.

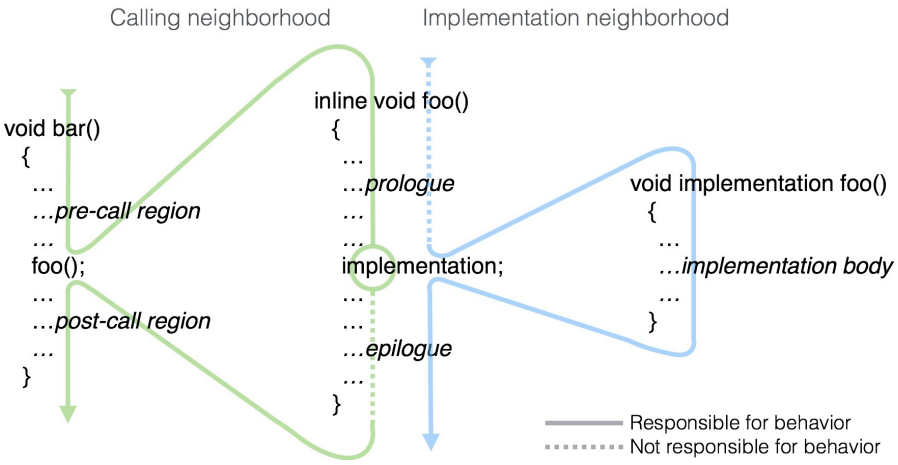


Figure 2: Neighborhoods overlap in interfaces

Where neighborhoods overlap, we designate one neighborhood as the *responsible* neighborhood, and look to that neighborhood to prove correct behavior of the code. Responsibility passes from caller to implementation on entry to the implementation, and returns to the caller on exit from the epilogue. Thus, for example, failure of an assertion in the prologue is blamed on the calling neighborhood, but failure of an assertion in the implementation body or epilogue is blamed on the implementation neighborhood.

The executions of the prologue and epilogue correspond to the functional notions of precondition and postcondition, respectively. But this correspondence is not exact — the prologue and epilogue are not expressions that may be true or false, but instead, procedural code whose execution may flow from one point to another.

The calling neighborhood is responsible for correct behavior in the prologue, but need not ensure that execution reaches the implementation point. If execution of the prologue returns to the caller, exits with an exception, or otherwise does not reach the implementation point, there is no implementation neighborhood. The implementation neighborhood, therefore, may not only assume correct behavior in the prologue, but also that the implementation point is reached. Likewise, the implementation neighborhood must ensure the correct behavior of any portions of the epilogue that are executed, but the calling neighborhood may further assume that the epilogue exited by returning (on execution paths continuing from the call site) or with an exception (on exceptional execution paths).

# 4 Local reasoning

One simple, but essential, guarantee forms the basis of all interfaces: each interface requires, at the point that responsibility passes to the implementation neighborhood, that no action with undefined behavior has yet been taken, and, at the point responsibility is returned on exit, guarantees that there has still been no undefined behavior.

```
inline void foo()
  {

   // No undefined behavior so far!
   implementation;

   // No undefined behavior so far!
  }
```

When these local guarantees are applied in all neighborhoods, we can stitch them together into a global proof: if there is no undefined behavior in the prologue of `main` or of any namespace-scope static initializer,[1] then there is no undefined behavior before entry into any neighborhood in the program. Further, if we give each fundamental operation an interface with a guarantee of defined behavior in its implementation, there will be no undefined behavior in the entire program.

To truly complete this proof, we need to ensure a similar guarantee when a function exits with an exception. For this we need an extension of the interface syntax, specifying an *exceptional epilogue*:

```
inline void foo()
  {
   // No undefined behavior so far!
   try implementation
     {
      // No undefined behavior so far!
     }
   catch ( ... )
     {
      // No undefined behavior so far!
      // (The exception is implicitly rethrown.)
     }
  }
```

[1] These prologues aren't entirely vacuous: they must make assertions about `argc`, `argv`, and the storage lifetime of the static objects. We will see that these assertions are always implicit.

# 5 Using assertions in interfaces

Undefined behavior in programs often stems from nonlocal causes. A dangerous condition in one neighborhood can lead to undefined behavior in another, as shown in figure 3.

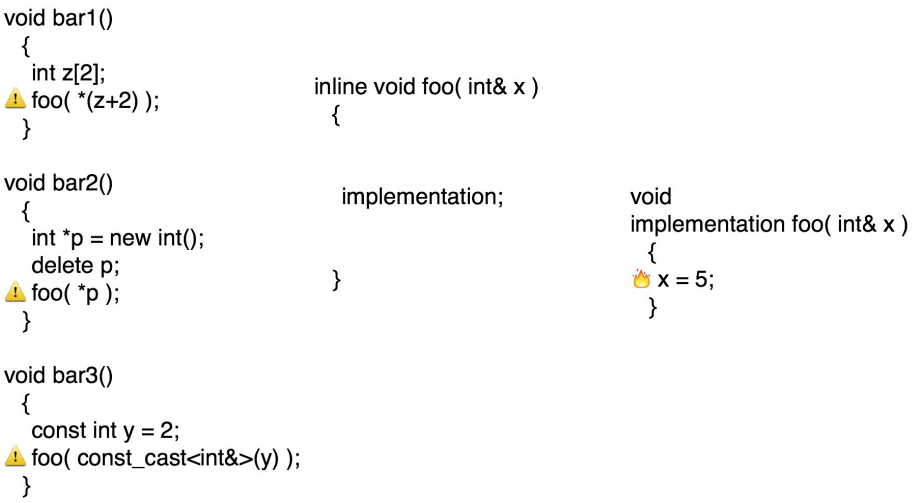


Figure 3: Dangerous behavior in one neighborhood may lead to undefined behavior in another neighborhood.

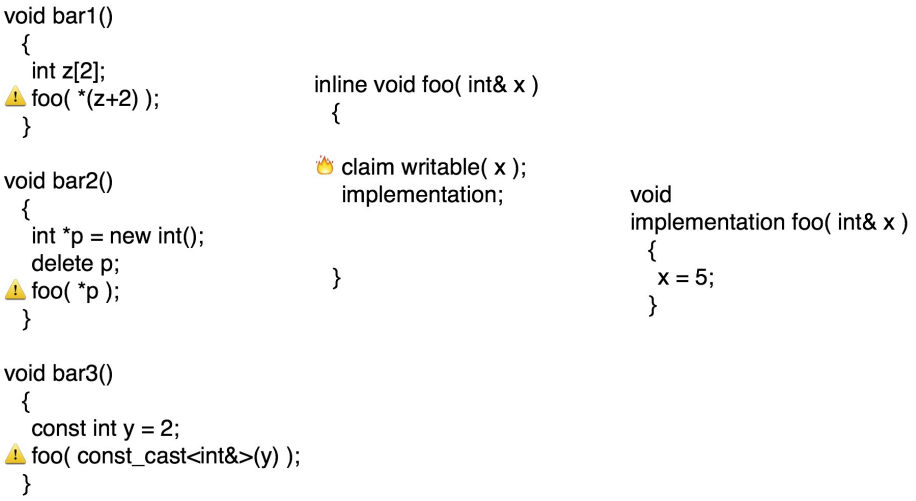


Figure 4: When the undefined behavior is moved into the interface, the neighborhood responsible for the danger becomes responsible for the undefined behavior.

Interfaces allow us to move the undefined behavior earlier in the program, making the same neighborhood responsible for both the danger and the undefined behavior. Once we have done so, local reasoning about the responsible neighborhood can identify the problem.

In figure 4, a statement "`claim writable(x);`" in the interface causes undefined behavior before the implementation is entered. The keyword `claim` introduces an assertion intended to be provable. Mathematically, it acts as an existential quantifier over execution paths: the claim is satisfied by an execution path reaching the next statement. The expression "`writable(x)`" expresses a *capability*: it has no effect when `x` is locally writable, and undefined behavior otherwise. The statement "`claim writable(x);`" therefore expresses the precondition for assignment: that execution passes, without undefined behavior, through the evaluation of `writable(x)`.

Other basic capabilities include `readable` (a precondition of lvalue-to-rvalue conversion), `destructible` (a precondition of destruction), `deallocatable` (a precondition of deallocation), and `callable` (a precondition of function call through a pointer or reference).

Capabilities can be combined to form more complex capabilities. One example of a complex capability is `proper`, which I use to mean "the usual conditions and capabilities expected of an lvalue, given its cv-qualified type." For example, a proper POD lvalue is readable, and if it is non-const, it is also writable. Propriety is fundamental to interfaces, and more fully discussed in section 9.4.

# 6 Assertions

Assertions, as used here, have a dual role. Computationally, they indicate the correct behavior of a function and provide tests to measure that correctness. And mathematically, they act as quantifiers over the possible execution paths in a function, indicating the mathematical statement we wish to prove about the function. To fill the mathematical role, we need three kinds of assertions, with different relationships to quantification:

**claim** is the most common form of assertion, introducing a condition intended to be proved. Mathematically, a claimed assertion acts an existential quantifier: "there exists a continuation of the current execution path to the completion of the claimed code or to an enclosed assertion or implementation that does not complete." We can use claimed assertions liberally to delineate correct behavior and prove strong statements about the correctness of our programs.

**posit** is a more dangerous form of assertion, introducing a condition intended to be true, but not proved. Mathematically, a posited assertion acts a universal quantifier: "for each continuation of the current execution path to the completion of the posited code or to an enclosed assertion or implementation that does not complete." (In deterministic contexts, this looks more like a conditional: "if the execution path continues...") Posited assertions

weaken the mathematical statements we prove about our programs, and we would ideally restrict them to the minimal axioms we must assume about our programs. But they may also be used to assert truths that are simply inconvenient to prove.

**require** is a neutral form of assertion, taking on the quantifier of a surrounding assertion — a required assertion is posited when it is nested in a posited assertion, and claimed otherwise. The principal use of `require` is in inline functions, to build complex conditions that can be either claimed or posited.

```
inline void deletable( int* p )
   {
    if ( p != 0 )
       {
        require destructible( *p );
        require deallocatable( *p );
       }
   }
```

When we examine a neighborhood of a program locally, we weaken the quantifiers associated with claims for which our neighborhood has no responsibility. To an implementation neighborhood, a claimed assertion in its prologue looks very much like a posited assertion; we might say it is *locally posited*. Likewise, a claimed assertion in the epilogue of a function is locally posited in the calling neighborhood. All claimed assertions still require proof, but the proof of a locally posited assertion is assumed to exist outside the neighborhood.

## 6.1 Testing assertions

An assertion, whether claimed or posited, is *tested* by evaluating its expression, and, if it has type other than `void`, contextually converting the result to `bool`. The test *succeeds* if the evaluation and conversion completes and the resulting boolean value, if any, is `true`. In particular, the test does not succeed if the expression or conversion exits with an exception, hangs, terminates, or otherwise does not complete.

The proof of a responsibly claimed assertion must therefore show that the assertion completes.[2] If the asserted expression may hang or have undefined behavior, the proof fails.

A program has undefined behavior if the test of an assertion fails. A C++ implementation may, of course, provide some useful behavior, particularly in the case of a boolean test producing `false`. In some situations, this could involve halting the program and communicating with diagnostic tools. But in other

[2]This sounds like solving the halting problem, but it's not. We rely upon the programmer to provide clear reasoning, and we may reject any program where the proof is not evident.

situations, the continued operation of the program may be paramount. Continuing execution with the next statement is not practical, as that statement may rely on the correctness of the assertion. But throwing an exception derived from `logic_error` strikes a middle ground, letting execution continue in a context that does not rely on the assertion.

To promote robust programming, our mathematical proofs should support the possibility that any failing assertion could throw `logic_error`, and prove the correctness of execution paths where that exception is handled. This prevents an execution path that catches `logic_error` from relying on the success of assertions within the *try-block*.

```
void foo( int x, int y )
   {
    posit x != 0;
    try
       {
        posit y != 0;
        possibly_throw_exception();
       }
    catch ( const std::logic_error& )
       {
        claim x != 0;   // provable: previously posited
        claim y != 0;   // not provable: posit may have failed
       }
    catch ( ... )
       {
        claim y != 0;   // provable: not a logic_error
       }
   }
```

## 6.2 Ignoring and assuming assertions

Testing assertions can slow down program execution tremendously. In some contexts we readily accept that cost, but in other contexts, particularly when we have confidence in the assertion's success, the testing becomes onerous. For this reason, whether or not an assertion is tested is undefined. A C++ implementation may specify that certain assertions are tested in certain contexts, perhaps under the control of a compiler flag, pragma, or attribute.

To allow for further optimization, a C++ implementation may *assume* an untested assertion — that is, provide defined behavior only in the case where the assertion, had it been tested, would have succeeded. Alternatively, the C++ implementation may *ignore* the untested assertion, letting the assertion have no effect, whether or not it would have succeeded.

Our mathematical proofs need to consider the possibility that an assertion may be untested. It is impractical in a procedural language to insist that assertions not have side-effects, but we can insist on a more narrowly tailored

condition. We will insist that each correct assertion have a *null reimplementation*: that is, that an assertion whose test would succeed will have no material effect on the result of the responsible neighborhood. (Or, more precisely, on the exercise of capabilities passed to the caller in the function's epilogue.)

```
int foo( int x )
   {
    posit x++ != 0;
    return x;   // proof failure: testing the assertion
                // materially affects the function result.
   }
```

## 6.3 Non-local assertions

Inline functions using `require` allow us to build complex assertions out of simpler ones, but they do so in a locally transparent way — each calling neighborhood contains the simpler assertions. But sometimes it is useful to form an opaque non-local assertion, separating it into an interface and an implementation. To a caller, these non-local assertions appear as axioms, theorems, and atomic capabilities; their implementations can be compiled separately and provide the proofs that support the caller's usage.

We create non-local assertions by attaching assertion keywords to implementation statements. As with local assertions, the assertion keyword determines the role the assertion plays in the proof of the calling neighborhood. But in each case, a non-local assertion has a null reimplementation, and may be ignored or assumed. All non-local assertions therefore have result type `void`.

### 6.3.1 Claiming non-local proofs

A non-local claimed assertion acts as the proof of a theorem: its prologue plays the part of the premise of the theorem, and its epilogue plays the part of the conclusion. The implementation body provides the logical connection between the premise and conclusion.

```
inline void fermat( unsigned x, unsigned y, unsigned z )
   {
    claim x != 0 && y != 0 && z != 0;

    claim implementation;

    claim x*x*x   + y*y*y   != z*z*z;
    claim x*x*x*x + y*y*y*y != z*z*z*z;
   }

void implementation fermat( unsigned x, unsigned y, unsigned z )
   {
    ... [The proof is too long to fit in this example.]
```

```
   }
```

### 6.3.2 Positing non-local assumptions

A non-local posited assertion acts as an axiomatic assumption. Its epilogue is assumed to follow from its prologue without proof. The implementation of such an assertion is never defined, and is always ignored.

```
inline void fermat( unsigned x, unsigned y, unsigned z )
   {
    claim x != 0 && y != 0 && z != 0;

    posit implementation;
    // The proof is too long to fit in this program.

    // These claims are not proved here, but are testable.
    claim x*x*x   + y*y*y   != z*z*z;
    claim x*x*x*x + y*y*y*y != z*z*z*z;
   }
```

### 6.3.3 Requiring non-local capabilities

Finally, a non-local required assertion encapsulates an abstract condition that may be either posited or claimed. Callers may reason about changes to the condition through the capabilities expressed in its interface. Within a required implementation, assertions using the `require` keyword are considered the responsibility of the caller, and so need not be proved in the implementation neighborhood.

```
inline void frangible( const int& x )
   {
    require implementation;
   }

void implementation frangible( const int& x )
   {
    require x % 6 == 0;
   }

inline void frange( int& x )
   {
    claim frangible( x );
    implementation;
    // x may have changed, and might not be frangible any more.
   }

inline void foo()
```

```
{
 int x = 6;
 posit frangible( x ); // Unproven, but testable.
 frange( x );          // frangible(x) is provable here.
 frange( x );          // Can't prove frangibility here.
}
```

Because the details of a non-local capability are hidden from its caller, the caller can't prove them directly.[3] A caller can only show that each responsible claim of the capability follows from a previous assertion of the same capability. Therefore, in local testing, it often suffices to treat non-local capabilities abstractly instead of, or in addition to, testing their implementations.

# 7 Capabilities

Assertions in a procedural interface have a purpose beyond expressing platonic preconditions and postconditions. They also express *capabilities* used by the implementation — ways the implementation can effect change or be affected by change. An assertion of non-volatile readability in a prologue indicates that the implementation may be affected by a previous change to an object; an assertion of writability indicates that the implementation may change the object.

In the epilogue, the roles of the neighborhoods are reversed: an assertion of readability allows future events in the caller to be affected, and an assertion of writability allows future events in the caller to effect changes.

We will endeavor to make these channels the exclusive means of communication between an implementation and its caller. That is, an implementation will be meaningfully affected by its caller only in its exercise of capabilities asserted in its prologue, and the caller will be meaningfully affected by the implementation only in its exercise of capabilities asserted in the epilogue.

Capabilities are represented by expressions of type `void`; often these are calls to opaque assertions that use `require implementation`. Because the type is `void`, a function may assert capabilities, but may not directly assert the absence of a capability or branch on the presence or absence of a capability.

That inability leads to relaxed requirements on the primitive capabilities: each primitive capability expression has no effect when the capability is locally present, and undefined behavior otherwise. A primitive capability may be implemented with a trivial function, or may be instrumented with elaborate bookkeeping and testing. Further, it is possible to use such instrumentation in some neighborhoods and not others. We will see that instrumentation contained within a neighborhood suffices for testing the assertions of capabilities for which the neighborhood is responsible.

[3] These details can be exposed by implementation friendship; see section 8.

## 7.1 The path of a capability

To demonstrate reasoning about a capability, consider a non-const, non-volatile byte `b`, and the capability `writable(b)`.

We first encounter `writable(b)` when the byte is created. If we imagine an interface for the construction of the byte, `writable(b)` would be asserted in the constructor's epilogue. The neighborhood invoking this constructor need not prove this assertion, because the invoking neighborhood is not responsible for the epilogue. The assertion transfers the writability to the invoking neighborhood.

After construction of the object, the neighborhood may responsibly claim `writable(b)`. If it does so as part of the prologue of a called function, the capability is shared with that function, and taken from the calling neighborhood on entry to the implementation. As the called function returns, it may assert the writability in its epilogue, and thus return the writability to its caller.

In this way, claims of writability trace a path starting from the construction of the object, passing in and out of the implementations of functions that may write to the byte. The path of writability bypasses other function calls, and we infer that those function calls do not alter the byte. While this path may pass through many neighborhoods, each step in the path is local: each claim of writability follows from a previous claim of writability in the same neighborhood.

The `destructible` capability also traces a path from the construction of the object to its destruction, but it need not trace the same path: some interfaces assert writability but not destructibility. We infer that those functions do not destroy the byte. But the reverse is never the case: because we want the writability to vanish at the point of destruction, we make writability a precondition to destructibility. To destroy our non-const byte, or even assert its destructibility, a function must hold the writability.

To avoid leaking objects, we want to ensure that the writability and destructibility persist until they reach the object's destruction. We can ensure this locally with a simple rule: when a neighborhood reaches the end of its execution (by returning or by throwing), the neighborhood must have no remaining unshared capabilities. If we think of destructibility as containing an element of anti-writability, this becomes a conservation law: every neighborhood conserves writability. Other capabilities obey similar local conservation laws.

## 7.2 Capabilities and cv-qualifiers

Many capabilities are associated with lvalues, and may be overloaded or specialized in ways that distinguish lvalues with different cv-qualifiers. For example, `writable( volatile int& )` is a capability that allows assignment only through a volatile reference, while `writable( int& )` allows assignment through either a volatile or a non-volatile reference.

When a cv-qualified object is created, the surrounding neighborhood receives an apparently weaker collection of capabilities than it would if the object were not cv-qualified. Creation of a const `int` provides no writability, and while creation of a volatile `int` may provide both readability and writability, it provides

the weaker forms that may only be used through volatile references.

To avoid leaking capabilities, we want each capability granted by the construction of an object to be consumed by its destruction. It follows that destruction of a cv-qualified object has a different precondition than destruction of an unqualified object — `destructible` must also vary with cv-qualification. Creation of a cv-qualified object produces a similarly cv-qualified destructibility, which may only be used to destroy an object with that specific cv-qualification.

When considering the conservation of capabilities, we can imagine that cv-qualified destructibility carries the non-const or non-volatile capabilities that are restricted during the lifetime of the object, releasing them only at the point the lifetime ends.

## 7.3 Locally testing capabilities

We can test a neighborhood's use of capabilities without tracking the capability from its creation to its destruction — it suffices to track it from the point it enters the neighborhood to the point where it leaves. In between those points, the capability may be responsibly claimed.

Usually, a capability enters a neighborhood through a claim for which the neighborhood is not responsible, in the neighborhood's prologue or in the epilogue of a function called from the neighborhood. A capability may also be introduced to the neighborhood by a posited assertion.

A capability leaves a neighborhood after it is asserted in the prologue of called function or in the epilogue of the neighborhood. A capability asserted in a called prologue leaves the neighborhood at the implementation point, and a capability asserted in the epilogue leaves the neighborhood when the epilogue completes. (A capability asserted in a non-exceptional epilogue does not leave the neighborhood when that epilogue exits with an exception; to leave the neighborhood, it must be reasserted in an exceptional epilogue.)

One subtlety makes the tracking of capabilities more complicated. A capability that is asserted only *indirectly* in the prologue of a function — that is, only in the epilogue of a function called from the prologue, and not in an assertion — is not available for use in the implementation body, but is shared with the implementation for the duration of the epilogue. These capabilities are generally associated with variables local to the interface, for which the implementation body has no name.

At the end of a neighborhood's execution, we check that each capability remaining in the neighborhood has been asserted in the epilogue, and is therefore not leaked.

## 7.4 Aliasing

When we reason about function execution, it is essential to understand when the exercise of one capability meaningfully affects the exercise of another. This understanding is frustrated by *aliasing,* the situation where two capabilities locally appear to be separate, but in fact allow effects to be communicated from

one to the other. We must, therefore, take care to express aliasing relationships in function interfaces.

When an object is created, the aliasing relationships between its capabilities are clear. Use of its destructibility has an effect on all its other capabilities, and, if the object is neither const nor volatile, use of its writability affects use of its readability.[4]

Since we have complete knowledge of aliasing relationships from the start, it suffices to remember those relationships as capabilities trace their paths through a program. Unexpected aliasing can only arise if the capabilities are passed from one neighborhood to another, but the aliasing is not. We can convey knowledge of aliasing using assertions of `aliased`, as in this interface:

```
inline int& operator+=( int& x, const int& y )
   {
    if ( &x == &y )
        claim aliased( x, y );

    claim writable( x );
    claim proper( y );
    // When aliased, changes to x meaningfully change y.

    implementation;

    claim aliased( x, result );
    // changes to x are the same as changes to result.

    claim proper( result );
    claim proper( y );
    claim proper( x );
   }
```

## 7.5 Testing adequacy of aliasing in an interface

For POD types, the claim of aliasing is easy to test at runtime: simply compare the addresses. But it is harder to test whether the claims of aliasing in an interface adequately express the actual aliasing. The nature of aliasing — locally distinct lvalues sharing an address — is lost at runtime when we identify an lvalue by its address. The distinction between the lvalues can only be maintained with elaborate instrumentation. But there is a simpler test, based only on the addresses, that should suffice to find many aliasing problems.

This test starts by counting the assertions of capabilities and of aliasing associated with each address and type. In each case, only direct assertions — that is, assertions that transfer the capability or aliasing — are counted. We define

[4]Exercise of the readability of a volatile object is not meaningfully affected by writing to it; any change is lost in the expectation of noise. "Only in silence the word, only in dark the light, only in dying life: bright the hawk's flight on the empty sky." —Ursula K. Le Guin

the *multiplicity* of capabilities for a type at an address as the difference between the number of capability assertions and the number of aliasing assertions.

For each address and type, we also track the kinds of capabilities asserted, and determine whether any capability mentioned affects another. That is, we note the addresses where both writability and nonvolatile readability are asserted, or where destructibility (and therefore also const volatile readability) has been asserted.

If the non-exceptional epilogue exits with an exception, the counting for the exceptional epilogue starts afresh.

At the implementation point, and again at the end of the epilogue or exceptional epilogue, we require these properties of the multiplicities:

**Positivity** Wherever aliasing or capabilities are asserted in a prologue or epilogue, the multiplicity must be positive. This prevents excessive assertions of aliasing, which can mask errors of insufficient aliasing.

**Exclusion (prologue)** At the implementation point, wherever any asserted capability affects another, the multiplicity must be one. This ensures that enough aliasing has been asserted to connect all the overlapping capabilities passed to the implementation.

**Exclusion (epilogue)** At the end of an epilogue, wherever capabilities are asserted in the epilogue but not in the prologue, if any asserted capability affects another, the multiplicity must be one. This ensures that enough aliasing has been asserted to connect all the overlapping capabilities created by the implementation.

**Conservation** Wherever capabilities are asserted in both the prologue and the epilogue, the multiplicity in the epilogue must be equal to the number of assertions of capabilities in the prologue. This ensures that enough aliasing is asserted when new lvalues refer to old capabilities, as when a function result refers to a parameter.

This test isn't perfect — an interface that has excessive aliasing may be invoked by a caller that has equally excessive, but different aliasing, and the test would succeed. But this test is simple to implement locally, and I expect it will catch many aliasing problems. Excessive aliasing in the interface can be detected in the least-aliased case, which is often the most-tested case. Once the excessive aliasing in the interface is corrected, calls with incorrect aliasing can be reliably detected.

```
inline void foo( int& a, int& b )
   {
    claim aliased( a, a );   // excessive

    claim proper( a );
    claim proper( b );
    implementation;
```

```
    claim proper( b );
    claim proper( a );
   }

void bar()
   {
    int x = 0;
    foo( x, x );   // incorrect aliasing,
                   // undetected by the multiplicity test.
   }

void baz()
   {
    int x = 0;
    int y = 0;
    foo( x, y );  // Here, the excessive aliasing in the prologue
                  // is detected. Once that is corrected,
                  // the error in bar will be detectable.
   }
```

## 7.6 Aliasing non-local capabilities

Exercises of non-local capabilities may also be related by causal effects, and the interfaces to non-local capabilities need to express these causal relationships. In the interfaces of such capabilities, we can use `require` to assert sub-capabilities, allowing the capabilities to overlap. The whole capability then cannot be asserted without also asserting the part.

```
inline void destructible( const int& x )
   {
    require readable( x );
    require implementation;
    require readable( x );
   }
```

These sub-capabilities also act to identify the whole capabilities: two executions of the same non-local capability are aliased if corresponding capabilities required in the prologue are aliased. If they are aliased, corresponding capabilities required in the epilogue are also aliased: this is proved in the implementation neighborhood, and may be relied upon in the calling neighborhood.

Non-local capabilities that share a sub-capability are related causally. In the example above, exercise of `destructible` certainly affects the future use of `readable`. But this relationship is actually asymmetric: use of `readable` does not affect the future use of `destructible`. We can express causal asymmetry in a capability by adding `const` or `volatile` to the implementation statement:

```
inline void readable( const int& x )
   {
    const volatile int& cv_x = x;
    require readable( cv_x );
    require const implementation;
    // Located at cv_x, but does not affect future events there.
    // The implementation requires only const capabilities.
    require readable( cv_x );
   }
```

Exercise of a non-local const capability has no meaningful causal effect on the future exercise of capabilities; exercise of a non-local volatile capability is not meaningfully affected by previous exercise of capabilities.

A capability that is both const and volatile is thus exempted entirely from causal relationships. But it still marks an area in which other capabilities may overlap. For example, const volatile readability marks the overlap between volatile writability and nonvolatile readability.

```
inline void writable( volatile int& x )
   {
    const volatile int& cv_x = x;
    require readable( cv_x );
    require volatile implementation;
    // Located at cv_x, but unaffected by past events there.
    // The implementation requires only volatile capabilities.
    require readable( cv_x );
   }
```

Specifically, the exercise of a non-local capability $a$ only meaningfully affects the exercise of a non-local capability $b$ when $a$ precedes $b$, $a$ is non-const, $b$ is non-volatile, and some capability required in the epilogue of $a$ is also required in the prologue of $b$.

# 8 Friends of the implementation

We occasionally need to prove properties of a function's implementation that can't easily be expressed in its interface. One example is the transitivity of an ordering function: the function has only two parameters, but transitivity is a relation between three objects.

```
inline bool operator<( const T& a, const T& b )
   {
    implementation;
    // We can state anti-reflexivity:
    claim !result || a != b;
    // But we can't state transitivity here.
   }
```

We can get around this problem by stating transitivity as a separate theorem:

```
inline void transitive_less( const T& a, const T& b, const T& c )
   {
    claim a < b && b < c;
    claim implementation;
    claim a < c;
   }
```

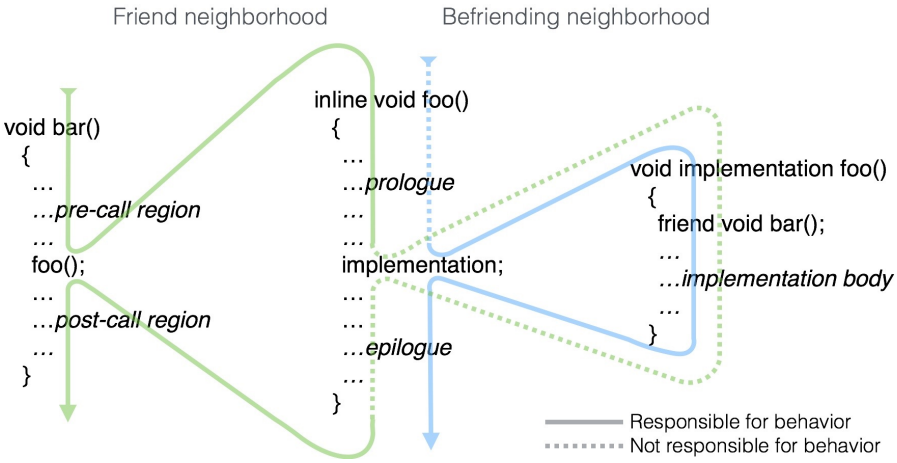


Figure 5: The friend neighborhood extends into the befriending implementation.

But the best way to prove transitivity may be to examine the particular implementation of `operator<`. We can allow this by making `transitive_less` a friend of the implementation of `operator<`.

```
bool implementation operator<( const T& a, const T& b )
   {
    friend void transitive_less( const T&, const T&, const T& );
    return false;   // Trivially transitive
   }

void implementation transitive_less
                    ( const T& a, const T& b, const T& c )
   {
    // Seeing operator<, we know this point is unreachable.
    claim false;
   }
```

To take advantage of implementation friendship, the implementation of the friend must be defined (or instantiated) after the befriending implementation, in the same translation unit. When it is, the neighborhood of the friend implementation extends into calls to the befriending implementation, allowing details

of the implementation to be used in proofs associated with the friend neighborhood.

# 9 The basic interface

Listing out every capability that passes through each interface becomes cumbersome. To streamline the process of writing interfaces, we will agree on a way to derive a *basic interface* from a function declaration. The intent here is to have the basic interface express the usual expectations placed on a function call, and let other expectations be expressed by limiting or adding to the basic interface.

Because the basic interface is uncomplicated by limitations or additions, it is simple to write. Its body holds only a simple implementation statement:

```
inline result_type basic_interface( parameter_types... )
   {
    implementation;
   }
```

But the basic interface is not truly empty. The strength of the basic interface comes from the implicit assertions we take to be present.

## 9.1 Non-inline non-implementation functions

The basic interface has a second purpose: it is the interface we assume for most functions which do not declare an interface explicitly. That is, when we see a declaration "`result_type foo( parameters... );`" we will understand `foo` to implement a basic interface. Many functions implement the basic interface, and for such functions, this form has a convenient brevity.

But, if the basic interface is to have any strength at all, there must be some existing functions that do not meet its requirements. Any meaningful test in the basic interface could introduce undefined behavior into existing programs. It is therefore imperative that a C++ implementation can ignore the tests in the basic interface, preserving the behavior of existing programs. We will allow this flexibility by putting nothing but assertions into the basic interface.

Still, our ultimate goal is for each function to meet the requirements of its interface. Over time, we want to find the functions that need non-basic interfaces and write appropriate interfaces for them. To do so, we need to enforce the assertions in the basic interface.

To make the transition, we need selective enforcement of basic interfaces with scalable control. C++ implementations should provide a mode where basic interfaces are not generally enforced, but allow enforcement to be controlled using attributes, pragmas, or compilation flags. For programs that have more fully specified their interfaces, C++ implementations should also provide a mode where basic interfaces are generally enforced, with exceptions carved out in the same ways.

## 9.2 Inline functions without implementations

Inline functions without an implementation statement can generally be seen as degenerate interfaces in which the implementation (and therefore also the epilogue) is unreachable. They consist of a prologue alone.

Such interfaces are very useful: they compactly express the exact behavior of a function. In doing so, they express the widest possible precondition and the narrowest possible postcondition for the given code. That's exactly what's needed for many small functions.

There is one exception to this treatment: when a reference or pointer to an inline function without an implementation statement is formed, it must (largely for compatibility reasons) be treated as if it were not inline, and instead as an implementation of a basic interface. The proof that it satisfies the basic interface can be associated with the translation unit forming the reference or pointer.

## 9.3 Regular interfaces

Certain operations have extra assumptions routinely laid on them because of their names. For example, we expect a function named `operator=` to behave as an assignment operator, rather than have a basic interface. It seems both cumbersome and error-prone to write out the assignment interface for every non-inline assignment operator. Instead, for a small list of such names, we can invent *regular interfaces*, which we can assume when no interface is explicitly specified.

## 9.4 Propriety

A major component of the basic interface comes from assertions of *propriety* — assertions that an lvalue has the usual properties and capabilities associated with its cv-qualified type. Propriety, in this way, becomes a way for the type system to affect the basic interface.

The basic interface contains implicit assertions of propriety for parameters at the end of its prologue, epilogue, and exceptional epilogues. In addition, propriety of the result is asserted at the end of the epilogue, and propriety of the thrown exception at the end of the exceptional epilogues. Destructibility is not asserted for parameters, but is asserted for thrown exceptions and for results of object type.

```
inline result_type basic_interface( parameter_types... )
   {
    // All assertions shown are implicit.
    // Only propriety and destructibility are shown.

    claim proper( parameter );          // for each parameter
    try implementation
       {
```

```
      claim proper( result );        // if non-void
      claim destructible( result );  // for object types only
      claim proper( parameter );     // for each parameter
     }
  catch ( ... )
     {

      claim proper( exception );
      claim destructible( exception );
      claim proper( parameter );     // for each parameter
     }
}
```

In some ways, propriety mirrors the more functional notion of class invariant. But propriety isn't invariant — it's just commonly asserted. Also, by asserting capabilities, `proper` connects a type to the ideas of value and change. An object's value is the meaningful information that can be extracted using the capabilities incorporated in its const propriety, and the object's value changes through exercise of the capabilities in its non-const propriety.

## 9.5 Alternate propriety

We sometimes need an escape from implicit claims of propriety. One approach would be to invent a qualifier, say, `improper`, that can modify declarations of parameter and result types, eliminating the corresponding implicit claims. That approach seems to be too harsh. We rarely want to eliminate the entire claim of propriety, but instead, want to substitute a different notion of propriety.

As an example, consider moving a container. A container type's propriety — the usual capability implied when a container is used as a parameter or result — certainly includes the propriety of its elements. But for many container types, moving the container does not require propriety of the elements. It requires a weaker capability, which we can call `movable`. A function that sees its parameters only as objects to be moved can substitute `movable` for `proper`:[5]

```
template < class T >
inline void swap( [movable] T& a, [movable] T& b )
   {
    // All assertions shown are implicit.
    // For many types, movable is a weaker condition than proper.

    claim movable( a );
    claim movable( b );
    try implementation
       {
```

[5] I don't like this syntax, and I haven't checked that it is even workable, but it will suffice for early discussion. I'm looking at "`T& a claim movable(a)`" as an alternative.

```
        // In a complete interface, we would assert "swapping"
        // here: that the upcoming movable(a) will repeat the
        // previous movable(b), and vice-versa.
        claim movable( b );
        claim movable( a );
       }
    catch ( ... )
       {

        claim proper( exception );
        claim destructible( exception );
        claim movable( b );
        claim movable( a );
       }
   }
```

For cases where we really do need no propriety at all, we can now define `improper` as an empty inline function.

```
template < class T >
inline void improper( T& )
   {}

inline int *operator&( [improper] int& x )
   {
    // All assertions shown are implicit.

    claim improper( x );                // vacuous
    try implementation
       {

        claim proper( result );
        claim destructible( result );
        claim improper( x );            // vacuous
       }
    catch ( ... )
       {

        claim proper( exception );
        claim destructible( exception );
        claim improper( x );            // vacuous
       }
   }
```

Of course, this scheme is not limited to specifying properties weaker than `proper`. Many interfaces can be completely described with just alternate propriety functions. But the more procedural syntax is needed to relate parameters

to each other, parameters to results, or conditions in the prologue to conditions in the epilogue.[6]

## 9.6 Defining propriety

Keeping in mind that "proper" expresses the usual expectations on a parameter or result lvalue, the definition of proper for lvalues of non-class types is a fairly straightforward combination of readability, writability, and propriety of subobjects.

Class types have greater flexibility: they may define functions `operator proper()` that are asserted as part of propriety. They may also declare members with alternate propriety functions:

```
struct maybe_initialized
   {
    bool initialized;
    [improper] int x;

    void operator proper() const
       {
        if ( initialized )
           require readable( x );
       }

    void operator proper()
       {
        require writable( x );
       }
   };
```

Putting it together, we want propriety to express these properties:

- A proper scalar lvalue is readable: lvalue-to-rvalue conversion will have defined behavior.
- A proper non-const scalar lvalue is writable: assignment to it will have defined behavior.
- Conversion of an lvalue to a more cv-qualified reference produces a proper lvalue.
- Each subobject of a proper non-union lvalue is proper, unless the declaration of that subobject specifies an alternate propriety function. If an

[6] A more elaborate version of this scheme would specify a type, rather than a function. The constructor and destructor of the type could then assert asymmetric conditions, and connect the condition in the prologue to that in the epilogue. If the destructor could somehow detect whether its scope was exited with an exception, the type could also express exceptional conditions, like the strong exception guarantee.

alternate propriety function `ap` is specified in the declaration of a member subobject `m`, execution of the assertion "`require ap(m);`" has defined behavior.

- Each byte of a proper nonempty[7] trivially-copyable lvalue is proper, whether cast to `char` or `unsigned char`.
- If a proper lvalue `v` has polymorphic type, the expressions "`typeid(v)`" and "`dynamic_cast<`*cv* `void *>(&v)`" have defined behavior. (And thus calls to virtual functions are possible.) The corresponding full object is proper.
- If a proper lvalue `v` has class type, "`require v.operator proper();`" has defined behavior.

## 9.7 Non-local propriety

In some circumstances, we need to assert the propriety of an lvalue even though the definition of its type is not locally available. In these situations, we treat propriety as a non-local or partially non-local assertion.

### 9.7.1 Propriety of objects with runtime type

Sometimes an object's type is only known through runtime information. Even in those cases, we need to make assertions about the propriety of the object. We can do this with four functions named `proper_for_type`:

```
inline void proper_for_type( const type_info&, void *object )
   {
    // Aliasing not shown.
    require implementation;
   }
// And three more, for const, volatile, and const volatile
// Alternatively, these could be member functions of type_info.
```

A thoroughly-testing C++ implementation could store pointers to the propriety tests in an extended `type_info` structure, and use them to test `proper_for_type`.

### 9.7.2 Propriety of objects of polymorphic type

When `x` is an lvalue of polymorphic type, the assertion `proper(x)` asserts the propriety of the full object referred to by `x`, which is a typical part of the interface to virtual function calls.

Here, `typeid(x)` is available, so `proper_for_type` can be used to test the propriety.

[7] An empty base class may not have any bytes to call its own.

### 9.7.3 Propriety of exception objects

Every interface implicitly asserts the propriety of any exception thrown by its implementation. If the exception handling mechanism has access to the type id of the thrown exception, `proper_for_type` can once again be used to test the propriety.

### 9.7.4 Propriety of objects of incomplete class type

A function may assert `proper(x)` for an lvalue `x` of incomplete class type. This is quite common, happening implicitly in the basic interface when a parameter or result is a reference to an object of incomplete type.

This case presents more of a problem, as `typeid(x)` is not allowed for incomplete class types. An implementation may not be able to test such assertions, particularly if the type is defined nowhere in the program. It can still treat the propriety as an abstract capability, tracing its path from one assertion to another.

# 10 Function types

The strength of the type system is improved when functions with different interfaces have different types. But the utility of pointers and references to functions depends on the existence of many functions sharing the same type, and therefore the same interface. This leads to two questions: how can functions share interfaces, and how do we specify the interface of a function type?

For the functions with the basic interface, we have easy answers. Functions with basic interfaces share an interface if their parameter types and result types match. The types of such functions can be written with the existing syntax.

```
void f( int& );  // f has the basic interface for void(int&).
```

To share non-basic interfaces, we take a single function to be the exemplar of the function type, and invent a new syntax that lets a function explicitly share another's interface instead of defining its own. That is, for each exemplar function `foo(parameters)`, there is a type "function implementing the interface of `foo(parameters)`."[8]

```
inline int foo( int )   { ... implementation; ... }
// foo has a non-basic interface, so its type is not int(int),
// but "function implementing the interface of foo(int)."
// foo is an exemplar function, with a novel type.

auto foo implementation bar( int );
// bar has the same interface and result type as foo(int),
// and type "function implementing the interface of foo(int)."
```

[8] I haven't checked this syntax thoroughly; it's probably not quite right.

```
auto bar implementation baz( int );
// baz has the same interface and result type as bar(int),
// and thus also as foo(int).  Its type is also
// "function implementing the interface of foo(int)."

auto foo implementation (*p)(int) = &bar;
// p points to function with the same interface as foo(int).
```

## 10.1 Declarations of implementations

Most of the time, the definition of an interface serves as a declaration of the the corresponding implementation. It defines the name, signature, and type of the function. But a handful of other specifiers may be associated with the implementation of a function: `inline`, `static`, `extern`, `virtual`, `override`, `final`, and `=0`. These specifiers can be put in a separate declaration of the implementation:

```
inline void foo()
   {
    inline void implementation foo();
    implementation;
   }
```

But such a declaration is repetitive, particularly if the parameter list is long. By inventing a new *implementation-part* for the function declaration syntax, we can work these specifiers into interface declarations and definitions.

```
inline void foo() implementation inline
  {
   implementation;
   // The entire combined function is inline.
  }
```

## 10.2 Declarations of interfaces

As with class types, it is sometimes useful to declare a function type — an interface — separately from the definition of the type. In particular, it is often convenient to defer definition of a member function's interface to a point outside the class declaration. We can use a degenerate case of the syntax above to declare that a function is the exemplar of its type:

```
int foo( int ) implementation;
// foo(int) is the exemplar of its type,
// "function implementing the interface of foo(int)."
// In particular, foo(int) does not have a basic interface.
```

```
inline int foo( int )
   {
    ...
    implementation; // interface definition OK
    ...
   }
```

When an interface has been declared in this manner, but not yet defined, the function type is incomplete. Pointers and references to functions of incomplete type may be formed, but functions of incomplete type may not be called.

### 10.3 Pure interfaces

As with virtual functions, sometimes we will want to define an abstract interface without an implementation of the same name. We can apply a *pure-specifier* to the implementation to do this:

```
inline int foo( int ) implementation = 0
   {
    // foo(int) names only an interface, not a complete function.
    // It may not be called, and it has no address.
    implementation;
   }
```

## 11 Virtual functions

Calls to virtual functions pose the same problem as calls through function pointers: the calling neighborhood is not bound to calling a particular complete function, but must be bound to calling a particular interface. The solution to this problem is similar to the solution for function pointers.

A non-inline virtual function using the existing syntax is taken to be an implementation of the basic interface corresponding to its declaration, or, for certain special functions, a regular interface based on the name and signature. Inline virtual functions take on such an interface only when called through the virtual function mechanism. When the virtual mechanism is suppressed by the use of a *qualified-id*, the inline function is taken to be an interface with unreachable implementation.

Non-basic interfaces for virtual functions can be implemented by attaching the qualifiers `virtual`, `override`, `final`, and `=0` to implementations, rather than interfaces. When this syntax is used, the complete function formed by combining the interface with an implementation is virtual, but all overriders share the same interface.

```
struct base
   {
    virtual void f();
```

```
    // uses a basic interface; overriders must also use one.

    inline virtual void g();
    // object.g() uses the basic interface, but
    // object.base::g() uses an unreachable implementation.

    void h() implementation virtual;
    // h() and its overriders share a non-basic interface.
    // The exemplar of this interface is h.

    virtual void vv() implementation virtual; // not allowed
   };

inline void base::h() { implementation; }

struct derived: base
   {
    void implementation h() override;
    // Shares the interface of base::h()
   };
```

It is occasionally appropriate for a derived class to provide a more specialized interface to a virtual function it overrides. Calls through the derived class can then rely on the more specialized interface. This creates a situation where two interfaces are paired with a single implementation, creating two distinct complete functions. (Either or both of the interfaces may be a basic interface.)

```
struct d2: base
   {
    void h() implementation override;
    // d2::implementation h() is paired with
    // two interfaces: base::h() and d2::h().
   };

void foo( d2& x )
   {
    // This calls d2::h() paired with d2::implementation h().
       x.d2::h();

    // This calls base::h() paired with base::implementation h().
       x.base::h();

    // This calls the final overrider of d2::h(),
    // which shares the interface of d2::h().
       x.h();

    base& b = x;
```

```
    // This calls the final overrider of base::h(),
    // which shares the interface of base::h().
       b.h();

    // The final overriders of d2::h() and base::h()
    // share an implementation.
   }

struct d3: d2
   {
    void implementation h() override;
    // pairs with base::h() to override base::h(),
    // and with d2::h() to override d2::h()
   };
```

A vtable-based C++ implementation can implement this by adding a new vtable entry for each interface with virtual implementation, regardless of whether that interface shares a name and signature with a base class interface.

When two interfaces share implementations in this way, they need not have the same result type; it suffices that the result of the common implementation can be converted to the result type of each interface. There seems to be no need to limit this to pointer and reference conversions, as with covariant return types. The conversion used can be looked up at the point the implementation is defined. Each implementation neighborhood is responsible for the correctness of any conversion it applies.

# 12 Interactions with various language features

The sections above describe the major features of this proposal. This section, in contrast, is a hodgepodge of details, describing the ways the new features interact with other language features.

## 12.1 Exceptions

If an exception thrown from the prologue of a function causes the function to exit, the implementation neighborhood never takes responsibility from the calling neighborhood.[9] The calling neighborhood remains responsible for behavior (e.g., the destruction of automatic objects) throughout the interface and until control leaves the calling neighborhood.

If an exception thrown from the implementation body or epilogue causes the function to exit, the implementation neighborhood is responsible for behavior until control leaves the implementation neighborhood.

[9] To keep this clear, we prohibit implementation statements in *function-try-blocks*.

In the latter case, the caller cannot rely on the successful execution of the epilogue for information about the state of the program, as the epilogue may not have been executed. To make assertions about the state of the program in the event of an exception, we use a *try-implementation-statement*:

```
inline void foo()
  {
   // prologue

   try implementation
     {
      // epilogue
     }
   catch ( ... )
     {
      // exceptional epilogue
     }
  }
```

The exceptional epilogue is executed when either the implementation body or the non-exceptional epilogue exits with an exception. The behavior of an exceptional epilogue is more tightly restricted than that of an ordinary exception handler: it may not return, and if it exits with an exception, `terminate` is called. If control reaches the end of an exceptional epilogue, the exception is rethrown.[10]

A *try-impementation-statement* may only appear as the last statement of an interface.

### 12.1.1 noexcept

The `noexcept` qualifier specifies that an entire function not exit with an exception. As the entire function exits when the interface does, a noexcept interface must not exit with an exception. A noexcept function may have an exceptional epilogue; the program terminates after such an epilogue is executed. (This allows exceptional epilogues in conditionally-noexcept functions.)

Because `noexcept` does not change a function's type, it also cannot change a function's interface. If the language were changed to make `noexcept` part of a function's type, this rule would be changed to implicitly add "`claim false;`" to the end of the exceptional epilogues of `noexcept` functions.

## 12.2 Automatic variables

Automatic variables declared in the prologue of an interface have lifetime that extends through execution of the epilogue or exceptional epilogue. The names

[10] The harshness of these rules provides certainty to the caller in the event of an exception. A caller can only rely on an assertion when it has certainty that execution has passed through the assertion. If we allow execptional execution paths that don't complete the exceptional epilogue, the caller has little to rely upon when catching an exception or when destroying an automatic object.

of these variables are not in the implementation's scope.

An exiting implementation neighborhood is responsible for the implicit destruction of automatic variables declared in its prologue. Of course, if the implementation in not entered, the calling neighborhood remains responsible for these destructions.

## 12.3 Function-scope static variables

A function-scoped static variable declared in an interface is shared by all complete functions sharing that interface. Likewise, when two complete virtual functions share an implementation, static variables declared in the implementation are shared between the functions.

Indirect calls to an inline function through a pointer or reference (which pass through a basic interface) share static variables with direct calls (which do not pass through a basic interface).

## 12.4 Function parameters

Any parameter names declared in the implementation refer to the same objects as those declared in the interface, not copies of the interface parameters. This is not only efficient, but allows the implementation to rely on assertions about lvalues.

To avoid confusion, each parameter in the implementation must have the same type as the corresponding parameter of its interface, including top-level cv-qualifiers.

## 12.5 Function results

Function results may only be returned from the prologue or implementation of a function. In a non-exceptional epilogue, the identifier `result` may be used to refer to the result returned by the implementation.

If the result has object type and the epilogue exits with an exception, the result is destroyed after automatic variables declared in the epilogue, and before execution of any exceptional epilogue or destruction of automatic variables declared in the prologue.

### 12.5.1 Return statements

A *return-statement* may appear in the prologue of an interface. If the result type of the function is not `void`, such a *return-statement* must contain an expression. When an interface returns from its prologue, responsibility never passes to the implementation neighborhood.

A *return-statement* may appear in the epilogue of an interface, but in this case, it may not contain an expression. When such a *return-statement* is executed, or when control reaches the end of the epilogue, the object returned by the implementation (not a copy) becomes the result of the function.

A *return-statement* may not appear in an exceptional epilogue.

### 12.5.2 noreturn

The `noreturn` attribute specifies that an entire function must not return. Since the entire function returns when the interface returns, the program has undefined behavior if the interface returns.

If a function marked `noreturn` returns from its prologue, the calling neighborhood is responsible for the undefined behavior. If such a function returns from its epilogue, the implementation neighborhood is responsible for the undefined behavior.

Because `noreturn` does not change a function's type, it also cannot change a function's interface. If the language were changed to make `noreturn` part of a function's type, this rule would be changed to implicitly add "`claim false;`" to the end of the non-exceptional epilogue of `noreturn` functions, and to ban return statements from such interfaces.

## 12.6 Jump statements

Jump statements are not allowed to jump from the prologue of an interface to its epilogue, or from the epilogue to the prologue. (It's possible to relax the latter restriction, but doing so seems needlessly complex.)

## 12.7 Linkage

The linkage of a function's name does not affect its callers, and may be thought of as a property of its implementation. While interfaces are always declared inline, the complete function formed by combining an interface with an implementation takes on the linkage declared for the implementation. A forward declaration can be used to specify a linkage other than `extern`.

```
static void implementation foo();

inline void foo()
  {
   implementation;
   // The combined function has internal linkage,
   // but its type has external linkage.
  }
```

This extra declaration is repetitive, but can be replaced by an *implementation-part* in an interface declaration or definition:

```
inline void foo() implementation static
  {
   implementation;
   // The combined function's name has internal linkage,
```

```
   // but its type has external linkage.
  }
```

## 12.8 Language Linkage

The language linkage of a function also affects both its name and its type. We will associate the effect on the type with the interface of the function, and the effect on the name with the implementation. Nested linkage specifications allow the language linkage of a function's name to be separated from the language linkage of its type.

```
extern "C"
  {
   inline void foo() implementation static
     {
      implementation;
      // The function name has internal linkage,
      // and the function type has C language linkage.
     }
  }
```

A language linkage specification applied to an implementation affects only the linkage of the name, and not the type.

```
inline void foo() implementation extern "C"
  {
   implementation;
   // The function name has external C linkage,
   // and the function type has C++ language linkage.
  }
```

## 12.9 Templates

The declarations and definitions of interfaces and implementations may be template specializations. The implicit matching of interfaces with implementations only matches template specializations to other template specializations with the same template arguments.

```
template < class T > inline void foo( T ) { implementation; }
// implemented by specializations of
// template <class T> implementation foo(T)

inline void foo(int) { implementation; }
// implemented by implementation foo(int),
// not implementation foo<int>(int)

template <> inline void foo( int ) { implementation; }
```

```
// implemented by implementation foo<int>(int),
// not implementation foo(int)


template < class T > void implementation foo( T ) {}
// implements specializations of template<class T> foo(T)

inline void implementation foo(int) {}
// implements foo(int), not foo<int>(int)

template <> inline void implementation foo( int ) {}
// implements foo<int>(int), not foo(int)
```

Each specialization of an implementation template creates a need for a proof of correctness of the corresponding implementation neighborhoods. This proof can be associated with the translation unit that specializes or instantiates the implementation template.

In the future, it may be possible to use concepts to produce and check a proof schema for a template, rather than checking the proof of each specialization.

## 12.10 Function signatures

The ability to prove — or detect failure to prove — the correct use of interfaces relies on knowledge of which interface applies to a function call. It is therefore best that function signatures, and therefore function overloading, be unaffected by a function's interface.

## 12.11 Constructors

Interfaces to constructors wrap the entire construction: the prologue is executed first, before initialization of bases and members by the implementation. A constructor interface must therefore not contain both an implementation statement and a *ctor-initializer*.

```
struct Foo
   {
    int m;

    Foo()
      : m( 0 )   // Ok; no implementation statement allowed
      {}

    Foo( const int x )
      {
       claim even( x );
       implementation;
       claim even( m );
```

```
      }
   };

Foo::implementation Foo( const int x )
   : m( x )
   {}
```

Non-inline constructors without interfaces are, as usual, considered implementations of a basic constructor interface. Of course, basic interfaces for constructors differ from basic interfaces of non-constructor functions.

```
inline result_type basic::basic()
   {
    // All assertions shown are implicit.
    // Not all assertions are shown.

    claim unoccupied_space( *this );
    try implementation
       {

        claim proper( *this );
        claim destructible( *this );
       }
    catch ( ... )
       {

        claim proper( exception );
        claim destructible( exception );
        claim unoccupied_space( *this );
       }
   }
```

The assertion `unoccupied_space` is used here to keep the books balanced: it combines the capabilities associated with the storage lifetime of the object. Because these capabilities are required for construction, we may not construct overlapping objects without being aware of the aliasing. (If `unoccupied_space` is an opaque capability, we may not construct aliased overlapping objects at all. Unoccupied space within an object may still be created by explicit destruction.)

If a constructor supplies an alternate propriety function for `*this`, that function is used in place of `proper` upon successful construction. Such a constructor may construct an improper object, but the constructed object must still be `destructible`.

```
inline int::int() [uninitialized_proper]
   {
    // All assertions shown are implicit.
    // Not all assertions are shown.
```

```
    claim unoccupied_space( *this );
    implementation;

    claim uninitialized_proper( *this );
    claim destructible( *this );
    // The destructor must not require initialization.
   }
```

Inline constructors without implementation statements make the calling neighborhood responsible for the entire execution of the constructor, including base and member initialization.

The lifetime of a constructed object begins on successful exit from the interface of its non-delegating constructor.

## 12.12 Destructors

Interfaces to destructors wrap the entire process of destruction. The lifetime of the object ends on entry to the prologue of its destructor. Return statements may not appear in the prologue of a destructor, and if the prologue of a destructor exits with an exception, `terminate` is called.[11] Member and base subobjects are destroyed by the implementation, and the epilogue is executed after the object is completely destroyed. The storage lifetime of the object extends through the epilogue.

Non-inline destructors without interfaces are considered to have a basic destructor interface, which has a form distinct from other basic member function interfaces.

```
inline result_type basic::~basic()
   {
    // All assertions shown are implicit.
    // Not all assertions are shown.

    claim proper( *this );
    claim destructible( *this );
    try implementation
       {

        claim unoccupied_space( *this );
       }
    catch ( ... )
       {

        claim proper( exception );
```

[11] The alternative seems to be making the calling neighborhood responsible for the destruction of subobjects in this case, but that gets messy.

```
        claim destructible( exception );
        claim unoccupied_space( *this );
       }
   }
```

For most types, `destructible` is a capability that requires, and is thus aliased to, `proper`. But if the destructor of a type specifies an alternate propriety function, `destructible` is aliased to that capability instead.

```
inline int::~int() [uninitialized_proper]
   {
    // All assertions shown are implicit.
    // Not all assertions are shown.

    claim uninitialized_proper( *this );
    claim destructible( *this );
    implementation;

    claim unoccupied_space( *this );
   }

inline void destructible( int& x )
   {
    require uninitialized_proper( x );
    require implementation;
    require uninitialized_proper( x );
   }
```

## 12.13 Implicitly declared and defined functions

All implicitly declared functions are inline; the usual rules for inline functions apply. For the most part, this means that they will be treated as interfaces with unreachable implementation. One exception of note is an implicitly defined virtual destructor, which is paired with a basic destructor interface when called through the virtual mechanism.

## 12.14 Lambda expressions

The member functions of the type associated with a lambda expression are also inline, and the usual rules for inline functions apply.

## 12.15 Recursion

The mathematical statements we prove to show local correctness have an unorthodox structure, because they are formed by quantifying over execution paths in a looping structure where the loops themselves may contain further quantifiers

(i.e., assertions and interface calls). In such a *procedural sentence*, a quantifier may be revisited an unbounded number of times.

The Borel determinacy theorem assures us that when the quantifiers are finitely nested, such a sentence has a truth value. Loops therefore cause no problem — an assertion within a loop must be decided before the loop can return to the assertion. If a loop continues without end, the closest enclosing assertion fails.

Nonlocal recursion, that is, recursion that passes through an implementation boundary, also causes no problem for local reasoning, as the entire execution of the implementation falls outside the neighborhood. Each nested execution of an implementation has its own neighborhood.

But a local recursion, in which interfaces or inline functions call each other indefinitely, may nest quantifiers without bound in a single neighborhood. Fortunately, such nesting is easy to detect and rule out through static analysis of the execution paths.

The most obvious quantifiers in a neighborhood are its assertions. We can keep them from nesting infinitely by forbidding local execution paths that reenter an assertion without exiting it.

Quantifiers are also introduced by function interfaces. A caller of a function interprets the called interface mathematically as "for every execution path through the prologue and implementation there exists an extension through the epilogue or an exceptional epilogue."[12] To keep these quantifiers from nesting infinitely, we also forbid local execution paths that reenter a function, or any function with the same interface, from the epilogue or exceptional epilogue of its interface, without completing the epilogue or exceptional epilogue.

[12]Implementation neighborhoods prove their own correctness as "every execution path through the prologue has an extension that either ends in a called implementation, completes the epilogue, or completes an exceptional epilogue." Program termination and infinite nonlocal recursion both cause local execution to end in a called implementation.