Classification of Contract-Checking Predicates

Document #:	P2712R0
Date:	2022-11-13
Project:	Programming Language C++
Audience:	SG21 (Contracts)

Reply-to: Joshua Berne <jberne4@bloomberg.net>

Abstract

A *function-level contract* is an agreement between a function provider and users of that function. A *contract check* encodes an algorithm to help determine whether a *function-level contract* has been violated. Here we explore ways to classify such algorithms: in particular, how to measure (albeit subjectively), for a given domain and context, the extent to which runtime evaluation of a contract check is safe.

Contents

1 Revision History

Revision 0

- Original version of the paper for discussion at the November 2022 WG21 meeting in Kona.

2 Introduction

The word *contract* is a heavily loaded term.

Definition: Contract

A *contract* is a binding agreement among two or more parties.

Definition: Contract Check

A *contract check* encodes an algorithm to help determine whether one or more aspects of a contract have been violated.

Definition: Contracts

The term *Contracts* denotes a framework intended to support contract checking in C++.

Contracts in general may involve legal issues such as those pertaining to licensing or government regulation. In software engineering, however, we tend to consider a much more specific subset of all possible contracts: those pertaining to interactions that occur within the software itself. In particular, we tend to focus on contracts that are specifically attached to an individual function.

Definition: Function-Level Contract

A *function-level contract* is a binding agreement between the provider of a function (the callee) and its users (the callers) that specifies (1) under what circumstances the associated function may be invoked properly (the preconditions) and (2) what must be true during and after execution of a properly invoked function (the essential behavior).

Definition: Precondition

A *precondition* of a function is a property that must hold prior to (and often during) the execution of that function.

Definition: Postcondition

A *postcondition* of a function is a property that must hold after invocation of the function is complete. Most postconditions apply only when a function returns normally, i.e., it does not exit via an exception.

Definition: Essential Behavior

The *essential behavior* of a function is the superset of the union of the postconditions of that function that includes what must be true of the behavior of the function and the external state of the program while the function is executing.

Essential behavior must be specified explicitly; in practice, however, many behaviors that are not specifically forbidden would be considered entirely unacceptable if they did occur. Examples might include printing a message, allocating and deallocating memory, and locking and unlocking a mutex when there is no rational reason to do so. Specific to each client and supplier is, clearly, a limit to the freedoms of QoI (quality of implementation) beyond which one can expect, contractually, a library to never go. For example, one generally assumes that a function can be trusted not to give away all of your money or commit a felony in your name. This unwritten agreement to stay within reasonably expected bounds is also a part of the complete function-level contract.

Definition: Contract Violation

A function-level *contract violation* is the failure of either the caller or, if the function is invoked properly, the callee to abide by the contract.

Upon invocation, of course, there is no universal guarantee that a contract will be satisfied. Whether due to a caller failing to satisfy a precondition or a callee failing to satisfy a postcondition (for a properly invoked function), any case in which the terms of function-level contract are not satisfied results in a contract violation. The purpose of a Contracts facility is to provide ways to detect contract violations so they can be remediated.

Definition: Natural-Language Contract

A *natural-language contract* is the formal specification in a spoken language, ideally common between the clients and the provider of a contract.

The role of a specification is to identify those salient aspects of behavior that are required for the function to be useful. Although every proper implementation of the function must satisfy the function-level contract, not every implementation will necessarily have language undefined behavior when the preconditions delineated in the function-level contract are violated. This observation gives rise to an implicit contract resulting from a particular implementation choice.

Definition: Implementation Contract

The *implementation contract* of a function is the implied set of preconditions under which that implementation will have well-defined (according to the implementation language) behavior, where the essential behavior is all observable aspects of the implementation's behavior.

Any input and state combination that would lead to the implementation contract being violated and the function-level contract being satisfied identifies a bug in the implementation. The implementation contract of a bug-free implementation is substitutable for the natural language contract. The implementation preconditions are no stronger than those of the natural language contract and,

within the domain of the natural language contract, the essential behavior of the implementation must be consistent with that required by the natural language contract.

Contract violations of the function-level contract, however, might not actually be contract violations of the implementation contract. In these cases, the behavior clients receive is well defined but is not the function implementer's *intended* behavior.

That users are violating a function-level contract is problematic, even if the the out-of-contract behavior seems to satisfy the caller's (unfounded) expectations. The provider's required preconditions might not be strictly necessary for a current implementation but often are required for future evolution of the software. Postconditions, similarly, guarantee certain behaviors but do not restrict future developments (and hopefully improvements) in the implementation of the function.

Realistically, however, many users of a function write software by experimentation rather than after thoroughly reading and carefully following the documentation. This aspect of human nature results in users depending on parts of the *implementation contract* that are, as far as the *natural-language contract* is concerned, *contract violations*. This aspect of human nature is often understood as *Hyrum's Law* [wright]:

> With a sufficient number of users of an API,
> it does not matter what you promise in the contract:
> all observable behaviors of your system
> will be depended on by somebody.

The gap between a function's contract and its implementation contract is often doomed by *Hyrum's Law* to shrink over time. To overcome this, numerous efforts have been undertaken to codify and enforce the *natural-language contract*. In practice, any case in which a client violates a function's contract but still obtains the intended behavior from the implementation contract is a bug, either current or latent.

Reducing this gap systematically requires communicating, in some form, the parts of the natural-language contract that are not intrinsic to the current implementation of the function. All efforts in this direction hinge on understanding the *contract check*. Recall that, poorly written, a contract check encodes an algorithm that, if evaluated at a well-defined point during the invocation of the function, could potentially identify one or more contract violations.

Definition: Contract-Checking Predicate

A *contract-checking predicate* is an expression that evaluates to a type that is contextually convertible to `bool` in which any result other than evaluating to `true`, such as evaluating to `false` or throwing, indicates that a contract violation has occurred. Note that a predicate whose evaluation has undefined behavior is also considered a contract violation, though often that might not be diagnosable at runtime.

The scope of perfectly plausible natural-language contract violations — such as that callers must have paid for a license — exceeds what could possibly be detected by a contract-checking predicate. Even within the software domain, certain preconditions, such as whether two pointer parameters are mutually reachable, might admit no suitable runtime contract-checking predicates. Moreover, essential behavior that cannot be expressed as a runtime post condition, such as not having locked

a resource, exceeds what can be diagnosed by any contracts facility currently under consideration. For now, we'll focus on those aspects of natural-language contracts that lend themselves to being enforced by contract-checking predicates.

3 Uses of Contract Checks

Prior to run time, i.e., during static analysis and compilation, any *contract-checking predicate* can be used to glean useful information about the expected state of a program. A predicate can be evaluated symbolically, and in cases where it can be concluded that the predicate will indicate a violation has occurred, appropriate action can be taken at compile time.

Static analysis tools typically use the knowledge of such violations to produce warnings to users that their software contains a logical bug. Compilers can use contract-checking predicates, if given permission to do so, to improve code generation for cases where the contract remains satisfied.

Most importantly though, a class of contract-checking predicates can be evaluated at *run time* to detect contract violations before they cascade into catastrophic problems. Consider a simple form of predicate that checks the bounds on an integral parameter[1]:

```
int clamp(int v, int low, int hi)
   [[ pre : low <= hi ]];
```

With a contract-checking predicate such as this, we can then define and produce (at least) two distinct builds of a program that invokes this function.[2]

Definition: Unchecked Build

A build of a program that does not consider any *contract-checking predicates* that may be in the source code is an *unchecked build*.

Definition: Checked Build

A *checked build* of a program involves contract-checking predicates being evaluated, and if they evaluate to false, the program is terminated, ideally with a helpful diagnostic.

These two builds will behave nearly identically for inputs that lead to no *contract violations*, i.e., if no bugs are encountered. The *checked build* will execute marginally more instructions, but beyond the excess heat generated and (possibly immeasurable) delay in producing results, the checked build will simply be better guarded against bugs.

Other predicates, however, may not have this same property of improving safety only by evaluating their contracts. Consider, for example, a function that works on an iterator range and requires at

[1] Because significant existing literature describes the syntax and its semantics, we will use the C++20 contract syntax for all concrete examples of contract checks. Note that nothing in this discussion is dependent on the specifics of that syntax.

[2] Note that there are many valid use cases for a much richer set of controls and distinctions than simply enforcing or ignoring all checks, observing violations (but not aborting), or assuming no violations as well as many variations on mixing these semantics intelligently for different contract checking statements in the same program. See, for example, [P1332R0] and [P1429R3].

least three elements be in that range:

```
template <typename ITER>
void processAtLeast3(ITER begin, ITER end)
  [[ pre : std::distance(begin,end) >= 3 ]];
```

A predicate such as this, when evaluated symbolically at compile time or during static analysis, is quite meaningful for all iterators. However, if `ITER` is an input iterator, the act of invoking `std::distance` on `begin` will consume all the elements in the range, such that the function itself never sees any contents of the ranges and will thus fail to satisfy its contract. Predicates of this form are clearly *unsafe* to evaluate at run time.

In other words, the act of observing whether a contract violation has occurred might alter the state of a program in such a way that a contract violation ensues. The invasive (Heisenberg-like) nature of such checks, if used, would clearly lead to reduced safety from runtime checking. In the next section, we will explore many variations on what forms of predicates could be considered for evaluation as runtime checks.

Definition: Safe Contract-Checking Predicate

A *safe contract-checking predicate* is a contract-checking predicate in which evaluating it does not cause the function to violate its natural-language contract.[a]

[a] Note that a predicate has the *ability* to do very bad things that don't appear to violate the explicitly stated natural-language contract, such as transferring all of one's money to the bank account of a randomly chosen teenager in Ohio. One can guess that the user of a library is working under the assumed contract that the library will not do such things, though this contract is not stated directly.

We hope to make clear that *there is no well-defined subset of predicates that are safe or unsafe*, and this distinction must be learned by engineers as they begin working with a system that supports encoding their contracts, just as has been the case with all preexisting uses of similar facilities, such as `<cassert>`.

4 Side Effects

Before delving into how to understand the sliding scale of safety between safe and unsafe contract-checking predicates, we must first have a common understanding of what it means for an expression to have side effects. Whether an expression has side effects is an important first qualifier when deciding how to reason about the safety of evaluating that expression at run time as a *contract-checking predicate*.

C++ has four forms of side effects:

1. Reading from a volatile object
2. Modifying any object
3. Invoking a library input/output function
4. Invoking any function that does one of the above

This definition leads to the most stringent categorization of contract-checking predicate.

Definition: Side-Effect-Free Expression

An expression that has no side effects within the C++ language is a *side-effect-free expression*.

Many simple expressions naturally fit in this category. Using a functional style, where no objects are ever modified after being initialized, allows for more extensive algorithms to be written as completely free of side effects. For practical reasons, C++ code is often written in an imperative style and thereby fails to be side-effect free.

A slightly broader kind of expression is one where none of the side effects of the expression are observable outside of that expression.

Definition: Pure Expression

A *pure expression* is one whose only side effects are modifications of nonvolatile objects whose respective lifetimes begin and end within the evaluation of the expression. This definition applies to postconditions but also to essential behavior *during* the evaluation of the function.[a]

[a] A number of subtly different uses of the word *pure* are commonly used in the literature related to this work, but this is the definition we have chosen for this paper.

Another way to understand such expressions is that they have no visible side effects outside of a *cone of evaluation* beginning at the expression and extending to all of the functions evaluated as part of that expression. This wider group of expressions allows for many algorithms to be applied to data to produce a result, but even these allowances are still highly limiting.

From a user's perspective, restricting contract-checking predicates to satisfy either definition above relating to side effects would be painfully limiting. Even identifying whether a given expression is pure or completely side-effect free can be challenging for expert programmers. What users care about is that, after a function is evaluated, nothing is left logically changed in the state of their application. This requirement is very similar to the guarantee users assume they have when invoking a `const` member function, even though such functions may occasionally cache values in mutable variables.

Definition: Logically Pure Expression

A *logically pure expression* is one which, after evaluation is complete, leaves *no* externally observable change in the state of any object. Note that this includes changes external to the process and that might have been accomplished through system calls.

This categorization includes but is not limited to heap allocations that are all freed, locking and unlocking a `mutex`, caching the results of computations in a `mutable` member for future reuse, and any number of other side effects that C++ types commonly encapsulate.

5 Forms of Expressible Contracts

Understanding that evaluating a contract-checking predicate at run time can itself introduce bugs is fundamental to writing safe contract predicates.

5.1 Simple, Pure Check

An expression that has no side effects outside of its cone of evaluation cannot alter the state of a program and result in violating any parts of a function's contract that depend on that state, leading to the strictest subset of contract-checking predicates.

Definition: Pure Contract-Checking Predicate

A *pure contract-checking predicate* is one that would have no observable side effects on the state of the program if it were evaluated.

As there is no observable side effect, a conforming C++ implementation is free to evaluate such checks at any point during run time under the *as-if* rule. One could also allow for such predicates to have side effects that apply only to objects created and destroyed during the evaluation of the predicate itself.

Pure contract checks can take many forms but in practice are limited to mathematical expressions on function parameters and global state:

```
double sqrt(double x) [[ pre : x >= 0 ]];
double clamp(const double v, const double low, const double hi)
  [[ pre : low <= hi ]]
  [[ post r : low <= r && r <= hi ]]
  [[ post r : (v < low && r == low) || (v >= hi && r == hi) || (v == r) ]]
```

On the other hand, even a pure predicate involves executing instructions that would not otherwise be executed. This evaluation generates heat, writes to memory on the stack (which might overflow), and adds to the run time of the program. In some cases, evaluating a pure check can alter the algorithmic performance of a function:

```
int* binarySearch(const std::vector<int> &v, int value)
  [[ pre : std::is_sorted(v.begin(), v.end()) ]];
    // Return a pointer to the element of v with the specified value, if
    // there is one, performing a number of computations logarithmic in the
    // size of v.
```

Though evaluating the precondition of `binarySearch` would not alter its ability to produce the correct response, the pure check would perform a number of operations that is linear in the size of `v`, violating the runtime guarantee of the natural-language contract of `binarySearch`. Here we see a *pure predicate* that is unsafe to evaluate.[3]

[3]Note that checks such as these might still be useful to evaluate in some environments where the inputs to be handled are small enough or time might be available to wait for excessively slow results. Historically, such checks that violate the runtime guarantees of a function without otherwise preventing a function from producing a correct result have been grouped as `audit`-level checks to be enabled primarily for testing in reduced environments.

In a similar fashion, *contract-checking predicates* are just as susceptible to bugs and undefined behavior as any other piece of code. Consider an alternative version of `sqrt` that uses an in-out parameter:

```
void inplaceSqrt(double *p)
    [[ pre : *p >= 0 ]]
    // Load into the specified p the square root of the value in *p.
{
    if (p) { *p = sqrt(*p); }
}
```

Evaluating the predicate, `*p >= 0`, has an implementation precondition that `p != nullptr` (which is not an implementation precondition of the actual implementation of `inplaceSqrt`). Enabling this precondition as a runtime check will elevate any case where `inplaceSqrt` was invoked with a `nullptr` value to be language UB (undefined behavior).

Even without overt undefined behavior in the contract-checking predicate itself, the simple act of evaluating the predicate might result in a stack overflow, thus introducing UB into a program just by enabling the check. So we can see that purity, on the surface, might indicate that a check is safe even when there are still real-world cases of pure contract-checking predicates whose evaluation would be unsafe.

5.2 Logically Pure Predicates

Many *contract-checking predicates* are *logically pure expressions*, i.e., from a user's perspective they have no side effects to be concerned with. As far as a user is generally concerned and as far as the correctness of a resulting program might be impacted, a function that modifies external state and then returns it to its original value is often functionally equivalent to one with no side effects.

5.2.1 Allocations

Consider allocations, which for any expression that properly employs RAII will be paired with associated deallocations prior to the expression completing:

```
void f(std::string_view key, std::map<std::string,std::string> &map)
    [[ pre : std::contains(map, key) ]];
```

Here we can see a *contract-checking predicate* that needs to create a temporary `std::string` object to invoke `std::contains`. For sufficiently large strings, this temporary will make an allocation to store the string data. The invocation of the system allocation and deallocation functions, `new` and `delete`, are observable side effects. In a highly resource-constrained system, allocations such as this might even lead to an allocation failure, making this same predicate unsafe in some environments.

5.2.2 Synchronization

Synchronization is often embedded in a type that is intended to be used concurrently. Understanding which methods will interact with such synchronization primitives requires breaking encapsulation and inspecting the contents of those methods:

```
class B {
private:
  bool       d_value;
  std::mutex d_mutex;
public:
  bool value() {
    std::lock_guard guard(d_mutex);
    return d_value;
  }
};
void foo(B& b) [[ pre : b.value() ]];
```

Here the `value` method will acquire a lock, read the data member `d_value`, and then release the lock. This non-const operation on the `std::mutex` member is an observable side effect that in a well-formed program designed to use types that guard themselves against concurrent access in this fashion, is typically innocuous.

Again, in some contexts, this form of side effect might introduce a bug. Should the lock in question be held by another thread indefinitely, this contract-checking predicate will never complete evaluating, potentially introducing a deadlock. This kind of problem is common to such improper use of synchronization irrespective of whether it arises from evaluating the contract-checking predicate.

More importantly, this form of synchronization is often completely encapsulated within an implementation file, so users of `B` might be completely unaware of the existence of a `mutex` that will be locked by invoking `value`. This encapsulation is a feature, not a bug, of C++, and the risk of unsafe operations in a contract-checking predicate such as this is taken on by any use of this form of concurrency.

5.2.3 Caching

Sometimes a value can be expensive to compute or perhaps even be not computable a second time. In such cases, a not-uncommon response is for an implementation of a class to lazily initialize a mutable data member only when the value is requested via a (`const`) accessor:

```
class ExtendedNumber {
private:
    long long                   d_value;
    mutable std::optional<bool> d_isPrime;
public:
    bool isPrime() const
    {
       if (!d_isPrime.has_value()) d_isPrime = NumericUtils::isPrime(d_value);
       return d_isPrime.value();
    }
};
void foo(const ExtendedNumber &value)
    [[ pre : value.isPrime() ]];
```

Here one can see the not-always-needed but expensive-to-compute primality of our `ExtendedNumber` type is computed on demand. That these values are cached is entirely encapsulated behind the

interface of `ExtendedNumber` when they are implemented correctly. Subtle changes in performance characteristics due to the earlier computation of the primality of a number might alter a program's behavior (perhaps even introducing bugs), but overall that seems highly unlikely.

The standard library itself is even mandating this behavior for certain types. Consider that the type `std::views::filter_view` from the Ranges library mandates that its `begin` function has amortized-constant-time performance and thus must cache the results of the first call in a mutable member.

The presence of such side effects are also exceedingly hard to detect for a human programmer. Consider a generic function that works on an arbitrary non-empty range:

```
template <typename R>
void someAlgorithm(R&& range)
    [[ pre : !std::ranges::empty(range) ]];
```

Regular use of such a function on, for example, a range of numbers generated by `std::views::iota`, makes the precondition of `someAlgorithm` a pure contract-checking predicate with constant time run time:

```
void foo()
{
    someAlgorithm(std::views::iota(0,17));    // contract check is pure
}
```

However, a more sophisticated use, say, involving ranges might seek to filter this range of numbers to include only even numbers using `std::views::filter`, which constructs a `std::views::filter_view` for us:

```
void foo2()
{
    auto even = [](int i) { return 0 == (i % 2); }
    someAlgorithm(std::views::iota(0,17)
           | std::views::filter(even));
}
```

The precondition of `someAlgorithm` will invoke `std::ranges::empty` on its argument, resulting in caching the value of `begin` in the `filter_view` that is that argument. Thus, we have a *logically pure* but not *pure* contract-checking predicate, depending on the template parameter. Here we can see that even identifying the extent of the side effects when using a modern C++ library can require deep knowledge of that library, losing the benefits of encapsulation that library was designed to provide.

Well-structured code built on top of encapsulation such as this, where abstractions hide side effects and provide sound components that are logically side-effect free, do not tend to be unsafe contract-checking predicates.

5.2.4 Reference Counting

Certain types that involve reference counting, such as `std::weak_ptr`, provide no way to use them productively without modifying their state. Any attempt to access the referred-to object, if it is

still live, must involve locking the `weak_ptr` and thus incrementing its reference count:

```
struct S {
  // ...
  bool ready() const;
  // ...
};
void nullOrReady(const std::weak_ptr<S> &wp)
{
  std::shared_ptr<S> sp = wp.lock();
  return sp ? sp->ready() : true;
}
void foo(const std::weak_ptr<S> &wp)
  [[ pre : nullOrReady(wp) ]];
```

The obvious side effect here is that the reference count for an object referred to by `wp` will be incremented and decremented during the evaluation of the precondition check for `foo`. If, at the end of the evaluation of the precondition check, this reference happens to be the sole remaining one, the destruction of that object will end up occurring as the contract predicate's evaluation completes.[4]

When using a shared ownership model responsibly, one must make no general assumptions about when exactly an owned object will be destroyed, so the impact of this contract-checking predicate on a properly implemented program should not be unsafe. In general usage, the only impact of a contract-checking predicate involving reference counting such as the one above will be a briefly incremented and immediately decremented reference count on the shared object.

5.3 Overt and Unrelated Side Effects

Some contract-checking predicates, by design or happenstance, end up with overtly observable side effects. One can legitimately ask if these kinds of predicates are ever safe.

5.3.1 `printf` Debugging

Often, during development, the simplest form of identifying whether a function is being invoked is to add a `printf` to it:

```
bool checkCondition()
{
    std::printf("checkCondition being called\n");
    return condition;
}
```

This form of debugging is quick and simple and is taught to developers during the first lessons of most software engineering classes. Supporting the continued effectiveness of this debugging method seems essential to keep C++ usable by real engineers throughout their careers.

[4] Note that, when checking a contract such as this related to the state of an object referenced by a `weak_ptr`, great care must be taken to be prepared for the case where the object is deleted between the evaluation of the contract-checking predicate and the body of the function.

Debugging via `printf` also introduces a clearly observable side effect on a program, that of writing a string to standard output. When used in a contract-checking predicate, this potentially unsafe expression makes the evaluation of that predicate clearly visible:

```
void someFunction() [[ pre : checkCondition() ]];
```

The author of `checkCondition` may be completely unaware that a contract-checking predicate elsewhere in their system is invoking their function. In fact, the desire to introduce this `printf` into `checkCondition` may be precisely because that author is exploring the scope of the uses of their function.

For most functions, a `printf` will also have no impact on the ability of that function to satisfy its natural-language contract. Functions that specifically output particular data to standard output will, of course, have that output corrupted, clearly making this a case where the safety of a contract predicate is *contextual*. Any pieces of software, however, that are producing structured output to standard output must, in general, carefully avoid any unwanted use of `std::printf` or `std::cout`, and this requirement applies to all code both inside and outside of contract-checking predicates.

5.3.2 Logging

Similar to a `printf`, many applications leverage a logging framework of some sort to write structured log messages to a central file from any function that needs to do so. The `printf` debugging above could have instead been accomplished using a logging facility:

```
bool checkCondition()
{
    BALL_LOG_TRACE << "checkCondition being called";
    return condition;
}
```

Outside of the logging system itself, a function that might use `checkCondition` as a precondition would be unlikely to also have, as part of its function-level contract, a promise that only very specific log messages are generated when the function is invoked. Logging facilities, along with tools that process and humans that read the resulting logs, must already be prepared to handle log messages written by a wide mix of different components within large applications. Therefore, functions that have trace logging added to them are even less likely to cause a contract-checking predicate that uses them to be unsafe than those that use a `printf`.

5.3.3 Metrics Tracking

As libraries evolve, they often develop a need to track how actively individual functions within that library are used. Such tracking can help measure the popularity of an API, track usage for billing, or to identify candidates for deprecation.

This tracking is, outside of the tracking system itself, invariably going to have no effect on the essential behavior of the functions that make use of the tracked functions, either in their implementations or in contract-checking predicates:

```
bool checkLibraryState() {
  // audit how often this function is called
```

```
  MetricsTracker::trackUsage("checkLibraryState");
  return libraryState;
}
void clientFunction()
  [[ pre : checkLibraryState() ]];
```

The client in this case, when writing `clientFunction`, has no need to know about and will not be adversely effected by the side effect within `checkLibraryState`.

5.4 Destructive Predicates

Finally we arrive at a class of function predicates that are often categorically unsafe:

Definition: Destructive Contract-Checking Predicate

A contract-checking predicate that, if evaluated at run time, will prevent the associated function from satisfying its function-level contract is a *destructive contract-checking predicate*.

These predicates express the *capability* to perform an action, often because the function itself is actually going to perform the action; however, no external method is present to identify the *validity* of that action. Destructive predicates should be carefully avoided in a runtime contract-checking system but can be powerful in what they can express to a static analyzer or compiler.

5.4.1 Input Iterators

Input iterators are a common pitfall when writing contract-checking predicates on generic functions that work on a range of iterators. Where a forward or random-access iterator will allow repeated iteration, an input iterator lets you advance it only once through the range of values it exposes.

Let's revisit the `processAtLeast3` function we introduced earlier:

```
template <typename ITER>
void processAtLeast3(ITER begin, ITER end)
  [[ pre : std::distance(begin,end) >= 3 ]];
```

Given a random-access iterator, this precondition is likely safe. For a forward iterator, the precondition might have unacceptable runtime performance but otherwise might be auditable. For an input iterator, however, evaluating the precondition will consume all of the values in the range, preventing the function body itself from processing those values.

5.4.2 Reachability

Destructive predicates can also express requirements that the language itself provides no well-defined facility to detect. Consider that, when given a pair of iterators, one has no way to determine definitively if they represent the beginning and end of a range. What the language does provide is the ability to attempt to iterate over the specified range:

```
template <typename ITER>
bool isForwardReachable(ITER it1, ITER it2)
{
```

```
  while (it1 != it2) { ++it1; }
  return true;
}
```

This function that otherwise does nothing will eventually return `true` in all cases where `it2` is reachable from `it1`. If that is not the case, this function will have undefined behavior; either the loop will never terminate, or `it1` will iterate past the end of the sequence it does point to.

At run time, this undefined behavior is not viable, and more importantly, a predicate like this cannot reliably detect a violation and abort when a bug is detected. During static analysis, however, reachability can be tracked through symbolic execution, and the requirement expressed by this predicate can be checked, leading to warnings when unreachable pairs are passed to a function that employs `isForwardReachable` in a precondition.

5.4.3 External Side Effects

Occasionally, systems are deployed where a single process is just a small part of a larger fullly encapsulated system. This can range from local files being used to cache large data structures to databases where structured data is accessed through remote APIs.[5] In all these cases, some operations might be performed to interact with — generally reading and writing data — these external components. Complicated workflows might have preconditions that the external component is in a certain state and postconditions that declare that they have made certain updates to the external state.

by Peter Brett, though that example expressed a strong need for preconditions and postconditions involving verifying the state of a database, while our example shows similar requirements on something external while using only the tools available in the standard library.

Consider a function that tests whether the last bytes written to a file are a particular string. A function such as this has overt side effects making system calls and updating the file counters. This very non-pure predicate still reverts all state changes when complete, making it a useful tool for verifying the contents of a file:

```
boolean atEndOfFile(std::fstream& stream, const std::string_view& expected)
{
    std::fstream::pos_type pos = stream.tellg();
    if (pos == -1 || pos < expected.size()) { return false; }
    stream.seekg(-expected.size(), std::fstream::seekdir::cur);
    std::string tail(std::istreambuf_iterator<char>(steam),
                     std::istreambuf_iterator<char>());
    if (tail != expected) { return false; }

    return true;
}

void appendToFile(std::fstream& stream, const std::string_view& tail)
  [[ post : atEndOfFIle(stream, tail) ]];
```

[5] This example was inspired by an example shared with the community

One could easily imagine a similar function to verify that a row has been inserted into a database or that a message has been sent to a remote server. A database read might even have more unrelated side effects that do not get undone after evaluation, such as a log of remote connections or queries that have been run.

Within the context of these larger systems, components that are separate from a single running task might still have preconditions on what state they are in before performing certain operations and might guarantee certain states after being evaluated.

6 Conclusion

A *function-level contract* delineates (in a natural language) a binding agreement between the provider of a function (callee) and its clients (callers). This contract encompasses all of the *preconditions* and *essential behavior* of its associated function. A contract check encodes an algorithm that, if run at an appropriate time during the invocation of a function, can detect if some aspect of the natural language contract is violated. Not all preconditions and essential behavior can be validated via runtime contract checks.

Of those terms that can be checked, not all of them can necessarily be checked without affecting observable behavior. In some cases, the effect on behavior may interfere with the function's ability to satisfy all the terms of its contract, thereby making a program incorrect simply by being evaluated. If and to what extent a contract-checking predicate might affect a function's ability to fulfill its contract defines the degree of *safety* associated with that predicate.

As we have demonstrated, virtually any useful contract-checking predicate, if evaluated, has the potential to cause a program to behave differently, exhibit undefined behavior, or fail outright. The more prominent or aggressive the side effect, the more likely such behavior is to interdict a function's ability to fulfill its natural-language contract. Moreover, such side effects are subjective as well as both domain and context dependent; hence, no compiler algorithm can determine objectively if and when a predicate is unsafe for use.

The absence of side effects is unnecessary and insufficient for a contract-checking predicate to be considered safe or appropriate for use. The subjective determination — like most aspects of programming, such as debugging itself — is a skill that is learned through practicing the trade.

Given this foundation, one clearly sees that, when enabling runtime checking of contracts, a universal concern is understanding if and to what extent a contract predicate would be safe in a given context. The degree of safety in evaluating a contract predicate at run time depends solely on how likely that evaluation is to break the function's ability to satisfy its own contract. Any predicate that is apparently safe in one function has the possibility to be unsafe in a different context. No simple rule would enable the useful contract-checking predicates we have described while preventing, within the language, any unsafe contract checks from being written.

Writing a contract check that does not impede a function's ability to execute properly has, in our experience, been generally intuitive and easy to teach. When a mistake happens, clear semantics for how a contract predicate evaluates make debugging and fixing the impacted function relatively straightforward.

Though destructive contract predicates, when evaluated, usually result in defects, such checks have historically been handled by supporting markup on a contract check to indicate that the check should *never* be evaluated at run time.

These sorts of checks, however, are the exception, not the rule, and are not part of a runtime-only contract-checking facility. Since using such compile-time-only checks might be considered an expert-only endeavor, they need not be considered as part of an MVP.

Rather than attempting to restrict contract predicates to a subset of the language that might appear to allow only safe contract predicates, we would instead encourage teaching a basic understanding of how to evaluate what is likely to be safe to do in a contract-checking predicate, thereby avoiding the need to define, standardize, and teach a "safe" (and far less effective) subset of the language for use within contract-checking predicates.

A Bibliography

[P1332R0] Joshua Berne, Nathan Burgers, Hyman Rosen, John Lakos, "Contract Checking in C++: A (long-term) Road Map", 2018 http://wg21.link/P1332R0

[P1429R3] Joshua Berne, John Lakos, "Contracts That Work", 2019 http://wg21.link/P1429R3

[wright] Hyrum Wright, "Hyrum's Law" http://hyrumslaw.com