Static Analysis of Contracts with P2900

Abstract

Some have suggested that P2900 is unsuitable for static analysis and that we must, therefore, pursue extensive changes to the proposal (if not the language itself) to achieve some perceived level of increased safety when using Contracts. In this paper, we explore some of those suggestions, offer some manifestly better alternatives that can be pursued in the future, and present real-world experience showing that static analysis is greatly benefited by the features inherent in [P2900R9] contract assertions.

Contents

Revision History

Revision 1 (Pre-Wrocław November 2024 meeting)

- Numerous cleanups and clarifications

- More direct conclusion

Revision 0 (October 2024 Mailing)

- Original version of the paper for discussion by EWG

1   Introduction

Contract assertions, as developed collectively by SG21 and detailed in [P2900R9], capture algorithms whose purpose is to identify when a program is correct. The C++ Standard, whose aim is to specify how C++ programs are translated and what happens when they are evaluated, must necessarily concern itself with the behavior contract assertions will have when evaluated and the various interactions they will have, at compile time, with other parts of the language. This need leads to a predominant focus on the use of contract assertions for *runtime* checking of correctness since that specification provides the foundation on which all other uses of Contracts will be built.

Static analysis is, however, another major use case for contract assertions that does not involve evaluating them at run time. Identifying at run time when a contract assertion is violated is critical to improving program correctness; identifying such violations at compile time is better, i.e., more immediate and, therefore, cost effective. Importantly, static analysis can identify edge cases that are rarely seen in production and that are missed by testing and can thereby allow bugs to be fixed even sooner.

Static analysis of this sort has been employed on C++ code for ages. Modern C++ compilers do copious amounts of static analysis at compile time to both improve code generation and produce diagnostics. Compilers are already aware of some current macro-based assertion facilities. Other tools perform similar and deeper analysis identifying violations of the preconditions of the core language itself. Finally, existing static-analysis tools have explored identifying violations of macro-based contract-checking facilities in multiple and varying ways.

In this paper, we will elaborate on a few points related to static analysis.

- We will give an overview of what mental models of contract assertions should be applied when analyzing a C++ program that contains them.

- We will describe our experience with applying existing static-analysis tools to codebases that contain macro-based contract-assertion facilities and how those tools will apply even more effectively in the presence of a language-based Contracts facility.

- We will consider whether static analysis would benefit from putting further limitations on what can be placed in a contract assertion or changing the behavior of the language when they occur during the evaluation of a contract-assertion predicate.

We aim to clearly explain, in this paper, that contract assertions as presented in [P2900R9] provide a powerful tool for enabling static analysis to improve the safety and correctness of all software to

which it is applied.

2 What Is Static Analysis?

A correct program is one that achieves its programmer's intent. Hence, by this definition, every programmer aspires to write correct programs.

Ideally, a well-reasoned programming language will make writing correct programs easy and incorrect ones less so. On the other hand, because so much of a programmer's intent is *not* captured in the code, a programming language cannot guarantee that only correct programs can be written.

The Contracts facility offers a large step toward bridging that gap via the introduction of a mechanism whose primary intent is to identify which behaviors of a program are correct. Many of the complexities of [P2900R9] focus on handling how the impact of running an incorrect program can be mitigated — i.e., through invocation of a contract-violation handler or terminating more aggressively.

Static analysis is our general term for any process that attempts to identify, without executing a given program, whether that program will be correct. Any given static-analysis procedure will have certain different properties, many of which interact with Contracts and are employed by users in wildly different ways.

- The question of correctness can be answered *globally* for a complete program on a specific platform or *locally* for just part of a program. In general, any analysis will necessarily have to make assumptions about behaviors at the boundaries of what is being inspected, such as functions not in the same translation unit, meeting their advertised contracts.

  - Even a supposedly global analysis will still be making *some* assumptions that the platform on which the program will run will itself behave correctly.

  - While global analysis might be viable for small programs or for those with unbounded resources ready to devote to verifying program correctness, any useful tool will require the ability to perform local analysis and chain together reasoning when composing a program from individual translation units.

  - The ability to annotate function *declarations* (not just *definitions*) with contract checks is essential to performing local analysis without needing to see the full definition of every function under analysis.

- Any analysis will produce information about whether the program being analyzed will be correct for certain hypothetical inputs. This analysis might produce a *false positive* by identifying as incorrect a case that is actually correct or a *false negative* by failing to identify incorrect programs.

  - Any analysis that might produce false positives demands a corresponding escape hatch to suppress those reports once manual analysis has proven the report to be false. On the other hand, such escape hatches are brittle and are not often revisited to ensure that unrelated changes have not turned them into real bugs now hidden by the escape hatch.

– An analysis with no false negatives cannot be considered *proof* that a particular program has no defects of the form for which the analysis is searching because defects outside the scope of the analysis might always cascade into a violation of that proof.

- Analysis can be done *portably*, aiming to determine if a program is correct with respect to any potential implementation of the C++ abstract machine, or *platform specifically*, considering the parameters of a specific machine. Even an analysis aiming for portability might make some assumptions, such as the sizes of various built-in types, that restricts the results to commonly available modern hardware and operating systems.

    – The least portable analysis might inspect a single generated program binary, determining correctness for only that singular build configuration. While useful as a final check before deployment of a risk-averse system, this form of analysis is least useful for those writing source code.

    – The other extreme would identify correctness for source code that is independent of platform, any preprocessor directives, or even compiler flags. This form of analysis can be powerful due to the force-multiplying effect of inspecting one set of source code and producing reliable results for all the programs that might be built on top of that code.

- When analyzing a program containing contract assertions, an important new variation must be considered: what implementation-defined choice of contract-assertion behavior has been selected. An analysis might be *semantic independent* and aim to prove a program correct independently of the chosen semantics of the contract assertions evaluated, or it might be *build specific*, producing a result for only a single build configuration.

    – As described in [P2900R9], as long as a contract assertion is not *destructive*, the correctness of a program evaluating it is independent of whether it is checked (by definition). Any static analysis might, therefore, assume checks are not destructive and then proceed to be semantic independent. A separate analysis of just the contract-assertion's predicate can then be done to identify potentially destructive predicates.

    – For a platform providing S different semantics with which contract assertions can be compiled and a program in which code is generated for contract assertions in N different locations, a semantic-independent analysis says something useful about S^N different real programs that could be built. This vast number of programs includes, *importantly*, the program in which all contract assertions have the *ignore* semantic and the full range of programs in which all contract assertions are *enforced*.

    – A build-specific analysis might still consider every single contract assertion's correctness and do so with the knowledge that some will be checked and some will not be checked in the final build. Knowing when an unchecked contract assertion would still be violated if checked is an important concern for determining when a program is correct, in particular when doing a build-specific analysis of the build that ignores all contract assertions.

- The analysis of contract assertions also need not be applicable to *all* contract assertions. A tool might be selective in determining which contract assertions it understands.

    – Tools that wish to track the state of *shadow variables* to do range analysis or similar verifications might limit the contract assertions they consider to only those that perform

simple operations on built-in types, e.g., integer arithmetic and comparisons, pointer comparisons, and so on.

– A tool might extend such analysis to function invocations where it can see the definition of the function.

– Opaque functions invoked by a contract-assertion predicate can be assumed to be independent of the values being tracked. This form of purity is contextual and can be separately proven once the definitions of a function in a contract assertion are visible.

– When, in the future, the ability to specify contract assertions that are *never* evaluated at run time is introduced, static analysis can consider pseudo-predicates that are bespoke to that analysis and do not have runtime-implementable equivalents. See [P1728R0] for more discussion on such possibilities and [P2755R1] Section 2.1.14 for how we might build support for this approach on top of [P2900R9].

- Static analysis also exists in a variety of contexts.

– Third-party tools can be applied at all scales to produce reports that are processed asynchronously to the normal programming workflow and that result in improved software.

– Analysis tools can be integrated into an IDE, heavily emphasizing those forms of analysis that can be computed in real-time as programmers make edits to their software.

– Within a compiler, significant amounts of static analysis is performed to optimize a program and produce warnings about commonly suspect behavior.

- Finally, no tool can prove the complete correctness of an arbitrary program.

– Any tool will be considering certain properties and attempting to validate that a program locally or globally satisfies those properties and *only* those properties. All assumptions a tool makes will be in relation to those properties.

– Tools might be verifying memory safety, lack of violation of certain contract assertions or any contract assertions, or satisfaction of specific requirements, such as not making use of the heap.

With all the above variations, static analysis is clearly a broad space with many different constituents, goals, requirements, and capabilities. Contract assertions can help advance the capabilities of almost all such tools.

3 Existing Static-Analysis Practice with Assertions

Given the long history of the C `assert` macro, many static-analysis tools already apply logic to assertions with the understanding that they are defensive checks of correctness.

- In **JetBrains**’ **CLion**’s DFA (Data Flow Analysis), assert macros are used and assumed to be true, resulting in warnings based on the information gathered from those assertions. For example, in the code below an assertion’s presence will be used to produce a warning on the `if` statement:

```
#include <cassert>

void f(int p) {
  assert(p > 7);

  if (p == 6)  // CLion displays: "Condition is always false".
    ;
}
```

Along with the warning comes tools to remove the unnecessary `if` statement entirely.

- **Synopsis**, in their static-analysis tools, also uses the C assert macro and some user-defined macros for which they have implemented bespoke support. Under normal analysis, they assume that assertions are true and use that information to produce better diagnostics. They also provide a checker to analyze the expressions used within assertions to verify that they do not have side effects.

- **PVS-Studio** also uses assert macros when performing static analysis. A key example of where this helps their users is in the removal of false positives by asserting information that the tool itself might otherwise warn as potentially erroneous:

  ```
  #include <cassert>

  //V_ASSERT_CONTRACT // <=

  struct Base
  {
    virtual ~Base() = default;
    int i;
  };
  struct Derived : Base
  {
    float f;
  };

  void foo(Base *p)
  {
    auto q = dynamic_cast<Derived *>(p);
    assert(q != nullptr);
    (void)q->f;  // no warning
  }
  ```

  Due to the presence of the assertion, a false positive is removed, and the debug build, when deployed, will help verify that the assertion itself is sound.

- Within Bloomberg, we have worked with **CodeQL** to integrate **Z3** from Microsoft to do code flow and range analysis to validate contract-assertion correctness. This process begins with a tool that builds a "spec database" of information about the contract assertions (expressed using the `BSLS_ASSERT` family of macros[1]) that exist inside the implementations of functions

---

[1]See [bde14].

within a library. Once that database is available, analysis of users of that library verifies that assertions of the forms understood by the analysis will not fire.

Primarily this inspection involves a range analysis applied to integers and pointers and then run through Z3 to identify potential violations.

- The **Clang** family of compilers understands a special attribute, `analyzer_noreturn`, whose primary purpose is to be placed within assertion macros to indicate that their violation-handling functions should be treated as `[[noreturn]]` when doing static analysis. This annotation allows analyzers to discard branches in which an assertion would fire and thus to reduce the false positives that might result from analyzing paths that are intentionally not supported.

Based on the experiences with the above tools, the overwhelming needs expressed for static analysis fall into a few categories:

- The limitations of a macro-based facility — in particular, that, after preprocessing, no evidence remains that an assertion is present — make it difficult for static-analysis tools to leverage assertion macros in all builds.

- Being unable to have a function's preconditions and postconditions visible to callers that see only the function's declaration is highly limiting to scalability. This lack of visibility led to the building of a bespoke tool to gather spec databases in our work with CodeQL. Importantly, other tools often fail to leverage assertions that are not somehow present in the header, e.g., in the bodies of inline functions. Forcing users to make a function `inline` just to get static analysis is highly suboptimal and problematic in practice, especially at scale.

- Finally, understanding arbitrary assertion macros is beyond the capability of static analysis. Many provide ways for a macro-based facility to integrate with certain static-analysis tools, such as Clang's `__attribute__((analyzer_noreturn))`, but the need for such collaboration between the static-analysis tools and providers of assertion macros can severely hinder the use of both. More feature-rich assertion macros that provide advanced configuration and flexibility in the choice of mitigation strategies only make implicit integration harder for new tools to provide.

Amazingly, all the concerns above completely go away with [P2900R9]. Contract assertions are not macros, are always present, can be placed on function declarations visible to all translation units, and will be standardized.

4 Unnecessary MVP Modifications for Static Analysis

Changes to the Contracts MVP could be considered to facilitate even more static analysis. Of course, any such change must be measured in terms of how much it limits a user's ability to express checks of the correctness of their programs and which forms of static analysis (of the many listed earlier) will concretely benefit from such a change.

4.1 Defining UB within Contract Assertions

One design goal proposed in [P3285R0] was to prevent or define as much undefined behavior as possible during the evaluation of a contract-assertion predicate. The claim is made in [P3362R0]

that this removal is necessary and sufficient to achieve a proof of correctness using static analysis.

Consider first the example presented in [P3285R0]:

```
int f(int a) { return a + 100; }
int g(int a) pre(f(a) > a);
```

When `f` is inlined into the code generated for the precondition of `g`, it gets optimized to the value `true`. Obviously, no well-defined cases exist where, for an `int a`, `a+100 <= a` is not `true`. Of course, should `a+100` overflow, this precondition will have undefined behavior. On a machine with signed integer arithmetic wraps (i.e., where `int` actually models $\mathbb{Z}_{2^{32}}$), passing an argument sufficiently close to `INT_MAX`, i.e., `a > INT_MAX - 100`, can result in the body of `g` evaluating `f` and getting a negative number.

The temptation arises to specify that, within a contract assertion, this undefined behavior does not happen and instead integer arithmetic will wrap. This specification would appear to solve the problem for this example, since the precondition check will then fail and `g` will be guarded against cases where somehow adding a positive number to its input produces a negative number. Nothing in the domains of these particular example functions will be mistreated by this approach. As we will see, we do not have to search far to find cases where wrapping arithmetic (or any fully defined algebra for integers of finite size) will result in massive and hard to reason about holes in the efficacy of our contract assertions.

To see that, let's consider some of the inherent real-world consequences of defining signed integers to perform wrapping arithmetic when evaluated within a contract-assertion predicate. By looking at a (slightly) more involved example of an integer-arithmetic-based precondition one might want to express, we can begin to see the problems in this approach. Consider a function that takes as input three positive integers, which must sum to a value less than 1000:

```
void f(int x, int y, int z)
  pre( x > 0 && y > 0 && z > 0 )
  pre( x + y + z < 1000 );
```

With normal C++ `int` arithmetic, the above expressions are well defined for only exactly the intended values — `x`, `y`, and `z` in the closed range $[1, 998]$ and where their sum is in the range $[3, 1000]$. Should these values be used to index into an array or as bounds for an iteration, being in the intended range is important — i.e., to prevent security vulnerabilities in addition to behaving correctly.

When a static analyzer sees the expressions above and knows that integer overflow is undefined behavior, it can come to significantly better conclusions about the expected bounds being demanded of the arguments to `f` and, even more importantly, can then produce diagnostics on any values that might be outside those ranges. In particular, given that the first preconditions restrict all the arguments to be positive values, the addition operation combined with a comparison can lead to an immediate reduction of that range to one that is comparatively small and even independent of the other arguments. Without needing to evaluate the addition, for example, any call where `x`, `y`, or `z` might be greater than 998 can be immediately flagged as problematic.

Now consider what values will satisfy the above preconditions if we instead define integer arithmetic as wraparound arithmetic, i.e., if we make integers model $\mathbb{Z}_{2^{32}}$ instead of modeling $\mathbb{Z}$. In such a case,

both static analysis and a user must now consider a vastly wider set of values that might satisfy the condition `x + y + z < 1000`. Just as an example, the call `f(1431655932, 1431655932, 1431655932)` would satisfy the above preconditions, though a child will easily tell you it shouldn't and would need to learn a significant amount of advanced math to understand why it might. If we cannot expect our computers to do better than our first graders at basic math, we have failed on a fundamental level.

Any other model in which we alter the actual algebra that is modeled by `int` will have an equally surprising and disastrous impact on the result produced by surprisingly simple contract-assertion predicates.

Of course, one might look at the above call and claim that leaving matters alone would instead make the contract-assertion evaluation itself have undefined behavior. This design, however, has served C++ fairly well since inception and comes with an important and often forgotten benefit: Undefined behavior can itself be made into any behavior, including detecting it as a contract violation. This entire idea is explored more thoroughly in [P3100R1].

4.2 Removing All Side Effects from Contract Assertions

Now consider another proposal from [P3285R0], which is to render impossible having side effects in a contract-assertion predicate outside its cone of evaluation.[2]

First, a property of this sort — that side effects do not happen outside the cone of evaluation of a function invocation — is not a *recursive* property. Consider, for example, the following function:

```
bool somethingAboutValuesBetween(int a, int b)
{
  if (a > b) { std::swap(a,b); }

  // Compute a value knowing that a <= b.
}
```

Nothing about the above function is even passed a reference to something outside its cone of evaluation. On the other hand, `swap` unconditionally modifies functions outside *its* cone of evaluation. To truly identify whether a function has this property, one must analyze the the entire definition (recursively) of every function invoked that is not somehow *a priori* known to have this property, or one must be restricted to avoiding most forms of idiomatic C++.

The situation gets worse for member functions. If we allow non-`const` member functions or `const` member functions that might modify a mutable member, some mechanism must be available to prevent invoking such functions on objects outside the cone of evaluation of the contract-assertion predicate. To provide such a mechanism, significant analysis would need to be done within a contract-assertion predicate to track the flow of values. Consider, for example, the various preconditions we might have for a free function, `g`:

```
struct S {
  mutable int d_x;
```

---

[2]Note that at this point we will not go into the insufficiency of limiting contract-assertion predicates to allow no side effects since that would preclude the ability to allocate, acquire locks, or trace function calls with logging from within any function invoked from a contract-assertion predicate. See [P1670R0] and [P2712R0] for more discussions on this topic.

```
  bool increment() const { ++d_x; return true;}
};

S* f(const S& s) {
  return const_cast<S*>(&s);
}

void g(const S& s)
  pre( s.increment() )       // probably outside cone of evaluation
  pre( (&s)->increment() )   // maybe outside cone of evaluation?
  pre( S().increment() )     // inside the cone of evaluation!
  pre( f(S{})->increment() ) // unknown without definition of f
  pre( [](
    S a;
    return ((S{}.increment()) ? &a : &s)->increment();  // getting harder
  )
}
```

Clearly, none of the above examples are good code, but to guarantee a lack of modifications, a mechanism to track values and disallow invalid ones must be fully specified that allows useful code to be written without ever allowing a modification outside the cone of evaluation. Deeper analysis indicates that such proof essentially precludes the ability to use a member function that modifies any members of an object.

C++ permits `const` member functions to mutate the state of any members marked `mutable`, and any other objects by following non-`const` pointers and references. Hence, even `const` member functions, unless known not to mutate their state, would be off limits unless the definition of the function (and all functions called by it) were also visible to the calling compiler and were willing to commit to maintaining this property indefinitely. Without the ability to use member functions, contract assertions reduce to only the most trivial expressions in terms of built-in types. Such restrictions would prevent contract-assertion predicates from taking advantage of the ability of C++ to encapsulate and leverage abstraction, making the Contracts facility nothing more than a toy for academics and useless to most real-world practitioners.

4.3 Evaluating Symbolic Predicates at Run Time

Another suggestion from [P3285R0] is to enforce object lifetime safety by requiring that a predicate `object_address` "holds" for a pointer whenever a pointer is used within a contract assertion or a conveyor function — i.e., the special kind of functions that are the only kind of functions usable from within nonrelaxed contract assertions as proposed by [P3285R0]. The problem here, however, is that a contract assertion states something about a single point in time and does nothing to guarantee any *persistence* or *stability* to the results they are asserting.

Now, such a predicate could have its value inserted by a compiler in many places where its result can be determined statically. Consider, for example, a function having a precondition it is passed a pointer to a live object; to have such a precondition the predicate of the precondition will invoke the *symbolic* function `object_address`:

```
void f(int* p) pre( object_address(p) );
```

As proposed in [P3285R0], when used at run time, this function will conservatively determine if the pointer passed to it actually denotes an object within its lifetime.[3] In particular, if the compiler cannot prove that a pointer will always point to an object within its lifetime when the predicate is evaluated and, more importantly, when compiling the translation unit that contains the contract-assertion evaluation that invokes `object_address`, then it must instead evaluate to `false`.

For simple cases in which an object is passed directly from an automatic variable whose address has never escaped, this might be fruitful:

```
template <typename T>
void g1()
{
  T x{};
  f(&x);  // precondition violation
}
```

Above, we can see that `x` could not have had its lifetime ended prior to the invocation of `f`, so the precondition of `f` will not detect a violation. Now, however, consider the case where we call `f` *two* times:

```
template <typename T>
void g2()
{
  T x{};
  f(&x);  // Precondition passes.
  f(&x);  // precondition violation
}
```

Clearly, the precondition check on the first call to `f` can be determined to pass. The result of the check of the same precondition assertion on the second call to `f`, however, is dependent on what the first call to `f` does. Suppose that the implementation of the function `f` sometimes ends the lifetime of its argument, and other times it doesn't:

```
template <typename T>
void f(T* p) pre( object_address(p) )
{
  static int i = 0;
  if ((++i % 2) == 0)
  {
    p->~T();
  }
  else {
    new (p) T{};
  }
}
```

With this definition of `f`, the above function `g` has perfectly well-defined behavior, but the second call to `f` is made with a pointer to an object outside of its lifetime. Given that `f` *might* be defined in

---

[3]It has been suggested that the intention of [P3285R0] is that any use of `object_address` for which the result will not be `true` will be ill-formed; i.e., it will fail to compile. With that understanding of the design, all the examples, discussed in this section, in which a contract violation is indicated would be ill-formed and even more unusable.

this manner, a conservative definition of `object_address` must fail on the second call to `f`, making `g` totally unusable. Even if we see this definition of `f`, we still lack enough information to make `g` usable since no information is available about how many times `f` has previously been invoked.

This same issue arises when an opaque function is invoked and pointers — including any pointer that was supplied to the function by its caller — for which `object_address` might *hold* are present but their values might have escaped the function. Consider another case in which we try to call `f` twice, with two different pointers:

```
template <typename T>
void h(T* p1, T* p2)
  pre( object_address(p1) )
  pre( object_address(p2) )
{
  f(p1);  // Precondition passes.
  f(p2);  // precondition violation
}
```

Again, the first call to `f` might get past its precondition given the precondition of `h`. The second call to `f`, however, has no way to know that the first call to `f` did not destroy or delete the object denoted by `p2`, so the second call to `f` will violate its precondition.

pointer to an object within its lifetime:

Even if we were to consider relaxing the requirements, we can clearly see how easily an assertion can be made that a pointer is valid at one point and quietly destroy the object that pointer references before anything can be done usefully with that assertion. Consider functions where our goal is to produce a pointer to an object and then pass it to another function, `consume`, whose precondition is that it is given a pointer for which `object_address` holds:

```
template <typename T>
void consume(T* t) pre( object_address(t) );
```

Now consider a `produce1` function that guarantees that it *returns* a pointer to an object within its lifetime:

```
template <typename T>
T* produce1()
  post(r : object_address(r) );
```

Caller-side postcondition checking can clearly never know if the body satisfies this postcondition, but callee-side checking might. When invoking a function, such as `produce1`, that has no parameters, nothing else is evaluated between the call to `produce1` and whatever comes after, and thus the validity of the object carries forward into the body of a function invoking `produce`:

```
template <typename T>
void produce_and_consume1()
{
  T* p = produce1();  // Postcondition might pass if checked in callee.
  consume(p);         // Precondition holds and can pass.
}
```

On the other hand, consider a different producer function, `produce2`, with just one trivial argument:

```
int* produce2(int i)
  post(r : object_address(r) );
```

The caller of `produce2` cannot assume that the definition of `produce2` is not one like this that returns a pointer to its function parameter:

```
int* produce2(int i)
  post(r : object_address(r) )
{
  return &i;  // At least this will warn you, but it's well formed.
}
```

The postcondition of the `produce2` function, above, is not violated because its parameter, `i`, is still within its own lifetime when the postcondition is evaluated. On the other hand, as soon as control returns to the caller of `produce2`, the lifetime of the `i` object ends, causing the address value returned to spontaneously cease to satisfy the condition when control continues further:

```
void produce_and_consume2(int i)
{
  int * p = produce(i);  // OK, postcondition passes (checked in definition only).
  consume(i);            // Error, precondition violation
}
```

Similarly, consider another variant production function, `produce3`, that has a function parameter with a nontrivial destructor that, if called, might invalidate *any* pointer returned from `produce` regardless of its type:

```
struct S {
  ~S();  // in another translation unit
};
int* produce3(S s)
  post(r : object_address(r) );
```

No assumption can be made that `~S()` does not free the dynamically allocated integer object returned by `produce`:

```
void produce_and_consume3()
{
  int* p = produce3(S{});  // OK, postcondition passes (checked in definition only).
  consume(p);              // Error, precondition violation
}
```

Again, we are left with a postcondition that does not help and functions that are uncallable in normal usage due to attempts to enforce a symbolic predicate whose true answer cannot be reliably determined.

A compiler could, in theory, analyze much deeper into the definitions of functions and determine, in some additional cases, that `object_address` holds slightly further than the semantics of local analysis could conclude. Having a built-in language feature depend so strongly on quality of implementation, however, would be an awful user experience and would be untenable to use at scale.

An external static-analysis tool can look at assertions that use symbolic predicates like `object_address` and even supplement them with arbitrary extra assumptions — in particular, that functions won't invalidate pointers with which they have no direct interaction. Although an external tool can be guided into making such additional assumptions, e.g., based on local design or coding standards, they would not apply ubiquitously and, hence, could not reasonably be codified in the Standard itself.

In general, a predicate like `object_address`, which cannot give accurate results without massive instrumentation, is far more appropriate for *unchecked* contract assertions that are never evaluated at run time. Such assertions, explored in [P1728R0] and parts of [P2755R1], can be stated explicitly by programmers and then independently verified, as appropriate, by auxiliary tools that understand these non–runtime-checkable assertions. We note that such external verification tools will have far more flexibility for false positives and false negatives than the compiler itself and will typically have a much wider range of freedom with respect to QoI.

On the other hand, having the semantics of a standard function, such as `object_address`, that must instead be used at run time would be so dependent on the compiler's quality of implementation as to render the entire system unacceptably fragile and thus unusable in practice.

4.4 Restricting Contract Assertions to Marked Functions

Another suggestion from [P3285R0] is to restrict contract-assertion predicates to a set of functions that have been specifically marked to be usable in that context, which will then follow special rules that might reduce the chance of writing destructive contract-assertion predicates. In particular, the rule suggested in [P3285R0] was to require that all functions used be marked with a new attribute, `[[conveyor]]`, that imposed new, well-defined meaning for (most) undefined behavior within the function and restricted the ability to make modifications to any values outside the cone of evaluation of the contract-assertion predicate.

This restriction comes with major drawbacks.

- Any member function that modifies program state will, in practice, almost always make modifications to the object to which it is applied. This observation implies either that the rules for modification must be lax enough to be meaningless or that all meaningfully non-`const` member functions must not be conveyor functions. In the latter case, being unable to modify object state would prevent almost all use of the abstractions of C++, limiting contract assertions to only the most trivial of predicates involving only built-in types.

- Introducing a new annotation that must be present on a function implicitly prevents any currently written functions from being used in contract-assertion predicates. Worse, as soon as changes in behavior occur under the restricted rules, such as the defining of undefined behavior proposed in [P3285R0], users will be required to write two versions of many functions, increasing work and bifurcating the library ecosystem. Such a burden is untenable.

- An alternative where functions are implicitly considered to have such an annotation,[4] might

---

[4]It is unclear if [P3285R0] proposes this to be the case or not. In one place it says "operations inferrred (at the point of use) as conveyor functions" may be used in conveyor functions, and it does not clarify whether this includes functions whose bodies are visible an currently qualify as conveyor functions. The original proposal for conveyor functions, [P2680R1], was clearer that functions would be implicitly treated as conveyor functions if their definitions

seem tempting as it allows use of many functions without the need to sprinkle a new annotation across all libraries. This use, however, puts a severe constraint on the evolution of those libraries which was never agreed to. For the vast number of library providers that prefer not to or are unable to release revisions that break client code, this means any visible function that meets the requirements must meet them indefinitely or risk breaking client code if there is a change in the property.

- Another significant drawback to defining a new subset of the language that must be marked with an annotation is that there is no reasonable escape hatch when a function must work outside those boundaries, even temporarily. If all side effects are banned in a type of function, even an operation as simple as a `printf` cannot be added to such functions without breaking any ability to compile. Taking off the attribute to get the function to compile will prevent all contract assertions that use the function from compiling, requiring a massive code change to execute a simple debugging operation. This form of brittle and strictly enforced rule is, in large part, why the original Concepts proposal was removed from C++ prior to C++11.

- Static analysis is more than capable of identifying those contract-assertion predicates that meet its requirements and ignoring those that do not, so limiting the Contracts feature to only those predicates that a particular static-analysis tool insists on needing disenfranchises both more flexible tools and everyone writing runtime predicates that do not fit in a narrow scope.

Overall, pursing the idea of introducing a new kind of function and restricting contract-assertion predicates to only those functions is unrealistically restrictive of runtime checks, has dubious long-term compile-time benefits, has no concrete proposal, and — to date — enjoys no clear evidence or consensus that it is worth pursuing.

5 Future Evolutionary Paths

The Contracts MVP provides a strong foundation for features that will allow us to evolve into a language with even greater support for runtime checking, static analysis, safety, and correctness. Importantly, a critical design goal of the MVP has been to preserve the ability to eventually support all relevant uses cases related to (especially runtime) checking of contract predicates. Nothing discussed in this section is precluded or hindered by the design in [P2900R9], and ongoing effort is being exerted to move toward providing these features in a future iteration of the Standard.

5.1 Integrating Contract Checking with the Core Language

Static analysis must contend with not only analyzing contract assertions, but also analyzing that users are properly making use of the C++ language itself. As explored in [P3100R1], many core-language constructs that have undefined behavior can instead be clearly phrased as having preconditions that must be satisfied, and such preconditions can be treated in essentially the same way we treat precondition assertions on any function invocation.

More importantly, we can assign distinct semantics — *ignore*, *enforce*, or *observe* — to these preconditions wherever they are encountered and still maintain the same program meaning modulo what is currently undefined behavior. Once tools become capable of applying such a transformation,

---

were visible.

we also gain the ability to apply the transformation conditionally and, in particular, to apply it when evaluating expressions within a contract-assertion predicate.

With such options, let's recall our earlier example of a function that does some simple integer math in its precondition assertions:

```
void f(int x, int y, int z)
  pre( x > 0 && y > 0 && z > 0 )
  pre( x + y + z < 1000 );
```

With the ability to flag integer overflow as a contract violation, we can turn the call `f(1431655932, 1431655932, 1431655932)` into a contract violation *within* the evaluation of the second precondition assertion of `f`. This functionality requires no special language rules to enable since the compiler is already free to do anything when integer overflow such as this occurs.

We can look again at the primary example in [P3285R0] to see how it might be addressed differently by the approach in [P3100R1]:

```
int f(int a) { return a + 100; }
int g(int a) pre(f(a) > a);
```

Now, with an implicit precondition on the addition operation applied to the inlined body of `f`, the inlined evaluation of the precondition of `g` will be transformed into something with no undefined behavior:

```
void g_pre(int a)
{
  contract_assert(std::can_add(a,100));  // boundary check from [P1619R2]
  int f_a = a + 100;                      // body of f with precondition
  contract_assert(f_a > a);
}
```

Even better, the above implementation is completely conforming with the specification in [P2900R9] today.

When we consider conventional functions invoked from a contract-assertion predicate, this solution is slightly less viable, though still possible. Binaries would have to be produced with alternate versions of each function that do such checking, which might lead to significant code bloat. Many might consider the safety gained for contract assertions worth the cost, and again the benefits of this approach all come with no need for a single additional word in the C++ Standard to enable them.

To illustrate how language-based contract-checking might work to good advantage, let's consider again an example shown in [P3285R0]:

```
int f(int x) { return x + 100; }
int g(int a) pre(f(a) > a);
```

Given the example above, invoking the function `f` with a value of `INT_MAX-90` will result in integer overflow during the evaluation of the precondition of `g`. In such a case, when inlining `f` today, a compiler might remove the precondition entirely since the only cases where it fails are those in which the evaluation has undefined behavior. With a better approach to handling undefined behavior in contract-assertion predicates, however, the above function, when inlined, could instead detect the

overflow and produce a contract violation, detecting a bug and prompting the user to fix the newly discovered problem.

5.2   Separately Verifying that Predicates Are Nondestructive

A delightfully promising aspect of contract assertions to static analysis is the preconditions and postconditions contract assertions place on declarations, which enables local analysis that is simply not possible (barring alternative methods to provide similar annotations) with C++ today.

Even more powerful, however, is the ability to use the full extent of the C++ language to encode the checking of correctness of a program. Rather than being limited to a bespoke expression language, contract assertions enable users to write checks using the same powerful abstractions available to them when writing the rest of their programs.

By combining the two, static analysis takes advantage of every check that a user can write, even when analyzing only a single translation unit. Many questions, however, might arise as to what possibilities exist in terms of unwanted side effects when invoking an opaque function from within a contract-assertion predicate. If such a predicate does mutate state that is relevant to the correctness of a program, the analysis must then contend with identifying the correctness of the S^N possible different program states that result from S potential semantics chosen for N contract-assertion evaluations.

On the other hand, simply assuming that contract-assertion predicates are *not* destructive reduces this space of S^N states down to exactly 1. While such an assumption might seem bold, we have, in practice, also seen that writing nondestructive contract assertions is remarkably easy — especially with the benefit of so-called `const`-ification to minimize the risk of unwanted side effects, as explored in [P3261R1].

The assumption that contract-assertion predicates are not destructive, like all other assumptions on which local analysis is made, can also be proven in a local fashion and then combined with other local proofs to achieve a whole-program correctness proof.

This can't be stressed enough: Concerns about a predicate being destructive can be addressed by analyzing the predicate itself. Once the concerns are addressed for all predicates in a program, correctness of each function can be analyzed on the assumption that predicates are not destructive, and by combining the two analysis results, we can achieve a robust proof of program correctness.

5.3   Identifying Uncheckable Contract Assertions with Labels

[P2755R1] introduces many use cases for a concept called *labels*, which allows users to introduce new identifiers that can be placed within a contract-assertion specifier to alter its behavior.

Primarily, labels are needed to allow for in-source control of how semantics are selected when evaluating a particular contract-assertion predicate. Control over semantics comes in two forms: providing compile-time (`consteval`) functions that select a semantic for the evaluation of a particular contract assertion and providing a *set* of allowed semantics for a particular contract assertion. This second feature enables the creation of labels that preclude *all* runtime-checked semantics, and when that happens, we open the door to a large number of possibilities that greatly benefit static analysis.

- Symbolic predicates, such as `object_address` (mentioned above), can be used freely within contract assertions whose predicates will never be evaluated at run time. Cases in which a definitive value cannot be determined are freely ignorable by a compiler, while static analyzers will enjoy the full freedom to introduce additional background assumptions to make such predicates even more useful.

- Because such predicates are only hypothetically evaluated but will never directly impact the state of a program, they may freely be expressed in terms of functions that make modifications to program state. This means that we could consider removing `const`-ification from such predicates to more easily facilitate expressing contract checks that can only be accomplished destructively.

- Again, because predicates are not evaluated, meaningful yet destructive contract-assertion predicates can be written without concern for breaking the state of a program. Consider, for example, a function that takes an iterator pair that must have at least three elements in its range:

```
template <typename Iter>
void f(Iter begin, Iter end)
  pre uncheckable ( std::distance(begin,end) >= 3 );
```

Without the `uncheckable` label above, the precondition will be destructive for any input iterator since the call to `std::distance` will consume the entire range. Even for forward iterators, the above precondition may have unacceptable complexity guarantees for `f` to meet its own contract. Hypothetical analysis that is not done at run time, however, can use the precondition to identify the required results of the iteration through the range without actually performing such evaluations.

6 Conclusion

In this paper, we have identified a wide scope of possibilities for static analysis as well as the potential for contract assertions, as provided for by [P2900R9], to facilitate even more useful results from many of those forms of static analysis.

Importantly, if one examines the key design decisions proposed by [P3285R0], the proposal currently suggested by [P3362R0] as an alternative worth pursuing, clearly do not provide a sound benefit

- Defining undefined behavior harms our ability to reason about program behavior and removes our ability to apply better solutions to handling undefined behavior such as [P3100R1].

- Attempting to enforce value-based symbolic predicates, such as `object_address`, is hard to reason about to the point of being unteachable, leaving only the smallest portion of usable functions for a feature that will be challenging to finish designing, specifying, and implementing.

- Introducing a new language subset of functions marked with an annotation, such as `[[conveyor]]`, is not scalable and prevents active use of the Contracts facility with real data structures and abstractions until a huge investment has been made in a tool with questionable benefits.

Overall, we hope we have made abundantly clear that changes to the Contracts MVP are neither needed nor desirable to have robust and effective static analysis built on top of the Contracts facility proposed in [P2900R9]. The existing tools that inspect assertion facilities and the virtually limitless possibilities for future features to grow the usefulness of such tools are each a strong indicator that the Contracts facility is on the right track. One quintessential milestone for the Contracts MVP remains, without which Contracts is all but useless for C++26: that we adopt it into the C++ working paper (WP). All the other benefits described in this paper can be achieved once we have a foundation on which to build.

Acknowledgments

Bibliography

[bde14]   "Basic Development Environment". Bloomberg
             https://github.com/bloomberg/bde/

[P1619R2]   Lisa Lippincott, "Functions for Testing Boundary Conditions on Integer Operations",
             2022
             http://wg21.link/P1619R2

[P1670R0]   Joshua Berne and Alisdair Meredith, "Side Effects of Checked Contracts and Predicate
             Elision", 2019
             http://wg21.link/P1670R0

[P1728R0]   Andrzej Krzemieński, "Preconditions, axiom-level contracts and assumptions – an in
             depth study", 2019
             http://wg21.link/P1728R0

[P2680R1]   Gabriel Dos Reis, "Contracts for C++: Prioritizing Safety", 2023
             http://wg21.link/P2680R1

[P2712R0]   Joshua Berne, "Classification of Contract-Checking Predicates", 2022
             http://wg21.link/P2712R0

[P2755R1]   Joshua Berne, Jake Fevold, and John Lakos, "A Bold Plan for a Complete Contracts
             Facility", 2024
             http://wg21.link/P2755R1

[P2900R9]   Joshua Berne, Timur Doumler, and Andrzej Krzemieński, "Contracts for C++", 2024
             http://wg21.link/P2900R9

[P3100R1]   Timur Doumler, Gašper Ažman, and Joshua Berne, "Undefined and erroneous be-
            haviour are contract violations", 2024
            http://wg21.link/P3100R1

[P3261R1]   Joshua Berne, "Revisiting `const`-ification in Contract Assertions", 2024
            http://wg21.link/P3261R1

[P3285R0]   Gabriel Dos Reis, "Contracts: Protecting The Protector", 2024
            http://wg21.link/P3285R0

[P3362R0]   Ville Voutilainen, "Static analysis and 'safety' of Contracts, P2900 vs. P2680/P3285",
            2024
            http://wg21.link/P3362R0