A Bold Plan for a Complete Contracts Facility

Document #: P2755R1
Date: 2024-04-11
Project: Programming Language C++
Audience: SG21 (Contracts)
Reply-to: Joshua Berne <jberne4@bloomberg.net>
Jake Fevold <jfevold@bloomberg.net>
John Lakos <jlakos@bloomberg.net>

Abstract

Standardizing a sufficiently full-featured C++ contract-checking facility, a.k.a. the *Contracts* facility, has proven — over more than two decades and several repeated attempts by diverse authors — to be far more challenging than a robust replacement for `<cassert>` would first appear to be. Steering a committee toward a solution that meets the wide range of use cases to which the Contracts facility can be applied is exceedingly challenging without a clear idea of the potential shape a complete feature might have. To solve this problem, we present here a vision of a world in which Contracts as proposed in [P2900R6] can evolve into a feature that fully meets the many needs of the C++ community.

Contents

Revision History

Revision 1

- Updated to support the syntax adopted with [P2961R2].
- Clarified evolution in terms of the baseline proposed in [P2900R6]
- Completed Section 4.4 to clarify how Contracts can be applied to the Standard Library.

Revision 0 (September 2023 mailing)

- Initial draft of motivating use cases and overview of features

1 Introduction

When making use of defensive programming techniques, developers and their organizations pass through phases that naturally introduce fundamental aspects of a complete Contracts facility:

- How to express checks for many aspects of the agreement that exists between a client and the provider of a piece of software
- How to manage the deployment and software lifecycle of checks as they are introduced, tested, analyzed, and evaluated in various running environments

The Contracts MVP that has been developed by SG21[1] addresses these needs for the simplest of contract checks, and questions of deployment are largely left to implementation-defined possibilities and places no finer-grained control over the effect of contract-checking annotations (CCAs) available within those annotations themselves.

This paper is organized into a few sections that will, when it is complete, leave the reader with a full understanding of the path a developer might take through using the full set of features we will propose.

- Section 2 walks through the individual use cases that C++ programmers might encounter as they find parts of our proposed features with which to satisfy those use cases. Some of these use cases are fully satisfied by the SG21 MVP, and others require the use of the additional features that we propose here.
- Section 3 describes, in depth, the motivation, design considerations, and proposal for each of the features that make up the Contracts facility we are describing. Where a relationship exists between these proposals, the features are presented in order of dependency, not in the order in which an instructor might present them or an engineer might need them.

  NOTE: Much of this section remains incomplete as we evolve future designs and have primarily been focusing on completing the initial, lowest-level features present in [P2900R6].
- Section 4 presents some large, high-level examples of what can be done with the features we propose.

[1]See [P2900R6] for the features proposed by SG21 as the initial Contracts feature that should be reviewed by EWG to be adopted into the draft C++ Standard.

- Section 5 contains discussions of a number of other aspects of the language and of the Contracts feature that will eventually need to be addressed and offers some possibilities for how those issues might be addressed in the next iterative step after this plan is complete.

2 A How-To Tour of C++ with the Contracts Facility

The Contracts facility's users arrive from diverse paths, at various stages of their careers, and with vastly different needs and goals. Each user's story is worth considering, because it presents a real case encountered by someone who seeks to make extensive use of the C++ Contracts facility. For each such story presented here, we will also be providing solutions that meet their needs.

2.1 A Career Spent Writing Contract-Checking Annotations (CCAs)

From students writing their very first programs in C++ to senior managers controlling the deployment of software at billion-dollar companies, the ability to leverage a robust Contracts facility in C++ provides ways to easily, safely, and consistently produce software having fewer defects and achieve higher levels of success.

2.1.1 Write precondition checks

When beginning the academic study of software engineering or while simply learning C++, you quickly see that you need to do something to check that you are using your functions correctly. Too often, you spend time chasing down bugs that appear far from the place where the actual defect occurred.

To express a simple precondition on a function, use a precondition assertion at the end of the function declaration:

```
double sqrt(double x)
  pre (x >= 0 );
```

The expression — e.g., `x >= 0` — can be any C++ conditional expression and will be checked after function parameters are initialized but before anything inside the body of the function is invoked.

On more involved function declarations, such as those making use of trailing return types, `requires` clauses, or other specifiers, the precondition assertion always comes at the end of the declaration, immediately prior to anything that defines the function itself such as `=default` or a function body:

```
struct S {
  virtual auto foo() final noexcept -> int pre(true) = default;
};
```

With this simple construct, a large number of opportunities open up.

- When reading your function, you — as well as any other developers who begin to make use of your function — will immediately better understand how to use it properly.
- Using your compiler documentation, you can determine how to configure your program so that precondition assertions are *enforced* and then run your program and find bugs. Any time a

contract violation is detected, your program will terminate and a diagnostic message informing you of where the violation happened will be printed.

- Static analyzers will begin to find opportunities to inform you when a particular invocation of your function *violates* a precondition or fails to guarantee that the precondition is *satisfied* and thus *might* have violated the precondition with certain inputs.
- You can begin to learn in which conditions writing functions having narrow contracts is best and, more importantly, in which conditions you should be writing functions having wide contracts.[2]

2.1.2 Write postcondition checks

As you write more functions and begin to test them, you must keep in mind the guarantees that a function makes to its clients after a successful invocation. Before you invest time to write thorough tests, you want a quick way to ensure you are meeting the basic guarantees you expect at a call site on all invocations.

To check a postcondition, add a postcondition assertion to the function declaration. Postcondition assertions and precondition assertions, collectively known as function contract assertions, can mix and match and may come in any order:

```
void push_back(std::vector<int> &v, int value)
  post(v.back() == value)
```

Postconditions may also name the return value to express properties of that return value:

```
int square(const int x)
  post(r : r == x * x);
```

When you name a function parameter in a postcondition, it must be `const`. Postconditions are checked when a function returns normally.

Postcondition checks facilitate finding bugs in a function's implementation. Thoroughly annotating postconditions also greatly facilitates the ability to apply static analysis tools to detect bugs or prove correctness without even running a program, as the postcondition provides information to the call site without needing to understand the implementation.

Thorough postconditions also aid in the thoroughness of unit testing. The postconditions declared for a function aid meaningfully in the development of unit tests and identifying boundary conditions that must be exercised by good unit testing. In addition, by executing all tests with contract checking enabled, failures in postconditions then increase the likelihood of finding edge conditions missed by the chosen testing strategy.

2.1.3 Write an assertion

As you learn the language and write more code, you find function contract assertions frequently helping you to achieve a correct and usable piece of software far more effectively than you could

[2]See [P2053R0].

before you began using this new Contracts facility. While checks on function boundaries seem to be useful, often you find that you need to validate some conditions at other points during program execution.

A *assertion statement* is a contract assertion that is evaluated where it is written; to write one, you use a new statement that is introduced with the `contract_assert` keyword. For example, a more involved precondition on the elements of a container might be easier to assert as you iterate over the container:

```
int sumPositiveValues(const std::vector<int>& v)
{
  int output = 0;
  for (auto x : v) {
    contract_assert( x > 0 );
    output += x;
  }
  return output;
}
```

Assertion statements can be used in a variety of places that are not readily covered by `pre` and `post`:

- When a precondition cannot be correctly or easily checked with a single expression but is correctly or more easily checked after a small amount of work has been done by the function, such as during iteration or after acquiring a lock to safely access a resource
- During a complex operation as a *sanity check* that an intermediate step has been successfully reached
- To verify that a separate function has itself satisfied expected postconditions that might not be checked
- To perform partial checks on the input *as* an algorithm executes, e.g., see Section 2.1.6

Assertions also provide a replacement for the `assert()` macro from `<cassert>` that is, in almost all ways, an improvement:

- The `assert()` macro expands to nothing at all when `NDEBUG` is defined. This behavior causes a fundamental problem, sometimes called *bit rot*, in which organizations that generally build with `NDEBUG` defined discover that the expression in their `assert()` macros no longer compiles when `NDEBUG` is not defined. For a project in which builds with `<cassert>` enabled are not regularly made, this technical debt quickly grows such that builds with `<cassert>` *cannot* be made due to the widespread failures, and the cost of fixing the failures becomes insurmountable. An assertion statement, however, does *not* suffer from this problem since the expression must always compile, even when choosing a build option that ignores contract assertions.
- Builds can also vary behavior beyond the `assert()` macro based on preprocessor checks of the `NDEBUG` macro. This approach can quickly lead to checked and unchecked builds being incompatible with one another and thus being unable to link together whenever, for example, the contents of a user-defined type depends on the state of the `NDEBUG` macro:

  ```
  #include <cassert>  // for assert()
  ```

```
struct S {
#ifndef NDEBUG
  bool d_debugFlag = true;
#endif

  void test()
  {
    assert(d_debugFlag);
  };
};
```

Even when changes in the layout of an object do not happen, any use of the `assert` macro in a header file will still be an ODR-violation should the same inline function be compiled in different translation units with different states for the `NDEBUG` macro.

For local code that supports the evaluation of contract assertions, there is the `contract_support` statement. Other code which would not be ABI compatible is not support by the Contracts facility, making such pitfalls much less likely to occur and generally enabling mixing of checked and unchecked libraries in the same program as needed.

2.1.4 Improve violation diagnostics

Upon joining a new project, you find yourself deploying software as part of applications with pre-existing logging frameworks that manage to deliver those logs in a highly configurable manner. You quickly see the need to report contract violations to the same logging framework.

When building an application, you can link to a single contract-violation handler[3] and thus replace the default provided by the platform:

```
void ::handle_contract_violation(
    const std::contracts::contract_violation& violation)
{
  MY_LOG_ERROR << "Contract violation detected at "
     << " file " << violation.location().file_name()
     << " line " << violation.location().line();
}
```

A wide variety of other choices can be made in an application-specific manner when implementing a custom contract-violation handler.

- Through the use of `std::stack_trace`, additional contextual information about the circumstances where the violation occurred can be logged.
- Application-specific state, such as information about the current request being serviced, can be captured to aid in problem diagnosis.
- For security-conscious applications that seek to avoid leaking information about the software implementation, the contract-violation handler can choose to log nothing at all or to encrypt the information that gets logged.[4]

[3]See [P2811R7] for the full proposal adopted by SG21.
[4]See [P2947R0] for further discussion of this need.

Custom contract-violation handlers can also be used for a variety of more advanced purposes; see [P2811R7] for a broader discussion of this subject.

2.1.5 Write and use expensive checks

As you go through your codebase, you find that many checks are trivial to both implement and execute. Occasionally, some requirement on the caller of a function would take longer to execute than the function itself. You still want to encode and occasionally check such conditions, but the overwhelming cost of evaluating them means they couldn't be deployed as part of a regular production build.

Each contract assertion kind supports having contract labels placed on it, and in this case, you need to reach for the `audit` label provided by the Standard Library:

```
#include <contracts>  // for audit contract label

int *binarySearch(int *begin, int *end, int value)
  pre audit ( std::is_sorted(begin,end) );
```

Without the `audit` label on this precondition, this binary search would certainly no longer have logarithmic ($O(log(N))$) complexity but would instead be linear ($O(N)$). Such increases in runtime complexity might transform other enclosing algorithms such that they are no longer able to easily handle high workloads and are now unable to continue processing quotidian business in a timely fashion.

The `audit` label's behavior is primarily controlled by a build knob, `std.contracts.cost`, which can be set to a number or named value. When the build knob is set to `audit`, which is generally done by using the `-Kcontracts.cost=audit` command line option for your compiler, contract assertions with the label `audit` will pick up the general semantic assigned that contract assertion by your platform configuration. When this knob is not configured or is configured to a lower level than `audit`, all such CCAs will be ignored.

Independently of the value of the build knob chosen, static analysis will still see checks marked with the label `audit` and attempt to provide guidance on whether they are violated, but the unacceptable runtime cost is avoided except for builds that explicitly opt-in to these checks.

Builds with such audit-level checks enabled are often reserved for unit testing or for diagnosing particularly challenging problems, but in some cases, a production system with extra levels of checks enabled can also be deployed as an early-warning system to detect problems with actual production inputs.

2.1.6 Heuristically check aspects of an expensive check

In the land of those without formal education, the person who passed computer science 101 reigns, and in your case, that means the `binarySearch` algorithm you wrote when joining your current employer has drastically improved the performance of hundreds of applications. With great power, however, comes great responsibility, and users keep coming to you with problems that you painstakingly track down to cases of users passing in unsorted arrays to your algorithm. You provided an expensive *`audit` precondition to detect this situation, but none of your clients are able to deploy that check with real workloads where their defects become visible.*

A binary search requires a sorted input to guarantee that a value will be returned whenever it is present in the provided list of values. When the needed preconditions are met, the search guarantees both the result and that the results have been found with $O(log(N))$ operations:

```
int *binarySearch(int* const begin, int* const end, const int value)
  pre audit ( std::is_sorted(begin,end) )
  post ( r : (r == end) || (*r == value) )
  post audit ( r : (r == end) || r == std::find(begin,end,value) );
```

Either of the `audit` labeled contract assertions above would find your users' fundamental problem — the cases in which they looked for an item in the range and failed to find it. The `audit` postcondition would also identify *when* a search failed but would not indicate *why* it might be failing. Both checks, however, would change the algorithmic complexity of the invocation of `binarySearch` from $O(log(N))$ to $O(N)$. For many use cases of this algorithm, however, that change in complexity will explode enclosing operations that were previously $O(N*log(N))$ to be $O(N^2)$ or worse, removing any chance of running them on realistic input sizes.

When attempting a binary search on an unsorted list, you face a significant chance that a pair of elements is encountered that are out of order. Checking only the pairs that are visited — instead of checking the entire list for pairs being sorted — will not alter the algorithmic complexity of your `binarySearch` function but will detect many (but not all) misuses that are otherwise difficult to diagnose. Such checks are challenging to encode as a precondition but are trivial to check *during* the evaluation of the search operation:

```
int *binarySearch(int* const begin, int* const end, const int value)
{
  int *rbegin = begin, rend = end;  // Range begin/end.

  while (rbegin != rend) {
      int * mid = rbegin + (rend-rbegin)/2;
      contract_assert ( *rbegin<=*mid && *mid<=*(rend-1) );  // #1

      if (*mid <= value) rbegin = mid ; else rend = mid;
  }
  return (rbegin == end || *rbegin != value) ? end : rbegin;
}
```

As the above binary search narrows its range, the assertion at `#1` verifies that the midpoint and ends of the range form a sorted triple (or a pair when the range becomes very small). This approach will not discover *all* misuses of `binarySearch` but will quickly find problematic calling code when presented with real inputs, thus diagnosing problems quickly without the need for complex reproductions or time consuming debugging sessions.

2.1.7 Deploy contract checks for the first time in a legacy application

With your organization's large legacy codebase, you find it questionable to enforce newly added contract assertions in production environments when the pre-existing application is clearly running *in a stable state. Having even one ill-fated release in which everything repeatedly crashes is likely to result in loss of money for your organization, loss of reputation for both your organization and you personally, and perhaps loss of employment for you. Such problems could lead to management mandates that no enforced contract assertions can be deployed to production or even that Contracts should not be used, and such rules hinder efforts to improve the safety of running systems.*

In the same way that your compiler allows you to specify that contract assertions should be *enforced* (and thus terminate the program if violated), you can specify that those same contract assertions are *observed* in a particular build. Thus, a contract-violation handler can be invoked without the risk of program termination.[5, 6]

Deploying such a build, monitoring the results, and addressing all the detected defects will typically lead to addressing and fixing outstanding issues without the need to first crash production systems. Once all issues are addressed, an enforced build can be re-released to prevent highly infrequent defects from running out of control when they do eventually occur.

Once deployed, the build with all checks *enforced* will catch rare and unexpected errors quickly, preventing them from being lost in a sea of unrelated log messages or, worse, being used to subvert a systemic flaw for nefarious purposes. Alternatively, a different cost-benefit analysis might lead to *ignoring* or *assuming* checks with the knowledge that quotidian input data does not lead to violation, removing the runtime cost of doing any checks.

2.1.8 Add a single contract check to already-checked applications

After running contract checks in production for a specified period with no violations detected, your management reasons that failing fast now, if something unexpected happens, is better than allowing execution after a potential vulnerability has been identified and finally decides to enforce all contract checks in production builds. Your team agrees that failing fast is better than allowing problems to linger undetected. As ongoing maintenance of legacy code continues, however, you discover new code that would greatly benefit from additional contract checks.

Each individual contract assertion may contain a sequence of labels that guide the selection of its semantics when evaluated. The label `new` will modify a contract assertion's behavior, including that specified by any previous labels, so that if the contract assertion would otherwise be *enforced*, it will instead be *observed*:

```
#include <contracts>  // for new contract label

void f(int x)
  pre new ( x > 0 ); // just added
```

Observing this contract assertions and others labeled `new`, while continuing to enforce stable (not `new`) contract assertions, enables introducing these new checks without risking unexpected termination and without removing the guard rails provided by the existing enforced checks.

[5]See [P2698R0] for motivation related to preferring solutions other than program termination.
[6]See [P2877R0] for more details on how contract semantics are chosen when evaluating a CCA and the *observe* semantic in particular.

2.1.9 Write an expensive, new check

As you add checks to code, you suddenly realize that a new `audit`-level check is needed. The decision on cost and the newness of a contract assertion are clearly orthogonal. What can be done?

Multiple labels can be placed on a single contract assertion,[7] separated by spaces:

```
#include <contracts>  // for audit and new contract labels

int *binarySearch(int *begin, int *end, int value)
  pre audit new ( std::is_sorted(begin,end) );
```

Now, even when other audit-level checks are enforced, *new* audit-level checks can instead be observed, which is particularly helpful when running audit-level checks in a canary deployment — i.e., a small subset of running production systems that trades performance for extensive enabled correctness checks.

Importantly, all labels are applied in the order in which they appear lexically. When computing the semantics for evaluation, the label `audit` will be passed the compiler-chosen semantic for this contract assertion, and by combining that with other configuration options, the passed-in semantic or the `ignore` semantic will be chosen. After that, the label `new` will take the semantic computed up to that point and choose a semantic appropriate to a newly added contract — *observe* instead of *enforce* and *ignore* instead of *assume*.

2.1.10 Write checks that are not yet ready to release

As you add more checks, you'll find that some are complicated and some are not yet tested, and the pressure of getting to market means that some additional checks are intended but are not yet fully implemented because time simply ran out. Skipping (e.g., by commenting out) or even removing entirely such not-yet-implemented checks, however, would hide technical debt, allowing it to fall through the cracks and remain unaddressed.

The Standard Library provides the label `always_ignore`, which will force the contract assertoin to which it is applied to always evaluate with the *ignore* semantic:

```
#include <contracts>
void f(int x)
  pre always_ignore ( x != 0 );  // always ignored
```

When the label `always_ignored` is applied to a contract assertion, the *enforce*, *observe*, and *assume* semantics (i.e., all semantics other than *ignore*) are no longer allowed when evaluating that contract assertion. Because evaluating this contract assertion's predicate at run time is no longer possible,

[7]The syntax presented here is one potential approach to specifying multiple labels on a CCA. An alternate syntax relies more heavily on using C++ expressions to combine distinct labels:

```
#include <contracts>  // for audit and new contract labels

int *binarySearch(int *begin, int *end, int value)
  pre<audit | new> ( std::is_sorted(begin,end) );
```

See Appendix A.1 for further discussion of this potential alternative.

regardless of how the software is built, the contract assertion's predicate will not require, as a condition for linking, that any functions it names have definitions; those functions will never need to be evaluated at run time:

```
bool is_foozible(int x);
  // Unimplemented function; return true if x is foozible.
  // Implementation will be provided in a future release.

void f(int x)
  pre always_ignore ( is_foozible(x) );
```

With a declaration and check already written, readers will know that the code was written with the expectation that this predicate should remain `true` when invoking `f`. When the time is found to implement `is_foozible`, the `ignore` label on the associated contract assertions can be removed.

2.1.11 Unit test a custom contract-violation handler

You've wisely designed a new logging framework to be readily unit-testable from the bottom up. As part of your framework, you also provide a replaceable contract-violation handler for users to link in. The argument to that handler, an object of type `std::contracts::contract_violation`, is not something you can create yourself, so how do you unit test your violation handler?

The Standard Library provides the label `always_observe` that, when applied to a contract assertion, will force that contract assertion to be evaluated having the *observe* semantic. By using this semantic, a unit test that invokes the currently installed contract violation is simple to write:

```
void testViolationHandler()
{
  // Prepare logging framework for recording.

#line 17 testfile.cpp
  contract_assert always_observe ( false );

  // Verify that violation handler logged line number 17 and
  // a file name of "testfile.cpp".
}
```

Unlike a normal assertion, the labeled assertion statement above will *always* be checked, and due to a predicate of `false`, it will *always* invoke the contract-violation handler in all build configurations.

2.1.12 Write libraries where contract checks are not optional

While working for a particularly safety-conscious oragnization you find that defensive checks, even when redudnant in a correct program, are mandated to always be on without any control over that behavior given to clients of the library.

Just like with `always_observe`, there is another label `always_enforce` that will cause a contract assertion to *always* use the *enforce* semantic:

```
double mylib::sqrt(double x) pre always_enforce (x >= 0);
```

All of the advantages of static analysis that can be applied to contract assertions apply to those marked `always_enforce`, including making the choice to document the function's requirements with code. As an owner of this library, however, you can remain confident that anyone building with this source code will never enter the body of `mylib::sqrt` when the parameter `x` is negative.

2.1.13 Experiment with specific contract assertion semantics

While doing some performance analysis of a highly important inner loop, you need to discover the exact impact each potential contract semantic has on the code generated for your function.

Firing up your preferred source of real-time code generation analysis (such as `https://godbolt.org` or `https://wandbox.org`), you can make use of any of the Standard Library concrete semantic labels to compare the same function with different contract assertion semantics:

```
void f_ignore(int x)  pre always_ignore  ( x != 0 ) { /* ... */ }
void f_observe(int x) pre always_observe ( x != 0 ) { /* ... */ }
void f_enforce(int x) pre always_enforce ( x != 0 ) { /* ... */ }
void f_assume(int x)  pre always_assume  ( x != 0 ) { /* ... */ }
```

Exploring the impact of the various semantics, you will find a few interesting results.

- Conditional checks — such as `if (x` $=$ 0)! — will be removed in `f_enforce` and `f_assume`.
- Entire branches guarded by `if (x == 0)` will be removed completely in `f_enforce` and `f_assume`.
- At some optimization levels, code in the `f_observe` body might be rearranged due to the compiler considering the `x == 0` case to be unlikely.
- Code that *invokes* `f_enforce` or `f_assume` will also optimize statements following the function invocation due to the knowledge that, in those blocks, `x != 0` will be true.

2.1.14 Write and use unimplementable checks

Looking over the functions you support, you question how you can express conditions for which you cannot write checks, and you hope some tool might be able to warn you about those situations if the tool knew that you wanted to be warned.

Many aspects of a function contract refer to conditions that cannot be verified within the C++ language, such as whether an object is within its lifetime or if a pair of pointers form a valid range. The inability to verify these conditions does not remove the benefit of expressing the requirement that they be true; expressing such requirements might allow tools other than runtime checking to notify you of a problem.

On the other hand, some checks are *destructive*[8] if they are evaluated.

- One source is checks with side effects that impact essential behavior even when they return true, such as any check that might advance a forward iterator.

  ```
  template <typenmae ITER>
  bool has_remaining(ITER* begin, ITER* end, int count)
  {
  ```

[8]See [P2751R1].

```
    return std::distance(begin,end) >= count;
  }
```

For input iterators, the above function will consume all available elements, while for other iterator categories, the expected condition will always be properly identified (although at linear cost for forward iterators).

- Another common form is checks that *accept* valid situations by returning `true` but have *undefined behavior* (UB) in some or all conditions where a violation would occur, such as checking `is_reachable` for two pointers by iterating between those pointers:

  ```
  template <typename T>
  bool is_reachable(T* a, T*b) symbolic
  {
    while (a != b) ++a;  // has UB if b is not reachable from a
    return true;
  }
  ```

  The `symbolic` function specifier indicates that this is a function that may not be invoked and is intended for use in unevaluated contexts only. These contexts include unevaluated operands such as a `sizeof` expression as well as contract assertion predicates that have no allowed checked semantics.

- Other checks are destructive for both reasons. When no violation occurs, the check would return `true` in a well-defined manner but would hinder further successful program execution by severely altering program state. When a violation occurs, the check would have UB but can make no guarantee that `false` will be returned. One such check would be verifying if a pointer has been allocated using `std::malloc` by leveraging the corresponding precondition on `std::free`:

  ```
  bool is_malloced(void *p) symbolic
  {
    std::free(p);
    return true;
  }
  ```

- Rather than have a mostly expository function body, the meaning of such symbolic functions can also be left entirely for the platform to implement internally, with no need for a definition for functions that will never be evaluated:

  ```
  template <typename T>
  bool is_reachable(T* a, T*b) symbolic

  bool is_malloced(void *p) symbolic;
  ```

For various reasons, obviously, these checks cannot be evaluated safely at run time, and any contract assertion that makes use of them would be dangerous if it did not actively prevent such runtime evaluation. The Standard Library label `uncheckable` makes such prevention simple:

```
int accumulate(int *first, int *last)
  pre ( uncheckable : is_reachable(first,last) );
```

Labeled in this way, the precondition of `accumulate` shown here will never — in any build configurations or with any other labels applied — be evaluated with the *enforce* or *observe* semantic. Thus, no dangerous evaluation of the predicate at run time will ever occur. Without the `uncheckable` label, this program would be ill-formed due to the `symbolic` specifier on `is_reachable`. On the other hand, static analysis and your compiler's optimizer might benefit from the explicit knowledge that the iteration within `is_reachable` would be expected to run to completion.

2.1.15 Disable a check that is sometimes destructive

You are now beginning to work on a generic library and hope to leverage the Contracts facility as much as is feasible. Some checks, however, are valid for certain template parameters and destructive for others.

The `uncheckable` label for the Standard Library supports a boolean parameter, which, when false, makes the label have no effect:

```
template <typename ITER>
void consume3(ITER first, ITER last)
  pre uncheckable<!std::random_access_iterator<ITER>>
      ( std::distance(first,last) >= 3 );
```

When the type `ITER` above is an input iterator, the call to `std::distance` would consume the input range in the process of measuring its size, leaving iterators representing an empty range at the start of the function body. This empty range will certainly no longer have three elements ready to consume within that function body.

2.1.16 Mimic Standard Library preconditions

The C++ Standard makes no distinction between UB specified in the core language and violation of preconditions on functions provided by the Standard Library. With user libraries, calling a function out of contract might provide no guarantees to the caller — i.e., it is library UB — but is not language UB that the optimizer is authorized to leverage to improve code generation. You would like to enable your library functions to benefit from the same performance optimizations that a Standard Library function would enjoy.

Any label may, when controlling the semantic that will be used when evaluating a contract assertion, specify the *assume* semantic for that contract assertion. The Standard Library `always_assume` label will always use that semantic.

This label allows for assertions that are functionally equivalent to a portable assumption attribute[9]:

```
int divide_by_32(int x)
{
  [[ assume( x >= 0 ) ]];               // UB if x < 0
  contract_assert always_assume ( x >= 0 );  // same thing.
  return x/32;
}
```

[9]See [P1774R8].

This same label, `always_assume`, can be used on preconditions to achieve the same optimizations for a user-defined function that would be applied to a Standard Library function. Consider, for example, the Standard Library function `std::unreachable` whose precondition is the unsatisfiable "`false` is `true`":

```
namespace std {
  void unreachable();
}
```

Any invocation of this function is UB, and your compiler is well aware of that function. Similar function can be achieved with a user-defined function that assumes its precondition:

```
void my_unreachable() post always_assume ( false );
```

A CCA used to unilaterally assume a condition, just like the `[[assume]]` attribute, is, of course, a sharp-edged tool and is potentially unsafe. Used properly, Contracts allows us to do considerably better.

2.1.17 Verify assumptions

A previous developer made heavy use of portable assumptions to improve performance. The last time one of those assumptions proved incorrect, the organization had to make a huge effort to identify the source of the problem. You want to use the Contracts facility to aid in identifying such issues in the future.

The label `checkable_assume` in the Standard Library denotes a contract assertion meant to function as an assumption for optimizations but with the option to check that assumption in certain builds. In normal circumstances, contract assertions so labeled will not evaluate their predicates but will introduce *UB* if their predicate *would have* evaluated to `false`.

```
void f(const std::vector<int>& v)
{
  contract_assert checkable_assume ( v.size() % 16 == 0 );
  for (auto it = v.begin(); it != c.end(); it += 16) {
    processBlock(it,it+16);
  }
}
```

For predicates that do not have adverse side effects if evaluated, the default behavior of the check above is precisely the same as if the `[[assume]]` semantic had been used:

```
void f2(const std::vector<int>& v)
{
  [[ assume(v.size() % 16 == 0) ]];
  for (auto it = v.begin(); it != c.end(); it += 16) {
    processBlock(it,it+16);
  }
}
```

The difference, however, is that you can recompile the code so that all contract assertions having the label `checkable_assume` have the *enforce* (or *observe* or *ignore*) semantic instead of the *assume* semantic. Running that build will immediately reveal where a client of `f` is passing a `vector` that does not have a multiple of 16 elements.

This same label can then be applied wherever you might have used `always_assume` to get an improved contract assertoin that lets you quickly identify faults when the introduced UB turns out to have been the wrong thing to do:

```
void my_better_unreachable() post checkable_assume ( false );
```

When using the label `checkable_assume`, one subtle difference needs to be remembered: `checkable_assume` does allow for the possibility of evaluation at run time and thus would never be appropriate for a destructive check, while the `[[assume]]` attribute never allows such evaluation. Destructive checks can be used with contract assertion, but they should generally have the label `uncheckable` (see Section 2.1.14) applied to them instead.

2.2 Expressing Wider Varieties of Contract Checks

Moving beyond the simple feature sets provided by the initial Contracts release in C++ (the SG21 Contracts MVP), you find a variety of new features that improve every developer's ability to verify larger portions of their software contracts clearly and robustly.

2.2.1 Render function contract assertions in the most appropriate locations

With so many rich features available to express a wide variety of contract checks and maximize your software's ability to identify defects as well as to control the details of how those checks will take effect in the plethora of ways your software is deployed, reading a function declaration can be overwhelming. How do you manage this ever-growing complexity?

Unlike the original MVP Contracts proposal from SG21, function contract assertion placement is not restricted to the first declaration of a function, offering us much more flexibility. The function contract assertionss for a function — i.e., all `pre`, `post`, or `interface` contract assertions attached to that function — may be on *any* declaration of a function as long as *every* declaration with function contract assertoins has the *same* function contract assertions or *no* function contract assertions:

```
void f(int x);                  // no CCAs
void f(int x) pre ( x >= 0 );   // OK
void f(int x) pre ( x >= 0 )    // OK, same CCAs
{
  return x * x;
}
```

These lists of function contract assertions must be in the same and identical (including labels, capture lists, `requires` clauses, and predicates) according to the one definition rule except for allowing the renaming of function parameters, template parameters, or return value identifiers:

```
namespace mylib {
int global = 5;

template <typename S>
int g(const S& s, int x);  // OK, no CCAs
```

```
template <typename T>
int g(const T& t, int x)
  pre ( x >= 0 )
  pre audit ( global == sizeof(T) )
  post ( r : r >= 0 );

%
template <typename U>     // renaming template parameter
int g(const U& u, int y)  // renaming  function parameters
  pre ( y >= 0 )   // OK
  pre audit ( mylib::global == sizeof(U) )  // OK
  post ( retval : retval >= 0 );            // OK
```

With this increased flexibility, function contract assertions — just like inline functions or templates — need not bloat the initial declaration that clients are expected to read.

When the ODR-use of a function does not know that said function has CCAs, then caller-side checking is disallowed, even on platforms that prefer it, and some implementation strategies can have difficulty supporting this case properly.

2.2.2 Inform readers of a subset of function contract assertions

The first functions you wrote all had simple preconditions and postconditions — `x>0`, `a<b`, and so on — and were very useful to present to readers of your function declarations. As work continued, however, more labels, `requires` clauses, large expressions, and expansive procedural interfaces made declarations far too large to digest, so you moved your function contract assertions to later declarations away from view. How do you recover the lost ability to convey the simple things without missing out on the more advanced possibilities?

Declaring function contract assertions on a function but not providing their full definitions with contract assertion *kinds* that have a `*` suffix is possible:

```
int f(int x)
  pre ( x > 0 )
```

Later declarations may add additional metadata or contract assertions:

```
int f(int x)
  pre* audit ( x > 0 )  // OK, adding label
  post* ( r : r > 0 );  // OK, adding additional postcondition assertion
```

When such declarations are seen, a full definition of the contract assertions of the function that is compatible with those declarations must be reachable from the function definition:

```
int f(int x)
  pre audit new ( x > 0 )  // OK, matches declared pre
  pre ( x < 10000 )        // OK, adding additional precondition assertion
  post audit ( r : r > 0 )
{}
```

Any pitfalls associated with ODR-use of a function when not knowing if that function has function contract assertions at all can be avoided by informing clients of the existence of function contract assertions with function contract assertion declaration on the function declarations those clients can reach:

```
// component.h
int f(int x) pre*;  // first declaration with empty
                    // function contract assertion declaration

// component.cpp
int f(int x)
  pre ( x > 0 );
{
  // ...
}

// client.cpp
#include <component.h>

void test()
{
   f(10);
}
```

Here, within `test`, the call to `f` might still, on some platforms, do call-side checking even without visibility of the specific checks that would be performed.

2.2.3 Write type-dependent contract assertions

Today you begin working on a generic library and are tasked with implementing a robust set of contract checks to take advantage of the safety and correctness improvements that modern C++ provides. Many types in your system support a concept in which they provide a `checkInvariants()` member function. On many API boundaries, you would benefit from being able to recheck those invariants by using this function in a precondition, ideally catching cases of object corruption outside your library at the earliest possible point.

With the addition of Concepts to C++20, declarations of templated functions (e.g., a function template or a member function of a class template) may have `requires` clauses that restrict their instantiation based on a boolean expression involving their template parameters. Very similarly, a `requires` clause may be added to the metadata sequence in a contract assertion on a templated function which, when not satisfied, will remove the contract assertion from that function:

```
template <typename T>
concept has_check_invariants
  = requires (const T& t) {
      t.checkInvariants() -> bool;
     };

template <typename T>
void f(const T& t)
 pre requires(has_check_invariants) ( t.checkInvariants() );
```

Such a requires clause might also use a requires expression directly, with the extra ()s that would not be needed using requires requires on a template elsewhere:

```
template <typename T, typename U>
void f(const T& t)
  pre requires(requires(const T& t) { t.checkInvariants() -> bool; })
      ( t.checkInvariants() );
```

Unlike function overload sets, each function contract assertion stands alone, so subsumption rules do not matter for these requires clauses.

2.2.4 Work with structured return values

Considering an API that returns multiple values as unnamed tuples, you find that writing postconditions becomes increasingly painful in terms of the tuples without using the return value the same way clients are expected to — i.e., initializing a structured binding to give meaningful names to the elements of the returned tuple.

Instead of naming a return value, a list of names enclosed within [and] may be provided that will be used as the names for a structured binding:

```
#include <tuple>    // for std::tuple
#include <vector>   // for std::vector

template <typename T>
std::tuple<T,T> minmax(const T& lhs, const T& rhs)
  post ( [min,max] : (min <= lhs) && (min <= rhs)
                   && (max >= lhs) && (max >= rhs) );
  // Return an ordered pair of values such that the
  // first is not larger than lhs or rhs and the second
  // is not smaller than lhs or rhs.
```

Such a postcondition not only gives users a clear hint as to the meaning of the elements of the returned tuple, but also uses that returned tuple similarly to a client's use of that return value.

2.2.5 Use an original value in a postcondition

Since many postconditions are written in terms of how a state is changed before and after a function invocation, you wonder how to write such a postcondition.

When referencing a by-value function parameter in a postcondition, the object being referenced is the parameter at the time the postcondition is evaluated. The parameter must be const; even if the parameter were not required to be const, the original value from prior to evaluation of the function body is long gone.

To express postconditions that need that original value, a *capture list* of variables may be added to the postcondition that will be initialized at the same time preconditions are evaluated — i.e., after function parameter initialization and before the body:

```
void increment(int& input)
  post [orig_input = input] ( input == orig_input+1 );
```

Other expressions can, of course, be used to initialize the captured value beyond just naming a function parameter:

```
void increment(int& input)
  post [expected = input + 1] ( input == expected );
```

The type of the captured value is computed just like that for a lambda init-capture; in this case, the type of `orig_input` is `int`.

2.2.6 Store a generic element's value for a postcondition

You're writing a generic sequence container and want to capture the value of an element being passed to `push_back` so that you can validate that the correct value is placed at the end of your sequence in a postcondition.

For a copyable type, a copy of the value could be captured, using a *capture list*, to be used in a postcondition:

```
template <typename T>
void mylib::vector<T>::push_back(T&& value)
  post requires(is_copyable_v<T>) [ input_value = value ]
     ( back() == input_value );
```

The Standard Library provides a pair of customization points, `std::memoize` and `std::memoization_equals`. The first, `std::memoize`, can be used to obtain a *memoization* of an object that supports this operation.

```
template <typename T>
void f1(const T& t)
{
  auto memoization = std::memoize(t);
}
```

The second customization point, `std::memoization_equals`, can be used to compare a memoization to an object to identify if its salient state has changed since the memoization was captured:

```
template <typename T>
void f2(const T& t)
{
  auto memoization = std::memoize(t);
  contract_assert ( std::memoization_equals(memoization, t) );
  ++t;
  contract_assert ( !std::memoization_equals(memoization, t) );
}
```

In addition, the Standard Library defines the type `std::memoize_t<T>` as the type returned by `std::memoize(std::declval<T>())`. Finally, the Standard Library concept `std::memoizable<T>` is `true` if both customization points can be used with `T`; i.e., the following two expressions are both valid:

```
  std::memoize(std::declval<T>());
  std::memoization_equals(
    std::declval<std::memoize_t<T>>(),
    std::declval<T>());
```

For copy-constructible and equality-comparable types `T`, `memoize` returns a copy and `memoization_equals` simply compares that copy using `operator==`. These customization points also work with various move-only Standard Library types. For example, `std::unique_ptr<T>` returns `get()` from `std::memoize` and compares that pointer with `get()` when `memoization_equals` is invoked. Users may provide their own, similar customizations for noncopyable types as well.

Using memoize, we can now extend our precondition to apply not just to copyable types, but to anything that is memoizable:

```
template <typename T>
void mylib::vector<T>::push_back(T&& value)
  post requires(std::is_memoizable<T>)
     [ input_value = std::memoize(value) ]
     ( std::memoization_equals(back(),input_value) );
```

Unlike the earlier postcondition on `push_back`, this version will properly identify defects when `T` is a move-only type such as `std::unique_ptr<U>`.

A postcondition on `std::swap` for memoizable types can be similarly written with a *capture list*:

```
template <typename T>
void swap(T& lhs, T& rhs)
  post requires(is_memoizable<T>)
     [ input_lhs = std::memoize(lhs), input_rhs = std::memoize(rhs) ]
     (     std::memoization_equals(rhs, input_lhs)
        && std::memoization_equals(lhs, input_rhs) );
```

2.2.7 Write a check for "Throws: Nothing"

Your quest for correctness spreads, and you now seek to implement checks for many aspects of plain language contracts on your functions. Each new category of checks that becomes expressible rewards you with measurable decreases in related defects in your software.

Many function contracts guarantee that, when invoked in contract (i.e., violating no preconditions), they do not throw an exception. The `noexcept` specifier fails to fully capture this intent because it covers behavior even when the preconditions of a function have been violated.

In addition to separate `pre` and `post` function contract assertions, a combined, more powerful kind of CCA known as a *procedural function interface*[10] may be used. These interfaces provide a more complete mechanism to check contracts along the entire boundary of a function invocation.

When using an interface, assertions may be placed in a code block that will be executed around the implementation of a function when the interface itself is evaluated:[11]

[10]See [P0465R0].

[11]This particular interface could even have a shorthand provided by the Standard itself in the form of a new contract *kind*, `throws_nothing`. See [P2946R0].

```
void f()
  // throws nothing
  interface {
    try {
      implementation;       // Invoke function.
    } catch (...) {
      [[ assert : false ]]; // should never happen
    }
  };
```

Any code could be placed before or after the call to the function itself indicated by the use of the special identifier `implementation`. Code before `implementation` is akin to checking preconditions, while code after `implementation` is akin to checking postconditions.

2.2.8 Write a check for the strong exception-safety guarantee

You begin to investigate implementing contract checks for an implementation of a sequence container, `mylib::vector`. One clause in the contract of `push_back` — that it provides the strong exception-safety guarantee — has no obviously implementable check using the kinds of contract assertions known to you.

Any code can be placed inside an interface, which supports the same facilities for labeling and having `requires` clauses that other contract assertions do. For copyable elements, the strong exception-safety guarantee can be checked with an interface that makes a copy of the object before invoking the implementation:

```
#include <type_traits>

template <typename T>
class vector {
  // ...
  void push_back(const T& t)
    interface requires std::is_copyable_v<T> {
      vector<T> origValue(*this);
      try {
        implementation;
      }
      catch (...) {
        [[ assert : origValue == *this  ]]
      }
    }
```

The copy of the original value of `*this` is stored in the automatic variable `origValue` in the interface's code block. When an exception is thrown, the strong exception-safety guarantee is checked, and then the exception will be automatically rethrown.

2.2.9 Widen a contract while maintaining backward compatibility

Today you are tasked with providing a replacement for an existing function that has new functionality outside the original function's domain. The replacement must meet the original function contract *when invoked within the original domain. After struggling with convoluted logical expressions, you search for a better tool to express your new function's contract checks.*

The original function you seek to replace has many preconditions and postconditions already checked, but for expository reasons, we'll abstract those into a single instance of each:

```
void orig_f()
  pre ( f_preconditions() )
  post ( r : f_postconditions(r) );
```

Attempting a replacement that accurately captures the situation in which different postconditions apply when the original preconditions were not met requires additional captures and boolean logic. If the invocation happened to be called within the original domain of `f` (i.e., without violating `f_preconditions`), then we are bound to meet the original postconditions of `f`. When invoked outside that domain, a (possibly entirely different) set of postconditions apply:

```
void new_f1()
  pre ( f_preconditions() || new_preconditions() )
  post [ f_preconditions_met = f_preconditions() ]
     ( r : f_preconditions_met ? f_postconditions(r) :
           new_postconditions(r) );
```

Using an `interface` function contract assertion instead, we can express these same ideas — albeit less concisely — in a potentially clearer, more straightforward form by exploiting conventional block-level control flow:

```
void new_f2()
  interface
     if (f_preconditions()) {
       // in domain of f
       const auto& r = implementation;
       contract_assert( f_postconditions(r) );
     }
     else if (new_preconditions()) {
       // outside domain of f, in domain of f2
       const auto& r = implementation;
       contract_assert( new_postconditions(r) );
     }
     else {
       // no preconditions met
       contract_assert( false );
       implementation;  // must still be invoked on all control paths
     }
  }
```

With this interface, you can see that all invocations that would be valid for the original `f` will meet the postconditions of `f`, while only the new postconditions will apply outside that domain. The implementation `new_f2` can safely replace `f` and can then used by newer code with the wider set of inputs.[12]

[12]In other words, `new_f2` is Liskov-substitutable for `f` within the bounds of the checked parts of both of their

2.2.10 Gain a better understanding of interfaces

As you write interfaces to express highly involved contract checks, you might reason that this new contract kind could subsume the need for `pre` and `post`. Can that be true?

The answer is clearly yes. With interfaces available, `pre` and `post` function contract assertions are syntactic sugar, although they are truly useful since they cover the vast majority of contract assertions that typical developers will encounter in their careers.

Any precondition or postcondition function contract assertions can be expressed as an equivalent interface:

```
int f_withprepost(const int x)
  pre ( x > 0 )
  post ( r : r > x );

int f_withinterface(const int x)
  interface {
     contract_assert ( x > 0 );
     implementation;
  }
  interface {
     auto r = implementation;
     contract_assert : r > x ]];
  }
```

Other more advanced features, like capture lists (e.g., `[init_x = x]`) and destructuring return values (i.e., `([a,b])`) for postconditions, can be used within an interface:

```
std::tuple<int,int> g_withpost(int x)
  post [init_x = x] ([a,b] : a < init_x && b > init_x );

std::tuple<int,int> g_withinterface(int x)
  interface {
     auto init_x = x;
     auto [a,b] = implementation;
     contract_assert ( a < init_x && b > init_x );
  }
```

Of course, these interfaces are equivalent yet clearly much more verbose than the corresponding uses of `pre` and `post` function contract assertions. As a tool, the more concise forms are sufficient for most contract-checking tasks.

2.2.11 Check a class's public invariants

Thinking back to that first computer science course you took, fond memories of learning about the benefits of well-designed objects that maintain robust invariants fill your heart with joy. Everything about class invariants sounds like a set of properties that could be checked defensively, but how do you encode that and what will checking the properties accomplish?

contracts. See [liskov94].

Invariant checks can be added to a user-defined class or struct using the `invariant` kind of contract assertion:

```
class MyIntVector {
  // ...
public:
  invariant ( size() < capacity() );
  invariant ( capacity() == 0 || data() != nullptr );
  invariant uncheckable ( is_reachable(begin(), end()) );

  // ...
};
```

Due to the invariants being in the `public` section of the class definition, they will all, by default, be added automatically, with the same labels, in a few places.

- Every public constructor will have all public invariants as postconditions.
- Every public, non`const` nonstatic member function will have all public invariants as preconditions (checked after any declared preconditions) *and* postconditions (checked before any declared postconditions).
- The destructor, if public, will have all public invariants as preconditions.

Invariants in a `protected` or `private` section of a class will apply to all constructors, member functions, and destructors with that access level or a less restrictive one (e.g., a protected invariant will apply to all public or protected member functions).

Member functions that are marked `const` do not check invariants because an often encountered problem when checking invariants ubiquitously is overwhelming costs or unbounded recursive invocations. Note that the invariants declared above on `MyIntVector` would recursively invoke themselves if `size()`, `data()`, `begin()`, or `end()` were non`const` member functions.

2.2.12 Check invariants related to a mutable member

You've been saved a few times from recursive explosions when writing invariant conditions that use `const` member functions. Today, you're dealing with a class that has a `mutable` member that caches some stored data, and invariants related to it must be checked even on your `const` member functions.

You know you need to be careful because an invariant can have the `const` label added to it, which will cause it to be checked on `const` member functions as well:

```
class Values {
  mutable int* d_value1;
  mutable int* d_value2;
public:

  invariant const
     ( (d_value1 == nullptr) == (d_value2 == nullptr) );

  int getValue1() const;
  int getValue2() const;
};
```

On invocations of `getValue1()` and `getValue2()`, this class will lazily initialize its `d_value1` and `d_value2` members. The invariant will be checked for these `const` member functions to guarantee that both member pointers are initialized if either one is initialized.

2.2.13 Check invariants of a function parameter

Today you're writing a friend of a class that has invariants, and you want to check that, when a function is passed objects of that type, those objects are currently in states where their invariants hold. How do you do that?

A `pre`, `post`, or `assert` contract check normally has an expression that converts to `bool`. A contract check with the contract label `check_invariants`, however, will instead inspect the return value of the expression and evaluate all public invariants of the returned expression:

```
template <typename T>
void f(T& t)
  pre check_invariants ( t );
```

Before invoking this function, all public invariants defined by the type `T` will be evaluated for the object `t`. If `T` is not a user-defined type or if it has no public invariants declared, the evaluation of this precondition will have no effect.

Similar labels exist to check the `const`, `protected`, and `private` invariants of a type. Labels on a contract assertion that uses `check_invariants` will be automatically combined with those on each of the individual invariant declarations.

2.2.14 Write function contract assertions for an abstract interface

As the resident expert on writing contract assertions, you have been asked to implement contract checking for the contracts on an abstract interface — i.e., a class with pure virtual functions that is used as the interface to a number of different potential implementations.

As with any other function, function contract assertions may be placed on virtual member functions:

```
struct Base {
  virtual void f()
    pre  ( pre1() )    // #1
    post ( post1() );  // #2
};
```

Any invocation of `Base::f`, whether on a concrete instance of type `Base` or through a pointer or reference, will evaluate the CCAs on `Base::f`:

```
void call_f_directly()
{
  Base b;
  b.f();  // checks #1 and #2
}
void call_f(Base& b)
{
  b.f();  // checks #1 and #2
}
```

Of course, this second invocation within `call_f` does not *know* that `b` refers to an object of type `Base` (the runtime type of this object may be a class derived from `Base` which overrides `f`), but the requirement that the function contract assertions of `Base::f` not be violated still stands with this invocation.

Derived classes that override `f` will have no function contract assertions by default, and the function contract assertions of the base-class function will be checked only when invoking the function through a pointer or reference of that base-class type:

```
  struct Derived : Base {
    void f() override;
  };
  void call_f_with_derived()
  {
    Derived d;
    d.f();  // checks #1 and #2

    Base& b;
    b.f();  // checks nothing
  }
```

The virtual dispatch through `Base& b` will evaluate the CCAs of `Base::f`, while the direct invocation of `Derived::f` will check the (empty) set of CCAs on that function.[13]

2.2.15 Widen the precondition assertions of a concrete implementation

Your next task is to address a series of subclasses in your system that have wider preconditions than your abstract interface requires. How do you implement and take advantage of that?

A derived class may specify function contract assertions on an overriding function that will be checked whenever that particular function is found through dynamic dispatch, or invoked for virtual dispatch:

```
  struct Derived2 : Base {
    void f() override
      pre ( pre2() )    // #3
      post (post2() );  // #4
  };
```

When used through a pointer or reference to the base class, the base-class function contract assertions will be evaluated in addition to those of the derived class function that is found through virtual dispatch.

```
  void call_f_with_derived2()
  {
    Derived2 d2;
```

[13]See [P3097R0] for a more complete exploration of the requirements for contract assertions on virtual functions and the impacts of this proposed design for their support.

```
  Base& b = d2;

  b.f();  // virtual dispatch, checks #1, #3, #4, and #2
}
```

Dispatch through a variable of the derived class, however, will only potentially evaluate the function contract assertions of the derived class, circumventing potentially narrower contract checks from the base class:

```
void call_f_with_derived2_2()
{
  Derived2 d2;
  d2.f();  // checks #3, #4

  Derived2& d2r;
  d2r.f();  // checks #3, #4 (twice)
}
```

When dynamic dispatch through a pointer or reference can be statically determined to find an object of the underlying type of that pointer or reference, the duplicate checks can be reduced by the compiler to only a single set of checks, leaving no observable difference between `d2.f()` and `d2r.f()` above.

The ability to specify distinct function contract assertions on a function override provides complete flexibility to implement derived classes for specialized purposes as well as those that are more general than the interfaces they choose to implement.

2.2.16 Reconcile distinct function contract assertions with multiple inheritance

You have been tasked with developing a computation engine that implements multiple interfaces with distinct requirements. Your implementation will meet these requirements simultaneously, but how do you express that?

Consider two different abstract interfaces for computation:

```
struct ComputeOdd
{
  // Given a positive number, compute an even number.
  virtual int compute(const int x)
    pre  ( x > 0 )
    post ( r : r % 2 == 0 ) = 0;
};
struct ComputeBig
{
  // Given an even number, compute a big number.
  virtual int compute(const int x)
    pre ( x % 2 == 0 )
    post ( r : r > 100 ]] ) = 0;
};
```

Extending both of these base classes and declaring no function contract assertions on the override of `compute` results in a function with no function contract assertions on it:

```
struct MyCompute1 : ComputeOdd, ComputeBig
{
  // Given a number, compute a number.
  int compute(const int x) override;
};
```

Of course, this lack of assertions causes all unit tests for `MyCompute1` to fail to verify that this object would work properly if used as either a `ComputeOdd` or a `ComputeBig`. No contract assertions are being evaluated when `MyCompute1::compute` is invoked directly.

To have a compute implementation that can be reliably used anywhere a `ComputeOdd` or a `ComputeBig` is needed, more specified preconditions and postconditions that are properly consistent with those of both base classes must be provided:

```
struct MyCompute2 : ComputeOdd, ComputeBig
{
  int compute(const int x) override
    pre ( (x > 0) || (x % 2 == 0) )
    post r ( (x > 0) ? (r % 2 == 0 )      // `ComuteOdd` postcondition
    post r ( (x % 2 == 0) ? (r > 100) );  // `ComputeBig` postcondition
};
```

On any use of `MyCompute2` as a `ComputeOdd` or a `ComputeBig`, the base-class precondition checks will be invoked along with the (redundant by construction) CCAs on `MyCompute2::compute`.

Of course, in many cases, these checks might be simplified further when you have an even wider contract or provide stronger guarantees:

```
struct MyCompute3 : ComputeOdd, ComputeBig
{
  int compute(const int x)
    pre ( true )                           // fully wide contract
    post ( r : (r > 100) && (r % 4 == 0) ) // narrower postconditions
};
```

2.2.17 Narrow the function contract assertions of a concrete implementation

Today you took delivery of a new hardware interface that can be used to implement your new computation engine with incredible efficiency. The hardware, however, requires an involved initialization sequence before it can be used effectively. How do you require that initialization sequence be done when such a requirement is not part of the abstract interface your clients are using?

No requirement is made that the preconditions on a derived class be wider than those of a base class; on any invocation involving virtual dispatch, both are checked. Consider a generalized `Compute` abstract interface that takes positive numbers and computes `int` results:

```
struct Compute
{
  virtual int compute(int x)
    pre (x > 0 );
};
```

The `HardwareCompute` class is, of course, free to require initialization as a precondition for its `compute` function:

```
struct HardwareCompute : Compute
{
  void initialize();
  bool isInitialized() const;

  int compute(int x)
    pre ( isInitialized() )
    pre ( x > 0 );
};
```

Such checks will catch cases in which a function working with a `Compute` object by reference has been passed a `HardwareCompute` that was not first properly initialized:

```
void use_compute(Compute& c)
{
  c.compute(100);
}

void bad_hardware_compute()
{
  HardwareCompute h;
  use_compute(h);  // violation because h is not initialized
}

void good_hardware_compute()
{
  HardwareCompute h;
  h.initialize();
  use_compute(h);  // ok
}
```

2.2.18 Prevent invocation of a default virtual function implementation

You've been tasked with extending an existing class hierarchy with new functions that need to be rolled out independently of clients implementing those functions. Only updated objects will have these new functions invoked, but all clients still need to compile so a pure virtual function is not an option. How do you provide an implementation that cannot be invoked?

The preconditions and postconditions on a virtual member function declaration are checked when using that virtual function to dispatch to the family of virtual functions that might override the named function. When a virtual function is used for both virtual dispatch *and* for providing an implementation, function contract assertions for the implementation itself might be implemented using assertion statements:

```
struct S {
  virtual void newFunction() { contract_assert( false ); }
};
```

Though cumbersome, postconditions and interfaces could also be transformed in this fashion, or placed on a separate helper function invoked by the implementation (with corresponding and potentially error-prone forwarding of all parameters an the return value).

Separately, contract assertions that apply only to the implementation — not to using the virtual function for virtual dispatch — can be declared using the `impl_only` label on the function contract assertions:

```
struct S {
  virtual void newFunction() [[ pre impl_only : false ]];
};
```

With the above declaration, the `false` contract assertionswill be evaluated (and violated) only when `S::newFunction` is invoked explicitly or found through virtual dispatch. Any code that tries to use this function with an `S` subclass that has not yet overridden `newFunction` will encounter contract violations.

2.2.19 Write contract assertions for a default virtual function implementation

The next new function you add to your existing class hierarchy has a fairly wide interface to clients, but the default implementation provides very strong guarantees. How do you implement that?

The label `impl_only` can be applied to any function contract assertion, and they will be evaluated when the annotated function is invoked directly or through virtual dispatch. Function contract assertions without that label will be invoked in both cases:

```
struct S {
  virtual int newFunction2()
    post ( r : r >= 0 )             // always non-negative, #1
    post impl_only ( r : r == 0 );  // default is always 0, #2
};

struct T : S {
  int newFunction2() override
    post (r : r >= 0 );  // always non-negative, #3
};

void f()
{
  S s;
  int i = s.newFunction2();  // checks #1, #2

  T t;
  int j = t.newFunction2();  // checks #3, #1

  S& ts = t;
  int k = ts.newFunction2(); // checks #3, #1
}
```

2.2.20 Write code needed by only contract assertions

While migrating a program from the use of `assert()` you find that there are a number of variables used in assertions which are declared and maintained inside `#ifndef NDEBUG` blocks. To implement the same functionality, a new feature that has not come up before is needed

The `contract_support` keyword can be used to denote statements that need only be evaluated when a contract assertion that depends on them will be evaluated with a checked semantic:

```
void f(List *list)
  // The behavior is undefined if the specified list is a linked
  // list with length greater than 5.
{
  contract_support { int length = 0; }  // #1

  while (list) {
    list = list->d_next;

    contract_support { ++length; }      // #2
    contract_assert( length < 5 );      // #3

    // process the next element in the list.
  }
}
```

Each `contract_support` statement injects its contained statements into the enclosing block in two cases:

- When a contract assertion odr-uses something declared in the `contract_support`'s denoted statements, and that contract assertion can be evaluated with a checked semantic (such as *observe* or *enforce*), the statement will be evaluated. For example, the contract assertion `#3` odr-uses `length`, so the declaration of `length` at `#1` will be evaluated (and the variable `length` will exist).
- When a `contract_support` has statements that themselves odr-use a declaration in another `contract_support` block which is evaluated. In this example, the incrementing of `length` at `#2` will be evaluated if the declaration of `length` at `#1` is itself evaluated.

Combining these rules, you can see that allof the statements which maintain the state of a variable declared in a `contract_support` block will be evaluated whenever that variable is needed by a contract assertion.

If, in a particular build the contract assertion at `#3` will have the *ignore* or *assume* semantic then none of the `contract_support` statements in this example will be evaluated at all, the `length` variable will not exist, and no instructions will be evaluated to maintain its value.

2.3 Expert-Level Contract Checking

All the features we've seen so far are higher-level, user-facing tools built on a core set of functionality provided by the C++ language, along with a variety of Standard Library features that have been adopted to make the most common use cases and best practices as easy to implement correctly *as practicable. Occasionally, other needs arise that require a deeper understanding of those core frameworks and more direct leveraging of the core language facilities.*

2.3.1 Write a custom contract label

As you begin to investigate making a suite of custom labels to use as part of a large-scale C++ library, the most basic question that might come to mind is how to create custom label types (to be used in contract assertions) in the first place.

When parsing a contract assertion and searching for contract-label types, the C++ compiler will perform a special form of name resolution to find a C++ user-defined type that is associated with each label. A type can be defined as a contract-label type by adding a `contract_label_id` specifier to its definition. For example, an otherwise empty type can have a label ID (with a namespace) assigned that lets it be used as a label:

```
struct my_label_type contract_label_id(mylib::my_label) {};
```

At any point where this definition is reachable, the label can then be used on a contract assertion:

```
void f() pre mylib::my_label ( true );
```

Label IDs having no namespace are reserved for the Standard Library, and all such label types are defined in the `<contracts>` header.

An empty label type, such as `my_label_type`, will have no effect on the program *other* than to be recorded in the list of labels on the contract assertion, which is available from the `contract_violation` object populated and passed to the contract-violation handler when the contract assertion detects a contract violation at run time.

Other functionality related to labels applies an iterative algorithm that looks for certain members of the label types for each label type in a specified order (left to right or right to left, depending on the property).

- Each type may provide a public member named `allowed_semantics` that must be a type convertible to `std::contracts::contract_semantic_set`. When an implementation is selecting the semantic with which to evaluate a contract assertion, only one of the allowed semantics will be chosen. If the set of allowed semantics is empty, a program will be ill-formed.
- Each type may provide a `static`, `consteval` (or `constexpr`) `compute_semantic` member function that takes and returns a value of the enumeration type `std::contracts::contract_semantic`. This function transforms the semantic determined by the implementation and earlier labels and produces a new resulting semantic. When all label types with `compute_semantic` functions have had that function evaluated, the contract assertion will be evaluated with the resulting semantic. A program is ill-formed if it produces a semantic not in the list of allowed semantics for that contract assertion.
- If two label types on the same contract assertion have the same `dimension` member, the program is ill-formed, allowing labels to express when they are mutually exclusive.
- A label type may provide a `handle_contract_violation` static member. This function will be invoked on contract violations of contract assertions with that label attached. Multiple such

functions will be visited on labels from right (closest to the predicate) to left (furthest from the predicate), followed by the global contract-violation handler. Such local contract-violation handlers may return `false` (or an object that converts to `false`, such as `std::false_type`) and prevent the invocation of earlier contract-violation handlers (or the global contract-violation handler) for the contract assertions to which they are applied.

2.3.2 Provide a new cost-of-evaluation–based contract label

A select few algorithms your advanced graphic library provides have contract assertions that are so expensive to evaluate they must be disabled even in `audit`-level builds. You still, however, wish to be able to occasionally turn the `contracts.cost` knob up even higher to enable them and would love to integrate them cleanly with the existing labels that express contract assertion's cost.

The Standard Library costs are associated with an `enum` defined in `<contracts>` that names the contract-checking levels and gives them arbitrary (yet ordered) numeric values:

```
namespace std::contracts {
  enum class cost : int {
    off          = 0,
    default_cost = 100,
    audit        = 200
  };
}
```

The Standard Library `audit` label is a subclass of a Standard Library template, `cost_label_type`, which has a single parameter of type `std::contracts::cost` and defines a label type having the appropriate behavior, i.e., computing a semantic by comparing its `cost` template parameter to the configured `contract.cost` knob. The type `std::contracts::audit_label_type` is an otherwise empty subclass of an instantiate of that class template, with the appropriate `contract_label_id` specifier:

```
namespace std::contracts {
class audit_label_type
  contract_label_id(audit)
  : public cost_label_type<cost::audit> {};
}
```

The `cost_label_type<cost Cost>` class template provides a suite of functionality that can be leveraged by other subclasses.

- The label type defines a `dimension` member having a single (arbitrary and unspecified) value, making all `cost_label_type` subclasses mutually exclusive.
- The `compute_semantic` member of `cost_label_type` is a `static consteval` function that determines the semantic for contract assertions with a specified `Cost`.
  - When the configured value of `contracts.cost` is equal to or greater than the `Cost` template parameter, the contract assertion will take on the default semantic.
  - Otherwise, when the configured value of `contracts.cost.new` is equal to or greater than the `Cost` template parameter, the contract assertion will apply the same logic as the label

`new` to the default semantic.

  – Otherwise, the contract assertion will have the *ignore* semantic.

With this simple, logical model, a complete range of contract assertion labels can readily be defined. In a checked build, as the `contracts.cost` value is raised, more contract assertions will be *enforced*. When a developer wants to deploy higher checking levels to an existing running system, the `contracts.cost.new` value can be raised first to *observe* the higher-cost checks before moving on to *enforce* them at a later point in time.

Specifying a brand-new label type on the same scale is as simple as providing a new, otherwise-empty label type that leverages `cost_label_type` for its implementation:

```
namespace mylib {
class extra_expensive_label_type
  contract_label_id(mylib::extra_expensive)
  : public cost_label_type<10 * std::contracts::cost::audit>
{};
}
```

With this label in hand, applying the desired behavior to any appropriate check is simple:

```
void optimize_graph(Graph *graph)
  post mylib::extra_expensive ( cubic_time_check(graph) );
```

2.3.3 Manage an ongoing library release cycle

One of your company's products is a library distributed to clients. Some clients adopt each revision as it is released, yet others take only major releases or even skip major releases when no compelling features entice them to upgrade.

The label `new` has worked effectively for adding contract checks safely to clients who upgrade continuously. After a single release cycle, the label is removed and the contract assertion becomes enforceable. Other clients with less frequent upgrade cadences, however, are demanding that they be provided the same functionality: Any contract assertoin that was not included in functions used in the previous version they had actually deployed needs to be observed, not enforced. How can you satisfy such a request?

The problem here is that when we add a contract assertion to a function that might have existed in a previous release, we don't want a client that doesn't update with every release to suddenly get a contract assertion that is enforced. To address this concern, a custom label can be created that captures the library version where the contract check was added and provides the appropriate logic to identify when such a check is being enabled for the first time by that specific client.

Throughout your library, when a contract check is added to a *pre-existing* function in a new library version, add a contract label defined by your library to that contract assertion. When a function is added with contract assertions on it, no particular label is needed:

```
namespace mylib {
  void f(int x)     // added in revision 1.0
    pre ( x > 0 );  // added in revision 1.0
}
```

On the other hand, suppose `mylib::g` gets added in revision `2.0`, but a check for its precondition was missed. In revision `2.1`, the missing contract assertion gets added, but now a label is put on that contract assertion to help guide the checking of this newly introduced contract assertion in code that might already be in use by some clients in production systems:

```
#include <mylib_contracts.h>

namespace mylib {
  void g(int x)                                // added in revision 2.0
     pre mylib::version<2,1,0> ( x > 0 );  // added in revision 2.1
}
```

When clients build their programs, they may specify the last stable version of your library that they had deployed, using a build environment flag specific to their library label:

```
gcc -Kmylib.stableversion=2.0.0
```

Clients that do not specify this version will have the previous version assumed, treating only the most recently introduced contract assertions as if they had the label `new`. Specifying the *current* version as the stable version — something you instruct your clients to do once they have deployed a version for an acceptable period of time — will treat *none* of the marked contract assertions as having the label `new` and will *enforce* all of them.

Regardless of stable version, any code that references `mylib::f` must have been built against a library revision of at least `1.0`, so the precondition check on `mylib::f` must have already been there.

Now consider Client A, who regularly updates with each new release of your library.

- Client A uses the default build out of the box and begins making use of `mylib::g` as soon as they download version `2.0`. The label will determine that the stable version is `1.17`, the last minor version that was released before the upgrade to `2.0`.
- The new function seems to work well and is deployed to production, but the missing CCA means Client A is not being made aware of (possibly subtle) defects that might result from violation the contract of `mylib::g`.
- When upgrading to revision `2.1`, the stable version is now `2.0`, the misuse is detected with a log message, the bug is fixed, and Client A finds their software works more reliably without having had a catastrophic production shutdown.
- When upgrading to revision `3.0`, the stable version is now `2.1`, the precondition check on `mylib::g` is now enforced, and a future mistake or unhandled edge case will be quickly caught before the production system goes completely awry due to a misused function.

Client B, however, takes only major revision releases. Because they have read the documentation of `mylib`, they know to set the stable version to the previous stable version with each upgrade.

- With each new library version, Client B specifies, as they build `mylib`, what their last stable version of the library was. When deploying version `2.0`, they set this value to `1.0`. When deploying version `3.0`, they set this value to `2.0` by adding `-Kmylib.stableversion=2.0` to their build command line.

- Just like Client A, Client B can use the new `mylib::g` in version `2.0` and might use it incorrectly.
- When Client B adopts version `3.0`, the precondition on `mylib::g` will still be treated as having the label `new` since the precondition was added after the configured, last stable version (`2.0`). Had the stable version not been set, Client B would see the check on `mylib::g` *enforced*, resulting in a terminating production system if a defect is discovered. However, unlike Client A, Client B sets the stable version to an earlier one when deploying version `3.0`, and thus the check on `mylib::g` is *observed* and defects can be detected without a production failure being guaranteed.

Each client retains the option to build the software with the *current* version as the last stable version once they are comfortable that no newly introduced contract checks require observation. This choice makes even the newest checks in the library into *enforced* checks, increasing the safety and correctness of each client's running software.

The custom contract label `mylib::version` is defined as a template in `mylib_contracts.h`:

```
// mylib_contracts.h:
#include <contracts>

namespace mylib {

class Version { /*...*/ }; // literal value-semantic abstraction for a version

constexpr Version stable_version();
  // implementation defined accessor for the configured stable version

template <int major, int minor, int patch = 0>
class version_label_type contract_label_id(mylib::version) {

  static consteval std::contracts::semantic compute_semantic(
    std::contracts::kind kind, std::contracts::semantic semantic)
  {
    if (stable_version() < make_version(major,minor,patch)) {
       // Act just like the label new.
       return std::contracts::new_label_type::compute_semantic(kind,semantic);
    }
    else {
       // Do not adjust the semantic.
       return semantic;
    }
  }
};
```

The `contract_label_id` specifier on this class template facilitates name lookup for contract labels finding this template as the implementation for the `mylib::version` label.

2.3.4 Make your own code depend on build configuration options

Most of these Standard Library contract labels will change their behavior based on how you build your program. How is that customization accomplished?

From the uncharted regions of the C preprocessor, source code has long provided mechanisms to configure compile-time and runtime behavior through the specification of preprocessor macros — usually via the `-D` compiler switch — during compilation. This great power provided equally great flexibility but at the cost of many real and perceived downsides of the preprocessor.

The *build environment* provides a flexible, powerful, and more hygienic and type-aware facility to pass an arbitrary set of values from the compiler command line such that those values can be used during constant evaluation.

The Standard Library provides a metafunction to access the values of named and controllable *knobs* in the *build environment*:

```
namespace std::meta {
  consteval char const* get_env_raw(char const* k);
    // Return the value of knob k, or return nullptr if
    // the knob does not have a value set.
}
```

The specific value returned is controlled in an implementation-defined manner, but these extrinsic values can vary based on build configurations — through explicit command-line flags, different defaults when choosing *Debug* or *Release* builds, or other mechanisms. Each Standard Library contract label's behavior is controlled by the values of Standard-specified control points, called *knobs*, in the build environment.

The `gcc` implementation, for example, allows users to set knobs on the command line via the `-K` compiler switch. Consider a simple program that, when built, simply outputs the value of a specific knob (e.g., `"testknob"`) from the build environment:

```
// main.cpp
#include <iostream>
#include <meta>

int main()
{
  std:: cout << "Test Knob Value:"
             << std::meta::get_env_raw("testknob") << std::endl;
}
```

This program can then be built and run in multiple ways by specifying various values for the knob:

```
$ gcc test.cpp
$ ./a.out
Test Knob Value:

$ gcc -Ktestknob=Hello test.cpp
$ ./a.out
Test Knob Value: Hello

$ gcc -Ktestknob=World test.cpp
$ ./a.out
Test Knob Value: World
```

Utilities are also provided to extract knob values as *typed* data, leveraging user-defined literals to parse strings into objects within the `consteval` metafunction framework.

2.3.5 Categorize a group of checks to be enabled as a unit

Working on your company's implementation of a sequence container, you have identified a number of situations in which container operations invalidated iterators that should not have been invalidated by certain operations. All the checks to identify these problems are at least linear in cost and, more importantly, require memory allocations, whereas the underlying function in the same situation would require none. Enabling these checks must happen only in highly specialized builds being run to deeply test all behaviors of your `vector` implementation.

A custom label (e.g., `mylib::itercheck`) can easily specify a completely independent knob in the build environment to control a decision between the *ignore* and *enforce* semantics:

```
namespace mylib {
class itercheck_label_type contract_label_id(mylib::itercheck) {
  static consteval std::contracts::semantic compute_semantic(
    std::contracts::kind kind, std::contracts::semantic semantic)
  {
    if (std::meta::get_env<bool>("mylib::iterchecks",false)) {
      return std::contracts::semantic::enforce;
    }
    else {
      return std::contracts::semantic::ignore;
    }
  }
};
}
```

Each time a CCA with this label is evaluated, this `compute_semantic` function will be invoked with the compiler-selected semantic for that CCA. This implementation of `compute_semantic`, however, will ignore that input and instead select between *enforce* and *ignore* based on the value of the build environment knob `"mylib::iterchecks"`.

Then, on each of the postconditions and interfaces that check iterator invalidation, this label can be applied so that the checks themselves occur in only the specific builds where you have enabled this group of checks:

```
template <typename T>
void mylib::vector<T>::push_back(const T& t)
  interface mylib::iterchecks {
     if (size() == capacity()) {
       // All iterators are invalidated; nothing to check.
       implementation;
     }
     else {
       // Leverage std vector to store iterators.
       std::vector<const_iterator> iters;
       for (const_iterator it = begin(); it != end(); ++it)
       { iters.push_back(it); }
```

```
      implementation;

      for (std::ptrdiff_t ndx = 0; ndx < std::ssize(*this)-1; ++ndx)
      {
        // Verify that each element of iters is the same as
        // the current iterator to that index.
        contract_assert ( iters[ndx] == begin() + ndx );
      }
    }
  ]];
```

In all builds that do not specify a value for the `"mylib::iterchecks"` knob or that specify that value to be `false`, the above interface will be *ignored* and thus do nothing. When the knob is set to `true`, however, the interface will validate that the address of all elements in the vector remains stable whenever `push_back` is invoked while there is available capacity for the new element.

A label such as `mylib::iterchecks` will also interoperate with other labels that transform the semantic that will be used to evaluate a contract assertion. For example, the above interface might use the `new` label when first introduced, resulting in the build environment variable toggling this contract assertion between the *ignore* and *observe* semantics:

```
template <typename T>
void mylib::vector<T>::push_back(const T& t)
  interface mylib::iterchecks new {

     // observed if "mylib::iterchecks" is true in build environment
     // ignored if "mylib::iterchecks" is false in build environment

     // ... same as above ...
  };
```

3 Proposed Features

This section contains discussion of all of the individual, relatively separable features that comprise the complete, robust Contracts facility we envision for C++. Each individual subsection will, in subsequent revisions of this paper, include complete descriptions of the proposals, motivations, usage examples, and a discussion of the chosen design along with various alternatives that were considered.

3.1 Existing Contract Functionality

A brief description of the essential parts of the currently proposed SG21 Contracts facility proposal — [P2900R6].

3.1.1 Nomenclature

A glossary of terms used throughout this paper when discussing Contracts.

3.1.2 Contract semantics

The defined set of semantics with which a contract assertion might be evaluated.

3.2 Supporting Features

Features to be added to Standard C++ to facilitate other features.

3.2.1 Build environment

A metafunction replacement for the preprocessor to allow configuration-built program behavior at compile time.

3.2.2 Standard Library memoization

A Standard Library extension point for defining how the value of a type may be captured for later comparison.

3.2.3 The `symbolic` function specifier

An attribute that indicates a function that is capable of affirmatively verifying certain runtime capabilities but which, in general, cannot reliably detect all such situations or which will invalidate those very capabilities should it actually be evaluated.[14]

3.3 New Contract Kinds

New contract *kinds*.

3.3.1 Procedural function interfaces

The `interface` contract *kind* to specify a block of code that, when checked, will be evaluated around the invocation.

3.3.2 Class invariants

The `invariant` contract *kind* and supporting functionality to specify class invariants, explicitly check them, and control which parts of a class interface have which invariants checked.

3.4 Updates to contract assertions

The proposed *kinds* of contract assertions can benefits from additional featuers in order to provide more expressivity when specifying contract checks.

3.4.1 Postcondition captures

The ability to specify an *init-capture* list on postconditions that is initialized when preconditions would be checked.

[14]This attribute is similar to the special return type mentioned in [P2176R0], but it allows (but does not require) an implementation and is not itself directly tied to the Contracts facility.

3.4.2 Postcondition return value destructuring

The ability to specify a sequence of names that will destructure the return value of a function.

3.4.3 Contract assertion `requires` clauses

Control whether a contract assertion is applicable based on a concept check on template parameters.

3.4.4 `contract_support` statements

The `contract_support` statement marks arbitrary other statements to only be conditionally evaluated when they are needed by a contract assertion that itself will be evaluated.

3.5 Contract Labels

Contract checks of any kind do not, on their own, provide all of the information a human or a compiler needs to understand the context of the check — when it should be checked, how its failure should be interpretted, and other nuances of how it might be handled. To provide that facility, we present a mechanism for producing user-defined labels that are associated with C++ types, along with how the contents of those types will control the semantics and other behaviors of the contract assertions to which the labels are applied.

3.5.1 Specifying labels

The framework to tie a class definition to a label and specify a sequence of (possibly parameterized) labels applied to a contract assertion.

3.5.2 Contract assertion semantic computation

When determining the semantics with which to evaluate a contract assertion, a metafunction on the label types will be used to transform the implementation-defined chosen semantic into the effective semantic.

3.5.3 Allowed semantics

Labels may specify a set of semantics that are the potential results of the computation semantics. Contract assertions with particularly restricted sets of allowed semantics have additional properties, such as no ODR use for a contract assertions that has no allowed checked semantics.

3.5.4 Local contract-violation handlers

A label may specify a contract-violation handler function that will be invoked prior to or instead of the global contract-violation handler.

3.5.5 Label dimensions

Labels may specify that they represent a particular dimension, and multiple labels having the same dimension are mutually exclusive within any given contract assertion.

3.5.6 Standard Library — Labels for concrete semantics

Standard library labels to select specific concrete Semantics with which a contract assertion should be evaluated, often useful for testing things directly related to the Contracts facility itself.

3.5.7 Standard Library — Labels for cost of evaluation

Labels for contract assertions to express the expected cost of evaluating the contract assertion's predicate, usable to selectively *not* check expensive conditions except in specialized builds that might not be universally deployable.

3.5.8 Standard Library — A label for newly introduced contract checks

A label to mark a contract check as one that has been newly introduced into already existing software, influencing the semantic to be *observe* in most conditions where they would otherwise be *enforce*.

3.5.9 Standard Library — A label for unchecked contracts

A contract label that indicates a predicate that should never be evaluated at run time and thus is useful for only static analysis, optimization, and documentation purposes.

3.6 Behavioral Improvements

The MVP deliberately did not pursue certain aspects of Contracts in order to maximize consensus and focus on a minimal, useful product on which we can build. Many of those decisions should be revisitted to provide a maximally flexible facility that meets the many needs users will have as Contracts become part of every C++ developer's daily workload.

3.6.1 The *assume* semantic

A semantic for contract assertions that does not evaluate the predicate and introduces undefined behavior on violations.

3.6.2 Contract-violation handler updates

Minor updates to the `contract_violation` object to capture the rest of the changes in this proposal, such as adding an accessor for the labels on a contract assertion when it is violated.

3.6.3 Contract assertions on virtual functions

Specification of how contract checks on virtual functions can vary across class hierarchies and which checks will be validated on any given virtual dispatch.

The labels `impl_only` and `virtual_only` denote a contract assertion as being invoked when a function is directly invoked or invoked via virtual dispatch, respectively.

3.6.4 Contract assertionsand where to find them

Extending the ability to specify function contract assertions on declarations that are not the first declaration of a function, with allowances for potentially repeating function contract assertions and invoking functions before having seen the function contract assertions attached to the invoked function.

3.6.5 Function contract assertions on trivial functions

The ability to specify function contract assertions on a special member function that is otherwise trivial by using a `trivial` label to indicate that, when the trivial function is implicitly invoked the corresponding function contract assertions may not be invoked.

3.6.6 Pack expansion of contract assertions

The ability to apply the pack expansion operator to a contract assertion, in a manner similar to which it can be applied to an attribute.

3.6.7 Integration with `<cassert>`

Changes to `<cassert>` to invoke the contract-violation handler when an assertion fires (prior to invoking `std::abort()` as currently specified) and a discussion about why no other changes can or should be proposed to `<cassert>`.

3.6.8 Integration with core-language UB

A proposal regarding the ability to attach contract labels to occurrences of checkable, language-undefined behavior within a scope, enabling the ability to, for instance, have all pointer dereferences checked that the pointer value is not null with the *enforce* semantic.

3.6.9 Precondition checking opreator

A new operator that may be applied to a function invocation that will return an `optional<std::contracts::contract_` indicating whether the given invocation would lead to a contract violation prior to evaluating any non-vacuous operations, returning immediately instead of evaluating any such non-contract assertion non-vacuous operations.

4 Real World Examples

In this section are various larger use cases that exhibit how various larger endeavours mike be accomplished with the pieces made available by this proposal.

4.1 Unit Testing CCAs

This subsection will present a complete example of how to manually perform negative testing on a function using the `precheck` operator as well as examples of how this testing can be incorporated into macros or generic functions.

4.2 Fuzz Testing Functions with Narrow Contracts

We'll show, via examples, how to write fuzz tests that use the `precheck` operator to test only in-contract invocations, and we'll discuss the various ramifications of this form of testing.

4.3 Third-Party Contract Labels

Various examples of third-party contract labels can be developed for a many purposes.

4.4 Contract assertions for `std::vector`

Two frequent questions come up in the context of the use of Contracts and the Standard Library:

1. Should the Standard Library specification make use of contract assertions in lieu of prose descriptions that currently exist for preconditions and postconditions?
2. How thoroughly can the contract specified by the Standard Library be asserted with contract assertions as specified by the Contracts facility?

Obviously, the answer to the first question is strictly bounded by the answer to the first question.

Until there is wider-spread availability of Contracts in the wild we strongly recommend that the Standard Library itself not adopt the use of contract assertions in its specification — such an adoption precludes implementation freedom to check function contracts using different mechanisms or provide conforming extensions that take advanatage of the undefined behavior currently in the standard. For that reason, it may be advisable to never make use of contract assertions in the Standard Library specification itself.

Even with no requirement in the Standard to use contract assertions, both the violation of a Standard Library precondition and a bug in a Standard Library implemenation that would lead to a failure to meet a postcondition are *undefined behavior*. Given that, an implementation is free to use a contract assertion to define that undefined behavior and invoke a contract violation, meaning any tools we provide can be used to benefit Standard Library users as soon as they are available, without any need to include the contract assertions in the Library specification.[15]

The first question, however, is an important one to guage the meaningfulness of the various features being proposed by both the Contracts MVP ([P2900R6]) and the extensions put forth in this and other papers.

To that end, we will explore, to the best of our ability, the range of contract assertions that might be placed on a container such as `std::vector`, calling out how we might leverage different features of both the Contracts MVP and this paper to produce as complete an example as possible of a `std::vector` implementation that can be fully checked for correctness and which exposes as much information as possible to static analysis tools.

To begin with, let's assume that our `vector` implementation uses the relatively common strategy of storing three pointers and an allocator as the private data members of `vector`:

[15]An LEWG policy regarding the use of contract assertions in the Standard Library should be adopted once Contracts are adopted into the working draft, following the procedures put forth in [P2267R0] and potentially the approach outlined in [P3005R0], should be developed and agreed upon prior to any use of Contracts in the Standard Library specification.

```
namespace std {

  template<class T, class Allocator = allocator<T>>
  class vector {
  public:
    // ... typedefs as specified in the standard

    using value_type     = T;
    using allocator_type = Allocator;
    using pointer        = typename allocator_traits<Allocator>::pointer;

  private:
    // data members

    pointer d_first;
    pointer d_last;
    pointer d_endOfStorage;
    allocator_type d_alloc;

  public:
    // member functions described below

    // ...
  };

}  // close namespace std
```

To begin specifying contract assertions for `vector`, we will start with some class invariants on the relationships between our pointer data members:

```
// public within std::vector
invariant (
  (d_first == nullptr) == (d_last == nullptr) == (d_endOfStoraget == nullptr)
);
invariant (
  d_first == nullptr || d_first <= d_last;
);
invariant (
  d_last == nullptr || d_last <= d_endOfStorage;
);
```

Next, we want to guarantee that the half-open range `[d_first,d_last)` is always a valid range of `value_type` elements:

```
invariant audit uncheckable (
  is_valid_range(d_first, d_last);
);
```

Due to the general inability to determine if two arbitrary pointers form a valid range with fully defined behavior, this invariant makes use of a `symbolic` function that declares this to be the case, `is_valid_range`. Because that function will be neccessarily linear (at least), it is also marked `audit`.

Next, we must consider that there is another requirement, which is that whenever it is not nullptr the region of memory delineated by d_first and d_endOfStorage is one that was allocated from d_allocator. For this, we might assume that some allocator types support an additional member function which identifies regions of memory allocated by that allocator. For such allocator types, we can add the additional invariant assertion:

```
invariant
  requires requires { d_alloc.is_allocated(d_first, d_endOfRegion) -> bool
  ( d_first == nullptr || d_alloc.is_allocated(d_first, d_endOfRegion) );
```

Now that we have our invariants defined, we can discuss some of the strategies tha will be used to check the various guarantees provided by vector's numerous member functions.

A frequent operation we will need to perform is to store a copy of the elements of our vector. To do this, we could use vector itself, but that would inevitably lead to cyclic invocations of the same contract assertions as we attempt to make our copies, which will fail. In general, the function contract assertions of a type should depend only on lower-level members of the type (forming an acyclic graph of dependencies between the members of the type) and lower-level entities outside of the type. So for this purpose, let's assume we have implemented a simple_vector that supports no modification, only const iteration, and can be constructed from a pair of pointers to copyable elements:

```
template <typename T>
class simple_vector {
  public:
    simple_vector(T* begin, T* end);
      pre uncheckable (is_reachable(begin,end))
      pre (begin == end || begin != nullptr)
      post ( size() == end-begin )
      post audit ( std::equal( begin, end, begin()) );

    T* begin() const;
    T* end() const;
    std::size_t size()
      post( r : r == end() - begin() );
};
```

With this simple_vector, we can add a customization for memoization that allows for memoization whenever our value_type is itself memoizable:

```
friend simple_vector<memoize_t<value_type>>
  memoize(const vector& v) requires memoizable<value_type>
{
  return simple_vector(v.d_first, v.d_last);
}
friend bool memoization_equals(const vector&                              v,
                               const simple_vector<memoize_t<value_type>>& m)
  requires memoizable<value_type>
{
  return std::equal( v.d_first, v.d_last, m.begin() );
}
```

Importantly, we have defined these friends to not make any use of any of the public member funtions of `vector`, and thus all of those members functions can make free use of memoization in their own function contract assertions.

Now, some contract assertions we might right could make very heavy-handed use of this memoization to verify that our functions which are marked `const` actually make no modifications to the salient state of our vectors or their elements. Due to the excessive cost of such checks, of course we will only want them enabled when we opt into them by enabling a custom label, as well as marking them with the standard `audit` label:

```
void const_function() const
  post audit stdlib::const_correctness
    requires memoizable<decltype(*this)>
    [ prev : std::memoize(*this) ] // store a copy of elements
    ( std::memoization_equals(prev, *this) );
```

Occasionally, functions which might throw are also `const`, and in these cases we want to additionally guarnatee the container's state remains unchanged when an exception is thrown.

```
void const_function() const
  interface audit stdlib::const_correctness
    requires memoizable<decltype(*this)>
    {
      auto memoization = std::memoize(*this);
      try { implementation; }
      catch (...) {}
      contract_assert(
        std::memoization_equals(*this, memoization) );
    };
```

One additional consideration for `vector` is that it has a non-salient property beyond its elements — its allocator. For const functions, in addition to the elements not changing we expect te allocator to not change. This requires that the allocator type be memoizable:

```
void const_function() const
  interface stdlib::const_correctness
    requires memoizable<allocator_type>
    {
      auto orig_alloc = std::memoize(get_allocator());
      try { implementation; }
      catch (...) {}
      contract_assert(
        std::memoization_equals(get_allocator(), orig_alloc) );
    };
```

All of these interfaces put together are clearly verbose, but it also identical for all `const` member functions (even those outside of `vector`), and thus we will use a preprocessor macro instead of repeating this requirement on all `const` member functions:[16]

[16]Yes, macros are ugly, but they are available and used regularly by real software engineers. A suitably powerful hygienic macro facility should take this use case under advisement and then this use of macros would not be needed.

```
#define POST_CONST_CORRECT                              \
  interface audit stdlib::const_correctness             \
    requires memoizable<decltype(*this)>                \
    {                                                   \
      auto memoization = std::memoize(*this);           \
      try { implementation; }                           \
      catch (...) {}                                    \
      contract_assert(                                  \
        std::memoization_equals(*this, memoization) );  \
    }                                                   \
  interface stdlib::const_correctness                   \
    requires memoizable<allocator_type>                 \
    {                                                   \
      auto orig_alloc = std::memoize(get_allocator());  \
      try { implementation; }                           \
      catch (...) {}                                    \
      contract_assert(                                  \
        std::memoization_equals(get_allocator(),        \
                                orig_alloc) );          \
    }
```

Similarly, there are a number of member functions that provide the strong exception-safety guarantee, for these we will define a similar macro that memoizes our object and verifies that it has not changed should an exception be thrown. Again, we apply the `audit` label and another library-specific label that controls checks on exception guarantees:

```
#define STRONG_GUARANTEE                                              \
  interface audit stdlib::exception_guarantees                        \
    requires memoizable<decltype(*this)> {                            \
    auto memoization = std::memoize(*this);                           \
    try {                                                             \
      implementation;                                                 \
    } catch (...) {                                                   \
      contract_assert( std::memoization_equals(*this, memoization) ); \
    }                                                                 \
  }
```

Now we can move on to the most basic accessors within `vector` on which we will build our other semantic guarantees:

```
constexpr T* data() noexcept
  post( r : r == d_first )
  POST_CONST_CORRECT;
constexpr const T* data() noexcept
  post( r : r == d_first )
  POST_CONST_CORRECT;
constexpr std::size_t size() const noexcept
  post( r : r == (d_last - d_first) )
  POST_CONST_CORRECT;
constexpr allocator_type get_allocator() const noexcept
  post requires(equality_comparable<allocator_type>)
       ( r : r == d_alloc )
```

```
    POST_CONST_CORRECT;
```

Note that even though it is not `const`, the non-`const` `data` function (and many of the other non-`const` member fuctions like) still guarantees that it will not modify the contents of the `vector`. Obviously, the non-`const` acessors still provide a reference to the contents of the vector which is mutable and thus the `const` guarantee will not apply after the member function completes.

Next we get to some functions that, beyond simply stating what they return in terms of the data members of `vector` indicate some additional semantic guarantees:

```
  constexpr std::size_t capacity() const noexcept
    post( r : r == (d_endOfStorage - d_first) )
    post( r : r >= size() )  // capacity is never less than size
    POST_CONST_CORRECT;

  [[nodiscard]] constexpr bool empty() const noexcept
    post( r : r == (size() == 0) )
    POST_CONST_CORRECT;
```

The variations on `begin` and `end` cna have their correctness expressed in terms of the data members of `vector`, or with more semantic meaning in terms of the more publicly visible interface of `vector`:

```
  constexpr iterator begin() noexcept
    post( r : r == d_first )

    // returned iterator points to the first element of data
    post( r : (empty() && r == end()) || &*r = data() )

    POST_CONST_CORRECT;

  constexpr const_iterator begin() const noexcept
    post( r : r == d_first )
    post( r : (empty() && r == end()) || &*r = data() )
    POST_CONST_CORRECT;

  constexpr const_iterator cbegin() const noexcept
    post( r : r == d_first )
    post( r : (empty() && r == end()) || &*r = data() )
    POST_CONST_CORRECT;

  constexpr iterator end() noexcept
    post( r : r == d_last )

    // returned iterator points one past the end of data
    post( r : empty() || &*r == data() + size() )

    POST_CONST_CORRECT;

  constexpr const_iterator end() const noexcept
    post( r : r == d_last )
    post( r : empty() || &*r == data() + size() )
    POST_CONST_CORRECT;
```

```
constexpr const_iterator cend() const noexcept
  post( r : r == d_last )
  post( r : empty() || &*r == data() + size() )
  POST_CONST_CORRECT;
```

For the reverse begin and end functions the first postcondition we specify is even more implementation-specific as we depend upon the use of `std::reverse_iterator`, and it becomes more clear that the more semantic postconditions in terms of the interface of `vector` express (and verify) what is happening more clearly:

```
constexpr iterator rbegin() noexcept
  post( r : r == reverse_iterator(d_last) )

  // returned iterator points to the last element of data
  post( r : (empty() && r == rend()) || (&*r == data() + size() - 1))

  POST_CONST_CORRECT;

constexpr const_iterator rbegin() const noexcept
  post( r : r == const_reverse_iterator(d_last) )
  post( r : (empty() && r == rend()) || (&*r == data() + size() - 1))
  POST_CONST_CORRECT;

constexpr const_iterator crbegin() const noexcept
  post( r : r == const_reverse_iterator(d_last) )
  post( r : (empty() && r == crend()) || (&*r == data() + size() - 1))
  POST_CONST_CORRECT;

constexpr iterator rend() noexcept
  post( r : r == reverse_iterator(d_first) )

  // returned iterator points one before the beginning of data
  post( r : empty() || &*r == data() - 1 );

  POST_CONST_CORRECT;

constexpr const_iterator rend() const noexcept
  post( r : r == d_last )
  post( r : empty() || &*r == data() - 1 );
  POST_CONST_CORRECT;

constexpr const_iterator crend() const noexcept
  post( r : r == d_last )
  post( r : empty() || &*r == data() - 1 );
  POST_CONST_CORRECT;
```

For `max_size`, there isn't a specific requirement in the Standard as to what it specifies, but we might choose to allow up to the maximum number of elements that could be allocated:

```
std::size_t max_size() const
  post( r : r == allocator_traits<allocator_type>::max_size(d_alloc))
```

```
    POST_CONST_CORRECT;
```

Now we can move on to the constructors:

```
  constexpr vector() noexcept(noexcept(Allocator()));
    post( empty() )          // guaranteed by standard
    post( capacity() == 0); // implementation guarantee

  constexpr explicit vector(const Allocator& alloc) noexcept
    post( empty() )          // guaranteed by standard
    post( capacity() == 0); // implementation guarantee
    post requires(std::equality_comparable<Allocator>)
         ( get_allocator() == alloc );

  constexpr explicit vector(const size_type n,
                            const Allocator& alloc = Allocator())
    post( size() == n && capacity() >= size() )
    post requires(std::equality_comparable<Allocator>)
         ( get_allocator() == alloc );

    // if memoizable, then all elements have the same memoization as a
    // default-constructed value_type
    post requires(memoizable<value_type>)
         (std::all_of(data(), data() + n,
                      [](auto && x){
                        return std::memoization_equals(x,std::memoize(T{}));
                      }););

  constexpr vector(const size_type n,
                   const T& value,
                   const Allocator& alloc = Allocator())
    post( size() == n && capacity() >= size() )
    post requires(std::equality_comparable<Allocator>)
         ( get_allocator() == alloc );

    // all n elements compare equal to the given value
    post requires(std::equality_comparable<T>)
         ( std::all_of(data(), data()+n, [](auto && x){return x == value}) );

  template<class InputIterator>
  constexpr vector(InputIterator first,
                   InputIterator last,
                   const Allocator& alloc = Allocator())
    pre uncheckable ( is_reachable(first, last) )

    // resulting size is distance between first and last
    post
      uncheckable<!std::forward_iterator<InputIterator>>
        // destructive for input iterators
      audit<!std::random_accesss_iterator<InputIterator>>
        // too complex for non-random-access iterators
      [ d = distance(first, last) ]
      ( size() == d)

    // elements of vector compare equal to the inputed range
    post audit
      requires(std::equality_comparable<T>) :
      std::equal(first, last, data()) ]]

    post requires(std::equality_comparable<Allocator>)
         ( get_allocator() == alloc );

  constexpr vector(const vector& x)
    post requires(std::equality_comparable<Allocator>)
         ( get_allocator() ==
           std::allocator_traits<Allocator>::
                select_on_container_copy_construction(x.get_allocator()) )

    post requires(std::equality_comparable<T>)
         ( *this == x );

  constexpr vector(vector&& x) noexcept
    // compares equal to the moved-from vector prior to move
    post requires(std::is_memoizable<T>)
         [ y = std::memoize(x) ]
         ( std::memoization_equals(y, x) )

    post requires(std::equality_comparable<Allocator>)
         ( get_allocator() == x.get_allocator() );

  constexpr vector(const vector& x, const Allocator& alloc)
    post requires(std::equality_comparable<T>)
         ( *this == x )

    post requires(std::equality_comparable<Allocator>)
         ( get_allocator() == alloc );

  constexpr vector(vector&& x, const Allocator& alloc)
    // compares equal to the moved-from vector prior to move
    post requires(std::is_memoizable<T>)
         [ y = std::memoize(x) ]
         ( std::memoization_equals(y, x) )

    post requires(std::equality_comparable<Allocator>)
         ( get_allocator() == alloc );

  %
  template <class T>
  constexpr vector(initializer_list<T> il, const Allocator& alloc = Allocator())
    // elements of the vector are equal to the elements of the initializer list
    post( size() == il.size() && std::equal(il.begin(), il.end(), data()) );
```

There is not much we can say usefully about the postconditions of our destructor, although it is possible that some allocators might provide a mechanism to determine if a certain range has been deallocated (though this would require not reusing addresses):

```
constexpr ~vector()
  post
    requires requires { d_alloc.is_deallocatoed(d_first, d_endOfregion) -> bool }
    [ orig_first = d_first, orig_end = d_endOfRegion ]
    ( d_alloc.is_deallocated( orig_first, orig_end ) );
```

Next we get to the assignment operators, which use many of the same tools as the constructors:

```
constexpr vector& operator=(const vector& x)
  // return value is reference to self
  post (r : &r == this )

  // elements are equal to the copied-from vector
  post requires(std::equality_comparable<T>)
       ( *this == x )

  // If std::allocator_traits<allocator_type>::propagate_on_container_copy_assignment::value
  // is true, the allocator of *this is replaced by a copy of other.
  post requires(std::equality_comparable<Allocator>)
       requires( std::allocator_traits<Allocator>
                 ::propagate_on_container_copy_assignment::value )
       ( get_allocator() == x.get_allocator() )
  post requires(std::is_memoizable<Allocator>)
       requires(!std::allocator_traits<Allocator>
                 ::propagate_on_container_copy_assignment::value )
       [ orig_alloc = std::memoize(get_allocator()) ]
       ( std::memoization_equals(orig_alloc, get_allocator()) );

constexpr vector& operator=(vector&& x)
  noexcept(allocator_traits<Allocator>::propagate_on_container_move_assignment::value ||
           allocator_traits<Allocator>::is_always_equal::value)
  // return value is reference to self
  post (r : &r == this )

  // elements are equal to the previous elements of the moved-from vector
  post requires(std::is_memoizable<T>)
       [ y = std::memoize(x) ]
       ( std::memoization_equals(y, *this) );

  // If std::allocator_traits<allocator_type>::propagate_on_container_move_assignment::value
  // is true, the allocator of *this is replaced by a move of other.
  post requires(std::equality_comparable<Allocator>)
       requires( std::allocator_traits<Allocator>
                 ::propagate_on_container_move_assignment::value )
       ( get_allocator() == x.get_allocator() )
  post requires(std::is_memoizable<Allocator>)
       requires(!std::allocator_traits<Allocator>
                 ::propagate_on_container_move_assignment::value )
       [ orig_alloc = std::memoize(get_allocator()) ]
```

```
        ( std::memoization_equals(orig_alloc, get_allocator()) );

  template <class U>
  constexpr vector& operator=(initializer_list<U> il)
    // return value is reference to self
    post (r : &r == this )

    // elements are equal to the elements of the initializer list
    post ( size() == il.size() &&
           std::equal(il.begin(), il.end(), data()) );
```

For the remaining member functions of `vector`, none are supposed to change the allocator. We have already expressed this postcondition for assignment operators when the corresponding propogation trait is `false`, but for other functions this postcondition is checkable as long as the allocator type can be memoized.

```
  #define ALLOCATOR_STABLE                               \
    post requires(std::is_memoizable<Allocator>)         \
         [ orig_alloc = std::memoize(get_allocator()) ] \
         ( std::memoization_equals(orig_alloc, alloc) );
```

Now we can move on to the overloads of the `assign` function:

```
  template<class InputIterator>
  constexpr void assign(InputIterator first, InputIterator last)
    // last is reachable from first (uncheckable for destructive iterators)
    pre uncheckable (is_reachable(first, last) );

    // elements are equal to iterated values (uncheckable for input iterators)
    post uncheckable<!std::forward_iterator<InputIterator>>
         requires(std::equality_comparable<T>)
         ( size() == std::std::distance(first, last) &&
           std::equal(first, last, data()) )

    ALLOCATOR_STABLE;

  constexpr void assign(const size_type n, const T& value)
    // size is n
    post ( size() == n )

    // all element are equal to the specified value
    post requires(std::equality_comparable<T>)
         ( std::all_of(data(),
                       data() + size(),
                       [&u](auto && x){return x == value;}) )

    ALLOCATOR_STABLE;

  constexpr void assign(initializer_list<T> il)
    // size is the size of il
    post( size() == il.size() )
    // elements are equal to elements in il
    post requires(std::equality_comparable<T>)
         ( std::equal(il.begin(), il.end(), data()) )

    ALLOCATOR_STABLE;
```

The overloads of `resize` make certain guarantees about size, as well as the values that will be present in the container afer the operation completes. In addition, there are guarantees that elements have not moved if the requested size is lower than the original capacity:

```
  constexpr void resize(const size_type sz)
    // size is sz
    post ( size() == sz )

    // elements up to sz are equal to their previous values (if any) and
    // elements beyond the prevous size (if any) are default constructed
    post requires(std::is_memoizable<T>) [ orig = std::memoize(*this) ]
         ( std::equal(cbegin(),
                      cbegin() + std::min(orig.size(), sz),
                      orig.cbegin(),
                      [](const T& t, const std::memoize_t<T>& m)
                        {std::memoization_equals(m, t);}) &&
           std::all_of(cbegin() + std::min(orig.size(), sz),
                       cend(),
                       [](const T& x){return x == T{};}) )

    // if sz is within previous capacity, iterators before sz are not invalidated
    post [ orig_data = &data(), orig_cap = capacity() ]
         ( sz > orig_cap || orig_data == &data() )

    ALLOCATOR_STABLE;

  constexpr void resize(const size_type sz, const T& value)
    // size is sz
    post ( size() == sz )

    // elements up to sz are equal to their previous values (if any) and
    // elements beyond the prevous size (if any) are equal to value
    post requires(std::is_memoizable<T>) [ orig = std::memoize(*this) ]
         ( std::equal(cbegin(),
                      cbegin() + std::min(orig.size(), sz),
                      orig.cbegin(),
                      [](const T& t, const std::memoize_t<T>& m)
                        {std::memoization_equals(m, t);}) &&
           std::all_of(cbegin() + std::min(orig.size(), sz),
                       cend(),
                       [&c](const T& x){return x == c;}) )

    // if sz is within previous capacity, iterators before sz are not invalidated
    post [ orig_data = &data(), orig_cap = capacity() ]
         ( sz > orig_cap || orig_data == &data() )
    ALLOCATOR_STABLE;
```

As we progress through the member functions, fewer new considerations arise:

```
  constexpr void reserve(const size_type n)
    // new capacity is not less than n
    post ( capacity() >= n )

    // new capacity is not less than old capacity
    post [ orig_capacity = capacity() ]
         ( capacity() >= orig_capacity )

    // contents of *this are unchanged
    post requires(std::is_memoizable<T>)
         [ orig = std::memoize(*this) ]
         ( std::equal(cbegin(), cend(), orig.cbegin()) );

    // references are not invalidated if capacity did not increase
    post [ orig_data = &data(), orig_capacity = capacity() ]
         ( (orig_capacity < capacity()) ? true : (orig_data == &data()))

    ALLOCATOR_STABLE;

  constexpr void shrink_to_fit()
    // does not increase capacity()
    post [ orig_capacity = capacity() ]
          ( capacity() <= orig_capacity() )

    // capacity should always exceed size
    post ( capacity() >= size() )

    // elements are unchanged
    post requires(std::is_memoizable<T>)
         [ orig = std::memoize(*this) ] :
         ( std::memoization_equals(orig, this) )

    // references are invalidated if and only if capacity actually decreases
    post [ orig_data = &data(), orig_capacity = capacity() ] :
         (orig_capacity > capacity()) == (orig_data == &data()) )

    ALLOCATOR_STABLE;
```

Now we get to one of the simplest yet mostimportant functions in the standard library with a narrow contract:

```
  constexpr reference operator[](const size_type n)
    // n is within the size of the vector
    pre ( n < size() )

    // the return value references the nth element
    post (r : &r == data() + n )
    POST_CONST_CORRECT;

  constexpr const_reference operator[](const size_type n) const
    pre ( n < size() )
    post (r : &r == data() + n )
    POST_CONST_CORRECT;
```

And the corresponding wide-contract function that throws when out of range access is requested, which we validate with a procedural interface:

```
  constexpr const_reference at(const size_type n) const
    // out_of_range is thrown if n is not within the size
    interface {
       bool threw = false;
       try { implementation; }
       catch (std::out_of_range&) { threw = true; }
       contract_assert( threw == (size() <= n) );
    }

    // the return value (on normal return) references the nth element
    post (r : &r == data() + n );

    POST_CONST_CORRECT;

  constexpr reference at(const size_type n)
    interface {
       bool threw = false;
       try { implementation; }
       catch (std::out_of_range&) { threw = true; }
       contract_assert( threw == (size() <= n) );
    }
    post (r : &r == data() + n );
    POST_CONST_CORRECT;
```

Accessors for the first and last element have preconditions that there is such an element and, like many other similar functions, even when not `const` do not modify the state of the `vector`:

```
  constexpr reference front()
    // is not empty
    pre( !empty() )

    // the return value references the first element
    post (r : &r == data() )

    POST_CONST_CORRECT;

  constexpr const_reference front() const
    pre( !empty() )
    post (r : &r == data() )
    POST_CONST_CORRECT;

  constexpr reference back()
```

```
    // is not empty
    pre( !empty() )

    // the return value references the first element
    post (r : &r == data() + size() - 1 )

    POST_CONST_CORRECT;

  constexpr const_reference back() const
    pre( !empty() )
    post (r : &r == data() + size() - 1 )
    POST_CONST_CORRECT;
```

The threeway comparison function is one where a procedural interface makes for a particularly more readable rendering:

```
  template<class T, class Allocator>
  constexpr synth-three-way-result <T>
  operator<=>(const vector<T, Allocator>& x,
              const vector<T, Allocator>& y)
    interface {
      const auto r = implementation;
      const auto m = std::mismatch(x.cbegin(), x.cend(),
                                   y.cbegin(), y.cend());
      if (0 == r) {
        contract_assert( m.first  == x.cend() );
        contract_assert( m.second == y.cend() );
      }
      if (0  > r) {
        contract_assert( m.first  != x.cend() );
        contract_assert( (m.second == y.cend()) ||
                          (*(m.first) > *(m.second)) );
      }
      if (0  < r) {
        contract_assert( m.second != y.cend() );
        contract_assert( (m.first == x.cend()) ||
                         (*(m.first) < *(m.second)) );
      }
    };
```

The `emplace` function must contend with the fact thta arguments cannot, in general, be moved-from multiple times. This might be an opportunity to extend memoization to include the ability to produce values mimicking constructors, alowing you to produce a memoization of the object htat would be created from `args...`.

```
  template<class... Args> constexpr reference emplace_back(Args&&... args)
    // After emplacing, element at back is as if constructed from args.
    // Uncheckable because evaluating args may be destructive.
    post uncheckable ( !empty() && T{args...} == back() )

    // all preexisting elements are unchanged
    post requires(std::is_memoizable<T>)
```

```
        [ orig = std::memoize(*this) ]
        ( orig.size() < size() &&
          std::equal(orig.cbegin(),
                     orig.cend(),
                     cbegin(),
                     [](const T& x, const std::memoize_t<T>& m)
                         { return std::memoize_equals(m, x); }) )

    // size() increases by 1 if successful, otherwise is unchanged
    interface {
        auto orig_sz = size();
        try {
            implementation;
            contract_assert( orig_sz + 1 == size() );
        }
        catch (...) {
            contract_assert( orig_sz == size() );
        }
    }

    // references are invalidated if and only if reallocation takes place
    post [ orig_data = data(), realloc = (size() == capacity()) ]
        ( realloc || (orig_data == data()) )

    STRONG_GUARANTEE
    ALLOCATOR_STABLE;
```

The `push_back` function, however, can compare to const arguments or memoize its argument when the value type supports it:

```
  constexpr void push_back(const T& x)
    // the new back element is equal to x
    post requires(std::equality_comparable<T>)
         ( !empty() && x == back() )

    // all preexisting elements are unchanged
    post requires(std::is_memoizable<T>)
        [ orig = std::memoize(*this) ]
        ( orig.size() < size() &&
          std::equal(orig.cbegin(),
                     orig.cend(),
                     cbegin(),
                     [](const T& x, const std::memoize_t<T>& m)
                         { return std::memoize_equals(m, x); }) )

    // size() increases by 1 if successful, otherwise is unchanged
    interface {
        auto orig_sz = size();
        try {
            implementation;
            contract_assert( orig_sz + 1 == size() );
        }
        catch (...) {
            contract_assert( orig_sz == size() );
        }
    }

    // references are invalidated if and only if reallocation takes place
    post [ orig_data = data(), realloc = (size() == capacity()) ]
        ( realloc || (orig_data == data()) )

    STRONG_GUARANTEE
    ALLOCATOR_STABLE;

  constexpr void push_back(T&& x)
    // the new back element is equal to x
    post requires(std::memoizable<T>)
         [ orig_x = std::memoize(x) ]
         ( !empty() && orig == back() )

    // all preexisting elements are unchanged
    post requires(std::is_memoizable<T>)
        [ orig = std::memoize(*this) ]
        ( orig.size() < size() &&
          std::equal(orig.cbegin(),
                     orig.cend(),
                     cbegin(),
                     [](const T& x, const std::memoize_t<T>& m)
                         { return std::memoize_equals(m, x); }) )

    // size() increases by 1 if successful, otherwise is unchanged
    interface {
        auto orig_sz = size();
        try {
            implementation;
            contract_assert( orig_sz + 1 == size() );
        }
        catch (...) {
            contract_assert( orig_sz == size() );
        }
    }

    // references are invalidated if and only if reallocation takes place
    post [ orig_data = data(), realloc = (size() == capacity()) ]
        ( realloc || (orig_data == data()) )

    STRONG_GUARANTEE
    ALLOCATOR_STABLE;

  constexpr void pop_back()
    // not empty
    pre( !empty() )

    // size() decreases by one
```

```
post [ orig_size = size() ]
     ( size() = orig_size - 1 )

// elements other than the one destroyed are unchanged
post requires(std::is_memoizable<T>)
    [ orig_v = std::memoize(*this) ]
    ( size() <= orig_v.size() &&
      std::equal(begin(),
                 end(),
                 orig_v.begin(),
                 [](const T& x, const std::memoize_t<T>& m)
                     { return std::memoize_equals(m, x); }) )

ALLOCATOR_STABLE;
```

There are still more functions that need to be described here, but time and paper size are finite.[17] Future iterations of this paper will likely expand this section towards completeness, and likely add new contract assertions to the above functions as new considerations are identified that could be validated.

There are some important takeaways form the above:

- Many of the simplest, and most important, preconditions in the standard library can be represented with precondition assertions as proposed in the Contracts MVP.
- Generic code that does not wish to duplicate functions entirely will benefit greatly from the ability to put requires clauses on contract assertions, which are ubiquitous through many of the preconditions and postconditions used in this section.
- Generic code similarly needs generic labels, as many properties of a contract assertion vary based on template parameters, even moreso when iterators of varying categories are involved.
- Generic code with requires clauses could also be written with procedural interfaces and liberal usage of `if constexpr`.
- Semantic properties of postconditions can often be expressed in terms of a type's public interface, but great care must be taken to order the member functions in a type and avoid cyclic dependencies amongst the functions when evaluating their contract assertions.

5 Further Evolution

In this section are features that we do not yet propose but which might arise in the future to build upon the components described above.

5.1 Core-Language Contract Labels

This subsection will present design considerations for contract labels that might be supported by the core language itself.

[17]This is largely held back because the initial efforts to produce this section were done with the attribute-like syntax for contracts, and converting large quantities of code between the two syntaxes without errors is a laborious process.

5.2 Contract Checks on Coroutines

Here we'll offer design considerations for what contract checks might be applicable to a coroutine and how they might be expressed.

6 Conclusion

The C++20 Contracts facility was a compromise solution that was intended to be the foundation for a full-featured system to enable consistent and powerful tools for defensive programming and static analysis in C++. SG21 has been heavily focused on finding the core set of features that can retain consensus *and* can evolve to support all current and future needs, yet the group has been doing so without a clear understanding of the full scope of those needs.

By adopting the features proposed here or comparable features that meet the same needs, C++ would have a robust, powerful, and user-extensible tool for contract checking that can be integrated into the many different platforms and workflows in which C++ exists today. Incorporating these features into the language and its Standard Library will enable developing, testing, and deploying vastly more robust software than can currently be accomplished and will do so uniformly and portably across the various toolchains that implement modern C++.

Acknowledgements

Much of the design and implementation of build environments, knobs, contract labels, and prechecking was done by Andrew Sutton and Lock3 Software. Andrew provided valuable insight into these features, and papers were produced but not (yet) published in hopes that SG21 would make more progress focusing on completing the MVP for C++23.

Lisa Lippincott has been actively involved in attempting to make C++ a provably correct language (since [P0465R0] at least) and has provided valuable insight into what contract checking might accomplish as well as feedback on these ideas.

Lori Hughes made great contributions to the linguistic quality of this paper, as always. Any lingering speling mitsakes, grammarish problems, unfinished,[18] or formatting issues are completely the fault of the authors.

Many others have contributed to the design in this paper through years of discussion, reviews of earlier drafts of these proposals, and ongoing interest in standardizing a robust contract facility in C++. Listing their names here would inevitably fail to list all who contributed meaningfully.

[18]sentences or paragraphs

A Design Alternatives

Some alternatives to the proposed designs might be considered if they evolve to be superior or if they seem to be more likely to achieve greater consensus..

A.1 CCA Control Objects

The proposal here for labels maps each label ID to a type whose properties are used to identify what affect the label has. The syntax for placing labels on a CCA is an extension of the C++20 Contracts syntax for labels, which used the same space with a small set of Standard-specified identifiers with special meaning. All mechanisms for combining multiple labels to determine the CCA properties are described in the language and vary based on which particular property is being considered.

An alternate possibility is for each CCA to provide a place where a single, optional *control object* maybe be specified with an arbitrary expression that is evaluated at compile time:

```
#include <contracts>

void f()
  [[ pre<audit> : true ]];
```

The `<contracts>` header provides two important elements to make the above example work:

1. The `constexpr` variable `std::contracts::labels::audit`, which is referenced in the control object expression `audit` in the precondition on `f`.
2. A `using contract namespace std::contracts::labels` declaration, which adds the Standard Library contract label namespace to the namespaces implicitly searched by CCA control-object expressions.

The labels provided by the Standard Library would also provide a consistent interface for combining them when appropriate, such as by using the bitwise or (`|`) operator:

```
#include <contracts>  // for audit and review labels
void g()
  [[ pre<audit | review> : true ]];
```

The `<contracts>` header would also provide utilities and base classes that simplify writing label objects such as `audit` and `review` that interoperate smoothly with other labels.

The type and value of the control object expression are used for the same purposes as label types. The expression's type and value would determine the same properties as labels, but combining labels together would be the purview of the label types themselves and their overloaded operators.

- Semantics would be determined by invoking member functions on the resulting `constexpr` object produced by the control object expression.
- If present, a local violation handler will be invoked on that same object.
- The `|` operator provided by the `<contracts>` header will appropriately combine the above operations into a single compound operation on the object it returns.

Due to specification of the control object being a regular C++ expression, the `<contracts>` header would also necessitate the use of a nonkeyword name for labels like `new`, hence our use of `review` as a Standard Library label above.

The primary difference, from a user perspective, between these choices will be where labels are placed within the CCA syntax and how multiple labels will be combined. Advanced users implementing labels will have more freedom to control how labels are combined but greater responsibility for properly interoperation with (rather than subverting the expected behavior of) Standard Library labels, such as `review`, or concrete semantic labels.

Bibliography

[liskov94] Barbara Liskov and Jeannette M. Wing, "A Behavioral Notion of Subtyping". *ACM Transactions Programming Language Systems*, 1994, volume 16(6):pp. 1811–1841

[P0465R0] Lisa Lippincott, "Procedural Function Interfaces", 2016
http://wg21.link/P0465R0

[P1774R8] Timur Doumler, "Portable assumptions", 2022
http://wg21.link/P1774R8

[P2053R0] Rostislav Khlebnikov and John Lakos, "Defensive Checks Versus Input Validation", 2020
http://wg21.link/P2053R0

[P2176R0] Andrzej Krzemieński, "A different take on inexpressible conditions", 2020
http://wg21.link/P2176R0

[P2267R0] Inbal Levi, Ben Craig, and Fabio Fracassi, "Library Evolution Policies", 2023
http://wg21.link/P2267R0

[P2698R0] Bjarne Stroustrup, "Unconditional termination is a serious problem", 2022
http://wg21.link/P2698R0

[P2751R1] Joshua Berne, "Evaluation of *Checked* Contract-Checking Annotations", 2023
http://wg21.link/P2751R1

[P2811R7] Joshua Berne, "Contract-Violation Handlers", 2023
http://wg21.link/P2811R7

[P2877R0] Joshua Berne and Tom Honermann, "Contract Build Modes, Semantics, and Implementation Strategies", 2023
http://wg21.link/P2877R0

[P2900R6] Joshua Berne, Timur Doumler, and Andrzej Krzemieński, "Contracts for C++", 2024
http://wg21.link/P2900R6

[P2946R0] Pablo Halpern, "A flexible solution to the problems of `noexcept`", 2023
http://wg21.link/P2946R0

[P2947R0] Andrei Zissu, Ran Regev, Gal Zaban, and Inbal Levi, “Contracts must avoid disclosing sensitive information”, 2023
http://wg21.link/P2947R0

[P2961R2] Timur Doumler and Jens Maurer, “A natural syntax for Contracts”, 2023
http://wg21.link/P2961R2

[P3005R0] John Lakos, Harold Bott, Bill Chapman, Mungo Gill, Mike Giroux, Alisdair Meredith, and Oleg Subbotin, “Memorializing Principled-Design Policies for WG21”, 2024
http://wg21.link/P3005R0

[P3097R0] Timur Doumler, Joshua Berne, and Gašper Ažman, “Contracts for C++: Support for Virtual Functions”