Doc No: N3997
Date: 2014-05-27
Authors: John Lakos (jlakos@bloomberg.net)
Alexei Zakharov (alexeiz@gmail.com)
Alexander Beels (abeels@bloomberg.net)

Centralized Defensive-Programming Support for Narrow Contracts (Revision 5)

Abstract

Reducing defects in software is a central goal of modern software engineering. Providing essentially defect-free library software can, in large part, be accomplished through thorough unit testing, yet even the best library software—if misused—can lead to defective applications. When invoking a function, not every combination of syntactically valid inputs will (or should) necessarily result in defined behavior. Functions for which certain combinations of inputs and (object) state result in *undefined behavior* are said to have *narrow contracts*. Aggressively validating function preconditions at runtime—commonly referred to as *defensive programming*—can lead to more robust applications by (automatically) detecting out-of-contract use of defensive library software early in the software development life cycle. Most classical approaches to such defensive contract validation, however, necessarily result in suboptimal runtime performance; moreover, when misuse is detected, the action taken is invariably determined by the library, not the application.

In this proposal, we describe a centralized facility for supporting generalized runtime validation of function contracts. What makes this overall approach ideally (and uniquely) suited for standardization is that it allows the application to (1) indicate coarsely (at compile time) the extent to which contract validation should be enabled based on how much "defensive" overhead the application (as a whole) can afford, and (2) specify exactly (at runtime) what action is to be taken should a contract violation be detected. Moreover, the flexibility of this supremely general solution to contract validation lends itself to a thorough, yet surprisingly easy-to-use testing strategy, often called *negative testing*, for which a supportive framework is also provided. Finally, this general approach to implementing and validating defensive checks is not just a good idea: It has been successfully used in production software at Bloomberg for over a decade, was presented at the ACCU conference in 2011, and is currently available along with copious usage examples embedded in running library code as part of Bloomberg's open-source distribution of the BDE library at https://github.com/bloomberg/bde.

Contents

1 Changes from N3963

This proposal is a revision to N3963.

In response to comments at Issaquah, a facility has been added to allow installation of a scoped, thread-local contract-violation handler. When a handler is installed using this facility, the handler will be used only by assertions in the thread from which the handler is installed, and only for the duration of the scope in which the handler is active. At the end of that scope, the previously installed handler for that thread will be restored -- i.e., either the previously installed local handler or (if no local handler was previously installed) the global one.

Additionally, an error in the specified behavior of the contract-assert test macros (section 11.2.5) has been corrected.

Finally, the formal wording has been extensively revised to increase clarity.

2 Introduction

Optimizing quality, cost, and time-to-market are all basic tenets of present-day application software development. A library feature that purports to significantly improve any one of these important aspects would make it well worth consideration for standardization; one that has been demonstrated for over a decade to improve all three at once in a real-world production setting fairly demands it.

First and foremost, our goal as library software developers must be to ensure that, to the extent possible, what is produced with our library code behaves as desired and is implemented without defects. We always try to design library software to be easy to understand and use, yet hard to misuse. Ideally, we prefer that—all other things being equal—any misuse be detected at compile time, rather than at runtime. Unfortunately, such designs are not always practical, or even possible. The standard library is rife with examples where misuse cannot be detected at compile time (see section 3).

What we are proposing here is a centralized, application-configurable standard library facility supporting the runtime detection of misuse of functions where such misuse cannot reasonably be detected at compile time.

3 Background: Narrow versus Wide Contracts

Some functions naturally have no preconditions apart from adhering to the general rules of the C++ language. For example, there is no precondition specified in the contract for

```
void std::vector::push_back(const TYPE&);                    // wide
```

that if violated would result in undefined behavior. The same is true in general for copy (and move) constructors and assignment operators. Such functions naturally have what are called *wide* contracts. On the other hand, both

```
const TYPE& std::vector::operator[](size_t index) const;     // narrow
```

and

```
void std::vector::pop_back();                                // narrow
```

would exhibit undefined behavior on an empty vector (which, in general, is simply not possible to detect at compile time). Such functions are said to have *narrow* contracts.

3.1 Is Artificially Widening Narrow Contracts a Good Idea?

Some advocate that widening the defined behavior to cover all combinations of syntactically valid arguments (and state) is somehow beneficial. Consider the standard member function

```
const TYPE& std::vector::at(size_t index) const;             // wide
```

which provides the same behavior as defined for

```
const TYPE& std::vector::operator[](size_t index) const;     // narrow
```

but, instead of being undefined when `index >= size()`, is required to `throw` an `std::out_of_range` exception. There is no combination of input and state information for which the behavior is *undefined*, and therefore could result in arbitrary behavior. Hence, the contract for the `at` method of an `std::vector` is considered (artificially) *wide* (we say "artificially" because we would never intentionally exploit the added behavior, and misuse can be detected more effectively without it; see below).

There are other ways in which one might widen what would otherwise be a useful narrow contract. For example, consider a value-semantic [1] date class that maintains, as one of its object invariants, a valid date value. Now consider a nominal member function, `set_ymd`, that sets a date object to have the value represented by the specified `year`, `month`, and `day`:

```
class date {
    // ...
  public:
    // ...
    void set_ymd(int year, int month, int day);             // narrow
        // Set the value of this object to the specified
        // 'year', 'month', and 'day'.  The behavior is
        // undefined unless 'year', 'month', and 'day'
        // together represent a valid/supported date value.
```

One way to widen this contract would again be to always validate the inputs and throw an exception if they do not represent a supported date value (incurring a runtime cost in every build mode). Another possibility would be to validate the inputs and then promise to silently do nothing if invalid (most likely masking a defect). A third possibility is to validate the inputs and again do nothing on invalid input, but return status either way (thus precluding *automatic* detection of bad date values in *any* build mode). Narrow contracts, on the other hand, do not suffer from any of these problems.

3.1.1 Artificially Widening Contracts is Misguided

We assert (pun intended) that artificially widening an otherwise useful narrow contract—just to eliminate any undefined behavior—is profoundly misguided for several reasons:

- Even if we do nothing else, validating input has costs.
- Widening forces us to define, document, and test questionably useful code.
- More code runs slower!
- Wide contracts make backward compatible extension *much* harder.
- Artificially wide contracts preclude defensive programming.

For example, consider the standard C function

```
size_t strlen(const char *string);
    // Return the number of characters in the specified (null-terminated)
```

```
    // 'string'.
```

What should the behavior be if `string` is 0? One possibility is that it be defined to return 0:

```
size_t strlen(const char *string)
{
    if (!string) {                                          // wide !!!
        return 0;
    }

    // Determine and return the length of 'string'.
}
```

Doing so, however, would necessarily have a non-zero added cost for everyone, including those who would never invoke the function on a null pointer. What's more, this kind of widening would serve to hide defects. We might instead consider returning `static_cast<size_t>(-1)` as a form of status, but that brings with it its own issues (see below). (Note that simply omitting the check would, in this case, most likely result in program termination, exposing the bug.) Finally, by artificially extending the defined behavior to cover null input, we necessarily eliminate the possibility of the library automatically warning that something is wrong in an application-customizable manner. The optimal solution is to leave the behavior for null strings undefined and, in some (but not all) build modes, detect and report misuse as directed.

Some will argue that correctness is more important than performance, and that always checking function preconditions is a small price to pay. But what if it's not? As a second example, let's revisit our `set_ymd` function discussed above. If we widen the contract to return status (or throw an exception, or even do nothing) on a bad date value, we will then always have to check the date value—even when we know that it is valid:

```
class date {
    short d_year;
    char  d_month;
    char  d_day;
    // ...
  public:
    // ...
    int set_ymd_if_valid(int year, int month, int day);    // wide
        // Set the value of this object to the specified
        // 'year', 'month', and 'day', if 'year', 'month',
        // and 'day' represent a valid/supported date value.
        // Return 0 on success, and a non-zero value if
        // 'year', 'month', and 'day' fail to represent a
        // valid/supported date value.
```

In the case of a date object that stores its year, month, and day value in three separate fields, the cost of validation overwhelms the cost of setting the date value:

```
inline
int date::set_ymd_if_valid(int year, int month, int day)   // wide
{
    if (!isvalid_ymd(year, month, day)) {  // relatively very expensive
```

```
        return -1;  // error: bad input
    }

    d_year  = year;
    d_month = month;
    d_day   = month;

    return 0;  // success
}
```

Precisely the same situation applies to throwing from the `date` value constructor:

```
inline
date::date(int year, int month, int day)                    // wide
: d_year(year), d_month(month), d_day(day)
{
    if (!isvalid_ymd(year, month, day)) {  // relatively very expensive
        throw std::bad_input;
    }
}
```

We know from profiling that such redundant checks can increase runtime by several hundred percent [2].

For anyone who cares about performance, always checking the validity of input values that the caller supplies is a non-starter because, in many circumstances, the caller will already know that their input is valid (if not, they are obligated to check it). In fact, in some cases (such as binary search on a sorted array) the cost of validating a precondition (that the array is in fact sorted) could be of a higher order complexity (O[n]) than that of the work done by the function (O[log(n)]). Hard coding the amount of validation into individual function contracts, and thereby widening them, is simply not the answer. What is needed is a way of allowing each application to coarsely indicate the overall runtime overhead it is prepared to dedicate to (redundant) precondition checking throughout the program.

3.2 Summary of Why Artificially Wide Contracts are Bad

This section provides a concise summary of how an appropriately narrow contract is superior to a corresponding artificially wide one:

- RUNTIME COST: Validating and/or otherwise analyzing input—even if we do nothing else—always has a runtime cost: Sometimes that cost is relatively small, sometimes it is not, and sometimes the cost completely overwhelms that of accomplishing the useful work the function is intended to perform.
- DEVELOPMENT COST: Artificially defining additional behaviors (i.e., beyond input validation) requires more up-front effort by library developers to design, document, implement, and test; the more significant cost, however, is born by application developers when these added behaviors serve only to mask defects resulting from library misuse.

- CODE SIZE: Implementing the additional behavior will necessarily result in larger executables. On all real-world computers, more code generally runs slower—even when that code is never executed!
- EXTENSIBILITY: Artificially defining behavior that is not known to be useful severely impedes adding backward-compatible extensions should new and truly useful functionality be discovered in the future.
- DEFENSIVE PROGRAMMING: Eliminating all undefined behavior precludes robust library implementations from detecting and reporting (in an appropriate build mode) out-of-contract use. If the local function contract always specifies the behavior for all possible input/state combinations, we lose the substantial benefit of this very important, extremely useful quality-of-implementation feature of robust library software for application development.

4 Motivation

Detecting defects early is widely held to be a goal of any good software development process. The benefits of so doing affects each of the various metrics—*quality*, *cost*, and *schedule*—for both library and application software. The sooner we detect a problem, the sooner and more economically we can repair it, leading to a higher quality product.

Unit testing is an effective way of ensuring that library software works as advertised when used properly. Functionality invoked out of contract, however, may accidentally produce the desired result, making such defects—including those within library software itself—resistant to detection by unit testing alone. Absent contract validation by lower-level library functions, the only effective way to detect such misuse is through detailed code reviews. Such reviews are not only expensive, they are subject to human error, and—to be fully effective—need to be repeated whenever an implementation is modified.

Armed with the considerable resources needed to do comprehensive testing and thorough peer review, it is possible to achieve exemplary quality without contract validation. In fact, our implementation experience over the past decade shows that enabling contract validation after library software has been thoroughly reviewed and tested rarely uncovers new defects within the library software itself. On the other hand, the time and effort to debug new library software is dramatically reduced when such contract validation is enabled during development—particularly the initial application of unit tests. Hence, even library software developers can benefit from such defensive contract validation.

When it comes to application software, the benefits of contract validation are unmistakable. Whereas the cost of developing infrastructure libraries can be amortized over many versions of many separate applications, such is seldom the case for the applications themselves, and unit testing—where it exists at all—is notoriously underfunded in many application development environments. Although contract validation is not a substitute for thorough testing, having a library that validates the preconditions of its narrow function contracts can—just by itself—go a

very long way towards improving quality, reducing development costs, and shortening time-to-market for application software that takes advantage of it.

In addition to the development benefits discussed above, if the library's defensive programming infrastructure can be configured to perform a specific action when it detects misuse, then it can be employed even beyond the development phase.

Consider a word processing program, such as the one used to write this proposal. When the program is in beta testing, we expect that there will be some defects. Nevertheless, we want some customers to use the program for real work as part of the beta. If the library infrastructure were to detect misuse and then unconditionally abort the program, it would be unacceptable to the customer, who might wind up losing hours of valuable work. On the other hand, if the application were to disable the defensive programming infrastructure and ignore misuse, it might still crash unexpectedly, or—even worse—corrupt the customer's document.

It is therefore imperative that the application be able to configure the library infrastructure to warn when it detects misuse (or possibly even an internal error) *without* necessarily terminating the program, so that the application can at least have the opportunity to save the customer's data before exiting.

4.1 So what's the problem?

Every application is unique, and every application developer has their own viewpoint. If you ask five application developers how much runtime overhead library software should incur checking for misuse by its client applications, it is quite possible you will get five different answers:

- None
- Negligible (e.g., < 5%)
- Not substantial (e.g., 10-20%)
- A constant factor (e.g., 50-300%)
- Bring it on! (e.g., an order of magnitude)

In fact, these answers will vary—depending on the maturity of the application software at issue. During the early stages of development, it may be that a fairly high degree of checking is both needed and affordable. Once the application is released to production, however, all that extra overhead may no longer be acceptable. For some high-performance applications, even relatively modest overhead may be unacceptable. In the most extreme case, the application owner may decide to allocate zero runtime overhead for contract validation. Our goal is that the same infrastructure library be able to support all these different application needs throughout all phases of their life cycles.

Even if we were able to get application developers to agree on the level of runtime contract validation, they would surely disagree on what should happen if a violation is detected. Some would argue that the program should terminate, since it is known to be broken and letting it continue is only asking for trouble. Others would say that

a function should always throw an exception so that the application has a chance of catching it and cleaning up before exiting. Still others might want the program to go into a busy loop, waiting for an operator to attach a debugger and then proceed on. The possibilities are endless. What should a general-purpose library do?

Standard library components must accommodate a diverse set of needs. We can absolutely guarantee that the library will *not* be reused to its full potential if we hard code either (1) the amount of runtime overhead that a reusable library expends trying to detect contract violations, or (2) what happens if a violation is detected. What is needed is a centralized facility that allows library (and even application) developers to conveniently instrument their software such that application owners are able to specify coarsely (at compile time) the relative amount of overall contract validation that is to occur within the program, and also to specify (at runtime) exactly what is to happen should a violation be detected.

4.2 High-Level Requirements

This section summarizes the essential high-level requirements of any centralized facility (especially one suitable for standardization) to be used for implementing application-configurable defensive checks in library software.

Library developers must be able to

- Easily implement defensive checks to be active in an appropriate defensive build mode.
- Easily test that defensive checks are working as intended.

Each individual application owner (i.e., of `main`) must separately be able to

- Coarsely specify (at compile time) the overall runtime validation overhead.
- Specify precisely (at runtime) the action to take if an error is detected.
- Link translation units compiled with different levels of runtime validation.

Additionally, we advocate that there should be some bilateral recommendation provided along with this centralized facility indicating how library and application developers are encouraged to apportion and assess, respectively, the runtime checking costs associated with each individual defensive build mode. The coarse categories suggested in section 3.1 provide a practical guideline consistent with our experience, which also happens to be closely tied to our heuristic, yet sound, practice for choosing whether or not to declare a function `inline`.

Libraries that employ a centralized, application-configurable strategy for detecting and handling out-of-contract function invocations, as discussed here, have already demonstrated enormous practical benefit by simultaneously improving *quality*, *cost*, and *schedule* metrics for application (and even library) developers that use them. What remains now is to specify a particular implementation of this strategy suitable for standardization.

5 Scope

This facility is intended for ubiquitous use across all library and application software. Every programmer—from novice to expert—is encouraged to understand and document the valid range of inputs (and state) for each function, and codify that information in a way that allows the application owner (as opposed to the immediate caller) to opine on what should happen if a contract violation is detected. Of all the headers in our BDE library [5], the one that defines this functionality, `bsls_assert.h`, is empirically among the most widely included.

6 Existing Practice

Defensive Programming, in its various guises, is a widely used software technique, spanning virtually all computer languages. Many C++ developers still use `<cassert>` to validate *contract conditions* (i.e. pre-, post-, and invariant-conditions), knowing that the runtime overhead can be eliminated in optimized builds. Others, afraid of aborting, hard code contract validation and then always throw an exception when contract violations are detected. Neither of these approaches is ideal, failing to address the flexibility for general purpose, reusable library software.

For more than a decade, Bloomberg's library infrastructure has employed the defensive programming strategy advocated here with excellent success across a wide range of applications and libraries. Copious examples of this strategy's application along with the components providing defensive-programming support are freely available for public scrutiny [5].

7 Impact on the Standard

What we propose requires no new language features. By its very nature, the addition of the centralized checking facility proposed here would have absolutely no direct required effect on any other components within the standard library; however, implementers of standard components would almost certainly want to take advantage of this facility to provide defensive checks to warn against client misuse.

In order for defensive programing to allow for maximum flexibility, we will want to avoid artificially defining behavior for standard functions. In particular, we will want to avoid the use of `noexcept` on narrow contracts, not only to facilitate negative testing [3], but also to allow application programs the opportunity to recover from their own errors and preserve valuable client data. After consideration in Madrid, the committee agreed with strong consensus on criteria [4] for all functions in the C++11 standard that preclude the use of `noexcept` on functions having narrow contracts (where it might impede defensive programming). We presume that all future standard functions will follow suit.

8 Design Decisions

Our proposed design for standardization addresses all of the high-level requirements identified in section 3.2. We have made every effort to adapt all of our implementation

experience to a facility suitable for standardization, consistent with standard naming conventions. There is, however, one departure that we feel deserves mention.

In our environment at Bloomberg, we have full control over the precise nature of how C++ code is rendered (i.e., in terms of `.h`/`.cpp` pairs) and therefore are able to provide some additional diagnostics via our negative testing facility beyond what we have proposed for standardization. In particular, given our logically and physically cohesive naming conventions, our `bsls_asserttest` component is able to determine automatically, during unit testing, whether a function under test was itself able to detect misuse rather than accidently relying on contract validation in a (physically) separate component (`.h`/`.cpp` pair) upon which it depends. In order to accommodate a non-restricted physical rendering style, we have chosen to remove this diagnostic from what is being proposed for standardized negative-testing support.

9 Summary of Proposal for Standardization

The defensive programming support facility that we are proposing for consideration for standardization consists of four parts:

- a set of defensive build modes that control how much additional runtime should be expended on contract validation
- a set of contract-assertion macros that test the validity of predicates
- a violation-handling mechanism that controls what is done when a contract violation is detected
- a set of test macros that can be used in test drivers to verify that contracts are being validated properly

9.1 Defensive Build Modes and Assert Macros

This contract-validation facility is based on the principle that the *application developer* should have at least coarse control (at compile time) over how much runtime resource is to be expended on contract validation in a program. This principle is embodied in three defensive build modes that control which contract-validation tests are run and which are skipped:

- Safe-Build mode is used when an application developer is willing to expend considerable resources on contract validation, perhaps slowing down the program by a constant factor (e.g., 50-300%). In Safe-Build mode, *all* contract-validation tests are enabled.
- Debug- (non-optimized) Build mode is used when an application developer is willing to expend some resources on contract validation, but is not willing to slow down the program appreciably (e.g., by more than 10-20%). In Debug-Build mode, more expensive contract-validation tests are skipped.
- Optimized-Build mode is used when an application developer is not willing to expend any appreciable resources on contract validation. In Optimized-Build

mode, only the most inexpensive (e.g., overhead < 5%) and critical contract-validation tests are performed.

A matching assert macro is provided for each defensive build mode, allowing the *library developer* to express how expensive it is to test each contract condition. Extremely expensive tests would be performed using the Safe-Mode assert macro, moderately expensive tests would be performed using the (non-optimized) Debug-Mode assert macro, and only very inexpensive and/or critical tests would be performed using the Optimized-Mode assert macro.

Once the library developer has implemented contract validation using the appropriate assert macros, the application developer can control the amount of runtime resources expended on checking by choosing the appropriate defensive build mode in which to compile the translation unit. Note that translation units compiled with different assertion levels may be linked together resulting in (typically benign) violations of the ODR.

9.2 Violation Handling

Another principle of the assertion facility is that the *application developer* should have full and complete control (at run time) over what happens when a contract violation is detected. A configurable violation-handler mechanism is provided so that the application owner (i.e., the owner of `main`) can choose to abort the program, throw an exception, or otherwise respond to the violation as they see fit.

9.3 Test Macros

A contract-checking facility is not fully useful unless the checks it supports can be tested. A set of test macros is provided to allow library developers to easily test that (a) no violation is detected when all contract conditions are met, (b) a violation is detected when any contract condition is not met, and (c) each contract condition is active in only the appropriate defensive build modes. Note that our implementation experience shows that test actions resulting in *in-contract* calls should always be honored, whereas *out-of-contract* calls should occur only when in a defensive build mode enabling a precondition check that can respond to the particular contract violation.

10 Examples

10.1 Assert a contract precondition in Debug-Build mode

In this example, a `strlen`-like function, `other_strlen`, enforces a narrow contract. The function contract has a precondition that `str` must not be null, and the function uses `CONTRACT_ASSERT` to check the precondition in Debug- and Safe-Build modes.

```
#include <experimental/contract_assert>

std::size_t other_strlen(const char *str)
{
    CONTRACT_ASSERT(str);
```

```
    // ... return string length
}
```

10.2 Assert a contract precondition in Safe-Build mode

In this example, a simple inline array-lookup function, `get_int_array_element`, enforces a narrow contract that is relatively expensive to check when compared to the cost of performing the look-up. The function contract has a precondition that `index` must be greater than or equal to 0 and less than `length`. Because checking this precondition is relatively expensive, we use `CONTRACT_ASSERT_SAFE`, so that we will pay the runtime cost only in Safe-Build mode.

```
#include <experimental/contract_assert>

int get_int_array_element(int *array, int length, int index)
{
    CONTRACT_ASSERT_SAFE(0 <= index         );
    CONTRACT_ASSERT_SAFE(     index < length);

    return array[index];
}
```

10.3 Throw an exception on a contract violation

In this example, a program defines and installs a contract-violation-handler function, `handle_violation`, that throws an exception of type `contact_violation_error`. When the program later calls a function, `some_function`, that unconditionally asserts a contract violation in Debug- or Safe-Build mode, the program can print a diagnostic message before exciting normally.

```
#include <experimental/contract_assert>

struct contract_violation_error
{
    contract_violation_error(
                      const std::experimental::contract_violation_info& info);

    std::experimental::contract_violation_info info;
};

contract_violation_error::contract_violation_error(
                       const std::experimental::contract_violation_info& info)
: info(info)
{
}

void throw_on_contract_violation(
                       const std::experimental::contract_violation_info& info)
{
    throw contract_violation_error(info);
}

int some_function()
{
```

```
    CONTRACT_ASSERT(false);  // throws contract_violation_error
}

int main()
{
    std::experimental::set_handle_contract_violation(
                                                    throw_on_contract_violation);

    try
    {
        some_function();
    }
    catch (contract_violation_error)
    {
        std::cout << "A contract violation was detected." << std::endl;
    }
    catch (...)
    {
        std::cout << "An unexpected exception was caught." << std::endl;
    }

    return 0;
}
```

Note that `contract_violation_error` does not inherit from `std::exception`. An exception thrown by a contract-violation handler should not be related to any exceptions used elsewhere in the program so as to minimize the possibility that the program's usual exception-handling facilities will intercept and hide the exception reporting the contract violation.

Also note that the contract-violation handler is installed using `set_handle_contract_violation`; hence, this handler will be the default handler for all threads.

10.4 Install a local contract-violation handler for negative testing

The same contract-violation handler used in Example 2 could be used to incorporate simple negative testing of `other_strlen` into a larger test driver by using a `handle_contract_violation_guard` instead of `set_handle_contract_violation`.

```
int run_negative_tests()
{
    std::experimental::handle_contract_violation_guard
                                              guard(throw_on_contract_violation);

    try
    {
        other_strlen(nullptr);
    }
    catch (contract_violation_error)
    {
       return 0;
    }

    return 1;
```

```
}

int main()
{
    int num_test_failures = 0;

    // Possibly spawn other threads.

    // ...

    num_test_failures += run_negative_tests();

    // Call other functions.

    // ...

    if (num_test_failures > 0)
    {
        std::cout << num_test_failures << " tests failed." << std::endl;
    }
    else
    {
        std::cout << "All tests passed." << std::endl;
    }

    return num_test_failures;
}
```

Note that any other threads spawned by `main` will default to using the global contract-violation handler, even after `run_negative_tests` installs its local handler; `run_negative_tests` will not be affected by any other local handlers installed in other threads. Similarly, other functions called in the main thread will also not be affected by the local handler installed in `run_negative_tests` since that local handler will be uninstalled when `guard` goes out of scope.

10.5 Test that a function correctly asserts its preconditions

In this example, a program makes both in-contract and out-of-contract calls to `other_strlen`, and uses the `TEST_CONTRACT_ASSERT_*` macros to validate that `other_strlen` checks its precondition in the appropriate defensive build modes.

```
#include <experimental/contract_assert_test>

int main()
{
    if (TEST_CONTRACT_ASSERT_PASS(other_strlen("a string")))
    {
        std::cout << "Correctly detects no contract violation.\n";
    }
    else
    {
        std::cout << "Incorrectly detects contract violation.\n";
    }

    if (TEST_CONTRACT_ASSERT_FAIL(other_strlen(nullptr)))
```

```
    {
        std::cout << "Successfully detects contract violation.\n";
    }
    else
    {
        std::cout << "Fails to detect contract violation.\n";
    }

    return 0;
}
```

11 Formal Wording

11.1 Definitions

Add three new definitions to clause 17.3:

17.3.X [**defns.contract**]

contract

A contract is the behavioral specification, including parameters, requirements, and observable behavior, for a function, macro, or template.

17.3.Y [**defns.contract.narrow**]

narrow contract

A narrow contract is a contract that, for some subset of its possible inputs, documents undefined behavior.

17.3.Z [**defns.contract.wide**]

wide contract

A wide contract is a contract that, for all possible inputs permitted by the language, documents no undefined behavior.

11.2 Defensive programming [**contract**]

11.2.1 In general [**contract.general**]

The header `<experimental/contract_assert>` defines macros, functions, and types that support defensive runtime validation of function contracts. A *contract assertion* is a conditionally evaluated test of an expression, using one of the *contract-assert macros* defined in [contract.assertions], that is intended to express a requirement of a function contract. When a contract assertion is evaluated and fails, a *contract violation* is detected. The conditions under which contract assertions are evaluated for a given translation of a program are controlled by *defensive build modes*. A program that utilizes the defensive programming facility relies on a global

contract-violation-handler function to be called when a contract violation is detected. Additionally, each thread in such a program may install its own local handler, which will be used instead of the global handler when a contract violation is detected in that thread.

The header `<experimental/contract_assert_test>` defines *contract-assert test macros* that provide a way to check that a contract violation is detected, due to appropriate use of contract assertions, if and only if a function is called out of contract. These macros make use of a *contract-assert test context* having a thread-local state indicating either success or failure. When first established, a contract-assert test context has a *success* state.

The following subclauses describe the assert macros, assert macro state flags, assertion handlers, and assert test macros comprised by this contract-assertion facility.

Header <experimental/contract_assert> synopsis

```
// contract-assert macros
#define CONTRACT_ASSERT_OPT(condition_expression)  // see below
#define CONTRACT_ASSERT_DBG(condition_expression)  // see below
#define CONTRACT_ASSERT_SAFE(condition_expression) // see below

#define CONTRACT_ASSERT(condition_expression)  \
            CONTRACT_ASSERT_DBG(condition_expression)

#define CONTRACT_ASSERT_OPT_IS_ACTIVE   // conditionally defined, see below
#define CONTRACT_ASSERT_DBG_IS_ACTIVE   // conditionally defined, see below
#define CONTRACT_ASSERT_SAFE_IS_ACTIVE  // conditionally defined, see below

#define CONTRACT_ASSERT_IS_ACTIVE       // defined when
                                        // CONTRACT_ASSERT_DBG_IS_ACTIVE
                                        // is defined

namespace std {
namespace experimental {

// types
enum class contract_assert_mode
{
    opt,
    dbg,
    safe
};

struct contract_violation_info;

using handle_contract_violation_handler =
                                         void (*)(const contract_violation_info&);

// handler manipulators
handle_contract_violation_handler
set_handle_contract_violation(handle_contract_violation_handler handler) noexcept;

handle_contract_violation_handler get_handle_contract_violation() noexcept;
```

```
// handler invoker
[[noreturn]] void handle_contract_violation(const contract_violation_info& info);

// thread-local contract-violation handler installation
struct handle_contract_violation_guard;

}  // namespace experimental
}  // namespace std
```

Header <experimental/contract_assert_test> synopsis

```
#include <experimental/contract_assert>

// contract-assert test macros
#define TEST_CONTRACT_ASSERT_OPT_FAIL(test_expression)     // see below
#define TEST_CONTRACT_ASSERT_DBG_FAIL(test_expression)     // see below
#define TEST_CONTRACT_ASSERT_SAFE_FAIL(test_expression)    // see below

#define TEST_CONTRACT_ASSERT_FAIL(test_expression)  \
            TEST_CONTRACT_ASSERT_DBG_FAIL(test_expression)

#define TEST_CONTRACT_ASSERT_OPT_PASS(test_expression)     // see below
#define TEST_CONTRACT_ASSERT_DBG_PASS(test_expression)     // see below
#define TEST_CONTRACT_ASSERT_SAFE_PASS(test_expression)    // see below

#define TEST_CONTRACT_ASSERT_PASS(test_expression)  \
            TEST_CONTRACT_ASSERT_DBG_PASS(test_expression)
```

11.2.2 Defensive build mode selection [**contract.modes**]

At any point during the translation of a program, at most one of the four defensive build modes described in Table 1 is in effect. Initially, no defensive build mode is in effect.

Table 1

Defensive Build Mode	Description
disabled	No contract conditions are validated.
optimized	Only the least expensive contract conditions are validated.
debug	Up to moderately expensive contract conditions are validated.
safe	All contract conditions are validated.

Each successive defensive build mode listed in Table 1 validates no fewer contract conditions than the defensive build modes that precede it, and can be said to be stronger (no weaker) than the preceding defensive build modes.

Each defensive build mode has an associated preprocessor token, referred to as a defensive build-mode-selection token, defined in Table 2.

Table 2

Defensive Build Mode	Defensive Build-Mode-Selection Token
disabled	`CONTRACT_ASSERT_LEVEL_NONE`
optimized	`CONTRACT_ASSERT_LEVEL_ASSERT_OPT`
debug	`CONTRACT_ASSERT_LEVEL_ASSERT_DBG`
safe	`CONTRACT_ASSERT_LEVEL_ASSERT_SAFE`

The program is ill-formed (diagnostic required) if `<contract_assert>` is included and multiple build-mode-selection tokens are defined. Each time `<contract_assert>` is included, if one of these tokens is defined, the corresponding build mode comes into effect, replacing any build mode that was under effect up to that point. If none of the build-mode-selection tokens are defined when `<contract_assert>` is included, the 'debug' build mode comes into effect.

11.2.3 Contract-assert macros [**contract.assertions**]

There are three contract-assert macros:

```
#define CONTRACT_ASSERT_OPT(condition_expression)
#define CONTRACT_ASSERT_DBG(condition_expression)
#define CONTRACT_ASSERT_SAFE(condition_expression)
```

In addition, there is one alias macro:

```
#define CONTRACT_ASSERT(condition_expression)  \
            CONTRACT_ASSERT_DBG(condition_expression)
```

The alias macro provides a convenient abbreviation for `CONTRACT_ASSERT_DBG`.

Each contract-assert macro corresponds to one defensive build mode, and has an associated *mode value* of the `enum` type `contract_assert_mode`, as defined in Table 3. A contract-assert macro is active only if the defensive build mode corresponding to the macro (or a yet stronger mode) is currently in effect.

Table 3

Contract-Assert Macro	Build Mode	Mode Value
`CONTRACT_ASSERT_OPT(condition_expression)`	optimized	`opt`
`CONTRACT_ASSERT_DBG(condition_expression)`	debug	`dbg`
`CONTRACT_ASSERT_SAFE(condition_expression)`	safe	`safe`

Note that no macro corresponds to 'disabled', which is the weakest defensive build mode. When the 'disabled' build mode is in effect, no contract-assert macros are active.

The `condition_expression` of a macro is not evaluated unless the macro is active in the defensive build mode currently in effect.

A `condition_expression` supplied to an active macro shall be contextually convertible to `bool`. A contract violation is detected if such an expression is evaluated to `false` when contextually converted to `bool`.

When a contract violation is detected due to evaluation of a `condition_expression` in a contract-assert macro, the implementation will call `std::experimental::handle_contract_violation` with a `contract_violation_info` structure, each member of which will be initialized to the values described in Table 4.

Table 4

Data Member	**Value**
`mode`	initialized to the mode value corresponding to the macro that triggered the contract violation.
`expression_text`	initialized to the stringification of the `condition_expression` that triggered the contract violation.
`filename`	initialized to the value of `__FILE__` at the location of the macro that triggered the contract violation.
`line_number`	initialized to the value of `__LINE__` at the location of the macro that triggered the contract violation.

11.2.4 Contract-assert-macro state flags [**contract.flags**]

There are three contract-assert-macro state flags, zero or more of which will be defined depending on the defensive build mode currently in effect:

```
#define CONTRACT_ASSERT_OPT_IS_ACTIVE
#define CONTRACT_ASSERT_DBG_IS_ACTIVE
#define CONTRACT_ASSERT_SAFE_IS_ACTIVE
```

In addition, there is one alias macro:

```
#define CONTRACT_ASSERT_IS_ACTIVE       // defined when
                                        // CONTRACT_ASSERT_DBG_IS_ACTIVE
                                        // is defined
```

The alias macro provides a convenient abbreviation for `CONTRACT_ASSERT_DBG_IS_ACTIVE`.

Each contract-assert macro state flag corresponds to a defensive build mode, as defined in Table 5, and is not defined unless the defensive build mode currently in

effect is the macro's corresponding defensive build mode or a stronger defensive build mode.

Table 5

Contract-Assert-Macro State Flag	Defensive Build Mode
`CONTRACT_ASSERT_OPT_IS_ACTIVE`	optimized
`CONTRACT_ASSERT_DBG_IS_ACTIVE`	debug
`CONTRACT_ASSERT_SAFE_IS_ACTIVE`	safe

11.2.5 Contract-assert test macros [**contract.test.macros**]

There are two kinds of contract-assert test macros: *contract-assert test-fail macros*, which validate that out-of-contract calls detect a contract violation, and *contract-assert test-pass macros*, which validate that in-contract calls do not detect a contract violation.

There are three contract-assert test-fail macros:

```
#define TEST_CONTRACT_ASSERT_OPT_FAIL(test_expression)
#define TEST_CONTRACT_ASSERT_DBG_FAIL(test_expression)
#define TEST_CONTRACT_ASSERT_SAFE_FAIL(test_expression)
```

In addition, there is one alias macro:

```
#define TEST_CONTRACT_ASSERT_FAIL(test_expression)  \
            TEST_CONTRACT_ASSERT_DBG_FAIL(test_expression)
```

The alias macro provides a convenient abbreviation for `TEST_CONTRACT_ASSERT_DBG_FAIL`.

Each contract-assert test-fail macro corresponds to a defensive build mode and mode value, as defined in Table 6. A contract-assert test-fail macro is active only in the macro's corresponding defensive build mode, or a stronger one.

Table 6

Contract-Assert Test-Fail macro	Defensive Build Mode	Mode Value
`TEST_CONTRACT_ASSERT_OPT_FAIL(test_expression)`	optimized	`opt`
`TEST_CONTRACT_ASSERT_DBG_FAIL(test_expression)`	debug	`dbg`
`TEST_CONTRACT_ASSERT_SAFE_FAIL(test_expression)`	safe	`safe`

The `test_expression` is not evaluated unless the macro is active in the defensive build mode currently in effect. If, after evaluating the `test_expression`, the contract-assert test context is in a failure state, and the mode value passed to `std::experimental::handle_contract_violation` matches the mode value for the

contract-assert test-fail macro, the macro expands to an expression that evaluates to `true`; otherwise it expands to one that evaluates to `false`.

There are three matching contract-assert test-pass macros, and one alias, providing in-contract tests paired with the out-of-contract tests performed by the contract-assert test-fail macros:

```
#define TEST_CONTRACT_ASSERT_OPT_PASS(test_expression)
#define TEST_CONTRACT_ASSERT_DBG_PASS(test_expression)
#define TEST_CONTRACT_ASSERT_SAFE_PASS(test_expression)

#define TEST_CONTRACT_ASSERT_PASS(test_expression)  \
            TEST_CONTRACT_ASSERT_DBG_PASS(test_expression)
```

Each contract-assert test-pass macro evaluates the `test_expression` within a contract-assert test context. If, after evaluating the `test_expression`, the contract-assert test context is in a failure state, the macro expands to an expression that evaluates to `false`; otherwise it expands to one that evaluates to `true`.

11.2.6 Contract-assert-handler functions [**contract.handler**]

handle_contract_violation_handler

```
using handle_contract_violation_handler =
                                        void (*)(const contract_violation_info&);
```

The type of a *contract-violation-handler function* to be called when a contract violation is detected.

Required behavior: A `handle_contract_violation_handler` shall not return normally to the caller. [*Note:* A handler may throw an exception. — *end note*]

Default behavior: The implementation's default global `handle_contract_violation_handler` calls `std::abort()`.

handle_contract_violation

```
handle_contract_violation_handler
set_handle_contract_violation(handle_contract_violation_handler handler) noexcept;
```

Effects: Establishes the function designated by `handler` as the current global contract-violation handler.

Remarks: A null pointer value designates the default contract-violation-handler function. The behavior is undefined unless every evaluation of a call to `set_handle_contract_violation` inter-thread happens before [intro.multithread] any evaluation of a call to `handle_contract_violation`.

Returns: The previous global contract-violation-handler function.

```
handle_contract_violation_handler get_handle_contract_violation() noexcept;
```

Returns: The (address of the) current global contract-violation-handler function. [*Note:* This address value can be null (indicating the default handle). – *end note*]

```
[[noreturn]] void handle_contract_violation(const contract_violation_info& info);
```

Remarks: Called by the implementation when any of the contract-assert macros fail. [*Note:* The `handle_contract_violation` function may also be called directly by a program. – *end note*]

Effects: Call the currently installed local contract-violation handler, if any. If there is no installed local contract-violation handler, call the global contract-violation handler instead. [*Note:* A default `handle_contract_violation_handler` is always considered a callable handler in this context. — *end note*]

`struct handle_contract_violation_guard`

An object of type `handle_contract_violation_guard` controls the installation of a local contract-violation handler for the current thread within a scope.

```
explicit
handle_contract_violation_guard(handle_contract_violation_handler handler);
```

Effects: `handler` becomes the local contract-violation handler for the calling thread.

```
~handle_contract_violation_guard();
```

Effects: The local contract-violation handler for the calling thread is restored to the state it had before this object was constructed.

Remarks: Must be called from the same thread that called the guard's constructor.

12 Contract-assert-facility reference implementation

12.1 Overview

The contract-assert-facility reference implementation consists of the following parts:

- Contract-Assert Component:
  1. A mechanism to ensure that only one of the defensive build-mode-selection tokens is defined on entry into the `experimental/contract_assert` header.
  2. Definitions of the contract-assert macros for each defensive build mode.
  3. Definitions of the global functions and types stipulated by the formal wording: `contract_assert_mode`, `contract_violation_info`, `handle_contract_violation_handler`, `set_handle_contract_violation`, `get_handle_contract_violation`, `handle_contract_violation`.
  4. Definitions of two contract-violation handlers: a global contract-violation handler and a thread-specific local contract-violation handler.
  5. Definition of a `handle_contract_violation_guard` type that performs the following tasks:

        - On construction, store the existing local contract-violation handler, and install a user-supplied contract-violation-handler function as the current thread's local contract-violation handler.
        - On destruction, restore the stored contract-violation-handler function as the current thread's local contract-violation handler.
    6. Definition of a default contract-violation-handler function that terminates the program.
- Contract-Assert-Test Component:
    1. Definition of a contract-violation-exception type.
    2. Definition of a contract-violation-handler function that throws the contract-violation exception.
    3. Definition of the function `test_contract_assert_imp`, which performs the following tasks:
        - Create a `handle_contract_violation_guard` that installs the exception-throwing contract-violation-handler function as the current thread's local contract-violation handler for the duration of `test_contract_assert_imp`'s scope.
        - Evaluate the `expression` under test inside a try/catch block.
        - Catch the contract-violation exception and verify that it was indeed expected to be thrown.
        - Uninstall the local contract-violation handler.

12.2 Implementation

The reference implementation below presents an instructive example implementation of `<experimental/contract_assert>` and `<experimental/contract_assert_test>`. In addition, a small sample program is provided to exercise the facilities. It is assumed that when the sample program is built, the `CONTRACT_ASSERT_LEVEL_ASSERT_DBG` macro will be set by the build system.

12.2.1 experimental/contract_assert

```
#include <cstdlib>

// macros

#undef CONTRACT_ASSERT_OPT_IS_ACTIVE
#undef CONTRACT_ASSERT_DBG_IS_ACTIVE
#undef CONTRACT_ASSERT_SAFE_IS_ACTIVE
#undef CONTRACT_ASSERT_IS_ACTIVE

#undef CONTRACT_ASSERT_OPT
#undef CONTRACT_ASSERT_DBG
#undef CONTRACT_ASSERT_SAFE
#undef CONTRACT_ASSERT
```

```
#if !defined(CONTRACT_ASSERT_LEVEL_NONE) &&        \
    !defined(CONTRACT_ASSERT_LEVEL_ASSERT_OPT) &&  \
    !defined(CONTRACT_ASSERT_LEVEL_ASSERT_DBG) &&  \
    !defined(CONTRACT_ASSERT_LEVEL_ASSERT_SAFE)
#define CONTRACT_ASSERT_LEVEL_ASSERT_DBG
#elif 1 !=                                         \
    defined(CONTRACT_ASSERT_LEVEL_NONE) +          \
    defined(CONTRACT_ASSERT_LEVEL_ASSERT_OPT) +    \
    defined(CONTRACT_ASSERT_LEVEL_ASSERT_DBG) +    \
    defined(CONTRACT_ASSERT_LEVEL_ASSERT_SAFE)
#error "Cannot define more than one CONTRACT_ASSERT_LEVEL_* macro at one time"
#endif

#ifdef CONTRACT_ASSERT_LEVEL_ASSERT_OPT

#define CONTRACT_ASSERT_OPT_IS_ACTIVE

#endif  // CONTRACT_ASSERT_LEVEL_ASSERT_OPT

#ifdef CONTRACT_ASSERT_LEVEL_ASSERT_DBG

#define CONTRACT_ASSERT_OPT_IS_ACTIVE
#define CONTRACT_ASSERT_DBG_IS_ACTIVE
#define CONTRACT_ASSERT_IS_ACTIVE

#endif  // CONTRACT_ASSERT_LEVEL_ASSERT_DBG

#ifdef CONTRACT_ASSERT_LEVEL_ASSERT_SAFE

#define CONTRACT_ASSERT_OPT_IS_ACTIVE
#define CONTRACT_ASSERT_DBG_IS_ACTIVE
#define CONTRACT_ASSERT_SAFE_IS_ACTIVE
#define CONTRACT_ASSERT_IS_ACTIVE

#endif  // CONTRACT_ASSERT_LEVEL_ASSERT_SAFE


#ifdef CONTRACT_ASSERT_OPT_IS_ACTIVE

#define CONTRACT_ASSERT_OPT(...)                                    \
    do                                                              \
    {                                                               \
        if (!(__VA_ARGS__))                                         \
        {                                                           \
            std::experimental::handle_contract_violation({          \
                    std::experimental::contract_assert_mode::opt,   \
                    #__VA_ARGS__,                                   \
                    __FILE__,                                       \
                    __LINE__                                        \
            });                                                     \
        }                                                           \
    } while (0)

#else

#define CONTRACT_ASSERT_OPT(...)   do {} while(0)
```

```
#endif // CONTRACT_ASSERT_OPT_IS_ACTIVE


#ifdef CONTRACT_ASSERT_DBG_IS_ACTIVE

#define CONTRACT_ASSERT_DBG(...)                                        \
    do                                                                  \
    {                                                                   \
        if (!(__VA_ARGS__))                                             \
        {                                                               \
            std::experimental::handle_contract_violation({              \
                    std::experimental::contract_assert_mode::dbg,       \
                    #__VA_ARGS__,                                       \
                    __FILE__,                                           \
                    __LINE__                                            \
            });                                                         \
        }                                                               \
    } while (0)

#else

#define CONTRACT_ASSERT_DBG(...)   do {} while(0)

#endif // CONTRACT_ASSERT_DBG_IS_ACTIVE


#ifdef CONTRACT_ASSERT_SAFE_IS_ACTIVE

#define CONTRACT_ASSERT_SAFE(...)                                       \
    do                                                                  \
    {                                                                   \
        if (!(__VA_ARGS__))                                             \
        {                                                               \
            std::experimental::handle_contract_violation({              \
                    std::experimental::contract_assert_mode::safe,      \
                    #__VA_ARGS__,                                       \
                    __FILE__,                                           \
                    __LINE__                                            \
            });                                                         \
        }                                                               \
    } while (0)

#else

#define CONTRACT_ASSERT_SAFE(...)  do {} while(0)

#endif // CONTRACT_ASSERT_SAFE_IS_ACTIVE


#define CONTRACT_ASSERT(...)       CONTRACT_ASSERT_DBG(__VA_ARGS__)


#ifndef INCLUDED_CONTRACT_ASSERT
#define INCLUDED_CONTRACT_ASSERT

namespace std {
```

```
namespace experimental {

// types
enum class contract_assert_mode
{
    opt,
    dbg,
    safe
};

struct contract_violation_info
{
    contract_assert_mode  mode;
    const char           *expression_text;
    const char           *filename;
    size_t                line_number;
};

using handle_contract_violation_handler =
                                        void (*)(const contract_violation_info&);

// handler manipulators
handle_contract_violation_handler
set_handle_contract_violation(handle_contract_violation_handler handler) noexcept;

handle_contract_violation_handler get_handle_contract_violation() noexcept;

// handler invoker
[[noreturn]] void handle_contract_violation(const contract_violation_info& info);

// local precondition violation handler installation
struct handle_contract_violation_guard
{
    explicit
    handle_contract_violation_guard(handle_contract_violation_handler handler);

    ~handle_contract_violation_guard();

    handle_contract_violation_guard(
                            const handle_contract_violation_guard&) = delete;

    handle_contract_violation_guard&
    operator=(const handle_contract_violation_guard&) = delete;

    handle_contract_violation_handler old_handler;
};

}  // namespace experimental
}  // namespace std

#endif  // INCLUDED_CONTRACT_ASSERT
```

12.2.2 experimental/contract_assert.cpp

```
#include <experimental/contract_assert>
#include <atomic>
```

```
namespace std {
namespace experimental {
namespace detail {

void abort_handler(const contract_violation_info&)
{
    std::abort();
}

std::atomic<handle_contract_violation_handler> handler{abort_handler};

thread_local
handle_contract_violation_handler installed_local_handler{nullptr};

}  // namespace detail


handle_contract_violation_handler
set_handle_contract_violation(handle_contract_violation_handler handler) noexcept
{
    if (handler)
    {
        return std::atomic_exchange(&detail::handler, handler);
    }
    else
    {
        return std::atomic_exchange(&detail::handler, detail::abort_handler);
    }
}

handle_contract_violation_handler get_handle_contract_violation() noexcept
{
    return std::atomic_load(&detail::handler);
}

[[noreturn]] void handle_contract_violation(const contract_violation_info& info)
{
    [[noreturn]]
    handle_contract_violation_handler handler = detail::installed_local_handler;
    if (handler)
    {
        handler(info);
    }
    handler = get_handle_contract_violation();
    if (handler)
    {
        handler(info);
    }
    std::abort();
}

handle_contract_violation_guard::handle_contract_violation_guard(
                                        handle_contract_violation_handler handler)
: old_handler(detail::installed_local_handler)
{
    detail::installed_local_handler = handler;
}
```

```
handle_contract_violation_guard::~handle_contract_violation_guard()
{
    detail::installed_local_handler = old_handler;
}

}  // namespace experimental
}  // namespace std
```

12.2.3 experimental/contract_assert_test

```
#ifndef INCLUDED_CONTRACT_ASSERT_TEST
#define INCLUDED_CONTRACT_ASSERT_TEST

#include <experimental/contract_assert>

// macros

#undef TEST_CONTRACT_ASSERT_OPT_PASS
#undef TEST_CONTRACT_ASSERT_DBG_PASS
#undef TEST_CONTRACT_ASSERT_SAFE_PASS
#undef TEST_CONTRACT_ASSERT_PASS

#undef TEST_CONTRACT_ASSERT_OPT_FAIL
#undef TEST_CONTRACT_ASSERT_DBG_FAIL
#undef TEST_CONTRACT_ASSERT_SAFE_FAIL
#undef TEST_CONTRACT_ASSERT_FAIL


#define TEST_CONTRACT_ASSERT_OPT_PASS(...)                          \
    std::experimental::detail::test_contract_assert_imp(           \
            std::experimental::contract_assert_mode::opt, true,    \
            [&]{ (__VA_ARGS__); })

#ifdef CONTRACT_ASSERT_OPT_IS_ACTIVE

#define TEST_CONTRACT_ASSERT_OPT_FAIL(...)                          \
    std::experimental::detail::test_contract_assert_imp(           \
            std::experimental::contract_assert_mode::opt, false,   \
            [&]{ (__VA_ARGS__); })

#else

#define TEST_CONTRACT_ASSERT_OPT_FAIL(...)  (true)

#endif // CONTRACT_ASSERT_OPT_IS_ACTIVE


#define TEST_CONTRACT_ASSERT_DBG_PASS(...)                          \
    std::experimental::detail::test_contract_assert_imp(           \
            std::experimental::contract_assert_mode::dbg, true,    \
            [&]{ (__VA_ARGS__); })

#ifdef CONTRACT_ASSERT_DBG_IS_ACTIVE

#define TEST_CONTRACT_ASSERT_DBG_FAIL(...)                          \
    std::experimental::detail::test_contract_assert_imp(           \
```

```
            std::experimental::contract_assert_mode::dbg, false,  \
            [&]{ (__VA_ARGS__); })

#else

#define TEST_CONTRACT_ASSERT_DBG_FAIL(...)  (true)

#endif // CONTRACT_ASSERT_DBG_IS_ACTIVE


#define TEST_CONTRACT_ASSERT_SAFE_PASS(...)                          \
    std::experimental::detail::test_contract_assert_imp(             \
            std::experimental::contract_assert_mode::safe, true,   \
            [&]{ (__VA_ARGS__); })

#ifdef CONTRACT_ASSERT_SAFE_IS_ACTIVE

#define TEST_CONTRACT_ASSERT_SAFE_FAIL(...)                          \
    std::experimental::detail::test_contract_assert_imp(             \
            std::experimental::contract_assert_mode::safe, false,  \
            [&]{ (__VA_ARGS__); })

#else

#define TEST_CONTRACT_ASSERT_SAFE_FAIL(...)  (true)

#endif // CONTRACT_ASSERT_SAFE_IS_ACTIVE


#define TEST_CONTRACT_ASSERT_PASS(...) TEST_CONTRACT_ASSERT_DBG_PASS(__VA_ARGS__)
#define TEST_CONTRACT_ASSERT_FAIL(...) TEST_CONTRACT_ASSERT_DBG_FAIL(__VA_ARGS__)


namespace std {
namespace experimental {
namespace detail {

struct contract_error
{
    explicit contract_error(contract_assert_mode mode);

    contract_assert_mode mode;
};

[[noreturn]] void exception_handler(const contract_violation_info& info);

template <typename LAMBDA>
bool test_contract_assert_imp(contract_assert_mode mode,
                              bool                 expect_pass,
                              LAMBDA               lambda)
{
    handle_contract_violation_guard guard{exception_handler};

    try
    {
        lambda();
```

```
        // The assert passed; return true if it was expected to pass.

        return expect_pass;
    }
    catch (const contract_error& error)
    {
        // The assert failed; return true if it was not expected to pass,
        // and the mode is correct.

        return error.mode == mode && !expect_pass;
    }
}

}  // namespace detail
}  // namespace experimental
}  // namespace std

#endif  // INCLUDED_CONTRACT_ASSERT
```

12.2.4 experimental/contract_assert_test.cpp

```
#include <experimental/contract_assert_test>

namespace std {
namespace experimental {
namespace detail {

contract_error::contract_error(contract_assert_mode mode)
: mode{mode}
{
}

[[noreturn]] void exception_handler(const contract_violation_info& info)
{
    throw contract_error(info.mode);
}

}  // namespace detail
}  // namespace experimental
}  // namespace std
```

12.2.5 Simple test driver for experimental/contract_assert

```
#include <experimental/contract_assert>
#include <experimental/contract_assert_test>
#include <cstddef>
#include <iostream>

std::size_t other_strlen(const char *str)
{
    CONTRACT_ASSERT(str);

    std::size_t length = 0;
    while (*str) {
        ++length;
        ++str;
    }
```

```
    return length;
}

// simple testing macro for use with a naive test harness

#define MY_ASSERT(expression)                                          \
do { if (!(expression)) {                                              \
    std::cerr << "Testing failure on expression at (" << __LINE__   \
              << "): " << # expression << std::endl;                   \
    std::exit(EXIT_FAILURE);                                           \
} } while (0)

int main()
{
    // Run some test test scenarios.

    MY_ASSERT(TEST_CONTRACT_ASSERT_PASS(other_strlen("a string")));
    MY_ASSERT(TEST_CONTRACT_ASSERT_FAIL(other_strlen(nullptr)));

    // If all things succeed, report success.

    return EXIT_SUCCESS;
}
```

13 References

[1] N2479 - Normative Language to Describe Value Copy Semantics

[2] N3344 - Toward a Standard C++ 'Date' Class

[3] N3248 - `noexcept` Prevents Library Validation

[4] N3279 - Conservative use of `noexcept` in the Library

[5] Bloomberg BDE Library, open-source distribution.