Doc No: N3963
Date: 2014-03-02
Authors: John Lakos (jlakos@bloomberg.net)
Alexei Zakharov

Centralized Defensive-Programming Support for Narrow Contracts (Revision 4)

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 defensive precondition checks, 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 defensive runtime validation of function preconditions. 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 precondition checking 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 precondition violation be detected. Moreover, the flexibility of this supremely general solution to precondition 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 N3877

This proposal is a revision to N3877.

The standardese has been extensively redrafted based on feedback from the LWG at Issaquah.

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 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 possible, or practical. The standard library is rife with examples where misuse cannot be detected at compile time (see section 2).

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 (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's 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'.  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 its value constructor:

```
    inline
    int date::date(int year, int month, int day)               // wide
    : d_year(year), d_month(monst), 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 how appropriately narrow contracts are superior compared to artificially wide ones:

- 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 it 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 out-of-contract use depending on the build mode. 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 precondition checking 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.

Given the considerable resources needed to do comprehensive testing and thorough peer review, it is possible to achieve exemplary quality without precondition checking. In fact, our implementation experience over the past decade shows that enabling precondition checks 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 precondition checking is enabled during development and initial application of unit tests. Hence, even library software developers can benefit from such defensive precondition checking.

When it comes to application software, the benefits of precondition checking 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 precondition 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 the 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 5 application developers how much runtime overhead library software should incur checking for misuse by its client applications, it is quite possible you will get 5 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, 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 precondition checking. Our goal is that the same infrastructure library be able to support all these different application needs throughout all phases of their lifecycles.

Even if we were able to get application developers to agree on the level of runtime precondition checking, 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 precondition checking 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 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 assertion-level 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 violation occurs. Of all the headers in our BSL 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 preconditions, knowing that the runtime overhead can be eliminated in optimized builds. Others, afraid of aborting, hard code precondition 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 precludes 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 precondition 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 build modes that control how much resources should be expended on precondition testing
- a set of precondition assertion macros that validate preconditions
- a violation handling mechanism that controls what is done when a precondition violation occurs
- a set of test macros that can be used in test drivers to verify that preconditions are properly validated

9.1 Build Modes and Assert Macros

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

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

- Optimized build mode is used when an application developer is not willing to expend any appreciable resources on precondition testing. In Optimized build mode only the most inexpensive (e.g., < 5%) and critical tests are performed.

A matching assert macro is provided for each build mode, allowing the *library developer* to express how expensive it is to test each precondition. 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 very inexpensive and/or critical tests would be performed using the Optimized mode assert macro.

Once the library developer has implemented precondition tests with the appropriate assert macros, it is possible for the application developer to control the amount of runtime resources expended on testing by choosing the appropriate build mode in with 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 control (at run time) over what happens when a precondition violation occurs. A configurable violation handler mechanism is provided so that the application owner (i.e., of `main`) can choose to abort the program, throw an exception, or otherwise respond to the violation.

9.3 Test Macros

A precondition-checking facility is not fully useful unless the checks it supports can be tested. A set of test macros are provided to allow library developers to easily test that (a) violations do not occur when all preconditions are met, (b) violations do occur when any preconditions are not met, and (c) each precondition is tested in all the appropriate 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 be allowed to transpire only when in a build mode corresponding to a defensive check that can respond to the particular precondition violation.

10 Examples

10.1 Assert a contract precondition in normal build mode

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

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

10.2 Throw an exception on contract precondition violation

```
#include <exception>
```

```
#include <experimental/contract_assert>

struct contract_error {
    contract_error(const std::experimental::out_of_contract& info)
        : info(info)
    {}

    std::experimental::out_of_contract info;
};

void handle_contract_violation(
        const std::experimental::contract_assert_info& info) {
    throw contract_error(info);
}

int main() {
    std::experimental::set_out_of_contract(handle_contract_violation);

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

10.3 Test that a function correctly asserts its contract preconditions

```
if (TEST_CONTRACT_ASSERT_FAIL(other_strlen(nullptr))) {
    std::cout << "other_strlen precondition assert is correct\n";
}
else {
    std::cout << "other_strlen precondition assert is incorrect\n";
}

if (TEST_CONTRACT_ASSERT_PASS(other_strlen("a string"))) {
    std::cout << "other_strlen precondition assert is correct\n";
}
else {
    std::cout << "other_strlen precondition assert is incorrect\n";
}
```

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, permits no undefined behavior.

11.2 Defensive programming support overview [contract.overview]

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

The *negative testing macros* provide a way to check that a precondition violation occurs when a function is called out of contract, due to appropriate use of precondition assertions. These macros make use of a *negative test context* having a thread-local state indicating either success or failure. When a negative test context is first established, it will have a success state.

The following subclauses describe the assert macros, assert macro state flags, assertion handlers and assert test macros that compromise this facility.

11.3 Header <experimental/contract_assert> synopsis [contract.synopsis]

```
// precondition assert macros
#define CONTRACT_ASSERT_OPT(precondition_expression)  // see below
#define CONTRACT_ASSERT(precondition_expression)      // see below
#define CONTRACT_ASSERT_SAFE(precondition_expression) // see below

#define CONTRACT_ASSERT_DBG(precondition_expression)  \
            CONTRACT_ASSERT(precondition_expression)


// Depending on the current build mode, zero or more of the following precondition
assert macro state flags
// will be defined.
#define CONTRACT_ASSERT_OPT_IS_ACTIVE
#define CONTRACT_ASSERT_IS_ACTIVE
#define CONTRACT_ASSERT_SAFE_IS_ACTIVE

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


// precondition assert test macros
```

```
#define TEST_CONTRACT_ASSERT_OPT_PASS(expression)      // see below
#define TEST_CONTRACT_ASSERT_OPT_FAIL(expression)      // see below
#define TEST_CONTRACT_ASSERT_PASS(expression)          // see below
#define TEST_CONTRACT_ASSERT_FAIL(expression)          // see below
#define TEST_CONTRACT_ASSERT_SAFE_PASS(expression)     // see below
#define TEST_CONTRACT_ASSERT_SAFE_FAIL(expression)     // see below

#define TEST_CONTRACT_ASSERT_DBG_PASS(expression)   \
            TEST_CONTRACT_ASSERT_PASS(expression)
#define TEST_CONTRACT_ASSERT_DBG_FAIL(expression)   \
            TEST_CONTRACT_ASSERT_FAIL(expression)

namespace std {
namespace experimental {

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

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

using out_of_contract_handler = void (*)(const out_of_contract_info&);

// handler manipulators
out_of_contract_handler set_out_of_contract(out_of_contract_handler handler)
                                                                        noexcept;
out_of_contract_handler get_out_of_contract() noexcept;

// handler invoker
[[noreturn]] void out_of_contract(const out_of_contract_info& info);

// negative testing support
[[noreturn]] void negative_testing_handler(const out_of_contract_info& info);


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

11.4 Build mode selection [**contract.modes**]

A Program may define no more than one of the following preprocessor tokens:

```
CONTRACT_ASSERT_LEVEL_NONE
CONTRACT_ASSERT_LEVEL_ASSERT_OPT
CONTRACT_ASSERT_LEVEL_ASSERT
CONTRACT_ASSERT_LEVEL_ASSERT_SAFE
```

Each denotes, respectively "no contract assertion" build mode, "optimized" build mode, "non-optimized" (or "debug") build mode, and "safe" (debug with extra checks) build mode. Upon inclusion of `<experimental/contract_assert>` in any translation unit, the following shall be true, after that point until the next inclusion of the `contract_assert` header:

- When `CONTRACT_ASSERT_LEVEL_NONE` is defined, none of `CONTRACT_ASSERT_OPT_IS_ACTIVE`, `CONTRACT_ASSERT_IS_ACTIVE`, nor `CONTRACT_ASSERT_SAFE_IS_ACTIVE` shall be defined.
- When `CONTRACT_ASSERT_LEVEL_ASSERT_OPT` is defined, only `CONTRACT_ASSERT_OPT_IS_ACTIVE` shall be defined, but neither `CONTRACT_ASSERT_IS_ACTIVE` nor `CONTRACT_ASSERT_SAFE_IS_ACTIVE` shall be defined.
- When `CONTRACT_ASSERT_LEVEL_ASSERT` is defined, both `CONTRACT_ASSERT_OPT_IS_ACTIVE` and `CONTRACT_ASSERT_IS_ACTIVE` shall be defined, but `CONTRACT_ASSERT_SAFE_IS_ACTIVE` shall be defined.
- When `CONTRACT_ASSERT_LEVEL_ASSERT_SAFE` is defined, all of `CONTRACT_ASSERT_OPT_IS_ACTIVE`, `CONTRACT_ASSERT_IS_ACTIVE`, and `CONTRACT_ASSERT_SAFE_IS_ACTIVE` shall be defined.

If more than one of the `CONTRACT_ASSERT_LEVEL` macros has been defined upon inclusion of `<experimental/contract_assert>` in any translation unit, the program shall be ill-formed.

`<experimental/contract_assert>` may be included more than once with no effect different from being included exactly once, except that the effect of each repeated inclusion depends on the lexically current definition of `CONTRACT_ASSERT_LEVEL_NONE`, `CONTRACT_ASSERT_LEVEL_ASSERT_OPT`, `CONTRACT_ASSERT_LEVEL_ASSERT`, and `CONTRACT_ASSERT_LEVEL_ASSERT_SAFE`.

11.5 Precondition assert macros [**contract.assertions**]

When a precondition violation occurs due to evaluation of a `precondition_expression` in a precondition assert macro, the implementation will call `std::experimental::out_of_contract` with the `__FILE__`, `__LINE__`, and stringization of the `precondition_expression`, and the `contract_assert_mode` corresponding to that precondition assert macro.

```
#define CONTRACT_ASSERT_OPT(precondition_expression)  // see below
```

This precondition assertion macro evaluates the `precondition_expression` only in optimized build mode; a precondition violation occurs if and only if the `precondition_expression` evaluates to `false`.

Effects: Evaluate the `precondition_expression` if and only if `CONTRACT_ASSERT_OPT_IS_ACTIVE` is defined. If the `precondition_expression` evaluates to `true`, do nothing, and otherwise a precondition violation occurs.

```
#define CONTRACT_ASSERT(precondition_expression)      // see below
#define CONTRACT_ASSERT_DBG(precondition_expression)  \
          CONTRACT_ASSERT(precondition_expression)
```

This precondition assertion macro evaluates the `precondition_expression` only in optimized or non-optimized (debug) build mode; a precondition violation occurs if and only if the `precondition_expression` evaluates to `false`.

Effects: Evaluate the `precondition_expression` if and only if `CONTRACT_ASSERT_IS_ACTIVE` is defined. If the `precondition_expression` evaluates to `true`, do nothing, and otherwise a precondition violation occurs.

```
#define CONTRACT_ASSERT_SAFE(precondition_expression) // see below
```

This precondition assertion macro evaluates the `precondition_expression` only in optimized, non-optimized (debug), or safe (debug with extra checks) build mode; a precondition violation occurs if and only if the `precondition_expression` evaluates to `false`.

Effects: Evaluate the `precondition_expression` if and only if `CONTRACT_ASSERT_SAFE_IS_ACTIVE` is defined. If the `precondition_expression` evaluates to `true`, do nothing, and otherwise a precondition violation occurs.

11.6 Precondition assert macro state flags [**contract.flags**]

```
#define CONTRACT_ASSERT_OPT_IS_ACTIVE
```

Effects: Defined only if `CONTRACT_ASSERT_OPT` asserts the precondition expression.

> Note: Rationale for using uppercase: this and other `IS_ACTIVE` flags are supposed to be tested with `#ifdef` just like `NDEBUG`, which is uppercase.

```
#define CONTRACT_ASSERT_IS_ACTIVE
#define CONTRACT_ASSERT_DBG_IS_ACTIVE
```

Effects: Both macros are defined only if `CONTRACT_ASSERT` and `CONTRACT_ASSERT_DBG` assert the precondition expression.

```
#define CONTRACT_ASSERT_SAFE_IS_ACTIVE
```

Effects: Defined only if `CONTRACT_ASSERT_SAFE` asserts the precondition expression.

11.7 Precondition assert test macros [**contract.test.macros**]

```
#define TEST_CONTRACT_ASSERT_OPT_PASS(expression)
```

This negative testing macro evaluates the `expression` only in optimized build mode. If no precondition violation occurs, this macro expands to an expression that evaluates as `true`, otherwise this macro expands to an expression that evaluates as `false`.

Effects: If and only if `CONTRACT_ASSERT_OPT_IS_ACTIVE` is defined, evaluate `expression` within a negative test context. If, after evaluating the `expression`, the negative test context is in a failure state, this macro expands to an expression that evaluates as `false`, and otherwise it expands to an expression that evaluates as `true`.

```
#define TEST_CONTRACT_ASSERT_OPT_FAIL(expression)
```

This negative testing macro evaluates the `expression` only in optimized build mode. If the `expression` is not evaluated, or a precondition violation occurs, this macro expands to an expression that evaluates as `true`, otherwise this macro expands to an expression that evaluates as `false`.

Effects: If and only if `CONTRACT_ASSERT_OPT_IS_ACTIVE` is defined, evaluate `expression` within a negative test context. If, after evaluating the `expression`, the negative test context is in a success state, this macro expands to an expression that evaluates as `false`, and otherwise it expands to an expression that evaluates as `true`.

```
#define TEST_CONTRACT_ASSERT_PASS(expression)
#define TEST_CONTRACT_ASSERT_DBG_PASS(expression)
```

This negative testing macro evaluates the `expression` only in optimized or non-optimized (debug) build mode. If no precondition violation occurs, this macro expands to an expression that evaluates as `true`, otherwise this macro expands to an expression that evaluates as `false`.

Effects: If and only if `CONTRACT_ASSERT_IS_ACTIVE` is defined, evaluate `expression` within a negative test context. If, after evaluating the `expression`, the negative test context is in a failure state, this macro expands to an expression that evaluates as `false`, and otherwise it expands to an expression that evaluates as `true`.

```
#define TEST_CONTRACT_ASSERT_FAIL(expression)
#define TEST_CONTRACT_ASSERT_DBG_FAIL(expression)
```

This negative testing macro evaluates the `expression` only in optimized or non-optimized (debug) build mode. If the `expression` is not evaluated, or a precondition violation occurs, this macro expands to an expression that evaluates as `true`, otherwise this macro expands to an expression that evaluates as `false`.

Effects: If and only if `CONTRACT_ASSERT_IS_ACTIVE` is defined, evaluate `expression` within a negative test context. If, after evaluating the `expression`, the negative test context is in a success state, this macro expands to an expression that evaluates as `false`, and otherwise it expands to an expression that evaluates as `true`.

```
#define TEST_CONTRACT_ASSERT_SAFE_PASS(expression)
```

This negative testing macro evaluates the `expression` only in optimized, non-optimized (debug), or safe (debug with extra checks) build mode. If no precondition violation occurs, this macro expands to an expression that evaluates as `true`, otherwise this macro expands to an expression that evaluates as `false`.

Effects: If and only if `CONTRACT_ASSERT_SAFE_IS_ACTIVE` is defined, evaluate `expression` within a negative test context. If, after evaluating the `expression`, the negative test context is in a failure state, this macro expands to an expression that evaluates as `false`, and otherwise it expands to an expression that evaluates as `true`.

```
#define TEST_CONTRACT_ASSERT_SAFE_FAIL(expression)
```

This negative testing macro evaluates the `expression` only in optimized, non-optimized (debug), or safe (debug with extra checks) build mode. If the `expression` is not evaluated, or a precondition violation occurs, this macro expands to an expression that evaluates as `true`, otherwise this macro expands to an expression that evaluates as `false`.

Effects: If and only if `CONTRACT_ASSERT_SAFE_IS_ACTIVE` is defined, evaluate `expression` within a negative test context. If, after evaluating the `expression`, the negative test context is in a success state, this macro expands to an expression that evaluates as `false`, and otherwise it expands to an expression that evaluates as `true`.

11.8 Precondition assert handler function [**contract.handler**]

```
using out_of_contract_handler = void (*)(const out_of_contract_info&);
```

The type of a *handler function* to be called when a precondition assert violation occurs.

Required behavior: An `out_of_contract_handler` shall not return to the caller. [*Note:* It may throw an exception. — *end note*]

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

Remarks: The implementation of a custom `out_of_contract_handler` should not hard-code the `out_of_contract_info` structure into its code, nor should the implementation rely on the exact layout of `out_of_contract_info`, as the structure is subject to extension in the future.

```
out_of_contract_handler set_out_of_contract(out_of_contract_handler handler)
                                                                    noexcept;
```

Effects: Establishes the function designated by handler as the current handler function for precondition assertion violations.

Remarks: It is unspecified whether a null pointer value designates the default `out_of_contract_handler`.

Returns: The previous `out_of_contract_handler`.

```
out_of_contract_handler get_out_of_contract() noexcept;
```

Returns: The current `out_of_contract_handler`. Note: This can be a null pointer value.

```
[[noreturn]] void out_of_contract(const out_of_contract_info& info);
```

Remarks: Called by the implementation when any of the precondition assertion macros fail. May also be called directly by a program.

Effects: Calls the current `out_of_contract_handler` function. Note: A default `out_of_contract_handler` is always considered a callable handler in this context.

11.9 Precondition assert testing components [contract.test.support]

```
[[noreturn]] void negative_testing_handler(const out_of_contract_info& info);
```

Remarks: This `out_of_contract_handler` function must be installed to support the test macros defined in [contract.flags].

Effects: If called within a negative test context, set the context to a failure state, and return control to the context. [*Note:* This may involve throwing an exception. — *end note*] Otherwise call `std::abort`.

12 Precondition assert test macros reference implementation

12.1 Overview

The precondition assert test macro implementation consists of the following parts:

1. Definition of an exception class that will be thrown on a precondition violation
2. Definition of a precondition violation handler function that throws the exception
3. Definition of the function `test_contract_assert_imp`, which performs the following tasks:
   - Replace the default precondition violation handler function.
   - Evaluate the `expression` under test inside a try/catch block.
   - Catch the exception and verify that it was indeed expected to be thrown.
   - Restore the original precondition violation function.

12.2 Implementation

The reference implementation below presents an example implementation of `experimental/contract_assert`. 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` macro will be set by the build system.

12.2.1 experimental/contract_assert

```
#include <atomic>
#include <cstdlib>

// macros

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

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

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

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

#ifdef CONTRACT_ASSERT_LEVEL_NONE
#ifdef CONTRACT_ASSERT_LEVEL_ASSERT_OPT
#error "Cannot define more than one CONTRACT_ASSERT_LEVEL macro at one time"
#endif
#ifdef CONTRACT_ASSERT_LEVEL_ASSERT
#error "Cannot define more than one CONTRACT_ASSERT_LEVEL macro at one time"
#endif
#ifdef CONTRACT_ASSERT_LEVEL_ASSERT_SAFE
#error "Cannot define more than one CONTRACT_ASSERT_LEVEL macro at one time"
#endif
#endif  // CONTRACT_ASSERT_LEVEL_NONE

#ifdef CONTRACT_ASSERT_LEVEL_ASSERT_OPT
#ifdef CONTRACT_ASSERT_LEVEL_ASSERT_NONE
#error "Cannot define more than one CONTRACT_ASSERT_LEVEL macro at one time"
#endif
#ifdef CONTRACT_ASSERT_LEVEL_ASSERT
#error "Cannot define more than one CONTRACT_ASSERT_LEVEL macro at one time"
#endif
#ifdef CONTRACT_ASSERT_LEVEL_ASSERT_SAFE
#error "Cannot define more than one CONTRACT_ASSERT_LEVEL macro at one time"
#endif

#define CONTRACT_ASSERT_OPT_IS_ACTIVE

#endif  // CONTRACT_ASSERT_LEVEL_ASSERT_OPT

#ifdef CONTRACT_ASSERT_LEVEL_ASSERT
#ifdef CONTRACT_ASSERT_LEVEL_ASSERT_NONE
#error "Cannot define more than one CONTRACT_ASSERT_LEVEL macro at one time"
```

```
#endif
#ifdef CONTRACT_ASSERT_LEVEL_ASSERT_OPT
#error "Cannot define more than one CONTRACT_ASSERT_LEVEL macro at one time"
#endif
#ifdef CONTRACT_ASSERT_LEVEL_ASSERT_SAFE
#error "Cannot define more than one CONTRACT_ASSERT_LEVEL macro at one time"
#endif

#define CONTRACT_ASSERT_OPT_IS_ACTIVE
#define CONTRACT_ASSERT_IS_ACTIVE
#define CONTRACT_ASSERT_DBG_IS_ACTIVE


#endif  // CONTRACT_ASSERT_LEVEL_ASSERT

#ifdef CONTRACT_ASSERT_LEVEL_ASSERT_SAFE
#ifdef CONTRACT_ASSERT_LEVEL_ASSERT_NONE
#error "Cannot define more than one CONTRACT_ASSERT_LEVEL macro at one time"
#endif
#ifdef CONTRACT_ASSERT_LEVEL_ASSERT_OPT
#error "Cannot define more than one CONTRACT_ASSERT_LEVEL macro at one time"
#endif
#ifdef CONTRACT_ASSERT_LEVEL_ASSERT
#error "Cannot define more than one CONTRACT_ASSERT_LEVEL macro at one time"
#endif

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

#endif  // CONTRACT_ASSERT_LEVEL_ASSERT_SAFE


#ifdef CONTRACT_ASSERT_OPT_IS_ACTIVE

#define CONTRACT_ASSERT_OPT(expr)                                        \
    do                                                                   \
    {                                                                    \
        if (!(expr))                                                     \
        {                                                                \
            std::experimental::out_of_contract_info info = {             \
                    std::experimental::contract_assert_mode::opt,        \
                    #expr,                                               \
                    __FILE__,                                            \
                    __LINE__                                             \
            };                                                           \
            std::experimental::out_of_contract(info);                    \
        }                                                                \
    } while (0)

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

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

#else

#define CONTRACT_ASSERT_OPT(expr)            do {} while(0)
#define TEST_CONTRACT_ASSERT_OPT_PASS(expr)  (true)
#define TEST_CONTRACT_ASSERT_OPT_FAIL(expr)  (true)

#endif // CONTRACT_ASSERT_OPT_IS_ACTIVE


#ifdef CONTRACT_ASSERT_IS_ACTIVE

#define CONTRACT_ASSERT(expr)                                            \
    do                                                                   \
    {                                                                    \
        if (!(expr))                                                     \
        {                                                                \
            std::experimental::out_of_contract_info info = {             \
                    std::experimental::contract_assert_mode::dbg,        \
                    #expr,                                               \
                    __FILE__,                                            \
                    __LINE__                                             \
            };                                                           \
            std::experimental::out_of_contract(info);                    \
        }                                                                \
    } while (0)

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

#define TEST_CONTRACT_ASSERT_FAIL(expr)                                  \
    std::experimental::detail::test_contract_assert_imp(                 \
            std::experimental::contract_assert_mode::dbg, false,         \
            [&]{ expr; })

#else

#define CONTRACT_ASSERT(expr)            do {} while(0)
#define TEST_CONTRACT_ASSERT_PASS(expr)  (true)
#define TEST_CONTRACT_ASSERT_FAIL(expr)  (true)

#endif // CONTRACT_ASSERT_IS_ACTIVE

#ifdef CONTRACT_ASSERT_SAFE_IS_ACTIVE

#define CONTRACT_ASSERT_SAFE(expr)                                       \
    do                                                                   \
    {                                                                    \
        if (!(expr))                                                     \
        {                                                                \
            std::experimental::out_of_contract_info info = {             \
                    std::experimental::contract_assert_mode::safe,       \
                    #expr,                                              \
                    __FILE__,                                           \
                    __LINE__                                            \
            };                                                          \
            std::experimental::out_of_contract(info);                   \
        }                                                               \
    } while (0)

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

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

#else

#define CONTRACT_ASSERT_SAFE(expr)            do {} while(0)
#define TEST_CONTRACT_ASSERT_SAFE_PASS(expr)  (true)
#define TEST_CONTRACT_ASSERT_SAFE_FAIL(expr)  (true)

#endif // CONTRACT_ASSERT_SAFE_IS_ACTIVE

#define CONTRACT_ASSERT_DBG(expr)           CONTRACT_ASSERT(expr)
#define TEST_CONTRACT_ASSERT_DBG_PASS(expr) TEST_CONTRACT_ASSERT_PASS(expr)
#define TEST_CONTRACT_ASSERT_DBG_FAIL(expr) TEST_CONTRACT_ASSERT_FAIL(expr)



namespace std
{
namespace experimental
{

// interface

enum class contract_assert_mode
{
    opt,
    dbg,
    safe
};

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

using out_of_contract_handler = void (*)(const out_of_contract_info&);

out_of_contract_handler set_out_of_contract(out_of_contract_handler handler)
```

```
                                                                      noexcept;
out_of_contract_handler get_out_of_contract() noexcept;
[[noreturn]] void out_of_contract(const out_of_contract_info& info);

[[noreturn]] void negaive_testing_handler(const out_of_contract_info& info);

// implementation

namespace detail
{

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

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

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

    contract_assert_mode mode;
};

out_of_contract_handler set_negative_testing_handler(
                                     out_of_contract_handler handler) noexcept;
out_of_contract_handler get_negative_testing_handler() noexcept;

struct negative_testing_guard
{
    explicit negative_testing_guard(out_of_contract_handler handler)
        : old_handler{set_negative_testing_handler(handler)}
    {}

    negative_testing_guard(const negative_testing_guard &) = delete;

    ~negative_testing_guard()
    {
        set_negative_testing_handler(old_handler);
    }

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

out_of_contract_handler old_handler;
};

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

thread_local
out_of_contract_handler installed_negative_testing_handler{nullptr};

inline
out_of_contract_handler set_negative_testing_handler(
                                      out_of_contract_handler handler) noexcept
{
    out_of_contract_handler old = installed_negative_testing_handler;
    installed_negative_testing_handler = handler;
    return old;
}

inline out_of_contract_handler get_negative_testing_handler() noexcept
{
    return installed_negative_testing_handler;
}

template <typename EXPRESSION>
bool test_contract_assert_imp(contract_assert_mode mode,
                              bool expect_pass,
                              EXPRESSION expr)
{
    negative_testing_guard guard{exception_handler};

    try
    {
        expr();

        // the assert passed, return true if it was expected
        return expect_pass;
    }
    catch (const contract_error& error)
    {
        // the assert failed, return true if it was expected
        // and the mode is correct
        return error.mode <= mode && false == expect_pass;
    }
}

}  // namespace detail

[[noreturn]] inline
void negative_testing_handler(const out_of_contract_info& info)
{
    [[noreturn]] out_of_contract_handler handler
                                      = detail::get_negative_testing_handler();
    if (!handler) {
        std::abort();
    }
    handler(info);
}

inline
out_of_contract_handler set_out_of_contract(out_of_contract_handler handler)
                                                                       noexcept
{
    return std::atomic_exchange(&detail::handler, handler);
}
```

```
inline
out_of_contract_handler get_out_of_contract() noexcept
{
    return std::atomic_load(&detail::handler);
}

[[noreturn]]
inline
void out_of_contract(const out_of_contract_info& info)
{
    [[noreturn]] out_of_contract_handler handler = get_out_of_contract();
    handler(info);
}

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

12.2.2 Simple test driver for experimental/contract_assert

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

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

    std::size_t length = 0;
    for (; *str; ++length, ++str)
        ;

    return length;
}

// A 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()
{
    // Enable negative testing.

    std::experimental::set_out_of_contract(
                                  std::experimental::negative_testing_handler);

    // Run some negative 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.