Doc No: N4135
Date: 2014-10-09
Authors: John Lakos (jlakos@bloomberg.net)
Alexei Zakharov (alexeiz@gmail.com)
Alexander Beels (abeels@bloomberg.net)
Nathan Myers (nmyers12@bloomberg.net)

Language Support for Runtime Contract Validation (Revision 8)

Abstract

With enough care we can build libraries that are essentially defect-free, but even the best library may fail catastrophically when misused. Runtime contract validation, the practice of checking functions' preconditions when they are called, helps discover misuse in early testing, speeding development and making software more robust. Extending support for contract validation to phases of development beyond early testing would yield substantial further benefits.

We propose simple facilities to help library developers, application developers, and language implementers cooperate toward our common goal of delivering efficient programs without defects. *Library developers* get a common framework to express the contracts offered by their library functions, without compromising performance or interface simplicity. *Application developers* get the option to specify, without reference to details of the libraries they use, how much run time to spend on validation, and precisely what to do when a violation is detected. *Implementations* get permission (and encouragement) to infer programmers' intentions directly from contract assertions, and to use the inferences in all phases of translation.

This design derives from over a decade of production software development at Bloomberg LP. A variant implementation is freely available today, along with copious usage examples baked into production-grade library code, in Bloomberg's open-source distribution of the BDE library at https://github.com/bloomberg/bde.

Contents

Table of Contents

1 Document History

This proposal is based on N4075, based on N3997, based on N3963.

The major changes from N4075 are the wholesale elimination of encouragement toward (1) implementation using the preprocessor, and (2) apparent ODR violations. In addition, names were changed, some parts were simplified, and the exposition was re-focused on standardization. The proposed facility is not presented as purely a library component, but it requires no new syntax, and no changes to current semantics.

2 Introduction

Any library may fail catastrophically if misused. We make our libraries as easy to use and as hard to misuse as we can, and we catch misuse at compile time wherever we can. Where we cannot, we are left to depend on runtime contract validation: actually checking checkable preconditions on function entry. Contractual methods, including runtime validation, have already delivered impressive gains in quality, cost, and productivity, but we have found that they can do much more.

This proposal offers *library developers* a concise notation to express contract preconditions that can be validated at runtime, and offers *application developers* simple means to control the runtime consequences of validation, thereby extending its benefits well beyond previous bounds, and resolving fundamental conflicts between library performance, interface simplicity, and safety.

We propose further to empower *implementations* to use inferences from the new annotations in all phases of translation, for better compile-time error detection and smaller and faster generated code.

3 Background

`std::vector<T>::push_back` may be called any time, on any vector instance, with no risk to your program state. This member function offers a *wide* contract: No combination of arguments and well-defined prior state can evoke undefined behavior. Another member, **`pop_back`**, offers a *narrow* contract: Its effect is defined only if its precondition—that the vector instance not be empty—is satisfied.

A program that may violate such a precondition harbors a defect. Violations can often be caught by runtime checking, but such checking always costs extra code space and run time. Such costs are often small, but can be very large. Catching a violation when it happens might be worth any expense; yet, where there is no defect, every cycle spent on checking is wasted. This conflict is fundamental, and cannot be resolved within a library component like `std::vector<>`.

It is precisely the undefined effect of a violation that gives us latitude to avoid the expense of checking, or detect the violation and act on it. Tools and methods to specify requirements and to instrument functions in this way have turned out to be powerful aids to meeting core software engineering goals.

4 Motivation

4.1 The Value of Checking

Library developers naturally prefer to check for bad usage where they can—catching users' mistakes early prevents both bugs and spurious bug reports—but the consequences on performance and interface design simplicity often forbid it. Resolve these, and developers will be free to make libraries as helpful as they can, without compromise.

Whereas library development costs can often be amortized over many downstream uses, applications typically support only their own development, and application-level testing is notoriously limited. Libraries instrumented to validate usage contracts amplify the effectiveness of whatever testing is done, anywhere a defect can produce a detectably bad library call.

4.2 Overhead and Response to Failure

Consider an interactive editor, close to release: The developer needs customers to use the program for real work, to flush out bugs. If runtime validation is enabled in the libraries the program uses, and upon detecting a violation the program just aborts, then customers, who would risk losing hours of valuable work, sensibly refuse to use it, and the developer learns nothing. Disable checking, and the program crashes anyway, a little later—or, worse, silently corrupts the customer's documents. Let the program instead log the violation, save the customer's data, and restart, and the libraries' runtime validation has helped even in preparations for release.

In different circumstances the same program, when it detects a violation, might better freeze and wait for a debugger to be attached, or abort immediately so a test script can start the next test. Similarly, during early development it would best perform every check possible, but in beta testing do only sanity checks, and in performance tuning avoid all checking. These are not choices that those who write the libraries that the program depends on can reasonably be expected to address in detail. Still, libraries that check for and report violations under control of downstream users offer benefits well beyond early development.

4.3 Compiler Hinting

Assertions' usefulness is not limited to testing. When compilers may infer programmers' intentions and the bounds on a program's runtime state space directly from contract-validation expressions, the benefits may be extended both backward to more thorough compile-time error checking, and forward to smaller and faster released code.

In particular, a compiler might determine that a call to an inline or template function passes values that would violate a precondition, and report an error. Furthermore, if a violated assertion can be interpreted to imply that the program's runtime state would not be well defined, the compiler can skip generating code that could only be

reached in such a case. Code elision can propagate back up the call chain; any path certain to reach the elided code can also be omitted. Eliding dead code reduces instruction-cache pressure, speeding execution of the live code that remains.

4.4 Design Goals

In short: Library authors need to easily code contract-validation checks, concisely express their cost relative to the useful work a function does, and verify that the checking they do is itself correct.

Program authors (i.e., of **`main`**) need to be able to choose, when building, how much contract-validation overhead to accept, and be able to specify the precise action to take when a violation occurs.

Implementations need the latitude to use the implications of contract-validation assertions, while compiling, to identify errors and to guide code generation.

5 Scope

This facility is intended for ubiquitous use across all library and application software.

6 Existing Practice

Contractual specifications with runtime enforcement are used in virtually all computer languages. C++ developers will be familiar with `<cassert>`'s limitations.

For more than a decade, Bloomberg's library infrastructure has successfully employed the strategy advocated here, across a wide range of applications and libraries. Copious usage examples are available for public scrutiny [1].

7 Impact on the Standard

This proposal requires, *for a minimal conforming implementation*, no new core language features. It introduces no new syntax. Adopting this proposal has no direct effect on the rest of the standard, although once it is accepted, library implementers may be asked by customers to instrument the standard library. Similarly, compiler implementers would be invited to use contract assertions to help improve static error detection and code generation.

8 Summary of Proposal for Standardization

We propose:

- three *validating build modes* to give application developers control over how much contract validation is built into their programs
- three corresponding *contract assertions* for use in library and application code to express preconditions, and to detect and report *contract violations*
- a common, configurable c*ontract-violation handler* to give application developers precise control over what happens when a violation is detected

- *contract-assertion unit-test* apparatus for use in test programs to verify that contract assertions are performing as intended

8.1 Validating Build Modes

Application developers need to control how much of a program's run time is spent on contract validation. *Build modes* let them select, at compile time, which of the contract-validation checks coded into functions are run, and which are skipped.

- Programs built in "safe" mode might spend more time checking preconditions than doing useful work; *all* checks are enabled.
- Programs built in "dbg" mode skip checks that would slow them very noticeably.
- Programs built in "opt" mode skip all but the least expensive and most critical sanity checks.
- Programs built without validation have all code to perform checking omitted.

Regardless of the build mode used, the compiler may use any guidance it can infer from contract assertions it sees to do better error checking and code generation, even when no checking code ends up in the compiled program.

8.2 Contract Assertions

We introduce three source code forms called *contract assertions*, one for each validating build mode above. Each expresses a contract precondition, analogously to the traditional `assert` macro. Library programmers will write disproportionately costly checks using the "safe" mode contract assertion, moderately expensive checks using the "dbg" assertion, and very inexpensive or critical checks using the "opt" assertion.

Thus, besides catching misuse, contract assertions implicitly record the programmer's assessment of their runtime cost and importance relative to the useful work the function performs. Furthermore, the contract assertions provide to the compiler extra information that it may use to detect more usage errors and produce faster, more compact object code.

8.3 Violation Handling

Application developers need precise control over what happens when a library detects a contract violation. In this proposal, the response is to call a *contract violation handler*, a function the program author (*i.e.*, of **`main`**) may provide, and which may do anything except return to its caller. In multi-threaded programs, each thread may have its own handler.

8.4 Contract-Assertion Unit Test forms

Runtime contract-validation checks are code, and like all code they need to be tested. This proposal includes *contract-assertion unit test* forms that library developers may

place in their test programs to help verify that the validation checks in their library are themselves performing correctly.

9 Examples

9.1 *Check a contract precondition in* safe *and* dbg*, but not* opt *build modes*

A **`strlen`**-like function, **`c_string_length`**, has a precondition that `string` must not be null. The form `contract_assert` checks the precondition when the program is built with "dbg" and "safe", but not "opt" build modes:

```
#include <contract_assert>
#include <cstddef>

namespace lib {
std::size_t c_string_length(char const* string)                    // O(n)
{
    contract_assert(string != nullptr);                            // O(1)
    ...
```

9.2 *Check a contract precondition only in* safe *build mode*

The function below is specified to run in *O(log n)* time. To validate its requirement for a sorted table would add *O(n)* time, violating the specification. Partial, incremental checking within the loop is almost as effective, and takes, cumulatively, only *O(log n)* time, yet it still nearly doubles the run time. By using `contract_assert_safe`, the heavy runtime cost is incurred only in the "safe" build mode:

```
#include <contract_assert>
#include <algorithm>
#include <cstddef>

template <typename T>
bool binary_search(T const* table, std::size_t length, T target)    // O(log n)
{
    contract_assert(table != nullptr)                               // O(1)
    // contract_assert_safe(std::is_sorted(table, table + length));    O(n): no
    while (len != 0) {
        std::size_t step = length / 2;
        T candidate = table[step];
        contract_assert_safe(table[0] <= candidate);
        contract_assert_safe(candidate <= table[length - 1]);       // O(log n)
        if (candidate < target) {
            table += step + 1;
            length -= step + 1;
        } else if (target < candidate) {
            length = step;
        } else
            return true;
    }
    return false;
}
```

9.3 *Throw an exception when a contract violation is detected*

A library function, `lib::`**`unimplemented_function`**, unconditionally asserts a contract violation. This program installs a contract violation handler function that throws its argument. When the program subsequently calls the function, the main program catches the exception and emits a diagnostic message before exiting normally.

```
#include <contract_assert>
#include <iostream>

namespace lib {
int unimplemented_function()
{
    contract_assert_opt(false);  //  contract forbids calling this
}
} // namespace lib

int main()
{
    std::set_contract_violation_handler(
        [](std::contract_violation_info const& info) { throw info; });

    try {
        // ...
        lib::unimplemented_function();
        // ...
    } catch (std::contract_violation_info const& info) {
        std::cerr << "Detected a contract violation at "
                  << info.filename << ":" << info.line_number << ".\n";
        return 1;
    }
    return 0;
}
```

This example uses `std::`**`set_contract_violation_handler`**, so the handler installed becomes the global default handler for all threads until the program ends.

9.4 *Install a* thread-local *contract violation handler*

This program installs a thread-local, block-scoped contract-violation handler (which throws) by constructing a local `std::contract_violation_guard` instance. Then, it calls **`c_string_length`** with a bad argument and catches the resulting exception:

```
namespace tst {
int unit_test_validation_checks()
{
    std::contract_violation_guard GUARD(
       [](std::contract_violation_info const& info) { throw info; });

    try {
        // ...
        c_string_length(nullptr);
        // ...
    } catch (std::contract_violation_info const&) {
         return 0;           // runs GUARD.~contract_violation_guard()
    }
```

```
    std::cerr << "negative test 1 failed.\n";
    return 1;                // runs GUARD.~contract_violation_guard()
}
} // namespace tst

int main()
{
    int num_test_failures = 0;

    // (A) Possibly spawn other threads . . .

    num_test_failures += tst::unit_test_validation_checks();

    // (B) Call other tests . . .

    std::cerr << num_test_failures << " tests failed.\n";
    return num_test_failures;
}
```

Any threads spawned by **main** at (A) will default to using the global contract violation handler, even after `tst::`**unit_test_validation_checks** installs its local handler, and `tst::`**unit_test_validation_checks** will be unaffected by any handlers installed by other threads. Similarly, other functions called at (B) will not be affected by the local handler installed in `tst::`**unit_test_validation_checks** because, by that point, the default handler has been restored by the `GUARD` object's destructor.

9.5 Verify that a function correctly asserts its preconditions

The *contract-assertion unit test* forms roll up the exception and contract violation handling seen in the previous example into one line:

```
#include <contract_assert>
#include <iostream>
#include "lib"

int main()
{
    std::cout << (contract_assert_pass( lib::c_string_length("a string") ) ?
        "Correctly detects no contract violation.\n" :
        "Incorrectly reports a contract violation.\n")
    std::cout << (contract_assert_fail( lib::c_string_length(nullptr) ) ?
        "Successfully detects a contract violation.\n" :
        "Fails to detect a contract violation.\n");
}
```

10 Discussion

For any organization that develops most of its code in-house, many of the benefits promised in this proposal may be had by simply copying the design; a minimal implementation is almost trivial. If independent library authors were to do the same, their users would face a forest of handler mechanisms, each slightly different from the last. With a single, common mechanism, instrumenting a library for contract validation adds value without adding to application developers' burdens.

The benefits of a minimal implementation end there, but contract assertions can do much more than just aid testing; they express, unambiguously, the intent of the programmer. An implementation permitted by the standard to treat the contract assertions as definitive can use inferences from them to improve semantic analysis, error detection, and code generation in *all build modes*, particularly those in which the check-expressions do not themselves end up in object code.

Under this proposal, a library author who wishes to expose validation expressions to callers would code them inline (perhaps followed by delegation to a private helper function) in a header. Additional validation might be added, removed, or changed in private library code without requiring downstream users to recompile. Users of libraries that are not instrumented (yet) may code validation at call sites. In an apparently-competing proposal, contract requirements would instead be expressed using new syntax in function *declarations*, making them necessarily part of the public interface. Consider the incremental checking seen in example 9.2 above: It illustrates a use case that the declarative approach alone would not support well.

Notably, a declarative mechanism could, without compromise, be added as a *pure extension* to what is proposed here, re-using its violation-handling machinery. This much simpler proposal could be approved, implemented and in use while details of the more ambitious design are still being worked out, and would remain useful thereafter.

11 Formal Wording

11.1 Definitions

Add three new definitions to clause 17.3 [definitions]:

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

contract

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

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

narrow contract

A narrow contract is a contract that specifies behavior for, and only for, a precisely and completely identified proper subset of all possible combinations of arguments and prior state that are consistent with the language definition. [*Note:* “Consistent with...” excludes from the set otherwise invalid programs, such as those passing misaligned pointers or already-destroyed objects, “null references” (but not null pointers), and all cases in which program's behavior is already undefined. — *end note*] Outside said subset, the behavior is entirely unconstrained—possibly, but not necessarily resulting in undefined behavior.

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

wide contract

A wide contract is a contract that specifies well-defined behavior for all possible combinations of arguments and prior program states permitted by the language.

11.2 Contract Support [**contract**]

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

The header `<experimental/contract_assert>` defines names and exposes apparatus including *contract-assertion* forms, *build-mode flags*, *contract-assertion unit-test* forms, and functions and types to manage *contract violation handlers*.

[*Note:* A *contract assertion* conditionally evaluates a *check-expression*. If it is evaluated, and found to be `false`, a *contract violation* is *detected*. On detecting a violation, a *contract violation handler* function is called: a thread-local handler, if the thread that detects the violation has one installed; otherwise, a global handler. The handler installed may be any syntactically compatible function that does not return to its caller. — *end note*]

[*Note:* A *build-mode flag* is a preprocessor symbol that is defined wherever a *validating build mode* is in effect, and indicates which *contract assertions* may detect violations. — *end note*]

[*Note:* A *contract-assertion unit test* provides a concise way to verify that a function's contract assertions actually detect specific misuses. — *end note*]

The following subclauses describe the contract-assertion forms, build-mode flags, contract violation handlers, contract-assertion unit-test forms, and other names defined in `<experimental/contract_assert>`.

Header <experimental/contract_assert> synopsis

```
// [contract.assertions] contract-assertion forms
// contract_assert_opt(check_expression)
// contract_assert_dbg(check_expression)
// contract_assert_safe(check_expression)
// contract_assert(check_expression)

// [contract.flag] contract assertion build-mode flags
// contract_assert_build_mode_safe
// contract_assert_build_mode_dbg
// contract_assert_build_mode_opt

// [contract.tests] contract-assertion unit-test forms
// contract_assert_fail_opt(test_expression)
// contract_assert_fail_dbg(test_expression)
// contract_assert_fail_safe(test_expression)
// contract_assert_fail(test_expression)
// contract_assert_pass(test_expression)

namespace std {
inline namespace experimental {
```

```
inline namespace fundamentals_v2 {

// [contract.assertions] types
enum class contract_assertion_mode { opt, dbg, safe };

// [contract.violation.info] struct contract_violation_info
struct contract_violation_info;

// [contract.handler.types] handler types
using contract_violation_handler = void (*)(contract_violation_info const& info);

// [contract.handler.manipulation] handler manipulation
contract_violation_handler
set_contract_violation_handler(contract_violation_handler handler) noexcept;

contract_violation_handler
get_contract_violation_handler() noexcept;

// [contract.handler.invocation] handler invocation
[[noreturn]] void
handle_contract_violation(contract_violation_info const& info);

// [contract.handler.guard] thread-local contract violation handler installer
struct contract_violation_guard {
    // unspecified members
};

}}}  // namespaces
```

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

At any point in a translation unit, at most one of four *build modes* described in Table 1 is in effect. Before the first inclusion of `<experimental/contract_assert>` in a translation unit, none of the names specified in [contract.general] are defined, and no build mode is in effect. Implementations shall provide a means, as part of initiating translation on each translation unit and outside of the program text, that any one of the build modes listed in Table 1 may be selected. If a build mode is so selected, it is in effect from the first place that the `<experimental/contract_assert>` header is included through to the end of the translation unit, not excepting included headers.

Table 1

Build Mode	Description
(none)	No contract preconditions are checked
“opt”	Only the least expensive contract preconditions are checked.
“dbg”	Up to moderately expensive contract conditions are checked.
“safe”	All expressed contract conditions are checked.

Each successive build mode listed in Table 1 is said to be *stronger* than the preceding build mode or modes. The final three are called *validating build modes*.

If a build mode was not selected at translation initiation, then an *implementation-specified* choice of one of the four build modes is in effect from the first place where `<experimental/contract_assert>` is included, through to the end of the translation unit. [*Note:* Implementations are encouraged to select the "dbg" mode by default, by analogy to `<cassert>` and `NDEBUG`. — *end note*]

11.2.3 Contract assertions [**contract.assertions**]

`<experimental/contract_assert>` defines three contract assertion forms,

```
contract_assert_opt(check_expression)
contract_assert_dbg(check_expression)
contract_assert_safe(check_expression)
```

and one alias

```
contract_assert(check_expression)
```

The alias is an abbreviation for `contract_assert_dbg`.

Table 2

Contract Assertions	**Build Mode**	**Mode Value**
`contract_assert_opt(check_expression)`	"opt"	`opt`
`contract_assert_dbg(check_expression)`	"dbg"	`dbg`
`contract_assert_safe(check_expression)`	"safe"	`safe`

Each contract assertion form corresponds to a validating build mode, and to a *mode value* of the enumeration `contract_assertion_mode`, as defined in Table 2. [*Note:* The mode value is used when a contract violation is detected. — *end note*] A contract assertion is *active* only if its corresponding build mode, or a stronger build mode, is in effect at the point in the translation unit where the assertion appears.

A contract assertion is a `void` expression, treated syntactically as identical to a function call with one function argument. The effect, designated *E1*, of evaluating a contract assertion depends on (1) the build mode in effect, (2) the effect, designated *E2* , and (3) the value, if any, that *would result* from evaluating `bool(check_expression)` in the context where the contract assertion appears. [*Note:* Evaluation produces no resulting value when it throws an exception. — *end note*]

— Regardless of the build mode in effect, if effect *E2* includes side effects [intro.execution] that are *not* preceded by detection of a contract violation, then effect *E1* is *undefined*. [*Note:* Implementations are encouraged to report predictable side effects as translation errors. — *end note*]

— Otherwise, if the contract assertion is *not* active, and effect *E2* does not include yielding a value for the full expression, or if the value is `false`, then effect *E1* is *undefined*. [*Note:* Implementations are encouraged to use the implications of contract assertion check-expressions to help analyze, diagnose, and optimize programs where they appear. — *end note*]

— Otherwise, if the contract assertion is active, and effect *E2* does *not* include yielding a value for the full expression, then effect *E1* is, identically, effect *E2*.

— Otherwise, if the contract assertion is active, and effect *E2* includes yielding a value for the full expression, and the value is `false`, then effect *E1* is to detect a contract violation.

— Otherwise, the value of the contract assertion is `void()`.

[*Note:* A constructor may avoid violating preconditions of subobject constructors by evaluating their arguments only after enforcing its own preconditions, e.g. in a comma expression. More elaborate validation may be delegated to the constructor of an initial empty base class. — *end note*]

11.2.4 Contract Violation Info [**contract.violation.info**]

```
struct contract_violation_info
{
    contract_assert_mode mode;
    char const* expression_text;
    char const* filename;
    unsigned long line_number;
    // ...
};
```

When the implementation calls `handle_contract_violation`, members of its argument are initialized as described in Table 3. The contract assertion mentioned in the table is the one that detected the contract violation. The implementation, and future standards, may define and initialize additional members.

Table 3

Member	Value
`mode`	the mode value that corresponds to the contract assertion.
`expression_text`	a MBCS containing the phase 3 [lex.phases] source text of the argument to the contract assertion, with white space treated as described in [cpp.stringize].
`filename`	the value that `__FILE__` would have at the position in the translation unit where the contract assertion appears.
`line_number`	the value that `__LINE__` would have at the position in the translation unit where the contract assertion appears.

11.2.5 Build-mode flag [**contract.flag**]

A *build-mode flag* is a preprocessor symbol that is defined where, and only where, its corresponding build mode, as defined in Table 4, or any stronger build mode, is in effect. The effect of `#define` or `#undef` applied to any build-mode flag is undefined. Where a build-mode flag is defined, its value is 1. [*Note:* These flags might be used to stub out a helper function that is used only in *check-expression*s, or to gate unit-test cases. — *end note*]

Table 4

Validating Build Mode	Build-Mode Flag
“opt”	`contract_assert_build_mode_opt`
“dbg”	`contract_assert_build_mode_dbg`
“safe”	`contract_assert_build_mode_safe`

11.2.6 Contract-assertion unit test forms [**contract.tests**]

`<experimental/contract_assert>` defines two kinds of *contract-assertion unit test*: *contract-assert-fail*, and *contract-assert-pass*.

A contract-assertion unit test is a `bool` expression. It is treated syntactically as a function call with one function argument. Regardless of the build mode in effect, the argument shall be a valid expression in the context where the contract-assertion unit test appears, and convertible to `bool` in that context.

When a contract-assertion unit test is evaluated, then, *if and only if* it is active, its argument expression is evaluated exactly once in the context where the contract-assertion unit test appears, in a manner such that if a contract violation would otherwise be detected during the evaluation, the contract violation is instead *intercepted*, and an implementation-specific exception is thrown which, if it escapes the argument expression, is caught and absorbed. [*Note:* Interception of contract violations supersedes any contract violation handler that is installed by the program before *or during* evaluation of the test-expression. — *end note*]

`<experimental/contract_assert>` defines three contract-assert-fail unit test forms,

```
contract_assert_fail_opt(test_expression)
contract_assert_fail_dbg(test_expression)
contract_assert_fail_safe(test_expression)
```

and one alias,

```
contract_assert_fail(test_expression)
```

The alias provides an abbreviation for `contract_assert_fail_dbg`.

Each contract-assert-fail unit test form corresponds to a validating build mode and mode value as defined in Table 5. A contract-assert-fail unit test is *active* if and only

if its corresponding validating build mode, or a stronger build mode, is in effect. If the contract-assertion unit test is not active, or if, in evaluating *test_expression*, (1) a contract violation is intercepted, and (2) the `mode` member that *would have been* passed to `handle_contract_violation` in consequence of detecting this contract violation corresponds to the contract-assert-fail unit test's validating build mode, and (3) the exception thrown as a consequence of the interception escapes the argument expression, then the contract-assert-fail unit test is `true`; otherwise, it is `false`. [*Note:* These forms do not prevent any undefined behavior that would result from such evaluation. — *end note*]

Table 5

Contract-Assert-Fail Unit Test Forms	Validating Build Mode	Mode Value
`contract_assert_fail_opt(`*test_expression*`)`	"opt"	`opt`
`contract_assert_fail_dbg(`*test_expression*`)`	"dbg"	`dbg`
`contract_assert_fail_safe(`*test_expression*`)`	"safe"	`safe`

In addition, one *contract-assert-pass unit test* is defined:

```
contract_assert_pass(test_expression)
```

It is *active* in all build modes. If, in evaluating *test_expression*, a contract violation is intercepted, then the contract-assert-pass unit test is `false`; otherwise, `true`.

11.2.7 Contract violation handler functions [contract.handler]

11.2.7.1 Contract violation handler types [contract.handler.types]

```
using contract_violation_handler = void (*)(contract_violation_info const& info);
```

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

11.2.7.2 [Modifying Clause 17] Handler functions [handler.functions]

1 The C++ Library Fundamentals Technical Specification provides default versions of the following handler function **types** (Clause 18 [language.support]):

- `unexpected_handler`
- `terminate_handler`
- **`contract_violation_handler`**

2 A C++ program may install different handler functions during execution by supplying a pointer to a function defined in the program or the library as an argument to (respectively):

— `set_new_handler`

— `set_unexpected`

— `set_terminate`

— **`set_contract_violation_handler`**

3 A C++ program can get the pointer to a current handler function by calling one of the following functions (respectively):

— `get_new_handler`

— `get_unexpected`

— `get_terminate`

— **`get_contract_violation_handler`**

11.2.7.3 Contract violation handler manipulation [**contract.handler.manipulation**]

```
contract_violation_handler
set_contract_violation_handler(contract_violation_handler handler) noexcept;
```

Remark: The function indicated by `handler` shall not return normally to the caller, nor itself detect a contract violation. [*Note:* It may throw an exception. — *end note*]

Effects: Establishes its argument as the current global contract-violation handler. Passing a null pointer value re-establishes the default version of the contract violation handler. The behavior is undefined if `set_contract_violation` is called during evaluation [intro.multithread], in another thread, of a call to `handle_contract_violation`.

Returns: The value passed to the most recent previous call, or the default handler the first time that `set_contract_violation_handler` is called.

```
contract_violation_handler
get_contract_violation_handler() noexcept;
```

Returns: The value passed as the argument to the most recent call to the function `set_contract_violation` or, if that function has not yet been called, the default contract-violation handler. [*Note:* If the result is null, it indicates the default handler. – *end note*]

11.2.7.4 Contract violation handler invocation [**contract.handler.invocation**]

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

Remark: Called immediately by the implementation when any contract assertion detects a contract violation. [*Note:* It may also be called directly by a program. – *end note*]

Effects: Calls the currently installed thread-local contract-violation handler, if any; otherwise, calls the currently established global contract-violation handler, or the default contract-violation handler if `set_contract_violation_handler` has not yet been called.

Default behavior: The implementation's default global contract-violation handler calls `std::abort()`.

11.2.7.5 Thread-local contract-violation handler Installation

[contract.handler.guard]

```
class contract_violation_guard
{
public:
    explicit contract_violation_guard(contract_violation_handler handler);
    ~contract_violation_guard();

    contract_violation_guard(contract_violation_guard const&) = delete;
    contract_violation_guard(contract_violation_guard&&)      = delete;
    void operator=(contract_violation_guard const&)           = delete;
    void operator=(contract_violation_guard&&)                = delete;
};
```

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

```
explicit contract_violation_guard(contract_violation_handler handler);
```

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

```
~contract_violation_guard();
```

Effects: The thread-local contract-violation handler for the calling thread is set to the value it held before this object was constructed.

Remarks: The effect is defined only if called from the same thread that called the object constructor.

12 References

[1] The Bloomberg BDE Library open source distribution, https://github.com/bloomberg/bde