Doc No: N4253
Date: 2014-11-21
Authors: Nathan Myers (nmyers12@bloomberg.net)
John Lakos (jlakos@bloomberg.net)
Alexei Zakharov (alexeiz@gmail.com)
Alexander Beels (abeels@bloomberg.net)

Language Support for Contract Validation (Revision 9)

Abstract

With enough care we can build libraries that are essentially defect-free, but even the best library may fail catastrophically when misused. Contract validation, the practice of checking functions' preconditions when and where 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.

We do *not* pretend to propose a comprehensive solution to the contract validation problem. In particular, this proposal introduces no new syntax, nor anything requiring new object-file support. Very deliberately, nothing here would interfere with any future comprehensive solution, but it defines features that would be necessary parts of any such solution.

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

1 Document History

This proposal is based on N4135, based on N4075, based on N3997, based on N3963.

The changes since N4135 are simplification: eliminating the scoped violation handler, and decoupling the contract assertion and contract assertion unit test forms from the header declaring the violation handling apparatus. It retains N4135's non-dependence on preprocessor macros, and lack of ODR problems, but delimits the problems it is intended to address, and not to address.

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.

This proposal does *not* pretend to specify a complete solution to the contract validation problem. In particular, it omits features acknowledged as practical necessities for optimal support of static analysis tools. It does identify a minimal subset of features immediately useful for the purposes it does address, that would remain upward-compatible with any complete solution eventually proposed.

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. 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; 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.

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, if a compiler can determine that a call to an inline or template function passes values that would violate an expressed precondition, we would like it to report that as an error. Similarly, if the standard allows a violated assertion to be interpreted to imply that the program's runtime state is already undefined, the compiler can skip generating code that could be reached only 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.

As noted, this proposal does not pretend to a comprehensive solution for static error checking and optimization. Instead, it offers the closest approach to such a solution that is possible without introducing new syntax and new object-file annotations, and without interfering with any more complete solution.

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.

It is very deliberately not a goal of this design to enable features that would require new syntax, but it carefully avoids interfering any such features that may be added later.

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>`.

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.

We do not propose to change or integrate with `logic_error` or `<cassert>`. Users may choose to install a contract violation handler that throws `logic_error` where they deem appropriate.

8 Summary of Proposal for Standardization

We propose:

- *validating build modes* to give application developers control over how much contract validation is built into their programs, and over how to treat *contract violations* at compile time and at run time
- corresponding *contract assertions* for use in library and application code to express preconditions, and to detect and report 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* forms 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 “test” 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 is explicitly permitted to use any guidance it can infer from contract assertions it sees to do better error checking and code generation, even (indeed, especially) when no checking code ends up in the compiled program.

This proposal does not prescribe how programs composed from translation units built in different build modes behave. Implementers will meet the practical needs of their customers just as they do now for units built with varying optimization levels, calling conventions, and styles of debug annotations. Implicitly, a program that violates no contract assertions and evokes no undefined behavior will run identically regardless of the build mode (or, where supported, build modes) used.

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 "test" 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. Note that details of the argument passed to the handler may be changed to integrate with other proposals accepted.

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 libraries perform as intended.

9 Examples

9.1 *Check a contract precondition in* safe *and* test*, 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 "test" 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>

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

9.3 *Print a message and quit when a contract violation is detected*

Here, 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 exception is caught, and a diagnostic message emitted, before it exits normally.

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

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

int main()
{
    std::set_contract_violation_handler(
        [](std::contract_violation_info const& info) {
             std::cerr << "Detected a contract violation at “
                       << info.filename << “:” << info.line_number << “.\n";
             std::abort();
        });

    lib::unimplemented_function();  // boom
}
```

9.4 Verify that a function correctly asserts its preconditions

The *contract-assertion unit-test* forms verify, concisely, that contract assertions correctly detect violations:

```
#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 nearly trivial. If independent library authors were to do the same, their users would face a forest of handler mechanisms, each slightly different from the other. 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 can, 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>` declares functions and types to manage *contract violation handlers*.

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

Header <experimental/contract_assert> synopsis

```
 namespace std {
inline namespace experimental {
inline namespace fundamentals_v2 {

// [contract.assertions] types
enum class contract_assertion_mode { opt, test, 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);

}}}  // namespaces
```

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

At any point in a translation unit, one of four *build modes* described in Table 1 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, and which is in effect throughout the translation unit.

Table 1

Build Mode	Description
(none)	No contract preconditions are checked
“opt”	Only the least expensive contract preconditions are checked
“test”	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. [*Note:* Implementations are encouraged to select the “test” mode by default, by analogy to `<cassert>` and `NDEBUG`. — *end note*]

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

Three contract assertion forms are defined:

```
contract_assert_opt(check_expression)
contract_assert_test(check_expression)
contract_assert_safe(check_expression)
```

and one alias:

```
contract_assert(check_expression)
```

The alias is an abbreviation for `contract_assert_test`.

Table 2

Contract Assertions	**Build Mode**	**Mode Value**
`contract_assert_opt(`*`check_expression`*`)`	“opt”	`opt`
`contract_assert_test(`*`check_expression`*`)`	“test”	`test`
`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. 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 designated here as *`check_expression`*. The *`check_expression`* shall be an expression convertible to `bool` in the context where the contract assertion appears. When a contract assertion is evaluated, its effect is determined as follows:

— If the contract assertion is *not* active, then if evaluating `bool(`*`check-expression`*`)` in the context where the contract assertion appears *would* yield `false`, or such evaluation *would not* yield a value, then the effect of evaluating the contract assertion itself is *undefined*. [*Note:* Implementations are encouraged to use the implications of contract assertion check-expressions to help analyze, diagnose, and optimize programs, and to report predictable side effects of evaluation as errors. — *end note*]

— Otherwise, if the contract assertion *is* active, then the effect of evaluating it is identically that of evaluating `bool(`*`check-expression`*`)` in the context where the contract assertion appears, except that (1) it is *unspecified* which, if any, side effects [intro.execution] that would occur during said evaluation do, in fact, occur; and (2) if said evaluation would yield `false`, a contract violation is detected [contract.handler.violation], and `handle_contract_violation` is called immediately, passing as its argument a `contract_violation_info` object initialized as specified in Table 3. [*Note:* Implementations are encouraged to warn of side effects of evaluation detectable at translation time. — *end note*]

— Otherwise, the contract assertion has no effect.

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

[*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 information [**contract.violation.info**]

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

The argument to `handle_contract_violation`. The implementation, and future standards, may define and initialize additional members.

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

A *build-mode flag* is a preprocessor symbol that is defined 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`
“test”	`contract_assert_build_mode_test`
“safe”	`contract_assert_build_mode_safe`

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

Two kinds of *contract-assertion unit-test form* are defined: *contract-assert-fail*, and *contract-assert-pass*.

A contract-assertion unit-test form 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 form appears.

When a contract-assertion unit-test form 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 form 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*]

There are three contract-assert-fail unit-test forms:

```
contract_assert_fail_opt(test_expression)
contract_assert_fail_test(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_test`.

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 form 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 form is not active, or if, in evaluating *`test_expression`*, (1) a contract violation is intercepted, (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 form'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 form 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_test(`*`test_expression`*`)`	“test”	`test`
`contract_assert_fail_safe(`*`test_expression`*`)`	“safe”	`safe`

In addition, one *contract-assert-pass unit-test form* 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 form 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 contract-violation handler. Passing a null pointer value re-establishes the default version of the contract violation handler.

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 the currently established 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 contract-violation handler calls `std::abort()`.

12 References

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