Doc No: N4378
Date: 2015-02-08
Authors: John Lakos (jlakos@bloomberg.net)
Nathan Myers (nmyers12@bloomberg.net)
Alexei Zakharov (alexeiz@gmail.com)
Alexander Beels (abeels@bloomberg.net)

Language Support for Contract Assertions (Revision 10)

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 at compile time or in early testing, speeding development and making software more robust.

We propose here complete Working Paper text defining 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 these inferences in all phases of translation.

We do *not* pretend this is a comprehensive solution to "the contract validation problem", however that may be defined. In particular, this proposal introduces no new syntax. Very deliberately, nothing here would interfere with a more ambitious future solution, but it defines necessary parts of any such solution. We expect to see (and make) proposals that extend the usefulness of contract assertions as defined here.

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 and clarification in response to discussion and straw votes in Urbana. This still provides complete WP text defining run-time and compile-time effects, but drops secondary features and delimits the goals it attempts.

2 Introduction

Any library may fail catastrophically if misused. We make our libraries as easy to use and as hard to misuse as is practical, and we catch misuse at compile time wherever possible. 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 with small additions 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 translated code.

This proposal does *not* pretend to specify a complete solution to the contract validation problem. In particular, it omits any features that would need new syntax. It does identify a minimally complete subset of features immediately useful for the purposes it does address, and that can remain upward-compatible with any further contributions proposed. Future proposals can build upon facilities defined here.

3 Background

The `std::vector` member **`push_back`** may be called on any vector object, with no risk to program state. It offers a *wide* contract; no combination of arguments and well-defined prior state can evoke undefined behavior. Another member, **`pop_back`**, has the precondition that the vector instance not be empty. It offers a *narrow* contract; its effects are *undefined* unless its precondition is satisfied.

Any program that may violate such a precondition harbors a defect. Such violations can sometimes be caught by a compiler that understands the preconditions. Violations that remain can often be caught by runtime checking. Such checking always costs extra code space and run time; the costs are often small, but can be very large. Catching a violation might be worth any expense; yet, where there is no defect, every cycle and byte spent 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 contract violation that gives us latitude to choose whether to avoid the runtime expense of checks, or to add code to detect a violation at compile time or at runtime, 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 attaining core software engineering goals.

4 Motivation

4.1 The Value of Checking

Library developers prefer to check for bad usage where they can—catching users' mistakes early prevents bugs and spurious bug reports—but the consequences on performance and interface complexity often forbid it. Whereas library development costs can often be amortized over many downstream uses, applications typically support only their own development, so application-level testing is notoriously limited. Libraries instrumented to validate usage contracts amplify the effectiveness of whatever testing is done, anywhere a defect leads to 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 to improve the product even during final 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 script can step to the next test. Similarly, during early development, it would best perform every check possible; in beta testing, do only sanity checks; and when performance tuning, avoid all checking.

These are not choices that library writers can reasonably be expected to provide for as they annotate their code with preconditions to help prevent misuse.

4.3 Compiler Hinting

Assertions' usefulness is not limited to testing. When compilers may infer programmers' true intent, and the bounds on a program's defined 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 that determines that a particular call would violate an expressed precondition could report it as an error, but only if permitted by the standard. Furthermore, if our compiler identifies a code path that would lead to such a violation, it may elide code for that path, and perhaps issue a warning. Finally, a compiler might be able to better optimize code following an expressed precondition under the assumption that the condition was met. Dead-code elision can propagate back up the call chain; any path certain to reach the elided code can itself be elided; likewise, for any that would run only in case of a violation. Eliding dead code reduces instruction-cache pressure, speeding execution of the live code that remains.

We cannot expect a compiler to simultaneously optimize code under the assumption that all preconditions are met, and also emit code to check the preconditions. Nor can we expect it to choose to do one or the other on its own. These are, necessarily, conflicting goals that we must decide according to our immediate needs.

As noted, this proposal does not pretend to be a comprehensive solution for all static error checking and optimization goals. 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 later additions, as a necessary first step toward a comprehensive solution.

4.4 Design Goals

In short: Library authors need to easily code contract-validation assertions, concisely expressing the cost to check them vs. the useful work a function does.

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

Implementations need the latitude to use the implications of contract-validation assertions to identify errors at translation time, and to guide optimization when runtime validation is not needed.

It is deliberately not a goal of this design to define features that would require new syntax. We have carefully avoided interfering with 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>` and its 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

For a minimal conforming implementation, this proposal requires no new core language features, and 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 their version of the standard library. Similarly, compiler implementers would be asked to use contract assertions to help improve static error detection and object-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, or to replace regular assertions with contract assertions.

8 Summary of Proposal for Standardization

We propose:

- three *contract assertion* forms for use in library and application code to express requirements, and to detect and report violations
- compile-time configurable *assertion levels* to determine which contract assertions to check at runtime and which to take as given
- a common, configurable c*ontract-violation handler* to give application developers precise control over what happens when a violation is detected anywhere in the program

8.1 Contract Assertions

We introduce three source code forms called *contract assertions.* Each expresses a contract requirement, analogously to the traditional `assert` macro. Library programmers will write disproportionately costly checks using the “max” contract assertion, checks that do not violate performance requirements using the “on” assertion, and very inexpensive or critical checks using the “min” 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 and report usage errors, and to produce faster, more compact object code.

8.2 Assertion Levels

Depending on circumstances, you want each assertion to be either verified at runtime or assumed as given. When compiling a translation unit, you can specify that, of assertions encountered during translation, none are to be checked at runtime, or only the “min” assertions, or only the “min” and “on” assertions, or all assertions. For those assertions not to be checked at runtime, the compiler may assume they are true and optimize accordingly.

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 are designed to be easily enhanced to integrate with other proposals accepted.

9 Examples

9.1 Check a contract precondition only at assertion levels “max” and “on”

A **`strlen`**-like function, **`c_string_length`**, has a precondition that its argument `string` must not be null. The form `contract_assert` checks the precondition at “max” and “on”, but not “min” or “off” assertion levels:

```
#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 during computation

The **binary_search** function below is specified to run in *O(log n)* time, given a sorted table to search in. To validate its requirement, as written, would add *O(n)* time, violating its performance specification. Partial, incremental checks within the loop are almost as effective, but take, cumulatively, only *O(log n)* time.

There are several points to notice here. First, using `contract_assert_max` for the literal requirement avoids incurring its incommensurate runtime cost in normal operation. Second, actually expressing the literal requirement, even though we do not usually expect to execute it, gives the compiler a much simpler expression of conditions that it may assume while optimizing than it could extract from the other assertions or the code. Finally, the incremental checks used here could not practically be placed anywhere other than deeply embedded in the body of the function:

```
#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_max(std::is_sorted(table, table + size));      // O(n)
    while (size != 0) {
        std::size_t step = size / 2;
        int candidate = table[step];
        contract_assert(table[0] <= candidate);                    //
        contract_assert(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, the **binary_search** function defined above is made to assert a contract violation. When the program calls the function in violation of its requirement, it emits a diagnostic message and terminates:

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

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();
        });

    binary_search(nullptr, 10, 0);  // boom
}
```

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, though, 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.

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 at *all assertion levels*, most particularly those in which check-expressions do not end up expressed in object code.

Notably, declarative mechanisms could, without compromise, be added as *pure extensions* to what is proposed here, re-using most of its machinery. This simple proposal could be approved, implemented and in production use while details of more ambitious designs are still being worked out, and would remain useful thereafter, both directly and as a basis for future work. While syntax might be added to support annotations outside function bodies, consider the incremental checking seen in example 9.2 above; it illustrates an important use case that any purely declarative approach alone would not support.

Support for static analysis tools that would be enabled by expressed preconditions need not depend on finding assertions in header files. A dedicated static checking tool might be shown all the sources to a program. Or, a compiler may annotate object files and library archives with the contract assertions it encounters during translation. When the language gets module support, modules might be a better place to keep such details. Header files are not the only possible source of information for a static-checking tool, and our reasonable desire to keep header-file declarations uncluttered need not limit the power of such tools.

Further discussion may be found in reference [2] N4379, “FAQ about Contract Assertions”.

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, and possibly, but not necessarily, undefined.

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, assertion-level flags, contract violation handlers, and the names defined in `<experimental/contract_assert>`.

11.2.2 Header <experimental/contract_assert> synopsis

```
namespace std {
```

```
inline namespace experimental {
inline namespace fundamentals_v2 {

// [contract.assertions] types
enum class contract_assertion_level { min, on, max };

// [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
```

11.2.3 Contract assertion levels [**contract.assertion.levels**]

A translation unit may be translated at any of the four *assertion levels* described in Table 1. Implementations shall provide a means, as part of initiating translation on each translation unit and outside of the program text, that any of the assertion levels listed in Table 1 may be selected.

Table 1

Assertion level	Description
off	No contract preconditions are checked
min	Only `contract_assert_min` assertions are checked
on	Only `contract_assert_min` and `contract_assert_on` assertions are checked
max	All contract assertions are checked

Each successive assertion level listed in Table 1 is said to be *stronger* than the preceding level or levels. The final three are called *validating* assertion levels.

If no assertion level is specifically selected at translation initiation, then an *implementation-specified* choice of level is used. [*Note:* Implementations are

encouraged to use the "on" level by default, by analogy to `<cassert>` and `NDEBUG`. — *end note*]

11.2.4 Contract assertions [**contract.assertions**]

Three contract assertion forms are defined:

```
contract_assert_min(check_expression)
contract_assert_on(check_expression)
contract_assert_max(check_expression)
```

and one alias:

```
contract_assert(check_expression)
```

The alias is an abbreviation for `contract_assert_on`.

Table 2

Contract Assertions	Assertion level	value
`contract_assert_min(`*`check_expression`*`)`	min	`min`
`contract_assert_on(`*`check_expression`*`)`	on	`on`
`contract_assert_max(`*`check_expression`*`)`	max	`max`

Each contract assertion form corresponds to an assertion level, and to one value of the enumeration `contract_assertion_level`, as defined in Table 2. A contract assertion is *active* only where translation is performed at its level or a stronger level.

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. If evaluating `bool(`*`check-expression`*`)` in the context where the contract assertion appears would result in side effects [intro.execution] during or as a consequence of said evaluation, the contract assertion is ill-formed (no diagnostic required). [*Note:* Implementations are encouraged to report an error if side effects of evaluating `check_expression` are detectable at translation time. — *end note*] 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 either *would* yield `false`, or *would not* yield a value, then the effect of evaluating the contract assertion itself is *undefined*; otherwise, the contract assertion has no effect. [*Note:* Implementations are encouraged to use the implications of contract assertion check-expressions to

help analyze and optimize programs, and to report predictable violations and side effects of evaluation as errors. — *end note*]

— If the contract assertion *is* active, the effect of evaluating it is identically that of evaluating `bool(`*check-expression*`)` in the context where the contract assertion appears, except that 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.

Table 3

Member	**Value**
`level`	the assertion level value corresponding 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 might be delegated to the constructor of an initial empty base class. — *end note*]

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

```
struct contract_violation_info
{
    contract_assert_level level;
    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.

11.2.6 Assertion-level flag [**contract.flag**]

An *assertion-level flag* is a preprocessor symbol that is defined when translation is performed at its corresponding assertion level, as related in Table 4, or any stronger level. The effect of `#define` or `#undef` applied to any assertion-level flag is undefined. When an assertion-level 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

Assertion Level	**Assertion-Level flag**
min	`contract_assertion_level_min`
on	`contract_assertion_level_on`
max	`contract_assertion_level_max`

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

[2] N4379: FAQ about Contract Assertions