Usage Experience for Contracts with BDE

Document #:	P3336R0
Date:	2024-06-23
Project:	Programming Language C++
Audience:	EWG (Evolution), SG21 (Contracts)

Reply-to: Joshua Berne <jberne4@bloomberg.net>

Abstract

Initial implementations of the Contracts facility as specified in [P2900R7] are becoming available. In this paper, we present the results of making use of one of those implementations on our internal codebase — which is heavily instrumented with contract assertions — with particular regard to how `const`-ification and repetition worked in practice.

Contents

Revision History

Revision 0

- Original version of the paper for discussion during an SG21 telecon

1 Introduction

The Contracts MVP produced by SG21, i.e., [P2900R7], provides a number of features that differentiate its semantics from those of classic macro-based assertion facilities (such as `<cassert>`). In particular, some of these features are designed to hinder writing contract-assertion predicates that alter the correctness of a program.

- Any use of an automatic variable or `*this` within a contract-assertion predicate is treated as `const`, preventing inadvertent mutation from within a contract assertion.
- The expression used with a contract assertion is a *conditional-expression*, which prevents the use of an *assignment expression* in an assertion, both reducing the chance of unintended modifications and the common typo of using `=` instead of `==`.
- Because contract assertions may be evaluated multiple times, platforms can provide a mechanism to identify destructive contract assertions.

At present, an in-progress implementation of [P2900R7] implements the first two bullets above — `const`-ification and the grammar requirements of contract assertions. The third feature, arbitrary repetition, can be easily replicated in a macro-based facility with a `for` loop.

To test the usability of the features above, we changed an existing macro-based contract-checking facility — `bsls_assert` from the BDE libraries[1] — to use `contract_assert`. In the rest of this paper, we describe the specifics of this change and our experiences building some large libraries that make use of `contract_assert`.

2 Target Libraries

The contract-checking facility we sought to modify is `bsls_assert`,[2] which provides a family of macros (alongside `bsls_review`) that are used for contract checking. The `bsls_assert` component provides a suite of assertion macros that parallel the functionality of `contract_assert` in [P2900R7]. Each macro, based on the configuration macros defined in the translation unit, expands in a different form based on whether the assertion should be *enforced*, *observed*, *ignored*, or *assumed*.

When *enforced*, each macro expands to the following:

```
#define BSLS_ASSERT_ASSERT_IMP(X,LVL) do {                                    \
        if (BSLS_PERFORMANCEHINT_PREDICT_UNLIKELY(!(X))) {                    \
            BSLS_PERFORMANCEHINT_UNLIKELY_HINT;                               \
            BloombergLP::bsls::AssertViolation violation(                     \
                                                    #X,                       \
```

[1] See [bde14].
[2] See bsls_assert.

```
                                                    BSLS_ASSERTIMPUTIL_FILE, \
                                                    BSLS_ASSERTIMPUTIL_LINE, \
                                                    LVL);                    \
            BloombergLP::bsls::Assert::invokeHandler(violation);             \
        }                                                                    \
    } while (false)
```

Like `contract_assert`, the above code evaluates an expression and then, if that expression has a value of `false`, invokes a contract-violation handler.

To test BDE and its dependent libraries with `contract_assert`, detection logic for the presence of contracts, based on the feature-test macro `__cpp_contracts`, was added to expand the above macro differently when `contract_assert` is available:

```
// ...
#ifdef __cpp_contracts
#define BSLS_ASSERT_ASSERT_IMP(X,LVL) contract_assert( X )
#endif
// ...
```

In a separate test, we changed the enforced assertion macro to use a loop instead of a single `if` statement:

```
#define BSLS_ASSERT_ASSERT_IMP(X,LVL) do {                                   \
    for ( int bslsAssertRepeatCount = 0;                                     \
          bslsAssertRepeatCount < BSLS_ASSERT_REPEAT_COUNT;                  \
          ++bslsAssertRepeatCount ) {                                        \
        if (BSLS_PERFORMANCEHINT_PREDICT_UNLIKELY(!(X))) {                   \
            BSLS_PERFORMANCEHINT_UNLIKELY_HINT;                              \
            BloombergLP::bsls::AssertViolation violation(                    \
                                                    #X,                      \
                                                    BSLS_ASSERTIMPUTIL_FILE, \
                                                    BSLS_ASSERTIMPUTIL_LINE, \
                                                    LVL);                    \
            BloombergLP::bsls::Assert::invokeHandler(violation);             \
            break;                                                           \
        }                                                                    \
        } } while (false)
```

This introduced a new configuration macro, `BSLS_ASSERT_REPEAT_COUNT`, which allowed us to run the full suite of unit tests with assertion-predicate evaluation repeated an arbitrary number of times.

With these changes to `bsls_assert`, we then compiled and ran tests for BDE and four higher-level internal libraries (see Table 1), all of which use `bsls_assert` extensively.

- Components are the number of `.h` and `.cpp` pairs, along with their associated test drivers.
- Lines and test lines are a count of lines of code, with a percentage of those lines that are not comments or blank to give an approximate measure of the size of the library, separated between production code (in the `.h` and `.cpp` files) and test driver code.
- Asserts and test asserts are the number of uses of the `bsls_assert` macros, with a percentage of nonempty, noncomment lines of code that are assertions, again split between production

code and test driver code. This percentage gives an indication of how consistently assertions are used in each individual library.

- Issues is the number of distinct issues to be addressed due to the switch to using `contract_assert` in the implementation of `bsls_assert`.

Table 1: Tested Libraries

Library	Components	Lines		Test Lines		Assertions		Test Assertions		Issues
BDE	3330	1.32M	(70.52%)	2.14M	(76.02%)	7749	(0.83%)	4743	(0.29%)	7
Library `#1`	1814	368.30K	(62.71%)	986.45K	(79.18%)	6307	(2.73%)	981	(0.13%)	4
Library `#2`	165	45.49K	(74.68%)	64.77K	(75.86%)	231	(0.68%)	240	(0.49%)	0
Library `#3`	1084	352.58K	(79.90%)	116.20K	(88.73%)	1844	(0.65%)	138	(0.13%)	2
Library `#4`	240	86.61K	(60.09%)	17.79K	(85.48%)	1156	(2.22%)	5	(0.03%)	1

3 Results

Two different experiments were performed: one to see the impact of `const`-ification on a few large libraries and another to verify that repeating assertions was sound for the BDE test-driver suite.

3.1 `const`-ification of assertions in a wide codebase

The first experiment was to compile all the above libraries with the experimental branch of `gcc` that contains the ongoing implementation of [P2900R7].[3]

All unit tests for these libraries passed, with the exception of tests that depend on the ability to specify the *observe* semantic in code and tests that failed due to code not yet ready to support C++23, the latest version of `gcc`, or the latest version of `libstdc++`, .

To compile these libraries after the switch to `contract_assert`, a variety of changes had to be made, and these modifications fall into a few broad categories. All names and specifics in the examples below have been changed both for security and to capture, with brevity, the essential aspects of the issues that were encountered.

- **APIs not supporting `const` usage**

  Due to `const`-ification within contract predicates, any use of an API that is not `const` correct will fail to compile.

  - Two functions (`size` and `empty`) on one container-like type defined in a test driver in BDE were not `const` qualified. Three member functions in Library `#1` were not `const` qualified. None of these functions actually modified any state, and adding `const` fixed the (existing) issue in all of them.

  - Eleven assertions in Library `#3` made use of a non-`const` base-class function that returned `enable_shared_from_this()`. This issue was by far the most involved to solve, requiring the addition of a `const`-qualified overload (that returned `shared_ptr<const T>` instead of `shared_ptr<T>`) to the base class that provided the original non-`const` function. With

[3] This branch is primarily a combination of the original C++20 Contracts implementation by Andrew Sutton, with work to support the current syntax and semantics implemented by Ville Voutilainen and Nina Ranns.

that addition to support `const` correctness, all uses of the function still compiled, and tests passed as expected.

- **Nonmodifying, non-`const` usage** Some predicates did not modify any essential program state but still needed to make non-`const` use of local variables.

  – Four assertions in BDE used, as part of their predicates, functions that had output parameters whose value was discarded. These functions included string formatting and serialization functions that performed a trial serialization and whose output status was then tested for validity. Another example was from a utility that produced human-readable error messages into an output parameter that was subsequently discarded. In all cases, these values were otherwise unused and conditionally compiled, for example:

```
int validate(bsl::ostream& errorStream) const;
  // Validate this object's state.  Return 0 if valid and a nonzero
  // value otherwise.  If not valid, populate the specified errorStream with
  // a descriptive message.
void f()
{
#ifdef BSLS_ASSERT_IS_USED
  bsl::osringstream oss;
  BSLS_ASSERT(0 == validate(oss));
#endif
}
```

  As we were fixing these issues, we saw the ease of making errors when spelling out the correct type of the variable being referenced. To rectify this easy mistake and to reduce the chance of misusing the needed `const_cast` operators, a simple utility macro was added that removes the `const` that might have been applied to an id expression due to `const`-ification:

```
#define BSLS_ASSERT_UNCONST(X) const_cast<decltype(X)&>(X)
```

  A tool provided by the language itself for this purpose could certainly be more accurate and give significantly better diagnostics if misused.

  – One assertion in Library `#1` used a `dynamic_cast` on the address of a reference parameter to validate the precondition that a particular concrete type was passed to the function in question via a reference to its base. This predicate needed to be changed from

```
void function(T& arg)
{
  BSLS_ASSERT(0 != dynamic_cast<T*>(&arg))
}
```

  to

```
void function(T& arg)
{
  BSLS_ASSERT(0 != dynamic_cast<const T*>(&arg));
}
```

While the original assertion was not an error, it failed to compile because a `dynamic_cast` may not remove a `const`. The updated predicate compiles with or without `const`-ification and tests the same condition.

- **Intentional Failures**

  Occasionally, tests of the contract-checking facility itself are written with the intent of failing, and modifying a local variable is one way to accomplish this result.

  – One contract assertion in BDE was of this form:

  ```
  BSLS_ASSERT(x = x += y)
  ```

  This assertion was intended to fail as part of testing the facility that integrates contract checking with fuzz testing. This assertion was easily changed to an alternate expression that failed in a similar fashion.

- **Destructive Predicates**

  Certain predicates would, if they were not consistently evaluated with the same semantic, cause themselves to no longer produce the correct result.

  – One contract assertion in BDE was used within another macro to verify that all elements of an enumeration were in sequential order when processed:

  ```
  int ii = 0;  (void)ii;
  #define CHECK_ARG_COUNT(x,id) \
    BSLS_ASSERT(ii++ == id);    \
    checkCount(x,id);

  CHECK_ARG_COUNT(0,0);
  CHECK_ARG_COUNT(0,e_ENUM_VAL_1);
  CHECK_ARG_COUNT(1,e_ENUM_VAL_2);
  CHECK_ARG_COUNT(1,e_ENUM_VAL_3);
  CHECK_ARG_COUNT(0,e_ENUM_VAL_4);
  ```

  Due to the nesting of macros within macros, properly transforming this assertion to separate modification of assertion-supporting state from the macro was non-trivial but was accomplished with the `bsls_assert` equivalent of `NDEBUG` — `BSLS_ASSERT_IS_USED`:

  ```
  #ifdef BSLS_ASSERT_IS_USED
    int ii = 0;
    #define CHECK_ARG_COUNT(x,id) \
      BSLS_ASSERT(ii == id);      \
      ii++;                       \
      checkCount(x,id);
  #else
    #define CHECK_ARG_COUNT(x,id) checkCount(x,id);
  #endif
  ```

  Making this modification removed any future risk of using `ii` in the code below it when assertions are not enabled and the variable's value is not maintained properly.

- **Bugs**

  Due to the ease of mistakenly writing assertions that modify state with the assignment operator when an equality comparison was intended, both the improved grammar and `const`-ification prevent such slips.

  – One file in Library `#3` contained 6 instances of assertions of this form:

  ```
  BSLS_ASSERT(x->y = enum_value)
  ```

  All of these assertions were obvious bugs, mistakenly using `=` instead of `==`, that were promptly fixed[4] to use equality comparison (`==`) instead of assignment.

  – One assertion in Library `#4` invoked a function that did nothing and then returned an error code if the target object was already in the correct state, asserting that this error code was returned. This usage is highly questionable, and this design decision will be revisited because the behavior when the object is *not* in the correct state is highly surprising (and vastly different from the behavior when the assertion macro is disabled).

Altogether, the effort to perform the migration from `bsls_assert` to `contract_assert` was minuscule (a few minutes of actual work), a small fraction of the many hours required to just compile an experimental branch of GCC on a relatively underpowered virtual machine.

Table 2: Summary of Issues

Library	Missing `const` Support	Nonmodifying Usage	Intentional Failures	Destructive	Bugs	Total Issues
BDE	1	4	1	1	0	7
Library `#1`	3	1	0	0	0	4
Library `#2`	0	0	0	0	0	0
Library `#3`	1	0	0	0	6	7
Library `#4`	0	0	0	0	1	1

3.2 Repeating assertions in BDE

The second test performed was to compile BDE with assertion macros configured to repeat the evaluation of assertion predicates 64 times. This experiment was performed with BDE's current production compiler and is independent of the implementation work on [P2900R7] in `gcc`.

Running this experiment allowed us to make several observations.

- To improve quality of runtime behavior and avoid repeating violations once they have been reported once, we decided that, when repeating evaluations, repetitions would stop after a violation was detected even if that violation was only *observed*. This decision had no impact on tests or on the code as written but is worth considering for compilers providing this feature.
- Two test drivers counted operations performed by the code they were testing, primarily to detect that certain synchronization primitives did only exactly what was asked of them (with

[4]Note that none of these bugs would happen if the comparisons were consistently putting the constant — the enumerator in this case — on the left side of the comparison operator rather than the right.

instrumented underlying implementations to facilitate the counting). In both cases, existing code accounted for the evaluations within assertion macros, and in both cases, expected operations needed to be multiplied by the number of expected repetitions.

Any testing strategy that required this level of detail or even that depended on whether contract-assertion predicates were evaluated would need to be applied only when the testing code is able to implement the appropriate logic to determine how many evaluations of contract-assertion predicates there will be.

- The assertion that was intended to fail as part of testing *fuzz testing* utilities still failed on the first repetition and was not detected as problematic with the repeated evaluation strategy.
- The assertion that was modifying a local counter failed spectacularly on repeated evaluations and will be fixed.

The development effort to fix the above issues was, overall, small, especially compared to the time required to perform a few compilation cycles of the entire BDE library (again as a background task on an underpowered virtual machine).

4 Conclusion

Up to this point, using `const`-ification and repetition to improve the correctness of contract assertions has been primarily a theoretical exercise guided by our own estimations of how such features would apply to real-world code. With an implementation in hand, we have now applied these features to millions of lines of code that have a 20-year history of making regular use of assertions. From this experiment, we can take away some important lessons.

- The vast majority of assertions work correctly with no changes needed.
- Not a single assertion in our tested libraries behaved incorrectly due to the potential change in overload resolution resulting from `const`-ification.
- Some real bugs and some questionable designs were uncovered by this experiment, and their discovery has already led to improvements in the quality of our libraries.

As implementations of the Contracts facility evolve and more features are added in the future to support the full range of functionality of `bsls_assert`, we will repeat these experiments to further validate that the facility we are proposing for C++ is one that will prove useful in large-scale real-world environments.

Acknowledgments

Thanks to Andrew Sutton and Jason Merrill for the effort that went into adding support for C++20 Contracts to `gcc` and to Ville Voutilainen and Nina Ranns for implementing [P2900R7] as an iteration on that foundation.

Thanks to Mungo Gill, Rostislav Khlebnikov, John Lakos, and Lori Hughes for valuable feedback on this paper.

Bibliography

[bde14] "Basic Development Environment". Bloomberg
https://github.com/bloomberg/bde/

[P2900R7] Joshua Berne, Timur Doumler, and Andrzej Krzemieński, "Contracts for C++", 2024
http://wg21.link/P2900R7