Minimizing Contracts

Document #:	P1607R1
Date:	2019-07-23
Project:	Programming Language C++
Audience:	EWG

Reply-to:	Joshua Berne <jberne4@bloomberg.net>
	Jeff Snyder <jeff-isocpp@caffeinated.me.uk>
	Ryan McDougall <mcdougall.ryan@gmail.com>

Abstract

Contracts in C++ 20, how they behave, and how they should be used are currently an open question. The existing proposed *contract levels* of *default*, *audit*, and *axiom* are the result of a great deal of compromise and are not an ideal solution for many of the large institutions hoping to make use of contracts in the language. The currently proposed semantics — choosing between undefined behavior and checking, with a single global flag to control continuation after checking — are equally problematic for use at scale. This paper proposes two solutions to that problem, one involving stripping the contracts feature to a bare minimum to avoid polluting the design space for the future, and one involving adding to that the explicit semantics of [P1429R1] as building blocks for experimentation.

Contents

1 Revision history

- R1 – Wording, all reviewed by core except for specifics about behavior of the assume semantic, which required decisions from EWG on desired behavior.
- R0 – Initial proposal with references to P1429R0

2 Minimal Solution

> I am a great fan of the incremental approach - getting a minimal change in place and then improving it based on feedback. I consider that engineering as opposed to the ideal of getting a change perfect in advance - which I consider naive and at odds with reality.
>
> Bjarne Stroustrup

We propose taking the following parts of the currently proposed solution away from the existing contracts proposal:

- All *contract levels* other than *default* (including any syntax for specifying default).
- Undefined behavior when a contract is not evaluated.
- All values for *build level* other than *off* and *default*
- *continuation mode*, with evaluated contracts never continuing if they fail.

This leaves the following functionality:

- Contract checks can be specified using `[[expects]]`, `[[ensures]]`, and `[[assert]]` (or `[[pre]]`, `[[post]]`, and `[[assert]]` if [P1344R0] is adopted).
- Contracts can be *on*. The predicates will be evaluated and result in an invocation of the *violation handler* if they evaluate to `false`, `std::abort` will be called if the violation handler returns normally.
- The *violation handler* is still establishable in an implementation-defined way.
- Contracts can be *off* and will not be evaluated, but still syntactically checked.

3 Additional Building Blocks

The specific behaviors that are useful for contracts have been inherent in the current form of the contract proposals since [P0542R5]. In 2017, Lisa Lippincott published [P0681R0] with a very thorough analysis of many possible semantics that might be useful for contract checks. [P1429R1] narrowed that focus down to four semantics and proposed allowing explicit use of those semantics by name within contract attributes. We propose adding in four *identifiers with special meaning* and the associated definitions of the semantics for experimenting with different contract behaviors before cementing a higher-level set of functionality into the language on top of those behaviors.

This entails adding the following:

- Four *identifiers with special meaning* that can be placed in a *contract-attribute-specifier* to translate that attribute with a specific semantic — **`ignore`**, **`assume`**, **`check_never_continue`**, and **`check_maybe_continue`**.
- The definitions of those semantics, wording for which is available in [P1429R1].
- Expand the *build mode* to allow choosing any of the defined semantics for contract attributes with no explicit semantic.

3.1 Example Usage 1: audit

This will allow users who wish to get behavior like the currently proposed *audit* to use a macro like this:

```
#if defined(BUILD_LEVEL_AUDIT) && defined(CONTINUATION_MODE)
  #define AUDIT_SEMANTIC check_maybe_continue
#elif defined(BUILD_LEVEL_AUDIT)
  #define AUDIT_SEMANTIC check_never_continue
#else
  #define AUDIT_SEMANTIC ignore
#endif
```

Then the following code with a precondition as might be currently specified:

```
T* binsearch(T*begin, T*end, const T&val)
  [[expects audit : is_sorted(begin,end) ]]
```

becomes this:

```
T* binsearch(T*begin, T*end, const T&val)
  [[expects AUDIT_SEMANTIC : is_sorted(begin,end) ]]
```

With control of the behavior based on using `-DBUILD_LEVEL_AUDIT` when compiling.

Similar macros and control macros for any scheme that has been proposed could then easily be built by any enterprise that wishes to use facilities like that, or different, simpler or more complicated facilities.

Once more widespread experience with these systems exists we hope to see a clearer consensus on what to standardize on top of the building blocks proposed here.

3.2 Example Usage 2: assuming

Similarly, users who wish to have a checkable assume could build macro structures like this:

```
#if defined CHECK_SAFER_ASSUME
  #define SAFER_ASSUME(P) [[assert check_never_continue : P]]
#else
  #define SAFER_ASSUME(P) [[assert assume : P]]
#endif
```

This provides a way to target specific points where introduced assumptions help performance, but for testing and diagnosing any issues that might be related to these cases one can switch these assumptions into enforced checks.

4 Voting

On July 16th, 2019, EWG discussed conntracts and had a number of votes. Relevant to this paper were the following:

P1607R0 – All those in favor of discussing it

SF	F	N	A	SA
7	12	10	5	4

Take away build levels and continuation?

SF	F	N	A	SA
10	18	8	2	7

Add literal semantics

SF	F	N	A	SA
15	13	5	3	10

Add literal semantics

SF	F	N	A	SA
15	13	5	3	10

In addition, regarding P1769R0 there was the following vote:

Adopt P1769R0 as presented?

SF	F	N	A	SA
16	20	8	0	2

Together, these poles indicate the following (based the acceptance of this paper's first draft and P1769R0):

- Remove *contract levels*.
- Add *literal semantics* as per P1429 (primarily now referencing P1429R2).
- Allow a single control for which semantic a contract with no literal semantic gets. (This has been labelled *implicit contract behavior* during core wording).
- Do not specify what value *implicit contract semantic* has if it is not specified – which is P1769R0's interpretation within the context of this paper.

5 Formal Wording

In [**lex.name**], remove from table [**tab:lex.name.special**] `audit` and `axiom`, and add `assume`, `ignore`, `inform`, and `enforce`.

In [**basic.def.odr**] Paragraph 12 item 12.6:

- ...
- if D invokes a function with a precondition, or is a function that contains an assertion or has a contract condition (9.11.4), it is implementation-defined under which conditions all definitions of **D** shall be translated using the same ~~build level and violation continuation mode~~ implicit contract behavior; and
- ...

In **[dcl.attr.contract.syn]** paragraph 1 the following is changed:

contract-attribute-specifier:
`[ [ expects` ~~*contract-level*$_{opt}$~~ *contract-behavior*$_{opt}$ `:` *conditional-expression* `] ]`
`[ [ ensures` ~~*contract-level*$_{opt}$~~ *contract-behavior*$_{opt}$ *identifier*$_{opt}$ `:` *conditional-expression* `] ]`
`[ [ assert` ~~*contract-level*$_{opt}$~~ *contract-behavior*$_{opt}$`:` *conditional-expression* `] ]`

contract-level:
`default`
`audit`
`axiom`

contract-behavior:
`assume`
`ignore`
`inform`
`enforce`

An ambiguity between a ~~*contract-level*~~ *contract-behavior* and an *identifier* is resolved in favor of ~~*contract-level*~~ *contract-behavior* .

Change **[dcl.attr.contract.syn]** paragraph 6 as follows:

The only side effects of a predicate of a checked contract that are allowed in a contract-attribute-specifier are modifications of non-volatile objects whose lifetime began and ended within the evaluation of the predicate. An evaluation of a predicate that exits via an exception invokes the function std::terminate (13.5.1). The behavior of any other side effect is undefined. [*Example:*

```
void push(int x, queue & q)
  [[expects enforce: !q.full()]]
  [[ensures enforce: !q.empty()]]
{
  /* ... */
  [[assert: q.is_valid()]];
  /* ... */
}

int min = -42;
constexpr int max = 42;

constexpr int g(int x)
  [[expects enforce: min <= x]]                    // error
```

```
  [[expects enforce: x < max]]                          // OK
{
  /* ... */
  [[assert enforce: 2*x < max]];
  [[assert enforce: ++min > 0]];                        // undefined behavior
  /* ... */
}
```

— end example]

[dcl.attr.contract.cond] paragraph 2 gets the following change:

> Two lists of contract conditions are the same if they consist of the same contract conditions in the same order. Two contract conditions are the same if their ~~contract levels~~ *contract-behaviors* are the same, or both are absent and their predicates are the same. Two predicates contained in *contract-attribute-specifier*s are the same if they would satisfy the one-definition rule were they to appear in function definitions, except for renaming of parameters, return value identifiers (if any), and template parameters.

In **[dcl.attr.contract.cond]** paragraph 7 update the example as follows:

> [*Example:*
>
> ```
> int f(int x)
>   [[ensures enforce r: r == x]]
> {
>   return ++x;                     // undefined behavior
> }
>
> int g(int * p)
>   [[ensures enforce r: p != nullptr]]
> {
>   *p = 42;                        // OK, p is not modified
> }
>
> int h(int x)
>   [[ensures enforce r: r == x]]
> {
>   potentially_modify(x);          // undefined behavior if x is modified
>   return x;
> }
> ```
>
> *— end example*]

[dcl.attr.contract.check] gets the following changes:

> If the *contract-level* of a *contract-attribute-specifier* is absent, it is assumed to be `default`. [*Note:* A `default` *contract-level* is expected to be used for those contracts where the cost of run-time checking is assumed to be small (or at least not expensive) compared to the cost of executing the function. An `audit` *contract-level* is expected to be used for those contracts where the cost of run-time checking is assumed to be large (or at least significant) compared to the cost of executing the function. An `axiom` *contract-level* is

expected to be used for those contracts that are formal comments and are not evaluated at run-time. — *end note*]

The contract behavior of a *contract-attribute-specifier* is one of `ignore`, `assume`, `enforce`, and `inform` as specified by the *contract-behavior*. If the *contract-behavior* is absent, the implicit contract behavior of the translation is used. The translation of a program consisting of translation units where the implicit contract behavior is not the same in all translation units is conditionally-supported. There should be no programmatic way of setting, modifying, or querying the implicit contract behavior of a translation unit.

[*Note:* Multiple contract conditions may be applied to a function type with the same or different ~~*contract-levels*~~ *contract-behaviors* . [*Example:*

```
int z;

bool is_prime(int k);

void f(int x)
  [[expects: x > 0]]
  [[expects ~~audit~~ enforce: is_prime(x)]]
  [[ensures assume: z > 10]]
{
  /* ... */
}
```

— *end example*] — *end note*]

The predicate of a contract with *contract-behavior* `ignore` or `assume` is an unevaluated operand (**expr.prop**). The predicate of a contract without *contract-behavior* where the implicit contract behavior is `ignore` or `assume` is not evaluated. [*Note:* The predicate is potentially evaluated (**basic.def.odr**). — *end note*]

If the predicate of a contract with the contract behavior `assume` would evaluate to `false`, the behavior is undefined.

> The violation handler of a program is a function of type "`noexcept`$_{opt}$ function of (lvalue reference to `const std::contract_violation`) returning `void`". The violation handler is invoked when the predicate of a checked contract evaluates to `false` (called a contract violation). There should be no programmatic way of setting or modifying the violation handler. It is implementation defined how the violation handler is established for a program and how the `std::contract_violation` (**support.contract.cviol**) argument value is set, except as specified below.
>
> > Implementations are encouraged to provide a default *violation handler* that outputs the contents of the `std::contract_violation` object and then returns normally
>
> If a precondition is violated, the source location of the violation is implementation defined [*Note:* Implementations are encouraged but not required to report the caller site. — *end note*] If a postcondition is violated, the source location

of the violation is the source location of the function definition. If an assertion is violated, the source location of the violation is the source location of the statement to which the assertion is applied.

If a violation handler exits by throwing an exception and a contract is violated on a call to a function with a non-throwing exception specification, then the behavior is as if the exception escaped the function body. [*Note:* The function `std::terminate` is invoked (**except.terminate**). — *end note*] [*Example:*

```
void f(int x) noexcept [[expects: x > 0]];

void g() {
  f(0);                                          // std::terminate() if violation handler throws
  /* ... */
}
```

— *end example*]

A checked contract is a contract with the contract behavior `inform` or `enforce`. If the contract behavior of a violated contract is `enforce` and execution of the violation handler does not exit via an exception, execution is terminated by invoking the function `std::terminate` (**except.terminate**). [*Note:* A contract behavior of `enforce` is for detecting and ending a program as soon as a bug has been found. A contract behavior of `inform` is provides the opportunity to instrument a contract into a pre-existing code base and fix errors before enforcing the check. — *end note*] [*Example:*

```
void f(int x) [[expects inform: x > 0]];
void g(int x) [[expects enforce: x > 0]];
void h() {
  f(0);          // Violation handler invoked.
  g(0);          // Violation handler invoked then std::terminate() after handler.
  /* ... */
}
```

— *end example*]

A translation may be performed with one of the following violation continuation modes: *off* or *on*. A translation with violation continuation mode set to *off* terminates execution by invoking the function `std::terminate` (**except.terminate**) after completing the execution of the violation handler. A translation with a violation continuation mode set to *on* continues execution after completing the execution of the violation handler. If no continuation mode is explicitly selected, the default continuation mode is *off*. [*Note:* A continuation mode set to *on* provides the opportunity to install a logging handler to instrument a pre-existing code base and fix errors before enforcing checks. — *end note*] [*Example:*

```
void f(int x) [[expects: x > 0]];

void g() {
```

```
  f(0);          // std::terminate() after handler if continuation mode is off;
                 // proceeds after handler if continuation mode is on
  /* ... */
}
```

— *end example*]

In **[expr.const]** paragraph 4 update the paragraph after item 4.23 to be a list:

> If `e` satisfies the constraints of a core constant expression, but evaluation of `e` would evaluate an operation that has undefined behavior as specified in Clause 15 through Clause 31 of this document, or an invocation of the `va_start` macro (**cstdarg.syn**), it is unspecified whether `e` is a core constant expression.
>
> If `e` satisfies the constraints of a core constant expression, it is unspecified whether `e` is a core constant expression if evaluation of `e` would do any of the following:
>
> - evaluate an operation that has undefined behavior as specified in Clause 15 through Clause 31 of this document;
> - evaluate a contract with the contract behavior `assume` (**dcl.attr.contract.check**) that would evaluate to false; or
> - evaluate an invocation of the `va_start` macro (**cstdarg.syn**).

[except.terminate] needs the following line modified:

> In some situations exception handling must be abandoned for less subtle error handling techniques. [Note: These situations are:
>
> - ...
> - when the violation handler has completed after a failed contract check ~~and the continuation mode is off~~ with the `enforce` contract behavior, or
> - ...
>
>]

6 Notes and Open issues

6.1 Wording changes from CWG

Working with Jens, the below name changes in how contracts are specified have been applied:

- Contract semantics are being referred to as “Contract Behaviors”. Jens found the use of semantics as a noun, especially when often used in the less common singular form, a distraction and we agreed Behavior to be the better term to fit in the standard.

6.2 Open questions for EWG

- `std::terminate` or `std::abort`
- `std::contract_violation` should have `assertion_level` removed and then replaced by `contract_behavior`. Should that be incorporated into this paper, go to LEWG, or both?
- P1704 – assume contracts are note evaluated and are unevalulated contexts.
- Should violation handler invocation/enforce be exception-agnostic with respect to the violation handler throwing?
- P1670R0

6.3 Open questions for CWG

- Feature test macro?

6.4 Wednesday Papers

I will be happy to prepare a summary of the impact instead of the proposal for EWG for the papers that have been subsumed by D1607R1.

P1704R0 – Undefined functions in axiom-level contract statements – This paper is now no longer relevant, contracts with the `assume` behavior have these semantics and capture the intent of this paper.

P1670R0 – Side Effects of Checked Contracts and Predicate Elision – This discussion and change is just as valid to discuss now, and can easily be applied to the wording changes in this paper.

P1448R0 – Simplifying Mixed Contract Modes – I have clarified the meaning of this small change with Jens and do not believe it needs addressing any longer.

P1672R0 – "Axiom" is a False Friend – This paper is no longer relevant.

P1671R0 – Contract Evaluation in Constant Expressions – This paper has been subsumed by the wording we have put in D1607R1.

6.5 FAQ

Q: If `axiom` is removed, how can I express conditions that cannot be meaningfully evaluated? A: If a contract behavior is explicitly `ignore` or `assume`, then it is an unevaluated context that remains visible to static analysis tools. For example, the following might be written against N4820:

```
template <input_itetator it, predicate test>
bool all_of(it first, it last, test fn)
  [[expects axiom : is_reachable(first, last)]];
```

might be rewritten as:

```
template <input_itetator it, predicate test>
bool all_of(it first, it last, test fn)
  [[pre ignore : is_reachable(first, last)]];
```

Q: If there is no global continue flag, how can I introduce contracts into my live system? A: Firstly, you can introduce new contracts into existing code using the `ignore` or (preferably) `inform` behaviors. If you are ingesting a large piece of code or library from a third party, you could map the implicit contract behavior to either `ignore` or `inform` while validating the system, or use libraries that provide their own well-documented configuration for determining what behaviors their libraries will have in order to get the `inform` behavior.

Q: If `audit` is removed, how can I indicate which predicates are too expensive to check at runtime by default? A: There is no immediate support for `audit`-like predicates in the new wording. However, the building blocks are present to provide a similar facility with macros while we gain more experience.

For example, the following might be written against N4820:

```
template <forward_itetator InIter, random_access_itetator OutIter, predicate Test>
bool sort( InIter first_in,  InIter  last_in,
           OutIter first_out, OutIter last_out,
           Test compare)
  [[expects axiom : is_reachable(first_in,  last_in )]]
  [[expects axiom : is_reachable(first_out, last_out)]]
  [[ensures       : is_sorted(first_out, last_ount)]]
  [[ensures audit : is_permutation(first_in, last_in, first_out, last_ount)]];
```

might be rewritten as:

```
#if defined(ENFORCE_EXPENSIVE_CHECKS)
# define AUDIT enfore
#else
# define AUDIT ignore
#endif

template <forward_itetator InIter, random_access_itetator OutIter, predicate Test>
bool sort( InIter first_in,  InIter  last_in,
           OutIter first_out, OutIter last_out,
           Test compare)
  [[pre assume : is_reachable(first_in,  last_in )]]
  [[pre assume : is_reachable(first_out, last_out)]]
  [[post       : is_sorted   (first_out, last_out)]]
  [[post AUDIT : is_permutation(first_in, last_in, first_out, last_ount)]];
```

7 References

[N4800] Richard Smith, *Working Draft, Standard for Programming Language C++*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4800.pdf

[P1429R1] Joshua Berne, John Lakos *Contracts That Work*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1429r1.pdf

[P1344R0] Nathan Myers *Pre/Post vs. Enspects/Exsures*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1344r0.md

[P0542R5] G. Dos Reis, J. D. Garcia, J. Lakos, A. Meredith, N. Myers, B. Stroustrup, *Support for contract based programming in C++*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0542r5.html

[P0681R0] Lisa Lippincott *Precise Semantics for Assertions*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0681r0.pdf