Doc. no.: P1711R0
Date: 2019-06-13
Audience: EWG
Reply to: Bjarne Stroustrup (bs@ms.com)

What to do about contracts?

Bjarne Stroustrup

Abstract

I think we have three choices:

1. Do a minimal fix to status quo
2. Do nothing: leave status quo in place
3. Remove contracts from the WP

I list them here in my order of preference. Below, I present arguments and my favorite “minimal fix.” I do not try to repeat all the alternatives, arguments, and examples presented over the last year. The arguments presented are brief summaries.

1. What are contracts for?

A contract is a statement of what is supposed to be true; that is, every contract is true in a perfectly correct program for its expected range of inputs.

I see three primary uses of contracts (in what I think as their order of importance):

1. Systematic and controlled run-time tests
2. Information for static analyzers
3. Information for optimizers

In addition, there have been suggestions to augment contracts with specific features to support testing, debugging, logging, and more. I consider those mission creep beyond the classical uses of contracts. I don’t deny that such specific support can be useful, but I think we should wait until we have further experience.

The three primary uses have seen decades of practical use in many languages and systems and are backed by significant academic research. Furthermore, in the context of WG21 only the support of ***all*** three constituencies could add up to consensus.

2. What is a successful strategy for new features?

Please remember that most languages fail and that essentially no language feature is perfect when it is first introduced. I therefore claim that the best strategy for introducing new features is typically to start with a core facility and gain experience (implementation, experimentation, large scale use) before elaborating and standardizing further. To succeed in absence of the rare stroke of genius, we need to rely on feedback.

This implies that it is important to start early and to start small. Examples:

- Constexpr
- Templates
- Lambdas
- concurrency facilities
- STL containers and algorithms

3. A few definitions

- Assumption – allowing the compiler to consider a contract predicate true and make optimization decisions based on it.
- Continuation – continue the execution of a program after returning from a violation handler.
- Axiom – a predicate that can be taken to be true without proof. A set of axioms can be inconsistent (and ideally, we detect such inconsistencies). A successful run-time check is a proof.

4. The case for Status quo

We have status quo. Contracts are in the WP. If this is good enough as a base for further work, we can save ourselves a lot of work and potential controversy by not touching it. At this late stage even removing contracts or parts of contracts is work even if that removal (most unlikely) is uncontroversial.

Status quo is good enough to replace simple ad-hoc checks (e.g., **assert()** and the ugly contract macros in the C++ Core Guidelines) with something standard. It would also be a boost for the introduction of contracts into education (where non-standards features are often avoided).

In my opinion, assumption by default is a mistake. However, since checking and assuming are incompatible (once you have checked, you aren't assuming; instead, you know), I can suppress undesirable assumption by enabling checking. In particular, "default" checks are supposed to be affordable in most context. Those are likely to be the majority of contracts and the ones most likely to be erroneous and to catch user errors. Assuming "axioms" is IMO fine because they are not executable, too obscure to be of interest to optimizers, and/or too expensive to realistically check (even in "audit"), thus de facto assumed by the programmer anyway. That leaves "audit" checks and those are likely to be so costly that they are disabled in production code anyway; thus reflecting de facto assumptions after testing.

Note that axioms are essential for static analyzers.

Assumption is considered essential by some. I am not arguing against assumption for optimization, only that it should not be the default.

I have come to dislike continuation. It's a major complicating factor and most unusual in contract systems outside the WG21 proposal. I accepted the logic for it and thought there was ample empirical evidence. In status quo, I can avoid problems by simply not enabling it. Also, there are people who consider the global continuation switch that is part of status quo useful for introduction of new contracts into a large code base.

Compiler/build options are likely to be designed to support the more useful aspects of status quo. It is likely that compilers will supply a way to suppress assumption.

There is an implementation of status quo.

Why ***not*** status quo?

- The use is too dependent on options
- The default to assume is too dangerous
- There is no guaranteed way to turn off assumption from axioms
- The implementation of continuation is tricky
- Continuation easily surprises programmers

5. The case for removal of contracts

For removal

- The contract system obviously isn't perfect.
- Many people want a rather large variety of incompatible extensions.
- Nothing we do now can please everybody.
- If we don't have contracts, we can't get them wrong.

Why ***not*** remove contracts?

- Removal without replacement deprives us of the benefits (incl. user experience) obtainable from status quo.
- Removal is work.
- Removal brings us back to square one with all the suggested diverse approaches we have seen over the last year re-emerging (syntax, semantics, language level, fundamental approaches, scope of features, build options, and terminology). Given the somewhat inflamed discussions, this could burn up a lot of time.
- Status quo represents a set of tradeoffs among the concerns of people involved; starting from scratch, we'd have to balance a new set of concerns.
- If removed, the correct approach afterwards would be to agree to a minimal facility for C++23 and then expand it for C++26. Basically, we are likely to be discussing ultimate scope of contracts and the meaning of minimalism in 2022 just as we are in 2019.
- This "C++23 minimal facility", achieved after much work and likely controversy, will almost certainly contain a nucleus similar to status quo that will be similar to core facilities from other languages and strongly resemble the suggested "minimal alternative" below.
- Without status quo or a minimal fix, we are stuck with simple ad-hoc checks (e.g., **assert()** and the ugly contract macros in the C++ Core Guidelines). It would remain difficult to introduce contracts into education (where non-standards features are often avoided) and different groups will (as ever) use incompatible schemes.

6. A minimal alternative

Several "minimal alternatives" to status quo have been offered (and a few not so minimal ones). Having looked at all the WG21 papers, talked with people, and read hundreds of reflector messages, I conclude that none were quite minimal enough and/or failed to offer something deemed essential by one or more of the three key constituencies.

"Minimal alternatives" (published and privately discussed) have tended to remove parts of status quo and then add a few novel features. I think the "add a few novel features" is untenable given the state of discussions and the time pressure. Late novel features are always suspect and likely to be opposed by some – if for no other reason than just because they are late and novel.

Here, I will discuss two interpretations of "minimal":

- Minimal change to status quo:
    - No contract is assumed by default, but there is a compile/build option to enable assumption.
- Minimal facility:
    - No contract is assumed by default, but there is a compile/build option to enable assumption.
    - no continuation.

Rationale that apply to both "minimal alternatives":

- Static analysis can use contracts in any way it likes. It does not directly affect run-time.
- Like for status quo, we need something to replace simple ad-hoc checks (e.g., **assert()** and the ugly contract macros in the C++ Core Guidelines) with something standard. It would also be a boost for the introduction of contracts into education (where non-standards features are often avoided).
- Either alternative would allow simple use and would give us experience that might be useful for more elaborate contract support.
- A minimal alternative would be a base for further work, thus saving us from redesigning and renegotiating contracts from scratch.
- The are some concerns that not being able to turn off assumption from axioms will lead to disuse or misuse of axioms. I don't share that concern, but the alternatives both avoids that problem.

6.1. Minimal change

A common reason to write a contract (arguably the most common reason) is to be able to check that an assumption holds and terminate if it does not. We often write such contracts because we suspect that there might be bugs somewhere so that they don't hold. Consequently, we should suspect that unchecked contracts can be violated and not by default assume based on them:

- No contract is assumed by default, but there is a compile/build option to enable assumption.

Brief rationale:

- Optimization based on assumption can be significant in some programs.
- There are programs that are so well tested that it is worthwhile to assume that the contracts are correct so we can let the optimizer do its best based on them without checking. Thus, avoiding assumption by default is good, but an option that allows assumption also makes sense.
- For optimization, we need a switch to enable assumption for unchecked contracts. We could consider a finer grain of control that just on/off, but ***all*** contract predicates are supposed to be

  true and only axioms can't be checked. If you don't trust your contracts, check them. Don't use axioms for contracts you don't trust.
- The assumption switch is in the nature of an optimization option and does not affect source code.

Why ***not*** this minimal alternative?

- It is work (if we go in this direction, we need a pruned wording ASAP).
- It doesn't add new features to address the many suggested changes to status quo.
- Discussing it will re-open old discussions/wounds.

6.2. Minimal feature

A common reason to write a contract (arguably the most common reason) is to make sure that a program does not proceed if it violates an assumption. Thus, we can have an even more minimal feature that the one from "minimal change"

- No contract is assumed by default, but there is a compile/build option to enable assumption.
- no continuation.

Brief rationale:

- Programmers must consider how to deal with the possibility of a violation handler changing the program state.
- Programmers must consider how to deal with the possibility of new code may be added (making new violations possible) and continuation enabled for a build after completing testing their code.
- Without the possibility of "continuation", we know that statements following a contract enabled for run-time checking will not be executed if the contract predicate fails. That's familiar and useful information.
- With continuation two different object files must be generated (because optimization can differ dependent on whether continuation is possible or not, e.g., if we cannot continue after violation, we know that the contract holds after the check).
- Many (all?) of the suggested improvements to continuation involve forms of selective continuation controlled by code annotations. A global annotation may turn out to be a temporary measure; we should not put temporary measures into an IS.
- The many suggested improvements beyond the classical contracts related to continuation is a sign of immaturity of the continuation option.

Why ***not*** this minimal alternative?

- It is work (if we go in this direction, we need a pruned wording ASAP).
- Removing continuation is too major a change (relative to the WP status quo).
- It doesn't add new features to address the many suggested improvements of status quo.
- Discussing it will re-open old discussions/wounds.

7. Conclusion

Adopt one of the "minimal alternatives." I prefer the "minimal feature" ("assumption switch" plus "no continuation).

If not, do nothing (aka stick to status quo).

8. Suggested wording changes

Ville Voutilainen helped me with the wording and the "minimal change" wording is intended to be identical to his P1710.

8.1. Minimal change (add assumptions mode)

In [basic.def.odr]/12.6, modify as follows:

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 and contract assumption mode; and

In [dcl.attr.contract.check]/3, modify as follows:

The translation of a program consisting of translation
units where the build level, continuation mode,
and contract assumption mode ~~is~~ are not the same
in all translation units is conditionally-supported.

In [dcl.attr.contract.check]/4, modify as follows:

A translation may be performed with one of the following
contract assumption modes: off or on.
If no contract assumption mode is explicitly selected, the default
contract assumption mode is off.
During constant expression evaluation (7.7), only predicates of
checked contracts are evaluated. In other
contexts, it is unspecified whether the predicate for a contract that
is not checked under the current build
level is evaluated; if the predicate of such a contract would evaluate to
false
and contract assumption mode is on, the behavior is undefined.

8.2. Minimal feature (add assumption mode and remove continuation mode)

In [basic.def.odr]/12.6, modify as follows:

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~~ and contract assumption mode; and

In [dcl.attr.contract.check]/3, modify as follows:

The translation of a program consisting of translation
units where the build level and contract assumption mode ~~is~~ are not the same
in all translation units is conditionally-supported.

In [dcl.attr.contract.check]/4, modify as follows:

A translation may be performed with one of the following
contract assumption modes: off or on.
If no contract assumption mode is explicitly selected, the default
contract assumption mode is off.
During constant expression evaluation (7.7), only predicates of
checked contracts are evaluated. In other
contexts, it is unspecified whether the predicate for a contract that
is not checked under the current build
level is evaluated; if the predicate of such a contract would evaluate to
false
and contract assumption mode is on, the behavior is undefined.

Strike [dcl.attr.contract.check]/7:

~~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 (14.5.1) 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]~~

Modify [except.terminate]/1.10 as follows:

when the violation handler has completed after a failed contract check ~~and the continuation mode is off~~, or