A minimal solution for contracts, or, what is an MVP?

P3889R0
Harald Achitz harald@swedencpp.se
Audience: EWG, CWG, LWG
2025-10-22

P2900 claims to be a minimum viable product (MVP), but is that true?
Four new compilation modes, a lot of implementation-defined behavior, and even by the authors acknowledged problems not addressable with today's tools.

It takes time to teach, time to understand, and offloads a massive workload on tooling implementors and toolchain maintainers. And much of this is due to the extensive use of the term implementation-defined. Since the C++ standard is largely unaware of tools, this opens an entire domain for interpretation and argumentation. One could say that, on paper, almost any position can be justified. A problem domain of its own.

This does not sound like an MVP.

Is it possible that parts of our C++ community are trapped in a thought loop of complexity, and lost focus in delivering the smallest and simplest next step? Let's do a thought experiment and have a look at what a simple solution could look like.

Disclaimer

The example given is a demonstration that essential contract functionality can be reached without requiring any implementation defined behavior and the introduction of new compiler flags. It is teachable in five minutes, and even newcomer to the language will understand this type of contracts in no time. However, there is no expectation that this will be considered as an implementation by WG21, even if it possible could be.

This paper is also a response the requires of communicating with papers, given in the SIS/TK611/AG09 paper-club meeting 2025-10-21.

Today's defensive programming

This paper takes the naming introduced in MS-GSL, `EXPECTS` and `ENSURES`, since this mirrors the mechanism many use-cases have today.

Consider the following code today:

```
void foo(Bar* p) {
    EXPECTS(p != nullptr);
    EXPECTS(p->hadData());
    // ....
}
```

There is no smart solution here! This is just plain defensive programming as you would expect. A condition is met, or some problem handler kicks in, probably log but in any case exits the program, or, at least the function. No surprise for the reader of the code for whom we also write the implementation.

There might be other, way more smart and complex solution that does things, introducing optional checkpoints instead of contracts and other smart things, but lets forget them for a moment. We will come back to those later.

Today's contract proposal

With the current contract proposal, this might result in undefined behavior in some of the four compilation modes. The implementation needs to be re-written as follows:

```
void foo(Bar* p) {
    contract_assert(p != nullptr && p->hadData());
    // ....
}
```

This is due to some of the various compilation modes of P2900, which will continue after a check.

This means:

- The existing code needs to change.
- The resulting log entry has severe information loss. Was `p` null, or did it have no data?

This problem already exists in practice when using assertion libraries with a *check-and-continue* mode. In such cases, users have effectively created a way to shoot themselves in the foot, their choice. What is less acceptable is standardizing such a dangerous mechanism and presenting it as a contract.

`pre` / `post` conditions have the same problem, plus some additional ones, so they are not even mentioned.

An MVP thought experiment

What if we had:

- only one contract assert that always fires
- the possibility to set the contract violation handler

The code would look like this:

```
void foo(Bar* p) {
    contract_assert(p != nullptr);
    contract_assert(p->hadData());
    // ....
}
```

- it will always fail, real defensive programming (as in the original code)
- for the special case it shall resume, use a contract handler that does nothing (only logging).
  I am unsure how that shall work in practice, since the code is UB, but some people claim they have a use-case for that.
- If more configuration options are wanted, people can make MACROS as they have today and have had since years. But they could use the condition check mechanism in a standardized way.

This might not be the 'perfect' scenario but it is in any case a simple one that highlights the nature of a MVP and gets a lot of jobs done.

And other smart and complex solutions can be implemented in the same way as they are today, but benefit from a standardized contract check.

QA

Question: That is way too simple and does not handle all the edge cases.
Answer: Yes, it is an MVP. No tool implementation-defined behavior, 100% under control of the developer, teachable in 5 minutes.

Question: What about all the special cases where I want to do ...
Answer: It is never a good idea to generalize the special cases to the base cases. Solve the base case and figure out the special cases later.

Question: What about pre and post conditions of the function signature?
Answer: It might be nice to have them, but so far, there seem to be quite a few issues that cause a lot of discussion. Which context are they executed in, how often are they executed, are they executed at all, and how to write them to not get new UB? It seems more research needs to be done to avoid that pre/post become only problematic syntax sugar.

Question: Why not add more, like ...
Answer: This is an MVP; we can do more in the future if it turns out this is required. It is easier to add something to the standard if it is missing than to remove something that was not fully explored.

Summary

This proposal might look way too simple, and maybe it is. But it is a response to the request of WG21 to communicate with papers. And it was quickly written to make a point. This paper does not claim that the contract proposal has to be implemented as here specified, even if it might be a good idea. But it claims that the current solution is not universally usable and simple enough to be an MVP for the C++ standard.

By adding four new compilation modes, pre + post conditions that might fire 0 to N times, but do not even handle the simplest default case without looking at compiler flags as an MVP, it is hard to believe that the average C++ user was the target of the contract proposal, and that the term MVP as it is used is based on a common understanding.

A simple solution, like the one mentioned here or something similar, is capable of serving more people effectively than many might expect. That might have been overlooked in the discussion of the topic.