P3460 C++ Contracts Implementers Report

Summary

TL;DR: C++ Contracts were implemented in GCC and Clang according to P2900, and usage experience was gathered across large code bases.

P2900's specification was clear and implementable. At the time of writing this paper, the most recent version of P2900 is P2900R8.

Deployment experience was collected from deploying contracts in

- LLVM and dependencies by replacing `<cassert>`
- The C++ standard library by replacing Libc++'s existing hardening, validation, and debugging macros.
- The BDE codebase

Constification discovered much const-incorrect code. Most were fixed once it was discovered by contracts.

Contracts in the C++ Standard Library improved the QoI for hardening and security conscious users.

	Constification	Result Names	Lambda Capture Rules	Contract Violation On Exception	Virtual Functions (P3097)	Coroutines (P2957)
Clang	✅	✅	✅	✅	➖	✅
GCC	✅	✅	➖	➖	✅	✅

✅ Complete ➖ Unimplemented

Preview implementations / Learning Context

Reading Rainbow Tip: Your early feedback helps! Try Contracts on Godbolt today with Clang & libc++ integration or GCC

Authors

- Eric Fiselier <eric@efcs.ca>
- Nina Dinka Ranns <dinka.ranns@googlemail.com>
- Iain Sandoe <iain@sandoe.co.uk>

Contents

Implementation in Clang

Result Name Introducer

By far the most novel and difficult part of the proposal to implement in Clang. Here are some minutiae deserving the attention of CWG.

Late Parsed or Dependent?

```
struct A {
 template <class U> bool f() const;
};
auto h() // Is '.template' required?
  post(v : v.template f<int>())
{ return A(); }
```

Clarification Needed: Deduced return types are not yet known when parsing the result name. The standard should clarify if they are late-parsed or treated as dependent.

We believe they should be late parsed.

Are Postconditions chained?

```
int f()
 post(r : ++const_cast<int&>(r) == 1)
 post(r : ++const_cast<int&>(r) == 2)
// OK?
{ return 0; }
```

Clarification Needed: Without an RVO slot, it's unclear if the same materialized temporary must be used in each post condition.

Constification

Constification was trivial and unsurprising to implement in both GCC and Clang.

Work is ongoing in Clang to improve the diagnostics emitted as a result of implicitly added const.

The deployment of const-ified contracts in LLVM and the C++ standard library are discussed in the section below

---

Coroutines

In Clang, P2957 was implemented on accident as a consequence of function level contracts. Additional work was needed to reject coroutine contracts as required by P2900.

Deployment Experience

Clang's contract implementations were deployed and tested in two ways.

The Standard Library

`contract_assert` was used to implement libc++'s internal `_LIBCPP_ASSERT` macro[1].

P2900 contracts provided a rich enough feature set to provide the same functionality,

[1] https://libcxx.llvm.org/Hardening.html

but with a standardized interface which is easier to communicate to users. The quick enforce semantic was particularly useful.

The deployment found one const-correctness bug in libc++'s internals, otherwise all other assertions compiled without change.

The libc++ maintainers support P2900 in C++26.

The experience from this deployment is limited to `contract_assert`, as no new `pre` and `post` conditions were added.

<assert.h>

```
#define assert(...) \
({ contract_assert(__VA_ARGS__); \
  ((void)0); \
})
```

Note: The above code is an example, not a proposal.

To test the effects of const-ification in existing codebases, `#define assert` was modified to use `contract_assert`[2]. The change was tested by compiling LLVM with assertions enabled.

Constification caused compilation errors in many existing assertions. A day of effort was needed to clean up or address the breakages. Of the compilation errors, the breakdown in order of frequency was

- missing `const` on actually const member function.
- Missing `const` on a parameter.
- Non-const modification of debug specific data structures.
- operator overload resolution failure; with user-defined iterator types being the most common cause.

Adding `const` was the most frequent solution to compilation failures.

[2] GNU statement-expressions were used to allow statements like: `return assert(true), 42;`

Implementation in GCC

P2900

The GCC implementation of P2900 was done on top of an already existing implementation of the (proposed, but removed) C++20 attribute-based contracts. This brought the additional challenge of maintaining the existing C++20 version in parallel with the implementation of P2900. Thankfully it did not create any significant challenges. At some point, a redesign may be done to have P2900 contracts implementation move away from fitting into the attribute mechanism it inherited from C++20. We do not expect this to be a significant challenge either.

For details about C++20 contracts implementation experience, please see P1680. GCC implementation of P2900 was used to gain usage experience in BDE. For deployment experience in BDE, please see P3336. Implementation is deployed to Compiler explorer under `contracts natural syntax`. The code has not been merged into any official branch, but https://github.com/villevoutilainen/gcc/tree/contracts-nonattr currently maintains a stable version of P2900 implementation.

To enable contracts all together, `-fcontracts` is needed.

To get P2900 specific behaviour, `-fcontracts-nonattr` is needed.

There is currently no way to specifically configure P2900 specified contract violation semantics. However, the default semantic maps onto the `enforce` semantic, and providing the `-fcontract-continuation-mode=on` will give `observe` semantic.

By default, the compiler will constify expressions appearing in contract assertions. Currently, constification is implemented as per P2900R7. It is possible to disable constification using the `-fcontracts-nonattr-noconst`.

The version available on Compiler Explorer currently supports `contract_violation` object as specified in N4820. The user can replace the default contract violation handler by providing a function with the signature

`void handle_contract_violation(const std::experimental::contract_violation &violation)`

We expect to have the P2900R8 version of the `contract_violation` published to the stable version branch and available on Compiler Explorer by the Wroclaw meeting.

Virtual Function Support (P3097)

GCC's stable branch of the P2900 implementation supports contracts on virtual functions as described in P3097. Virtual function call expressions are replaced with a new one that invokes a compiler generated TU-local "wrapper" function of unspecified (but implementation-private) name. The wrapper performs the contract checks of the static type's function and invokes the original call expression. The callee side checks are performed as part of the dynamic function invocation. Implementation did not present a significant challenge.

Some Mitigation of the effects of UB in Contract Checks

Since contract checks are regular C++, and they are usually compiled with the optimisation set for the entire TU, the compiler is free to apply optimisations based on the premise that code cannot be reached if to do so would invoke UB. An example of this is shown in P3285 where the compiler can choose to elide a contract check completely on the basis that it cannot be reached without integer overflow.

How to handle this generally is a matter of on-going design deliberations, and what is described here is only one potential facet of those (providing a mechanism to disable UB-based optimisations in contract checks).

We have made an implementation in which the contract checks are performed as outlined functions. In order to provide some mitigation against such checks being elided we are able to compile the outlined check functions with different optimisation to the main function body. Trivially, one could compile them "-O0" preventing inlining and other analysis/optimisation that would result in the checks being seen as removable. One could, of course, be more sophisticated (in the particular example given, it would be sufficient to allow integer wrap-around). In some future implementation this could also form the basis of segregating the compilation of contract checks, if the eventual design requires it.

Coroutine Support (P2957, P3387)

Since support for contracts on coroutines is considered very important (even for the MVP), we have implemented a proposed design described in P2957 (with more detailed design deliberations spelled out in P3387). The implementation is available on both mainline GCC and the "contracts natural syntax" branch. The design follows the intent that coroutine-ness is an implementation detail, and therefore from the caller's perspective it should adopt the same rules as any regular function (including those pertinent to contracts).

Note that many (probably most) real-life coroutines return some object to the caller that allows management of the coroutine (let's call this the 'management object' for now); identifying this is relevant to contracts in that it represents the returned object for the called function that can be inspected in a post condition (even tho the user never authors a `return` statement for it).

From the implementer's perspective it can be helpful to think about a coroutine *definition* as separated into two layers:

- A top layer responsible for setting up the coroutine, returning any management object to the caller and handling exceptions that occur during the setup (we call this the 'ramp').
- A lower layer consisting of the re-written body of the coroutine (which is the user's authored function

body, wrapped in some housekeeping functionality mandated by the standard).

The ramp layer has ownership of the original function parameters, and any management object. The re-written body has ownership of parameter copies, and any local state associated with the user's code. The lifetime *ends* of the two sets of data are unsequenced; a consequence of the fact that the re-written body of the function is (most often) going to execute after the ramp layer has completed and returned the management object to its caller.

For callee-side contract checks:

- Require precondition checks as the first action of the ramp function (that means we may access the original function params) - exactly as per a 'regular' function.
- Identify that `contract_assert` behaves 'as normal' in the function body (i.e. there are no special requirements).
- Post conditions are required to be evaluated on every non-exceptional edge out of the ramp (i.e. the `ramp` behaves exactly as a regular function, returning the management object to the caller). The management object is the one referred to in post conditions (exactly as for a 'regular' function - since it's the returned value).

In GCC these requirements were met with existing functionality; the pre-conditions are simply inserted at the ramp start, and the post conditions are applied in a try-finally wrapper around the ramp. As noted above, no changes are required to the existing contracts implementation to cater for `contract_assert`.

For caller-side contracts (currently only implemented in GCC for virtual functions) coroutines behave exactly the same way as any 'regular' function, as per the intent that coroutine-ness is an implementation detail.

There is currently a missing diagnostic for post conditions (which are not permitted to access non-reference params) but that is not expected to present any difficulty.

Deployment experience

The GCC implementation of P2900 was used to gain usage experience in BDE, this is described in P3336.

Links

- GCC Godbolt: https://godbolt.org/z/7rPxa7TP6
- Clang Godbolt: https://godbolt.org/z/qEo1vGhqM
- GCC experimental branch source code: https://github.com/villevoutilainen/gcc/tree/contracts-nonattr

References

- P2900R9 -Berne, J., Doumler, T., Krzemieński, A., Gašper Ažman, Louis Dionne, Tom Honermann, John Lakos, Lisa Lippincott, Jens Maurer, Ryan McDougall, Jason Merrill, & Ville Voutilainen. (2024). *Contracts for C++*.
- P3336R0 Berne, J. (2024). *Usage Experience for Contracts with BDE.*
- P3097R0 Doumler, T., Berne, J., & Ažman, G. (2024). *Contracts for C++: Support for Virtual Functions*.
- P1680R0 Sutton, A., & Chapman, J. (2019). *Implementing Contracts in GCC*.
- N4820 Smith, R. (2019) *Working Draft, Standard for Programming Language C++*.
- P2957R2 Krzemieński, A., Sandoe, I., Berne, J. & Doumler, T. (2024). *Contracts and coroutines*.
- P3387R0 Doumler, T., Berne, J., Sandoe, I., & Bindels, P. (2024). *Contract assertions on coroutines*