Contracts have failed to provide a portable "assume"

Timur Doumler (papers@timur.audio)

Document #: P1773R0
Date: 2019-06-17
Project: Programming Language C++
Audience: Evolution Working Group

Abstract

The "assume" semantic, i.e. a condition that is not checked but can be assumed by the compiler, is a powerful tool for generating both faster and smaller code. However, the contract checking facility (CCF) in the current C++20 working draft does not provide this semantic in a useable way. This can be solved by 1) allowing to spell contract semantics directly, instead of contract "levels" and 2) introducing a lower-level facility outside of the CCF that implements this semantic.

1 The "assume" semantic

Contract checking statements (CCS) can have essentially one of three possible semantics:

— **check**: The statement is checked at runtime. If the statement is false, some form of runtime error handling is invoked.

— **ignore**: The statement is not checked at runtime. The compiler ignores the statement.

— **assume**: The statement is not checked at runtime. The compiler assumes that the statement is true, and is free to optimise based on that assumption. If the statement is false, the behaviour is undefined.

Only one of them, "assume", is not already portably implementable today in C++. It can currently only be achieved using platform-specific compiler intrinsics such as `__builtin_assume` (Clang) or `__assume` (MSVC, Intel). It is also a very powerful tool for generating both faster and smaller code. Providing this semantic in a portable way in C++20 would be a huge benefit for low-latency and performance-critical C++ programs.

Adding a portable "assume" was already proposed once [N4425] and discussed by EWG in 2015 in Lenexa[1]. The paper was rejected. EWG's guidance was that this functionality should be provided within the proposed contract checking facility (CCF), and not as a separate facility.

Unfortunately, four years later we are now in a situation where this functionality is neither part of the CCF, nor available as a separate facility. In this paper we argue why the current CCF does not provide a useable "assume" semantic, and how this situation can be improved.

[1] https://cplusplus.github.io/EWG/ewg-closed.html#179

2 Use cases

Correctly using "assume" statements in performance-critical code can lead to generation of both faster and smaller machine code, without changing the observable behaviour of the program.

Consider the following function:

```
int divide_by_32(int x)
{
  __builtin_assume(x >= 0);
  return x/32;
}
```

Without the assumption, the compiler has to generate code that works correctly for all possible input values. With the assumption, it can implement the calculation using a single instruction (shift right by 5 bits). Here is the output generated by clang (trunk) with `-O3`:

Without `__builtin_assume`:

```
mov eax, edi
sar eax, 31
shr eax, 27
add eax, edi
sar eax, 5
ret
```

With `__builtin_assume`:

```
mov eax, edi
shr eax, 5
ret
```

Another example: consider looping over an array of numbers and performing math on the elements. Often, there are invariants on the array size such as: it's a power of two, it's a multiple of the SIMD register size, etc (all very common e.g. in audio processing code). Telling the optimiser about such invariants leads to a much better optimisation and vectorisation of the loop:

```
void limiter(float* buffer, size_t size)
{
  __builtin_assume(size % 8 == 0);
  for (size_t i = 0; i < size; ++i)
     data[i] = std::clamp(data[i], -1.0f, 1.0f);
}
```

For this function, clang (trunk) with `-O3` generates 70 lines of assembly without the assumption, and only 42 lines with it.

See [Regehr2014] for more examples and use cases.

3 The problems with the current CCF

Such "assume" statements typically have the following four characteristics in common:

1. The assumption is a simple expression (for example, a comparison) and therefore would be easy to check.
2. The assumption is an invariant that is guaranteed to hold by code elsewhere – i.e. the intent of writing it is *not* to check for bugs, but to actually perform the optimisation.
3. The assumption is typically a local implementation detail found inside performance-sensitive code, and *not* part of a public API.
4. The assumption expression is guaranteed to be unevaluated.

The current working draft however does not let us spell the intended contract semantics at all. Instead, it only gives us "contract levels" which are then assigned semantics using "build modes": essentially global compiler switches that live outside of the code.

This model works against all four of the characteristics discussed above:

1. Statements that are not meant to be evaluated are represented by "axioms". However, the concept of "axiom" conflates two very different kinds of statements: on the one hand, easy-to-check statements that are meant to be used as guides for optimisation (like shown above), and on the other hand, conditions that are hard or impossible to check by the compiler (such as "is this object moved from?" or "is this a valid range?") that are essentially used as an enhanced "comment" to be ignored by the compiler, but possibly consumed by other tools.

2. The fact that you can only spell "contract levels" in code, but not the intended semantics of a contract, prevents the developer from spelling their *intent* in C++ code. There is no way to express that the primary intent of a particular CCS is "assume this statement to perform optimisation".

3. The fact that semantics can only be assigned globally by a compiler switch, as opposed to locally, prevents the usage of assumptions as an implementation detail. Consider this important use case: you ship a header-only library providing optimised implementations of algorithms. You use "assume" statements internally as an implementation detail to get the desired optimisation. You want to be able to ship this library without the user being aware of this implementation detail, and without them being able to change its semantics. With "contract levels" and "build modes", this is not possible.

4. Because CCS can have different semantics assigned to them at build time, including "check", we can never have CCS that are guaranteed to be unevaluated. Either they will be evaluated (if using a semantic that checks it), or it is unspecified whether they are evaluated (if using a semantic that does not). This makes it impossible to use assumptions for optimisation that would have side effects when evaluated, which is useful (consider `++ptr != end`).

We therefore believe that the CCF, as it exists currently in the C++20 working draft, is not adequate and should not be standardised in its current form.

4 Proposed solution

4.1 Literal semantics

The problems 1. and 2. can be solved by allowing to spell the semantics of a contract explicitly in a CCS. This can be done either *instead* of the contract levels, as proposed in the second half of [P1607R0] ("Minimizing contracts"), or *in addition* to contract levels, as proposed in [P1429R2] ("Contracts that work"). We therefore support adopting one of those papers, or some similar solution that allows the developer to express contracts semantics in C++ code, for C++20. This would allow us to spell an optimisation hint like this:

```
[[assert assume: x >= 0]]
```

Note that the effective semantics of a CCS might still potentially be subject to a build mode. This is useful: you can use CCS as an optimisation hint, but still fall back to a different semantic, for example, for QA purposes.

4.2 `std::assume`

Adding literal semantics to CCS however still does not solve problems 3. and 4. For use cases like assumptions as a local implementation detail in a header-only library (where the end user should not be able to change the semantics or even be aware of that implementation detail), a lower-level facility outside of the CCF is needed. This can be achieved by additionally standardising the facility currently provided by `__builtin_assume` (Clang) and `__assume` (MSVC, Intel). We could then spell an assumption directly like this (modulo bikeshed):

```
std::assume(x >= 0);
```

Please see the companion paper [P1774R0] for a formal proposal to standardise `std::assume` for C++.

In order to enable all use cases of the "assume" semantic discussed here, we need both the CCS literal semantics and the lower-level `std::assume` facility, as both serve different needs. Note that the CCS "assume" semantic can be naturally implemented in terms of the lower-level `std::assume`.

References

[N4425] Hal Finkel. Generalized Dynamic Assumptions. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4425.pdf, 2015-04-07.

[P1429R2] Joshua Berne and John Lakos. Contracts That Work. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1429r2.pdf, 2019-06-14.

[P1607R0] Joshua Berne, Jeff Snyder, and Ryan McDougall. Minimizing Contracts. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1607r0.pdf, 2019-03-10.

[P1774R0] Timur Doumler. Portable optimisation hints. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p1774r0.pdf, 2019-06-17.

[Regehr2014] John Regehr. Assertions Are Pessimistic, Assumptions Are Optimistic. https://blog.regehr.org/archives/1096, 2014-02-05.