Portable assumptions

Timur Doumler (papers@timur.audio)

Document #: P1774R7
Date: 2022-05-15
Project: Programming Language C++
Audience: Core Working Group

Abstract

We propose a standard facility providing the semantics of existing compiler built-ins such as `__builtin_assume` (Clang) and `__assume` (MSVC, ICC). It gives the programmer a way to allow the compiler to assume that a given C++ expression is true, without evaluating it, and to optimise based on this assumption. This is very useful for high-performance and low-latency applications in order to generate both faster and smaller code.

1 Motivation

All major compilers offer built-ins that give the programmer a way to allow the compiler to assume that a given C++ expression is true, and to optimise based on this assumption. They are very useful for high-performance and low-latency applications in order to generate both faster and smaller code. Use cases include more efficient code generation for mathematical operations, better vectorisation of loops, elision of unnecessary branches, function calls, and more.

Consider the following function (example from [Regehr2014]), using a Clang compiler built-in:

```
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, there is no need to generate code that handles the case of a negative numerator. The calculation can therefore be performed using a single instruction (shift right by 5 bits). Here is the output generated by Clang 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
```

Assumptions are a useful expert-level feature and have been existing practice in C++ for many years. All major compilers offer this functionality by providing the following built-ins:

- MSVC and ICC have `__assume(`*expr*`);`
- Clang has `__builtin_assume(`*expr*`);`
- GCC does not have an assumption built-in, but it can be emulated as follows:
  `if (`*expr*`) {} else { __builtin_unreachable(); }`

Macros like this are currently used in an attempt to make assumptions portable:

```
#if defined(__clang__)
  #define ASSUME(expr) __builtin_assume(expr)
#elif defined(__GNUC__) && !defined(__ICC)
  #define ASSUME(expr) if (expr) {} else { __builtin_unreachable(); }
#elif defined(_MSC_VER) || defined(__ICC)
  #define ASSUME(expr) __assume(expr)
#endif
```

Unfortunately, this has slightly different semantics on all compilers. On GCC, this will evaluate[1] *expr*, while on the other compilers, *expr* is not evaluated, only ODR-used; Clang's built-in will error out if *expr* contains a top-level comma, while on the other compilers, it won't; Clang and ICC will ignore non-constant expressions inside the built-in during constant evaluation, while MSVC errors out on them; and so on. And of course, there are also all the problems associated with macros. Most importantly, while compiler documentation gives us an idea of how to work with the assumption built-ins, the exact semantics of assumptions are not properly defined anywhere.

The goal of this proposal is to standardise assumptions in order to make them portable. We propose a unified standard syntax for assumptions as well as unified, precisely defined semantics, in a way that fits well into the existing C++ standard and is compatible with all existing compiler implementations (including compilers that do not have an assumption facility).

Examples of how assumptions affect code generation on existing compilers are given in section 2. In section 3, we discuss the proposed syntax (considered, but not proposed alternatives are listed in section 3.2). Section 4 is dedicated to the proposed semantics and all its subtleties. In section 5, we summarise the history of standardising assumptions and discuss related work such as contracts, assertions, and `std::unreachable`. Previous WG21 subgroup polls on this proposal are listed in section 6. Section 7 contains the proposed wording.

2 Examples

Many basic examples for assumption usage can be found in [Regehr2014] and [P2064R0], including elimination of loop branches and more efficient instructions generated for mathematical expressions. We won't repeat those here, but we will add a couple other interesting examples.

All examples in this section have been tested on Compiler Explorer with the latest[2] trunk versions of MSVC, ICC, Clang and GCC, using the highest optimisation setting available (`/O2` and `-O3`, respectively) and the `ASSUME` macro shown above.

[1] At least notionally; in practice, if the evaluation of `expr` has no side effects, it will often get optimised out.

[2] At the time of writing.

2.1 Limiter

Consider looping over a range of `floats` and clamping all values to the range [-1, 1]. This operation is often used in audio processing and is known as a *limiter*:

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

Often, such data is subject to invariants which are guaranteed to hold, but this information is invisible to the optimiser (for example because the code is too complex for the optimiser to see through, there is a TU boundary in between, or the invariants are properties of the file format or network protocol used). We can inject such invariants via assumptions. In this example, we inject the knowledge that data buffers contain at least 32 frames and the buffer size is a multiple of 32 (a common scenario in audio processing), and that the data does not contain NaNs or infinity:

```
void limiter(float* data, size_t size) {
    ASSUME(size > 0);
    ASSUME(size % 32 == 0);

    for (size_t i = 0; i < size; ++i) {
        ASSUME(std::isfinite(data[i]));
        data[i] = std::clamp(data[i], -1.0f, 1.0f);
    }
}
```

On all compilers except ICC, using `ASSUME` leads to significantly less code being emitted (see Figure 1, left panel). With the injected assumptions about the array size, we get a better optimised vectorised loop with the prologue and epilogue eliminated. Additionally, both MSVC and GCC manage to eliminate unnecessary code inside `std::clamp`.

However, interestingly, on GCC (the only surveyed compiler that lacks an assumption built-in), for some reason the assumption containing `std::isfinite` interferes with the auto-vectoriser, and as a result SIMD is no longer used inside the loop if this assumption is present. This is actually a good argument *for* standardising assumptions, because evidently, the emulation we can achieve on GCC today with `__builtin_unreachable()` is suboptimal.

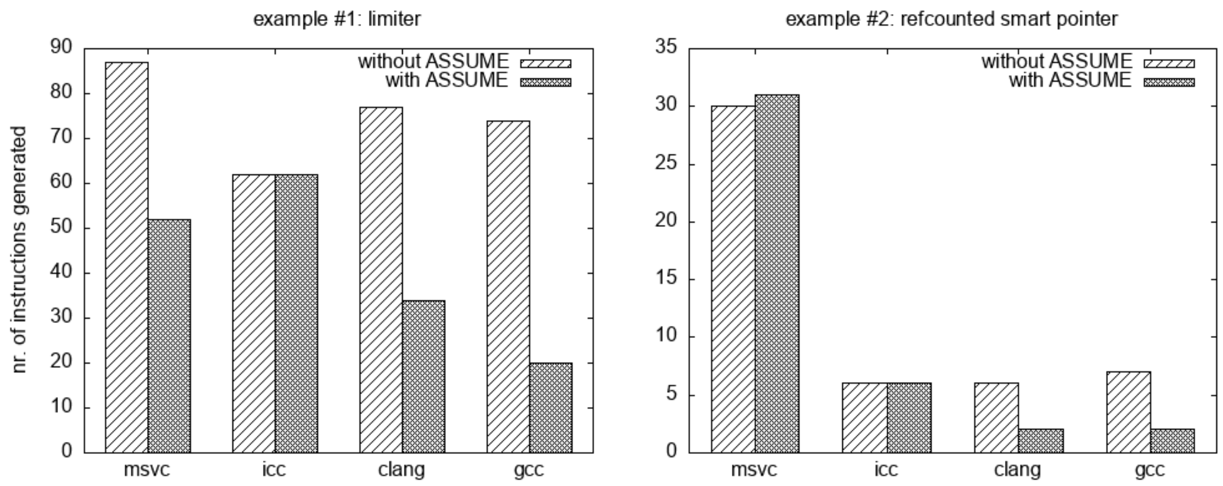

Figure 1: Number of instructions generated by each compiler with and without assumptions.

2.2 Refcounted smart pointer

Here is another, somewhat less obvious example, contributed by Peter Dimov:

```
void destroy() noexcept;

struct Sp {
  int* pn;

  Sp (const Sp& r) noexcept : pn(r.pn) {
    ASSUME(*pn >= 1);
    ++*pn;
  }

  ~Sp() {
    if (--*pn == 0)
    destroy();
  }
};

void g1(Sp p) {}

void g2(Sp p) {
  g1(p);
}
```

where `Sp` is a reference-counted smart pointer.

In the copy constructor we know that there is at least one reference, namely `r`, so we can assume that the reference count is at least one. This assumption enables the compiler to optimise out the reference count increments and decrements and the conditional calls to `destroy` entirely. Both GCC and Clang perform this optimisation, while on MSVC and ICC the assumptions don't significantly change the emitted code (see Figure 1, right panel).

3 Syntax

3.1 Proposed

We propose to spell assumptions as an attribute:

`[[assume(`*expr*`)]];`

First of all, we propose that the word "assume" is used in the spelling this feature. This is the name already used in existing built-ins, therefore choosing it means standardising existing practice. This name will be least surprising and most self-explanatory to the user.

Using standard attribute syntax means that assumptions are backwards-compatible with a compiler that does not support this feature.

Making assumptions an attribute makes it clear that assumptions share an important property with the other C++ attributes: given a valid C++ program that contains the attribute, ignoring it does not change the observable semantics of such a program (see [P2552R0] for a thorough discussion of the ignorability of standard attributes).

It is further consistent with existing optimisation-related attributes (`[[likely]]`, `[[unlikely]]`, `[[carries_dependency]]`) as well as existing attributes that increase the space of undefined behaviour in a C++ program (`[[noreturn]]`). More generally, attributes tend to target the back-end of the compiler and/or other tools in the C++ ecosystem, rather than the front-end. This is true for assumptions as well, which are targeting the optimiser. Therefore, assumptions should be an attribute.

Attribute syntax has the least impact on the existing core language as opposed to the alternatives discussed below, as it avoids adding any new syntax or grammar and therefore does not add more complexity to the language.

Finally, the attribute syntax would also allow to add this feature to the C language with the same spelling (SG22 voted in favour of this; see polls in Section 6). Note that the existing assumption built-ins work in C in the same way they do in C++.

Herb Sutter argues in [P2064R0] against this attribute syntax, saying that it would "make assumes awkward to write in the one place they should appear, which is a statement", and that it "would allow assumes to be written outside of function bodies". Neither of these are true. We specify `[[assume(expr)]]` to be an attribute that can only be applied to a null statement, just like we already do with `[[fallthrough]]`. The effect of this is that it can only appear on its own, as a statement, followed by a semicolon, and only inside a function body, which is exactly the intended use.

3.2 Alternatives considered (not proposed)

3.2.1 New syntax

We explored syntax involving a colon, such as `[[assume:` *`expression`*`]]`, the syntax used in [P0542R5] for contracts, and other variations that deviate from existing C++ attribute grammar.

We do not see any benefit of introducing a novel syntax to C++ over using existing attribute syntax. New syntax would increase the complexity of C++ and require otherwise unnecessary changes to the C++ grammar, making it harder to add assumptions to existing code due to lack of backwards-compatibility, while not giving us anything that we can't do just as well with attribute syntax (however, see discussion in section 4.7).

In addition, using syntax too similar to that used by contracts-related proposals is actively harmful: assumptions are a feature completely separate from contracts (see section 5.1) and assertions (see section 5.3), and the syntax should therefore be separate from them as well.

3.2.2 Keyword

An assumption could be described as an operator, somewhat similar to `decltype(`*`expr`*`)`, where `decltype` is a keyword and *expr* is an unevaluated operand. We therefore considered to add a new keyword for assumptions, so that the spelling becomes:

`assume(`*`expression`*`)`

We could also spell such a keyword differently. [P2064R0] suggests the spelling `unsafe_assume` to highlight that this is a narrow, low-level, expert-only feature, with the potential to inject undefined behaviour into an otherwise valid program, and should therefore be used with great care.

However, for exactly this reason, we believe that a new keyword is not the right approach. Adding a new keyword is a very significant change to the language. A narrow, expert-only feature that will only be used by a small fraction of developers does not justify a new keyword.

3.2.3 Macro

Instead of introducing a keyword, we could introduce an `assume` macro, analogous to how `assert` is already defined as a macro (and again, we could spell it in different ways). However, macros are known to cause many problems. Their lack of scoping can lead to name clashes, the preprocessor grammar makes it impossible to use curly braces inside the expression, etc. For these and other reasons, modern C++ tries to minimise the use of macros. We don't see any good reason to deviate from this principle.

3.2.4 "Magic" library function

At first glance, it seems very attractive if we could spell an assumption as a "magic" library function:

```
std::assume(expression);
```

Herb Sutter [P2064R0] and John Lakos (personal communication, 2019) have both argued for such an approach. However, a deeper analysis reveals that this is not a viable route. Making assumptions a function would introduce a weird novelty into the C++ language: something that is syntactically a function call, yet does not evaluate its argument. This would be very different in nature to all existing "magic" library functions. Apart from not evaluating the argument of the function call, such a function would differ from other C++ language functions in many other ways. It would look like a standard C++ function, but it would behave like built-ins such as `__builtin_assume` behave today: the only thing that you can do with them is to directly call them. You can't take their address, you can't assign them to a function pointer, etc. By making assumptions a function, we would essentially be saying that it's a function but it's so special that the only properties it shares with an actual function is that it has a name and an argument list. It would effectively be a namespaced keyword.

Significant core language changes would be needed to make such a novelty work, adding more complexity to a fundamental part of the core language (what is a function call?). We do not believe that assumptions come anywhere close to justifying such changes to the language. The proposed attribute syntax avoids all this complexity by using a mechanism that already exists in the language.

It has been pointed out that the spelling `std::assume` would be consistent with the related `std::assume_aligned`, which was adopted for C++20. However, as should be clear from the above discussion, they are fundamentally different. For `std::assume_aligned`, unlike for an assumption, the argument may be evaluated, just like for any other function call in C++. The problem described above does therefore not arise for `std::assume_aligned` (or any other existing "magic" library function in C++).

4 Proposed semantics

We corresponded with compiler engineers from MSVC, GCC, Clang, ICC, and EDG, to make sure that the semantics proposed here for standard C++ are implementable on all these compilers and are compatible with the de-facto semantics of all the existing assumption built-ins. We also incorporated feedback from all previous rounds of EWG review as well as discussions on the WG21 reflectors.

4.1 Constraints on the attribute argument clause

The argument clause of an `assume` attribute must be present and must contain a single expression contextually convertible to `bool`. The proposed specification requires this expression to be a *conditional-expression*. This has the effect that expressions with top-level commas and *assignment-expression*s are not allowed by the grammar under this proposal, even if they are contextually convertible to `bool`. If necessary, this restriction can be bypassed by adding an extra pair of parentheses around the expression:

```
[[assume(expr1, expr2)]];  // Error
[[assume((expr, expr2))]]; // OK
[[assume(x = 1)]];         // Error
[[assume(x == 1)]];        // OK
[[assume((x = 1))]];       // OK
```

If we were to allow writing `[[assume(expr1, expr2)]]`, a user might erroneously read this as "*expr1* and *expr2* are both assumed", whereas in reality, only *expr2* is assumed. What this code is actually saying is "assume *expr2* after *expr1* has been evaluated just for its side effects". Since assumed expressions are not actually evaluated, reasoning about side effects can get confusing (see discussion in section 4.5) and such assumptions should be used with special care. It is therefore preferable to make this more explicit and more difficult to spell by requiring an extra pair of parentheses.

In order to disallow top-level commas, previous revisions of this paper went down one level in the *expression* grammar production, and used the *assignment-expression*. However, in [P2507R1], it was argued that we should go down one more level and require the expression to be a *conditional-expression*, to additionally exclude *assignment-expressions*[3] such as `[[assume(x = 1)]]`. EWG voted in favour of the direction of [P2507R1], so we adopted this change in R7.

We are currently not aware of any use case where assuming the result of an assignment expression like `[[assume(x = 1)]]` would be useful[4]: it would most likely be a typo for `[[assume(x == 1)]]`. So excluding it is the safer choice. If the user really wants to assume the converted `bool` value of an assignment, they can always do so by adding an extra pair of parentheses.

Existing practice on allowing top-level commas is inconsistent. Clang does not accept them in its `__builtin_assume`, while MSVC and ICC do accept them in their `__assume`. However, according to Gabriel Dos Reis, "the 'acceptance' by MSVC is a parser accident – don't use it as existing practice to standardise", and we decided to follow his advice. Assumptions with *assignment-expressions* are allowed by today's compiler built-ins: `__builtin_assume(x = 1)` will compile on Clang, and `__assume(x = 1)` will compile on both MSVC and ICC. However, after speaking with compiler implementers, it seems that this is more coincidental than intentional, similar to top-level commas.

4.2 The expression is not evaluated

The expression inside an assumption is unevaluated, like for example the operand of `decltype`. This is a fundamental property of assumptions and followed by all existing assumption built-ins. The expression is assumed without checking it.

Expressions with side effects are allowed inside an assumption, but any such side effects will not be executed and will not affect the behaviour of the program. This is compatible with both the semantics of attributes in C++ and the semantics of existing assume built-ins (for an in-depth discussion of assumptions with side effects, see section 4.5).

GCC is currently the only major compiler that doesn't have an assumption built-in and therefore doesn't provide a way to express assumptions with unevaluated expression semantics. In GCC, we currently have to emulate assumptions like this:

```
if (expr) {} else { __builtin_unreachable(); }
```

which evaluates `expr`. However, there is an easy path for GCC to implement conforming assumption semantics using their existing facilities. The strategy is as follows. First, it can check whether `expr` can have side effects if evaluated (GCC has a facility for this). If it can prove that it cannot, it means that evaluation of the expression won't affect the observable behaviour of the program. Under the as-if rule, it can then express the assumption in terms of its existing `__builtin_unreachable()`. Instructions emitted for evaluating the expression will typically be optimised out again. If it cannot prove that `expr` is free of side effects, it can simply ignore the assumption.

[3] Note that the other possible grammar productions under *assignment-expression*, *throw-expression* and *yield-expression*, are already excluded due to the requirement that the expression shall be contextually convertible to `bool` and that the expression shall appear only within a suspension context of a function, respectively.

[4] `__assume(x = 1)` is a tautology, since evaluating this expression would always return `1`, and therefore equivalent to `__assume(true)`, i.e. a null statement. Conversely, `__assume(x = 0)` is equivalent to `__assume(false)`, which in turn is equivalent to `__builtin_unreachable()`. In both cases, `x` is not actually being modified.

Ignoring assumptions altogether is also a conforming implementation. A trivial implementation of assumptions is therefore possible on any C++ compiler. The only requirement is that the assumed expression is checked for well-formedness (see section 4.7).

4.3 Assumptions that would not evaluate to `true` cause undefined behaviour

The expression inside the argument clause of an assumption is not evaluated, however the optimiser may analyse it, and deduce information from that analysis that it can use to optimise the program. The crucial property of an assumption is that if the expression would evaluate to `true` at the point where the assumption appears, the assumption has no effect, otherwise the behaviour is undefined. This gives the compiler the freedom to optimise away any code path that could be reached if the assumption were not `true`. This includes so-called "time travel" optimisation. Consider the following function (example from [P2064R0]):

```
int f(int j) {
  int i = 42;
  if (j == 0)
    i = 0;

  [[assume(j != 0)]];
  return i;
}
```

The proposed semantics allow the optimiser to assume that `j != 0` was already true before the code reached the assumption, since `j` was not modified. It can therefore remove the branch before the assumption, and reduce the whole function to `return 42`. This is merely specifying existing practice: both GCC and Clang actually perform this optimisation.

Because the expression is never evaluated, it is never checked. This is a common misconception about the semantics of assumptions. The implementation will not try to determine whether or not the expression would evaluate to `true`; there is no "hypothetical evaluation" of the unevaluated expression or anything along those lines. Instead, it will *assume* that the expression would evaluate to `true` at the point where the assumption appears, and optimise based on that assumption.

There is a subtle difference between behaviour being undefined if the expression would evaluate to `false`, or if the expression would *not* evaluate to `true`. The latter (proposed here) also includes the assumption that the expression would actually return a value, not throw an exception, and not exhibit undefined behaviour if it were evaluated. This enlarges the space of assumptions that can be stated by the programmer. Undefined behaviour inside the assumed expression is therefore effectively allowed to escape the assumption, despite the fact that the expression is not evaluated.

4.4 Assumptions ODR-use their argument

At first glance, this requirement seems unnecessary. If the argument of an assumption is not evaluated, only analysed, why would we want to specify that it is ODR-used? ODR-use means that, even if the assumption is otherwise ignored, the assumed expression can trigger template instantiations and lambda captures, which is observable both at compile time and at runtime. For example, the following two functions will return two different values:

```
constexpr auto f1(int i) {
  return sizeof( [=] { [[assume(i == 0)]]; } );
}

constexpr auto f2(int i) {
  return sizeof( [=] {} );
}
```

This is due to the fact that all existing implementations of assumptions require ODR-use. Implementing a useful assumptions facility without ODR-use of the argument would be extremely difficult, if not impossible. MSVC, ICC, and Clang all follow the same basic principle to implement assumptions. The compiler generates intermediate representation for the expression inside the assumption (which requires ODR-use). This code is then used during optimisation of the program. At a later stage of the optimiser, the assumption-related code is then stripped out again (the exact mechanics of this vary from compiler to compiler).

The observability of ODR-using assumed expressions can easily be demonstrated with today's compilers. If we replace `[[assume]]` in the code example above with the `ASSUME` macro from section 1 and compile this program, then on all of MSVC, ICC, GCC, and Clang, `f1` will return `4`, while `f2` will return `1`, both during constant evaluation and at runtime.

In practice, this should not have any negative impact. This observability of the `[[assume]]` attribute is not categorically different from the observability of the existing standard attribute `[[no_unique_address]]`. In both cases, the effect can be observable, but the semantics of the program do not actually change, as the standard does not mandate any particular value for `sizeof`. However, in the `[[no_unique_address]]` case, this observability will be much more frequent, as it's literally designed to modify the memory layout of a class, while for `[[assume]]`, code that would trigger template instantiations or lambda captures would be a very odd usage of assumptions for which no known use cases exist. Moreover, "stateful metaprogramming" that depends on template instantiations or the size of a lambda type is discouraged in general.

Note also that ODR-using the expression means you cannot use functions in an assumption that have a declaration but no definition.

4.5 Semantics of side effects

Assuming expressions with side effects is occasionally useful (consider `[[assume(++ptr != end)]]`). MSVC, Clang, and ICC all allow to write such assumptions, and at least MSVC uses them for some optimisations. But at first glance, it does not seem obvious how to formally define the semantics of such an assumption in terms of the C++ abstract machine. If `ptr` is not actually incremented at the point where the assumption occurs, how can we reason about a counterfactual world in which `ptr` is incremented, and make assumptions about the program (in which `ptr` is not incremented at that point) based on that? It seems that we would need to introduce some novel concept of "hypothetical evaluation" of an unevaluated expression in the standard, requiring herculean efforts.

As we will show below, in fact no such thing is needed to understand the semantics of assumptions. We begin by categorising all expressions that could appear in an assumption into three categories:

— Category 1. The assumed expression has no side effects when evaluated.

— Category 2. The assumed expression may have side effects, but they are deterministic.

— Category 3. The assumed expression may have non-deterministic side effects.

Let us now discuss the semantics of each category.

Category 1. This is the most common type of assumptions and the most straightforward. Consider the following minimal example:

```
int f(int i) {
  [[assume(i == 42)]];
  return i;
}
```

The implementation can assume that `i == 42` evaluates to `true`, and optimise based on this assumption. The evaluation of this expression has no side effects, therefore it doesn't actually matter if the expression is evaluated: any instructions emitted for such an evaluation won't affect the observable behaviour of the program and can be optimised away afterwards. This is the only category of assumptions that can be emulated by GCC's `__builtin_unerachable()` and similar constructs. An implementation is allowed to either ignore the assumption, or optimise `f` as follows:

```
int f(int i) {
  return 42;
}
```

Category 2. Let us now consider the following, slightly different example:

```
int f(int i) {
  [[assume(++i == 43)]];
  return i;
}
```

This assumption has a side effect: evaluating the expression would modify `i`. The implementation is not allowed to do this. However, it is allowed to "analyse the form of the expression and deduce information used to optimise the program". Note that since the semantics of integer increment are known and deterministic, the statement `++i == 43`, which would have side effects if evaluated, can be transformed to an equivalent statement that does *not* have side effects if evaluated: `i == 42`. Assuming this new statement is equivalent to assuming the original statement. This program is therefore equivalent to the one in the previous example.

In other words, the statement about a hypothetical program in which `i` would be incremented can be reduced to a statement about the real program, in which `i` is not being incremented, at the point where the assumption occurs. The resulting statement is a Category 1 assumption, which has well-defined semantics. Since we know that the side effects are deterministic, such a reduction to a Category 1 assumption is always possible, at least theoretically.

Of course, real-world compilers won't be able to perform the required transformation in all cases. This is not a problem, since a compiler is allowed to just ignore the assumption if it cannot derive any useful information from it.

Category 3. Let us now consider the following pathological example (from Martin Uecker and Aaron Ballman):

```
int f(int i) {
  [[assume((std::cin >> i, i == 42))]];
  std::cin >> i;
  return i;
}
```

Of course, nobody should ever write such an assumption, as it obviously does not express an invariant of the program and therefore cannot serve a useful purpose. But nevertheless we need to be able to determine what assumption this code expresses and what semantics it has. The crucial difference to a Category 2 assumption is that the input value received from `std::cin` is non-deterministic. We cannot determine whether the assumption holds and the program is well-defined by analysing the expression, only by actually calling `std::cin`, but we are not allowed to do that, since an assumed expression is unevaluated.

At first glance, this may seem like a paradox, and various contradicting interpretations seem possible, including:

— There is no useful information that can be derived from this assumption, therefore it should have no effect. The compiler must translate the program as written ignoring the assumption.

— The compiler cannot actually call `std::cin` inside the assumption, since assumptions are unevaluated. It is therefore impossible to determine what the value of the expression would be. Since it does not “evaluate to `true`”, the program is undefined behaviour, and the compiler is allowed to optimise out the whole function `f` and all code paths leading to it.

— The compiler can assume that a call to `std::cin` at the point of the assumption would read the number 42. Since there is no change in program state between this point and the point where `std::cin` is actually called (on the next line), the compiler is allowed to optimise out the call to `std::cin` and replace the code with `int f(int) { return 42; }`

It seems that we cannot answer which of these interpretations is correct without specifying the semantics further, in particular without specifying what it means to have an unevaluated expression whose value nevertheless affects the program semantics. However, as it turns out, this is not necessary. It is actually straightforward to reason about assumptions, using the specification in this proposal, as soon as we give up this idea of “hypothetical evaluation”. This is fundamentally the wrong mental model to reason about assumptions. Remember that the expression is never checked, only assumed, therefore there is no need to determine what it would evaluate to.

If we follow the correct reasoning, it turns out that actually all three of the above interpretations are incorrect. It goes as follows.

First of all, note that we do not need to consider the behaviour on any system where the above assumption does not hold, since the specification does not put any constraints on the behaviour of such a system. Therefore, we only need to consider systems where the assumption *does* hold, i.e. where `std::cin`, if executed at the place where the assumption appears, would always read in the number 42. This can be, for example, a computer controlled by a robot which is programmed to always enter the number 42 when prompted. On this system, the assumption is doing exactly what it is intended to do: it expresses an invariant of the system[5] (the robot will always type 42) which the C++ compiler cannot see (as it is unaware of the robot).

Now, on such a system where the assumption holds, its semantics are precisely defined: it has no effect. The program therefore *must* call `std::cin`, as this is an observable side effect; the compiler is not allowed to optimise out the call. However, under the as-if rule, it is allowed to throw out the value read by `std::cin`, as it knows that it will always be 42. Therefore, the compiler can either ignore the assumption, or optimise `f` as follows:

```
int f(int) {
  int tmp;
  std::cin >> tmp;
  return 42;
}
```

In other words, any Category 3 assumption (i.e. an assumption containing a non-deterministic expression) can be reduced to a Category 2 assumption by considering only systems where the assumption expresses an actual, real invariant of the program (because that is the only thing that assumptions are ever allowed to express). Therefore, the expression is not actually non-deterministic. On such systems, the assumption has no effect, making the semantics of the program well-defined. On all other systems, the behaviour is undefined.

To give yet another example of assumptions with side effects, let us consider the following code (from Gašper Ažman):

```
int f(ForwardIterator auto almost_last, ForwardIterator auto last) {
  [[assume(++almost_last == last)]];
  // do something...
}
```

[5] Although it is not actually useful even on such a system: a more efficient approach is to simply not read the input at all, since we already know the result.

Let us start by categorising this assumption as above. The first question is whether incrementing the forward iterator and then comparing it to the other iterator is deterministic.

If ForwardIterator is e.g. `std::forward_list<int>::iterator`, the expression is deterministic and the assumption is therefore in Category 2. Compared to our previous Category 2 example `[[assume(++i == 43)]]`, we now cannot easily derive an equivalent side-effect-free equality expression like `[[assume(i == 42)]]` by reversing the increment, because `operator++` on a ForwardIterator is not reversible. However, we are never incrementing it in the first place, as the assumed expression is not evaluated (not even "hypothetically evaluated"), only analysed. We could therefore perform the reduction to Category 1 by transforming the assumption into a side-effect-free statement about `almost_last` and `last`, such as: if `node` is the linked list node associated with the object that `almost_last` points to, then we can assume `node->next == last`. It doesn't matter if `node` is an implementation detail of `std::forward_list`, `.next` is a private member, and `f` isn't allowed to access them: the expression is not being evaluated, only analysed, and the implementation can analyse whatever it chooses.

If on the other hand, ForwardIterator is e.g. `std::istream_iterator<char>`, then the behaviour is non-deterministic, we are in Category 3, and we can apply the same reasoning as in the `std::cin` example (and the assumption is most likely nonsensical).

It is important to remember that the above reasoning only serves to understand the semantics of expressions inside assumptions as formally defined in the proposed wording. In practice, the compiler is allowed to use a completely different strategy, including simply discarding the assumption, as long as it is compatible with these semantics.

4.6 Behaviour of assumptions during constant evaluation

What should happen if an assumption is encountered during constant evaluation? This is unlikely to occur in practice, since assumptions are inherently a run-time utility, but for completeness' sake we need to specify this as well. Consider the following code:

```
constexpr int f() {
  return 0;
}

constexpr int g() {
  [[assume(f() == 1)]];   // assumption doesn't hold
  return 1;
}

int main() {
  return g();
}
```

We propose that, if such an assumption would not evaluate to `true`, it is implementation-defined whether the program is ill-formed or not. This way, we leave freedom for implementations to conduct such an analysis at compile time and emit a compiler error for a failed assumption (which can be useful), while not requiring an implementation to do so (because it might be difficult to implement for all cases, and currently none of MSVC, GCC, or Clang implement this check with `__assume` and `__builtin_assume`, respectively: the code above passes on all of them).

If an assumption holds during constant evaluation, this should have no effect.

Another subtlety is the question what should happen if inside a `constexpr` function we encounter an assumption that would evaluate to `true`, but can not be evaluated during constant evaluation? Currently, there is implementation divergence. MSVC rejects the following code when using its assumption built-in instead of the attribute, while ICC and Clang accept it:

```
int foo() {        // not a constexpr function
  return 0;
}

constexpr int bar() {
  [[assume(foo() == 0)]];  // this assumption holds but isn't constexpr
  return 1;
}

int main() {
  return bar();
}
```

We propose that this code should be well-formed. If an assumption cannot be checked at compile time, the assumption should simply be ignored, rather than making the whole program ill-formed. Otherwise, in order to be able to make the function `constexpr`, the user would have to branch on `std::is_constant_evaluated()` just for the purpose of using such an assumption, which does not seem reasonable.

In order to fully specify the behaviour of assumptions during constant evaluation, we need to distinguish the following cases:

1. Would evaluate to `true` following the constant evaluation rules
2. Would evaluate to `false` following the constant evaluation rules
3. Would not evaluate to either `true` or `false` following the constant evaluation rules because of undefined behaviour, non-termination, etc.
4. Doesn't satisfy the constant evaluation rules but is still somehow known to evaluate to `true`.
5. Doesn't satisfy the constant evaluation rules but is still somehow known to evaluate to `false`.
6. Undecidable during constant evaluation what it would evaluate to or whether evaluation would succeed due to incomplete information (eg, using runtime state or calling a function from a different translation unit).

To be consistent with the design described above, we propose that cases 1, 4, 5, and 6 should be well-formed, while for cases 2 and 3, the behaviour should be implementation-defined.

It seems desirable to make case 4 well-formed, but case 5 ill-formed or implementation-defined. However, it is not clear how we could even specify when something is "still somehow known to evaluate to" `true` or `false` and normatively distinguish these cases. It is much clearer (and also consistent) to specify that an assumption should simply be ignored whenever it cannot be evaluated following the constant evaluation rules.

4.7 Do ill-formed expressions need to be diagnosed?

The C++ standard specifies that attributes not recognised by an implementation can be ignored. However, it is not entirely clear whether this also extends to attributes that are part of the C++ standard itself. For the latter, the standard imposes constraints on both the argument clause of the attribute (e.g. `[[noreturn]]` shall have none, `[[deprecated]]` can have one optionally but it must be a string literal, and so on) and what entities the attribute may appertain to.

There is currently some debate about what the standard says should happen if these constraints are violated. One possible interpretation is that in this case, the program is ill-formed and the compiler must issue a diagnostic, since the constraints are imposed normatively in wording, for example for `[[deprecated]]`: "an *attribute-argument-clause* may be present and, if present, it shall have the form...".

Another possible interpretation is that the compiler is permitted to completely ignore an attribute even if the constraints are violated, since the wording says "For an *attribute-token* [...] not specified in this document, the behavior is implementation-defined. Any *attribute-token* that is not recognized by the implementation is ignored". It is ambiguous whether the second sentence refers to all *attribute-token*s, or only the non-standard ones. Different WG21 members and compiler implementers have voiced diverging opinions on this matter. A currently open Core issue [CWG2538] tries to address this ambiguity.

Since assumptions are attributes, the same ambiguity applies here. However, since the arguments of assumptions can be arbitrary C++ expressions (and not just string literals like for existing standard attributes), the implications are more severe. A conforming compiler doesn't have to implement an assumption facility, and is free to ignore a well-formed assumption. However, if the `assume` attribute is written in the wrong place, or doesn't have an expression as its argument, or the expression is not contextually convertible to `bool` or ill-formed, the compiler might still have to detect this and issue a diagnostic, which involves fully parsing the expression. Further, an assumed expression is ODR-used, which can trigger template instantiations. If any of these instantiations makes the program ill-formed, for example by containing a `static_assert` that does not evaluate to `true`, this might have to be diagnosed as well.

It's however important to point out that the question of whether or not ill-formed standard attributes need to be diagnosed has nothing to do specifically with this proposal, but concerns the design space of C++ standard attributes in general. There are other proposals for standard C++ currently in flight that use expressions inside attributes, such as `[[trivially_relocatable(expr)]]` [P1144R5], which are affected to the same degree by this question. The ignorability question also came up in SG22 when discussing an attribute-like syntax with double square brackets for contracts.

We therefore decided to separate out the discussion about this ambiguity into a separate paper [P2552R0]. That paper deals specifically with specifying more precisely to what degree and under what circumstances C++ standard attributes should be "ignorable" and what this means exactly. This is independent from the question of how to specify assumptions.

4.8 Pack expansion

The grammar for C++ attributes allows an attribute to be followed by an ellipsis. [dcl.attr.grammar] specifies: "In an attribute-list, an ellipsis may appear only if that attribute's specification permits it. An attribute followed by an ellipsis is a pack expansion."

We could therefore hypothetically permit the `assume` attribute to directly support pack expansion:

```
template <int... args>
void f() {
    [[assume(args >= 0)...]];
}
```

However, we do not propose this. It would require substantial additional work for a very rare use case. Note that this can instead be expressed with a fold expression, which is equivalent to the above and works out of the box without any extra effort:

```
template <int... args>
void f() {
    [[assume(((args >= 0) && ...))]];
}
```

4.9 Appertaining to non-null statements

In theory, we do not have to limit assumptions to appertain to a null statement, and could allow them to appertain to other statements, for example such that it would be well-formed to write

```
[[assume(x >= 0)]] f(x);
```

However, we do not propose this, as there is no existing practice for it: all existing assumption built-ins can only be used as a single statement followed by a semicolon. We are also not aware of any use case that could not be spelled equivalently with the syntax proposed here.

5 History and related work

5.1 N4425 and pre-C++20 contracts proposals

Adding portable assumptions was already proposed in [N4425][6] and discussed by EWG in 2015 in Lenexa[7]. The paper was rejected. EWG's guidance was that this functionality should be provided within the proposed contracts facility, and not as a separate feature.

Ironically, contracts as merged into the C++20 working draft in June 2018 in Rapperswil [P0542R5], actually failed to provide the functionality of assumptions [P1773R0]. And later, in July 2019 in Cologne, contracts were pulled from C++20 altogether. Progress on assumptions had been blocked for no good reason at all.

5.2 Current work on contracts

More recent proposals for adding contracts to C++ [P2388R4], [P2461R1], [P2487R0] no longer include the possibility to assume contracts for purposes of optimisation. Contracts will also need more development time. Assumptions are useful, well-understood, existing practice, and we should standardise them now, rather than waiting for progress on contracts.

Contracts and assumptions are very different features. The purpose of contracts is to find and avoid bugs, and to document pre- and postconditions in code; they are meant to be used at API boundaries; they are primarily targeting the front-end of the compiler (or a static analyser); and they are a "cross-cutting" feature that is meant to be used widely throughout a codebase by many developers. By contrast, the purpose of assumptions is to make specific invariants of your code visible to the optimiser; they are meant to be an implementation detail; they are primarily targeting the back-end of the compiler; and they are a "local" feature that will only be used rarely, at specific locations in performance bottlenecks, and by experts only.

Further, the expressions that are typical for assumptions tend to look very different from the ones typically found in contracts. Assumptions are practically always either statements about a `bool`, or very simple mathematical expressions involving a single number or pointer. By contrast, contract preconditions and postconditions can contain significantly more complicated statements about the program, even including lambdas.

Standardising the existing practice of a low-level assumptions facility that is independent of contracts is not closing off future work. In case contracts or other higher-level features will incorporate assumptions in some form in the future, this can be specified and implemented using the low-level facility proposed here as a building block.

[6] The syntax proposed then was different: `true(`*expr*`)` and `false(`*expr*`)`, but the semantics were essentially the same as in this proposal.

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

5.3 Assertions vs. assumptions

Assertions (whether as a subset of contracts or as a standalone feature) and assumptions are fundamentally different in nature. We are not aware of any study that could conclusively show that there is a measurable performance benefit from turning assertions into assumptions throughout a codebase. [P2064R0] found that it actually degrades performance, while [Amini2021] found that it makes no statistically significant difference at all. There are cases where injecting (formally correct) assumptions can actually degrade performance, which is also true for other C++ features interacting with the optimiser such as `[[likely]]` and `[[unlikely]]`.

Therefore, we should not combine assertions and assumptions in the same language feature, we should make the syntax of assertions look different from assumptions, and we should especially not introduce a generic way to assume assertions. Instead, we should use assumptions explicitly in the few cases where it provably matters for performance. Assertions, on the other hand, should be a "safe-to-use" feature that primarily exists to find and avoid bugs. They should not be able to degrade performance of optimised code or inject undefined behaviour and "time travel" into an otherwise valid program.

For a much more detailed discussion of assertions vs. assumptions, see [P2064R0].

5.4 `std::unreachable`

[P0627R6] is a related paper proposing a function `std::unreachable()`, standardising GCC's `__builtin_unreachable()`: a function that has undefined behaviour when called, and therefore can be used to mark unreachable code paths.

It is important to recognise that the functionality provided by `std::unreachable()` is a strict subset of the functionality provided by assumptions as proposed here. `std::unreachable()` has the exact same semantics as `[[assume(false)]]`. Assuming an expression without side effects can be expressed with either `assume` or `std::unreachable` (although the latter is significantly more verbose), while assuming an expression with side effects can only be expressed with `assume`. Therefore, `assume` is the more general feature, and the one that should be standardised first.

That being said, the possibility to spell `[[assume(false)]]` as `std::unreachable` might still be desirable. If what the user wants to do is to mark unreachable control flow (unreachable branches, unreachable switch cases etc.), for example to avoid compiler warnings, then the spelling `std::unreachable` better communicates that intent. We therefore do not see a problem with both features coexisting.

6 Previous polls

Below are the polls taken by WG21 subgroups on previous revisions of this paper, in chronological order.

EWG, Belfast (November 2019)

1.. P1774 with `[[assume(expr)]]` syntax.

SF	F	N	A	SA
15	5	1	0	0

2. P1774 with `std::assume(expr)` syntax.

SF	F	N	A	SA
1	3	4	10	4

EWG, Prague (February 2020)

1. We want assumptions now and independent of future contract facilities.

SF	F	N	A	SA
18	5	1	3	3

2. We like the proposed semantics for assumptions.

SF	F	N	A	SA
18	5	4	2	0

3. We want exploration on a mode which can check assumptions, including side effects.

SF	F	N	A	SA
1	0	9	9	5

4. We like the proposed attribute syntax `[[assume(expr)]]`

SF	F	N	A	SA
9	8	5	5	1

5. We'd like more exploration on macro assume, like assert

SF	F	N	A	SA
0	0	1	10	16

6. We'd like more exploration on keyword such as one of `unsafe_assume` / `assume` / `__assume` / `_Assume` / . . .

SF	F	N	A	SA
5	7	9	5	2

7. We'd like more exploration on magic library function such as `std::assume(expr)`.

SF	F	N	A	SA
0	0	0	9	14

SG21, Prague (February 2020)

Assumptions should proceed independently of contracts.

SF	F	N	A	SA
9	8	5	6	5

EWG, online telecon (2021-12-02)

1. In D1774R5, we should spell the assume as `[[assume: expr]]`.

SF	F	N	A	SA
0	0	1	12	5

Consensus against

2. In D1774R5, we prefer assume's parameter to be just an "attribute-grammar-conforming token soup", not an expression.

SF	F	N	A	SA
0	0	2	8	6

Consensus against

3. Send D1774R5 to electronic polling for forwarding to CWG for inclusion in C++23, in Bucket 2.

SF	F	N	A	SA
6	8	5	0	0

Consensus

EWG electronic poll (January 2022)

Forward P1774R5 "Portable assumptions" to Core for C++23.

SF	F	N	A	SA	
14	17	3	4	0	Consensus

Abstain: 7

SG22, online telecon (2022-02-11)

Does SG22 encourage proposing the functionality in P1774R5 to WG14?

Committee	For	Against	Abstain	
WG14	6	1	1	Consensus
WG21	8	1	2	Consensus

EWG, online telecon (2022-04-14)

EWG would like P1774 to be modified with the changes proposed in P2507R1 before inclusion into C++23.

SF	F	N	A	SA	
4	6	6	0	0	Consensus

7 Proposed wording

The proposed changes are relative to the C++ working draft [N4910].

Add the following sub-clause to [dcl.attr]:

Assumption attribute [dcl.attr.assume]

The *attribute-token* `assume` may be applied to a null statement; such a statement is an assumption. An *attribute-argument-clause* shall be present and shall have the form:

`(` *conditional-expression* `)`

The expression is contextually converted to `bool` [conv.general]. The expression is not evaluated. If the converted expression would evaluate to `true` at the point where the assumption appears, the assumption has no effect. Otherwise, the behavior is undefined.

[*Note:* The expression is potentially evaluated [basic.def.odr]. The use of assumptions is intended to allow implementations to analyze the form of the expression and deduce information used to optimize the program. Implementations are not required to deduce any information from any particular assumption. — *end note*]

[*Example:*

```
int divide_by_32(int x)  {
  [[assume(x >= 0)]];
  return x/32;    // The instructions produced for the division
                  // may omit handling of negative values
}

int f(int y) {
  [[assume(++y == 43)]];     // y is not incremented
  return y;                  // Statement may be replaced with return 42;
}
```

— *end example*]

Modify [expr.const] as follows:

> An expression E is a *core constant expression* unless the evaluation of E, following the rules of the abstract machine ([intro.execution]), would evaluate one of the following:
>
> — an operation that would have undefined behavior as specified in [intro] through [cpp], excluding [dcl.attr.assume];
>
> It is unspecified whether E is a core constant expression i~~I~~f E satisfies the constraints of a core constant expression, but evaluation of E would evaluate
>
> — an operation that has undefined behavior as specified in [library] through [thread],~~ or~~
>
> — an invocation of the `va_start` macro ([cstdarg.syn]), or
>
> — a statement with an assumption ([dcl.attr.assume]) whose converted *conditional-expression*, if evaluated where the assumption appears, would not disqualify E from being a core constant expression and would not evaluate to `true`.
>
> [*Note:* E is not disqualified from being a core constant expression if the hypothetical evaluation of the converted *conditional-expression* would disqualify E from being a core constant expression. — *end note*]
>
> ~~it is unspecified whether `e` is a core constant expression.~~

Document history

- **R0**, 2019-06-17: Initial version.
- **R1**, 2019-10-06: Updated text to reflect removal of Contracts from C++20; made proposed attribute syntax backwards-compatible by replacing colon with parentheses.
- **R2**, 2019-11-25: Changed title to “Portable assumptions”; changed semantics from UB if expression would evaluate to `false` to UB if expression would *not* evaluate to `true`; changed syntax section to propose attribute-syntax only, dropping “magic” library function syntax as a viable alternative.
- **R3**, 2020-01-13: Updated text to clarify the discussion of the proposed semantics and syntax.
- **R4**, 2021-11-15: Added wording. Added polls. Added code size measurement results. Updated and restructured text, adding discussion of proposed semantics and recent related work.
- **R5**, 2021-12-09: Updated wording (removed feature-test macro, allowed duplicate attributes, added clarifications). Updated and restructured text, expanding semantics section to reflect discussion in EWG and on the WG21 reflectors. Added Peter Dimov’s refcounted smart pointer code example.
- **R6**, 2022-02-15: Updated polls section; minor fixes.
- **R7**, 2022-05-15: Changed *assignment-expression* to *conditional-expression* following EWG guidance; updated wording following CWG review; expanded discussion of different cases during constant evaluation; updated discussion of whether ill-formed expressions inside assumptions need to be diagnosed; added example of `[[assume]]` triggering a lambda capture; updated polls section.

Acknowledgements

Many thanks to Herb Sutter, Chandler Carruth, Joshua Berne, Michael Spencer, Jonathan Caves, Hal Finkel, Erich Keane, Judy Ward, Inbal Levi, Eric Brumer, Nathan Sidwell, Daveed Vandevoorde, Jens Maurer, Gašper Ažman, Peter Dimov, Gabriel Dos Reis, Arthur O'Dwyer, Aaron Ballman, Martin Uecker, Peter Brett, and Richard Smith for their help with this proposal.

References

[Amini2021] Parsa Amini. Asserting Your Way To Faster Programs. CppCon talk, 2021-10-28.

[CWG2538] Jens Maurer. Core issue 2538. Can standard attributes be syntactically ignored? https://cplusplus.github.io/CWG/issues/2538.html, 2022-03-26.

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

[N4910] Thomas Köppe. Working Draft, Standard for Programming Language C++. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/n4910.pdf, 2022-03-17.

[P0542R5] G. Dos Reis, J. D. Garcia, J. Lakos, A. Meredith, N. Myers, and B. Stroustrup. Support for contract based programming in C++. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0542r5.html, 2018-06-08.

[P0627R6] Melissa Mears and Jens Maurer. Function to mark unreachable code. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p0627r6.pdf, 2021-10-15.

[P1144R5] Arthur O'Dwyer. Object relocation in terms of move plus destroy. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1144r5.html, 2020-03-01.

[P1773R0] Timur Doumler. Contracts have failed to provide a portable "assume". http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1773r0.pdf, 2019-06-17.

[P2064R0] Herb Sutter. Assumptions. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2064r0.pdf, 2020-01-13.

[P2388R4] Andrzej Krzemieński and Gašper Ažman. Minimum Contract Support: either *No_eval* or *Eval_and_abort* contracts. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2388r4.html, 2021-11-15.

[P2461R1] Andrzej Krzemieński. Attribute-like syntax for contract annotations. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2487r0.html, 2021-11-12.

[P2487R0] Gašper Ažman, Caleb Sunstrum, and Bronek Kozicki. Closure-Based Syntax for Contracts. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2461r1.pdf, 2021-11-15.

[P2507R1] Peter Brett. Limit `[[assume]]` to conditional-expressions. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2507r1.pdf, 2021-12-15.

[P2552R0] Timur Doumler. On the ignorability of standard attributes. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2552r0.pdf, 2022-02-15.

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