Portable assumptions

Timur Doumler (papers@timur.audio)

Document #:	P1774R4
Date:	2021-11-15
Project:	Programming Language C++
Audience:	Evolution Working Group

Abstract

We propose a standard facility providing the semantics of existing compiler intrinsics 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 (from [Regehr2014]):

```
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 (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
```

All major compilers offer this functionality by providing the following built-ins (see [N4425] and Section 3 below for a more thorough discussion):

— MSVC and ICC have `__assume(`*`expr`*`);`

— Clang has `__builtin_assume(`*`expr`*`);`

— GCC has `__builtin_unreachable()`, which works slightly differently. Assuming an expression can be written as

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

However, this has slightly different semantics: on GCC, *`expr`* will be evaluated, at least notionally. In practice, this will often be optimised away if evaluating *`expr`* has no side effects.

Assumptions are a useful expert-level feature, and they are existing practice. However, they are spelled differently on every compiler and subject to subtle differences and ambiguities in semantics, which are not properly defined anywhere. The goal of this proposal is to introduce a unified syntax and well-defined semantics for assumptions in a way that is compatible with all existing compiler implementations and fits well into the existing C++ standard.

2 Syntax

2.1 Proposed

We propose an attribute syntax to spell portable assumptions. `__builtin_assume(`*`expression`*`)` instead becomes:

```
[[assume(expression)]]
```

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.

The syntax above (using parentheses) is chosen such that it is fully compatible with standard attribute syntax and therefore backwards-compatible with a compiler that does not support this feature.

Making this an attribute also 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.

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]]`).

We further believe that this syntax has the least impact on the core language as opposed to the alternatives (see below).

Finally, the attribute syntax would also allow to add this feature to the C language with the same spelling.

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 asuumes 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.

2.2 Alternatives considered (not proposed)

2.2.1 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 new syntax to C++ over using existing attribute syntax. New syntax would require otherwise unnecessary changes to the C++ grammar, making it harder to add assumptions to existing code due to lack of backwards-compatibility. At the same time, it does not provide any benefits over the attribute syntax we propose.

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

2.2.2 Keyword

An assumption can be characterised as an operator with an unevaluated operand, somewhat similar to `decltype(expression)`, where `decltype` is a keyword. 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.

2.2.3 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.

2.2.4 "Magic" library function

One option that at first glance seems very attractive is to spell an assumption as a "magic" library function:

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

Herb Sutter [P2064R0] and others have 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 operand. 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 operand 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++).

3 Semantics

We collaborated 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.

3.1 Assumptions do not evaluate their operand

The expression inside an assumption is an unevaluated operand, like for example the operand of `decltype`. This is a fundamental property of assumptions. The whole point is to assume the operand without checking it (see also section 5.1, and Herb Sutter's discussion in [P2064R0] why assumptions are fundamentally different from assertions).

Therefore, expressions with side effects are allowed, which is useful (consider `[[assume(++ptr != end)]]`), and there is a guarantee that those side effects will not be executed and will not affect the behaviour of the program. This is consistent with both the semantics of attributes in C++ and the semantics of existing `__assume` and `__builtin_assume`.

GCC is currently the only major compiler that doesn't have an intrinsic with the unevaluated operand semantics. In GCC, we have to spell assumptions like this:

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

which evaluates `expr`. However, GCC can implement assume in a way conforming to the semantics proposed here relatively easily, by employing the following strategy. First, we check whether `expr` can have side effects if evaluated. If we can prove that it cannot, we can use the existing GCC mechanism to implement assumptions. Otherwise, we simply ignore the assumption. Ignoring assumptions with potential side effects is a conforming implementation of the semantics proposed here.

Ignoring assumptions altogether is also a conforming implementation. A trivial implementation of assumptions is therefore possible on any C++ compiler.

3.2 Assumptions that would not evaluate to `true` are undefined behaviour

The operand of an assumption is not evaluated, however the optimiser may analyse it, and deduce from that information used to optimise the program. The crucial property of an assumption is that if such analysis reveals that it would not evaluate to `true`, the behaviour is undefined. This gives the compiler the freedom to optimise away any code path that could be reached if the assumption would not hold. 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.

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 itself would not result in UB if it were evaluated. This enlarges the space of assumptions that can be stated by the programmer (Thanks to Joshua Berne for pointing this out). In other words, UB inside the assumed expression is allowed to escape the assumption, even if the expression is notionally unevaluated.

3.3 Assumptions ODR-use their operand

At first glance, this requirement seems unnecessary. If the operand of an assumption is not evaluated, why would we want to specify that it is ODR-used?

The answer is: ODR-use is what existing implementations do. Implementing assumptions without ODR-use of the operand would be extremely difficult. As far as we know, such an implementation does not exist.

MSVC, ICC, and Clang all follow the same basic principle to implement assumptions. The compiler actually 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).

3.4 Behaviour of assumptions during constant evaluation

What should happen if an assumption is encountered during constant evaluation? 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 very difficult to implement). 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, while GCC and Clang accept it (when using the appropriate platform-specific built-in for the assumption):

```
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.

4 Benchmarks

Since assumptions are not standardised yet, for this study we use an `ASSUME` macro instead of the syntax proposed above. The macro is defined as follows and works on all major compilers:

```
#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
```

As discussed above, there are differences in semantics across compilers, in particular whether or not `expr` is evaluated; however, by assuming only expressions without side effects we can make sure this has no impact on the benchmarks.

4.1 Code size

Consider looping over a range of `floats` and clamping all values between -1 and 1. This is an operation that often occurs 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 has invariants which are guaranteed by the surrounding code, but this information is invisible to the optimiser, for example because the code is too complex for the optimiser to see through, or because there is a TU boundary in between. 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);
    }
}
```

We have compiled both versions of this function on godbolt.org with MSVC, GCC, Clang, and ICC, respectively. We have used the latest trunk versions of these compilers at the time of writing, and compiled the code with the maximum optimisation setting (`-O3` and `/O2`, respectively). As a crude estimate of code size, we counted the number of instructions emitted, with the following results.

Compiler	Without `ASSUME`	With `ASSUME`
MSVC	88	52
GCC	77	42
Clang	74	20
ICC	62	62

On all compilers except ICC, using `ASSUME` leads to significantly less code being emitted. The injected assumptions lead to a better optimisation of the loop as well as elimination of unnecessary code inside `std::clamp`.

4.2 Execution time

Benchmarks comparing execution time of the same code compiled with and without assumptions are planned for the next revision of this paper.

5 History and related work

5.1 N4425 and Contracts

Adding portable assumptions 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 contracts facility, and not as a separate feature.

However, contracts as merged into the C++20 working draft in June 2018 in Rapperswil [P0542R5], actually failed to provide such portable optimisation hints [P1773R0]. Later, in July 2019 in Cologne, contracts were pulled from C++20 altogether. More recent proposals [P2358R0] for adding contracts to C++ no longer include the possibility to assume contracts for purposes of optimisation. These ideas will also take time to develop. 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; 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; and they are a "local" feature that will only be used rarely, at specific locations in performance bottlenecks, and by experts only.

More specifically, assertions and assumptions are fundamentally different in nature (see also [P2064R0]). 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. Therefore, we should not combine assertions and assumptions in the same language feature. Instead, we should use assumptions explicitly in the few cases where it actually matters for performance, while making sure that assertions are a "safe-to-use" feature that cannot inject undefined behaviour and "time travel" into a program.

We need a low-level assumptions facility that is independent of contracts or assertions. The present paper focuses on proposing exactly this low-level facility. In case contracts or other higher-level features will incorporate assumptions in some form in the future, they can then be specified and implemented in terms of this low-level facility.

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

5.2 `std::unreachable`

[P0627R6] is a related paper proposing a function `std::unreachable()`, standardising GCC's `__builtin_unreachable()` (see above): a function that invokes UB 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`, and at least one existing compiler (MSVC) is capable of using such assumptions for optimisations. 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. If we want to standardise `std::unreachable`, we should specify it in terms of assumptions as proposed here.

6 Previous polls

Below are the polls taken by WG21 subgroups on previous revisions of this paper.

6.1 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

6.2 SG21, Prague (February 2020)

Assumptions should proceed independently of contracts.

SF	F	N	A	SA
9	8	5	6	5

6.3 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

7 Proposed wording

Add the following subclause 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. The attribute-token `assume` shall appear at most once in each attribute-list. An *attribute-argument-clause* shall be present and shall have the form:

`(` *assignment-expression* `)`

The epression shall be contextually convertible to `bool`. The expression is not evaluated, however, it is potentially evaluated [basic.ref.odr]. If the expression would not evaluate to `true`, the behavior is undefined.

[*Note:* The use of assumptions is intended to allow implementations to analyze the form of the expression and deduce from that information used to optimize the program. — *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
}
```

— *end example*]

Modify [expr.const] as follows:

If `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] of this document, a statement with an assumption ([dcl.attr.assume]) that has a predicate that would not evaluate to `true`, or an invocation of the `va_start` macro ([cstdarg.syn]), it is unspecified whether `e` is a core constant expression.

Add to [cpp.predefined]/table 21:

`__cpp_assume` with the appropriate value.

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.

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, and Daveed Vandevoorde for their help with this proposal.

References

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

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

[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`, 2017 (accessed 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.

[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.

[P2358R0] Gašper Ažman, John McFarlane, and Bronek Kozicki. Defining contracts. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2358r0.pdf`, 2020-01-14.

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