Portable assumptions

Timur Doumler (papers@timur.audio)

Document #: P1774R3
Date: 2020-01-13
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, Intel). 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. This is existing practice, but it would be much more accessible and easy-to-use if it were a standardised, portable C++ facility.

1.1 History and context

Adding such portable optimisation hints 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.

Unfortunately, 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.

Regardless of whether contracts will eventually make it into a future C++ standard, and whether or not assumptions might be a feature of such contracts, we need an independent low-level "assume" facility. The present paper focuses on proposing exactly this low-level facility. Its purpose is to introduce a standard way to provide an optimisation hint to the compiler, as an implementation detail of a C++ program, locally, with clearly defined semantics that are expressed in C++ code, independent of any build modes, build flags, etc.

In case contracts or other higher-level features will make use of such assumptions in the future, it can then be implemented in terms of this low-level facility.

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

1.2 Existing practice

The major compilers offer the following built-ins providing this functionality:

— MSVC and Intel provide `__assume(`*`expression`*`);`

— Clang provides `__builtin_assume(`*`expression`*`);`

— GCC does not provide an analogous built-in. A similar effect can be achieved with `if (`*`expression`*`) {} else { __builtin_unreachable(); }`
However, an important difference is that here, unlike in the other cases, *`expression`* will be evaluated.

See [N4425] for a more thorough discussion.

1.3 Examples

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.

2 Proposed solution

2.1 Proposed semantics

The design goal is to provide a portable facility closely following the compiler built-ins `__assume` and `__builtin_assume`, therefore standardising existing practice. The facility should be implementable with the existing built-ins on those compiler implementations who have them, without unnecessarily constraining implementations who do not. Therefore, we propose the following semantics:

— It is a statement with a single argument, which is a C++ expression contextually convertible to `bool`.

— The expression is an unevaluated operand, like for example the operand of `decltype`. Therefore, expressions with side effects are allowed (which is useful, consider `++ptr != end`). However, as the expression is not evaluated, these side effects do not affect the behaviour of the program[2].

— However, the optimiser may analyse the form of the expression, and deduce from that information used to optimise the program.

— The behaviour is undefined if the expression would *not* evaluate to `true`[3]. This allows the optimiser to optimise the program based on the assumption that it always will.

— Simply ignoring the whole statement is a conforming implementation, i.e. the optimiser is not required in any way to make use of the assumption.

2.2 Proposed syntax

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.

We advise against a syntax involving a colon, such as `[[assume:` *`expression`*`]]` or other variations that deviate from existing C++ attribute grammar, because this would require otherwise unnecessary changes to the C++ grammar, and make it harder to add assumptions to existing code.

Making this an attribute also makes it clear to the user that, 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 also believe that this syntax has the least impact on the core language as opposed to the alternatives (see below).

[2] However, just like with `decltype`, the expression might cause a template instantiation if this is required to analyse the expression. Thanks to Nathan Sidwell for pointing this out.

[3] Note that 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 undefined behaviour 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.

3 Syntax alternatives

3.1 Keyword

An assumption can be characterised as an operator with an unevaluated operand, somewhat similar to `decltype(`*`expression`*`)`. We could consider proposing a new keyword for this new operator, such that `__builtin_assume(`*`expression`*`)` instead becomes:

```
assume(expression)
```

However, portable assumptions are a low-level expert feature, with the potential to inject undefined behaviour into an otherwise valid program. It should be used carefully and sparingly. We therefore advise against introducing a new keyword for this feature.

3.2 Macro

Instead of introducing a keyword, we could introduce an `assume` macro, analogous to how `assert` is already defined as a macro. 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 therefore advise against introducing a new macro for this feature.

3.3 "Magic" library function

Alternatively, we could introduce portable assumptions in the form of a "magic" library function, so `__builtin_assume(`*`expression`*`)` instead becomes:

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

However, such a spelling would introduce a weird novelty into the C++ language: something that is syntactically a function call, yet does not evaluate its operand. It would essentially be something like a "namespaced keyword", and very different in nature to all existing "magic" library functions. Significant core language changes would be needed to make it work. We also believe that such a construct would be surprising to C++ developers. We therefore advise against adding this feature as a library function.

A seeming advantage is the consistency with the closely related `std::assume_aligned` [P1007R3], which was adopted for C++20. However, they are different. For `std::assume_aligned`, unlike for an assumption, the operand may be evaluated, so the problem above does not arise for `std::assume_aligned` (or, in fact, any other current "magic" library function).

4 Wording

The formal wording for this proposal will be provided in a future revision.

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.

References

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

[P0542R5] Gabriel Dos Reis, Jose Daniel Garcia, John Lakosand Alisdair Meredith, Nathan Myers, and Bjarne Stroustrup. Support for contract based programming in C++. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0542r5.html, 2018-06-08.

[P1007R3] Timur Doumler. std::assume_aligned. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1007r3.pdf, 2018-11-07.

[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.

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