Assumptions

Document Number: **P2064 R0**
Date: 2020-01-13
Reply-to: Herb Sutter (hsutter@microsoft.com)
Audience: SG21, EWG

Abstract

This paper describes why assertions and assumptions are different but related, and how the answers can help to inform the design of how to expose assertions (contracts) and assumptions in the standard.

Contents

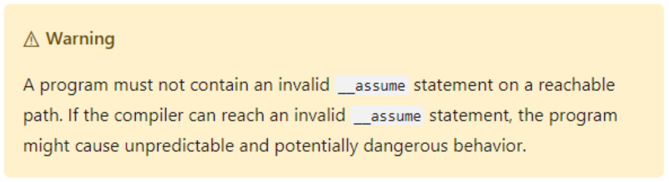
1 Assert ≠ Assume

1.1 Definitions

SG21 is working on a lexicon of preferred terms to use when talking about these and related concepts. Until that lexicon exists, this paper will use (and strongly recommends our lexicon adheres to) the terms **assertion** and **assumption** because they are the universal existing practice for >70 years and >20 years, respectively.

1.2 Summary

	Assert(*expr*)	**Assume(*expr*)**
Why used: Purpose	Program bug detection Document expected state to detect bugs when analyzing or executing this code	Optimization Inject an additional fact to compile this code better, that cannot be inferred (practically or at all) by the optimizer from the program source
What *expr* is	*expr* is a compilable expression that documents expected program state at this point *expr* is always evaluated if checked Assertions are “baby/one-line unit tests,” and are related to unit testing (unit tests should exercise at least all assertions in the code under test)	*expr* is a compilable expression that is a claimed truth injected into the optimizer *expr* is never evaluated The fact expressed is that the assumed expression is known a priori to be guaranteed to evaluate to `true` (If *expr* is unconditionally `false`, the fact expressed is that the code where the assumption appears is unreachable. This is functionally a separate feature; see §3.)
Where written	Inside a function body On a function declaration, for pre- and post-conditions At class scope, for invariants	Inside a function body (specifically, on a control flow edge)
Who uses	All programmers of all skill levels should use assertions liberally to document expected state and aid debugging	For careful use by experts only, on a case by case basis to enable a specific desired optimization in a specified source location (measuring before and after to validate, as with all optimizations)
When used	Pervasively, the more the better ~1000× more often than Assume If in doubt, assert; you cannot go wrong by writing an assertion, the worst that can happen is that it fires and you fix or remove an incorrect assertion	Rarely, case by case where the optimizer could/should be performing an optimization (e.g., loop unrolling, vectorization) but cannot without explicit guidance If in doubt, never assume; it is the strongest “trust me” with the most dangerous consequences if `false`
How implemented	In library code/macro, or infetrinsic Code is always generated when checked: Equivalent to an ordinary `if(!`*expr*`)` program branch	In optimizer Code is never generated when used, is not even a true intrinsic: The optimizer can make use of the expressed fact as true without proof or checking

Relationship to Assert checks	Can never remove another enabled assertion check[1]	Can remove an enabled assertion check (in both directions, including backwards 'time travel' where reachable)
Relationship to program correctness	Can be `false` If all assertions are true, then increases confidence that the program is correct	Must never be `false` Must always be consistent with the program (no inference either way)
Meaning if `false`	There is a bug in the program or in the Assert's *expr*	There is a bug in the Assume's *expr*
Consequences if `false`	A bug is diagnosed (if checking is on) Examples: program terminates unexpectedly with last-ditch terminate handlers called, typically at test time	UB + miscompilation The program is inconsistent: The optimizer has facts that cannot all be true, which can never happen with facts derived from the program source code Examples: wild writes (e.g., elided data bounds checks), arbitrary code execution (e.g., elided `switch` jump table bounds checks), nonsense code generation
Safety	Safe for widespread use by non-experts	Inherently unsafe, can inject UB more broadly than ordinary UB (by persisting into late optimization + time travel; see §3.2) + miscompilation
Existing practice, in general and in C or C++	1947: "assertion" [von Neumann 1947] (see §3) 1949: "assertion" [Turing 1949] (see §3) 1972: K&R C `assert(expr)`	1998: MSVC 6.0 `_assume(expr)`[2] 2011: GCC 4.5.3 and Clang 3.0.0 via `if(expr){}else{__builtin_unreachable();}` 2015: Clang 3.6 `__builtin_assume(expr)`
Existing practice documentation warnings	Only syntactic (mainly, `assert` is a macro and so commas in template argument lists have to be protected by parentheses)	MSVC `__assume` documentation has long carried the warning:  ⚠ Warning A program must not contain an invalid `__assume` statement on a reachable path. If the compiler can reach an invalid `__assume` statement, the program might cause unpredictable and potentially dangerous behavior.

[1] Even if it's a duplicate of another assertion, the optimizer can remove redundant evaluations via vanilla as-if CSE, but the check is still performed.

[2] Renamed from `_assume` to `__assume` in 2003 for standards conformance.

2 How Assert and Assume are related

2.1 Assert $\not\Rightarrow$ Assume

An Assert should never be implicitly Assumed. The two features are different in every row in §1, such as:

- Assert and Assume have unrelated purposes.
- Assert exists to be checked for `false`, whereas Assume must be guaranteed to never be `false`.
- Assert evaluates its expression, whereas Assume never evaluates it.
- Assert is a safe debugging aid that should be used pervasively by all programmers, whereas Assume is a dangerous power tool for experts only, and is in practice used ~1000× less frequently than Assert.
- Asserts cannot elide other checked Asserts, whereas Assumes can and do elide Asserts.
- … and the other differences mentioned in §1.

Consider that the only time it would be safe to assume an assertion (i.e., assume a test) is if the program is going to terminate anyway if the assertion is false, and will still terminate even if the assertion is assumed. However, guaranteeing that is rarely possible (recall that the assumption can elide a check), and even then rarely gives enough noticeable benefit to be desirable.

It is common for the same expression to be both Asserted and Assumed, but because it is the other direction: It is an Assume that is Asserted in debug mode (see §2.3). For it to be a debugging Assert that is Assumed in release mode is backwards, even though that was the direction promoted by the former draft C++20 contracts feature.

2.1.1 Field experience: MSVC C1xx (C++ front-end) and C2 (back-end)

The only large-scale case I know of that did assume assertions was in the Microsoft C++ compiler front-end and back-end. In summary: It arose unintentionally (the intent was the reverse, to assert assumptions), caused reliability problems, and has been removed (with minor performance gains).

In 1998 when the MSVC team added `__assume(expr)`, they also intentionally added a macro `DASSERT(expr)` that would become essentially `assert(expr)` in debug builds and `__assume(expr)` in release builds. The intent of the macro was to write assumptions that would be asserted at test time (the opposite direction to this section, covered in §2.3), so in hindsight the macro should have been named `DASSUME` and carried explicit warnings. But because the name of the macro used the word "assert," in practice this led to its use for assertions. Mark Hall reports how it led to unstable behavior and difficult to reproduce bugs:

> For quite some time we expanded `DASSERT(e)` to `__assume(e)` in the retail compiler (this was another design goal). However, years later we had to admit that `DASSERT(e)` was too often used as a defensive programming tool, rather than a way to express a known invariant. Too many times we found that `e` would prove false in the wild, leading to ICEs [the compiler crashing].
>
> This is even though the way we used it in the front end ended up ignoring the large majority of the generated `__assume` statements, focusing only on `__assume(0)` and `__assume(expr)` for simple expressions. I just did a count, and the fraction was about 10%. But even though only 10% of the `DASSERT`s were actually assumed, it still wreaked havoc. It was fortunate for us that the optimizer only took advantage of our simplest `__assume(e)` intrinsics, but even that low percentage exposed us for, as Mark Twain would say, a bunch of 'darned liars.'

Eric Brumer reports regarding the MSVC C1xx front-end and C2 back-end:

> Since the 1990s, the C2 back-end has many types of assertions, which were all some variant of an ASSERT macro. All would crash in debug builds, and would either `__assume`, crash, or do nothing in retail builds.
>
> In 16.5, we changed the `__assume` behavior to do diagnostic logging in retail builds [For the reasons Mark Hall gave].
>
> An example of confusion is when folks work with the macro that crashes in debug mode, but `__assume`d in retail builds, and used it for defensive programming because it was named "assert." This is code (paraphrased) that was in the front-end, and seems totally reasonable if doing defensive programming (but is totally unreasonable if writing an assumption):
>
> ```
> int DoSomething(int x) {
>     DASSERT(x != 0);
>     if (x == 0) {
>         return -1; // special return code
>     }
>     // do useful work
> }
> ```
>
> Here the FE team wanted crash information when `x == 0` in debug mode, but for the function to return a special return code in release mode. Things were fine [worked as expected] in debug mode, but in retail the `DASSERT` was expanding to an `__assume`, and our optimizer was turning it all into just:
>
> ```
> int DoSomething(int x) {
>     // do useful work
> }
> ```
>
> … which is not the desired effect and reduced the quality of our product, by increasing compiler crashes and making problems in optimized release builds much more difficult to debug because it is hard to reason about code reliably when we were compiling a significantly different program than we thought.

Notes: (1) Assuming assertions disables this class of "defense in depth" coding pattern: to Assert something in testing to minimize actual occurrences, but then in production still provide fallback handling for robustness. (2) This practical experience may support that Assume should not use a syntax that appears to be like an Assert (contract), as it was in the former draft C++20 contracts design, because in this case doing so has misled developers into misusing it.

Brumer reports the results of removing `__assume` from C1xx and C2, work championed and implemented by Natalia Glagoleva:

> When we removed uses of `__assume` from our compiler's own code base, we unmasked many errors of this kind that Assert in debug mode and do special handling in release mode, which now worked as intended.
>
> We also discovered many warnings about variables not being initialized on all paths that had been masked by `DASSERT`s, which we fixed by initializing the variables.

> Both changes were a solid improvement.
>
> Additionally, removing all usage of `__assume` in the C2 back-end resulted in C2 running 1% faster. This is an indication that `__assume` hasn't played nicely with our optimizer (we use ourselves to compile ourselves). However, it's possible that other optimizers fare better with `__assume` than we do.

The last paragraph is a reminder that Assume is not free; it is actively asking the optimizer to perform additional work and persist more internal data for longer times (see §3 for additional details).

Xiang Fan, the developer responsible for removing `__assume` specifically from the MSVC front-end, adds:

> Much to our surprise, the C1xx front end actually got a 1-2% faster with `__assume` statements removed.

Jonathan Caves added for MSVC:

> When we removed `__assume` from the front-end it fixed a lot of strange crashes in our compiler.
>
> For example, we had cases where a function with a `switch` statement used a `DASSERT(false)` for the default case for debugging, but later at the end of the function was a `return nullptr;` statement as a historical default that was exercised in release mode:
>
> ```
> int* sample() {
>
>     switch(...) {
>     /* all cases intended to be covered, and return */
>     default: DASSERT(0);
>     }
>
>     //...
>     return nullptr;
> }
> ```
>
> In retail mode, the `DASSERT` became an `__assume` which elided the `return nullptr;` as unreachable, and so the caller was getting back a pointer value that was whatever happened to be in the EAX register.
>
> Then the compiler crashed when the pointer was used, typically not at the immediate call site but several functions later, even though the later function was correctly guarded with a null check but where we blew past the null check because the pointer value was an arbitrary bit pattern, typically not `0`.

2.2 Assert $\not\perp$ Assume

Assert and Assume are not orthogonal. They cannot be independently combined in sensible ways.

2.2.1 Counterexample: “Assumed precondition”

I raised the following question in Cologne (2019) during the draft C++20 contracts discussions:

> “What does an *assumed precondition* mean? What does it express, and how should it be used?”

In the then-status quo, an “assumed precondition” would be spelled using a precondition with a level for which checking is not enabled:

```
[[pre /*unchecked level*/: expr]]  // then-status quo permitted assuming expr if unchecked
```

And in some contemporaneous variant proposals such as [P1607] it would be spelled more directly as:

```
[[pre assume: expr]]
```

This case also came up in various discussions at meetings and on committee email lists, for example this from a committee email list post in summer 2019 (emphasis added):

> … Consider: [[**pre assume**: …

It’s important to understand why the answer is “this is self-contradictory.” Consider what each term means:

- A *precondition* is an expression written by a function author that states expectations on values of the function’s arguments that every call site is expected to make true. So it is **a statement about code the precondition’s author does not control** (the call site is typically written by someone else who uses the precondition to unit-test the correctness of their own calling code, and most call sites often don’t even exist yet when the precondition is written) **and cannot guarantee to be true** (in fact, it could still do something sensible with violating values in release mode, such as in the `DoSomething` and `sample` examples in §2.1.1; also, preconditions are most valuable when found to be `false`). A `false` precondition, when checked, **injects a run-time diagnostic into the caller’s local call site location**.
- An *assumption* means to inject an unverifiable fact into the compiler that the compiler could not deduce from the source code. So it is **a statement about code the assumption’s author controls and can guarantee to always be true**. A `false` assumption **injects non-local hard language UB into the caller’s whole program** including via time travel before where a reachable assumption appears — in practice typically including, but not limited to, wild data writes and arbitrary code execution (e.g., from elided `switch` jump table bounds checks).

So writing an “assumed precondition” means writing an expression that we cannot guarantee to be true, and cannot bear to be false, which is a contradiction.

Since it has no sensible meaning, nobody should ever write such a thing, and a high-quality design for Assert (contracts) and Assume would make this combination difficult or impossible to spell. Fortunately, in existing practice, by design we cannot write MSVC `__assume()` or Clang `__builtin_assume()` on a function declaration as a precondition statement about someone else’s code we have never even seen and cannot validate, because they are as-if magic functions which cannot appear in that syntactic location today. Any standardized version of Assume should maintain this as a feature; it is not a bug. Assumptions are only ever used locally, and even then very tactically and sparingly when we know optimization is needed and that the assumption will enable an optimization, and very carefully because it had better never be `false`.

2.3 Assume ⇒ Assert

This brings us to the correct relationship between Assert and Assume:

Every Assume should be Asserted so that it can be checked at test time, because the consequences of violating an Assume are so dire. This informs why we should follow existing practice and provide it as-if a magic function.

Additionally, consider where Assumes are used:

2.3.1 Assume should Assert its parameter as a precondition

Recall from §1 that Assumes are used in function bodies only. This is a strong reason to lexically express Assume with the syntax of a function call expression (even though it is not really a function call, not even an intrinsic one), which follows all existing practice, including the previous proposal by Hal Finkel in [N4425] and as one of the options proposed by Timur Doumler in [P1774R1].

So if we standardize Assume, the ideal way to express it is *as-if an intrinsic function* (following existing practice and limiting natural uses to within function bodies) *that Asserts its parameter* (so that it is checked at test/debug time, ideally as a precondition but in the function body will do too). For example, using former draft C++20 syntax:

```
// IMO an ideal declaration of “Assume” that embodies their correct relationship
void /*std::*/unsafe_assume(bool b) [[pre: b]] ;    // (or “expects:”) draft C++20 syntax
```

Alternatively, in the absence of a general contracts feature, with an `assert` (or similar) in the body:

```
void /*std::*/unsafe_assume(bool b) { assert(b); }
```

or else *as-if declared as a function-like macro* (note this method is already popular in existing practice to Assert in debug mode and Assume in release mode):

```
#ifdef NDEBUG
    #define __unsafe_assume(b) __compiler_magic(b)
#else
    #define __unsafe_assume(b) assert(b)
#endif
```

The name should include the word **unsafe** because Assume is inherently unsafe. As detailed in §3, it enables a strict superset of all the things we currently call unsafe, including the set of all hard language undefined behaviors (including in more places) and the standard term “*vectorization-unsafe*,” in addition to allowing contradictions in the optimizer equivalent to miscompilation. A standardized Assume would be the most dangerous tool in our standard, so it deserves the word “unsafe” if anything does.[3]

[3] Eric Brumer expresses this opinion based on his experience supporting the MSVC `__assume` implementation: “I tell anyone who asks, ‘don’t use `__assume`, ever.’ It rarely gives you what you want, and simply opens you up to horrible-ness that could happen if you get it wrong.”

2.3.2 Assume should not be expressed as an attribute

I think that the suggestion to express Assume as an attribute, as directed by SG17 Belfast and reflected in [P1774R2], is a suboptimal choice for several reasons:

- Assume with attribute syntax would make Assumes awkward to write in the one place they should appear, which is as a statement (see §1).
- Assume with attribute syntax would allow Assumes to be written outside function bodies (e.g., on function declarations), where they are not meaningful and actively harmful (see §2.2.1).
- Assume with attribute syntax would make it harder to express that it Asserts its parameter as a precondition for test time diagnostics if contracts (Asserts) are eventually also added as attributes, because we can't write an attribute on an attribute. In contrast, `unsafe_assume(bool b) [[pre: b]]` is easy to write naturally and exactly documents their correct relationship.
- Assume with attribute syntax would be a novel invention not supported by any existing practice in the past >20 years in shipping commercial compilers.

Additional reasons why Assume should not be an attribute:

- Assume with attribute syntax would imply that if the program is correct in an implementation that uses the attribute, then ignoring the attribute does not affect program meaning. However, Assume has a stronger effect on program meaning than any currently standard feature (see also the causality violation examples in §3.4.2).
- Assume with attribute syntax would be inconsistent with EWG direction for C++20 `std::assume_aligned`. See [P1007R3], which includes EWG Jacksonville (2018) direction to not make it an attribute.

However, one salient difference is that Assume(*expr*) must not evaluate *expr*, which is unlike a normal function call. So if exposed as a magic function-like syntax, the function would be magic in this respect (not evaluating its argument) as well. Alternatively, Assume could be a hardwired language intrinsic, like `sizeof`, which also does not evaluate its argument. Either is better than an attribute.

There is one counterexample I know of that is a form of assumption but is an attribute: `[[noreturn]]` is more than a hint and, like `std::assume_aligned`, is a specific form of assumption.

However, note that hints like `[[likely]]`, `[[unlikely]]`, and `inline` are not assumptions; they are hints, they do not inject facts. For example, `[[likely]]` and `[[unlikely]]` hint at how often a branch is expected to be used so as to improve code generation, but they do not state that a branch is not reachable at all so as to actually change code generation such as by pruning the branch; this is why "likely" and "unreachable" intrinsics are separate in practice. Similarly, `inline` is a hint about the expected best way to treat it during compilation of call sites, but it cannot change whether a function is callable or not so as to remove it. Unlike these hints, Assume is not a hint; it does inject facts that do change program meaning.

3 As-if < UB < Assume(`false`) < Assume(expr) ≤ Miscompile

3.1 Definitions

We enable optimizations primarily via the as-if rule, which cannot change the observable behavior of a program:

- **As-if rule**, such as to allow common subexpression elimination (CSE) or loop inversion in the absence of observable side effects. This is basic permission for ordinary optimizations to happen.

The following three kinds of UB can change a program's behavior, and are related but not equivalent:

- **UB**, such as `*(volatile int*)0 = 0xDEAD`. This is "vanilla" or "garden-variety" hard language UB that grants permission to translate to an executable containing nasal demons, hard drive reformat, etc. This is useful for optimization, but primarily passively (to omit expensive checks), only secondarily actively (by inferring facts from UB).
- **Assume(`false`)**, such as MSVC `__assume(0)` and GCC/Clang `__builtin_unreachable()`, to inform the optimizer that a branch is unreachable.
- **Assume(*expr*)**, such as MSVC `__assume(expr)` and Clang `__builtin_assume(expr)`, to inject a compilable (but unevaluated) data relationship fact into the optimizer. (This paper does not discuss related narrower features, such as GCC/Clang `__builtin_assume_aligned` and C++17 `std::par_unseq`.)

Because all are UB, we could try to indirectly emulate the later ones in terms of preceding ones. For example:

```
// Assume(expr) and Assume(false) in terms of UB

#define __hand_rolled_assume(expr)   if(expr){}else{ *(volatile int*)0 = 0xDEAD; }
#define __hand_rolled_assume(expr)   if(expr){}else{ const int i = 0; (int&)i=0xDEAD; }
#define __hand_rolled_assume_false() (*((volatile int*)0)=0xDEAD)

// Assume(expr) in terms of __builtin_unreachable

#define __hand_rolled_assume(expr)   if(expr){}else{ __builtin_unreachable(); }
```

But these emulations are not equivalent. They can be practically equivalent only by teaching the optimizer to recognize, and teaching the programmer to use, a specific pattern to infer that the programmer intended to express the higher-level Assume(`false`) or Assume(*expr*). This has three problems: it supports the feature via an ornate indirect spelling instead of a direct spelling; it opens the door to mistakenly recognizing it in cases the programmer did not intend; and it incurs compile-time overhead by requiring the optimizer to retain more information for a longer time.[4]

Every major C++ implementation has added Assume(`false`), and except for GCC also Assume(*expr*), with a direct spelling, even though the previous one(s) in the list were already available in the same compiler and they could have relied on canonizing a known indirect spelling.[5]

[4] Similarly, some can also be indirectly emulated using other language features. For example, a `__builtin_unreachable()` magic intrinsic can be emulated as an ordinary function `[[noreturn]] __builtin_unreachable();`, but again indirectly and requiring the compiler to recognize and use a pattern.

[5] Although in MSVC both are spelled with the word `__assume` and documented on the same page, the documentation calls out `__assume(0)` as a separate feature from `__assume(expr)` that has different effects and usage guidance.

3.2 Summary

This section summarizes the differences. The next sections adds implementer discussion and a compiler survey.

	As-if rule	**UB**	**Assume(`false`)**	**Assume(*expr*)**
Why used: Purpose	Allow ordinary same-thread optimization, preserving observable behavior	Avoid requiring the compiler to perform/emit potentially-expensive checks Enable diagnostic tools for some classes of errors (e.g., sanitizers)	Directly inform the front-end or optimizer that a specific branch is not reachable (dead in the AST or CFG)	Directly inform the optimizer of a data relationship
What it enables	"Vanilla / garden-variety" same-thread optimization transformations Examples: CSE, memory access reordering, loop inversion	Not diagnosing problems Examples: Omit an integer overflow check (to save execution time), omit diagnosing a construct (to save compilation time)	Actively pruning an individual branch (edge/path in the AST or CFG), inferring simple facts by proving reachability Example: Suppress variable uninit warning along some path, eliminate `switch` jump table bounds check	Actively generating different code for other statements using the related data Example: elide other branches (e.g., Asserts), enable vectorization
Allows adding facts?	No	Inferring simple facts by proving reachability	Inferring simple facts by proving reachability	Directly adding arbitrary facts without proof
Where written	n/a, implicit	Implicitly (whitespace) in program source code	Explicitly within a specific branch, with effects aimed at enabling pruning that branch only	Explicitly anywhere in a function body, with general effects on uses of the variables involved
Who uses	n/a, implicit	No one,[6] typically used only accidentally	For careful use by expert programmers	For careful use by expert programmers
When used	n/a, implicit	Ideally never, programs should never rely on UB	To prune a specific local branch	To get different code generation
How implemented	In front-end, optimizer, and code generator, to respect/enforce language rules	In front-end and optimizer, which recognize specific permission to assume certain program constructs have certain characteristics	In front-end and optimizer, to prune an AST or CFG path or infer a simple fact from proving reachability	In optimizer and code generator, to record a potentially complex data relationship fact
Represented in IL?	Yes, deeply for inter-thread constraints that	Typically no, rather by the absence of a constraint	Yes, pruned early Clang: Pruned aggressively	Yes, long-lived usually to the end of optimization,

[6] Of course, tool vendors can exploit it. This row is about how a programmer writing source code intentionally uses the feature.

	affect generated object code		MSVC: Pruned early, used only to suppress warnings and for reachability optimization	including to persist otherwise-empty functions (which widens the effect of the Assume)
Represented in generated object code?	Yes for inter-thread constraints (e.g., atomics, memory fences)	No	No (only changes generated code to remove unused code)	Yes, can change generated code (e.g., use vector instructions)
Consequences on program meaning if present (and, for Assert, not true)	None	Wild writes and arbitrary code execution, with global effects (grants permission for the whole program to have arbitrary meaning), compilers can indirectly infer simple facts when they can prove reachability Indirectly allows pruning a branch or inferring a contradictory fact Time travel where reachable	Same cases as UB + strictly more, because all major compilers indirectly infer simple facts more often from Assert(`false`) than from most ordinary UB in the same position Directly allows pruning a branch (CFG edge/path) Time travel where reachable	Same cases as Assert(`false`) + strictly more, because it allows stating arbitrarily complex facts, and those facts are persisted longer and used throughout optimization phases Directly allows injecting contradictory facts into the optimizer,[7] allowing results that are equivalent to miscompilation (injecting a bug into the compiler) Time travel where reachable
Inferable from columns to the left?	n/a	No, unrelated to as-if	Sometimes, but requires separate Assume(`false`) feature to reliably express branch pruning intent	Rarely – Yes by recognized coding pattern, but that is an ornate spelling for Assume(*expr*)

[7] UB and Assume(`false`) only inject contradictory facts when the compiler aggressively infers facts from UB or unreachability, which is much more difficult in principle and much less common in practice. Even aggressive optimizers infer mainly simple facts such as pointer non-nullness and then on code paths they can prove, not arbitrary data relationship expressions to be believed without verification.

Other "trust me" statements also allow injecting undefined behavior into a program if the "trust me" is wrong, but the consequences are strictly weaker. For example, calling `for_each(par, first, last, []{/*body*/});` implicitly says "trust me, run copies of `body` in parallel; I take responsibility that `body` does not use unsynchronized shared data or perform any other `par`-unsafe operations," but the consequences of that being untrue are "only" a data race causing undefined run-time behavior (e.g., torn/partially-constructed objects, wild writes, random code execution, causality violation), and cannot in general inject contradictory facts into the optimizer (e.g., miscompile code, equivalent to injecting a compiler bug).

3.3 Discussion

I consulted developers familiar with these features (where possible, the original authors) to ask why they did not implement it in terms of the previous one(s) in the list. Here are their responses and additional related discussion.

3.3.1 Why not implement Assume(*expr*) in terms of Assume(`false`)?

Hal Finkel for Clang:

> There is no fundamental semantic difference between the two, `__builtin_assume(false)` is essentially `__builtin_unreachable()`, but there were a couple of issues with generalizing that to `__builtin_assume(expr)`...
>
> Going back to the time when the feature was designed, while Clang would certainly parse `{ if(!expr){ __builtin_unreachable(); } }`, it didn't have the same effect as in GCC. Specifically, in LLVM, the representation of `{ if(!expr){ __builtin_unreachable(); } }` is aggressively pruned, and so the optimizer cannot later use the information about *expr* – and if *expr* has no other users, it too will be removed. The tradeoffs around changing this were not favorable:
>
> 1. Not only do existing users add `__builtin_unreachable` in order to help the optimizer reduce code size, as do other LLVM frontends, but the internal representation for unreachable is generated by some transformations knowing it will trigger applicable DCE / CFG simplification later in the pipeline.
> 2. Keeping dead code in the form of `{ if(!expr){ __builtin_unreachable(); } }` is relatively expensive (at least in LLVM's representation) because it has multiple basic blocks, and breaks up otherwise-straightline code making it more difficult to analyze, and adds additional uses of values which restricts the optimizer's ability to perform transformations. Thus, even if it all goes away in the end, keeping multiple basic blocks at intermediate points in the pipeline can still interfere with code quality.
> 3. `__builtin_assume`, and friends, were viewed as relatively rare compared to other places where the optimizer deduces dead code, and *even rare compared to cases where a user inserts `__builtin_unreachable`* [emphasis added] to help the optimizer reduce code size. Use of `__builtin_assume` is more common now than it once was, but I believe this is still true. As a result, we don't want to keep around all representations of unreachable code just in case it might help the optimizer later (because the likelihood of negatively impacting code quality is significant), and this is still true even considering that the programmer might have put in the `__builtin_unreachable` directly (because, chances are, they just wanted to reduce code size).
>
> Thus, in terms of an unreachable representation, we would end up wanting two things: a `__builtin_unreachable` representation that we pruned aggressively (which we already had) and a `__builtin_unreachable_but_keep_me` that would be used to represent assumptions (which, presumably considering that the user felt it was important enough to add explicitly, we should not prune aggressively). However, the only use case we had for `__builtin_unreachable_but_keep_me`, either internally or at the source level, was to represent assumptions. Internally, it's better to have an explicit `assume` representation (so we don't have more basic blocks than necessary), and from a language-design standpoint, `__builtin_assume` seemed better than `__builtin_unreachable_but_keep_me` (regardless of bikeshedding).

Thus, to be explicit, we keep around the internal representation of `__builtin_assume` until the very end of the pipeline. We even keep it in functions that are otherwise empty (in case the assumption might be later useful if the function is inlined). In general, we take no such care to keep around unreachable code. However...

LLVM does now, although this is a relatively-recent change (May of 2019), internally transform the canonical `{ if(!expr){ __builtin_unreachable(); } }` pattern directly into an assumption. This was done as a compromise for GCC compatibility, and because it seems like there's not a huge overlap between that specific pattern and other cases where unreachable is generated by the optimizer or other frontends for which you want the aggressive pruning behavior. Some of the discussion on this is on the associated review thread (https://reviews.llvm.org/D61409). I think that this will probably stick, but if we end up finding cases where we need to disable the transformation for performance reasons, I will not be surprised.

Eric Brumer for MSVC:

`__assume(expr)` is represented by a bunch of IR in our back-end.

Pretty early in our back-end, we find `__assume(0)` and mark all those blocks in the flow graph as unreachable (this is akin to LLVM's pruning).

We leave the remaining `__assume(expr)`'s around and run a large chunk of our optimizer. Some of these optimizations understand `__assume(expr)` and take advantage of it, and some optimizations get hampered by it.

We remove the remainder of all `__assume(expr)`'s near the end of the optimizer.

Brumer adds:

`__assume(0)` is treated differently than other `__assume(expr)` statements by the MSVC optimizer: `__assume(0)` is hooked to only warning-emission code and reachability optimizations. For instance, the back-end emits a warning here:

```
int test(bool cnd1, bool cnd2) {
    int x;

    if (cnd1)      x = 5;
    else if (cnd2) x = 6;

    return x;    // warning C4701: ‘x’ potentially uninitialized
}
```

But adding an `else __assume(0)` silences the warning:

```
int test(bool cnd1, bool cnd2) {
    int x;

    if (cnd1)      x = 5;
    else if (cnd2) x = 6;
    else __assume(0);

    return x;    // ok, no warning
}
```

Similar, but different, is this:

```
int test(int x) {
    switch (x) {
        case 1: return 5;
        case 2: return 7;
        case 3: return 2;
        case 4: return 1;
        case 5: return 2;
        case 6: return 4;
        case 7: return 0;
    }
    return 3;    // never executed, just avoids a warning
}
```

Compiled as-is, we emit a jump table that's guarded by an `if(x>7)` check. Adding `default: __assume(0);` causes the optimizer to elide the `if(x>7)` check, which can result in arbitrary code execution.

```
int test(int x) {
    switch (x) {
        case 1: return 5;
        case 2: return 7;
        case 3: return 2;
        case 4: return 1;
        case 5: return 2;
        case 6: return 4;
        case 7: return 0;
        default: __assume(0); // no if(x>7) check, arbitrary code execution
    }
    return 3;                 // never executed, just avoids a warning
}
```

Louis Lafrenière for MSVC:

In practice, supporting `__assume` didn't fit the back-end's global optimizer design well, and it is also quite risky to use.

The main practical usage seems to be the `__assume(0)` in the default of a `switch` known to cover all possible cases. We should have probably just implemented something like `__builtin_unreachable()`.

3.3.2 Why not implement either Assume in terms of UB?

Mark Hall for MSVC:

> Assume(`false`) or Assume(*expr*) are difficult to express using arbitrary undefined behavior, where it's hard to know whether the programmer intended the UB to imply unreachability or fact injection. If there was a well-known canonical form or intrinsic such as `__builtin_undefined_behavior`, so that the programmer could write `if (e) __builtin_undefined_behavior` to explicitly say that `e`'s value deliberately should be considered undefined behavior, that style could potentially be used to the same effect. But the compiler would need to provide that feature, recognize that pattern, and we would have to teach programmers that it has special meaning and how to use it. `__assume` is a more convenient notation and directly expresses the intent.
>
> `__assume` was created to give the optimizer additional information it could not obtain in a single translation unit (or any state of the art) analysis. In theory the expression could be treated as a symbolic predicate which could be taken as an invariant.
>
> The `__builtin_unreachable` style predicate is expressed in our compiler by the degenerate case `__assume(0)`. But `__assume(e)` was meant to be fully general. For example, in the presence of `__assume(k>=0)`, we intend the optimizer to eliminate half of the range check for `k` in:
>
> ```
> switch (k)
> {
> case 0: ...
> case 1: ...
> default: ...
> }
> ```
>
> Other possible optimizations include calling a special form of `delete` or `free` that wouldn't have to check for null when `delete p` or `free p` was written with an `__assume(p!=nullptr)`.

Jason Merrill for GCC:

> GCC tends to replace code that is always UB with a trap instruction rather than `__builtin_unreachable()` [and so infers fewer facts from UB].

Eric Brumer notes:

> The computed-`goto` optimization requires an explicit instruction to `default: __assume(0);` in order to kick in. This can be an extremely powerful optimization to make very fast parsers/pattern matchers, and it seems to only fire when the optimizer directly knows that the default case is never executed. This appears to be the case in LLVM and MSVC. It involves some key branch duplication that is only beneficial for cheaper branches. Right now the only way to get this powerful optimization is with `__assume(0)`. I'd like there to be a better way.

3.4 Surveying real-world compilers: Cases and insights

3.4.1 Sample survey: Actual branch elision on major compilers and -O levels

I ran a set of tests each containing two branches (doing non-local or local work) separated by one of a few kinds of UB, `abort`, `[[noreturn]]`, Assume(`false`), and Assume(*expr*), to see which compilers at which optimization levels will elide surrounding branches in which direction. Godbolt: GCC + Clang, and MSVC + ICC.

This shows that compilers exploit fewer facts inferred from UB than from Assert(`false`) than from Assert(*expr*).

Below, "func" and "local" mean that the surrounding branches that are candidates to be elided contain function calls vs. only local variable manipulation, respectively, which is controlled by a macro in the above sample test code. An * asterisk means that the elided branch is additionally replaced with an abort instruction.

Case	**Clang 9.0.0**		**GCC 9.2**		**MSVC 19.22**		**ICC 19.0.1**	
	func	**local**	**func**	**local**	**func**	**local**	**func**	**local**
UB1: `*(volatile int*)0 = 0xDEAD;`								
removes pre branch at…	—	—	—	—	—	—	—	—
removes post branch at…	—	—	O2*	O2*	—	—	O2*	O2*
UB1-nonvolatile: `*(int*)0 = 0xDEAD;`								
removes pre branch at…	—	O1*	—	O2*	—	—	—	—
removes post branch at…	O1*	O1*	O2	O2*	—	O1	O2*	O2*
UB2: `const int i = 0; (int&)i=0xDEAD;`								
removes pre branch at…	—	—	—	—	—	—	—	—
removes post branch at…	—	—	—	—	—	—	—	—
UB3: `*(char*)"xyzzy"='Z';`								
removes pre branch at…	—	—	—	—	—	—	—	—
removes post branch at…	—	—	—	—	—	—	—	—
UB4: `auto x = 0xDEAD/0;`								
removes pre branch at…	—	—	—	—	(n/a, program rejected due to UB)		—	—
removes post branch at…	—	—	—	—			—	—
Noreturn: Call `[[noreturn]]` function								
removes pre branch at…	—	O1	—	O1	—	—	—	O1
removes post branch at…	O1	O1	O1	O1	O1	O1	O1	O1
Unreachable1: Assume(`false`)								
removes pre branch at…	—	O1	—	O1	—	—	—	—
removes post branch at…	O1	O1	O1	O1	—	—	O1	O1 [8]
Assume1: Assume(`i!=0`)								
removes pre branch at…	—	O1	—	O1	—	O1	—	—
removes post branch at…	O1	O1	O1	O1	O1	O1	O1	O1

[8] The generated code has an unusual empty branch left over, similar to source code of `if(i!=0){;}`. This seems to confirm that ICC applies Assume(`false`) very late in compilation, after running flowgraph-cleaner optimizations to remove empty branches: During cleanup the branch is still nonempty, then the Assume(`false`) processing removes its contents.

3.4.2 Existing products' usability limitations on using facts via time travel: Violations of sequential consistency and causality in current practice

While discussing this material with compiler implementers and doing the experiments in §3.4.1, it became clear that current optimizers deliberately restrain themselves from applying (inferred or stated) facts via time travel optimizations even for optimizations that are currently legal, because the results surprise users. In at least some cases documented in this section, it was because of direct resistance from users.

In particular, C++ implementations deliberately avoid applying a fact via time travel optimization if it would elide a non-local effect. For example, they avoid eliding branches that lexically precede the inferred/assumed fact if the branch contains more than accesses of local variables.

I would characterize this as an aspect of *sequential consistency* (SC), the memory model we already adopted for C++ concurrency.[9] SC is important because without SC the programmer cannot reason reliably about their code, and with SC they can reason about their code in the order it is written in the source (which is the only order the programmer can see), including that concurrent threads behaves as-if executed as some interleaving of each thread's code in its source order.

Importantly, SC includes being able to reason about our code with *causality*: Even in the presence of arbitrarily aggressive same-thread "as-if" optimizations, the happens-before relation ensures that another thread can never observe the effects of those optimizations as long as the program is correctly synchronized. If this were not guaranteed, then threads could observe (among many other things) causality violations, such as being able to see an effect before seeing its cause, which would be a paradox and mean that our program is literally unreasonable (cannot be reasoned about sensibly).

Similarly, regardless of what the standard allows, implementations already pragmatically limit applying optimizations before the point where they occur in a sequentially consistent reading of the program, also known as "time travel" optimizations — there's a reason we call them by that term which embodies the surprise involved. "Time travel" optimizations involve the same kind of causality violation that SC avoids (and why we require SC in the absence of data races in concurrency contexts): Users appear to expect that undefined behavior does not "bite" until the point it is *actually exercised*, not just when it is *reachable*. Informally, they do not expect undefined behavior to be treated as thoughtcrime (in the Orwellian and *Minority Report* sense), punishable before the crime actually occurs.

Jason Merrill reports for GCC:

> Here is an example of a time travel optimization:

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

[9] SC-DRF, or DRF-0, meaning sequential consistency if the programmer did not write a data race.

> In GCC, this reduces to `return 42;` at -O1.
>
> But if the effects of the first if-statement are visible outside the function, for example if `i` is a global variable, we don't eliminate them.

In the cases tested in §3.4.1, Clang and MSVC also follow much the same distinction, as shown by the results of the "func" vs. "local" variations of the Assume(*expr*) test case in §3.4.1.

Based on this, I offered this a combined example that illustrates how this can be a user surprise:

```
// My followup example: https://godbolt.org/z/US25Gd

auto test(int x) {
    int local = 0;
    local += x;

    f(local);                  // f's argument is 'local'
    int local2 = local;        // return value is 'local'

    ASSUME(x==0);
    return local2;
}
```

The same local variable has its value read in the two adjacent lines, but observes different simultaneous values in GCC and Clang under -O1, where the result is as if the function were written as just:

```
// test() is transformed to this:

auto test(int x) {             // 'local' is observed to have two values simultaneously:
    f(x);                      //   f's argument is x: 'local' is observed to be x
    return 0;                  //   return value is 0: 'local' is observed to be 0
}
```

The value of `local` was used in adjacent lines near (in this case *before*) the Assume operation: as the argument to `f()` where GCC does not apply the Assume, and to set the return value where GCC does apply the Assume. To the programmer trying to reason about an SC execution of their code, however, it appears that `local` holds two different values simultaneously, which is an impossibility.[10]

Eric Brumer agrees that MSVC follows a similar distinction, citing an example of active user resistance:

> We actually do the same as GCC for *implied* assumptions in our new optimizer. If you consider this code:
>
> ```
> void unreachable2(int *p, int *s1, int *s2) {
>     if (p == 0) { *s1 = 1; }
>     *p = 5;
>     if (p == 0) { *s2 = 2; }
> }
> ```

[10] In this toy example, the assumption was written directly and so maybe we could take the view that the code deserves to be broken because the programmer did a stupid thing. However, this shows how dangerous assumptions can be if they can ever be `false`; furthermore, in variations of this example the assumption could have as easily come from an inlined library function or other source, as was commonly proposed for assumed preconditions in draft C++20 contracts which would have been able to inject such arbitrary facts into call sites that would turn otherwise-benign bugs into language UB.

> Here the unconditional `*p` implies the fact `p != 0` for the entire scope, as per the standard. We originally had our new optimizer turn the function into:
>
> ```
> void unreachable2(int *p, int *s1, int *s2) {
>     *p = 5;
> }
> ```
>
> But when we spoke to kernel folks in Windows they essentially told us "absolutely not," so we do the same thing as GCC, and we emit this:
>
> ```
> void unreachable2(int *p, int *s1, int *s2) {
>     if (p == 0) { *s1 = 1; }
>     *p = 5;
> }
> ```
>
> My thoughts are: If developers need our optimizer to stop taking advantage of what we're already allowed to assume, then handwritten `__assume` statements make that split even more complex.

Brumer adds:

> When customers speak to me about `__assume`, it's usually in the context of "why the heck is the compiler doing this?" or "why the heck isn't the compiler doing this?" Our experience is that customers find `__assume` fairly confusing, and I believe there's a disconnect in how `__assume` is implemented in compilers from how developers think about their code.
>
> Consider this simple example:
>
> ```
> void test(int x) {
>     func1(x);
>     __assume(x == 0);
>     func2(x);
> }
> ```
>
> `x` is never modified in the body of `test()`. The user probably did not desire or intend that it allows the compiler to apply the `__assume` statement to instances of `x` *before* that `__assume`.
>
> In normal code like the following where we infer the value of `x` without `__assume`, we do that only along CFG *forward* edges:
>
> ```
> int test(int x) {
>     if (x == 1)
>         return x+2;
>     return func(x);
> }
> ```
>
> Here, we replace the `return x+2;` with just `return 3;`. Unlike `__assume`, all predicate information is transferred along forward CFG edges. Edges are easy to spot: They show up as lexical scope, and are generally always scoped intentionally by the user. However, `__assume` transfers such information into the whole visible scope, including backwards along CFG edges.

Note that CFG edges are directed. Having an effect travel backwards along a CFG edge is a source of surprise for the same reason that having a car travel the wrong way on a one-way street is a source of surprise.

Brumer continues:

> My theory about why users regularly report problems with using `__assume` is that predicate information is consumed by the compiler along forward edges, and the user understands forward control flow. If the user writes `__assume` in an existing scope, then according to any sane optimizer that assumption applies to that entire scope including before the `__assume` (and possibly even enclosing scopes), and that scope increase is probably not what the user intended.
>
> [Even with only forward fact propagation,] I personally find `__assume(expr)` too difficult for users to use, in just about every case. It creates the potential for unsafe code and unintended code, and the benefit is not worth it (aside from the computed-`goto` case [§3.3.2]). In [various examples], if the input conditions don't satisfy the `__assume`'d expression, then you have uninitialized variables, wild writes, or an unguarded jumptable which could be a random code execution security vulnerability.
>
> There are too many ways that `__assume` can yield different results than what the user intended. In a vacuum it's easy, but in 'real code' things get wonky where the optimizer subverts the user's expectations.

4 Acknowledgments

Thank you to Eric Brumer, Jonathan Caves, Gabriel Dos Reis, Timur Doumler, Xiang Fan, Hal Finkel, Mark Hall, Louis Lafrenière, and Jason Merrill for their contributions and for reviewing drafts of this paper.

5 Bibliography

In publication order.

[von Neumann 1947] H. H. Goldstine and J. von Neumann. “Planning and Coding of Problems for an Electronic Computing Instrument.” (Part II, Volume I, 1947-04-01). Page 12, emphasis original:

> “It may be true, that whenever C actually reaches a certain point in the flow diagram, one or more bound variables will necessarily possess certain specified values, or possess certain properties, or satisfy certain properties with each other. Furthermore, we may, at such a point, indicate the validity of these limitations. For this reason we will denote each area in which the validity of such limitations is being asserted, by a special box, which we call an *assertion* box.”

Note this paper was just about expressing facts as all, before compilers came along that could mechanically transform code including to check assertions (1950s) or optimizing compilers that could assume facts (1960s) with the modern distinction. The assertions were written by and for the programmer who would manually check the assertions during testing (by inspecting the proofs written in the assertion boxes; pages 17-18), and also assume (rely on) them as they wrote the next program steps (the programmer was the hand-optimizer). This early use was to write down checkpoints to divide a program into segments that could be reasoned about in isolation.

[Turing 1949] A. Turing. “Checking a large routine” (Friday, 24th June notes). Opening statements:

> “How can one check a routine in the sense of making sure that it is right? … the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole program easily follows.”

[Regehr 2014] J. Regehr. “Assertions are pessimistic, assumptions are optimistic” (Blog post, 2014-02-05).

[N4425] H. Finkel. “Generalized Dynamic Assumptions” (WG21 paper, 2015-04-07).

[P1007R3] T. Doumler and C. Carruth. “`std::assume_aligned`” (WG21 paper, 2018-11-07).

[P1773R0] T. Doumler. “Contracts have failed to provide a portable ‘assume’” (WG21 paper, 2019-06-17).

[P1607R1] J. Berne, J. Snyder, and R. McDougall. “Minimizing Contracts” (WG21 paper, 2019-07-23).

[P1774R1] T. Doumler. “Portable optimization hints” (WG21 paper, 2019-10-06). Proposed standardizing existing practice, using function-style syntax. (See §2.1.)

[P1774R2] T. Doumler. “Portable assumptions” (WG21 paper, 2019-11-25). Followed SG17 Belfast feedback to pursue using attribute syntax.