Function contracts (N3825)

Declaration-level static assertions v2

Céleste Ornato <celeste@ornato.com>

February 8, 2026

- **Proposal category**: Feature request
- **Target audience**: Developers writing libraries
- **Previous version**: N3641

This paper replaces the earlier N3641. Whilst the proposed feature is the same, the rationale changed to be more realistic. In short, the previous paper made the assumption that static assertions would get stronger as time goes on, following the trend of C23's `contexpr` and C2y's N3600 both making it easier to have compile-time guarantees. In this version, we put a stronger emphasis on the feature being used as a standardised and — most of all — **convenient** attribute, that can document code and be picked up by language servers, compilers, and sanitisers.

The latest version of this paper can be found on codeberg in the `pdf`, `tex`, and `org` formats.

N.B: "Secure" in this paper refers to source code that is semanticaly correct, with no undefined or loosely-defined behaviour, and that is easy to debug both at compilation and execution.

1 Abstract

This proposal aims to add the `[[assume(expr)]]` and `[[assume(expr, msg)]]` attributes to function declarations, allowing the designer of a library to specify constraints in function parameters.

This feature allows for added safety on the caller side, by making debugging easier through both static and execution-time analysis tools, and for further optimisation at compilation.

2 Introduction

Inattention and forgetfulness cause developers to write insecure code.

If one wishes to convey to the user of a library that certain function parameters can cause undefined or unwanted behaviour, they can only do so in written documentation. This leaves the developer in a precarious position: they can either add checks at execution-time, which come with an unwanted performance overhead and having to set up and document an error code system, or they can assume that everyone will read the documentation, which may create vulnerabilities.

`[[assume]]` seeks to be a standardised manner of conveying preconditions, with the added possibility of stopping compilation if a precondition is verifiably broken.

This feature can be seen as an extension to the C99 "static array index in function parameters" feature in that it both allows for compiler optimisations, and for analysis tools (linters, clang's `UBSan`) to detect condition breaches.

3 Proposal

3.1 Technical Description

When the attribute `[[assume(expr)]]` is associated with the declaration of a function `foo`, any call to `foo` will require the compiler to check that `expr` is **not provably false** (see 3.4) given the parameters.

It can then be assumed that `expr` is `true`, meaning the opposite case is undefined, both for the caller (upon calling the function) and for the callee (upon entering the function).

`expr` may address the function parameters by their name, or by calling them `$n`, where `n` is the 0-indexed number of the parameter from left to right.

If a referenced variable is an array, assertions will treat it as a pointer. Whilst keeping arrays as-is could have allowed for more features, this would have brought bug-prone semantics depending on whether the author of `expr` expected a pointer or an array. In any case, most problems related to the size of arrays are already solved by static indices in function signatures.

`[[assume(expr, msg)]]`, with `msg` being a string constant expression, is there to attach a message to the assumption as part of the source code, which compilers and debuggers may choose to use if they detect a contract breach at compilation-time.

3.2 Rationale

Introducing new undefined behaviour in C2y may be controversial; it would certainly not be seen as "Enabling secure programming" at a first glance.

This feature is meant for cases where the developer already considers certain parameters to be "Undefined behaviour". At this point, it does not matter whether the standard considers the code to be defined, because the results would still be unexpected and prone to breakage.

Allowing for further optimisation is only a welcome consequence of one being able to specify their own undefined behaviour. In reality, the wanted feature is to ease debugging, by compulsively making static assertions when possible and giving the opportunity for the checks to be done by debugging tools otherwise.

Function signatures using this attribute also self-document, making it easier to understand the assumptions made by the developer when writing the function. As it is code, language servers and linters may be able to directly mention those preconditions, unlike comments.

3.3 Example

```
[[assume($1 != 0)]]
int division(int, int);

[[assume(a >= b, "Substraction requires a >= b")]]
unsigned int subtract(unsigned int a, unsigned int b);

int main(void) {
    // These compile, with no execution-time overhead.
    int result1 = division(9, 3);
    unsigned int result2 = subtract(200, 60);

    // Error: "Assumption '$1 != 0' is false."
    int result3 = division(2, 0);
    // Error: "Substraction requires a >= b."
    unsigned int result4 = subtract(2, 9);
    // Compiles, debugging tools may pick up on the false assumption
    unsigned int a = 6;
    unsigned int b = 7;
    unsigned int result5 = subtract(a, b);
}
```

3.4 Quirks

An assumption whose validity cannot be proven will be treated as always valid. This should not be a problem, as this would just mean coming back to the status quo of having to be careful as a user. This fully aligns with how static array indices work in function signatures, including the fact that it would in all likeliness be picked up by Undefined Behaviour sanitisers.

Provably true expressions would for now only concern those only containing constant expressions and literal values, though that may change if `constexpr` functions get added to the standard. If postconditions get added later on, they may also be concerned.

4 Prior art

There are three popular systems that come close to our idea:

- C++26 Function Contract Specifiers,
- Clang's `__builtin_assume`; and
- MSVC's `__assume` keyword (which is functionally equivalent to `__builtin_assume`.)

As such, it is interesting to compare those ideas to ours.

4.1 In contrast to Contract Specifiers

```
unsigned divide(unsigned a, unsigned b)
    pre(b != 0)
    post(r: r <= a) {
    contract_assert(b != 0);
    return a/b;
}

unsigned main(void) {
    unsigned a = 7;
    unsigned b = 6;
    unsigned result = divide(a, b);
}
```

Figure 1: Example of C++26 Function Contract Specifiers

It's important to note that the C++ feature has its place; it is a more complex language and thus can and should add new syntax and semantics if one judges it beneficial to the language.

However, wanting to keep the C language simple (both to write and to implement), we shall see how this implementation may not be ideal.

Whilst similar in the abstract, there are several meaningful differences between Contract Specifiers and our version of the feature:

4.1.1 Portability

Whereas any compiler capable of recognising attributes (even if it's to ignore them) could compile code made using `[[assume]]` without impacting semantics, the specific `pre(expr) post(expr)` syntax would have to be known by every compiler; making it less likely for the feature to be used in code that wants itself compiler-portable.

4.1.2 Complexity

The C++ version entails more features and caveats than ours. Obviously it also includes a return specifier, but the main issue is `contract_assert(expr)`.

`contract_assert(expr)` is fine, but semantically heavier than it could be: expressions must be checked in the correct order in the case of multiple assertions, and the effects of a failed assertion are implementation-defined, ranging from nothing to an exception. Not only that, but the fact that it has to be done on the callee side means that we may still have checks done at execution-time in non-debug code.

Security-critical programs may not always want to rely on errors only occurring *sometimes*, and speed-critical programs probably do not want extraneous checks that could realistically only be done in debug builds. As such, this implementation is not ideal for C.

4.2 In contrast to `__assume`

```
unsigned int divide(unsigned int a, unsigned int b) {
    __assume(b != 0);
    return a/b;
}

int main(void) {
    unsigned int a = 6;
    unsigned int b = 9;
    __assume(b != 0);
    unsigned int result = divide(a, b);
    __assume(result <= a);
}
```

Figure 2: Example of `__assume`, functionally equivalent to the previous example

The `__assume(expr)~/~__builtin_assume(expr)` statement is simple, it just tells the compiler that an expression is *assumed* to evaluate to `true`.

There is little to be said about these. The main issue compared to `[[assume]]` would be that the assumptions have to be written on both the caller and callee side, that they are not compulsory, and that one cannot quickly see the assumptions from just the prototype.

4.3 With `[[assume]]`

```
[[assume(b != 0)]]
unsigned int divide(unsigned int a, unsigned int b)
{
    /* Assumption that b != 0 is made here, though only if the attribute was
     * visible when the function was compiled.  The visible effects of the
     * function do not change in any case*/
    return a/b;
}

int main(void) {
    unsigned int a = 0;
    unsigned int b = 6;
    /* Assumption that b != 0 is also made here */
    unsigned int result = divide(a, b);
}
```

The above snippet is **nearly** equivalent to the other two, as we do not conserve the assumption `result <= a`. Because they would make the feature more complex and add syntax, postconditions are not discussed in this paper; their addition would however be welcome.

Several things can be noticed:

- The attribute syntax makes it easier for compilers to either take into account or ignore the assumption, which will at worst lose out on some possible optimisation without impacting semantics;
- Because the assumption is source code, it can be picked up by linters and language severs;

- The caller cannot ignore the assumption assuming the attribute is supported, resulting in a compile error or in something that an execution-time analysis tool would pick up otherwise; and
  - This also serves to optimise the caller code without adding extraneous lines of code, as `expr` can in most cases be assumed to remain true (when passing by value, for instance).

5 Implementation

Seeing as the more complex C++ contract system will in any case be implemented by C++26-compliant compilers, having this feature should not pose a problem to C2y-compliant compilers. In any case, as was previously stated, calling a function compiled with `assume` would still work on previous standards in library-compliant contexts.

Though similar ones have been evoked, seemingly no C compiler extension allows for this exact feature. One could imagine possible function-like macros being able to replicate it, but it would certainly be non-trivial.

Even then, macro-based implementations would not be ideal, as they would:

1. allow for the library user to call the function without its underlying assumptions,
2. make compile-time optimisations impossible without extensions (`__builtin_assume`),
3. clutter the program with extraneous definitions if we have one macro per function,
4. be incompatible with style guides wherein parameters are unnamed in declarations,
5. generally worsen the user experience, as macros are not always well-supported by language servers,
6. make the assumptions messy and hard to modify; and
7. come with the usual points of failure of macros (CERT-PRE31-C, notably).

Indeed, it would be much more interesting for assumptions to be a standard feature, rather than being bound by the rules of macros.