Document number: P4021R1
Date: 2026-02-22
Audience: EWGI, EWG
Project: ISO Programming Language C++
Reply-to: Jonathan Grant jg@jguk.org

compile_assert – optimization-enforced conditions at compile time

Abstract
Following the recent std-proposals discussion of my 2023 compile_assert(), this paper introduces compile_assert(expression, message), a new C++ keyword for enforcing assertions at compile time within ordinary (non-constexpr) functions.
compile_assert provides advanced asserts at compile time (only in optimized builds), not runtime.

It is used for bounds checking, avoiding nullptr dereference, parameter validation and data validation at compile time. It only works in Optimized enabled compilers. All three major compilers (GCC, Clang, MSVC) are supported with a sample implementation. The GCC/Clang implementation use the a function error attribute:

```
main4.c:15:9: note: in expansion of macro ‘compile_assert’
 15 | compile_assert(i < buf_size, "check buf index within buffer bounds");
error
```

1. **Introduction**

This paper proposes `compile_assert(expression, message)`, a facility for expressing assertions that are enforced at compile time based on the compiler's ability to prove whether a given control path is reachable. It does this by relying upon the optimizer to remove code paths that are unreachable, if the compiler determines the failure path is reachable, the program is ill-formed and an error is emitted with file and line. compile_assert() has had a reference implementation and been in use since 2023 in code bases.

`compile_assert` enables expressing preconditions and invariants inside ordinary functions to provide compile-time diagnostics without introducing runtime overhead, programming within the rules of the constraints clearly expressed by the system architect as “design by contract”. This requires programmers to specify the constraints desired, and then guarantees consistency and supports formal verification. Of course I appreciate compile_assert finding issues means programmers will need to add defensive code for edge cases and malformed data validation.

No formal proof obligation would be imposed on the C++ compiler prove the validity of certain complicated conditions. The extent of reasoning would be left to the discretion of the compiler’s existing optimization framework and the resources it wishes to devote. (NB. This approach is only active in an optimized build).

Some compilers that are not advanced might not meet the requirement to validate, therefore I propose to only standardize the compile_assert keyword and leave each compiler to choose how much effort to spend confirming the constraints specified by compile_assert(). The benefit of this approach is users get a standard keyword, however they do not get a full warranty that everything will be enforced. There are many intractable programming questions, leading me to compare with Turing's Halting problem, whether the optimizer will finish running or continue to run for ever. There is no general algorithm that can determine whether an arbitrary Turing machine will keep running or halt.

If a compile_assert significantly slows compilation, it may be disabled once the code is stable, serving primarily as a development-time verification mechanism.

The compiler must see all call sites to provide conditions, meaning non-static could be unpredictable. It cannot magically reason about runtime data (eg the result of a DNS query), only conditions it can provide based on compile-time information. Typically constraints verify that nullptr is not de-referenced, that buffer accesses are within bounds, that a return code is checked for all actual returned values.

Examples in the repository show this being used for, pointer non-NULL preconditions, array index bounds, value ranges (eg 0 – 100%), offsets into buffers, API precondition checks across translation units.

Jump to section 9 to see the implementation.

2. **Motivation and Scope**

C++ currently provides:
static_assert - requires constant expressions.
assert - runtime check, calls abort() to terminate, optionally disabled.
Contracts – runtime checks, in progress.
Profiles – not yet standardized.

There is no general mechanism that allows compile time assertions inside ordinary functions from regular compilers. Having such a mechanism saves the need to run a separate static analysis tool. Given the compiler is generating the machine code, it's important the compile time assert is output from the compiler, not a separate static analysis tool that may or may not determine control flow the same way. There's limited visibility of what static analysis tools detect, they would need a dead code removal optimization too, as they do not output assembly like a compiler does.

Does not require the predicate to be a constant expression like static_assert, which requires everything to be axiomatic.
Produces compile-time diagnostics with file and line information
Has zero runtime cost.

In many cases, the optimizer can determine that certain branches are unreachable or that specific conditions are always true or always false.

This proposal exposes that capability for user-written assertions.

The intended audience includes:
Library authors
Security-sensitive systems developers
Low-level infrastructure code
Embedded systems programmers

The intended commercial audience:
Aviation
Automotive
Medical

The feature is not intended to replace runtime validation. It is intended to prevent code that provably violates invariants from compiling successfully. Some conditions depend on runtime data and cannot be established through compile-time reasoning and must therefore be validated during execution rather than at compile time.

3. **Design Goals and Non-Goals**

Goals:
Zero runtime cost.

Usable inside non-constexpr functions.
Leverages existing compiler analysis.
Produces clear diagnostics (file and line)
Requires no new core syntax beyond a new statement form.
Non-Goals:
Not intended to replace static_assert, assert.

4. **Proposed Design**

The proposal introduces the following syntax:
compile_assert(expression);
compile_assert(expression, message);

The ‘message’ is optional, if specified, it can be `""` (empty string), or nullptr, in which case the compiler would not output a message.

Semantics:
compile_assert(Expression, message) requires that the compiler prove that Expression cannot evaluate to false along any reachable execution path.

If the compiler determines that a failure path is reachable, the program is ill-formed.

If the compiler can prove that all reachable paths satisfy the condition, the program is well-formed.

The compile_assert construct has no runtime effect. Where a condition is not met, the translation unit (object) will not be produced as the error is fatal.

5. **Design Rationale**

compile_assert() relies on the compiler's optimizer and control-flow analysis.

Specifically:

Constant propagation
Dead-code elimination
Reachability analysis
Branch pruning

static_assert does not require the condition to be a constant expression. Instead, it is evaluated in the context of the optimizer's proven state at that point in the control-flow graph.

This is fundamentally different from static_assert, which operates purely in the constant-evaluation mode defined by the language.

Modern optimizing C++ compilers already eliminate unreachable branches and perform inter-procedural constant analysis. compile_assert() formalizes this capability into a portable language facility with a standardized keyword.

Keeping compile_assert separate from static_assert preserves the conceptual distinction between constant evaluation, and the different way that compile_assert relies on control flow optimization.

6. **Interaction With Existing Features**

No interaction with costexpr, concepts, templates, modules.
Contracts express runtime checks, compile_assert() is enforced at compile-time.

7. **Implementation Experience**

A header-only implementation demonstrates this behavior by placing an ill-formed construct in a branch that the optimizer determines to be reachable.

Requires user source code or build to enable
```
#define __ENABLE_COMPILE_ASSERT__ 1
```
GCC defines `__OPTIMIZE__`
Without these, the macros compile out.
It defines `COMPILE_ASSERT_ACTIVE` which an application can check.

A user who receives a file within which they wish to disable compile_assert could also
```
#undef compile_assert
#define compile_assert(expression, message)
```

GCC and Clang both support __attribute__ ((error(message))), GCC since gcc-4.5.3 in 2011. The attribute is not standardized. They do both support [[gnu:error(message)]] which can also be used.

compile_assert should not impact control flow, as it compiles out when expressions are true. Of course comparing assembly would be advisable.

Reliance on Optimizer, which in itself is not standardized is an issue. There will be nuances in the way compilers optimize control flow. Test case main24_b is validated by

GCC but not Clang. GCC appears to reason the result of sqrt(). No two compilers are alike.

Programmers do not require an extra static analyzer, as the compiler's Optimizer is deployed for static analysis. compile_assert is not active in a build with no optimization.

Functions may need to be in an anonymous namespace, or static so they cannot be called from external translation units (if they can be called, and they do not check their own parameters before any compile_assert() it would still error).

8. **Impact on the Standard**

This proposal introduces a new statement form:

compile_assert(expression, message);

The expression need not be a constant expression.

This feature:
It has no runtime impact.
No ABI impact
Does not change overloads

The primary specification work would define:

What constitutes a reachable failure path, that the implementation must diagnose when such a path exists. It may be clearer to only standardize the compile_assert keyword, and leave the implementation to the compiler.

9. **Implementation**

While there is no compiler supporting this feature directly, I created the following compile_assert macro for GCC and Clang:

```
#ifdef __GNUC__
#if defined(__OPTIMIZE__) && defined(__ENABLE_COMPILE_ASSERT__)
#define GCC_COMPILE_ASSERT
#define COMPILE_ASSERT_ACTIVE
#endif // defined(__OPTIMIZE__) && defined(__ENABLE_COMPILE_ASSERT__)
#endif // __GNU__

#ifdef GCC_COMPILE_ASSERT

/**
 * @brief Function to stop compilation with an error message if a
compile_assert condition is not satisfied.
 * There is no implementation as it is only used to stop the compiler.
 * @see compile_assert
 */

/**
 * @def compile_assert
 * @brief Macro for compile-time assertion in optimized builds.
 * @param expression The compile-time condition to be checked.
 * @param message A description of the assertion (unused).
 */
#define compile_assert(expression, message) \
    do { \
        void _compile_assert_fail() __attribute__
((error(message))); \
        if (!(expression)) { \
            _compile_assert_fail(); \
        } \
    } while (0)

// NB, would rather pass NULL to this.
#define compile_assert0(expression) compile_assert(expression, "")

#else
#define compile_assert(condition, description)
#define compile_assert0(expression)
#endif
```

Compilers that do not support GCC’s error attribute could stop the compilation of the translation unit another way, eg inline assembler of an instruction that does not exist, eg asm(“stop_build”); this also works on GCC.

MSVC implementation relies on a unique missing symbol indicating the constraint was not met, requires a macro from the build system with the filename. (A simpler macro could just use __LINE__).

```
C:\dev\> cl /DCOMPILE_FILE=__FILE_msvc18_cpp_

#define MERGE2(a,b) a##b
#define MERGE1(a,b) MERGE2(a,b)
#define MERGE3(a,b,c) MERGE1(a, MERGE1(b,c))

#define compile_assert(expr, message) \
do { \
    if (!(expr)) { \
      extern void MERGE3(_compile_assert, COMPILE_FILE,
__LINE__)(); \
      MERGE3(_compile_assert, COMPILE_FILE, __LINE__)(); \
    } \
} while (0)
```

The output shows the file and line the constraint was not met:

```
error LNK2019: unresolved external symbol "void __cdecl
_compile_assert__FILE_msvc18_cpp_23(void)"
```

MSVC alternatives
* `dumpbin /DISASM` look for the call to `_stop_compile()`
* Compile `cl /FAs` (assembly with source), look through for the call to _stop_compile()

Several alternative mechanisms exist for deliberately causing compilation or linkage failure in the presence of a violated constraint. One approach is to emit an invalid inline assembly instruction in the error path, relying on the assembler to diagnose the failure and report the corresponding source location. Another technique embeds a distinctive string in the object file and uses a post-compilation build rule (e.g. via make) to scan generated assembly or object output for that marker. A further method, commonly used in MSVC environments, is to reference a deliberately undefined external symbol in the error case, thereby triggering a link-time failure.

10. **Example source**

Example 1:

```
static void log_message(const char * p)
{
    compile_assert(p, "check not nullptr");
    printf("%s\n", p);
}

void output_string(const char * ptr)
{
    // NB. The following line is needed
    //if(nullptr != ptr)
    {
        log_message(ptr);
    }
}
```

Example 2 main4.c:

```
int main()
{
    const int buf_size = 4;
    char buf[buf_size];

    for(int i = 0; i != 5; ++i)
    {
        // will fire, as out of bounds
        compile_assert(i < buf_size, "check buf index");
        buf[i] = 3;
    }
}
```

Example 3 main17.cpp:

```
// force calling code to check for divide-by-0
static int divide(int num, int denominator)
{
    compile_assert(denominator != 0, "divide by zero");
    return num / denominator;
}

int main(void)
{
    int num = 10;
    int result = divide(num, 0);

    return result;
}
```

If the compiler can prove that the pointer in example 1 is always valid at the assertion site (because the negative branch returns), the program is well-formed.

If a reachable path exists where ptr may be nullptr, the program is ill-formed.

10. **LTO – Link Time Optimization**

One approach is to enforce `compile_assert(handle >= 0)` on a function such as `api_function(int handle)` at the API boundary while deferring final validation to the linker.

Inside `api_function(int handle)`, the assertion could be implemented as a call to a `[[deprecated]]` function when `(handle<0)`. This would emit a warning during object generation if the failure path is instantiated. The linker could then determine whether any call to api_function() with a negative handle actually remains in the final program.

It feels more rational for the programmer just to put the if(handle>=0) in api_function(int handle) as best practice, the Optimizer could then eliminate redundant checks. A programmer may validate parameters multiple times, the compiler's Optimizer may remove redundant checks, so there is no extra runtime cost and guarantees safety.

This combines early diagnostics with whole-program validation, though it depends on optimization and symbol elimination behavior, which may vary across compilers and those with LTO features.

11. **Sample examples and tests**

The reference link shows various examples have been incorporated in the repository, and specifically there is a testsuite folder which outputs PASS or FAIL for tests which supports Clang and GCC, just type `$ make`

main.c Argument validation within a static function

main2.c Validating arguments before they are passed to function

main3.c - illustrates the use of compile_assert to validate that a given percentage falls within the acceptable range of 0 to 100%.

main4.c - demonstrates compile_assert ensuring all indices accessing an array remain within the specified bounds of the array.

main5.c - demonstrate compile_assert checking array access via another array of offset indices into that array are within bounds.

main6.c - demonstrate compile_assert checking a TGA image data file header is valid.

main7.cpp - demonstrates using compile_assert to validate the size of an std::string object.

main9.c - demonstrate compile_assert checking with multiple conditions.

main10.c - demonstrate compile_assert checking array ranges, based on values computed at runtime.

main11.c - demonstrate compile_assert checking array ranges, based on values read from a file to avoid a buffer overflow.

main12. c - demonstrate compile_assert checking an offset resolved to a pointer is within the range bounds of a buffer (avoids buffer overruns) at runtime.

main13.c - demonstrates how compile_assert can be used with multi file projects. The two files are compiled to objects, and then linked.

main17.cpp - demonstrate divide by zero caught by compile_assert.

Example output:

```
In file included from main4.c:5:
main4.c: In function 'main':
<snip>
main4.c:15:9: note: in expansion of macro 'compile_assert'
   15 |         compile_assert(i < buf_size, "check buf index");
      |         ^~~~~~~~~~~~~~

In file included from proposal.c:4:
In function 'log_message',
    inlined from 'main' at proposal.c:16:5:
<snip>
proposal.c:9:5: note: in expansion of macro 'compile_assert'
    9 |     compile_assert(p != NULL, "check not null");
      |     ^~~~~~~~~~~~~~

$ make
gcc -Wall -Wextra -O3 -std=c11 -c -o main13.o main13.c
In file included from main13.c:8:
main13.c: In function 'main':
<snip>
main13_api.h:14:5: note: in expansion of macro 'compile_assert'
   14 |     compile_assert((str != NULL), "cannot be NULL"); \
      |     ^~~~~~~~~~~~~~
main13.c:16:5: note: in expansion of macro 'log_api'
   16 |     log_api(str);
      |     ^~~~~~~
make: *** [makefile:17: main13.o] Error 1
```

Adjacent string literals can be concatenated, so can even write:
```
compile_assert(condition, "main1_a check not null in: "
__FILE__);

error: call to '_compile_assert_fail' declared with attribute
error: main1_a check not null in: main1_a.c
```

12. **Compilers supported**

The sample implementation of compile_assert works in the big three compilers: GCC, Clang and MSVC.

14. **Notes**

Static analysis tools can also check the constraints specified by each compile_assert keyword.

compile_assert() isn't suited for everything. It doesn't always "drop in" to replace assert() and other required runtime checks. It's there to be used by a systems architect to spec out APIs and functions, specifying the constraints for the programmer's implementation to satisfy.

15. **Alternative keyword name instead of compile_assert**

Feedback has been there may be a better name than compile_assert. When I considered, I looked at existing keywords. I know static_assert is at compile time, and consteval is always at compile time, constexpr is sometimes at compile time, here are some alternative suggestions:

enforce()
require()
guarantee()
verify()
compiler_assert()
optimizer_assert()
condition_assert()
consteval_assert()
const_assert()
const_constraint()
eval_constraint()
require_const()
prove_constraint()
unreachable_assert()
expr_constraint()
invariant_assert()
condition_check()

15. **Other approaches considered**

When compile_assert is implemented as a macro, the reported file and line number are accurate. If it is changed to an inline function, the reported file and line number instead refer to the inline function itself so it is retained as a macro.

Tried to somehow get a static_assert() to work, by hiding in macros, structs, but it was always evaluated against if the conditions could be determined const before the optimizer could optimize out those code paths it would not reach.

MSVC has __assume() and C++23 has using the statement attribute [[assume]], I could not get either to work.

There are other builtin functions in GCC, some may achieve the a similar result, eg specifying some invalid alignment in an error case of an if(condition) check that the Optimizer otherwise removes.

I provide examples that show how invalid asm() can be used, as this is only checked after the compiler has Optimized out redundant code paths.

I researched using compile_assert together with `__builtin_constant_p` to determine if the expression was constant; however, the results were not reliable. Likewise `__builtin_choose_expr` requires a constant expression so cannot be used. None of the old static assert style macros work (invalid enums, arrays with negative values), because they are all evaluated before the Optimizer has removed the failure cases that were validated.

Calling an external error function that does not exist causes the linker to output the file and line location of the compile_assert constraint failure. This is the method used with MSVC.

GCC has supported __attribute__((error("message"))) since version 4.3 in 2008 , since GCC 5 in 2015 `[[gnu::error(message)]]` support was added.

Changing from `[[gnu::error(message)]]` to `[[deprecated(message)]]` would make compile_assert flexible as a warning, or can be made an error by `-Werror=deprecated-declarations`

MSVC supports since `[[deprecated(message)]]` C++14. Could consider changing the MSVC macro.

I explored and shared a GSL `not_null` that worked at compile time, it's a ticket on GSL github. compile_assert has `compile_assert_never_null()`. main16.cpp demonstrates a C++ templated class never_null_ptr(T* ptr) that is guaranteed to never hold a nullptr at compile time. The conventional class never_null_ptr(std::nullptr_t) delete; approach only identifies those places where the nullptr is being passed directly

C++ code typically has `#include <stdexcept>` and variations upon throw `std::invalid_argument("Error cannot be nullptr");` If a compiler's static analyzer could utilize this information the compiler could also output build errors when constraints are not met.

Tried many approaches with consteval and constexpr but could not get gcc to stop the build based on obvious conditional checks in C++ as it is evaluated before the Optimizer prunes redundant code paths.

Tried static_assert, this simple example always triggers the static_assert, so is not usable because it is evaluated before the Optimizer can remove this check.

```
const int a = 0;
if(a == 1)
{
    static_assert(sizeof(int) == 0);
}
```

static_assert(false); even stops a build in a constexpr function that is never called. Lambda functions don't provide a workable approach either, as the compiler evaluates static_assert as soon as it sees it, even if inside a lambda function.

std::unreachable was introduced in C++23, compilers do not verify control flow never reaches that point. If control flow does reach that point, behavior is undefined.

In safety-critical systems, deliberately removing certain symbols, eg. `realloc`, to ensure that any unintended use is detected at link time, thereby establishing a hard enforcement boundary within the toolchain rather than relying on code review or external static analysis. The MSVC implementation of compile_assert() follows a similar principle by placing a call to a deliberately undefined symbol from the exact code location that should be unreachable, causing a link-time failure if the compiler cannot eliminate that path. The same approach could be applied with an extern to set a missing variable.

16. **Runtime vs compile-time constraints**

Some conventional development practices favor immediate failure mechanisms, such as segmentation faults (SEGV), runtime assertions that trigger traps, or debugger breakpoints. While effective aids during development, these approaches rely on execution with known inputs and expose only the defects encountered under those specific conditions.

Their limitations become apparent when the software is exposed to unknown (untrusted) data or inputs. A missing resource (eg. network connection timeout) may lead to a nullptr being passed on to an API (as the software was always on a reliable network connection during testing), or a truncated or corrupted image download may result in abrupt termination if not properly handled. Hard disks and portable mass storage devices do fail, and may leave files truncated or corrupted. Of course miscreants may craft malicious files (eg PDF, GIF, CUE sheet) that exploit such APIs as we see.

For safety-critical systems such as an x-ray machine or avionics software, these crashes are not merely inconvenient but potentially hazardous. These contexts therefore require robust validation, defensive programming, and fault-tolerant design rather than reliance on high performance mechanisms alone.

ALGOL 68 introduced null references, and implementations allowed the checks to be disabled at compile time (the inventor Sir Tony Hoare calls it his “billion-dollar mistake”. Pascal also has nil pointers.

17. **Additional compiler implementations**

Straightforward to add support for additional compilers. Either use the MSVC style approach of calling a missing function `_compile_assert_fail`, linker then reports on constraint failure. Call `asm("invalid instruction"),` or use `[[deprecated]].`

17. **Acknowledgments**

Thanks to those who have discussed compile_assert() with me.
Including Jonathan Wakely and Alejandro Colomar on the function attribute error approach, `[[deprecated(message)]]` (a way to have the information as purely a warning), and how to get a linker error with file and line.

18. **References**

“On Computable Numbers With an Application to the Entscheidungsproblem” May 1936, Alan Turing.
https://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf

compile_assert reference implementation as a header and examples
https://github.com/jonnygrant/compile_assert/blob/main/README.md

attribute error ("message") aka [[gnu::error(message)]] [[deprecated(message)]]
https://gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html

Sir Tony Hoare introduced Null references in ALGOL W back in 1965, describing it as “a billion-dollar mistake”.
https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/

GSL not_null proposal (at compile time)
https://github.com/isocpp/CppCoreGuidelines/issues/2071

19. **External resources**

ISO 26262 Automotive Functional Safety.
ISO 27001 Information security, cybersecurity and privacy protection.
ISO 29147 Information technology Security techniques.
ISO 30111 Information technology Security techniques Vulnerability handling processes.

Secure Software Development Framework SSDF https://csrc.nist.gov/Projects/ssdf

20. **ChangeLog**

Since previous R0 (2026-02-22)

a) MSVC is now a supported compiler.
b) Notes section, alternative approaches that were considered.
c) compile_assert ‘message’ string is now displayed on the output in the reference GCC/Clang implementation.
d) Added "Runtime vs compile-time constraints" section
e) Tests added that verify constraints before and after fixes at compile time.
https://github.com/jonnygrant/compile_assert/tree/main/testsuite

21. **Revision history**

2026-02-22: R0
2026-02-28: R1