

# compile\_assert() – control-flow enforced conditions at compile-time

Proposal for C2x

Date: 2026-03-09

Document: n3846

Audience: WG14

Proposal Category: New Features

Target Audience: General Developers, Safety Critical Software, Compiler/Tooling Developers

Reply-to: Jonathan Grant [jg@jguk.org](mailto:jg@jguk.org)

## Abstract

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 a way to specify constraint predicates that are validated at compile time and reported to the user via a diagnostic.

The new keyword is used for bounds checking, avoiding NULL dereference, parameter validation and data validation at compile-time. All three major compilers (GCC, Clang, MSVC) are supported with a sample implementation macro I published in 2023 that works by re-purposing the Optimizer to identify failure branches that should be unreachable. The GCC/Clang implementation uses the a function error attribute. A brief example:

```
const int buf_size = 4;
char buf[buf_size];

for(int i = 0; i != 5; ++i)
{
    // will fire at compile-time due to being out of bounds
    compile_assert(i < buf_size, "check buf index within buffer bounds");
    buf[i] = 3;
}
```

## Compiler output:

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

## Contents

1. Introduction to `compile_assert`
2. Motivation and Scope
3. Design Goals and Non-Goals
4. Proposed Design
5. Design Rationale
6. Interaction With Existing Features
7. Implementation Experience
8. Impact on the Standard
9. Sample implementation
10. Example source

10. LTO – Link Time Optimization
11. Sample examples and tests
12. Compilers supported
13. Notes
14. Other approaches considered
15. Runtime vs compile-time constraints
16. Additional compiler implementations
17. Limitations
18. Acknowledgments
19. References

## 1. Introduction

This paper proposes:

```
compile_assert(expression);  
compile_assert(expression, message);
```

This is 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 through static analysis. The sample macro does this by relying upon the compiler 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 has been used in codebases since 2023

`compile_assert` enables the expression of compile-time preconditions and invariants inside ordinary functions, as well as postconditions. This provides 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 desired constraints and helps guarantee consistency while supporting formal verification. A corollary of adding `compile_assert` and finding issues means programmers will need to add defensive code for edge cases and malformed data validation. `compile_assert` helps prevent undefined behavior by bringing potential UB to the programmer’s attention.

Compilers may implement the keyword as a warning, with the option for it to be a hard compile error.

No formal proof obligation would be imposed on the compiler to prove the validity of complex conditions. The extent of reasoning would be left to the discretion of the compiler’s existing static analysis framework and the resources it wishes to devote. (NB. This sample 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 static analysis will finish running or continue to continue to run forever. There is no general algorithm that can determine whether an arbitrary Turing

machine will keep running or halt. Some program behaviors cannot be determined statically.

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 NULL 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` - a runtime check that calls `abort()` to terminate and may be disabled.

There is no general mechanism that allows flexible compile-time assertions inside ordinary functions from regular compilers based on control flow analysis. Having such a mechanism saves the need to run a separate static analysis tool. Since the compiler generates the machine code, it is important that the compile-time assertion originates from the compiler; rather than a separate static analysis tool which may or may not determine control flow the same way. There is limited visibility of what static analysis tools detect, they would need unreachable dead code removal, and a way to output their analysis, 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 compiler 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 argument shall be a string literal.

Optionally: as an additional possibility 'message' could be a user-generated string-like object as their diagnostic message (given the compiler knows these values). It could be any expression that produces a character sequence.

Eg. `compile_assert(idx < max_size, fmt("Error: idx {} < {} max_size", idx, max_size));`

A pseudo printf

```
compile_assert(idx < max_size, format("Error: idx %zu < %zu max_size", idx, max_size));
```

Could be supported, interpreted by the compiler solely for formatting the diagnostic message. That would require the compiler to have a parser to generate the string literal. The 'message' is optional, if specified, it can be "" (empty string), or NULL, in which case the compiler would not output a message.

Desired diagnostic output:

```
main.c:15:9: error: compile_assert failed  
 15 | compile_assert(i < buf_size, "Error: idx %zu < %zu max_size");  
main.c:15:9: message: "Error: idx [0 .. 4,294,967,295] < 4 max_size"  
main.c:15:9: note: the comparison reduces to '( i [0 .. 4,294,967,295] < 4)'  
error
```

Semantics:

`compile_assert(expression, message)` requires the compiler to 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()` sample relies on the compiler's control-flow analysis.

Specifically:

- Constant propagation
- Dead-code elimination
- Reachability analysis
- Branch pruning

`static_assert` requires the condition to be a constant expression. `compile_assert` is fundamentally different from `static_assert`, which operates purely in the constant-evaluation mode defined by the language. Instead, it is evaluated in the context of the compiler's proven state at that point in the control-flow graph.

Modern 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 analysis.

## 6. Interaction With Existing Features

No interaction with `constexpr`, `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 compiler determines to be reachable. The sample implementation is only active in enabled builds.

Requires the user source code or build system to enable it.

```
#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 version 4.5.3 (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 compiler dead code analysis, which in itself is not standardized is an issue. There will be nuances in the way compilers consider control flow. No two compilers are alike.

Programmers do not require an extra static analyzer, as the compiler is deployed for static analysis.

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. Sample 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 COMPILER_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 certain 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, ie. `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 /DCOMPILER_FILE=__FILE__ msvc18_c_

#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, COMPILER_FILE,
__LINE__) (); \
MERGE3(_compile_assert, COMPILER_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_c_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()`

A makefile can determine the macro `COMPILER_FILE := $(subst .,_,$(notdir $(SRC)))`

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.

With GCC we can verify that any fixes made are still present in the object file by looking at intermixed assembly output by passing `-S -fverbose-asm file.c` or `-save-temps` Or use `objdump -d -S -M intel` makes more readable. Or `(gdb) disassemble /s main`

## 10. Example source

### Example 1:

```

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

```

```

void output_string(const char * ptr)
{
    // NB. The following line is needed
    //if(NULL != 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:

```

// 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;
}

```

```

example3.c:14:5: macro 'compile_assert'
compile_assert(div != 0, "divide by zero");
      ^~~~~~

```

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 NULL, 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, LTO could then eliminate redundant checks. A programmer may validate parameters multiple times, the compiler may remove redundant checks, so there is no extra runtime cost and guarantees safety. Parameters could be validated 3 times and compile out 3 times. In the future when a change is introduced, the issue could be identified quickly during compilation.

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

On the repository `compile_assert/experiments/lto` provides an example where the constraint expressed in `lto2.c` is reported during final LTO linking.

## 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.

`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.

## 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.

## 13. Notes

The proposed keyword `compile_assert` was chosen as it clearly describes that the `assert` is evaluated at "compile time" and it does not specify the method the constraint is validated. The constraint may be validated by a static analyzer, a compiler's control-flow analysis, or an interpreter.

C may standardize as `__Compile_assert()` which can then be defined as `compile_assert()`.

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.

#### 14. 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 compiler could remove those code paths it would not reach.

MSVC has `__assume()` and GCC has `__builtin_assume()`, I could not get either to work.

As an alternative, 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 compiler otherwise removes.

I provide examples that show how invalid `asm()` can be used in the error condition case, so if present in generated assembler it will fail and give the file and line number from the original source code.

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 compiler 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 the sample implementation from `[[gnu::error(message)]]` to `[[deprecated(message)]]` would change `compile_assert` to be flexible as a warning, or it could be made an error by `-Werror=deprecated-declarations`

Could consider changing the MSVC macro in the sample implementation to use `__declspec(deprecated("message"))`.

Tried many approaches with `constexpr` and `constexpr` but could not get gcc to stop the build based on obvious conditional checks as it is evaluated before the compiler 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 compiler 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.

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.

GCC added `-param=min-pagesize=` in 2023, which should invalidate memory access below eg 4KB (would identify NULL pointer dereferences that get into generated assembly), I could not get it to work in my research.

## 15. 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 NULL 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) that exploit such APIs as we see in the news.

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.

## 16. 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. Limitations

The sample implementation of `compile_assert` requires a compiler with inlining, if a function `get_index()` has attribute `[[gnu::noinline]]` `compile_assert` cannot reason about the use of the value it returns to satisfy any constraint imposed. If `-fno-inline` is passed, it prevents many cases from being identified reasoned.

Where a function is not-inlined in an build, it will likely report a failure for each constraint expressed via `compile_assert`. You can then decide whether to inline the function directly, refactor the function for simplicity, or isolate the checks into a smaller helper function that the compiler can inline; in any event, the issue is clear, as the object file fails to link.

ie. `complex_function(Project * data)` contains a `compile_assert()` that fails to validate a constraint predicate as true. Refactor so the `compile_assert` can validate the constraint:

```
const bool result = validate_complex_data(Project * data);
if(result) = complex_function(Project * data);
```

Shared library (dll, so) are not yet demonstrated, they would need to validate their arguments to satisfy their internal `compile_assert` checks on their public APIs.

## 18. Acknowledgments

The author thanks those who have discussed `compile_assert()`. 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.

## 19. References

[1] “On Computable Numbers With an Application to the Entscheidungsproblem” May 1936, Alan Turing.

[https://www.cs.virginia.edu/~robins/Turing\\_Paper\\_1936.pdf](https://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf)

[2] `compile_assert()` reference implementation as a header and examples  
[https://github.com/jonnygrant/compile\\_assert/blob/main/README.md](https://github.com/jonnygrant/compile_assert/blob/main/README.md)

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

[4] `int __builtin_constant_p (exp)`  
[https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html#index-\\_005f\\_005fbuiltin\\_005fconstant\\_005fp](https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html#index-_005f_005fbuiltin_005fconstant_005fp)

[5] 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/>

## 21. 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>

ISO C++ Proposal P4021 `compile_assert()`, currently R1

## 22. ChangeLog

R0 initial proposal

## 23. Revision history

2026-03-09: R0