

Abstract

`compile_assert` provides advanced asserts at compile time, not runtime. Used for bounds checking, parameter validation, data validation at compile time.

```
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 by relying upon the optimizer to remove code paths that are not reachable. `compile_assert()` has had a reference implementation and been in use since 2023 in code bases.

Unlike `static_assert`, which requires a constant expression, `compile_assert()` relies on the compiler's optimizer and control-flow analysis to determine whether the asserted condition can ever evaluate to false. If the compiler determines that a failure path is reachable, the program is ill-formed and the compile fails, with a file:line number, and an error message.

This enables expressing preconditions and invariants inside ordinary functions to provide compile-time diagnostics without introducing runtime overhead. It can also be used for design by contract approaches. It requires programmers to specify the constraints desired, and then guarantees consistency and supports formal verification. Although I appreciate `compile_assert` finding issues means programmers will be required to add defensive code.

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 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, as they do not output assembly like a compiler does).

Does not require the predicate to be a constant expression like `static_assert`.

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 feature is not intended to replace runtime validation. It is intended to prevent code that provably violates invariants from compiling successfully. Some data is only known at runtime, and must be validated while running.

There is an implementation with examples listed in the references section.

While there is no compiler supporting this feature directly, since 2023 I have used the following approach:

```
#if defined(__OPTIMIZE__) && defined(__ENABLE_COMPILE_ASSERT__)

void _stop_compile() __attribute__((error("'compile_assert  
error detected'"))));

/**
 * @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 { \
        if (!(expression)) { \
            _stop_compile(); \
        } \
    } while (0)

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

When standardised, I suggest naming `_Compile_Assert()` furthermore a macro to `#define _Compile_Assert compile_assert`
This way, a user can `#undef compile_assert`

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:

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

The 'message' is optional. If specified it can be "" (empty string), or nullptr (and be ignored).

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.

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 lines are 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:

```

#define __ENABLE_COMPILE_ASSERT__ 1
#include "compile_assert.h"

// force calling code to check for 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.

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

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

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

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

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. (It relies upon the Optimizer, so is turned off in unoptimized builds)

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

Requires user source code or build to enable

```
#define __ENABLE_COMPILE_ASSERT__ 1
```

GCC sets `__OPTIMIZE__`

Without these, the macros compile out. Makes it straightforward for anyone with an older compiler or wishing to not use `compile_assert`.

A user who receives a file within which they wish to disable `compile_assert` could also

```
#undef compile_assert
```

```
#define compile_assert(expression, message)
```

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

This technique depends on:

The compiler diagnosing invalid constructs in reachable code,

The optimizer removing unreachable failure branches.

Existing compilers such as GCC and Clang support this pattern today.

GCC has had attribute error since gcc-4.5.3 in 2011, so this may work with older compilers.

The proposal does not mandate a particular optimization strategy; it specifies only the observable effect: a program is ill-formed if a failure path is reachable.

`compile_assert` should not impact control flow, as it compiles out when expressions are true. Of course checking assembly will show for sure.

Reliance on Optimizer, which in itself is not standardized is an issue. There will be nuances in the way compilers optimize control flow, although I have not identified issues.

No extra static analyzer needed, as the compiler's Optimizer is deployed for static analysis.

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.

I accept that all compilers are different, and it is impossible at present to have this active in an unoptimized build.

9. Examples

The reference link shows various examples.

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
```

10. Notes

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

10. Acknowledgements

Thanks to those who have discussed `compile_assert()` with me over recent years. Including Jonathan Wakely on the function attribute error approach, and Alejandro Colomar and others.

11. References

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

attribute error ("message") <https://gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html>

11. 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>