constant_assert

Document #:	R4019R1
Date:	2026-01-14
Project:	Programming Language C++
Audience:	EWG, EWGI

Reply-to:	Jonas Persson <jonas.persson@iar.com>

Contents

1 Introduction **1**

2 Revisions **2**

3 Proposal **2**
3.1 constant_assert(expr[, message]) . . . 2

4 Use cases **2**
4.1 Safer assert . . . 2
4.2 Inspect optimizer abilities . . . 2
4.3 Verified `[[assume()]]` . . . 2
4.4 Contract semantics . . . 2

5 Unspecified behaviour **2**

6 Non optimizing modes **3**

7 Interaction with UB **3**

8 Side effects **3**

9 Implementation **3**
9.1 Why not a library function? . . . 4
9.2 LTO . . . 4

10 References **4**

1 Introduction

The optimizer has much of the same functionality as a static analysis tools. It could perform the same verifications as a separate tool, making correctness checks more accessible to the programmer, and at the same time provide better control over the optimizations. What is lacking is a simple way for a programmer to interact with it.

This paper introduces a new type of asserts. We already have `assert()` that verify a statement at runtime. It is costly performance wise and introduces possible terminations into the application.

Then we have `static_assert()` that check things in at compile time. No performance overhead and no terminations. But it can only check a limited set of statements, strictly defined by the language.

A related feature `[[assume()]]` is sometimes used when we think we know the truth and want to make sure the optimizer know, hoping it will improve performance. With the caveat of introducing UB if our guess is wrong.

But there is one missing to this set. An assert, at compile time, proved by the optimizer based on all its knowledge. What we get is an assert that can prove a lot more than a `static_assert()`, without the performance overhead and termination risk of `assert()` and with the same effect as an `[[assume()]]` without the risk of UB.

2 Revisions

2.0.0.1 Revision 1

Fixed error in the implementation.

Added a user supplied error message

Add section on UB interaction.

Add section on side effects

3 Proposal

3.1 constant_assert(expr[, message])

if expr is not dead code, not constant folded and do not evaluate to true at code generation phase, the program is ill formed.

A user supplied message is an optional second argument.

The diagnostics should tell if the reason is that the expression is not constant or if it is false.

4 Use cases

4.1 Safer assert

Check for all possible values without a runtime cost and without risk of terminating the application.

4.2 Inspect optimizer abilities

This is a nice but rather niche use. Having a way to check that the optimizer work as expected without resorting to reading assembler output can save many hours when fine tuning code or identifying regressions.

4.3 Verified `[[assume()]]`

Macros that assert in debug and assume in release is not uncommon, and also not safe. Sometimes such check can be replaced by a `constant_assert()`, giving the same optimization hint but without relying on anecdotal proof and risk of UB.

4.4 Contract semantics

As a contract semantic, this will unlock some really powerful uses for both safety and performance as contracts can be forced to resolve at compile time. Use of this in contracts brings a whole lot of intricate details so it is postponed to a separate paper after this basic feature has settled.

5 Unspecified behaviour

This feature is by design based on unspecified behaviour. The part where the compiler figure out if the expression can be resolved at compiler time is not possible to specify.
It will differ between compilers, compiler options and context. But once it has been decided that it is known

at compile time, the truth of the expression will be well deP4019R1_constant_assert.pdffined and evaluate the same everywhere.

This is how we want it. The idea here is to tap into the ingeniousness of the unconstrained optimizer and use it as a tool for correctness.

`constant_assert` will most likely fail or succeed differently between compilers, but hopefully each compiler brand will improve over time, so once the *constant_assert* has passed with a compiler, it will continue to pass with newer versions.

A good thing is that it will make differences in compilers visible and perhaps spur some competition.

6 Non optimizing modes

Compilers have different optimization levels and many `constant_asserts` will need maximum optimizations.
This mean that `constant_assert` will have to be wrapped and replaced with something else in unoptimized builds.
But once in place this will hopefully drive optimizers to work differently. `constant_assert` should be seen as a primary optimization driver and the optimizer should start with proving these checks on all levels of optimizations. Once this is done it can use what has been proven to subsequent optimization steps, or throw it away if the no optimization is desired.

7 Interaction with UB

A compiler is allowed to assume UB never happen, and based on that the optimizer is allowed to remove code that lead to or follows UB. A `constant_assert` may fall into this trap, so it must be clear how it interacts with UB.

`constant_assert` happen at compile time, before any UB, preventing UB in the following code.

```
    constant_assert(a!=0);
    y = x/a;
```

The `constant_assert` will fail if `a` can possibly be 0. The optimizer can optimize on the assert but not on the UB.
The simplest solution is that `constant_assert` behave as a runtime assert would, preventing any UB in following code.

8 Side effects

That the expression can be constant evaluated at compile time does not preclude the expression from containing side effects. The library implementation example below may produces side effects at runtime.
This may cause some unwanted costs and add restrictions on where a `constant_assert` can be used, so this proposal suggest that evaluating a constant_assert do not produce any side effects at runtime.

```
  int i = 3;
  constant_assert(++i == 4);
  constant_assert(i == 3);
```

9 Implementation

constant_assert has not been added in any implementation yet, but such a check is already implementable in user code.

Gcc implements a ___builtin_constant_p(expr) [constant_p] that tells whether the expression has been constant folded by the optimizer or not.

This let us implement a prototype as a library function. ___builtin_constant_p has some problems that we can avoid by doing `constant_assert` as a dedicated language feature.

```
[[gnu::always_inline]]
constexpr void constant_assert(bool cond) {
  if (not __builtin_constant_p(cond))
    [] [[gnu::noinline, gnu::error("constant assert - not constant")]] () { }();
  else if (not cond)
    [] [[gnu::noinline, gnu::error("constant assert - false")]] () { }();
}
```

9.1 Why not a library function?

To extract the expression and present it in the compile output it would need be a macro, and we would need better ways to produce error messages.

Basing this on the constant expression query is bad. It already has uses as a way to check if code is optimized and if not replace with something else.
This mean it should strictly follow what the optimizer actually apply to the code.
For `constant_assert` on the other hand, we want the optimizer to do whatever it takes to prove it, even if it is not going to use the result to optimize the code.

With a library solution it is hard to control that no side effects are produced and that it behave correctly with UB.

9.2 LTO

With LTO, final constant folding and code generation may not take place until linking. Allowing `constant_assert` to be deferred to the link step would make it very hard for programmers to act upon and identify the cause of errors.
Is is suggested that `constant_assert` is always resolved at the compilation step, and a separate feature is later added for linker resolved assert if needed.

10 References

[constant_p] Gcc. Built-in Function: int ___builtin_constant_p (exp).
https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html#index-__005f__005fbuiltin__005fconstant__005fp