Proposal for C2Y

WG14 n3637

Title: Static assertions in expressions, v2 (updates n3538)

Author: Vincent Mailhol <mailhol.vincent@wanadoo.fr>

Date: 2025-07-07

Proposal category: Change

Target audience: Implementers, users

Abstract: Allow `static_assert` in expressions

Prior art: C23

Static assertions in expressions, v2 (updates n3538)

Reply-to: Vincent Mailhol <mailhol.vincent@wanadoo.fr>

Document: n3637

Date: 2025-07-07

This proposal extends the semantic of `static_assert` and allows it to be used as an operator which has type `void`. This way, `static_assert` can be used in expressions, typically when defining a function-like macro. If used as a declaration, the behaviour is unchanged.

This proposal updates n3538 by changing the type of `static_assert`'s result from `int` to `void`.

Change Log

2025-05-02 : n3538 initial version

2025-07-07 : change the type of `static_assert`'s result from `int` to `void`.

Table of Contents

1 Problem Description

When defining a function-like macro, it is sometimes useful to add compile time checks. For example, when writing:

```
/* Number with the nth bit set, starting count at zero */
#define BIT(type, n) ((type)1 << (n))
```

you may want to statically check that the argument `n` is within the range[1]

[`0`; `sizeof(type) * CHAR_BIT - 1`]

Performing such a static check within a function is impossible because the argument `n` would no longer be an integer constant expression. Even the as-yet-to-be-introduced `constexpr` functions wouldn't solve the issue entirely because these would not account for type polymorphism as a function-like macro would.

Currently, C does not offer a straightforward way to add such checks to macro definitions. Indeed, `static_assert` cannot be used in an expression because it can only be used as a declaration. Using it in an expression is invalid.

A few workarounds exist which we briefly describe in the following sections.

1.1 Create a constraint violation if the assertion fails

It is possible to perform static assertions in expressions by creating a constraint violation if the assertion fails and returning zero otherwise. The constraint violation can be, for example, an array or a bit field with a negative size. For example:

```
#define static_assert_int(cond) (!sizeof(char[(cond) ? 1 : -1]))
#define BIT(type, n) (                                          \
     static_assert_int(n >= 0 && n < sizeof(type) * CHAR_BIT) + \
          ((type)1 << (n))                                      \
)
```

If the condition is false, `static_assert_int` declares an array of negative size; breaking the compilation. Otherwise, `static_assert_int` yields the integer constant expression zero of type `int`.

The diagnostic message will be unrelated to the actual check which is being performed.

[1] Similar to clang or gcc's `-Wshift-count-negative` and `-Wshift-count-overflow` diagnostics. For this example, let's assume that the compiler may not have those diagnostics and the user wants to manually implement these.

A possible variation of above example is:

```
#define static_assert_void(cond) (void)sizeof(char[(cond) ? 1 : -1])
#define BIT(type, n)                                                \
    _Generic(static_assert_void(n >= 0 && n < sizeof(type) * CHAR_BIT), \
             void: (type)1 << (n))
```

static_assert_void is similar to static_assert_int except from the result type which has been changed from int to void.

1.2 Encapsulate the static_assert in a structure

While static_assert cannot be used in expressions, it can be used in structure declarations. By wrapping static_assert in a structure, it becomes possible to build a function-like macro similar to static_assert that can be used in expressions. For example:

```
#define static_assert_int(cond) \
    (!sizeof(struct {static_assert(cond); char a;}))
#define BIT(type, n) (                                      \
    static_assert_int(n >= 0 && n < sizeof(type) * CHAR_BIT) + \
        ((type)1 << (n))                                    \
)
```

To avoid declaring a structure of size zero (which is a GNU extension), a dummy char attribute is used. sizeof's value is negated so that static_assert_int yields the integer constant expression zero of type int.

The diagnostic message, while relevant, would be polluted by the wraparound logic.

As shown before, the example can be rewritten to return a void type instead of int:

```
#define static_assert_void(cond) \
    (void)sizeof(struct {static_assert(cond); char a;})
#define BIT(type, n)                                                \
    _Generic(static_assert_void(n >= 0 && n < sizeof(type) * CHAR_BIT), \
             void: (type)1 << (n))
```

1.3 Use GNU's compound statement expressions

The compound statement expressions (GNU extension) are the only method which allows the direct use of static_assert declarations. For example:

```c
#define BIT(type, n) ({                                              \
      static_assert(n >= 0 && n < sizeof(type) * CHAR_BIT); \
      (type)1 << (n);                                                \
})
```

The drawback is that the returned value is not an integer constant expression anymore and that this is not portable.

Consequently, existing workarounds are either non-trivial or nonstandard. Also, the compiler diagnostic message is polluted by all the wraparound logic and becomes less readable on some of these workarounds.

2 Prior work

Linux kernel BUILD_BUG_ON_ZERO* function like macros

Workarounds are commonly used, for example, in the Linux kernel to declare function-like macros which can be used to perform static assertions in expressions. For example:

- The `BUILD_BUG_ON_ZERO` function-like macro declares a bit field of negative size: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/linux/build_bug.h?h=v6.15#n16
- The `__BUILD_BUG_ON_ZERO_MSG` function-like macro wraps `static_assert` in a structure declaration: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/linux/compiler.h?h=v6.15#n197

Here, the current state of the art consists of having the macro yield the constant expression 0 of type `int` so that the result can then be added to another expression.

`shadow-utils` project

In the shadow-utils project, Alejandro Colomar declares the `must_be` function-like macro by wrapping `static_assert` in a structure declaration: https://github.com/shadow-maint/shadow/commit/10f31a97e2b2.

Here also, the `must_be` function-like macro yields the integer constant expression 0 of type `int`.

`cmp_int` project

The cmp_int project by Robert C.Seacord and Aaron Ballman also relies on encapsulating the `static_assert` in a structure to perform static assertion in a function-like macro, but, unlike the last two prior works, the value is casted to `void` and is then used as the left hand operand of the comma operator:

https://github.com/rcseacord/cmp_int/blob/f6a757b67e9958da08f21297835bfc45fbe1716a/include/cmp_int.h#L98-L103

3 Type and value

As described in the previous section, the type of static assertions is inconsistent: some implementations yield the integer zero while some yield `void`.

Yielding `void` has one drawback; the natural way to use the `void` type in an expression is to combine it with the comma operator. For example:

```
#define static_assert_void(cond) \
     ((void)sizeof(struct {static_assert(cond); char a;}))
#define BIT(type, n) (                                        \
    static_assert_void(n >= 0 && n < sizeof(type) * CHAR_BIT), \
    (type)1 << (n)                                            \
)
int arr[BIT(unsigned int, 2)];
```

However, above construct is not an integer constant expressions for two reasons:

- The comma operation is not allowed in integer constant expressions (cf. §6.6.1 ¶3)
- The `void` type is not allowed in integer constant expressions (cf. §6.6.1 ¶10)

Indeed, because `static_assert_void` yields `void`, `BIT` no longer returns an integer expression as `arr` is now a variable length array.

For this reason, the previous version of this proposal, n3538, discarded the idea of the `void` type and instead preferred to follow what the majority of the prior work did: yield the integer constant expression zero.

Following the discussions on n3538 on the reflector mailing list, many participants pointed out that the type `void` was more idiomatic. In this discussion, Martin Uecker pointed out that if the static assertion is passed to the controlling expression of a generic selection, the result is still an integer constant expression. This construct was already illustrated in sections 1.1 and 1.2 of this paper.

A final option is to have `static_assert` yield the integer constant expression 1. For example:

```
#define static_assert_int(cond) \
      (!!sizeof(struct {static_assert(cond); char a;}))
#define BIT(type, n) (                                        \
     static_assert_int(n >= 0 && n < sizeof(type) * CHAR_BIT) ? \
            (type)1 << (n) : 0                                \
)
```

This last option is mentioned for completeness but has not been encountered in prior art.

Following the discussions on the reflector list, this updated proposal gives `static_assert` the type void so that it can be used in conjunction with the comma operator.

4 Proposal

This proposal extends the semantics of `static_assert` by allowing it to be used as an operator which has type `void`. This way, `static_assert` can be used in expressions without the need for any of the previously described workarounds. For example:

```
#define BIT(type, n) (                                         \
      static_assert(n >= 0 && n < sizeof(type) * CHAR_BIT), \
      (type)1 << (n)                                        \
)
```

The future directions listed in section 6 would allow this construct to be an integer constant expression. As of now, a generic selection can be used as a workaround:

```
#define BIT(type, n)                                                    \
      _Generic(static_assert(n >= 0 && n < sizeof(type) * CHAR_BIT), \
               void: (type)1 << (n))
```

This proposal simplifies the use of static assertions in function-like macros. This is one step closer to making C a safe language.

This solution may overlap with the as-yet-to-be-introduced `constexpr` functions. `constexpr` functions would indeed at least solve the issue for when the argument type is known. To work with multiple types (typically scalar types), function-like macro remains useful. So, unless function-like macros are replaced by a new feature, the `static_assert` operator remains complementary with other future directions of C.

A block item containing only a `static_assert` directly followed by a semicolon is explicitly defined as a declaration. Consequently, the following construct, which otherwise would be ambiguous:

```
void func() {
       static_assert(1);
}
```

must be interpreted as a `static_assert` declaration. Otherwise, `static_assert` is an operator. For example:

```
void func() {
        static_assert(1), 0;
}
```

Prior to this change, `static_assert` could only be used as a declaration. This disambiguation ensures that the existing behaviour is unchanged. The semantic is only changed for constructs which were previously invalid. Preserving the existing behaviour guarantees that this is not a breaking change.

5 Proposed text

Proposed wording changes are against proposal n3525. If n3525 is superseded, modifications shall be reflected accordingly.

Subclause 6.5.4.1, paragraph 1

Replace n3525 subclause 6.5.4.1, paragraph 1 with the following text. The text in green contains changes while the **text in black** does not.

6.5.4 Unary operators

6.5.4.1 General

Syntax

1 *unary-expression:*

postfix-expression

++ *unary-expression*

-- *unary-expression*

unary-operator cast-expression

_Lengthof *unary-expression*

_Lengthof (*type-name* **)**

sizeof *unary-expression*

sizeof (*type-name* **)**

alignof (*type-name* **)**

static-assertion

unary-expression: one of

& * + - ~ !

Move subclause 6.7.12 to 6.5.4.6

In n3525, move subclause 6.7.12 to 6.5.4.6. The **Syntax** and the **Semantics** paragraphs are modified, the **Constraints** and **Recommended practice** paragraphs are left untouched. A new EXAMPLE paragraph is added to illustrate the use of static assertions in expressions. The text in green contains additions while the ~~strikeout text in red~~ contains definitive deletions. Text which is simply moved across sections is coloured in ~~strikeout purple~~ for the original location and in blue for the final location.

6.5.4.6 Static assertions

Syntax

1 *static-assertion:*

`static_assert` (*constant-expression* , *string-literal*)

`static_assert` (*constant-expression*)

Constraints

2 The constant expression shall be an integer constant expression with a nonzero value.

Semantics

3 A static assertion has no effect. If used as a unary expression, the result has type `void`.

Forward references: `static_assert` declaration (6.7.1).

Recommended practice

4 If the constraint is violated with an integer constant expression of value zero, the diagnostic message should include the text of the string literal, if present.

5 EXAMPLE When combined with the comma operator, static assertions can be used in expressions, typically in function-like macros.

```
#include <limits.h>

#define BIT(n) (                                          \
        static_assert(n >= 0),                            \
        static_assert(n < sizeof(unsigned int) * CHAR_BIT), \
        1U << (n)                                         \
)
```

(...)

~~6.7.12 Static assertions~~

~~Syntax~~

~~1 *static_assert-declaration:*~~

~~**static_assert** (*constant-expression* , *string-literal*) ;~~

~~**static_assert** (*constant-expression*) ;~~

~~**Constraints**~~

~~2 The constant expression shall be an integer constant expression with a nonzero value.~~

~~**Semantics**~~

~~3 A static assertion has no effect.~~

~~**Recommended practice**~~

~~4 If the constraint is violated with an integer constant expression of value zero, the diagnostic message should include the text of the string literal, if present.~~

Subclause 6.7.1, paragraph 1

Replace n3525 subclause 6.7.1, paragraph 1 with the following text.

Syntax

1 *declaration:*

declaration-specifiers init-declarator-listopt ;

attribute-specifier-sequence declaration-specifiers init-declarator-list ;

static_assert-declaration

attribute-declaration

declaration-specifiers:

declaration-specifier attribute-specifier-sequence$_{opt}$

declaration-specifier declaration-specifiers

declaration-specifier:

storage-class-specifier

type-specifier-qualifier

function-specifier

init-declarator-list:

init-declarator

init-declarator-list , init-declarator

init-declarator:

declarator

declarator = initializer

static_assert-declaration:

static-assertion ;

attribute-declaration:

attribute-specifier-sequence ;

simple-declaration:

attribute-specifier-sequence$_{opt}$ *declaration-specifiers declarator = initializer*

Subclause 6.7.1, paragraph 14

In n3525 subclause 6.7.1, insert a new paragraph 14 with the following text.

14 Aside from not having a type, `static_assert` declarations have the same semantic as the `static_assert` expressions. A block item of the form

static-assertion ;

shall be interpreted as a *static_assert-declaration*.

6 Future directions

A future proposal will allow the use of both the comma operator and the `void` type in integer constant expressions so that the result of the `BIT` function-like macro given as an example in §6.5.4.6 ¶5 in the proposed text would become an integer constant expression.

The generic selection construct, as illustrated in section 1.1 and 1.2, was not mentioned in the proposed text because it is seen as a workaround.

7 Acknowledgements

We would like to recognize the following people for their help reviewing this work: Robert C. Seacord, Aaron Ballman, Joseph Myers, Jens Gustedt, Alejandro Colomar and Martin Uecker.