Not-So-Magic - typeof(...) in C

JeanHeyd Meneide <phdofthehouse@gmail.com>

Shepherd (Shepherd's Oasis) <shepherd@soasis.org>

October 26th, 2020

— 1 Introduction & Motivation
— 2 Implementation & Existing Practice
— 3 Wording

Document: n2593
Previous Revisions: None
Audience: WG14
Proposal Category: New Features
Target Audience: General Developers, Compiler/Tooling Developers
Latest Revision: https://thephd.github.io/vendor/future_cxx/papers/source/n2593.html

Abstract:

Getting the type of an expression in Standard C code.

1 Introduction & Motivation

`typeof` is a extension featured in many implementations of the C standard to get the type of an expression. It works similarly to `sizeof`, which runs the expression in an “unevaluated context” to understand the final type, and thusly produce a size. `typeof` stops before producing a byte size and instead just yields a type name, usable in all the places a type currently is in the C grammar.

There are many uses for `typeof` that have come up over the intervening decades since its first introduction in a few compilers, most notably GCC. It can, for example, help produce a type-safe generic printing function that even has room for user extension (see: https://slbkbs.org/tmp/fmt/fmt.h). It can also help write code that can use the expansion of a macro expression as the return type for a function, or used within a macro itself to correctly cast to the desired result of a specific computation’s type (for width and precision purposes). The use cases are vast and endless, and many people have been locking themselves into implementation-specific vendorship that have locked them out of other compilers (for example, Microsoft’s Visual C Compiler).

2 Implementation & Existing Practice

Every implementation in existence since C89 has an implementation of `typeof`. Some compilers (GCC, Clang, EDG, tcc, and many, many more) expose this with the implementation extension `typeof`. But, the Standard already requires `typeof` to exist. Notably, with underlined emphasis added,

> The `sizeof` operator yields the size (in bytes) of its operand, which may be an expression or the parenthesized name of a type. **The size is determined from the type of the operand.** — N2573, §6.5.3.4 The `sizeof` and `_Alignof` operators, Semantics

Any implementation that can process `sizeof("foo")` is already doing `sizeof(typeof("foo"))` internally. This feature is the most “existing practice”-iest feature to be proposed to the C Standard, possibly in the entire history of the C standard.

Furthermore, putting a type or a VLA-type computation results in an idempotent type computation that simply yields that type in most implementations that support the feature.

3 Wording

The following wording is relative to N2573.

Adjust the Syntax grammar of §6.7.2 Type specifiers

> *type-specifier*:
>
>   **`void`**
>
>   …

typedef-name
typeof-specifier

Add a new §6.7.2.5 The Typeof specifier

§6.7.2.5 The Typeof specifier

Syntax

typeof-specifier:
  `_Typeof` *unary-expression*
  `_Typeof` **(** *type-name* **)**

Constraints

The *typeof-specifier* shall not be applied to an expression that has function type or an incomplete type, to the parenthesized name of such a type, or to an expression that designates a bit-field member.

Semantics

The *typeof-specifier* applies the **`_Typeof`** operator to a *unary-expression* (6.5.3) or a *type-specifier*. If the **`_Typeof`** operator is applied to a *unary-expression*, it yields the *type-name* representing the type of its operand[11�0)]. Otherwise, it produces the *type-name* with any nested *typeof-specifier* evaluated [11�1)]. If the type of the operand is a variable length array type, the operand are evaluated; otherwise, the operand is not evaluated.

Type qualifiers (6.7.3) of the type from the result of a **`_Typeof`** operation are preserved.

[11�0)] When applied to a parameter declared to have array or function type, the **`_Typeof`** operator yields the adjusted (pointer) type (see 6.9.1).

[11�1)] If the operand is a **`_Typeof`** operator, the operand will be evaluated before evaluating the current **`_Typeof`** operation. This happens recursively until a *typeof-specifier* is no longer the operand.

Add the following examples to the new Typeof section

[5] **EXAMPLE 1** Type of an expression

```
_Typeof(1) main () {
    return 0;
}
// equivalent to:
```

```
// int main() {
//     ...
// }
```

6 **EXAMPLE 2** Equivalence of `sizeof` and `typeof`.

```
int main (int argc, char* argv[]) {
    // this program has no constraint violations
    _Static_assert(sizeof(_Typeof('p')) ==
        sizeof(char));
    _Static_assert(sizeof(_Typeof('p')) ==
        sizeof('p'));
    _Static_assert(sizeof(_Typeof("meow")) ==
        sizeof(char[5]));
    _Static_assert(sizeof(_Typeof("meow")) ==
        sizeof("meow"));
    _Static_assert(sizeof(_Typeof(argc)) ==
        sizeof(int));
    _Static_assert(sizeof(_Typeof(argc)) ==
        sizeof(argc));
    _Static_assert(sizeof(_Typeof(argv)) ==
        sizeof(char**));
    _Static_assert(sizeof(_Typeof(argv)) ==
        sizeof(argv));
    return 0;
}
```

7 **EXAMPLE 3** Nested `_Typeof( ... )`.

```
int main (int argc, char*[]) {
    float val = 6.0f;
    // equivalent to a cast and return
    return (_Typeof(_Typeof(_Typeof(argc))))val;
    // return (int)val;
}
```

8 **EXAMPLE 4** Variable Length Arrays and `_Typeof`.

```
#include <stddef.h>

size_t vla_size (int n) {
    typedef char vla_type[n + 3];
    vla_type b; // variable length array
    return sizeof(
        _Typeof(b)
    ); // execution-time sizeof, translation-time
        _Typeof
}
```

```
int main () {
    return (int)vla_size(10); // vla_size returns 13
}
```

Add a new §7.� Typeof `<stdtypeof.h>`

The header `<stdtypeof.h>` defines two macros.

The macro

`typeof`

expands to `_Typeof`.

The macro

`__typeof_is_defined`

is suitable for use in `#if` preprocessing directives. It expands to the integer constant `1`.