Not-So-Magic - typeof(…) in C | r1

JeanHeyd Meneide <phdofthehouse@gmail.com>

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

December 1st, 2020

Document: n2619
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/C%20-%20typeof.html

Abstract:

Getting the type of an expression in Standard C code.

1 Changelog

1.1 Revision 1 - December 1st, 2020

— Completely Reformulate Paper based on community, GCC, and LLVM implementation feedback.
— Address major implementation contention of qualifiers with both `_Typeof` and `_Unqual_typeof`.
— Add section about not using C++'s `decltype` identifier for this and other compatibility issues.
— Completely rewrite the wording section.

1.2 Revision 0 - October 25th, 2020

— Initial release.

2 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 example implementation). 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. This keeps their code out of other compilers (for example, Microsoft's Visual C Compiler) and weakens the ISO C ecosystem overall.

3 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 emphasis (not found in the standard) 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, Programming Languages C - Working Draft, §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. The feature was also mentioned in an "extension round up" paper that went over the state of C Extensions in 2007[1].

3.1 Corner cases: Types and VLAs

Putting a normal or VLA-type computation results in an idempotent type computation that simply yields that type in most implementations that support the feature. If the compiler supports Variable Length Arrays, then `__typeof` – if it is similar to GCC, Clang, tcc, and others – then it is already supported with these semantics. These semantics also match how `sizeof` would behave (computing the expression or having an internal placeholder "VLA" type), so we propagate that same ability in an identical manner.

Notably, this is how current implementations evaluate the semantics as well.

3.2 Why not "decltype"?

C++ has a feature it calls `decltype(...)`, which serves most of the same purpose. "Most" is because it has a subtle difference which would wreak havoc on C code if it was employed in shared header code:

```
int value = 20;

#define GET_TARGET_VALUE (value)

inline decltype(GET_TARGET_VALUE) g () {
    return value;
}

int main () {
    int& r = g();
    return r;
}
```

The return type of `g` would be `int&` in C++, and `int` in C. Other expressions, such as array indexing and pointer dereferencing, also have this same issue. This is due to the parentheses in the expression. Macros in both languages frequently see extra parentheses employed around expressions to prevent mixing of precedence or other shenanigans from token-based macro expansion and subsequent language parsing; this would be a footgun of large proportions for C and C++ users, and create a divergence in standard use that would rise to the level of a liaison issue that may become unfixable. This is also part of the reason why `decltype` was given that keyword in C++, and not `typeof`: they did not want this kind of subtle and brutal change to afflict C and C++ code. `typeof` does not have this problem because – if a Sister Paper ever proposes it for C++ – it will have identical behavior to `std::remove_reference_t<decltype(T)>`.

This was also addressed when C++ was itself trying to introduce `dectlype` and competing with `typeof` in WG21 for C++[2].

3.3 C++ Compatibility

A similar feature should be proposed in C++, albeit it will likely take the keyword name `typeof` rather than `_Typeof`. This paper intends to have a similar paper brought before the C++ Committee – WG21 – through its Liaison Study Group, if this paper is successful.

3.4 Qualifiers

There is some discussion about what happens with qualifiers, both standard and implementation-defined. For example, "Named Address Space" qualifiers are subject to issues with GCC"s `typeof` extension, as shown here[3]. The intention of one of the GCC maintainers from that thread is:

> Well, I think we should fix typeof to not retain the address space. It's probably our implementation detail of having those in TYPE_QUALS that exposes the issue… — Richard Biener, GCC Maintainer, November 5th, 2020

There is also some disagreement between implementations about what qualifiers are worth keeping with respect to `_Atomic` between implementations. Therefore, this feature *strips all qualifiers from the computed type result*. The reason for this is that a user can add specifiers and qualifications to a type, but can not take them away once they are part of the expression. For example, consider the specification of `<complex.h>` that contains macro-provided constants like `_Imaginary_I`. These constants have the type `const float _Imaginary`: should all `typeof(_Imaginary_I)` expressions therefore result in a `const float _Imaginary`, or a `float _Imaginary`? What about `volatile`? And so on, and so forth.

There is an argument to strip all type qualifiers (`_Atomic`, `const`, `restrict`, and `volatile`) from the final type expression is because they can be added back by the programmer easily. However, the opposite is not true: you cannot add back qualifiers or create macros where those qualifiers can be taken in as parameters and re-applied to the function. This does leave some room to be desired: some folk may want to deliberately propagate the `const`-ness, `volatile`-ness, or `_Atomic`-ness of an expression to its end users.

3.4.1 Qualifiers - The Solution

Given how Qualifiers are, we propose that `_Typeof` be enhanced with a second version: `_Unqual_typeof`. That means there will be two new keywords introduced in this paper: `_Typeof` and `_Unqual_typeof`, where the first one keeps all the qualifiers of the original, and the second strips all qualifiers from the type. This importantly solves a (currently ongoing) debate for the above-linked GCC patch and conversation about how to strip qualifiers in the Linux Kernel properly. We note that this is better than the forced *lvalue conversion* route, which decays arrays and function types and thusly may not directly represent the original type.

/

3.4.2 In General

Separately, we should consider a Macro Programming facility for C that can address larger questions. This paper strives to focus on the material gains from existing practice and the pitfalls of said existing practice. Therefore, this paper proposes only `_Typeof` and `_Unqual_typeof`.

After this paper is handled, further research should be given to handling qualifiers, function types, and arrays in Macros for generic programming. Further research should be done in the area of conversions, which may aid in ABI issues from, e.g., certain function names creating ABI problems (c.f. the entire `intmax_t` discussion[4] currently in deadlock right now). `_ExtInt`[5] also brings up interesting consequences for `_Generic`, with respect to how to match various types of `_ExtInt` without having to write out a truly enormous list of explicit bit size associations, from 1 to some large `N`. Answering those questions may prove useful, but this paper does not explore any further than the existing practice.

4 Wording

The following wording is relative to N2573.

Modify §6.3.2.1 Lvalues, arrays, and function designators, paragraphs 3 and 4 with footnote 68:

3 Except when it is the operand of the ~~**sizeof** operator~~**sizeof**, or typeof operators, or the unary & operator, or is a string literal used to initialize an array, an expression that has type "array of *type*" is converted to an expression with type "pointer to *type*" that points to the initial element of the array object and is not an lvalue. If the array object has register storage class, the behavior is undefined.

4 A *function designator* is an expression that has function type. Except when it is the operand of the ~~**sizeof** operator~~**sizeof** operator, a typeof operator[68] or the unary & operator, a function designator with type "function returning *type*" is converted to an expression that has type "pointer to function returning *type*".

[68] Because this conversion does not occur, the operand of the ~~**sizeof** operator~~**sizeof** and typeof operators remains a function designator and violates the constraints in 6.5.3.4.

Add a keyword to the §6.4.1 Keywords:

```
_Thread_local
_Typeof
_Unqual_typeof
```

Modify §6.6 Constant expressions, paragraphs 6 and 8:

6 An integer constant expression[125] shall have integer type and shall only have operands that are integer constants, enumeration constants, character constants, **sizeof** expressions whose results are integer constants, **_Alignof** expressions, and floating constants that are the immediate operands of casts. Cast operators in an integer constant expression shall only convert arithmetic types to integer types, except as part of an operand to the ~~**sizeof**~~typeof operators, **sizeof** operator, or **_Alignof** operator.

...

8 An arithmetic constant expression shall have arithmetic type and shall only have operands that are integer constants, floating constants, enumeration constants, character constants,**sizeof** expressions whose results are integer constants, and **_Alignof** expressions. Cast operators in an arithmetic constant expression shall only convert arithmetic types to arithmetic types, except as part of an operand to the ~~**sizeof**~~typeof operators, **sizeof** operator, or **_Alignof** operator.

Adjust the Syntax grammar of §6.7.2 Type specifiers, the paragraph 2 list, and paragraph 4 Semantics:

> *type-specifier*:
> **`void`**
> ...
> *typedef-name*
> *typeof-specifier*

...

— enum specifier
— typedef name
— typeof specifier

4 Specifiers for ~~structures, unions, enumerations, and atomic types~~structures, unions, enumerations, atomic types, and typeof specifiers are discussed in 6.7.2.1 through ~~6.7.2.4~~6.7.2.5. Declarations of typedef names are discussed in 6.7.8. The characteristics of the other types are discussed in 6.2.5.

Adjust the footnote 133) in §6.7.2.1 Structure and union specifiers:

133) As specified in 6.7.2 above, if the actual type specifier used is `int` or a typedef-name defined as `int`, then it is implementation-defined whether the bit-field is signed or unsigned. This includes an **`int`** type specifier produced by the use of the typeof specifier (6.7.2.5).

Add a new §6.7.2.5 The Typeof specifiers:

§6.7.2.5 The Typeof specifiers

Syntax

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

Constraints

2 The *typeof-specifier* shall not be applied to an expression that designates a bit-field member.

Semantics

3 The *typeof-specifier* applies the **`_Typeof`** or **`_Unqual_typeof`** operator (collectively, the *typeof operators*) to a *unary-expression* (6.5) or a *type-specifier*. If the typeof operators are applied to an *expression*, they yield the *type-name* representing the type of their operand[11�0)]. Otherwise, they produce the *type-name* with any nested *typeof-specifier* evaluated [11�1)]. If the type of the operand is a variably modified type, the operand is evaluated; otherwise, the operand is not evaluated.

4 All qualifiers (6.7.3) on the type from the result of a **`_Unqual_typeof`** operation are removed, including **`_Atomic`**. Otherwise, for **`_Typeof`** operations, all qualifiers 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 new §6.7.2.5 The Typeof specifier:

5 **EXAMPLE 1** Type of an expression.
The following program:

```
_Typeof(1+1) main () {
    return 0;
}
```

is equivalent to this program:

```
int main() {
    return 0;
}
```

6 **EXAMPLE 2** Types and qualifiers.
The following program:

```
const _Atomic int purr = 0;
const int meow = 1;
const char* const mew[] = {
    "aardvark",
    "bluejay",
    "catte",
};

_Typeof(purr) main (int argc, char* argv[]) {
    _Unqual_typeof(purr)           plain_purr;
    _Typeof(_Atomic _Typeof(meow)) atomic_meow;
    _Typeof(mew)                   mew_array;
    _Unqual_typeof(mew)            mew2_array;
    return 0;
}
```

is equivalent to this program:

```
const _Atomic int purr = 0;
const int meow = 1;
const char* const mew[] = {
    "aardvark",
    "bluejay",
    "catte",
};

int main (int argc, char* argv[]) {
    int                 plain_purr;
    const _Atomic int atomic_meow;
    const char* const mew_array[3];
    const char*       mew2_array[3];
    return 0;
}
```

7 **EXAMPLE 3** Equivalence of sizeof and typeof.

```
int main (int argc, char* argv[]) {
    // this program has no constraint violations

    _Static_assert(sizeof(_Typeof('p')) == sizeof(int));
    _Static_assert(sizeof(_Typeof('p')) == sizeof('p'));
    _Static_assert(sizeof(_Typeof((char)'p')) == sizeof(char));
    _Static_assert(sizeof(_Typeof((char)'p')) == sizeof((char)'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(const char**));
    _Static_assert(sizeof(_Typeof(argv)) == sizeof(argv));

    _Static_assert(sizeof(_Unqual_typeof('p')) == sizeof(int));
    _Static_assert(sizeof(_Unqual_typeof('p')) == sizeof('p'));
    _Static_assert(sizeof(_Unqual_typeof((char)'p')) == sizeof(char));
    _Static_assert(sizeof(_Unqual_typeof((char)'p')) == sizeof((char)'p'));
    _Static_assert(sizeof(_Unqual_typeof("meow")) == sizeof(char[5]));
    _Static_assert(sizeof(_Unqual_typeof("meow")) == sizeof("meow"));
    _Static_assert(sizeof(_Unqual_typeof(argc)) == sizeof(int));
    _Static_assert(sizeof(_Unqual_typeof(argc)) == sizeof(argc));
    _Static_assert(sizeof(_Unqual_typeof(argv)) == sizeof(const char**));
    _Static_assert(sizeof(_Unqual_typeof(argv)) == sizeof(argv));
    return 0;
}
```

8 **EXAMPLE 4** Nested _Typeof(...).
The following program:

```
int main (int argc, char*[]) {
    float val = 6.0f;
    return (_Typeof(_Unqual_Typeof(_Typeof(argc))))val;
}
```

is equivalent to this program:

```
int main (int argc, char*[]) {
    float val = 6.0f;
    return (int)val;
}
```

9 **EXAMPLE 5** Variable Length Arrays and typeof operators.

```
#include <stddef.h>

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

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

10 **EXAMPLE 6** Nested typeof operators, arrays, and pointers.

```
int main () {
    _Typeof(_Typeof(const char*)[4]) y = {
        "a",
        "b",
        "c",
        "d"
    }; // 4-element array of "const pointer to char"
    return 0;
}
```

Modify §6.7.3 Type specifiers, paragraph 6:

6 If the same qualifier appears more than once in the same specifier-qualifier list or as declaration specifiers, either directly, via one or more typeof specifiers, or via one or more **typedef**s, the behavior is the same as if it appeared only once. If other qualifiers appear along with the **_Atomic** qualifier the resulting type is the so-qualified atomic type.

Modify §6.7.6.2 Array declarators, paragraph 5:

5 If the size is an expression that is not an integer constant expression: if it occurs in a declaration at function prototype scope, it is treated as if it were replaced by *; otherwise, each time it is evaluated it shall have a value greater than zero. The size of each instance of a variable length array type does not change during its lifetime. Where a size expression is part of the operand of a typeof or **sizeof** operator and changing the value of the size expression would not affect the result of the operator, it is unspecified whether or not the size expression is evaluated. Where a size expression is part of the operand of an **_Alignof** operator, that expression is not evaluated.

/

Modify §6.9 External definitions, paragraphs 3 and 5:

3 There shall be no more than one external definition for each identifier declared with internal linkage in a translation unit. Moreover, if an identifier declared with internal linkage is used in an expression ~~(other than as a part of the operand of a `sizeof` or `_Alignof` operator whose result is an integer constant),~~ there shall be exactly one external definition for the identifier in the translation unit~~.~~, unless it is:

— part of the operand of a **`sizeof`** operator whose result is an integer constant,
— part of the operand of a **`_Alignof`** operator whose result is an integer constant,
— or, part of the operand of any typeof operator whose result is not a variably modified type.

…

5 An *external definition* is an external declaration that is also a definition of a function (other than an inline definition) or an object. If an identifier declared with external linkage is used in an expression (other than as a part of the operand of a ~~`sizeof`,~~typeof operator whose result is not a variably modified type, or a **`sizeof`** or **`_Alignof`** operator whose result is an integer constant), somewhere in the entire program there shall be exactly one external definition for the identifier; otherwise, there shall be no more than one.$^{173)}$

Add a new §7.� Typeof `<stdtypeof.h>`:

1 The header `<stdtypeof.h>` defines four macros.

2 The macros

```
typeof
unqual_typeof
```

expands to **`_Typeof`** and **`_Unqual_typeof`**, respectively.

3 The macros

```
__typeof_is_defined
__unqual_typeof_is_defined
```

are suitable for use in **`#if`** preprocessing directives. They expand to the integer constant `1`.

---

1. Nick Stoughton. Potential Extensions For Inclusion In a Revision of ISO/IEC 9899. ISO/IEC SC22 WG14 - Programming Languages C. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1229.pdf.↩

2. Jaakko Järvi and Bjarne Stroustrup. Decltype and auto (revision 3). ISO/IEC JTC1 SC22 WG21 - Programming Languages C++. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1607.pdf.↩

3. Uros Bizjak. typeof and operands in named address spaces. GNU Compiler Collection. https://gcc.gnu.org/pipermail/gcc/2020-November/234119.html.↩

4. Martin Uecker. intmax_t, again. ISO/IEC JTC1 SC22 WG14 - Programming Languages C. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2498.pdf↩

5. Aaron Ballman, Melanie Blower, Tommy Hoffner, and Erich Keane. Adding a Fundamental Type for N-bit integers. ISO/IEC JTC1 SC22 WG14 - Programming Languages C. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2590.pdf↩