Proposal for C2y
WG14 N3260

Title:	Generic selection expression with a type operand
Author, affiliation:	Aaron Ballman, Intel
Date:	2024-05-12
Proposal category:	New Features
Target Audience:	Developers working in type-generic programming domains

Abstract: Extends the _Generic operator to accept a type operand which allows selecting an association with a qualified type instead of a type after lvalue conversion is applied to the operand.

Prior Art: Clang

Generic selection expression with a type operand

Reply-to: Aaron Ballman (aaron@aaronballman.com)
Document No: N3260
Revises Document No: N3214
Date: 2024-05-12

Summary of Changes

N3260

- Fixed wording to allow non-object and incomplete types in generic associations
- Fixed wording confusion between “type name” and “type”
- Fixed wording to ensure the controlling operand is not evaluated even if it is a type name

N3214

- Original proposal

Introduction and Rationale

Currently, generic selection expressions require the first operand to be an expression. This expression is not evaluated, but the type of the expression is compared to the types supplied by the association operands to determine which association matches (if any). However, the type used is the type **after lvalue conversion** which means it is not possible to match qualified types directly. Because lvalue conversion only drops top-level qualifiers, you might instead try to take the address of the expression and use pointer types as the associations. In other words, your code would start out looking something like this:

```
#define EXPR_HAS_TYPE(Expr, Type) _Generic(&(Expr), Type * : 1, default : 0)
const int i = 12;
_Static_assert(EXPR_HAS_TYPE(i, const int));
```

However, this won't work if the expression isn't an lvalue, so `EXPR_HAS_TYPE(12, int)` does not expand to valid code. It turns out to be surprisingly difficult to write a macro that will work in a type-generic way to provide "type traits" in C, and so users are left with partial or overly complex solutions.

Clang (and GCC, etc) have a builtin that comes close to solving this need, `__builtin_types_compatible_p`, however this also strips qualification from the given types. So it is close, but it has the same struggles as `_Generic`. Further, the builtin `__is_same` is only exposed in C++, and so it also doesn't solve the issue.

Proposed Solution

C has a few operators that take either a type or an expression, such as `sizeof`. It is natural to extend that idea to `_Generic` so that it can also accept a type for the first operand. This type does not undergo any conversions, which allows it to match qualified types, incomplete types, and function types. C23 has the `typeof` operator to get the type of an expression before lvalue conversion takes place, and so it keeps the qualification. This makes `typeof` a straightforward approach to determining a type operand for `_Generic` that considers qualifiers. Now our macro becomes:

```
#define EXPR_HAS_TYPE(Expr, Type) _Generic(typeof(Expr), \
                                           Type : 1, default : 0)
```

which can be called with an expression of any value category (no need to be an lvalue) and will test against (almost) any type. Many thanks to Thiago Adams for suggesting this approach!

This does mean the same operator has slightly different semantics when called with a type argument as opposed to an expression argument, which is not a behavior that `sizeof` has. An alternate keyword was considered by the author and the Clang community, but ultimately was not pursued because the semantics of the two forms are sufficiently distinguishable and a new keyword would be heavy-handed. Values can have conversion operations applied to them which modify the type, but a type by itself has no such chance for an implicit conversion, so it seems defensible that the semantics of a type inspection feature be tied to the operand form.

The proposed solution was implemented as an extension in Clang 17. Interested committee members can try out the feature for themselves on Compiler Explorer.

Other Differences Worth Noting

`_Generic` with a type operand will relax the requirements of what can be a valid association. Specifically, it allows incomplete types and non-object types (but still prevents use of variably-modified types). This relaxation only happens for the type operand form; the expression operand form continues to behave as it always has.

This extension allows incomplete and non-object types because the goal is to better enable type-generic programming in C, and so it should allow any typed construct where the type can be determined statically. There is no reason to prevent matching against `void` or function types, but this does explain why we continue to prohibit variably-modified types.

Further, allowing incomplete types enables "tag dispatch" functionality without requiring a complete type, which can be quite useful for generic programming. e.g.,

```
#define TAG_TO_INDEX(tag) _Generic(tag, \
  struct red_channel : 0, \
  struct green_channel : 1, \
  struct blue_channel : 2)

#define GET_TAGGED_VALUE(array, tag) array[TAG_TO_INDEX(tag)]

...
int colors[3];
int blue = GET_TAGGED_VALUE(colors, struct blue_channel);
```

Proposed Straw Poll

Does WG14 want to adopt the proposed wording from N3214?

Proposed Wording

All proposed wording in this document is a diff from WG14 N3149. Green text is new text, while ~~red~~ text is deleted text.

Modify 6.5.6.1p1:

generic-selection:

`_Generic` (~~*assignment-expression*~~*generic-controlling-operand* , *generic-assoc-list*)

generic-controlling-operand:

assignment-expression

type-name

Modify 6.5.6.1p2:

A generic selection shall have no more than one `default` generic association. The type name in a generic association shall specify a ~~complete object~~ type other than a variably modified type. No two generic associations in the same generic selection shall specify compatible types. If the generic controlling operand is an assignment expression, t~~T~~he *controlling type* of the ~~controlling~~generic selection expression is the type of the assignment expression as if it had undergone an lvalue conversion, array to pointer conversion, or function to pointer conversion. Otherwise, the controlling type of the generic selection expression is the type designated by the type name. ~~That~~ The controlling type shall be compatible with at most one of the types named in the generic association list. If a generic selection has no `default` generic association, its controlling ~~expression~~type shall have type compatible with exactly one of the types named in its generic association list.

Modify 6.5.6.1p3:

The generic controlling ~~expression~~operand ~~of a generic selection~~ is not evaluated. If a generic selection has a generic association with a type name that is compatible with the type of the controlling ~~expression~~type, then the result expression of the generic selection is the expression in that generic association. Otherwise, the result expression of the generic selection is the expression in the `default` generic association. None of the expressions from any other generic association of the generic selection is evaluated.

Add a new Example after 6.5.6.1p5:

EXAMPLE The following two generic selection expressions select different associations because the assignment expression operand undergoes lvalue conversion while the type name operand is unchanged:

```
void func(const int i) {
  _Generic(i,
    int : 0, // 'int' is selected
    const int : 1,
    default : 2);

  _Generic(typeof(i),
    int : 0,
    const int : 1, // 'const int' is selected
    default : 2);
}
```

Acknowledgement

I would like to recognize the following people for their help in this work: Thiago Adams and Joseph Myers.