Proposal for C2Y

WG14 N 3652

Title:	Composite types, v1.3
Author, affiliation:	Robert C. Seacord, Woven by Toyota, rcseacord@gmail.com Martin Uecker Graz University of Technology *uecker@tugraz.at*
Date:	2025-07-20
Proposal category:	Defect
Target audience:	Implementers, users
Abstract:	Clarify composite types and remove UB
Prior art:	C23

Composite types, v1.3

Reply-to: Robert C. Seacord (rcseacord@gmail.com)

Document No: **N 3638**

Reference Document: **N 3301**

Date: 2024-07-03

This proposal changes the rule for forming composite types

Change Log

2024-11-09:

- Initial version

2025-03-13:

- Eliminated proposed wording option retaining UB
- Eliminated the term “pairing”
- Reworked examples.
- Changed text before examples to make it clear that we are only referring to the example and not making a broader assertion.
- Updated to be consistent with n3508

2025-07-03:

- Removed the italics for "specified" in one place in the new text doesn't match the usual conventions in the standard (not indicating a definition, nor the name of a syntax production within the Syntax).
- Removed the word "function" from The example of composite type of double[] and double[]

2025-07-20:

- Removed a footnote which is more confusing than helpful.

Table of Contents

1 Problem Description

1.1 Array Type Variants

There are a variety of ways to express an array length which creates different array type variants.

Known Constant Length

```
int[1] // known constant length
```

Regular arrays with known constant length. The length expression is an integer constant expression.

Known Variable Length

```
int[n] // known variable length, n can be evaluated or unevaluated
```

A variable length array where the size is given by an expression evaluated at run-time. There are situations where the length expressions are not evaluated but the type is needed, which can currently lead to run-time undefined behavior in conditional expressions.

Unspecified Length

```
int[*] // array of known variable length, but the length is unspecified
```

Arrays of known variable length, but the length is unspecified. Those arrays are currently only used in contexts where the actual type later used then has a length which is specified. As such the `*` is a placeholder for a length expression in arrays that need to observe the same rules as regular arrays but where the length is never actually needed, because they occur in unevaluated code.

Unknown Length

```
int[ ] // array of unknown length
```

Arrays of unknown length can only be used if their length is not needed. Such arrays have an incomplete type, which can not be used in any situation where the length is needed in principle.

1.2 Relevant Standard Clauses

C2Y working draft n3301, Subclause 6.7.7.3, paragraph 4 states that:

> If the size is an integer constant expression and the element type has a known constant size, the array type is not a variable length array type; otherwise, the array type is a variable length array type.

This means that an array whose length is an integer constant expression but whose elements do not have a constant size are considered variable length arrays.

N3301 subclause 6.2.5, paragraph 28 states that:

> A type has known constant size if it is complete and is not a variable length array type.

N3301 subclause 6.2.7, paragraph 3 has a set of rules that are applied when one type is “a variable length array” or “an array of known constant size”.

According to existing practice, the logic for determining composite types should be based on whether the array has a known constant number of elements and NOT if the array is a variable length array.

2 Proposal

This proposal changes the semantics of the language by changing the composite type rules to allow a variable length array whose size is specified by an expression that is not evaluated to have a composite type of a VLA of specified length. Furthermore, this paper suggests removing undefined behavior when forming composite types based on N3397 and includes revised wording for this.

Each rule is based on the array length expression and if it has known constant length, is an unevaluated expression, a VLA of specified length, a VLA of unspecified length, or has unknown length.

N3397 Slaying A Triple-Headed Demon seeks to eliminate undefined behavior related to variably modified types. This overlaps significantly with how composite types are handled, so this proposal provides wording that incorporates that proposal.

The following table shows the proposed new behavior for the composite type of two array expressions when the UB associated with an unevaluated size expression is treated as a constraint violation as described in N3397:

	Known	VLA of	VLA of	Unevaluated	Unknown

	constant length	specified length	unspecified length	expression (now also unspecified)	length
Known constant length	Known constant length	Known constant length	Known constant length	Known Constant length	Known constant length
VLA of specified length		VLA of specified length	VLA of specified length	VLA of specified length†	VLA of specified length
VLA of unspecified length			VLA of unspecified length	VLA of unspecified length*	VLA of unspecified length
Unevaluated expression (now also unspecified)				VLA of unspecified length*	VLA of unspecified length*
Unknown length					Unknown length

The cases marked with * and † are undefined behavior in C23 without the additional change to the conditional operator. With the additional change from N3397, the unevaluated expressions are now considered to be unspecified lengths too and treated in the exact same way as other unspecified lengths. The cases marked with * then formally become arrays of unspecified length. Additional constraints to the conditional operator prevent their appearance outside of function prototypes, so that these cases cannot appear anywhere they might cause undefined behavior (EXAMPLE 12 to 6.5.16). The cases marked with † become defined because the known constant or specified length takes precedence over the unspecified (formerly unevaluated) length (EXAMPLE 13 to 6.5.16).

3 Proposed Text

Proposed wording changes are against C2Y working draft n3467.

Subclause 6.2.7, paragraph 3

Replace N3301 subclause 6.2.7, paragraph 3 with the following text. The text in green contains changes while the text in **black** does not.

A composite type can be constructed from two types that are compatible. If both types are the same type, the composite type is this type. Otherwise, it is a type that is compatible with both and satisfies

the following conditions:

— If both types are structure types or both types are union types, the composite type is determined recursively by forming the composite types of their members.

— If both types are array types, the following rules are applied:

• If one type is an array of known constant length, the composite type is an array of that length.

EXAMPLE Given the following two types:

```
double[3]
double[]
```

The resulting composite type is:

```
double[3]
```

• Otherwise, if one type is a variable length array whose length is specified, the composite type is a variable length array of that length.

EXAMPLE Given the following two types:

```
double[]
double[size]
```

The resulting composite type is:

```
double[size]
```

• Otherwise, if one type is a variable length array of unspecified length, the composite type is a variable length array of unspecified length.

EXAMPLE Given the following two types:

```
double[]
double[*]
```

The resulting composite type is:

```
double[*]
```

• Otherwise, both types are arrays of unknown length, and the composite type is an array of unknown length.

EXAMPLE Given the following two types:

```
double[]
```

```
double[]
```

The resulting composite type is:

```
double[]
```

The element type of the composite array type is the composite type of the two element types.

— If both types are function types, the type of each parameter in the composite parameter type list is the composite type of the corresponding parameters.

— If one of the types has a standard attribute, the composite type also has that attribute.

— If both types are enumerated types, the composite type is an enumerated type.

— If one type is an enumerated type and the other is an integer type other than an enumerated type, it is implementation-defined whether or not the composite type is an enumerated type.

These rules apply recursively to the types from which the two types are derived.

6.5.16 Conditional operator

Constraints

5 If one operand is a pointer to a variably modified type and the other operand is a null pointer constant or has type `nullptr_t`, the variably modified type shall not depend on array length expressions that would remain unevaluated when the corresponding operand is not evaluated. When recursively forming a composite type to determine the result type, no composite type shall be formed from an array of unknown length that is not part of a declaration for a function parameter and an array whose length expression remains unevaluated when the corresponding operand is not evaluated.

Semantics

~~8 If one operand is a pointer to a variably modified type and the other operand is a null pointer constant or has type nullptr_t, the behavior is undefined if the type depends on an array size expression that is not evaluated~~

8 All array length expressions that are not an integer constant expression and which are part of the type of the operand that is not evaluated are treated as unspecified lengths in the determination of type compatibility and when forming the composite type according to 6.2.7.

12 EXAMPLE Both conditional expressions in the `foo` function are constraint violations.

```
void foo(bool cond, void* p1, void* p2)
{
  int n = 2;
  // An array whose length might not be evaluated is
```

```
  // a constraint violation
  auto a = cond ? nullptr : (char(*)[n])p1;
  // Forming the composite type of an array of unknown length and an
array
  // whose length might not be evaluated is a constraint violation
  auto b = cond ? (char(*)[ ])p1 : (char(*)[n])p2;
}
```

13 EXAMPLE All conditional expressions in the `foo` function have defined behavior.

```
void foo(bool cond, void *p1, void *p2)
{
  int n = 2;
  int m = 3;
  char (*a)[3] = cond ? (char(*)[3])p1 : (char(*)[m])p2; // known
constant length
  char (*b)[cond ? n : m]
     = cond ? (char(*)[n])p1 : (char(*)[m])p2; // active branch length
  char (*p3)[ ] = p1;
  char (*p4)[m] = p2;
  char (*c)[m] = cond ? nullptr : p4; // previously evaluated
  char (*d)[m] = cond ? p3 : p4; // previously evaluated
}
```

6.7.7.3 Array declarators

6 For two array types to be compatible, both shall have compatible element types, and if both array length expressions are present, and are integer constant expressions, then both array length expressions shall have the same constant value. If the two array types are used in a context which requires them to be compatible, it is undefined behavior if ~~the two size specifiers~~ the lengths of both are specified and the corresponding array length expressions evaluate to unequal values.

4 Interaction with other proposals

n3416 Objects of known constant size changes subclause 6.7.7.3 describing how array declarators are interpreted.

The paper integrates changes from n3397.

5 Acknowledgements

We would like to recognize the following people for their help with this work: Aaron Ballman, Joseph S. Myers, Jens Gustedt, Anthony Williams, Tyler Kowalis, and Caleb McGary.