Slaying A Triple-Headed Demon

Martin Uecker, Graz University of Technology

Number: N3397
Date: 2024-11-23

The conditional (tertiary) operator can introduce run-time UB related to variably modified types in three different ways.

Variants of Array Types

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

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

```
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 size expressions are not evaluated but the type is needed, which can currently lead to run-time undefined behavior in conditional expressions.

```
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 star is a placeholder for a size 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.

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

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

Sources of Undefined Behavior

We have a rule that clarifies that size expressions for the length have to match for two arrays types to be compatible, where constant sizes have to match at compile time while run-time expressions cause run-time undefined behavior if they do not match at run-time:

6.7.7.3p6: “For two array types to be compatible, both shall have compatible element types, and if both size specifiers are present, and are integer constant expressions, then both size specifiers 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 evaluate to unequal values.”

For two compatible types, we can form a composite type, which then also introduces undefined behavior when one size expression is not evaluated:

6.2.7: “— If both types are array types, the following rules are applied:
• If one type is an array of known constant size, the composite type is an array of that size.
• Otherwise, if one type is a variable length array whose size is specified by an expression that is not evaluated, the behavior is undefined.

• Otherwise, if one type is a variable length array whose size is specified, the composite type is a variable length array of that size.
• Otherwise, if one type is a variable length array of unspecified size, the composite type is a variable length array of unspecified size.
• Otherwise, both types are arrays of unknown size and the composite type is an array of unknown size.”

Finally, since C23 we have an explicit rule for th e conditional operator introducing undefined behavior:

6.5.16p8: “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”

Example

Now consider the following example using the conditional operator and assume `cond` is true. The example shows the different situations where we can end up having undefined behavior.

```
void foo(bool cond, void* ptr1, void* ptr2)
{
	int n = 2;
	int m = 3;
	cond ? 0 : (char(*)[n])ptr1;
	// UB according to 6.5.16p8

	cond ? (char(*)[ ])ptr1 : (char(*)[n])ptr2;
	// UB according to 6.2.7 (cond == true)

	cond ? (char(*)[3])ptr1 : (char(*)[n])ptr2;
	// UB according to 6.2.7 (cond == true)

	cond ? (char(*)[n])ptr1 : (char(*)[m])ptr2;
	// UB according to 6.2.7

	char (*ptr3)[n] = ptr1;
	char (*ptr4)[m] = ptr2;

	cond ? ptr3 : ptr4;
	// UB according to 6.7.7.3p6
}
```

https://godbolt.org/z/brTbE6hxe

A complication is that it can depend on the runtime condition whether there is UB or not.

Proposed Change

It is observed that the UB in the rules for the composite type only arises due unevaluated expressions in the conditional operator. If we add constraints to the conditional operator to avoid all problematic cases, we can simply eliminate this case. To achieve this, we first declare all unevaluated size expressions that appear in the unevaluated branch to be unspecified sizes and remove the then vacuous case in 6.2.7 that causes UB in this situation. This has one problem though: For the conditional operator when the size is paired with a missing size, the composite type may have either a specified or an unspecified size depending on whether the condition is true or not at run-time. We therefor make all such dubious cases constraint violations.

Possible Future Change: The composite types seems to be a questionable choice for the conditional operator as it constructs a most specific type, instead of generalizing two given types. For example, if a function pointer with *unsequenced* attribute would be given in one branch but not the other, the result type would have this attribute even though it is not guaranteed that both arguments actually fulfill its requirements. A possible solution is to define a new "conditional type" which would be similar to the composite type, but adapted to the requirements of the conditional operator.

Wording

6.2.7 Compatible type and composite type

A composite type can be constructed from two types that are compatible; it is a type that is compatible with both of the two types and satisfies the following conditions:
— If both types are array types, the following rules are applied:
~~**• If one type is an array of known constant size, the composite type is an array of that size.**~~
~~**• Otherwise, if one type is a variable length array whose size is specified by an expression that is not evaluated, the behavior is undefined.**~~
• ~~**Otherwise,**~~ if one type is ~~**a variable length**~~ **an** array whose size is specified, the composite type is ~~**a variable length**~~ **an** array of that size.
• Otherwise, if one type is a variable length array of unspecified size/length, the composite type is a variable length array of unspecified size.
• Otherwise, both types are arrays of unknown size and the composite type is an array of unknown size.
The element type of the composite 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

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 in the determination of the result type, arrays of unknown length that are not part of a function prototype shall not be paired with arrays whose length expressions remain 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 sizes expressions of arrays which do not have a known constant length and which are part of the type of the operand that is not evaluated are treated as unspecified sizes in the determination of type compatibility^{YYY} and when forming the composite type according to 6.2.7.

YYY) Hence, there is no undefined behavior even when sizes of variably modified types in the two operands do not agree at runtime.

12 EXAMPLES A constraint is violated for both conditional expressions.

```
void foo(bool cond, void* ptr1, void* ptr2)
{
	int n = 2;
	int m = 3;
	auto a = cond ? (void*)0 : (char(*)[n])ptr1;
	auto b = cond ? (char(*)[ ])ptr1 : (char(*)[n])ptr2;
}
```

13 EXAMPLES The conditional expressions have defined behavior.

```
void foo(bool cond, void* ptr1, void* ptr2)
{
	int n = 2;
	int m = 3;

	auto a = cond ? (char(*)[2])ptr1 : (char(*)[m])ptr2;
	auto b = cond ? (char(*)[n])ptr1 : (char(*)[m])ptr2;

	char (*ptr3)[ ] = ptr1;
	char (*ptr4)[m] = ptr2;

	auto c = cond ? 0 : ptr4;       // Ok, previously evaluated
	auto d = cond ? ptr3 : ptr4;    // Ok, previously evaluated
}
```

6.7.6.2

6 For two array types to be compatible, both shall have compatible element types, and if both size specifiers are present, and are integer constant expressions, then both size specifiers 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 size expressions** evaluate to unequal values.