Type Compatibility: Ghosts, Readability, and A Missing Rule for Atomic

Author: Martin Uecker
Document: n3713
Date: 2025-09-21

For defining type compatibility, several paragraphs use “shall” in a semantics section, i.e. outside of a constraints section, without expressing a requirement and without implying undefined behavior (cf 4 Conformance). For qualifiers, array types, and function types, a change was already proposed in N3484 and adopted into C2y in Graz, but a similar change is missing for pointer types. This is fixed by Change 1a and Change 1b below. Change 1b is alternative wording that – for consistency with similar rules - drops rules about qualifiers that are already specified elsewhere and are therefor redundant at this point.

When investigating this issue, it was also noted that we miss a corresponding rule for atomic types, which seems to be an omission. I am not aware of any compiler that would not treat atomic versions of compatible types as compatible. Such a rule is added by Change 2.

Finally, it is proposed to move all wording related to type compatibility including Change 1b, Change 2, and the already adopted changes from N3484 (with minor wording tweaks) into section 6.2.7 (Change 3).

Proposed Wording (N3550)

Wording Change 1a (pointers)

6.7.7.2 Pointer declarators

~~For~~ **T**wo pointer types ~~**to be**~~ **are** compatible, **if and only if** both ~~**shall be**~~ **are** identically qualified and both ~~**shall be**~~ **are** pointers to compatible types.

Wording Change 1b (alternative to 1a)

6.7.7.2 Pointer declarators

~~For~~ **T**wo pointer types ~~**to be**~~ **are** compatible, **if and only if** ~~**both shall be identically qualified and**~~ both ~~**shall be**~~ **are** pointers to compatible types.

Wording Change 2 (missing rule for atomic)

6.7.3.5 Atomic type specifiers

Two atomic types are compatible, if and only if the corresponding non-atomic types are compatible.

Wording Change 3 (harmonized and consolidated version including the changes from N3484)

6.2.7 Compatible type and composite type

1 Two types are compatible types if they are the same. ~~**Additional rules for determining whether two types are compatible are described in 6.7.3 for type specifiers, in 6.7.4 for type qualifiers, and in 6.7.7 for declarators.**~~45)

2 Two qualified types are compatible if and only if they are identically qualified versions of compatible types. For unqualified types the following holds.

3 Two pointer types are compatible, if and only if both are pointers to compatible types.

4 Two atomic types are compatible, if and only if the corresponding non-atomic types are compatible.

5 Two function types are compatible if and only if all of the following hold:
-- they specify compatible return types;
-- the parameter type lists agree in the number of parameters and whether the function is variadic or not;
-- and the corresponding parameters have compatible types.

In the determination of type compatibility and of a composite type, each parameter declared with function or array type is taken as having the adjusted type (as in 6.7.7.4) and each parameter declared with qualified type is taken as having the unqualified version of its declared type.

6 Two array types are compatible, if and only if the following holds.
-- they have compatible element types;
-- and, if both size specifiers are present and are integer constant expressions, they have the same constant value.

If the two array types are used in a context which requires them to be compatible, the behavior is undefined if the two size specifiers evaluate to unequal values.

7 ~~**Moreover,**~~ **T**wo complete structure, union, or enumerated types declared with the same tag are compatible **if and only if the** members satisfy the following requirements.

— there ~~**shall be**~~ **is** a one-to-one correspondence between their members such that each pair of corresponding members are declared with compatible types;
— if one member of the pair is declared with an alignment specifier, the other is declared with an equivalent alignment specifier;
— and, if one member of the pair is declared with a name, the other is declared with the same name.

For two structures, **there is the additional requirement that** corresponding members ~~**shall be**~~ **are** declared in the same order. For two union declared in the same translation unit, **there is the additional requirement that** corresponding members ~~**shall be**~~ **are** declared in the same order. For two structures or unions, **there is the additional requirement that** corresponding bit-fields ~~**shall**~~ have the same widths. For two enumerations, **there is the additional requirement that** corresponding members ~~**shall**~~ have the same values; if one has a fixed underlying type, then the

other ~~shall~~ **is required to** have a compatible fixed underlying type. **The enumerated type is compatible with the underlying type of the enumeration.**

For determining type compatibility, anonymous structures and unions are considered a regular member of the containing structure or union type, and the type of an anonymous structure or union is considered compatible with the type of another anonymous structure or union, respectively, if their members fulfill the preceding requirements.

6.7.3 Type specifiers
6.7.3.3 Enumeration specifiers

16 The enumeration member type for an enumerated type with fixed underlying type is the enumerated type. ~~**The enumerated type is compatible with the underlying type of the enumeration.**~~ After possible lvalue conversion a value of the enumerated type behaves the same as the value with the underlying type, in particular with all aspects of promotion, conversion, and arithmetic. Conversion to the enumerated type has the same semantics as conversion to the underlying type.145)

6.7.4 Type qualifiers
6.7.4.1 General

~~**11 For two qualified types to be compatible, both shall have the identically qualified version of a compatible type; the order of type qualifiers within a list of specifiers or qualifiers does not affect the specified type.**~~

6.7.7.2 Pointer declarators

~~**2 For two pointer types to be compatible, both shall be identically qualified and both shall be pointers to compatible types.**~~

6.7.7.3 Array declarators

~~**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, the behavior is undefined if the two size specifiers evaluate to unequal values.**~~

6.7.7.4 Function declarators

~~**15 For two function types to be compatible, both shall specify compatible return types. Moreover, the parameter type lists shall agree in the number of parameters and in whether the function is variadic or not; corresponding parameters shall have compatible types. In the determination of type compatibility and of a composite type, each parameter declared with function or array type is taken as having the adjusted type and each parameter declared with qualified type is taken as having the unqualified version of its declared type**~~