Document number: N3444
Submitter: László Érsek <laszlo.ersek@posteo.net>
Subject: integer promotions and conversion ranks, and alignments, in C17

(1) EXAMPLE 2 in 5.1.2.3p11 states,

> the "integer promotions" require that the abstract machine promote the value of each variable to `int` size and then add the two `int`s

Especially the phrase "two `int`s" seems restrictive. For example, on an implementation where the representations of both `signed char` and `int` matched that ot `int16_t`, and where the representations of all of `char`, `unsigned char`, and `unsigned int` matched that of `uint16_t`, the `c1` and `c2` variables' values would be promoted to `unsigned int`.

Suggested change: replace the above-cited language with

> the "integer promotions" require that the abstract machine promote the value of each variable to (`signed` or `unsigned`) `int` and then add the two `(signed or unsigned`) `int`s

(2) Assume an implementation where the following exact-width integer types (7.20.1.1) are provided:

```
typedef unsigned long uint64_t;
typedef   signed long int64_t;
typedef unsigned int  uint32_t;
typedef   signed int  int32_t;
```

and where the implementation permits `uint64_t` for bit-fields (6.7.2.1p5).

In the following code fragment:

```
struct { uint64_t bf:32; } s = { 0 };
(void)printf("%u\n", s.bf);
```

can we deduce – and if so, *how exactly* can we deduce –, from the rules in 6.3.1.1, whether the expression `s.bf` is, or is not, promoted to `unsigned int`?

Namely, the second subclause of 6.3.1.1p2 does not apply (`s.bf` does not have type `_Bool`, `int`, `signed int`, or `unsigned int`). In turn, the first subclause of 6.3.1.1p2 depends on whether `s.bf`'s integer conversion rank is less than or equal to that of `int` and `unsigned int`.

For determining that, I fail to construct a rigorous chain of arguments from the rules in 6.3.1.1p1:

1. Per rule#4, the rank of **`uint64_t:32`** equals that of **`int64_t:32`**.
2. The precision of **`int64_t:32`** is 31 (value) bits (6.2.6.2p6).
3. Therefore, per rule#2, the rank of **`int64_t:32`** is strictly less than that of **`signed long`** (whose precision is 63 bits).
4. However, rule#2 does not say anything about ranks when the precisions of two signed integer types are identical (such as those of **`int64_t:32`** and **`int`** – both have 31 value bits).
5. Rule#1 appears to prohibit **`int64_t:32`** and **`int`** from having identical ranks, in spite of their identical representations; therefore, one of those types must have a strictly lesser rank than the other. The actual ranking order seems undecidable from the rules, and with that, it seems undecidable whether `s.bf` is promoted to **`unsigned int`**.

(3) Rules #1 and #4 from 6.3.1.1p1:

> – No two signed integer types shall have the same rank, even if they have the same representation.
>
> – The rank of any unsigned integer type shall equal the rank of the corresponding signed integer type, if any.

appear to render the part of 6.3.1.1p2 that is underlined below redundant:

> An object or expression with an integer type (other than **`int`** or **`unsigned int`**) whose integer conversion rank is less than <u>or equal to</u> the rank of **`int`** and **`unsigned int`**.

An integer type that differs from both **`int`** and **`unsigned int`** cannot have the same rank as **`int`** or **`unsigned int`** per rules #1 and #4 in 6.3.1.1p1, therefore “equal to” is constant false, and “P or false” is just P.

The words “or equal to” seem to originate from the resolution of Defect Report #230, and therefore they seem justified.

The conflict / redundancy should be eliminated by reworking rule #1 of 6.3.1.1p1. Namely, 6.2.5p17 specifies,

> […] and the enumerated types are collectively called *integer types*. [...]

and 6.3.1.1p1 rule #8 states,

> The rank of any enumerated type shall equal the rank of the compatible integer type (see 6.7.2.2).

Therefore 6.3.1.1p1 rule#1 states falsehood.

Rule#1 further conflicts with rule#6:

> The rank of `char` shall equal the rank of `signed char` and `unsigned char`.

Suggested change: eliminate 6.3.1.1p1 rule#1 altogether.

(4) The concept of "alignment of pointers" is used in two senses over the standard, and therefore 6.2.5p28 is ambiguous.

- Meaning #1: consider 6.2.5p26:

  > Any type so far mentioned is an unqualified type. […] The qualified or unqualified versions of a type are distinct types that belong to the same type category and have the same representation and alignment requirements.48) […]

  In this context, "alignment requirements" clearly refers to the "requirement that *[pointer objects themselves]* be located on storage boundaries with addresses that are particular multiples of a byte address" (see 3.2). See also (informative) footnote 48:

  > The same representation and alignment requirements are meant to imply interchangeability as arguments to functions, return values from functions, and members of unions.

- Meaning #2: consider 6.3.2.3p5:

  > An integer may be converted to any pointer type. Except as previously specified, the result is implementation-defined, might not be correctly aligned, might not point to an entity of the referenced type, and might be a trap representation.

  In this context, "correctly aligned" clearly refers to the "requirement that *[objects referenced by a particular pointer type]* be located on storage boundaries with addresses that are particular multiples of a byte address" (see 3.2 and 6.2.5p20).

Both meanings distinctly differ, but each example context above makes clear which sense is intended. Contrast that with 6.2.5p28:

> A pointer to `void` shall have the same representation and alignment requirements as a pointer to a character type.48) Similarly, pointers to qualified or unqualified versions of compatible types shall have the same representation and alignment requirements. All pointers to structure types shall have the same representation and alignment requirements as each other. All pointers to union types shall have the same representation and alignment requirements as each other. Pointers to other types need not have the same representation or alignment requirements.

(Same footnote 48 referenced as above.)

In this context, it is ambiguous whether the "same alignment requirements" between the noted pointer types refer to the pointer objects *themselves*, or to the objects *referenced* by the pointer objects – or even both. The request is for the Committee to please clarify this.

(5) (Conceptual) division of an object's address by the required alignment should always yield a zero remainder, and this is not immediately clear from 6.2.8p1:

> An alignment is an implementation-defined integer value representing the number of bytes between successive addresses at which a given object can be allocated.

This language does not preclude (numeric) addresses of the form `k*2`^{n}`+r`, where `k` is a nonnegative integer, `2`^{n} is the alignment (with `n` being a positive integer), and `r` is a *positive* integer less than `2`^{n}.

In comparison, 3.2 is clear that `r` must be zero:

> requirement that objects of a particular type be located on storage boundaries with addresses that are particular multiples of a byte address

Suggested change: replace the cited sentence in 6.2.8p1 with

> An alignment is an implementation-defined integer value representing <u>the byte address at the whole multiples of which</u> a given object can be allocated.

(6) **`_Alignof(char)`** should be explicitly defined as **`(size_t)1`**.

This already follows from e.g. 6.3.2.3p7:

> […] When a pointer to an object is converted to a pointer to a character type, the result points to the lowest addressed byte of the object. Successive increments of the result, up to the size of the object, yield pointers to the remaining bytes of the object.

6.2.8p6 is not explicit enough per se; it only says

> […] The types **`char`**, **`signed char`**, and **`unsigned char`** shall have the weakest alignment requirement.

6.5.3.4p4 states:

> When **`sizeof`** is applied to an operand that has type **`char`**, **`unsigned char`**, or **`signed char`**, (or a qualified version thereof) the result is 1. […]

A similar statement should be added about the **`_Alignof`** operator.

Suggested change: insert, as a new paragraph, between 6.5.3.4p4 and 6.5.3.4p5:

> When **`_Alignof`** is applied to an operand that has type **`char`**, **`unsigned char`**, or **`signed char`**, (or a qualified version thereof) the result is 1.