Chapter 47: Implementation-defined behaviour

Section 47.1: Right shift of a negative integer

```
int signed_integer = -1;

// The right shift operation exhibits implementation-defined behavior:
int result = signed_integer >> 1;
```

Section 47.2: Assigning an out-of-range value to an integer

```
// Supposing SCHAR_MAX, the maximum value that can be represented by a signed char, is
// 127, the behavior of this assignment is implementation-defined:
signed char integer;
integer = 128;
```

Section 47.3: Allocating zero bytes

```
// The allocation functions have implementation-defined behavior when the requested size
// of the allocation is zero.
void *p = malloc(0);
```

Section 47.4: Representation of signed integers

Each signed integer type may be represented in any one of three formats; it is implementation-defined which one is used. The implementation in use for any given signed integer type at least as wide as `int` can be determined at runtime from the two lowest-order bits of the representation of value `-1` in that type, like so:

```
enum { sign_magnitude = 1, ones_compl = 2, twos_compl = 3, };
#define SIGN_REP(T) ((T)-1 & (T)3)

switch (SIGN_REP(long)) {
   case sign_magnitude: { /* do something */ break; }
   case ones_compl:     { /* do otherwise */ break; }
   case twos_compl:     { /* do yet else  */ break; }
   case 0:  { _Static_assert(SIGN_REP(long), "bogus sign representation"); }
}
```

The same pattern applies to the representation of narrower types, but they cannot be tested by this technique because the operands of `&` are subject to "the usual arithmetic conversions" before the result is computed.