Chapter 23: Sequence points

Section 23.1: Unsequenced expressions

Version ≥ C11

The following expressions are *unsequenced*:

```
a + b;
a - b;
a * b;
a / b;
a % b;
a & b;
a | b;
```

In the above examples, the expression `a` may be evaluated before or after the expression `b`, `b` may be evaluated before `a`, or they may even be intermixed if they correspond to several instructions.

A similar rule holds for function calls:

```
f(a, b);
```

Here not only `a` and `b` are unsequenced (i.e. the `,` operator in a function call *does not* produce a sequence point) but also `f`, the expression that determines the function that is to be called.

Side effects may be applied immediately after evaluation or deferred until a later point.

Expressions like

```
x++ & x++;
f(x++, x++); /* the ',' in a function call is *not* the same as the comma operator */
x++ * x++;
a[i] = i++;
```

or

```
x++ & x;
f(x++, x);
x++ * x;
a[i++] = i;
```

will yield *undefined behavior* because

- a modification of an object and any other access to it must be sequenced
- the order of evaluation and the order in which *side effects*1 are applied is not specified.

1 Any changes in the state of the execution environment.

Section 23.2: Sequenced expressions

The following expressions are *sequenced*:

```
a && b
a || b
```

```
a , b
a ? b : c
for ( a ; b ; c ) { ... }
```

In all cases, the expression `a` is fully evaluated and *all side effects are applied* before either `b` or `c` are evaluated. In the fourth case, only one of `b` or `c` will be evaluated. In the last case, `b` is fully evaluated and all side effects are applied before `c` is evaluated.

In all cases, the evaluation of expression `a` is *sequenced before* the evaluations of `b` or `c` (alternately, the evaluations of `b` and `c` are *sequenced after* the evaluation of `a`).

Thus, expressions like

```
x++ && x++
x++ ? x++ : y++
(x = f()) && x != 0
for ( x = 0; x < 10; x++ ) { ... }
y = (x++, x++);
```

have well defined behavior.

Section 23.3: Indeterminately sequenced expressions

Function calls as `f(a)` always imply a sequence point between the evaluation of the arguments and the designator (here `f` and `a`) and the actual call. If two such calls are unsequenced, the two function calls are indeterminately sequenced, that is, one is executed before the other, and order is unspecified.

```
unsigned counter = 0;

unsingned account(void) {
   return counter++;
}

int main(void) {
   printf("the order is %u %u\n", account(), account());
}
```

This implicit twofold modification of `counter` during the evaluation of the `printf` arguments is valid, we just don't know which of the calls comes first. As the order is unspecified, it may vary and cannot be depended on. So the printout could be:

> the order is 0 1

or

> the order is 1 0

The analogous statement to the above without intermediate function call

```
   printf("the order is %u %u\n", counter++, counter++); // undefined behavior
```

has undefined behavior because there is no sequence point between the two modifications of `counter`.