Doc. No.: WG14/N1188
Date: 2006-09-04
Project: Programming Language C
Reply to: Thomas Plum <tplum@plumhall.com> +1-808-882-1255

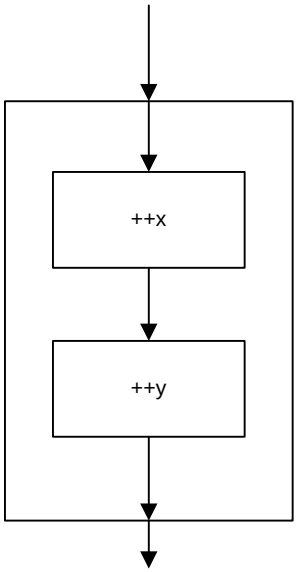
Sequence Points

A clearer presentation of the sequence-points rules could be useful for C, and also for C++ because specification of concurrency and threading involves sequence-point issues. Obviously, this clearer presentation should avoid making any substantive changes to the sequence-point rules as they are understood by implementers of C and C++. This paper is proposed as additional material for the Rationale, if approved by WG14.

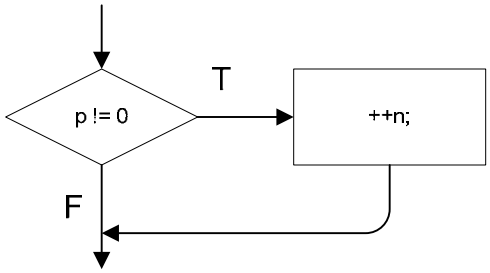
Greater clarity might result from a more graphical presentation of the rules. Each C and/or C++ program can (to a certain level of detail) be presented as a conventional flow-chart using boxes for actions and diamonds for choices:

1. Each full-expression is shown as a box.
2. If there is a side-effect within a condition (inside an if, switch, for, while, do-while, etc.), the evaluation takes place within a box, followed by a diamond specifying a test (e.g., upon a temporary) without side-effects.
3. If the top-level operator within a box is a comma, question-colon, logical-and, or logical-or (the "sequenced" operators), then the box is subdivided into smaller boxes, one for each operand of the top-level operator.
4. Each statement produces a flow-chart according to its semantics.

For example, the statement (block) `{ ++x; ++y; }` produces the following flow-chart (as does the expression-statement `++x, ++y;`):



We can remove any enclosing boxes, leaving only the innermost boxes (call them the "little boxes"). Then the if-statement `if (p != 0) ++n;` produces the following flow-chart:



The result of this process is a flow-chart composed of the "little boxes". Each arrow that flows into a box or out of a box specifies one sequence point. By and large, this much of the sequence-point model is well-understood by programmers and implementers. However, it is only the top-level sequenced operators that produce little boxes in the flow chart; that detail is sometimes not fully understood.

So we have gotten the relatively easy part out of the way first. Most of the subtleties of the sequence-point model concern the contents of the little boxes. The rules of C and/or C++ will in general permit a finite set of allowable orderings for the expressions and side-effects within each little box. The operative rule for C is described in 6.5p2:

> **Between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be read only to determine the value to be stored.**

In C++, the corresponding words are in 5p4:

> **Except where noted, the order of evaluation of operands of individual operators and subexpressions of individual expressions, and the order in which side effects take place, is unspecified. Between the previous and next sequence point a scalar object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be accessed only to determine the value to be stored. The requirements of this paragraph shall be met for each allowable ordering of the subexpressions of a full expression; otherwise the behavior is undefined.**

The C++ version explicitly specifies the role of the "allowable orderings". (In C, the same rule was implicitly understood in the process of interpreting 6.5p2. Obviously, the explicit rule in C++ is the preferable approach to "standardese", but the committees do not differ on the substance of the rule.)

We propose to represent the allowable orderings of each little box in a vertical column of operations. Within each little box there may be sequence points; we represent each sequence point as a semicolon on a line by itself. Consider this sequence of statements:

```
a = 0; b = (a = 1, 2*a) + 3*a;
```

This produces two little boxes, one for each expression-statement. The second little box permits a set of allowable orderings:

Ordering #1	Ordering #2	Ordering #3	Ordering #4	…
store 1 in a ; T1=2*a T2=3*a T3=T1+T2 store T3 in b	T1=3*a store 1 in a ; T2=2*a T3=T1+T2 store T3 in b	store 1 in a ; T1=3*a T2=2*a T3=T1+T2 store T3 in b	store 1 in a T1=3*a ; T2=2*a T3=T1+T2 store T3 in b	

Orderings #2 and #4 (and others) violate the rule about "the prior value shall be accessed only to determine the value to be stored", and therefore this statement has undefined behavior.

Assuming that the code within one little box passes the rule about "behavior of allowable orderings", then we can start each ordering by indicating a conceptual "cacheing" of the initial value of each fetched object. For example, consider this statement:

```
c = a++ + b++;
```

There is no top-level sequenced operator, so this produces one little box. We will add the initial-value fetch as the first step in each ordering. There are twelve allowable orderings:

Ordering #1	Ordering #2	Ordering #3	…	Ordering #12
Ta=a, Tb=b	Ta=a, Tb=b	Ta=a, Tb=b		Ta=a, Tb=b
incr a incr b T1=Ta+Tb store T1 in c	incr a T1=Ta+Tb store T1 in c incr b	incr a T1=Ta+Tb incr b store T1 in c		T1=Ta+Tb incr b incr a store T1 in c

In general, the set of allowable orderings is a kind of cross-product of all the alternative orderings permitted. For example, the order of evaluation of function arguments is unspecified, but there is a sequence point after evaluation of all arguments. For a function invocation with N arguments there may be 2^N (or more) orderings of argument evaluation. Consider this statement:

```
f( g(), b++);
```

There are twelve allowable orderings (using the abbreviation "ret" for the returned value from the preceding function call):

Ordering #1	Ordering #2	Ordering #3	…	Ordering #12
Tb=b	Tb=b	Tb=b		Tb=b
; call g Arg1=ret Arg2=Tb incr b ; call f	Arg2=Tb incr b ; call g Arg1=ret ; call f	Arg2=Tb ; call g incr b Arg1=ret ; call f		incr b Arg2=Tb ; call g Arg1=ret ; call f

There is a sequence point prior to each of the two function calls, and the operation of calling `f` is always the last operation performed, so there are four operations whose order varies among the allowable orderings: "call g", "Arg1=ret", "Arg2=Tb", and "incr b". Complete freedom to order these would produce 4! orderings, but half of those are excluded because "Arg1=ret" must occur after "call g". Therefore, there are 4!/2 allowable orderings, i.e. twelve orderings (again).

Note that it is unspecified whether the incrementation of `b` takes place before or after the call to `g`, but it definitely takes place before the call to `f`.

One interesting result of the discussions within the C++ Concurrency Extensions Subgroup was the discovery that some modern hardware architectures may execute a different ordering of the same instructions at different points in the computation. Until this discovery, many of us had assumed that prior to code generation one specific allowable ordering was chosen to define the semantics as determined by the abstract machine; then optimization and code generation would proceed by the "as-if" rule to produce the observable semantics. Allowing multiple orderings does not fundamentally change the model; each time a little box is executed, one specific allowable ordering will be executed on this iteration.

The C++ Concurrency subgroup intends to define certain aspects of the threading model in terms of two or more abstract machines interacting with each other.