# The New C Standard (Excerpted material)

## An Economic and Cultural Commentary

**Derek M. Jones**
derek@knosof.co.uk

## 6.5.2.4 Postfix increment and decrement operators

### Constraints

postfix operator constraint

The operand of the postfix increment or decrement operator shall have qualified or unqualified real or pointer type and shall be a modifiable lvalue. 1046

#### Commentary

C99 added complex types to the set of scalar types. However, the Committee did not define a meaning for the postfix increment and decrement operators for the complex types. (A natural result, had polar coordinates been chosen for the representation, would have been to increment or decrement the magnitude; one possibility in a cartesian coordinate representation would be to increment the real part, which is the effect in `C+=1`.)

The postfix **++** operator might be used in conjunction with an object of type **`_Bool`** to perform an atomic test-and-set operation. However, C99 does not require an implementation to support this usage, even if the Library typedef name `sig_atomic_t`, defined> has type **`_Bool`**.

#### C90

The C90 Standard required the operand to have a scalar type.

#### C++

D.1p1 *The use of an operand of type **`bool`** with the postfix **++** operator is deprecated.*

#### Other Languages

These operators are usually seen in languages that claim to be *C-like*.

#### Common Implementations

Many processors include instructions that are special cases of other instructions. For instance, addition and subtraction instructions that encode a small constant value as part of the instruction (removing the need to load the constant into a register before using it as an operand). The number of bits used for encoding this constant in an instruction tends to be greater for wider instructions. In some cases (usually 8-bit processors) the instruction may be simply an increment/decrement by one.

Looping constructs often involve incrementing or decrementing an objects value by one. The object may be a loop counter or a pointer into storage. Some processors combine a counting operation, comparison of modified value, and branch based on result of the comparison in a single instruction, while other processors support addressing modes that modify the contents of a register (or storage location) as part of the execution of the instruction.[1] In some cases these addressing modes can be used to efficiently access successive elements of an array. To be able to use these specialized instructions the appropriate constructs need to occur in the source and a translator needs to be capable of detecting it. For instance, the Motorola 68000[1] includes a postincrement and a predecrement addressing mode, which are only useful in some cases.

iteration statement syntax

```
1  while (*p++ == *q++) /* Postincrement */
2     ;
3  while (*--p == *--q) /* Predecrement */
4     ;
5  /* Motorola 68000 addressing modes no use for the following */
6  while (*++p == *++q) /* Preincrement */
7     ;
8  while (*p-- == *q--) /* Postdecrement */
9     ;
```

### Coding Guidelines

Incrementing or decrementing the value of an object by one is a relatively common operation. This usage created two rationales for the original inclusion of these postfix operators in the C language. It simplified the writing of a translator capable of generating reasonable quality machine code and it provided a shortened notation (in both the visual and conceptual sense) for developers. Another rationale for the use of these operators has come into being since they were first created— existing practice. Because they occur frequently in existing source developers have acquired extensive experience in recognizing their use in source. Their presence in existing code ensures that future developers will also gain extensive experience in recognizing their use. These operators are part of the culture of C.

punctuator syntax

The issue of translator optimization increment/decrement operators is not as straight-forward as it might appear; it is discussed elsewhere. The conclusions reached are even more applicable to the postfix operators because of their additional complexity.

prefix ++ incremented

From the coding guideline perspective use of these operators can be grouped into the following three categories:

1. *The only operator in an expression statement.* In this context the result returned by the operation is ignored. The statement simply increments/decrements its operand. Use of the postfix, rather than the prefix, form follows the pattern seen at the start of most visible source code statement lines— an identifier followed by an operator (see Figure **??**). For this reason the postfix operators are preferred over prefix operators, an expression statement context.

prefix ++ incremented

2. *One of the operators in a full expression that contains other operators.* Experience shows that developers' strategy of scanning down the left edge of the source looking for objects that are modified sometimes results in them failing to notice object modifications that occur in other contexts (i.e., a postfix or prefix operation appearing in the right operand of an assignment, or the argument of a function call). It is possible to write the code so that a postfix operator does not occur in the same expression as other operators. Moving the evaluation back before the containing expression removes the need for duplicate operations in each arm of an **if**/**else** (an **else** arm will have to be created if one does not exist) or **switch** statement. For an expression statement the operation can be moved after the containing expression. (If the usage is part of an initializer, the first form is only available in C99, which permits the mixing of declarations and statements.)

full expression

```
...i++...
```

becomes the equivalent form:

```
t=i;
i++;
...t...
```

or:

```
...i...
i++;
```

The total cognitive effort needed to comprehend the first equivalent form may be more than the postfix form (the use of `t` has to be comprehended). However, the peak effort may be less (because the operations may have been split into smaller chunks in serial rather than nested form). The total cognitive effort needed to comprehend the second equivalent form may be the same as the original and the peak effort may be less. Are more faults likely to be introduced through miscomprehension, through a visual code scanning strategy that may fail to spot modifications, or through the introduction of a temporary object? There is insufficient evidence to answer this question at the time of this writing. Although in the case of expression statements there is the possibility of a benefit (at a very small initial, typing cost) to moving the modification operation to after the expression. The issue of side

effects occurring within expressions containing operators that conditionally evaluate their operands is discussed elsewhere.

&&
second operand

3. *The only operator in a full expression that is not an expression statement.* When the postfix operator is the only operator in a full expression it might be claimed that it will not be overlooked by readers. There is no evidence for, or against, this claim.

Your author has encountered other coding guideline authors, whose primary experience is with non-C like languages, who recommend against the postfix and prefix operators. These recommendations seem to be driven by unfamiliarity. Experience suggests that once the novelty has worn off, these developers become comfortable with and even prefer the shorter C forms.

By their very nature the ++ and -- operators often occur in expressions used as indices to storage locations. The off-by-one mistake may be the most commonly occurring mistake associated with array subscripting. For some coding guideline authors, this has resulted in guilt by association. There is no evidence to show that the use of these operators leads to more, or less, mistakes being made than if an alternative expression were used.

prefix ++
incremented

The rationale for preferring prefix operators over postfix operators in some contexts is given elsewhere.

Cg 1046.1

A postfix operator shall not appear in an expression unless it is the only operator in an expression statement.

### Semantics

---

postfix ++
result

The result of the postfix ++ operator is the value of the operand. 1047

#### Commentary

That is, the value before the object denoted by the operand is incremented.

#### Common Implementations

Having to return the original value as the result, and to update the original value, requires at least one more register, or storage location, than the prefix form.

#### Coding Guidelines

cognitive
switch

An occurrence of the postfix ++ operator may cause the reader to make a cognitive task switch (between evaluating the full expression and evaluating the modification of a storage location). It also requires that two values be temporarily associated with the same object. Having to remember two values requires more memory resources than recalling one. There is also the possibility of interference between the two, closely semantically associated, values (causing one or both of them to be incorrectly recalled). As the previous C sentence discussed, rewriting the code so that a postfix operator is not nested within an expression evaluation may reduce the cognitive resources needed to comprehend that piece of source.

---

After the result is obtained, the value of the operand is incremented. 1048

#### Commentary

That is, at the next sequence point the value of the object will be one larger than it was before the evaluation of this operator (assuming no other value is stored into the object before this sequence point is reached, which would result in undefined behavior).

#### C++

5.2.6p1

*After the result is noted, the value of the object is modified by adding 1 to it, unless the object is of type* **bool**, *in which case it is set to* **true**. *[Note: this use is deprecated, see annex D. ]*

The special case for operands of type **bool** also occurs in C, but a chain of reasoning is required to deduce it.

Bool
large enough to store 0 and 1

### Common Implementations

The two values created by the postfix operators (the value before and the value after) are likely to simultaneously require two registers, to hold them. The extent to which the greater call on available processor resources, compared to the corresponding prefix operators, results in less optimal generated machine code will depend on the context in which the operator occurs (i.e., a change to the algorithm to use a different operator may affect the quality of the machine code generated for other operators).

---

1049 (That is, the value 1 of the appropriate type is added to it.)

### Commentary

If USHRT_MAX has the same value as INT_MAX, then the following increment would have undefined behavior, if the value 1 used has type **int**. However, it is defined if the value 1 used has type **unsigned int** (it is assumed that the phrase *appropriate type* is to be interpreted as a type for which the behavior is defined).

```
1  unsigned short u = USHRT_MAX;
2
3  u++;
```

---

1050 See the discussions of additive operators and compound assignment for information on constraints, types, and conversions and the effects of operations on pointers.

postfix operators
see also

### Commentary

The implication is that these constraints and semantics also apply to the postfix ++ operator. The same references are given for the postfix operators.

addition
operand types
compound assignment
constraints
additive operators
semantics
compound assignment
semantics

### C++

The C++ Standard provides a reference, but no explicit wording that the conditions described in the cited clauses also apply to this operator.

*See also 5.7 and 5.17.*

prefix operators
see also

5.2.6p1

---

1051 The side effect of updating the stored value of the operand shall occur between the previous and the next sequence point.

### Commentary

This is a requirement on the implementation. It is a special case of a requirement given elsewhere. The postfix ++ operator is treated the same as any other operator that modifies an object with regard to updating the stored value. It is possible that the ordering of sequence points, during the evaluation of a full expression, is nonunique. It is also possible that the operand may be modified more than once between two sequence points, causing undefined behavior.

sequence points
sequence points

### C++

The C++ Standard does not explicitly specify this special case of a more general requirement.

sequence points

### Common Implementations

Like the implementation of the assignment operators, there may be advantages to delaying the store operation (keeping the new value in a register), perhaps because subsequent operations may modify it again.

assignment-expression
syntax

sequence ?? points all orderings give same value

### Coding Guidelines

The applicable guideline recommendation is discussed elsewhere.

---

postfix -- analogous to ++

The postfix `--` operator is analogous to the postfix `++` operator, except that the value of the operand is decremented (that is, the value 1 of the appropriate type is subtracted from it). 1052

### Commentary

postfix ++ 1047 result

The effects of the `--` operator are not analogous to the postfix `++` operator when the operand has type **`_Bool`** (otherwise the same Commentary and Coding guideline issues also apply). The `--` operator always inverts the truth-value of its operand.

```
1  #include <stdio.h>
2
3  void f(_Bool p)
4  {
5  _Bool q = p;
6
7  q--; q++;
8  p++; p--;
9
10 if (p == q)
11    printf("This implementation is not conforming\n");
12 }
```

### C++

5.2.6p2 *. . . except that the operand shall not be of type* ***`bool`****.*

A C source file containing an instance of the postfix `--` operator applied to an operand having type **`_Bool`** is likely to result in a C++ translator issuing a diagnostic.

---

**Forward references:** additive operators (6.5.6), compound assignment (6.5.16.2). 1053

# References

1. Motorola, Inc. *MOTOROLA M68000 Family Programmer's Reference Manual*. Motorola, Inc, 1992.