

# **The New C Standard** (Excerpted material)

---

**An Economic and Cultural Commentary**

**Derek M. Jones**

derek@knosof.co.uk

## 6.5.16 Assignment operators

assignment-expression  
syntax

```
assignment-expression:
    conditional-expression
    unary-expression assignment-operator assignment-expression
assignment-operator: one of
    = *= /= %= += -= <<= >>= &= ^= |=
```

### Commentary

The syntax is written so that these operators associate to the right. This means that in `x=y=z`, `z` is assigned to `y`, which is then assigned to `x`. Defining assignment as an operator is consistent with the specification that it returns a value. Its definition as an operator makes it possible for a full expression to contain more than one assignment.

One benefit (to translator vendors) of compound assignment operators is that they remove the perceived need for translators to perform certain kinds of optimization. One potential disadvantage to developers of compound assignment operators is that they are likely to have the effect of causing vendors to write translators that do not search for certain kinds of optimization. The underlying reason is the same in both cases. The use of these operators affects the characteristics of the source that translator vendors expect to frequently encounter (e.g., because developers are expected to write `x*=3` rather than `x=x*3`). Thus vendors don't tune optimizers to search for assignments in code that don't make use of compound assignment operators (if this usage is possible).

```
1  struct T {
2      struct fred *next;
3      /* ... */
4      } s_ptr;
5
6  s_ptr = s_ptr->next;
7  s_ptr ->= next;    /* There is no ->= operator. */
```

### C++

```
5.17 assignment-expression: conditional-expression logical-or-expression
assignment-operator assignment-expression throw-expression
```

footnote  
85

For some types, a cast is an lvalue in C++.

### Other Languages

Most languages do not treat assignment as an operator, but as part of the syntax of the assignment statement. Languages in the Algol family use `:=` as the assignment token. The token `/=` denotes the not equal operator in some languages (e.g., Ada). Fortran contains the **ASSIGN** statement, which can be used to assign a label to an object having type **INTEGER**. Perl (and BCPL) support assigning multiple values to multiple objects within the same assignment (e.g., `v1, v2, v3 = e1, e2, e3`). Algol 68 does not treat assignment as an operator, but does treat compound assignment as operators. This results in `x += y := p -= q := r` being parsed as `(x += y) := (p -= q) := r`.

### Common Implementations

The base document supported `+=`, `*=`, and the reversed form of the other compound assignment operators.<sup>[3]</sup>

### Coding Guidelines

What are the costs and benefits of using multiple assignment operators in the same full expression?

The potential benefits, compared to source code where full expressions contain at most one assignment operator, include:

base document

- The resulting program image may contain higher-quality (faster and/or smaller) machine code. However, this benefit may not exist (if the translator used is sufficiently sophisticated that it generates comparable machine code for both cases). The benefit may also be of no significance—the difference in program performance is not noticeably different and the size savings is of no consequence.
- When carefully reading the source (rather than scanning it quickly), a multiple assignment may require less cognitive effort to comprehend than two assignments. For instance, if the objects being assigned to are denoted by some complicated expression, it is necessary to deduce that, for instance, `complicated_y` represents the same object in both cases. This deduction does not need to be performed when a multiple assignment is used:

```

1  complicated_x = complicated_y = some_expression;
2
3  complicated_y = some_expression;
4  complicated_x = complicated_y;
```

- Use of the form `x = y = exp` is not usually taken to imply a causal connection between `x` and `y`, while an assignment of the form `x=y` often implies a causal connection between the two objects. When `exp` is complicated, a multiple assignment may help clarify that any causal association should be to it, not between the two objects assigned to.

The main potential costs are reader miscomprehension of the expression and a possible increase in the maximum cognitive load required to comprehend the expression, as follows.

- The miscomprehension can occur because of the techniques developers use to read source code. When searching for objects that are modified, developers often visually scan along the left edge of the source (making the assumption that all identifiers denoting modified objects appear along this edge). If multiple assignment operators occur within the same expression, it is possible that only the first one of them will be seen. reading  
kinds of
- Increase in cognitive load can be caused by the task switch needed to update a reader's mental model of a program's state. This issue is discussed elsewhere. postfix ++  
result

Multiple objects are sometimes initialized to the same value in a single statement (e.g., `a=b=c=d=0`;). The developer has saved the typing of a few characters at the expense of less-readable source code and an increased likelihood that changes to the source lead to errors. For instance, assigning a different value to `c` implies that the original assignment to `c` would be removed; poor editing could create the expression `a=b==d=0`; (failure to spot the original assignment to `c`, resulting in it being left in the source, could introduce errors if it occurred after a different value had been stored into `c`).

Cg 1288.1

A full expression shall contain at most one assignment operator, and it shall occur as the *top-level* operator in that expression.

The issue of assignment operators occurring in other types of statements that contain expressions is discussed elsewhere. controlling  
expression  
if statement

### Example

```

1  extern int ***x,
2          *y;
3
4  void f(void)
5  {
6  **x+++==***x++;
7  *y*==*y;
8  *y--***x---==*y--;
9  }
```

## Usage

For a comparison with load frequencies see Table ??.

**Table 1288.1:** Common token pairs involving the assignment operators (as a percentage of all occurrences of each token). Based on the visible form of the .c files. Note: entries do not always sum to 100% because several token sequences that have very low percentages are not listed.

Token Sequence	% Occurrence of First Token	% Occurrence of Second Token	Token Sequence	% Occurrence of First Token	% Occurrence of Second Token
identifier %=	0.0	100.0	v++ =	7.6	0.7
identifier /=	0.0	99.3	+= <i>integer-constant</i>	21.7	0.3
identifier >>=	0.0	99.3	= identifier	77.0	0.2
identifier <<=	0.0	97.5	+= identifier	68.0	0.2
identifier +=	0.3	96.3	>>= <i>integer-constant</i>	87.1	0.1
identifier *=	0.0	96.0	-= <i>integer-constant</i>	24.2	0.1
identifier -=	0.1	95.2	&= <i>integer-constant</i>	12.4	0.1
identifier  =	0.3	93.9	= <i>integer-constant</i>	10.7	0.1
identifier &=	0.1	93.1	-= identifier	65.1	0.1
identifier =	9.4	90.9	+= C	6.5	0.1
identifier ^=	0.0	85.9	= C	12.0	0.1
&= ~	75.0	52.5	<<= <i>integer-constant</i>	85.1	0.0
= +v	0.0	45.1	/= <i>integer-constant</i>	52.1	0.0
= <i>floating-constant</i>	0.1	15.7	*= <i>integer-constant</i>	39.8	0.0
= <i>character-constant</i>	0.8	14.2	^= <i>integer-constant</i>	34.5	0.0
= -v	1.6	12.0	%= <i>integer-constant</i>	31.5	0.0
] ^=	0.0	11.1	&= identifier	8.6	0.0
= &v	1.9	10.2	%= identifier	68.1	0.0
= *v	1.1	9.9	^= identifier	46.4	0.0
= <i>integer-constant</i>	19.6	9.0	*= identifier	44.2	0.0
] =	21.8	6.8	/= identifier	34.6	0.0
= identifier	62.5	6.5	<<= identifier	13.4	0.0
= <b>sizeof</b>	0.3	5.9	>>= identifier	10.5	0.0
] &=	0.2	5.7	<b>#error</b> =	16.9	0.0
]  =	0.4	4.6	-= C	7.0	0.0
= C	9.1	3.5	/= C	5.8	0.0
*= <i>floating-constant</i>	6.3	1.6	^= C	13.9	0.0

**Table 1288.2:** Occurrence of executed store instructions (as a percentage of all instructions executed) in two different kinds of functions (*Leaf* functions do not call any other functions, while *Non-Leaf* do). Adapted from Calder, Grunwald, and Zorn.<sup>[2]</sup>

Program	Leaf	Non-Leaf	Program	Leaf	Non-Leaf
burg	34.3	7.7	eqtott	0.0	11.4
ditroff	8.3	8.3	espresso	6.5	3.9
tex	15.1	9.8	gcc	9.6	12.0
xfig	8.0	11.7	li	0.0	16.3
xtex	8.3	11.2	sc	1.2	11.1
compress	83.5	9.2	Mean	15.9	10.2

## Constraints

assignment operator  
modifiable lvalue  
modifiable lvalue

An assignment operator shall have a modifiable lvalue as its left operand.

### Commentary

An object declared using the **const** is an lvalue, but it is not modifiable.

### C90

The C99 Standard has removed the requirement, that was in C90, which lvalues refer to objects. This has

resulted in the conformance status of the assignment `1=3` changing from a constraint violation to undefined behavior. The lvalue `1` does not designate an object and is not const-qualified. Therefore it is not ruled out from being modifiable in C99. lvalue

### Common Implementations

Some implementations support an extension that allows the result of the cast operator to be an lvalue (e.g., assignments of the form `(char)i = 3` are supported).

### Example

In some cases an implementation is likely to diagnose a syntax violation, in an assignment expression, rather than this constraint violation.

```
1 (int)x = 0; /* Syntax violation. */
2 +(int)x = 0; /* Syntax is valid, but violates this constraint. */
```

## Semantics

1290 An assignment operator stores a value in the object designated by the left operand.

### Commentary

The standard does not specify when the value is stored, only that it occurs between the previous and next sequence point. Neither does the standard specifies how the store should be implemented. For instance, storing into an object having a floating type does not require that the value be treated as having a floating type; simple assignment of `a` to `b` may be implemented using machine code that copies blocks of bits, or as the instruction sequence `load floating value/store floating value`. For floating types this difference can matter because assigning a signaling NaN will not raise an exception in the former case, but will in the latter.

modified  
objects  
received cor-  
rect value  
1293 assignment  
when side effect  
occurs

### Other Languages

Assignment is not universal to all programming languages. Some pure functional languages regard assignment as causing a side effect that make it difficult to mathematically prove the correctness of source code. They do not contain any means of assigning a value to an object.

Some languages, often used in distributed computing, perform what is sometimes known as *deep assignment*. For instance, if `X` refers to a tree-like data structure, then assigning `X` to `Y` has the effect of making a copy of the tree and assigning a reference to it to `Y`; `X` and `Y` will then point at different trees, which contain nodes having the same values (apart from the pointers to other nodes).

### Common Implementations

Whether the store into an object specified in the source code actually occurs in the generated machine code is invisible to the developer. Optimizers frequently try to remove stores by keeping the value in a register (perhaps until a new value is assigned). For processors with many registers, functions containing a small number of automatic objects may never need to store assignments to those objects.

### Usage

A study by Lepak, Bell, and Lipasti<sup>[4]</sup> investigated value locality with respect to store operations (using the SPEC95 benchmarks). They defined a *silent store* to be a store operation that does not change the system state (i.e., the value being written matches the value already held at the location being stored to). They also defined *program structure store value locality* (PSSVL) to refer to the same value being stored from the same program location and *message-passing store value locality* (MPSVL) to refer to the same value being stored to the same address in storage (which may be holding different objects at different times during the execution of a program). value locality

**Table 1290.1:** Percentage of stores that are *silent*. The results from two instruction sets, the PowerPC (PPC) and SimpleScalar (SS), are given for silent stores. The measurements for Program Structure Store Value Locality (PSSVL) and Message-Passing Store Value Locality (MPSVL) are for the PowerPC only. Adapted from Lepak, Bell, and Lipasti.<sup>[4]</sup>

Program	Silent stores (PPC/SS)	PSSVL (PPC)	MPSVL (PPC)	Program	Silent stores (PPC/SS)	PSSVL (PPC)	MPSVL (PPC)
go	38/27	30	36	tomcatv	47/33	40	45
m88ksim	68/62	56	65	swim	34/26	20	19
gcc	53/46	37	49	mgrid	23/ 7	24	17
compress	42/39	35	16	applu	37/35	35	28
li	34/20	32	34	apsi	21/25	22	20
jpeg	43/33	52	46	fpppp	15/15	15	14
perl	49/36	39	42	wave5	25/22	30	20
vortex	64/55	71	57				

Results for two processors and their associated translators are given in Table 1290.1. Differences in internal housekeeping operations, such as saving registers as part of a function-call operation, can affect the results. The same authors<sup>[1]</sup> found that zero was the most common silent store value (46% of silent stores), with one being 7%, 2 to 9 being 4%, and values greater than 100 million being 26%. For floating-point, values greater than 100 million were the most common at 65%, with zero being 37%.

function call  
preparing for

An assignment expression has the value of the left operand after the assignment, but is not an lvalue.

1291

### Commentary

Using the value of an assignment expression does not mean that the side effect of storing that value in the left operand has taken place.

WG14/N847 *If a guarantee is needed that the assignment will have taken place before the value is used, or an lvalue is required. The expression  $((a=b), a)$  can be used.*

*Two interpretations have been put on this wording:*

- *the value of the assignment expression is the value that will also be stored in the left operand ("same-value" semantics);*
- *the value of the assignment expression is the result of reading the left operand after storing the value in it ("write-then-read" semantics).*

*These two have different results when the left operand is a volatile object that can be changed by external causes (such as a clock or a memory-mapped device register). This ambiguity needs to be resolved.*

*Consider the code:*

```
int x;
extern volatile int system_timer; // precision of 1 microsecond
extern volatile int serial_port; // writing sends a word, reading
// ...
// returns the next word received
// ...
x = system_timer = 42; // statement 1
serial_port = 66; // statement 2
```

*With same-value semantics, statement 1 will set x to 42 and will send the value 66 to the serial port. With write-then-read semantics, statement 1 will set x to some other value (the change in the timer between writing to it and reading it back).*

*More important, though, is the effects of statement 2 in write-then-read semantics. Because a statement expression is evaluated for its side effects, it is reasonable to require the value of the assignment statement to be determined before being thrown away (in particular, there is *\*no\** statement in the Standard as to when the value of the assignment expression is or is not evaluated). This means that statement 2 always has the side effect of reading a word from the serial port, and there is no way to write without reading.*

assignment  
value of

**C++**

... ; the result is an lvalue.

5.17p1

The C++ DR #222 (which at the time of this writing is at the drafting stage) queries some of the consequences of the result being an lvalue.

Source developed using a C++ translator may contain assignments that are a constraint violation if processed by a C translator.

```

1  extern int glob;
2
3  void f(void)
4  {
5  int x;
6  volatile int y;
7
8  (glob += 5) += 6; /* constraint violation */
9                  // current status undefined behavior, object modified
10                 // twice between sequence points. The response to DR #222
11                 // may add a sequence point, making the behavior defined
12
13  x = y = 0; /* equivalent to y=0; x=0; */
14            // equivalent to y=0; x=y;
15  }
```

**Other Languages**

In the past languages have not generally treated assignment as an operator. Whether more recent languages are treating it as an operator because of the effects of the widespread teaching of C to students over several decades or for other reasons is not known.

**Coding Guidelines**

Experience shows that developers sometimes incorrectly assume that the result of the assignment operator is the value of the right operand before it is converted to the type of the left operand.

Rev 1291.1

If the result of an assignment operator is used and the types of its two operands are different, the source shall be checked to ensure that the value of the right operand is not being expected.

**Example**

In the first statement in the following the value of LONG\_MAX will be implicitly converted (to the value 2147483647) and assigned to ui.

```

1  #include <limits.h>
2
3  void f(void)
4  {
5  unsigned char uc; /* Assume 8 bits. */
6  unsigned short us; /* Assume 16 bits. */
7  unsigned int ui; /* Assume 32 bits. */
8
9  uc=us=ui=LONG_MAX;
10 ui=us=uc=LONG_MAX;
11 }
```

This value is what is assigned to us, but first it has to be converted, giving a value of 65535. This value is then assigned to uc, but first it must be converted, giving a value of 255. In the second statement the value 255 is assigned to all three objects.

assignment  
result type

The type of an assignment expression is the type of the left operand unless the left operand has qualified type, in which case it is the unqualified version of the type of the left operand. 1292

### Commentary

This requirement is consistent with the value of an assignment expression being that of the left operand.

### Other Languages

Algol 68 treats an assignment as returning an lvalue, so it is possible to write:

```
1  INT x;
2  REF INT r;
3  r := x := 42;
```

which causes r to point to x, which is assigned the value 42.

### Coding Guidelines

The incorrect developer assumption described in the previous C sentence also applies to the type of the result.

Rev 1292.1

If the result of an assignment operator is used and the types of its two operands are different, the source shall be checked to ensure that the type of the right operand is not being expected.

### Example

```
1  extern unsigned char uc;
2  extern float fl;
3
4  void f(void)
5  {
6  if ((fl += 1) == 3)           /* Equality comparison of floating type. */
7  ;
8  if (((uc = 3.5) + 0.5) == fl) /* Adding 0.5 to an unsigned char? */
9  ;
10 }
```

assignment  
when side effect  
occurs

The side effect of updating the stored value of the left operand shall occur between the previous and the next sequence point. 1293

### Commentary

This is a requirement on the implementation. Like other operators that generate side effects, the exact time when the side effect occurs is not specified, only the bounds between when it must occur. Most assignment operators occur in the context of an expression statement, which is a full expression and hence has a sequence point after its evaluation.

### C++

The C++ Standard does not explicitly state this requirement.

### Other Languages

In languages that do not treat assignment as an operator, updating the stored value is the last operation performed in that statement (which may contain function calls that modify objects).

### Coding Guidelines

It is not always possible to predict the ordering of sequence points chosen by a translator, and the guideline recommendation dealing with expression order evaluation is applicable here.

sequence  
points  
expression  
statement  
syntax  
full ex-  
pression  
expression  
statement  
full ex-  
pression  
sequence point

sequence  
points  
sequence ??  
points  
all orderings  
give same value

1294 The order of evaluation of the operands is unspecified.

assignment  
operand eval-  
uation order

### Commentary

Many developers have a mental model of assignment that involves evaluating one of the operands first. (This is often a hang over from how they were first taught to write programs.) The evaluation of the operands has the same behavior as most other binary operators in C. This sentence is simply a restatement of this fact.

expression  
order of evaluation

### C++

The C++ Standard does not explicitly make this observation.

### Other Languages

Few languages specify an ordering on the evaluation of the left and right operands of an assignment, even if they do not consider assignment to be an operator. Java specifies a left-to-right evaluation order.

### Common Implementations

Unsophisticated translators will evaluate the right operand first, on the basis that it is likely to be the most complex (and therefore require the greater number of temporary registers). More sophisticated translators will estimate the complexity of both operands and evaluate the most complex one first.

### Coding Guidelines

The issues generated by some developers, who have a mental model of assignment that involves one operand always being evaluated before the other, are the same as for the other binary operators. The guideline recommendation dealing with expression order evaluation is applicable here.

expression  
order of evaluation  
?: sequence  
points  
all orderings  
give same value

### Example

```

1  #include <stdio.h>
2
3  char arr[20];
4
5  void f(void)
6  {
7  arr[printf("Hello ")] = printf("World\n");
8  }
```

1295 If an attempt is made to modify the result of an assignment operator or to access it after the next sequence point, the behavior is undefined.

### Commentary

In some cases it is possible to create an lvalue from the result returned by the assignment operator (these invariably involve constructs whose implementation involves the use of temporary storage locations). The order of evaluation of most operands is unspecified and when it is specified, a sequence point also occurs. This C sentence effectively specifies that there is no requirement for implementations to support developer access to these temporary objects.

### C90

This sentence did not appear in the C90 Standard and had to be added to C99 because of a change in the definition of the term lvalue.

lvalue

### C++

The C++ definition of lvalue is the same as C90, so this wording is not necessary in C++.

### Common Implementations

The temporary storage locations used internally by an implementation are usually allocated within the stack frame of the function invocation that uses them. As such, they will cease to exist when that function returns. These storage locations may also be used to hold temporary results from the evaluation of other operations.

### Coding Guidelines

For this undefined behavior to occur an expression must contain more than one assignment operator. The guideline recommendation dealing with the use of multiple assignment operators in the same expression is therefore applicable. The code that needs to be written to generate this behavior is sufficiently obscure and unlikely to occur that no guideline recommendation is given here.

### Example

```
1  struct {
2      int m[10];
3      } x, y;
4
5  void f(void)
6  {
7      unsigned char uc_1;
8      int *p_1 = &((x = y).m[3]);
9
10     *p_1 = 4;
11     p_1 = &((x = y).m[uc_1=0, 2]);
12
13     /*
14     * Result accessed before sequence point, but behavior is unspecified(?)
15     * because the object written to is also read from in the same subexpression.
16     */
17     y.m[4] = (*p_1 = &((x = y).m[3])) + *(p_1++);
18 }
```

full ex-1288.1  
pression  
at most one  
assignment

## References

1. G. B. Bell, K. M. Lepak, and M. H. Lipasti. Characterization of silent stores. In *International Conference on Parallel Architectures and Compilation Techniques (PACT'00)*, pages 133–144. IEEE, Oct. 2000.
2. B. Calder, D. Grunwald, and B. Zorn. Quantifying behavioral differences between C and C++ programs. *Journal of Programming Languages*, 2(4):313–351, 1995.
3. B. W. Kernighan. Programming in C-A tutorial. Technical Report ???, Bell Laboratories, Aug. ???
4. K. M. Lepak, G. B. Bell, and M. H. Lipasti. Silent stores and store value locality. *IEEE Transactions on Computers*, 50:1174–1190, 2001.