

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

---

**An Economic and Cultural Commentary**

**Derek M. Jones**

derek@knosof.co.uk

## 6.5.16.1 Simple assignment

### Constraints

simple as-  
signment  
constraints  
equality  
operators  
constraints

One of the following shall hold:<sup>94)</sup>

1296

#### Commentary

This list is very similar to that given for the equality operators.

#### C++

The C++ Standard does not provide a list of constraints on the operands of any assignment operator (5.17). Clause 12.8 contains the specification that leads the following difference:

- C1.8 *The implicitly-declared copy constructor and implicitly-declared copy assignment operator cannot make a copy of a volatile lvalue. For example, the following is valid in ISO C:*

```
struct X { int i; };
struct X x1, x2;
volatile struct X x3 = {0};
x1 = x3;      // invalid C++
x2 = x3;      // also invalid C++
```

*Rationale: Several alternatives were debated at length. Changing the parameter to volatile const X& would greatly complicate the generation of efficient code for class objects. Discussion of providing two alternative signatures for these implicitly-defined operations raised unanswered concerns about creating ambiguities and complicating the rules that specify the formation of these operators according to the bases and members.*

— the left operand has qualified or unqualified arithmetic type and the right has arithmetic type;

1297

#### Commentary

Any arithmetic value can be assigned to any object that has an arithmetic type. Conversions are specified to handle all cases where the two arithmetic types are not the same. This constraint does not prohibit the type of the left operand being const-qualified. However, such an object would not be a modifiable lvalue, and such usage would violate a constraint that applies to all assignment operators.

#### C++

operators  
cause conversions  
modifiable  
lvalue  
assignment  
operator  
modifiable lvalue

- 5.17p3 *If the left operand is not of class type, the expression is implicitly converted (clause 4) to the cv-unqualified type of the left operand.*

The conversions in clause 4 do not implicitly convert enumerated types to integer types and vice versa.

```
1 extern int glob;
2
3 enum {E1, E2};
4
5 void f(void)
6 {
7   glob = E1; /* does not affect the conformance status of the program */
8             // ill-formed
9 }
```

## Other Languages

Strongly typed languages sometimes require the names of the types used to declare the operands to be the same. There are usually special rules to cover literal values.

## Common Implementations

Simple assignments of the form  $x=y$ ; are very common. Some processors (e.g., the Motorola 68000<sup>[2]</sup>) support memory-to-memory copies (no implicit conversions are performed, so both operands must have the same representation). A processor that supports memory-to-memory copies needs to encode two addresses—the source and destination—within the instruction. Such encoding requires a large number of bits (some Motorola 68000 instructions are 13 bytes long). Considering the design of instruction encodings from a more global perspective, there are cost/benefit processor implementation advantages in having fixed-width instructions, and there is often a greater benefit to be had in using the available instruction bits to specify other operations. For these reasons it is very rare to find a modern processor that supports memory to memory operations. On most processors the assignment  $x=y$  is implemented as “load value into register, store register contents into object”. Depending on the local source code context, optimizers may be able to reuse the value held in the register.

## Coding Guidelines

The guideline recommendation for binary operators with an operand having an enumeration type is applicable here.

?? enumeration  
constant  
as operand

1298— the left operand has a qualified or unqualified version of a structure or union type compatible with the type of the right;

assignment  
structure types

## Commentary

Structure/union assignment was introduced in C90; it was not supported in the base document. Structure assignment was first specified in a document listing extensions made to the base document.

base docu-  
ment

Structure assignment can be more efficient than using the `memcpy` library function. This is because the implementation of `memcpy` has to assume worst-case alignment and copy a byte at a time (some very good implementations do better). The translator knows the alignment of an object having structure type and may be able to copy its contents in larger chunks. It may even be worthwhile to copy an element at a time.

footnote  
42

Structures sometimes have an array as their only member, which effectively allows arrays to be assigned.

## C++

Clause 13.5.3 deals with this subject, but does not discuss this particular issue.

## Other Languages

It seems to be quite common for the first versions of a language definition to not support structure assignment. Such support is often added in later revisions of language definitions.

## Common Implementations

Copying a structure a member at a time (or even in larger units— e.g., a long’s worth of bits) produces the fastest code, but at the potential cost of a large number of instructions. Generating a loop is often more compact (except for the case of small structures), but runs more slowly because of the housekeeping overhead of controlling a loop (and the backward jump, potentially flushing the instruction cache). Most implementations generate machine code to copy a member at a time for small structure types and looping machine code for larger one. Depending on the method selected by the implementation padding bits, may or may not also be copied.

cache

## Coding Guidelines

Use of the assignment operator can result in the object assigned to containing a different sequence of bits than if the `memcpy` library function had been used. In the former case a call to the `memcpy` library function may not return zero, while in the latter it will always return zero. The reason for using `memcpy` is that the equality operators do not support operands having a structure or union type. A strong case can be made for

equality  
operators  
constraints

using `memcmp` to compare two structure objects, other than the (usually) spurious efficiency reason; if new members are added to a structure type, a comparison using `memcmp` will not have to be modified, while one that checks individual members will need to be updated to reflect the presence of the new member.

Cg 1298.1

Objects having structure or union type that are compared using the `memcmp` library function shall not appear as the immediate operands of an assignment operator.

### Example

```

1  #include <stdio.h>
2  #include <string.h>
3
4  struct {
5      long m1;
6      char m2;
7      } x, y;
8
9  int main(void)
10 {
11     x = y;
12     if (memcmp(&x, &y, sizeof(x)) == 0)
13         printf("either there are no padding bits, the assignment operator copies all"
14              "padding bits, or the padding bits happen to be the same\n");
15
16     memcpy(&x, &y, sizeof(x));
17     if (memcmp(&x, &y, sizeof(x)) != 0)
18         printf("something wrong here\n");
19
20     memcpy(&x, &y, sizeof(x));
21     x.m1=y.m1;
22     if (memcmp(&x, &y, sizeof(x)) != 0)
23         printf("member assignment affects padding bits\n");
24 }
```

---

— both operands are pointers to qualified or unqualified versions of compatible types, and the type pointed to by the left has all the qualifiers of the type pointed to by the right; 1299

### Commentary

Given the following declarations (where `Q_0` and `Q_1` represent zero or more qualifiers, and `T` is an unqualified type; with the lowercase names denoting the same entities):

```

1  T Q_1 * Q_0 left_operand;
2  t q_1 * q_0 right_operand;
```

this constraint requires that: the types `T` and `t` be compatible, and that all the qualifiers in `q_1` also be in `Q_1`. It is permitted for `Q_1` to contain qualifiers that are not in `q_1`. The case of `Q_0` containing the `const` qualifier is covered by wording given elsewhere.

### C++

The C++ wording (5.17p3) requires that an implicit conversion exist.

The C++ requirements (4.4) on which implicit, qualified conversions are permitted are those described in the Smith paper (discussed elsewhere).

The pointer assignments supported by C++ are a superset of those supported by C. Source developed using a C++ translator may contain constraint violations if processed by a C translator, because it contains

pointer  
qualified/unqualified  
versions

compati-  
ble type  
if  
assignment  
operator  
modifiable lvalue

pointer  
converting qual-  
ified/unqualified

assignments between incompatible pointer types. The following example illustrates differences between the usages supported by C and C++ when types using two levels of pointer are declared.

```

1 void Jon_Krom(void)
2 {
3 /*
4  * The issue of what is safe or unsafe is discussed elsewhere.
5  * An example of case 3 is given in the standard.
6  */
7
8 typedef int T; /* for any type T */
9
10 T      *      * ppa ;
11 T      *      * ppb ;
12
13 T      * const * pcpa ;
14 T      * const * pcpb ;
15
16 T const *      * cppa ;
17 T const *      * cppb ;
18
19 T const * const * cpcpa ;
20 T const * const * cpcpb ;
21
22                // Safe      Allowed   Allowed
23                // or       in        in
24                // Unsafe   C99      C++
25                // -----
26 ppb = ppa ;     // Safe      Yes       Yes    1
27 pcpb = ppa ;    // Safe      Yes       Yes    2
28 cppb = ppa ;    // Unsafe   No        No     3
29 cpcpb = ppa ;   // Safe      No        Yes    4
30
31 ppb = pcpa ;    // Unsafe   No        No     5
32 pcpb = pcpa ;   // Safe      Yes       Yes    6
33 cppb = pcpa ;   // Unsafe   No        No     7
34 cpcpb = pcpa ;  // Safe      No        Yes    8
35
36 ppb = cppa ;    // Unsafe   No        No     9
37 pcpb = cppa ;   // Unsafe   No        No    10
38 cppb = cppa ;   // Safe      Yes       Yes   11
39 cpcpb = cppa ;  // Safe      Yes       Yes   12
40
41 ppb = cpcpa ;   // Unsafe   No        No    13
42 pcpb = cpcpa ;  // Unsafe   No        No    14
43 cppb = cpcpa ;  // Unsafe   No        No    15
44 cpcpb = cpcpa ; // Safe      Yes       Yes   16
45 }

```

## Other Languages

Languages that support some form of type qualifier (e.g., **readonly**) usually have similar asymmetric constraints on pointer assignment.

## Coding Guidelines

If differently qualified pointers are used to access the same object, in an overlapping time frame, it is possible that readers of the source will make an incorrect assumption about the type of one or more of them. For instance, if a pointer to **const** T and a pointer to T both point at the same object, a developer (who is not aware of this state of affairs) may assume that the pointer to const-qualified type cannot have its value changed. Qualified pointers are a special case of object aliasing and the conclusion reached there is applicable.

— one operand is a pointer to an object or incomplete type and the other is a pointer to a qualified or unqualified version of `void`, and the type pointed to by the left has all the qualifiers of the type pointed to by the right; 1300

**Commentary**

generic pointer

Either operand can have a pointer to `void` type. This combination of operands provides implicit support for the concept of a generic pointer type (no explicit casts are required).

The issues relating to qualifiers are the same as those in the previous C sentence.

**C++**

5.17p3 *If the left operand is not of class type, the expression is implicitly converted (clause 4) to the cv-unqualified type of the left operand.*

The C++ Standard only supports an implicit conversion when the left operand has a pointer to `void` type, 4.10p2.

```
1 char *pc;
2 void *pv;
3
4 void f(void)
5 {
6 pc=pv; /* does not affect the conformance status of the program */
7 // ill-formed
8 }
```

**Coding Guidelines**

equality operators pointer to incomplete type

The coding guideline discussion on the equality operators is applicable here.

— the left operand is a pointer and the right is a null pointer constant; 1301

**Commentary**

A null pointer constant can be converted to any pointer type.

**Other Languages**

Languages that support pointer data types invariably have some form of null pointer that can be assigned to an object having any other pointer type.

**Coding Guidelines**

The issue of explicit casts is discussed elsewhere.

or— the left operand has type `_Bool` and the right is a pointer. 1302

**Commentary**

This specification is consistent with operands in other contexts having pointer type being implicitly compared for equality with 0 (the null pointer constant).

**C90**

Support for the type `_Bool` is new in C99.

**C++**

Support for the type `_Bool` is new in C99 and is not specified in the C++ Standard. However, the C++ Standard does specify (4.12p1) that rvalues having pointer type can be converted to an rvalue of type `bool`.

operand compared against 0 && operand compare against 0 if statement operand compare against 0 \_Bool converted to

## Other Languages

Other languages do not usually perform an implicit comparison when the operand has point type.

## Coding Guidelines

Support for the `_Bool` type is new in C99 and at the time of this writing there is insufficient experience available in its use to know if any guideline recommendation is worthwhile.

## Semantics

1303 In *simple assignment* (`=`), the value of the right operand is converted to the type of the assignment expression and replaces the value stored in the object designated by the left operand.

simple as-  
assignment

## Commentary

This defines the term *simple assignment*. Because it is the most commonly seen form of assignment operator developers invariably shorten it to *assignment*. The less-frequent forms of assignment are known by longer terms (see Table ??).

The type of the result of the assignment operator is the unqualified type of its left operand. The standard does not guarantee that assignment is an atomic operation unless the left operand has type `sig_atomic_t`. Assigning an object having a structure type is not always identical to assigning its members individually. There may be an order dependency.

assignment  
result type  
modified  
objects  
received cor-  
rect value

```

1  struct S_Pair;
2
3  typedef struct Object {
4      struct S_Pair *addr;
5      int tag;
6      } Object;
7  struct S_Pair {
8      Object car;
9      Object cdr;
10     };
11
12  Object x;
13
14  void copy_obj(void)
15  {
16  x = x.addr->cdr;
17
18  /* is not the same as: */
19
20  x.addr = x.addr->cdr.addr;
21  x.tag = x.addr->cdr.tag;
22  }
```

## Common Implementations

In most implementations the store operation for an assignment operator, for scalar types whose bit representation is not wider than the width of the processor data bus (which is unlikely to include complex types), is an atomic operation. Unless objects having structure or union types that can fit in a single register, stores of their values are unlikely to be atomic operations.

A surprising number of assignment operations store a value that is equal to the value already held in memory. Researchers<sup>[1]</sup> are starting to adapt algorithms used to detect redundant load operations to optimize away such redundant store operations.

assignment-  
expression  
syntax  
redundant  
code

## Coding Guidelines

If the right operand does not have the same type as the left operand, it will be implicitly converted to that type. Assignment is different from all other binary operators in that it can cause the value of one of its operands to

be implicitly converted to a type that has a lower integer rank, or a narrower floating type. A consequence of such a conversion is that the assignment `x=y` does not guarantee that a subsequent equality operation `x==y` will be true. This situation does not occur if the guideline recommendation specifying the use of a single integer type is followed.

The guideline recommendations applicable to assignment include those given for conversions.

If the value being stored in an object is read from another object that overlaps in any way the storage of the first object, then the overlap shall be exact and the two objects shall have qualified or unqualified versions of a compatible type;

### Commentary

This requirement (on programs) is needed because implementations may choose to perform an assignment by copying values from the right operand to the left operand a byte at a time. It would be surprising if the assignment `x=x` did not always deliver the expected result (an unchanged `x`; unless it is declared with the **volatile** qualifier).

For a given implementation it is possible for two types to have the same size and to overlap exactly, but not be compatible (e.g., types **int** and **long**). The conformance status of a program cannot depend on the implementation, hence the additional requirement on compatible types. The issue of overlapping objects is primarily of importance when assigning members of an object having a union type.

```

1  union {
2      int m_1;
3      long m_2;
4      int m_3;
5      struct {
6          char m_4;
7          long m_5;
8      } m_6;
9  } x;
10
11 void f(void)
12 {
13     x.m_1=x.m_2;    /* Not compatible types. */
14     x.m_1=x.m_3;    /* Covered by this requirement. */
15     x.m_2=x.m_6.m_5; /* Overlap (if it exists) not exact. */
16 }

```

Requiring that all overlapping assignments work as expected would sometimes involve the use of temporary storage locations. Such an overhead during program execution was considered to be excessive. It would require the use of the `memmove` library function rather than the `memcpy` library function.

### C++

The C++ Standard requires (5.18p8) that the objects have the same type. Even though the rvalue may have the same type as the lvalue (perhaps through the use of an explicit cast), this requirement is worded in terms of the object type. The C++ Standard is silent on the issue of overlapping objects.

```

1  enum E {E1, E2};
2  union {
3      int m_1;
4      enum E m_2;
5  } x;
6
7  void f(void)
8  {
9     x.m_1 = (int)x.m_2; /* does not change the conformance status of the program */
10                    // not defined?
11 }

```

object ??  
int type only  
operand  
convert au-  
tomatically

assignment  
value overlaps  
object

1304

object types

## Other Languages

Some languages treat the value of the right operand as being independent of the object referred to by the left operand. In these languages if there is any possibility of overlap and the generated machine code has to perform the assignment by copying multiple storage units of the value, then a temporary has to be used to hold the value so that partial stores into the object do not modify the value being assigned.

1305 otherwise, the behavior is undefined.

### Commentary

Objects can overlap in a variety of ways and implementations can copy objects using a variety of techniques. The Committee chose not require implementations to support any particular set of behaviors.

### Common Implementations

For objects having scalar types, the copying is likely to be performed by loading the value into a register. Once the value has been completely loaded, how the subsequent store is performed does not affect the final result. It is those assignments that require more than one load instruction (where the source and destination overlap) where the behavior can vary. The result will depend on the relative addresses of the objects and whether the copying starts at the lowest or the highest byte of the objects.

### Coding Guidelines

No guideline recommendation is given for this situation because occurrences of this usage are assumed to be rare.

1306 EXAMPLE 1 In the program fragment

```
int f(void);
char c;
/* ... */
if ((c = f()) == -1)
    /* ... */
```

the `int` value returned by the function may be truncated when stored in the `char`, and then converted back to `int` width prior to the comparison. In an implementation in which “plain” `char` has the same range of values as `unsigned char` (and `char` is narrower than `int`), the result of the conversion cannot be negative, so the operands of the comparison can never compare equal. Therefore, for full portability, the variable `c` should be declared as `int`.

### Commentary

There are also less visibly obvious situations involving a value of -1. For instance, the EOF macro.

limit  
case labels

### Coding Guidelines

The coding guideline discussion on the assignment operator appearing in a controlling expression is applicable here.

controlling  
expression  
if statement

1307 94) The asymmetric appearance of these constraints with respect to type qualifiers is due to the conversion (specified in 6.3.2.1) that changes lvalues to “the value of the expression” which and thus removes any type qualifiers from the type category of the expression that were applied to the type category of the expression (for example, it removes `const` but not `volatile` from the type `intvolatile*const`).

footnote  
94

### Commentary

The definition of type category does not include qualifiers. There is even a C sentence that says so. All of the contexts where the conversion is not performed do not apply to the immediate left operand of the assignment operator.

type category  
qualifiers  
representation and  
alignment  
lvalue  
converted to  
value

The wording was changed by the response to DR #272.

**C++**

Even though the result of a C++ assignment operator is an lvalue, the right operand still needs to be converted to a value (except for reference types, but they are not in C) and the asymmetry also holds in C++.

---

EXAMPLE 2 In the fragment:

1308

```
char c;
int i;
long l;

l = (c = i);
```

the value of `i` is converted to the type of the assignment expression `c = i`, that is, **char** type. The value of the expression enclosed in parentheses is then converted to the type of the outer assignment expression, that is, **long int** type.

**Commentary**

While C++ supports the form `(l = c) = i`;, C does not.

**Coding Guidelines**

The coding guideline discussion on the use of more than one assignment operator in an expression is applicable here.

assignment  
value of

full ex-??  
pression  
at most one  
assignment

---

EXAMPLE 3 Consider the fragment:

1309

```
const char **cpp;
char *p;
const char c = 'A';

cpp = &p;           // constraint violation
*cpp = &c;         // valid
*p = 0;           // valid
```

The first assignment is unsafe because it would allow the following valid code to attempt to change the value of the const object `c`.

**Commentary**

The term *unsafe* is often used to refer to an operation, which although itself is conforming, whose resulting value may subsequently be a cause of undefined behavior.

EXAMPLE  
const pointer

## References

1. K. D. Cooper and L. Xu. An efficient static analysis algorithm to detect redundant memory operations. In *Workshop on Memory Systems Performance (MSP '02)*, pages 97–107. ACM Press, June 2002.
2. Motorola, Inc. *MOTOROLA M68000 Family Programmer's Reference Manual*. Motorola, Inc, 1992.