

The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones

derek@knosof.co.uk

6.5.16.2 Compound assignment

Constraints

1310

compound
assignment
constraints

For the operators += and -= only, either the left operand shall be a pointer to an object type and the right shall have integer type, or the left operand shall have qualified or unqualified arithmetic type and the right shall have arithmetic type.

Commentary

addition
operand types

The discussion on the constraints for the additive operators are applicable here.

Coding Guidelines

The issue of operands having a boolean or symbolic role is discussed elsewhere.

boolean role
symbolic
name
addition
operand types

Table 1310.1: Occurrence of assignment operators having particular operand types (as a percentage of all occurrences of each operator; an `_` prefix indicates a literal operand, occurrences below 2.3% were counted as *other-types*). Based on the translated form of this book's benchmark programs.

Left Operand	Operator	Right Operand	%	Left Operand	Operator	Right Operand	%
other-types	=	other-types	34.5	float	/=	float	6.4
other-types	+=	other-types	33.5	unsigned short	=	_int	6.2
other-types	=	other-types	32.8	ptr-to	+=	_int	6.2
int	%=	_int	31.0	unsigned long	=	int	6.1
ptr-to	=	ptr-to	29.7	unsigned int	-=	unsigned int	5.9
int	*=	_int	29.5	unsigned short	>>=	_int	5.8
long	-=	long	28.9	unsigned char	<<=	_int	5.7
unsigned int	<<=	_int	28.3	other-types	%=	other-types	5.7
unsigned int	>>=	_int	28.2	long	+=	_int	5.6
unsigned int	^=	unsigned int	26.7	long	*=	_int	5.3
int	>>=	_int	26.2	unsigned long	&=	int	5.1
int	<<=	_int	25.5	unsigned long	/=	_int	5.0
int	/=	_int	23.8	unsigned int	&=	unsigned int	4.6
int	+=	int	22.1	unsigned int	=	unsigned int	4.6
unsigned char	&=	int	19.7	long	%=	_int	4.6
unsigned int	&=	int	19.4	unsigned short	/=	_int	4.5
int	-=	int	17.4	unsigned char	&=	_int	4.3
long	^=	long	16.9	unsigned long	=	_int	4.1
other-types	*=	other-types	16.8	unsigned char	=	int	3.9
other-types	&=	other-types	16.7	long	<<=	_int	3.8
int	&=	int	16.2	float	*=	_double	3.7
unsigned long	<<=	_int	15.9	unsigned int	+=	unsigned int	3.5
other-types	^=	other-types	15.3	long	&=	int	3.5
other-types	/=	other-types	14.4	unsigned int	=	unsigned int	3.4
other-types	=	other-types	13.5	int	%=	unsigned int	3.4
unsigned int	/=	_int	12.9	unsigned long	^=	int	3.3
ptr-to	+=	int	12.8	float	*=	double	3.3
unsigned int	%=	_int	12.6	unsigned long	*=	_int	3.1
int	%=	int	12.6	unsigned char	^=	unsigned char	3.1
int	=	int	12.3	unsigned char	^=	int	3.1
unsigned int	=	_int	12.1	ptr-to	+=	unsigned long	3.1
float	*=	float	12.1	double	*=	double	3.1
int	=	_int	12.0	unsigned short	/=	unsigned short	3.0
unsigned char	=	_int	11.7	unsigned short	=	int	3.0
unsigned int	%=	unsigned int	11.5	int	/=	unsigned int	3.0
unsigned char	%=	_int	11.5	float	/=	int	3.0
int	/=	int	11.4	double	/=	double	3.0
unsigned long	^=	unsigned long	11.3	unsigned int	+=	_int	2.9
int	^=	_int	11.1	float	*=	_int	2.9
int	=	_int	11.0	unsigned long	+=	unsigned long	2.8
unsigned char	>>=	_int	10.3	unsigned long	=	unsigned long	2.8
other-types	>>=	other-types	9.6	unsigned long	=	long	2.8
unsigned long	>>=	_int	9.5	long	=	long	2.8
int	*=	int	9.3	int	&=	_int	2.8
unsigned short	<<=	_int	8.9	float	=	float	2.8
unsigned int	*=	_int	8.4	unsigned int	-=	int	2.7
int	-=	_int	8.0	int	>>=	int	2.7
unsigned short	&=	int	7.9	int	^=	int	2.7
long	>>=	_int	7.7	unsigned char	=	_int	2.6
unsigned int	=	int	7.5	float	-=	float	2.6
long	/=	_int	7.4	unsigned long	=	unsigned long	2.5
int	+=	_int	7.4	unsigned long	<<=	unsigned int	2.5
int	=	int	7.4	int	<<=	int	2.5
unsigned short	%=	_int	6.9	float	/=	_double	2.5
other-types	<<=	other-types	6.7	int	*=	float	2.4
unsigned char	^=	_int	6.4	unsigned char	=	unsigned char	2.3

For the other operators, each operand shall have arithmetic type consistent with those allowed by the corresponding binary operator. 1311

Commentary

The discussion in the respective C sentences for these corresponding binary operators also apply here.

C++

5.15p7 *In all other cases, E1 shall have arithmetic type.*

Those cases where objects may not have some arithmetic type when appearing as operands to operators (i.e., floating types with the shift operators) are dealt with using the equivalence argument specified earlier in 5.15p7.

Semantics

A *compound assignment* of the form **E1 op = E2** differs from the simple assignment expression **E1 = E1 op (E2)** only in that the lvalue **E1** is evaluated only once. 1312

Commentary

This defines the term *compound assignment*. The original rationale for this form of operator was that it removed the need for translators to spot the commonly occurring case of the value of an object being operated on and stored back into the original object (an optimization that is made by translators for other languages).

There is no requirement that the evaluation order of E1 and E2 in the expression **E1 op= E2** be the same as in the expression **E1 = E1 op E2**. In compiler terminology, E1 is known as a *common subexpression*.

Other Languages

Java specifies a left to right evaluation order and so in the following example the final values assigned to a and b is 9 in both cases. The C behavior in both cases is undefined.

```

1 void f(void)
2 {
3     short a = 6;
4     int   b = 6;
5
6     a += (a=3);    /* Undefined behavior. */
7     b = b + (b=3); /* Undefined behavior. */
8 }
```

In Java compound assignment operators perform a (narrowing primitive) conversion:

Java 15.26.2 *A compound assignment expression of the form **E1 op = E2** is equivalent to **E1 = (T)((E1) op (E2))**, where **T** is the type of **E1**, except that **E1** is evaluated only once.*

This implicit conversions means that the expression **a += b** is conforming, while **a = a + b** generates a translation time diagnostic and an explicit cast (to short) has to be added.

Common Implementations

Because of the availability of these operators many translators make no direct effort to detect the optimizations possible in expressions such as **x=x*y** (some cases may be detected as a consequence of common subexpression detection). For the simple cases there is unlikely to be any differences in the generated machine code. In the more complex cases (e.g., x is an array element or structure member access) the opportunity to reuse the calculated address of an object is self-evident when a compound assignment operator is used and has to be deduced when the equivalent longer form is used.

compound
assignment
semantics

expression
order of evaluation

Coding Guidelines

Compound assignment requires less effort to comprehend than its equivalent two operator form. Readers only need to comprehend two operands, and there is no need to notice that one operand is the same as another. These operators directly represent the concept of performing an operation on the value held in an object and storing it back into that object. Measurements of existing source shows that developers regularly use compound assignment operators. A guideline recommending this usage would serve no worthwhile purpose.

References