

The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones

derek@knosof.co.uk

6.5.5 Multiplicative operators

multiplicative-expression
syntax

multiplicative-expression:

```

cast-expression
multiplicative-expression * cast-expression
multiplicative-expression / cast-expression
multiplicative-expression % cast-expression

```

Commentary

The use of the term *multiplicative* is based on the most commonly occurring of the three operators.

Table ?? lists various results from operating on infinities and NaNs; annex F (of the C Standard) discusses some of the expression optimizing transformations these results may prohibit.

C++

In C++ there are two operators (pointer-to-member operators, `.*` and `->*`) that form an additional precedence level between *cast-expression* and *multiplicative-expression*. The nonterminal name for such expressions is *pm-expression*, which appears in the syntax for *multiplicative-expression*.

Other Languages

Many languages designed before C did not support a remainder operator. Pascal uses **rem** to denote this operator, while Ada uses both **rem** and **mod** (corresponding to the two interpretation of its behavior).

Common Implementations

Multiplicative operators often occur in the bodies of loops with one of their operands being a loop counter. It is sometimes possible to transform the loop, by making use of a regular pattern of loop increments; multiplicative operations can be replaced by additive operations,^[7] while replacing division, remainder, or modulo operations requires the introduction of a nested loop.^[22]

Coding Guidelines

No deviation is listed in the parenthesizing guideline recommendation for the remainder operator because developers are very unlikely to have received any significant practice in its usage (compared to the other two operators, which will have been encountered frequently during their schooling).

Unlike multiplication, people do not usually learn division tables at school. The published studies of multiplicative operators have not been as broad and detailed as those for the additive operators. Many of those that have been performed have investigated mental multiplication, with a few studies of mental division (those that have been performed involved operands that gave an exact integer result, unlike division operations in source code which may not be exact), and your author could find none investigating the remainder operation.

- A study by Parkman^[20] showed subjects a series of simple multiplication problems of the form $p \times q = n$ and asked them to respond as quickly as possible as to whether the value n was correct (both p and q were single-digit values). The time taken for subjects to respond was measured. The results followed the pattern found in an earlier study involving addition; that is, it was possible to fit the results to a number of straight lines (i.e., $RT = ax + b$). In one case x was determined by the value $\min(p, q)$, in the other by $p * q$.
- A study by LeFevre, Bisanz, Daley, Buffone, Greenham, and Sadesky^[15] gave subjects single-digit multiplication problems to solve and then asked them what procedure they had used to solve the problems. In 80% of trials subjects reported retrieving the answer from memory; other common solution techniques included: 6.4% deriving it (e.g., $6 \times 7 \Rightarrow 6 \times 6 + 1$), 4.5% number series (e.g., $3 \times 5 \Rightarrow 5, 10, 15$), and 3.8% repeated addition (e.g., $2 \times 4 \Rightarrow 2 + 2$).
- A study by Campbell^[4] measured the time taken for subjects to multiply numbers between two and nine, and to divide a number (that gave an exact result) by a number between two and nine. The results

expression ??
shall be paren-
thesized

additive-
expression
syntax

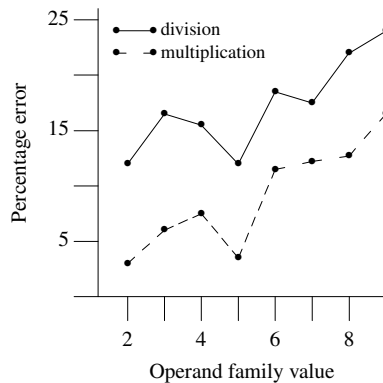


Figure 1143.1: Mean percentage of errors in simple multiplication (e.g., 3×7) and division (e.g., $81/9$) problems as a function of the operand value (see paper for a discussion of the effect of the relative position of the minimum/maximum operand values). Adapted from Campbell.^[4]

showed a number of performance parallels (i.e., plots of their response times and error rates, shown in Figure 1143.1, had a similar shape) between the two operations, although division was more difficult. These parallels suggest that similar internal processes are used to perform both operations.

- A study by LeFevre and Morris^[16] supported the idea that multiplication and division are stored in separate mental representations and that sometimes the solution to difficult division problems was recast as a multiplication problem (e.g., $56/8$ as $8 \times ? = 56$).

Adults (who spent five to six years as children learning their times tables) can recall answers to single digit multiplication questions in under a second, with an error rate of 7.6%.^[5] The types of errors made can be put into the following categories^[12] (see Table 1143.1):

- *Operand errors.* One of the digits in an operand being multiplied is replaced by another digit. For example, $8 \times 8 = 40$ is an operand error because it shares an operand, 8, with $5 \times 8 = 40$.
- *Close operand errors.* This is a subclass of operand errors, with the replaced digit being within ± 2 of the actual digit (e.g., $5 \times 4 = 24$). This behavior is referred to as the *operand distance effect*.
- *Frequent product errors.* The result of the multiplication is given as one of the numbers 12, 16, 18, 24 or 36. These five numbers frequently occur as the result of multiplying operands between 2 and 9.
- *Table errors.* Here the result is a value that is an answer to multiplying two unrelated digits, than those actually given (e.g., $4 \times 5 = 12$).
- *Operation errors.* Here the multiplication operation is replaced by an additional operation (e.g., $4 \times 5 = 9$).
- *Non-table errors.* Here the result is a value that is not an answer to multiplying any two single digits; for instance, $4 \times 3 = 13$, where 13 is not the product of any pair of integers.

Table 1143.1: Percentage breakdown of errors in answers to multiplication problems. Figures are mean values for 60 adults tested on 2×2 to 9×9 from Campbell and Graham,^[5] and 42 adults tested on 0×0 to 9×9 from Harley.^[12] For the Campbell and Graham data, the operand error and operation error percentages are an approximation due to incomplete data.

	Campbell and Graham	Harley
Operand errors	79.1	86.2
Close operand errors	76.8	76.74
Frequent product errors	24.2	23.26
Table errors	13.5	13.8
Operation error	1.7	13.72
Error frequency	7.65	6.3

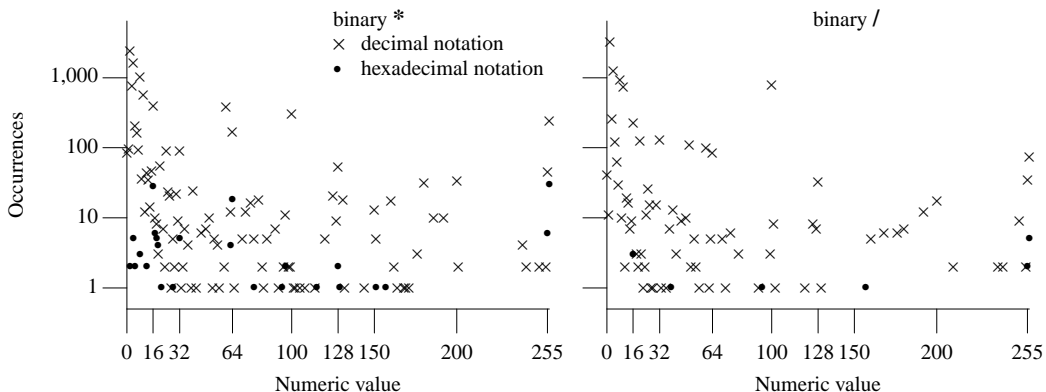


Figure 1143.2: Number of *integer-constants* having a given value appearing as the right operand of the multiplicative operators. Based on the visible form of the .c files.

For a discussion of multiplication of operands containing more than one digit see Dallaway.^[9]

Example

See annex G.5.1 EXAMPLE 1 and EXAMPLE 2 for an implementation of complex multiple and divide functions.

Table 1143.2: Common token pairs involving multiplicative operators (as a percentage of all occurrences of each token). Based on the visible form of the .c files. Note: a consequence of the method used to perform the counts is that occurrences of the sequence *identifier ** are over estimated (e.g., occurrences of a typedef name followed by a * are included in the counts).

Token Sequence	% Occurrence of First Token	% Occurrence of Second Token	Token Sequence	% Occurrence of First Token	% Occurrence of Second Token
identifier *	3.4	92.1	/ sizeof	9.0	3.6
identifier %	0.0	57.7	* identifier	76.5	2.8
identifier /	0.1	54.3	*)	14.4	2.0
) /	0.3	33.9	floating-constant /	5.8	1.8
) %	0.1	31.8	/ integer-constant	53.5	0.5
* floating-constant	0.2	12.5	% integer-constant	44.8	0.1
* sizeof	1.6	11.2	/ identifier	27.5	0.1
integer-constant /	0.1	8.5	floating-constant *	6.8	0.1
, %	0.0	6.5	/ (7.9	0.1
/ floating-constant	2.0	6.4	% identifier	47.6	0.0
* *v	1.4	4.4			

Constraints

Each of the operands shall have arithmetic type.

Commentary

These operators have no common usage for anything other than arithmetic types.

Coding Guidelines

Enumerated types are often thought about in symbolic, not arithmetic, terms. The use of operands having an enumerated type is discussed elsewhere.

A boolean value may be thought about in terms of a zero/nonzero representation. In this case a multiplication operation involving an operand representing a boolean value will return a boolean (in the zero/nonzero sense) result. In this case multiplication is effectively a logical-OR operation. Can a parallel be drawn between such usage and using bitwise operations to perform arithmetic operations (with the aim of using

multiplicative operand type

symbolic name
enumeration set of named constants
boolean role

similar rationales to make guideline recommendations)? Although this might be an interesting question, given the relative rareness of the usage these coding guidelines do not discuss the issue further.

Table 1144.1: Occurrence of multiplicative operators having particular operand types (as a percentage of all occurrences of each operator; an `_` prefix indicates a literal operand). Based on the translated form of this book's benchmark programs.

Left Operand	Operator	Right Operand	%	Left Operand	Operator	Right Operand	%
<code>int</code>	<code>%</code>	<code>_int</code>	40.6	<code>_unsigned long</code>	<code>*</code>	<code>_int</code>	2.8
<code>int</code>	<code>/</code>	<code>_int</code>	25.6	<code>int</code>	<code>/</code>	<code>float</code>	2.7
other-types	<code>*</code>	other-types	18.1	<code>long</code>	<code>/</code>	<code>_int</code>	2.5
other-types	<code>/</code>	other-types	16.2	<code>_unsigned long</code>	<code>%</code>	<code>int</code>	2.3
<code>_int</code>	<code>*</code>	<code>_int</code>	13.4	<code>_int</code>	<code>*</code>	<code>unsigned short</code>	2.2
<code>unsigned int</code>	<code>%</code>	<code>_int</code>	12.6	<code>_int</code>	<code>*</code>	<code>_unsigned long</code>	2.2
<code>int</code>	<code>%</code>	<code>int</code>	12.2	<code>_unsigned long</code>	<code>*</code>	<code>int</code>	2.1
<code>int</code>	<code>*</code>	<code>_int</code>	12.1	<code>unsigned long</code>	<code>*</code>	<code>_unsigned long</code>	1.9
<code>_int</code>	<code>/</code>	<code>_int</code>	11.0	<code>int</code>	<code>%</code>	<code>unsigned int</code>	1.8
<code>_unsigned long</code>	<code>/</code>	<code>_unsigned long</code>	9.9	<code>float</code>	<code>/</code>	<code>float</code>	1.8
<code>_unsigned long</code>	<code>*</code>	<code>unsigned char</code>	9.5	<code>_unsigned long</code>	<code>/</code>	<code>_int</code>	1.6
<code>int</code>	<code>*</code>	<code>_unsigned long</code>	8.8	<code>unsigned int</code>	<code>%</code>	<code>int</code>	1.6
<code>float</code>	<code>*</code>	<code>float</code>	8.8	<code>unsigned long</code>	<code>%</code>	<code>unsigned long</code>	1.5
other-types	<code>%</code>	other-types	7.3	<code>unsigned short</code>	<code>/</code>	<code>_int</code>	1.3
<code>unsigned long</code>	<code>/</code>	<code>_int</code>	6.6	<code>unsigned long</code>	<code>/</code>	<code>unsigned long</code>	1.3
<code>_int</code>	<code>*</code>	<code>int</code>	6.5	<code>unsigned int</code>	<code>*</code>	<code>_int</code>	1.3
<code>int</code>	<code>*</code>	<code>int</code>	5.9	<code>unsigned int</code>	<code>*</code>	<code>_unsigned long</code>	1.2
<code>unsigned long</code>	<code>/</code>	<code>_unsigned long</code>	5.8	<code>int</code>	<code>/</code>	<code>_unsigned long</code>	1.2
<code>unsigned int</code>	<code>/</code>	<code>_int</code>	5.3	<code>_double</code>	<code>/</code>	<code>_double</code>	1.2
<code>int</code>	<code>/</code>	<code>int</code>	5.0	<code>float</code>	<code>*</code>	<code>_int</code>	1.1
<code>unsigned int</code>	<code>%</code>	<code>unsigned int</code>	4.2	<code>unsigned long</code>	<code>*</code>	<code>_int</code>	1.0
<code>int</code>	<code>%</code>	<code>unsigned long</code>	4.2	<code>unsigned int</code>	<code>%</code>	<code>unsigned long</code>	1.0
<code>int</code>	<code>%</code>	<code>_unsigned long</code>	3.9	<code>int</code>	<code>/</code>	<code>unsigned long</code>	1.0
<code>long</code>	<code>%</code>	<code>_int</code>	3.7	<code>_int</code>	<code>*</code>	<code>unsigned int</code>	1.0
<code>unsigned long</code>	<code>%</code>	<code>_int</code>	3.1				

1145 The operands of the `%` operator shall have integer type.

Commentary

The modulus operator could have been defined to include arithmetic types. However, processors rarely include instructions for performing this operation using floating-point operands.

Semantics

1146 The usual arithmetic conversions are performed on the operands.

Commentary

These conversions may result in the integer promotions being performed.

Common Implementations

All three operators are sufficiently expensive to implement^[24] (both in terms of transistors needed by the hardware and time taken to execute the instruction) that some processors contain instructions where one or more operands has a type narrower than `int`. A translator can make use of such instructions, invoking the as-if rule, when the operands have the appropriate bit width; for instance, multiplying two 8-bit operands to yield a 16-bit result.

A number of processors do not support integer multiply and/or divide instructions in hardware (e.g., early high-performance processors CDC 6x00 and 7x00, Cray; most low-cost processors; RISC processors seeking to simplify their design— HP PA-RISC 1.0, 1.1, 2.0,^[13] HP—was DEC— Alpha;^[23] Sun SPARC, pre-version 8^[26]).

arithmetic
conversions
integer promotions

A recently proposed hardware acceleration technique^[6] is to store the results of previous multiplication and division operations in a cache, reusing rather than recalculating the result whenever possible. (Dynamic profiling has found that a high percentage of these operations share the same operands as previous operations.)

The result of the binary `*` operator is the product of the operands.

1147

Commentary

Source code can contain implicit multiplications as well as explicit ones; for instance, array indexing. If imaginary types are supported, the expression `Infinity*I` is simply a way of obtaining an infinite imaginary value.

Common Implementations

Multiplication is sufficiently common and generally executes sufficiently slowly that implementations often put a lot of effort into alternative implementation strategies. A common special case is multiplying by a known (at translation time) constant.^[17,25]

A multiply instruction can be expressed as a combination of one or more add, subtract, or left shift (by any amount) instructions. For instance, multiplying by 9 is equivalent to shifting a value left 3 bits and adding in the original value, the later probably executing in a few machine cycles. A general expression giving the minimum number of add/subtract/shift instructions required for any constant is not known, but the following are some upper bounds:

- multiplication by an n -bit constant can always be performed using at most n instructions,^[25]
- if the constant n contains many 0-bits, an algorithm that generates $4g(n) + 2s(n) - 1 - \theta$ instructions, where $g(n)$ is the number of groups of two or more consecutive 1-bits in the constant n , $s(n)$ is the number of single 1-bits, and θ is 1 if n ends in a 1-bit and 0 otherwise, may produce a shorter sequence,
- if a fused shift-add instruction is available (e.g., HP PA-RISC^[13]) six or less of these instructions are required to multiply by any constant between 0 and 10,000.^[17]

Approximately half of the hardware needed to perform a IEEE compliant floating-point multiply is only there to guarantee a correctly rounded result. Significant savings in power consumption and execution delays can be achieved by not providing this guarantee^[21] and a number of vendors^[10,14,18] support multiplication instructions that do not guarantee this behavior (e.g., they always truncate).

Usage

Measurements by Citron, Feitelson, and Rudolph^[6] found that in a high percentage of cases the operands of multiplication operations repeat themselves (59% for integer operands and 43% for floating-point). Measurements were based on maintaining previous results in a 32-entry, 4-way associative, cache.

The result of the `/` operator is the quotient from the division of the first operand by the second;

1148

Commentary

The identity $1/\infty \Rightarrow 0$ is consistent with the possibility that the infinity was the result of a division by zero.

Common Implementations

Because of its iterative nature, a divide operation is expensive in terms of the number of transistors required to implement it (in hardware) and its execution time. Divide instructions are often the slowest operation (involving integer operands) on a processor (it can take up to 60 cycles on the HP—was DEC—Alpha 21064 and 46 cycles on the Intel Pentium). Internally it is often implemented as a sequence of steps, computing the quotient digit by digit using some recursive formula; combinatorial implementations replicate the hardware that performs the division step (up to n times, where n is the number of bits in the significand). Because it is an infrequently used operation processor designers often trade significant amounts of chip resources (transistors) against performance (greater performance requires greater numbers of transistors). Many

binary *
resultarray
n-dimensional
reference
pointer
arithmetic
addition resultmultiply
always truncate
IEC 60559
correctly
rounded
resultbinary /
result

low-cost processors do not contain a divide instruction. For different reasons most of the early versions of the commercial RISC processors did not include a divide instruction. What was provided were one or more simpler instructions that could be used to create a sequence of code capable of performing a division operation. The length of the sequence is usually proportional to the number of bits in the narrowest operand.

Dynamic profiling information of hydro2d (from the SPEC92)^[3] found that 64% of the executed divide instructions had either a 0 for its numerator or a 1 for its denominator. Using value profiling techniques, they were able to reduce the execution time of hydro2d by 15%, running on a HP—was DEC— Alpha 21064 processor; and were able to reduce the number of cycles executed by an Intel Pentium running some games' programs by an estimated 5%.

value profiling

An alternative to performing a divide is to take the reciprocal and multiply. This technique was used by Cray for floating-point operands in the hardware implementation of some of their processors.^[8] The Diab Data compiler^[11] -Xieee754-pedantic option enables developers to specify that the convert divide-to-multiply optimization should not be attempted. At least one processor vendor has proposed this optimization for integer operations.^[1]

A common optimization is to replace division by a constant divisor that is a power of 2 by the appropriate right shift, when the numerator value is known to be positive. A less well known optimization^[25] enables any division by a constant to be replaced by a multiply. The basic idea is to multiply by a kind of reciprocal of the divisor (this reciprocal value, on a 32-bit processor, has a value close to $2^{32}/d$) and to extract the most significant 32 bits of the product.

Magenheimer, Peters, Pettis, and Zuras^[17] describe an algorithm for obtaining sequences of shifts and adds that are equivalent to division by a constant.

Coding Guidelines

Many developers are aware of the relatively poor performance of the divide operator and sometimes transform the expression to use another operator. The issue of a right-shift operator being used to divide by a power of two is discussed elsewhere, as is the issue of using a bitwise operator to perform an arithmetic operation. A divide by a floating constant might be rewritten as a multiply by its reciprocal $x/5.0$ becoming $x*0.2$. However, such a transformation does not yield numerically equivalent expressions unless the constants have an exact representation (in the floating-point representation used by the implementation executing the program).

right-shift result bitwise operators

Testing code containing a division operator can require checking a number of special cases. It is possible for the left operand to be incorrect and still obtain the correct answer most of the time. For instance, if $(x+1)/101$ had been written instead of $(x+2)/101$, the result would be correct for 99% of the values of x . A white-box testing strategy would use test cases that provide left operand values just less than and just greater than the value of the right operand, however, testing is not within the scope of these coding guidelines.

Usage

Measurements by Oberman^[19] found that in a high percentage of cases division operations on floating-point operands repeat themselves (i.e., the same numerator and denominator values). The measurements were done using the SPECfp92 and NAS (Fortran-based) benchmarks.^[2] Simulations using an infinite division operand cache found a hit rate (i.e., cache lookup could return the result of the division) of 69.8%, while a cache containing 128 entries had a hit rate of 60.9%. A more detailed analysis by Citron, Feitelson, and Rudolph^[6] found a great deal of variability over different applications, with multimedia applications having hit rates of 50% (using a 32 entry, 4-way associative cache).

SPEC benchmarks

1149 the result of the % operator is the remainder.

% operator result

Commentary

The character % is one of the few characters not already used as an operator in other programming languages that appears on most keyboards. It also is also visually similar to the character used to represent the divide operator token.

Common Implementations

In some processors the division instruction also calculates the remainder, putting it in a separate register. While a few processors have instructions that only perform the remainder operation, many more do not support this operation in hardware.

binary / 1148
result The reciprocal-multiply algorithm listed as a possible replacement for division by a constant can also be used to calculate the remainder when the right operand is constant.

Coding Guidelines

binary / 1148
result Like the divide operator, the performance of the remainder operator is generally poor and there are a number of well-known special cases that can be optimized. For instance, when the first operand has an unsigned type and the value of the second operand is a power of two, a bitwise-AND operation can zero out the top bits. Replacing an arithmetic operator by equivalent bitwise operations is discussed elsewhere.

bitwise operators The result of this operator, when the left operand is positive, will range from zero to one less than the value of the right operand. Sometimes a value that varies between one and the value of the right operand is required. This can lead to off-by-one mistakes like those commonly seen in array subscripting.

In both operations, if the value of the second operand is zero, the behavior is undefined.

1150

Commentary

The usual mathematical convention (for divide) is to say that the result is infinity. The C integer types do not support a representation for infinity; however, the IEC 60559 representation does. The IEC 60559 Standard defines three possible results, depending on the value of the left operand: $+\infty$, $-\infty$, and NaN.

IEC 60559 The behavior can also depend on the setting of the FENV_ACCESS pragma.

Other Languages

Some languages regard a second operand of zero for this operator as being more serious than undefined behavior (e.g., Ada requires an exception to be raised).

Common Implementations

Many processors raise an exception if the second operand is zero and has an integer type. This exception may be catchable within a program via a signal handler.

Coding Guidelines

guidelines
not faults Having a zero value for the second operand rarely makes any sense, algorithmically. It is unusual for a developer to intend it to occur. These coding guidelines are not intended to recommend against the use of constructs that are obviously faults.

When integers are divided, the result of the / operator is the algebraic quotient with any fractional part discarded.⁸⁸⁾

1151

Commentary

C99 reflects almost universal processor behavior (as does the Fortran Standard). This definition truncates toward zero and the expression $-(a/b) == (-a)/b$ && $-(a/b) == a/(-b)$ is always true. It also means that the absolute value of the result does not depend on the signs of the operands; for example:

```
+10 / +3 == +3    +10 % +3 == +1    -10 / +3 == -3    -10 % +3 == -1
+10 / -3 == -3    +10 % -3 == +1    -10 / -3 == +3    -10 % -3 == -1
```

C90

When integers are divided and the division is inexact, if both operands are positive the result of the / operator is the largest integer less than the algebraic quotient and the result of the % operator is positive. If either operand is negative, whether the result of the / operator is the largest integer less than or equal to the algebraic quotient or the smallest integer greater than or equal to the algebraic quotient is implementation-defined, as is the sign of the result of the % operator.

If either operand is negative, the behavior may differ between C90 and C99, depending on the implementation-defined behavior of the C90 implementation.

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      int x = -1,
6          y = +3;
7
8      if ((x%y > 0) ||
9          ((x+y)%y == x%y))
10         printf("This is a C90 translator behaving differently than C99\n");
11  }
```

Quoting from the C9X Revision Proposal, WG14/N613, that proposed this change:

The origin of this practice seems to have been a desire to map C's division directly to the "natural" behavior of the target instruction set, whatever it may be, without requiring extra code overhead that might be necessary to check for special cases and enforce a particular behavior. However, the argument that Fortran programmers are unpleasantly surprised by this aspect of C and that there would be negligible impact on code efficiency was accepted by WG14, who agreed to require Fortran-like behavior in C99.

WG14/N613

C++

If both operands are nonnegative then the remainder is nonnegative; if not, the sign of the remainder is implementation-defined⁷⁴.

5.6p4

Footnote 74 describes what it calls *the preferred algorithm* and points out that this algorithm follows the rules by the Fortran Standard and that C99 is also moving in that direction (work on C99 had not been completed by the time the C++ Standard was published).

The C++ Standard does not list any options for the implementation-defined behavior. The most likely behaviors are those described by the C90 Standard (see C90/C99 difference above).

Other Languages

Ada supports two remainder operators (**rem** and **mod**) corresponding to the two interpretations of the remainder operator (round toward or away from zero when one of the operands is negative).

Common Implementations

All C90 implementations known to your author follow the C99 semantics.

Coding Guidelines

While experience shows that developers do experience difficulties in comprehending code where one or more of the operands of the / operator has a negative value, there are no obvious guideline recommendations.

Example

```
1  int residua_modulo(int x, int y) /* Assumes y != 0 */
2  {
3  if (x >= 0)
4      return x%y;
5  else
6      return x%y + ((y > 0) ? +y : -y);
7  }
```

If the quotient a/b is representable, the expression $(a/b)*b + a\%b$ shall equal a .

1152

Commentary

This requirement was in C90 and only needed to be this complex because there were two possibilities for the result of the division operator in C90.

References

1. R. Alverson. Integer division using reciprocals. In P. Kornerup and D. W. Matula, editors, *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, pages 186–190, Grenoble, France, 1991. IEEE Computer Society Press, Los Alamitos, CA.
2. D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, Dec. 1995.
3. B. Calder, P. Feller, and A. Eustace. Value profiling. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-97)*, pages 259–269, Los Alamitos, Dec. 1–3 1997. IEEE Computer Society.
4. J. I. D. Campbell. On the relation between skilled performance of simple division and multiplication. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 23(5):1140–1159, 1997.
5. J. I. D. Campbell and D. J. Graham. Mental multiplication skill: Structure, process, and acquisition. *Canadian Journal of Psychology*, 39(2):338–366, 1985.
6. D. Citron, D. Feitelson, and L. Rudolph. Accelerating multi-media processing by implementing memoing in multiplication and division units. In *Proceedings of 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pages 252–261, 1998.
7. K. D. Cooper, L. T. Simpson, and C. A. Vick. Operator strength reduction. *ACM Transactions on Programming Languages and Systems*, 23(5):603–625, Sept. 2001.
8. Cray. *Cray-1 Computer Systems: Cray-1 S Series Hardware Reference Manual*. Cray Research, Inc, Nov. 1981.
9. R. Dallaway. Dynamics of arithmetic connectionist view of arithmetic skills. Technical Report CSR 306, University of Sussex, 1994.
10. A. M. Devices. *3DNow! Technology Manual*. Advanced Micro Devices, Inc, 2000.
11. Diab Data. *D-CC & D-C++ Compiler Suites User's Guide*. Diab Data, Inc, www.ddi.com, 4.3 edition, June 1999.
12. W. S. Harley. *Associative Memory in Mental Arithmetic*. PhD thesis, Johns Hopkins University, Oct. 1991.
13. Hewlett-Packard. *PA-RISC 2.0*. Hewlett-Packard, 2.0 edition, 1995.
14. Intel. *Data Alignment and Programming Issues for the Streaming SIMD Extensions with the Intel C/C++ Compiler*. Intel Corporation, 1.1 edition, Jan. 1999.
15. J.-A. LeFevre, J. Bisanz, K. E. Daley, L. Buffone, S. L. Greenham, and G. S. Sadesky. Multiple routes to solution of single-digit multiplication problems. *Journal of Experimental Psychology: General*, 125(3):284–306, 1996.
16. J.-A. LeFevre and J. Morris. More on the relation between division and multiplication in simple arithmetic: Evidence for mediation of division solutions via multiplication. *Memory & Cognition*, 27(5):803–812, 1999.
17. D. J. Magenheimer, L. Peters, K. W. Pettis, and D. Zuras. Integer multiplication and division on the HP precision architecture. *IEEE Transactions on Computers*, 37(8):980–990, 1988.
18. Motorola, Inc. *AltiVec Technology Programming Interface Manual*. Motorola, Inc, 1999.
19. S. F. Oberman. Design issues in high performance floating point arithmetic units. Technical Report CSL-TR-96-711, Stanford University, Dec. 1996.
20. J. M. Parkman. Temporal aspects of simple multiplication and comparison. *Journal of Experimental Psychology*, 95(2):437–444, 1972.
21. M. J. Schulte, K. E. Wires, and J. E. Stine. Variable-correction truncated floating point multipliers. In *Conference Record of the Thirty-Fourth Asilomar Conference on Signals, Systems and Computers*, pages 1344–1348. IEEE, 2000.
22. J. W. Sheldon. Strength reduction of integer division and modulo operations. Thesis (m.s.), MIT, May 2001.
23. R. L. Sites and R. L. Witek. *Alpha AXP architecture reference manual*. Digital Press, 12 Crosby Drive, Bedford, MA 01730, USA, second edition, 1995.
24. P. Soderquist and M. Leeser. Area and performance tradeoffs in floating-point divide and square root implementations. *Computing Surveys*, 28(3):518–564, Sept. 1996.
25. J. H. S. Warren. *Hacker's Delight*. Addison–Wesley, 4th edition, 2003.
26. D. L. Weaver and T. Germond. *The SPARC Architecture Manual*. Prentice Hall, Inc, ninth edition, 2000.