

The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones

derek@knosof.co.uk

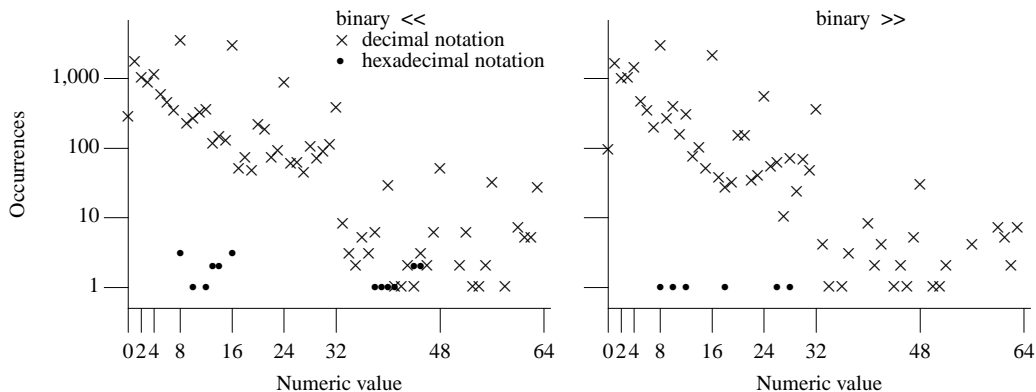


Figure 1181.1: Number of *integer-constants* having a given value appearing as the right operand of the shift operators. Based on the visible form of the `.c` files.

6.5.7 Bitwise shift operators

shift-expression
syntax

shift-expression:

additive-expression

shift-expression << *additive-expression*

shift-expression >> *additive-expression*

Commentary

Bit shift instructions are ubiquitous in processor instruction sets. These operators allow writers of C source to access them directly. Many processors also contain bit rotate instructions; however, the definition of these instructions varies (in some cases the rotate includes a bit in the status flags register), and they are not commonly called for in algorithms.

Other Languages

Most languages do not get involved in providing such low level operations, just because such instructions are available on most processors. Java also defines the >>> operator.

Table 1181.1: Common token pairs involving the shift operators (as a percentage of all occurrences of each token). Based on the visible form of the `.c` files.

Token Sequence	% Occurrence of First Token	% Occurrence of Second Token	Token Sequence	% Occurrence of First Token	% Occurrence of Second Token
identifier >>	0.1	63.9] <<	0.5	5.3
identifier <<	0.1	37.3	<< <i>integer-constant</i>	63.4	0.8
<i>integer-constant</i> <<	0.5	36.1	>> <i>integer-constant</i>	79.8	0.7
) >>	0.2	28.0	<< identifier	28.4	0.1
) <<	0.2	20.3	<< (8.1	0.1
] >>	0.4	6.2	>> identifier	15.9	0.0

Constraints

Each of the operands shall have integer type.

Commentary

This constraint reflects the fact that processors rarely contain instructions for shifting non-integer types (e.g., floating-point types), which in turn reflects the fact that there is no commonly defined semantics for shifting other types.

1181

1182

Table 1182.1: Occurrence of shift 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>	29.4	<code>unsigned char</code>	<code><<</code>	<code>_int</code>	2.8
<code>_int</code>	<code><<</code>	<code>_int</code>	27.1	<code>_long</code>	<code><<</code>	<code>_long</code>	2.8
<code>unsigned int</code>	<code>>></code>	<code>_int</code>	26.1	<code>unsigned int</code>	<code>>></code>	<code>int</code>	2.6
<code>_long</code>	<code><<</code>	<code>_int</code>	11.9	<code>_int</code>	<code>>></code>	<code>_int</code>	2.5
<code>int</code>	<code><<</code>	<code>_int</code>	11.8	<code>int</code>	<code>>></code>	<code>int</code>	2.1
<code>unsigned long</code>	<code>>></code>	<code>_int</code>	11.3	<code>long</code>	<code>>></code>	<code>_int</code>	2.0
<code>_int</code>	<code><<</code>	<code>int</code>	7.3	<code>unsigned long</code>	<code>>></code>	<code>int</code>	1.8
<code>unsigned short</code>	<code>>></code>	<code>_int</code>	7.0	<code>unsigned long</code>	<code><<</code>	<code>_int</code>	1.8
other-types	<code>>></code>	other-types	6.9	<code>long</code>	<code>>></code>	<code>int</code>	1.7
<code>int</code>	<code><<</code>	<code>int</code>	6.0	<code>_unsigned long</code>	<code><<</code>	<code>int</code>	1.3
other-types	<code><<</code>	other-types	5.8	<code>unsigned int</code>	<code>>></code>	<code>unsigned int</code>	1.2
<code>unsigned int</code>	<code><<</code>	<code>int</code>	5.3	<code>signed long</code>	<code>>></code>	<code>_int</code>	1.2
<code>_unsigned long</code>	<code><<</code>	<code>_int</code>	4.9	<code>unsigned short</code>	<code><<</code>	<code>_int</code>	1.1
<code>unsigned int</code>	<code><<</code>	<code>_int</code>	4.2	<code>long</code>	<code><<</code>	<code>_int</code>	1.1
<code>unsigned char</code>	<code>>></code>	<code>_int</code>	4.0	<code>int</code>	<code><<</code>	<code>unsigned long</code>	1.1
<code>unsigned long</code>	<code><<</code>	<code>int</code>	3.8				

Semantics

1183 The integer promotions are performed on each of the operands.

Commentary

This is the only implicit conversion performed on the operands.

Common Implementations

The shift instructions on some processors can operate on operands of various widths, provided both widths are the same (e.g., those on the Intel x86 processor family can operate on either 8, 16, or 32 bit operands). However, the result returned by these instructions usually has the same width as the operand. For instance, shifting an operand having type `unsigned char` and value `0xf0` left by two bits returns `0xc0` rather than `0x3c0`. If the next operation is an assignment to an object having the same type as the operand type, an optimizer might choose to make use of one of these narrower-width instructions; otherwise, the width corresponding to the promoted type has to be used. The width of the left operand will be the same as the object assigned to when the compound assignment form of these operators is used.

Coding Guidelines

The type of the left operand needs to be taken into account when thinking about the result of a shift operation. However, some developers have a mental model that does not include the integer promotions. The effect of the integer promotions on an object having type `signed char` or `short` (when `sizeof(short) != sizeof(int)`), and a negative value, is to increase the number nonzero bits in the value representation. The issue of right-shifting negative values is discussed elsewhere. Using shift operators to perform arithmetic operations is making use of representation information; this is covered by a guideline recommendation. In this case unsigned types need to be used and a deviation is provided to cover this situation.

1184 The type of the result is that of the promoted left operand.

Commentary

The usual arithmetic conversions are not performed on the operands. The right operand does not affect the type of the left operand.

Common Implementations

In some prestandard implementations the usual arithmetic conversions were performed on both operands, which meant the result type might not have been that of the promoted left operand.

shift operator
integer pro-
motions

assignment-
expression
syntax

1196 right-shift
negative value
?? represen-
tation in-
formation
using
?? object
int type only

Example

```

1  extern unsigned long glob;
2
3  void f(void)
4  {
5  unsigned int loc;
6
7  /*
8  * The result of the << operation is int (bits may be shifted off the
9  * end, which might not occur if loc were promoted to unsigned long),
10 * which is then promoted to type unsigned long.
11 */
12 loc = glob % (loc << glob);
13 }

```

If the value of the right operand is negative or is greater than or equal to the width of the promoted left operand, the behavior is undefined. 1185

Commentary

This specification takes into account the variation in behavior of processor shift instructions when asked to shift by a negative amount. For instance, some processors require the shift amount to be loaded into a special register, which is only capable of holding a limited range of values. Loading a value greater than the width of the operand into one of these registers may result in a shift amount of zero, while loading a negative value may result in the largest shift amount.

Other Languages

Java ensures that the value of the right operand is always in range by specifying that the equivalent of a bitwise-AND is performed on it (the number of bits extracted depending on the type of the left operand). Algol is unusual in that it defines shifting by a negative amount as reversing the direction of the shift operator (e.g., $x \ll -2$ shifts x right by 2).

Common Implementations

The behavior may be decided by the translator or by the characteristics of the processor that executes the machine code generated to perform the operation. For instance, many translators will fold $(1 \ll 32)$ to 0, while generating machine code to evaluate $(y \ll 32)$. The Intel Pentium^[1] SAL instruction (generated by both gcc and Microsoft C++ to evaluate left-shifts) only uses the bottom five bits of the shift amount (leading to y being shifted by zero bits in the example below).

```

1  #include <stdio.h>
2
3  int y = 1;
4
5  int main(void)
6  {
7  if ((1 << 32) != (y << 32))
8  printf("1 != %d\n", y);
9  }

```

Coding Guidelines

Developers sometimes incorrectly assume that a very large shift value will generate a result of all bits zero or all bits one (i.e., when right-shifting negative values is implemented arithmetically). Shifting by a negative amount has no common meaning associated with it (such usage is invariably a fault and nothing is served by having a guideline recommending against faults).

-
- 1186 89) Another way to approach pointer arithmetic is first to convert the pointer(s) to character pointer(s): In this scheme the integer expression added to or subtracted from the converted pointer is first multiplied by the size of the object originally pointed to, and the resulting pointer is converted back to the original type.

Commentary

This describes common implementation practice.

-
- 1187 For pointer subtraction, the result of the difference between the character pointers is similarly divided by the size of the object originally pointed to.

Commentary

This describes common implementation practice.

-
- 1188 When viewed in this way, an implementation need only provide one extra byte (which may overlap another object in the program) just after the end of the object in order to satisfy the “one past the last element” requirements.

Commentary

A *one past the last element* value needs to point at something. Because an implementation is not required to support the dereferencing of any pointers to this location, it need only consist of a single byte.

one past
the end
accessing

Common Implementations

Most implementations do not allocate storage for this *one past the last element* location. In many cases such an address is the first byte of the next object defined in the source. Using the address of another object as the value of the *one past the last element* location makes it difficult for execution-time verification of pointer usage. Such a checking implementation would either have to leave some extra storage at the end of every object or use an alternative representation. The Model Implementation C Checker^[2] uses a special region of storage to represent this concept. It contains information on which object is being pointed one past of, and the execution-time system knows that all pointers to this area of storage are pointing one past the end of some object.

-
- 1189 The result of $E1 \ll E2$ is $E1$ left-shifted $E2$ bit positions;

Commentary

A left-shift moves bits from less significant positions to more significant positions within the representation. Once bits are shifted passed the most significant bit, they cease to be part of the value representation. Unlike the arithmetic operators, it is accepted that bits may be *lost* during a shift operation.

Common Implementations

Many processors contain instructions for shifting by a constant amount (the value is embedded in the instruction). Others require that the amount to shift by be loaded into a register. Some offer both forms (e.g., Intel x86). Some processors implement this instruction such that it performs the complete shift in one cycle (a barrel shifter). Other processors may take a cycle-per-bit position shifted.

Coding Guidelines

Left-shifting is commonly known to be equivalent, in a binary representation, to multiplying a positive value by a power of two. The issue of replacing arithmetic operations by bitwise operations is covered by the guideline recommendation dealing with the use of representation information.

?? represen-
tation in-
formation
using

-
- 1190 vacated bits are filled with zeros.

Commentary

The vacated bits occur in the least significant bit, as the operand is shifted left.

left-shift
of positive value

If **E1** has an unsigned type, the value of the result is $E1 \times 2^{E2}$, reduced modulo one more than the maximum value representable in the result type. 1191

value rep-
resentation**Commentary**

This is one of the consequences of requiring that all bits in the value representation participate in forming a value. There are no more bits visible to a shift operator shift than are visible from an arithmetic operator.

Common Implementations

This C sentence is a mathematical description of the behavior. In practice processor implementations have circuitry that shifts bits rather than multiplying them.

positive
signed in-
teger type
subrange of
equivalent
unsigned type

If **E1** has a signed type and nonnegative value, and $E1 \times 2^{E2}$ is representable in the result type, then that is the resulting value; 1192

Commentary

A positive value, having a signed type, has the same representation as the corresponding unsigned type.

C90

This specification of behavior is new in C99; however, it is the behavior that all known C90 implementations exhibit.

C++

Like the C90 Standard, the C++ Standard says nothing about this case.

Other Languages

Java defines this to be the behavior of the left-shift operator for all values (including negative ones) of **E1**.

left-shift
undefined

otherwise, the behavior is undefined. 1193

Commentary

A negative value incorporates a sign bit. There is no consistent behavior across all processors and, given the desire to efficiently implement the C Standard on a wide range of processors, the Committee was not able to agree on a behavior.

C90

This undefined behavior was not explicitly specified in the C90 Standard.

C++

Like the C90 Standard, the C++ Standard says nothing about this case.

Common Implementations

In practice the defined behaviors are invariably one of the following:

- sign bit is treated just like the other bits (e.g., it is shifted).
- sign bit is ignored and it remains unchanged by the shift instruction (e.g., the Unisys A Series^[4]).
- sign bit is *sticky* (i.e., as soon as a 1 is shifted through it, its value stays set at 1).

Coding Guidelinesobject ??
int type only

These guidelines recommend the use of a single integer type, which is signed. This undefined behavior, for signed types, means that developers wanting to portably manipulate a scalar type as a sequence of bits are going to have to declare an object to have an unsigned type.

Dev ??

An object whose value is always treated as a sequence of bits, rather than an arithmetic value, may be declared to have an unsigned type.

1194 The result of `E1 >> E2` is `E1` right-shifted `E2` bit positions.

right-shift
result**Commentary**

A right-shift moves bits from most significant positions to less significant positions within the representation. Once bits are shifted passed the least significant bit, they cease to be part of the value representation. Unlike the arithmetic operators, it is accepted that bits may be *lost* during a shift operation. On processors that use either one's complement or sign and magnitude representation for the integer types, the right-shift operator is always equivalent to a divide by the appropriate power of 2. Steele^[3] gives examples of how often right-shift and division by powers of 2 are incorrectly treated as being equivalent when integer types are represented using two's complement.

Coding Guidelines

Shifting right is commonly known to be equivalent, in a binary representation, to dividing a positive value by powers of 2. The issue of replacing arithmetic operations by bitwise operations is covered by the guideline recommendation dealing with the use of representation information.

?? represen-
tation in-
formation
using

1195 If `E1` has an unsigned type or if `E1` has a signed type and a nonnegative value, the value of the result is the integral part of the quotient of `E1 / 2E2`.

Commentary

The issues are the same as for left-shift.

1191 left-shift
of positive value**Common Implementations**

This C sentence is a mathematical description of the behavior. In practice processor implementations have circuitry that shifts bits rather than dividing them.

1196 If `E1` has a signed type and a negative value, the resulting value is implementation-defined.

right-shift
negative value**Commentary**

Two different forms of right-shift instruction are invariably implemented by processors. One form treats the sign bit like the other value bits and vacated bits are filled with zeros (sometimes known as a logical right-shift). The other form fills vacated bits with the value of the sign bit (sometimes known as an arithmetic right-shift). Recognizing that it is possible for implementations to predict the behavior in this case, the C Committee specified the behavior as implementation-defined.

Other Languages

Java defines the `>>` operator to use sign extension and the `>>>` operator to use zero extension.

Common Implementations

Many processors have instructions that perform either form of right-shift. In such cases the decision on which to generate is made by the translator. Vendors often provide an option to select between arithmetic and logical right-shift.

Coding Guidelines

This issue is addressed by a deviation.

?? object
int type only

References

1. Intel. *IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture*. Intel, Inc, 2000.
2. D. M. Jones. *The Model C Implementation*. Knowledge Software Ltd, 1992.
3. G. L. Steele Jr. Arithmetic shifting considered harmful. Technical Report A.I. Memo No. 378, M.I.T., Sept. 1976.
4. Unisys Corporation. *C Programming Reference Manual, Volume 1: Basic Implementation*. Unisys Corporation, 8600 2268-203 edition, 1998.