The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

Copyright ©2002-2008 Derek M. Jones. All rights reserved.

6.5.10 Bitwise AND operator

AND-expression syntax bitwise &

1234

AND-expression:
equality-expression
AND-expression & equality-expression

Commentary

Dennis Ritchie

From decvax!harpo!npoiv!alice!research!dmr Fri Oct 22 01:04:10 1982 Subject: Operator precedence Newsgroups: net.lang.c

The priorities of && || vs. == etc. came about in the following way.

Early C had no separate operators for & and && or | and ||. (Got that?) Instead it used the notion (inherited from B and BCPL) of "truth-value context": where a Boolean value was expected, after "if" and "while" and so forth, the & and | operators were interpreted as && and || are now; in ordinary expressions, the bitwise interpretations were used. It worked out pretty well, but was hard to explain. (There was the notion of "top-level operators" in a truth-value context.)

The precedence of & and | were as they are now.

Primarily at the urging of Alan Snyder, the && and || operators were added. This successfully separated the concepts of bitwise operations and short-circuit Boolean evaluation. However, I had cold feet about the precedence problems. For example, there were lots of programs with things like

if (a==b & c==d) . . .

In retrospect it would have been better to go ahead and change the precedence of & to higher than ==, but it seemed safer just to split & and && without moving & past an existing operator. (After all, we had several hundred kilobytes of source code, and maybe 3 installations. . . .)

Other Languages

Languages that support bitwise operators tend to either use the same operator as C, or the keyword **and**. Support for an unsigned integer type was added in Ada 95 and the definition of the logical operators was extended to perform bitwise operations when their operands had this type.

Coding Guidelines

logical-AND-expression syntax

The issue of rearranging expressions, involving bitwise operators, to reduce the number of operators they contain is discussed elsewhere.

controlling expression if statement

Experience shows that developers sometimes confuse the two operators **&** and **&&** with each other, and sometimes they intentionally swap the use of these operators. The confusion may be caused by their similar visual appearance, a temporary lapse of concentration, or some other reason. They may return different results for the same pair of operands, as the following example shows:

```
0x10 & 0x01  ⇒ 0
0x10 && 0x01 ⇒ 1
```

The two operators return the same numeric result when the evaluation of the second operand generates no side effects, and

boolean role

- the value of either operand is 0,
- the values of both operands is restricted to the range 0 or 1 (a boolean role represented as one of two values), and
- the least significant bit of both operands is set and the operands have no other set bits in common.

When there is no order dependency between the operands (i.e., the evaluation of the second operand is not conditional on the evaluation of the first), developers sometimes make use of one of these equivalencies to select what they consider to be the *most efficient* operator. For instance, if the one of the operands is a function call, **&&** might be used with the call as its right operand. Alternatively, if both operands are simple object accesses, the **&** operator might be used in the belief that unconditionally evaluating both operands is faster than a conditional evaluation (because of the conditional jump).

There are two checks that might be used to attempt to confirm that the operator appearing in the source is the one intended:

1. *The roles of the operands.* If both operands have boolean roles, the operators **&** and **&&** both return the same result and the author is free to select which operator to use. Operands having a bit-set role might be expected to be operands of the **&** operator, while those having a numeric role appear as operands of the **&&** operator.
2. *The role played by the result.* The result of the **&** operator has a bit-set role, while that of the **&&** operator has a boolean role. For instance, the measurements in Table 1234.1 show that most occurrences of the **&&** operator are in a controlling expression (both are boolean roles).

object role boolean role bit-set role numeric role

The issue of operands having the same role is covered by the guideline recommendation dealing with objects having a single role (the definitions of bit-set and numeric roles are based on objects appearing as operands of certain kinds of operators. An object that appeared as the operand of both kinds of operator would have two roles).

role operand matching ?? object used in a single role bit-set role numeric role

Other coding guideline documents sometimes specify that these two operators should not be confused, or list them in review guidelines.[2] Using **&&** where **&** was intended, or vice versa, is clearly unintended (a fault) and these coding guidelines are not intended to recommend against the use of constructs that are obviously faults. However, it may be possible to reduce the likelihood of confusion by using the operator appropriate to the role. The issue of the role of operands matching that of their operators is discussed elsewhere.

MISRA guidelines not faults 1234 role operand matching

Table 1234.1: Occurrence of the **&** and **&&** operator (as a percentage of all occurrences of each operator; the parenthesized value is the percentage of all occurrences of the context that contains the operator). Based on the visible form of the `.c` files.

Context	Binary &	&&
if control-expression	51.4 (10.5)	82.4 (10.4)
other contexts	45.3 (—)	7.7 (—)
while control-expression	2.1 (8.1)	6.9 (18.4)
for control-expression	0.3 (0.6)	3.0 (4.7)
switch control-expression	0.8 (5.2)	0.0 (0.0)

Table 1234.2: Common token pairs involving one of the operators **&**, **|**, or **^** (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 |	0.4	74.0	& identifier	57.1	0.6
identifier &	0.7	67.5	| identifier	79.8	0.4
identifier ^	0.0	51.1	& (	7.4	0.3
) ^	0.0	38.7	| (	14.4	0.3
& ~	4.6	30.1	^ *v	5.5	0.1
) &	1.1	27.7	| *integer-constant*	5.5	0.1
) |	0.4	20.8	^ *integer-constant*	20.8	0.0
] ^	0.0	5.1	^ identifier	55.5	0.0
] &	1.4	4.2	^ (	16.1	0.0
& *integer-constant*	30.6	1.5			

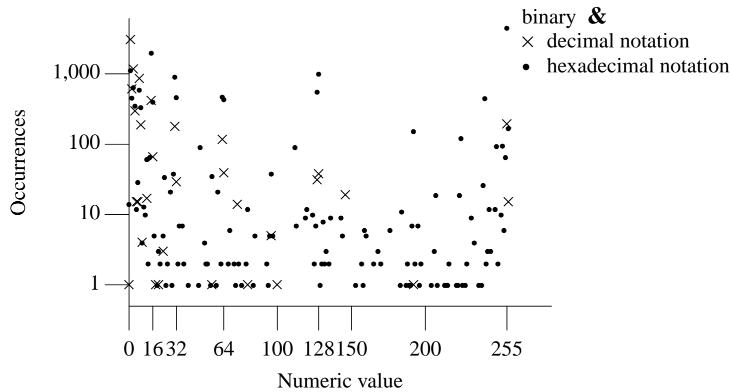

Figure 1234.1: Number of *integer-constants* having a given value appearing as the right operand of the binary **&** operator. Based on the visible form of the `.c` files.

Constraints

& binary operand type

Each of the operands shall have integer type. 1235

Commentary

This constraint reflects the fact that processors very rarely contain instructions for performing this operation on non-integer types. In turn this reflects the fact that there is no commonly defined semantics for bitwise ANDing other types and that there are very few algorithms requiring values having other types to be treated as a sequence of bits.

C++

The wording of the specification in the C++ Standard is somewhat informal (the same wording is given for the bitwise exclusive-OR operator, 5.12p1, and the bitwise inclusive-OR operator, 5.13p1).

5.11p1

The operator applies only to integral or enumeration operands.

Coding Guidelines

represen-?? tation information using
boolean role
bit-set role

The binary **&** operator operates on the bit representation of its operands. As such, the guideline recommendation on making use of representation information is applicable. However, deviations are suggested for operands having a boolean or bit-set role.

Table 1235.1: Occurrence of bitwise 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.

<table>
<tr><th>Left Operand</th><th>Operator</th><th>Right Operand</th><th>%</th><th>Left Operand</th><th>Operator</th><th>Right Operand</th><th>%</th></tr>
<tr><td>int</td><td>|</td><td>_int</td><td>27.1</td><td>unsigned int</td><td>|</td><td>unsigned int</td><td>4.0</td></tr>
<tr><td>int</td><td>&</td><td>_int</td><td>24.3</td><td>unsigned long</td><td>&</td><td>_int</td><td>3.8</td></tr>
<tr><td>_int</td><td>|</td><td>_int</td><td>23.0</td><td>unsigned int</td><td>|</td><td>unsigned long</td><td>3.4</td></tr>
<tr><td>unsigned int</td><td>^</td><td>unsigned int</td><td>17.7</td><td>unsigned int</td><td>^</td><td>_int</td><td>3.3</td></tr>
<tr><td>other-types</td><td>&</td><td>other-types</td><td>13.9</td><td>unsigned int</td><td>^</td><td>int</td><td>3.1</td></tr>
<tr><td>int</td><td>|</td><td>int</td><td>13.6</td><td>unsigned long</td><td>&</td><td>int</td><td>2.6</td></tr>
<tr><td>_int</td><td>^</td><td>_int</td><td>13.5</td><td>long</td><td>^</td><td>long</td><td>2.6</td></tr>
<tr><td>unsigned long</td><td>^</td><td>unsigned long</td><td>12.2</td><td>unsigned char</td><td>&</td><td>int</td><td>2.5</td></tr>
<tr><td>unsigned int</td><td>&</td><td>_int</td><td>11.5</td><td>unsigned long</td><td>|</td><td>unsigned long</td><td>2.4</td></tr>
<tr><td>unsigned char</td><td>&</td><td>_int</td><td>10.3</td><td>unsigned long</td><td>&</td><td>unsigned long</td><td>2.0</td></tr>
<tr><td>int</td><td>^</td><td>_int</td><td>10.3</td><td>unsigned int</td><td>^</td><td>unsigned char</td><td>1.8</td></tr>
<tr><td>other-types</td><td>^</td><td>other-types</td><td>9.9</td><td>unsigned short</td><td>^</td><td>unsigned short</td><td>1.7</td></tr>
<tr><td>int</td><td>^</td><td>int</td><td>9.8</td><td>int</td><td>^</td><td>unsigned char</td><td>1.7</td></tr>
<tr><td>unsigned int</td><td>|</td><td>int</td><td>9.6</td><td>unsigned short</td><td>&</td><td>unsigned short</td><td>1.5</td></tr>
<tr><td>other-types</td><td>|</td><td>other-types</td><td>8.9</td><td>unsigned short</td><td>^</td><td>_int</td><td>1.5</td></tr>
<tr><td>unsigned short</td><td>&</td><td>_int</td><td>7.1</td><td>long</td><td>&</td><td>int</td><td>1.4</td></tr>
<tr><td>int</td><td>&</td><td>int</td><td>6.3</td><td>int</td><td>|</td><td>unsigned char</td><td>1.4</td></tr>
<tr><td>unsigned int</td><td>&</td><td>int</td><td>5.7</td><td>unsigned short</td><td>&</td><td>int</td><td>1.3</td></tr>
<tr><td>long</td><td>|</td><td>long</td><td>5.5</td><td>unsigned int</td><td>^</td><td>unsigned short</td><td>1.3</td></tr>
<tr><td>unsigned int</td><td>&</td><td>unsigned int</td><td>4.6</td><td>long</td><td>&</td><td>_int</td><td>1.2</td></tr>
<tr><td>unsigned char</td><td>^</td><td>unsigned char</td><td>4.6</td><td>_int</td><td>|</td><td>int</td><td>1.2</td></tr>
<tr><td>unsigned char</td><td>^</td><td>_int</td><td>4.2</td><td>int</td><td>^</td><td>unsigned short</td><td>1.1</td></tr>
</table>

Semantics

1236 The usual arithmetic conversions are performed on the operands.

& binary operands converted

Commentary

The rationale for performing these conversions is a general one that is not limited to the operands of the arithmetic operators.

operators cause conversions

C++

The following conversion is presumably performed on the operands.

> *The usual arithmetic conversions are performed;*

5.11p1

Common Implementations

If one of the operands is positive and has a type with lower rank than the other operand, it may be possible to make use of processor instructions that operate on narrower width values. (Converting the operand to the greater rank will cause it to be zero extended, which will cancel out any ones in the other operand.) Unless both operands are known to be positive, there tend to be few opportunities for optimizing occurrences of the **^** and **|** operators (because of the possibility that the result may be affected by an increase in value representation bits).

Coding Guidelines

Unless both operands have the same type, which also has a rank at least equal to that of **int**, these conversions will increase the number of value representation bits in one or both operands. Given that the binary **&** operator is defined to work at the bit level, developers have to invest additional effort in considering the effects of the usual arithmetic operands on the result of this operator.

A probabilistic argument could be used to argue that of all the bitwise operators the **&** operator has the lowest probability of causing a fault through an unexpected increase in the number of value representation

bits. For instance, the probability of both operands having a negative value (needed for any additional bits to be set in the result) is lower than the probability of one operand having a negative value (needed for a bit to be set in the result of the ^ and | operators).

bit-set role
representation information using ??

Unless the operands have a bit-set role, the guideline recommendation dealing with use of representation information is applicable here.

The result of the binary & operator is the bitwise AND of the operands (that is, each bit in the result is set if and only if each of the corresponding bits in the converted operands is set). 1237

Commentary

This information is usually expressed in tabular form.

	0	1
0	0	0
1	0	1

Common Implementations

The Unisys A Series[1] uses signed magnitude representation. If the operands have an unsigned type, the bit used to represent the sign in signed types, which is present in the object representation on this processor, does not take part in the binary & operation. If the operands have a signed type, the sign bit does take part in the bitwise-AND operation.

Coding Guidelines

usual arithmetic conversions
object role

Although the result of the bitwise-AND operator is the common type derived from the usual arithmetic conversions, for the purpose of these guideline recommendations its role is the same as that of its operands.

---

footnote 92

92) Two objects may be adjacent in memory because they are adjacent elements of a larger array or adjacent members of a structure with no padding between them, or because the implementation chose to place them so, even though they are unrelated. 1238

Commentary

array contiguously allocated set of objects
structure members later compare later
structure unnamed padding
storage layout

By definition elements of an array are contiguous and the standard specifies the relative order of members and their possible padding. Some implementations treat objects defined by different declarations in the same way as the declaration of members in a structure definition. For instance, assigning the same relative offsets to objects local to a function definition as they would the equivalent member declarations in a structure type. The issue of the layout of objects in storage is discussed elsewhere.

This footnote lists all the cases where objects may be adjacent in memory.

C90

The C90 Standard did not discuss these object layout possibilities.

C++

The C++ Standard does not make these observations.

Other Languages

Most languages do not get involved in discussing this level of detail.

Common Implementations

Most implementations aim to minimize the amount of padding between objects. In many cases objects defined in block scope are adjacent in memory to objects defined textually adjacent to them in the source code.

If prior invalid pointer operations (such as accesses outside array bounds) produced undefined behavior, subsequent comparisons also produce undefined behavior. 1239

Commentary

This describes a special case of undefined behavior. (It is called out because, in practice, it is more likely to occur for values having pointer types than values having integer types.) Once undefined behavior has occurred, any subsequent operation can also produce undefined behavior. The standard does not limit the scope of undefined behaviors to operations involving operands that cause the initial occurrence.

Common Implementations

On many implementations the undefined behavior that occurs when additive operations, on values having an integer type, overflow is *symmetrical*. That is, an overflow in one direction can be undone by an overflow in the other direction (e.g., `INT_MAX+2-2` produces the same result as `INT_MAX-2+2`). On implementations for processors using a segmented architecture this symmetrical behavior may not occur, for values having pointer types, because of internal details of how pointer arithmetic is handled on segment boundaries.

pointer
segmented architecture

On some processors additive operations, on integer or pointer types, saturate. In this case the undefined behavior is not symmetrical.

arithmetic
saturated

C90

The C90 Standard did not discuss this particular case of undefined behavior.

References

1. Unisys Corporation. *C Programming Reference Manual, Volume 1: Basic Implementation*. Unisys Corporation, 8600 2268-203 edition, 1998.
2. W. D. Yu. A software fault prevention approach in coding and root cause analysis. *Bell Labs Technical Journal*, Apr.-June 1998.