The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

6.5.13 Logical AND operator

logical-AND-
expression
syntax

```
logical-AND-expression:
            inclusive-OR-expression
            logical-AND-expression && inclusive-OR-expression
```

Commentary

The relative precedence of the binary **&&** and **||** operators is the same as that of the binary **&** and **|** operators.

Other Languages

bitwise
&

In many languages the logical-AND operator has higher precedence than the logical-OR operator. In a few languages (Ada, Fortran which uses the keyword **.AND.**), they have the same precedence. Ada supports two kinds of logical-AND operations: **and** and **and then**, the latter having the same semantics as the logical-AND operator in C (short-circuit evaluation). These operators also have the same precedence as the logical-OR and logical-XOR operators.

Coding Guidelines

AND-
expression
syntax

The issue of swapping usages of the **&** and **&&** operators is discussed elsewhere.

There are a number of techniques for simplifying expressions involving boolean values. One of the most commonly used methods is the Karnaugh map.[5] (For equations involving more than five operands, the Quine-McCluskey technique[7] may be more applicable; this technique is also capable of being automated.), while some people prefer algebraic manipulation (refer to Table 1248.1).

Although simplification may lead to an expression that requires less reader effort to comprehend as a boolean expression, the resulting expression may require more effort to map to the model of the application domain being used. For instance, (A && (!B)) || ((!A) && B) can be simplified to A ^ B (assuming A and B only take the values 0 and 1, otherwise another operation is required). However, while the use of the exclusive-OR operator results in a visually simpler expression, developers have much less experience dealing with it than the other bitwise and logical operators. There is also the possibility that, for instance, the expression (A && (!B)) occurs in other places within the source and has an easily-deduced meaning within the framework of the application model.

mind
logical operator

Logical operators are part of mathematical logic. Does the human mind contain special circuitry that performs this operation, just like most processors contain a group of transistors that perform this C operation? Based on experimental observations, the answer would appear to be no. So how does the human mind handle the logical-AND operation? One proposal is that people learn the rules of this operator by rote so that they can later be retrieved from memory. The result of a logical operation is then obtained by evaluating each operand's condition to true or false and performing a table lookup using previously the learned logical-AND table to find the result. Such an approach relies on short-term memory, and the limited capacity of short-term memory offers one explanation of why people are poor at evaluating moderately complex logical operators in their head. There is insufficient capacity to hold the values of the operands and the intermediate results.

The form of logical deduction that is needed in comprehending software rarely occurs in everyday life. People's ability to solve what appear to be problems in logic does not mean that the methods of boolean mathematics are used. A number of other proposals have been made for how people handle *logical* problems in everyday life. One is that the answers to the problems are simply remembered; after many years of life experience, people accumulate a store of knowledge on how to deal with different situations.

Studies have found that belief in a particular statement being true can cause people to ignore its actual status as a mathematical expression (i.e., people believe what they consider to be true rather than logically evaluating the true status of an expression). Estimating developers' performance at handling logical expressions therefore involves more than a model of how they process mathematical logic.

While minimizing the number of operands in a boolean expression may have appeal to mathematically oriented developers, the result may be an expression that requires more effort to comprehend. It is not yet

possible to calculate the reader effort required to comprehend a particular boolean expression. For this reason the following guideline recommendation relies on the judgment of those performing the code review.

Rev 1248.1

> Boolean expressions shall be written in a form that helps minimize the effort needed by readers to comprehend them.

A comment associated with the simplified expression can be notated in at least two ways: using equations or by displaying the logic table. Few developers are sufficiently practiced at boolean algebra to be able to fluently manipulate expressions; a lot of thought is usually required. A logic table highlights all combinations of operands and, for small numbers of inputs, is easily accommodated in a comment.

Table 1248.1: Various identities in boolean algebra expressed using the `||` and **`&&`** operators. Use of these identities may change the number of times a particular expression is evaluated (which is sometimes the rationale for rewriting it). The relative order in which expressions are evaluated may also change (e.g., when A==1 and B==0 in `(A && B) || (A && C)` the order of evaluation is A⇒ B⇒ A⇒ C, but after use of the distributive law the order becomes A⇒ B⇒ C).

Relative Order Preserved	Expression ⇒ Alternative Representation
	Distributive laws
no	`(A && B) || (A && C)` ⇒ `A && (B || C)`
no	`(A || B) && (A || C)` ⇒ `A || (B && C)`
	DeMorgan's theorem
yes	`!(A || B)` ⇒ `(!A) && (!B)`
yes	`!(A && B)` ⇒ `(!A) || (!B)`
	Other identities
yes	`A && ((!A) || B)` ⇒ `A && B`
yes	`A || ((!A) && B)` ⇒ `A || B`
	The consensus identities
no	`(A && B) || ((!A) && C) || (B && C)` ⇒ `(A && B) || ((!A) && C))`
yes	`(A && B) || (A && (!B) && C)` ⇒ `(A && B) || (A && C)`
yes	`(A && B) || ((!A) && C)` ⇒ `((!A) || B) && (A || C)`

An expression containing a number of logical operators, each having operands whose evaluation involves relational or equality operators, can always be written in a number of different ways, for instance:

```
1   if ((X < 4) && !(Y || (Z == 1)))
2       /* ... */
3
4   if ((Y != 0) && (Z != 0) && (X < 4))
5       /* ... */
6
7   if (!((X >= 4) || Y || (Z == 1)))
8       /* ... */
9
10  if (X < 4)
11      if (!(Y || (Z == 1)))
12          /* ... */
```

An example of complexity in an English sentence might be "Suppose five days after the day before yesterday is Friday. What day of the week is tomorrow?" Whether the use of a less complex (i.e., having less cognitive load) expression has greater cost/benefit than explicitly calling out the details of the calculation needs to be determined on a case-by-case basis.

A study by Feldman[3] found that the subjective difficulty of a concept (e.g., classifying colored polygons of various sizes) was directly proportional to its boolean complexity (i.e., the length of the shortest logically equivalent propositional formula). categorization performance predicting

Table 1248.2: Common token pairs involving **&&**, 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	48.5	**&&** defined	0.9	6.2
) **||**	0.9	42.7	**||** **!**	11.3	6.0
identifier **||**	0.2	39.3	*character-constant* **||**	4.2	4.2
) **&&**	1.1	34.9	*character-constant* **&&**	5.3	3.3
|| defined	4.8	21.0	**&&** **(**	28.7	0.9
integer-constant **||**	0.3	12.4	**||** **(**	29.7	0.6
integer-constant **&&**	0.4	11.5	**&&** identifier	53.9	0.5
&& **!**	13.5	11.3	**||** identifier	51.8	0.3

Constraints

&&
operand type

Each of the operands shall have scalar type.   1249

Commentary

The behavior is defined in terms of an implicit comparison against zero, an operation which is only defined for operands having a scalar type in C.

C++

5.14p1   *The operands are both implicitly converted to type* **bool** *(clause 4).*

Boolean conversions (4.12) covers conversions for all of the scalar types and is equivalent to the C behavior.

Other Languages

Languages that support boolean types usually require that the operands to their logical-AND operator have boolean type.

Coding Guidelines

& binary
operand type
if statement
controlling
expression
scalar type

The discussion on the bitwise-AND operator is also applicable here, and the discussion on the comprehension of the controlling expression in an **if** statement is applicable to both operands.

Table 1249.1: Occurrence of logical 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	%
int	**||**	**int**	87.7	**_long**	**||**	**_long**	2.2
int	**&&**	**int**	73.9	**int**	**&&**	ptr-to	2.2
other-types	**&&**	other-types	12.8	**int**	**&&**	**char**	1.8
other-types	**||**	other-types	8.4	**int**	**||**	**_long**	1.7
ptr-to	**&&**	**int**	4.5	**int**	**&&**	**_int**	1.3
char	**&&**	**int**	2.3	ptr-to	**&&**	ptr-to	1.1

Semantics

&&
operand compare
against 0

The **&&** operator shall yield 1 if both of its operands compare unequal to 0;   1250

Commentary

The only relationship between the two operands is their appearance together in a logical-AND operation. There is no benefit, from the implementations point of view, in performing the usual arithmetic conversions or the integer promotions on each operand.

It is sometimes necessary to test a preliminary condition before the main condition can be tested. For instance, it may be necessary to check that a pointer value is not null before dereferencing it. The test could be performed using nested **if** statements, as in:

```
1   if (px != NULL)
2       if (px->m1 == 1)
```

More complex conditions are likely to involve the creation of a warren of nested **if** statements. The original designers of the C language decided that it was worthwhile creating operators to provide a shorthand notation (i.e., the logical-AND and logical-OR operators). In the preceding case use of one of these operators allows both tests to be performed within a single conditional expression (e.g., if ((px != NULL) && (px->m1 == 1))). The other advantage of this operator, over nested **if** statements, is in the creation of expressions via the expansion of nested macro invocations. Generating nested **if** statements requires the use of braces to ensure that any following **else** arms are associated with the intended **if** arm. Generating these braces introduces additional complexities (at least another macro invocation in the source) that don't occur when the **&&** operator is used.

C++

> *The result is* **true** *if both operands are* **true** *and* **false** *otherwise.* 5.14p1

The difference in operand types is not applicable because C++ defines equality to return **true** or **false**. The difference in return value will not cause different behavior because **false** and **true** will be converted to 0 and 1 when required.

Other Languages

The mathematical use of this operator returns true if both operands are true. Languages that support a boolean data type usually follow this convention.

Common Implementations

As discussed elsewhere, loading a value into a register often sets conditional flags as does a relational operation comparing two values. This means that in many cases machine code to compare against zero need not be generated, as in, the following condition:

logical
negation
result is
relational
operators
result value

```
1   if ((a == b) && (c < d))
```

which is likely to generate an instruction sequence of the form:

```
                compare a with b
                if not equal jump to else_or_endif
                compare c with d
                if greater than or equal to jump to else_or_endif
                code for if arm
else_or_endif:  rest of program
```

Coding Guidelines

Should the operands always be explicitly compared against zero? When an operand is the result of a relational or equality operator, such a comparison is likely to look very unusual:

```
1    if (((a == b) == 0) && ((c < d) == 0))
```

It is assumed that developers think about the operands in terms of boolean roles and the result of both the relational and equality operators have a boolean role. In these cases another equality comparison is redundant. When an operand does not have a boolean role, an explicit comparison against zero might be appropriate (these issues are also discussed elsewhere).

!
equivalent to
selection
statement
syntax

---

otherwise, it yields 0.   1251

Commentary

That is, a value of zero having type **int**.

&&
result type

The result has type **int**.   1252

Commentary

relational
operators
result type

The rationale for this choice is the same as for relational operators.

C++

5.14p2   *The result is a **bool**.*

The difference in result type will result in a difference of behavior if the result is the immediate operand of the **sizeof** operator. Such usage is rare.

Other Languages

In languages that support a boolean type, the result of logical operators usually has a boolean type.

Common Implementations

If the result is immediately cast to another type, an implementation may choose to arrange that other type as the result type; for instance, in:

```
1    float f(int ip)
2    {
3    return (ip > 0) && (ip < 10);
4    }
```

an implementation may choose to return 0.0 and 1.0 as the result of the expression evaluation, saving a cast operation.

Coding Guidelines

relational
operators
result type

The coding guideline issues are the same as those for the relational operators.

&&
evaluation order

Unlike the bitwise binary **&** operator, the **&&** operator guarantees left-to-right evaluation;   1253

Commentary

The left-to-right evaluation order of the **&&** operator is required; it is what makes it possible for the left operand to verify a precondition for the evaluation of the right operand. There is no requirement that the evaluation of the right operand, if it occurs, take place immediately after the evaluation of the left operand.

sequence ??
points
all orderings
give same value

Coding Guidelines

This issue is covered by the guideline recommendation dealing with sequence points.

Example

In the following:

```
1   #include <stdio.h>
2
3   void f(void)
4   {
5   (printf("Hello ") && printf("World\n")) + printf("Goodbye\n");
6   }
```

possible output includes:

Hello World
Goodbye

and

Hello Goodbye
World

1254 there is a sequence point after the evaluation of the first operand.

&&
sequence point

Commentary

Unlike the nested **if** statement form, there is no guaranteed sequence after the evaluation of the second operand, if it occurs.

1250 &&
operand com-
pare against
0

C++

All side effects of the first expression except for destruction of temporaries (12.2) happen before the second expression is evaluated.

5.14p2

The possible difference in behavior is the same as for the function-call operator.

function call
sequence point

Coding Guidelines

While the sequence point ensures that any side effects in the first operand have completed, relying on this occurring creates a dependency within the expression that increases the effort needed to comprehend it. Some of the issues involving side effects in expressions are discussed elsewhere.

object
modified once
between sequence
points

Example

```
1   #include <stdio.h>
2
3   extern int glob;
4
5   void f(void)
6   {
7   if (--glob && ++glob)
8      printf("The value of glob is neither 0 nor 1\n");
9   }
```

1255 If the first operand compares equal to 0, the second operand is not evaluated.

&&
second operand

Commentary

An alternative way of looking at this operator is that x && y is equivalent to x ? (y?1:0) : 0.

Common Implementations

This operand can be implemented, from the machine code generation point of view, as if a nested **if** construct had been written.

Coding Guidelines

Some coding guideline documents prohibit the second operand containing side effects. The rationale is that readers of the source may fail to notice that if the second operand is not evaluated any side effects it generates will not occur. The majority of C's binary operators (32 out of 34) always evaluate both of their operands. Do developers make the incorrect assumption that the operands of all binary operators are always evaluated? Your author thinks not. Many developers are aware that in some cases the **&&** operator is more efficient than the **&** operator, because it only evaluates the second operand if the first is not sufficient to return a result.

Although many developers may be aware of the conditional evaluation of the second operand, some will believe that both operands are evaluated. While a guideline recommendation against side effects in the second operand may have a benefit for some developers, it potentially increases cost in that the alternative construct used may require more effort to comprehend (e.g., the alternative described before). Given that the alternative constructs that could be used are likely to require more effort to comprehend and the unknown percentage of developers making incorrect assumptions about the evaluation of the operands, no guideline recommendation is given.

coverage testing **Coverage testing**

There are software coverage testing requirements that are specific to logical operators. Branch condition decision testing involves the individual operands of logical operators. It requires that test cases be written to exercise all combinations of operands. In:

```
1   if (A || (B && C))
```

it is necessary to verify that all combinations of A, B, and C, are evaluated. In the case of the condition involving A, B, and C eight separate test cases are needed. For more complex conditions, the number of test cases rapidly becomes impractical.

Modified condition decision testing requires that test cases be written to demonstrate that each operand can independently affect the output of the decision. For the **if** statement above, Table 1255.1 shows the possible conditions. Using modified condition, *MC*, coverage can significantly reduce the number of test cases over branch condition, *BC*, coverage.

Table 1255.1: Truth table showing how each operand of (A || (B && C)) can affect its result. Case 1 and 2 show that A affects the outcome; Case 2 and 3 shows that B affects the outcome; Case 3 and 4 shows that C affects the outcome.

Case	A	B	C	Result
1	FALSE	FALSE	TRUE	FALSE
2	TRUE	FALSE	TRUE	TRUE
3	FALSE	TRUE	TRUE	TRUE
4	FALSE	TRUE	FALSE	FALSE

The FAA requires[6] that aviation software at Level A (the most critical, defined as that which could prevent continued safe flight and landing of the aircraft) have the level of coverage specified by MC/DC.[1] This requirement has been criticized as not having a worthwhile cost (money and time) and benefit (the number of errors found) ratio. An empirical evaluation of the HETE–2 satellite software[2] looked at the faults found by various test methods and their costs. Logical operators can also appear within expressions that are not a condition context. The preceding coverage requirements also hold in these cases.

Horgan and London[4] describe an Open Source tool that measures various dataflow coverage metrics for C source.

References

1. BCS SIGIST. Standard for software component testing. Technical Report Working Draft 3.4, British Computer Society and Special Interest Group in Software Testing, Apr. 2001.

2. A. Dupuy and N. Leveson. An empirical evaluation of the MC/DC coverage criterion on the HETE-2 satellite software. In *Proceedings of the Digital Aviation Systems Conference (DASC)*, pages 1B6/1–1B6/7, Oct. 2000.

3. J. Feldman. Minimization of boolean complexity in human concept learning. *Nature*, 407:630–633, Oct. 2000.

4. J. R. Horgan and S. London. Data flow coverage and the C language. In *Proceedings of the 4th Symposium on Software Testing, Analysis, and Verification*, pages 87–97. ACM Press, Oct. 1991.

5. M. Karnaugh. The map method for synthesis of combinational logic circuits. *AIEE Transactions Comm. Elec*, 72:593–599, 1953.

6. RTCA. Software considerations in airborne systems and equipment certifications, DOD-178B. Technical report, RTCA, Washington D.C., 1992.

7. T. Q. M. S. Tool. Botsch. freshmeat.net/projects/qmc, 2003.