

The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones

derek@knosof.co.uk

6.5.15 Conditional operator

conditional-expression
syntax

conditional-expression:

logical-OR-expression

logical-OR-expression ? *expression* : *conditional-expression*

Commentary

The conditional operator is not necessary in that it is possible to implement the functionality it provides using other operators, statements, and sometimes preprocessing directives. However, C source code is sometimes automatically generated and C includes a preprocessor. The automated generation C source code is sometimes easier if conditional tests can be performed within an expression.

The second operand of a *conditional-expression* is *expression* which means that the ? and : tokens effectively act as brackets for the second expression; no parentheses are required. For instance, a ? (b , c) : d , e can also be written as a ? b , c : d , e. The conditional operator associates to the right.

operator
associativity

C++

5.16 *conditional-expression*: *logical-or-expression* *logical-or-expression* ? *expression* : *assignment-expression*

assignment-expression
syntax

By supporting an *assignment-expression* as the third operand, C++ enables the use of a *throw-expression*; for instance:

```
z = can_I_deal_with_this() ? 42 : throw X;
```

Source developed using a C++ translator may contain uses of the conditional operator that are a constraint violation if processed by a C translator. For instance, the expression `x?a:b=c` will need to be rewritten as `x?a:(b=c)`.

Other Languages

In some languages (e.g., Algol 68) statements can return values (they are treated as expressions). For such languages the functionality of a conditional expression is provided by using an **if** statement (within an expression context). BCPL supports conditional expressions, using the syntax: *expression* -> *expression* , *expression*.

Common Implementations

MetaWare High C (in nonstandard mode), some versions of pcc, and gcc support the syntax:

conditional-expression:

logical-or-expression

logical-or-expression ? *expression* : *assignment-expression*

compound
expression

gcc allows the second operand to be omitted. The expression `x ? : y` is treated as equivalent to `x ? x : y`; gcc also supports compound expressions, which allow **if** statements to appear within an expression.

Coding Guidelines

Most developers do not have sufficient experience with the conditional operator to be familiar with its precedence level relative to other operators. Using parentheses removes the possibility of developers mistakenly using the incorrect precedence.

expression ??
shall be paren-
thesized

Table 1264.1: Common token pairs involving ? or : (to prevent confusion with the : punctuation token the operator form is denoted by ?:) (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
) ?	0.4	44.7	? <i>string-literal</i>	20.1	1.5
identifier ?	0.1	44.0	?: <i>integer-constant</i>	28.7	0.3
identifier ?:	0.1	40.3	? <i>integer-constant</i>	20.2	0.2
<i>integer-constant</i> ?:	0.3	23.1	? identifier	43.9	0.1
<i>string-literal</i> ?:	1.5	20.2	?: identifier	35.9	0.1
) ?:	0.1	11.6	?: (7.2	0.1
<i>integer-constant</i> ?	0.1	9.6	? (6.2	0.1
?: <i>string-literal</i>	21.0	1.6			

Constraints

1265 The first operand shall have scalar type.

Commentary

This is the same requirement as that given for a controlling expression in a selection statement and is specified for the same reasons.

if statement
controlling
expression scalar
type

C++

*The first expression is implicitly converted to **bool** (clause 4).*

5.16p1

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

Coding Guidelines

Many of the issues that apply to the controlling expression of an **if** statement are applicable here.

if statement
controlling
expression scalar
type

1266 One of the following shall hold for the second and third operands:

Commentary

The result of the conditional operator is one of its operands. The following list of constraints ensures that the value of both operands can be operated on in the same way by subsequent operators (occurring in the evaluation of an expression).

conditional
operator
second and
third operands
1277 conditional
operator
result

Table 1266.1: Occurrence of the ternary : operator (denoted by the character sequence ?:) 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	%
ptr-to	?:	ptr-to	29.5	int	?:	_int	5.7
other-types	?:	other-types	12.1	_char	?:	_char	3.4
_int	?:	_int	10.4	unsigned int	?:	unsigned int	2.2
int	?:	int	10.0	unsigned short	?:	unsigned short	1.2
void	?:	void	9.4	signed int	?:	_int	1.1
unsigned long	?:	unsigned long	7.9	char	?:	void	1.1
_int	?:	int	6.0				

1267 — both operands have arithmetic type;

Commentary

In this case it is possible to perform the usual arithmetic conversions to bring them to a common type.

Coding Guidelines

The guideline recommendation dealing with objects being used in a single role implies that the second and third operands of a conditional operator should have the same role.

object ??
used in a
single role

— both operands have the same structure or union type;

1268

Commentary

While the structure or union types may be the same, the evaluation of the expressions denoting the two operands may be completely different. However, a translator still has to ensure that the final result of both operands, used by any subsequent operators, is held in the same processor location.

— both operands have void type;

1269

Commentary

In this case the conditional operator appears either as the left operand of the comma operator, the top-level operator of an expression statement, or as the second or third operand of another conditional operator in these contexts. In either case alternative constructs are available. However, when dealing with macros that may involve nested macro invocations, not having to be concerned with replacements that result in operands having **void** type (invariably function calls) is a useful simplification.

comma
operator
syntax

— both operands are pointers to qualified or unqualified versions of compatible types;

1270

Commentary

The discussion on the equality operators is applicable here.

C++

conditional
expression
pointer to com-
patible types

equality
operators
pointer to com-
patible types

5.16p6

— *The second and third operands have pointer type, or one has pointer type and the other is a null pointer constant; pointer conversions (4.10) and qualification conversions (4.4) are performed to bring them to their composite pointer type (5.9).*

These conversions will not convert a pointer to an enumerated type to a pointer to integer type.

If one pointed-to type is an enumerated type and the other pointed-to type is the compatible integer type. C permits such operands to occur in the same *conditional-expression*. C++ does not. See pointer subtraction for an example.

subtraction
pointer operands

— one operand is a pointer and the other is a null pointer constant; or

1271

Commentary

The discussion on the equality operators is applicable here.

— one operand is a pointer to an object or incomplete type and the other is a pointer to a qualified or unqualified version of **void**.

1272

Commentary

The discussion on the equality operators is applicable here.

equality
operators
pointer to in-
complete type

C++

The C++ Standard does not support implicit conversions from pointer to **void** to pointers to other types (4.10p2). Therefore, this combination of operand types is not permitted.

```

1  int glob;
2  char *pc;
3  void *pv;
4
5  void f(void)
6  {
7  glob ? pc : pv; /* does not affect the conformance status of the program */
8                // ill-formed
9  }
```

Semantics

1273 The first operand is evaluated;

Commentary

Much of the discussion on the evaluation of the controlling expression of an **if** statement is applicable here.

selection
statement
syntax

Common Implementations

The only difference between evaluating this operand and a controlling expression is that in the former case it is possible for more processor registers to be in use, holding results from the evaluation of other subexpressions. (This likelihood of increased register pressure may result in spilling for processors with relatively few registers— e.g., Intel x86.) However, whether the overall affect of using a conditional operator is likely to be small, compared to an **if** statement, will depend on the characteristics of the expression and optimizations performed.

controlling
expression
if statement

register
spilling

1274 there is a sequence point after its evaluation.

Commentary

Unlike the controlling expression in a selection statement, this operand is not a full expression, so this specification of a sequence point is necessary to fully define the evaluation order. The discussion on the logical-AND operator is applicable here.

conditional
operator
sequence point

selection
statement
syntax
full expres-
sion
&&
sequence point

C++

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

5.16p1

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

function call
sequence point

1275 The second operand is evaluated only if the first compares unequal to 0;

Commentary

Because the operand might not be evaluated, replacing a conditional operator by a function call, taking three arguments, would not have the same semantics.

conditional
operator
operand only
evaluated if

Coding Guidelines

The guideline discussion on the logical-AND operator is applicable here.

&&
second operand

1276 the third operand is evaluated only if the first compares equal to 0;

Commentary

Either the second or the third operand is always evaluated, but never both.

the result is the value of the second or third operand (whichever is evaluated), converted to the type described below.⁹³⁾

1277

Commentary

The conversion, to a common compatible type, ensures that the result of evaluating either operand has the same value representation.

Common Implementations

This places a requirement on the generated machine code to ensure that the result value is always left in the same register or storage location. Subsequent machine instructions will reference this single location.

Coding Guidelines

The difference between an **if** statement and a conditional operator is that in the latter case there is a result that can appear as the operand of another operator. Measurements of existing source suggest that developers have significantly more experience dealing with **if** statements than conditional operators. Using a conditional operator in a context in the visible source, where its result is not used (i.e., is not an operand to another operator), fails to take advantage of a developer's greater experience in dealing with **if** statements. However, such usage is rare and a guideline recommending the use of an **if** statement in this case is not considered worthwhile.

The conditional operator is not necessary for the writing of any program; it is always possible to rewrite source so that it does not contain any conditional operators. Similarly, source can be rewritten, replacing some **if** statements by conditional operators. How does the cost/benefit of using an **if** statement compare to that of using a conditional operator? The two main issues are comprehension and maintenance:

- From a reader's perspective, the comprehension of a conditional operator may, or may not, require more cognitive effort than comprehending an **if** statement. For instance, in the following example the use of an **if** statement highlights the controlling expression while the single assignment highlights that the array `a` is being modified (which needs to be deduced by reading two statements in the former case);

```

1  if (x)
2    a[y]=0;
3  else
4    a[z]=0;
5
6  a[x ? y : z]=0;
```

When the expression being assigned to is complex, significant reader effort may be required to deduce that the two complex expressions are the same. Use of a conditional operator removes the need to make this comparison:

```

1  if (x)
2    a[complicated_expr]=y;
3  else
4    a[complicated_expr]=z;
5
6  a[complicated_expr]=(x ? y : z);
```

When an expression in a substatement, of an **if** statement, contains a reference to an object that also occurs in the controlling expression, for instance:

```

1  if (x < 5)
2    a=10-x;
```

```

3  else
4      a=x;
5
6  a=(x < 5 ? 10-x : x);

```

readers may need to keep information about the condition in their minds when comprehending the source in both cases (i.e., maximum cognitive load is very similar, if not the same). When an expression in a substatement does not contain such a reference, readers do not need to evaluate information about the conditional to comprehend that statement. Compared to selection statements the maximum cognitive effort is likely to be less.

selection
statement
syntax

- In the following example a source modification requires that both y and z be assigned to b, instead of a, can be performed by editing one identifier in the case of the conditional operator usage, while the **if** statement usage requires two edits. However, a modification that required additional statements to be executed, if x were true (or false), would require far less editing for the **if** statement usage.

```

1  if (x)
2      a=y;
3  else
4      a=z;
5
6  a = (x ? y : z);

```

if statements occur in the visible source much more frequently than conditional operators. One reason is that in many cases the **else** arm is not applicable (only 18% of **if** statements in the visible form of the .c files contain an **else** arm). Because on this usage pattern readers receive more practice with the use of **if** statements. Given this difference in familiarity, it is not surprising that conditional operators are used less frequently than might be expected.

1278 If an attempt is made to modify the result of a conditional operator or to access it after the next sequence point, the behavior is undefined.

conditional
operator
attempt to modify

Commentary

The discussion on the function-call operator result and its implementation details are applicable here.

function
result
attempt to modify

C90

Wording to explicitly specify this undefined behavior is new in the C99 Standard.

C++

The C++ definition of lvalue is the same as C90, so this wording is not necessary in C++.

lvalue

Coding Guidelines

The code that needs to be written to generate this behavior is sufficiently obscure and unlikely to occur that no guideline recommendation is given here.

Example

```

1  typedef struct {
2      int mem_1;
3      } A_S;
4  extern int glob;
5  extern A_S s_1, s_2;
6
7  void f(void)
8  {
9  int *p_1 = &(((glob == 1) ? s_1 : s_2).mem_1);
10
11  *p_1 = 2; /* Undefined behavior. */
12  }

```

conditional operator
arithmetic result

If both the second and third operands have arithmetic type, the result type that would be determined by the usual arithmetic conversions, were they applied to those two operands, is the type of the result. 1279

Commentary

There is no interaction between the types of the second and third operands and the first operand.

Other Languages

In Java:^[1]

15.25 *If one of the operands is of type T where T is **byte**, **short**, or **char**, and the other operand is a constant expression of type **int** whose value is representable in type T, then the type of the conditional expression is T.*

Coding Guidelines

If developers use the analogy of an **if** statement when reasoning about the conditional operator, they are unlikely to consider the effects of applying the usual arithmetic conversions to the operands (although your author is not aware of a case where a failure to take account of these conversions has resulted in a program fault). However, this may be due to use of the conditional operator being comparatively rare (and these guideline recommendations are not intended to cover rarely occurring construct).

For the purposes of these guideline recommendations, the role of the result is the same as the role of the second and third operands.

if statement
controlling expression
scalar type

guideline recommendations
selecting role
operand matching

If both the operands have structure or union type, the result has that type. 1280

Commentary

Both operands are required to have the same structure or union type.

Common Implementations

The generated machine code may depend on what operation is performed on the result of the conditional operator. For instance, a simple member access `(x ? s1:s2).m` may generate the same machine code as if `(x ? s1.m : s2.m)` had been written. For more complicated cases, an implementation may load the address of the two operands into a register for subsequent indirect accesses.

Coding Guidelines

The form `(x ? s1:s2).m` has the advantage, over `(x ? s1.m : s2.m)`, that a change to the member selected only requires a single source modification (one of the two modifications needed in the latter form may be overlooked). Also the former form requires less effort to recognize as always accessing member `m`. (In the latter form the names of the two selected members needs to be checked.)

conditional operator
1268
structure/union type

If both operands have void type, the result has void type. 1281

Commentary

This is one of the three cases where an operator does not return a value.

Coding Guidelines

If the operands have **void** type, the only affect on the output of the program is through the side effects of their evaluation. Such usage embedded in the visible form of an expression (that is not automatically generated, or the result of nested macro expansions) may be making assumptions about the order in which the operands of an expression are evaluated. This issue is covered by a guideline recommendation.

sequence ??
points
all orderings
give same value

conditional operator
pointer to qualified types

If both the second and third operands are pointers or one is a null pointer constant and the other is a pointer, the result type is a pointer to a type qualified with all the type qualifiers of the types pointed-to by both operands. 1282

Commentary

Specifying that the result has all the qualifiers of both operands requires that a translator make worst-case assumptions about subsequent operations. Type qualifiers on a pointed-to type do not affect the representation or alignment requirements of the pointer type, which means subsequent operators are able to treat either operand in the same way.

pointer
to quali-
fied/unqualified
types

In subsequent C sentences, in this C Standard paragraph, the term *appropriately qualified* refers to *all the type qualifiers* specified here (as is shown by the EXAMPLE).

1287 **EXAMPLE**
?: common pointer
type

1283 Furthermore, if both operands are pointers to compatible types or to differently qualified versions of compatible types, the result type is a pointer to an appropriately qualified version of the composite type;

Commentary

The specification for forming composite types lists a set of properties the result must have. The previous C sentence ensured that the resulting pointed-to type has all the qualifiers of the two operands (two types that are pointers to differently qualified compatible types are not compatible). The composite type will contain at least as much, if not more, information than either of the types separately and will therefore be as restrictive (with regard to type checking).

composite
type

C90

Furthermore, if both operands are pointers to compatible types or differently qualified versions of a compatible type, the result has the composite type;

The C90 wording did not specify that the appropriate qualifiers were added after forming the composite type. In:

```

1  extern int glob;
2  const enum {E1, E2} *p_ce;
3  volatile int *p_vi;
4
5  void f(void)
6  {
7  glob = *((p_e != p_i) ? p_vi : p_ce);
8  }
```

the pointed-to type, which is the composite type of the **enum** and **int** types, is also qualified with **const** and **volatile**.

1284 if one operand is a null pointer constant, the result has the type of the other operand;

Commentary

This places a requirement on the implementation to implicitly convert the operand that is a null pointer constant to the type of the other operand (different pointer types may use different representations for the null pointer). The discussion on the equality operators is applicable here.

null pointer
conversion yields
null pointer
equality
operators
null pointer
constant converted

1285 otherwise, one operand is a pointer to **void** or a qualified version of **void**, in which case the result type is a pointer to an appropriately qualified version of **void**.

Commentary

Although the null pointer constant may be represented by a value having a pointer to **void** type, the previous C sentence takes precedence. The discussion on the equality operators is applicable here.

null pointer
constant
equality
operators
pointer to void

C90

otherwise, one operand is a pointer to **void** or a qualified version of **void**, in which case the other operand is converted to type pointer to **void**, and the result has that type.

C90 did not add any qualifiers to the pointer to **void** type. In the case of the **const** qualifier this difference would not have been noticeable (the resulting pointer type could not have been dereferenced without an explicit cast to modify the pointed-to object). In the case of the **volatile** qualifier this difference may result in values being accessed from registers in C90 while they will be accessed from storage in C99.

C++

The C++ Standard explicitly specifies the behavior for creating a composite pointer type (5.9p2) which is returned in this case.

Coding Guidelines

The coding guideline discussion on the equality operators is applicable here.

Example

```

1  const int *p_ci;
2  volatile int *p_vi;
3
4  void f(void)
5  {
6  const volatile int *p_cvi = ((p_vi != (void *)0) ?
7                               p_ci :
8                               (volatile void *)0); /* Not a null pointer constant. */
9  }
```

equality
operators
pointer to in-
complete type

93) A conditional expression does not yield an lvalue.

1286

Commentary

This footnote points out a consequence of specifications appearing elsewhere in the standard.

```

1  struct {
2      int m1[2];
3      } x, y;
4  int glob;
5
6  void f(void)
7  {
8  (glob ? x : y).m1; /*
9                      * An array not a pointer to the first element. Converting the
10                     * array to a pointer to its first element requires an lvalue.
11                     */
12 }
```

C++

5.16p4 *If the second and third operands are lvalues and have the same type, the result is of that type and is an lvalue.*

Otherwise, the result is an rvalue.

Source developed using a C++ translator may contain instances where the result of the conditional operator appears in an rvalue context, which will cause a constraint violation if processed by a C translator.

```

1  extern int glob;
2
3  void f(void)
4  {
5  short loc_s;
6  int loc_i;
7
8  ((glob < 2) ? loc_i : glob) = 3; /* constraint violation */
9                                // conforming
10 ((glob > 2) ? loc_i : loc_s) = 3; // ill-formed
11 }

```

Common Implementations

Some implementations (e.g., gcc) support, as an extension, a conditional operator yielding an lvalue.

Example

Using indirection, it is possible to assign to objects appearing as operands of a conditional operator.

```

1  (x ? y : z) = 0; /* Constraint violation. */
2  *(x ? &y : &z) = 0; /* Strictly conforming. */

```

1287 **EXAMPLE** The common type that results when the second and third operands are pointers is determined in two independent stages. The appropriate qualifiers, for example, do not depend on whether the two pointers have compatible types.

Given the declarations

```

const void *c_vp;
void *vp;
const int *c_ip;
volatile int *v_ip;
int *ip;
const char *c_cp;

```

the third column in the following table is the common type that is the result of a conditional expression in which the first two columns are the second and third operands (in either order):

c_vp	c_ip	const void *
v_ip	0	volatile int *
c_ip	v_ip	const volatile int *
vp	c_cp	const void *
ip	c_ip	const int *
vp	ip	void *

Commentary

The effect is to maximize the number of type qualifiers and minimize the amount of type representation information (i.e., pointer to **void** is a generic pointer type) in the common type.

C90

This example is new in C99.

EXAMPLE
?: common
pointer type

References

Addison–Wesley, 1996.

1. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*.