

The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones

derek@knosof.co.uk

6.5.3.3 Unary arithmetic operators

Constraints

The operand of the unary + or – operator shall have arithmetic type;

1101

Commentary

The unary – operator is sometimes passed as a parameter in a macro invocation. In those cases where negation of an operand is not required (in the final macro replacement), the unary + operator can be passed as an argument (empty macro arguments cause problems for some preprocessors). The symmetry of having two operators can also simplify the automatic generation of source code. While it would have been possible to permit the unary + operator to have an operand of any type (since it has no effect other than performing the integer promotions on its operand), it is very unlikely that this operator would ever appear in a context that the unary – operator would not also appear in.

C++

The C++ Standard permits the operand of the unary + operator to have pointer type (5.3.1p6).

Coding Guidelines

While applying the unary minus operator to an operand having an unsigned integer type is seen in some algorithms (it can be a more efficient method of subtracting the value from the corresponding `U*_MAX` macro, in `<limits.h>`, and adding one), this usage is generally an oversight by a developer.

Rev 1101.1

The promoted operand of the unary – operator shall not be an unsigned type.

of the ~ operator, integer type;

1102

Commentary

There are algorithms (e.g., in graphics applications) that require the bits in an integer value to be complemented, and processors invariably contain an instruction for performing this operation. Complementing the bits in a floating-point value is a very rarely required operation and processors do not contain such an instruction. This constraint reflects this common usage.

Other Languages

While many languages do not contain an equivalent of the ~ operator, their implementations sometimes include it as an extension.

Coding Guidelines

Some coding guideline documents only recommend against the use of operands having a signed type. The argument is that the representation of unsigned types is defined by the standard, while signed types might have one of several representations. In practice, signed types almost universally have the same representation—two’s complement. However, the possibility of variability of integer representation across processors is not the only important issue here. The ~ operator treats its operand as a sequence of bits, not a numeric value. As such it may be making use of representation information and the guideline recommendation dealing with this issue would be applicable.

two’s com-
plement

represent-??
ation in-
formation
using

of the ! operator, scalar type.

1103

Commentary

The logical negation operator is defined in terms of the equality operator, whose behavior in turn is only defined for scalar types.

! 1113
equivalent to
equality
operators
constraints

!
operand type

C++

The C++ Standard does not specify any requirements on the type of the operand of the `!` operator.

The operand of the logical negation operator `!` is implicitly converted to `bool` (clause 4);

5.3.1p8

But the behavior is only defined if operands of scalar type are converted to `bool`:

An rvalue of arithmetic, enumeration, pointer, or pointer to member type can be converted to an rvalue of type `bool`.

4.12p1

Other Languages

Some languages require the operand to have a boolean type.

Coding Guidelines

The following are two possible ways of thinking about this operator are:

1. As a shorthand form of the `!=` operator in a conditional expression. That is, in the same way the two forms `if (x)` and `if (x == 0)` are equivalent, the two forms `if (!x)` and `if (x != 0)` are equivalent.
2. As a logical negation operator that reverses the state of a boolean value (it can take as its operand a value in either of the possible boolean representation models and map it to the model that uses the 0/1 for its boolean representation).

boolean role

A double negative is very often interpreted as a positive statement in English (e.g., “It is not unknown for double negatives to occur in C source”). The same semantics that apply in C. However, in some languages (e.g., Spanish) a double negative is interpreted as making the statement more negative (this usage does occur in casual English speech, e.g., “you haven’t seen nothing yet”, but it is rare and frowned on socially^[1]).

The token `!` is commonly called the *not* operator. This term is a common English word whose use in a sentence is similar to its use in a C expression. Through English language usage the word *not*, or an equivalent form, can appear as part of an identifier spelling (e.g., `not_finished`, `no_signal`, or `unfinished`). The use of such identifiers in an expression can create a double negative (e.g., `!not_finished` or `not_finished != 1`).

English
negation

A simple expression containing a double negation is likely to require significantly more cognitive resources to comprehend than a one that does not. Changing the semantic associations of an identifier from (those implied by) `not_finished` to `finished` would require that occurrences of `not_finished` be changed to `!finished` (plus associated changes to any appearances of the identifier as the operand of the `!` or the equality operators).

Calculating the difference in cognitive cost/benefit between using an identifier spelling that represents a negated form and one that does not requires information on a number of factors. For instance, whether any double negative forms actually appear in the source, the extent to which the *not* spelling form provides a good fit to the application domain, and any cognitive cost differences between the alternative forms `not_finished` and `!finished`. Given the uncertainty in the cost/benefit analysis no guideline recommendation is given here.

Table 1103.1: Occurrence of the unary `!` operator in various contexts (as a percentage of all occurrences of this operator and the percentage of all occurrences of the given context that contains this operator). Based on the visible form of the `.c` files.

Context	% of !	% of Contexts
<code>if</code> control-expression	91.0	17.4
<code>while</code> control-expression	2.3	8.2
<code>for</code> control-expression	0.3	0.7
<code>switch</code> control-expression	0.0	0.0
other contexts	6.4	—

Semantics

The result of the unary + operator is the value of its (promoted) operand.

1104

Commentary

Also, the result of the unary + is not an lvalue.

Other Languages

Many languages do not include a unary + operator.

Common Implementations

Early versions of K&R C treated `p+=2` as being equivalent to `p+=2`.^[3]

Coding Guidelines

One use of the unary + operator is to remove the lvalue-ness of an object appearing as an argument in a macro invocation. This usage guarantees that the object passed as an argument cannot be modified or have its address taken.

Use of the unary + operator is very rare in developer-written source. If it appears immediately after the = operator in existing code, the possible early K&R interpretation might be applicable. The usage is now sufficiently rare that a discussion on whether to do nothing, replace every occurrence by the sequence `+=`, introduce a separating white-space character, parenthesize the value being assigned, or do something else is not considered worthwhile.

Example

```

1  /*
2   * Sum up to three values. Depending on the arguments
3   * passed + is either a unary or binary operator.
4   */
5  #define ADD3(a, b, c)    (a + b + c + 0)
6
7      ADD3(1, 2, 3) => (1 + 2 + 3 + 0)
8      ADD3(1, 2, ) => (1 + 2 +   + 0)
9      ADD3(1, , 3) => (1 +   + 3 + 0)
10     ADD3(1, , ) => (1 +   +   + 0)
11     ADD3( , , ) => (   +   +   + 0)

```

The integer promotions are performed on the operand, and the result has the promoted type.

1105

Commentary

The two contexts in which the integer promotions would not be performed, unless the unary + operator is applied, are the right operand of a simple assignment and the operand of the `sizeof` operator.

simple as-
assignment
sizeof
result of

The result of the unary - operator is the negative of its (promoted) operand.

1106

Commentary

The expression `-x` is not always equivalent to `0-x`; for instance, if `x` has the value `0.0`, the results will be `-0.0` and `0.0`, respectively.

Common Implementations

Most processors include a single instruction that performs the negation operation. On many RISC processors this instruction is implemented by the assembler using an alias of the subtract instruction (for integer operands only). On such processors there is usually a register hardwired to contain the value zero (the IBM/Motorola PowerPC^[4] does not); subtracting the operand from this register has the required effect. For IEC 60559 floating-point representations, the negation operator simply changes the value of the sign bit.

Coding Guidelines

If the operand has an unsigned type, the result will always be a positive or zero value. This issue is discussed elsewhere.

1101.1 unary minus
unsigned operand

Example

The expression `-1` is the unary `-` operator applied to the integer constant `1`.

1107 The integer promotions are performed on the operand, and the result has the promoted type.

Commentary

Because unary operators have a single operand, it is not necessary to perform the usual arithmetic conversions (the integer promotions are performed on the operands of the unary operators for the same reason).

usual arith-
metic conver-
sions
integer pro-
motions

Coding Guidelines

The integer promotions may convert an unsigned type to a signed type. However, this can only happen if the signed type can represent all of the values of the unsigned type. This is reflected in the guideline recommendation for unsigned types.

signed
integer
represent all
unsigned integer
values
1101.1 unary minus
unsigned operand

1108 The result of the `~` operator is the bitwise complement of its (promoted) operand (that is, each bit in the result is set if and only if the corresponding bit in the converted operand is not set).

Commentary

The term *bitwise not* is sometimes used to denote this operator (it is sometimes also referred to by the character used to represent it, tilde). Because its use is much less frequent than logical negation, this term is rarely shortened.

bitwise com-
plement
result is
1111 logical
negation
result is

Common Implementations

Most processors have an instruction that performs this operation. An alternative implementation is to exclusive-or the operand with an all-bits-one value (containing the same number of bits as the promoted type). The Unisys A Series^[5] uses signed magnitude representation. If the operand has an unsigned type, the sign bit in the object representation (which is treated as a padding bit) is not affected by the bitwise complement operator. If the operand has a signed type, the sign bit does take part in the bitwise complement operation.

Example

```
1  V &= ~BITS; /* Clear the bits in V that are set in BITS. */
```

1109 The integer promotions are performed on the operand, and the result has the promoted type.

Commentary

Performing the integer promotions can increase the number of representation bits used in the value that the complement operator has to operate on.

Coding Guidelines

The impact of the integer promotions on the value of the result is sometimes overlooked by developers. During initial development these oversights are usually quickly corrected (the results differ substantially from the expected range of values and often have a significant impact on program output). Porting existing code to a processor whose `int` size differs from the original processor on which the code executed can cause latent differences in behavior to appear. For instance, if `sizeof(int)==sizeof(short)` on the original processor, then any integer promotions on operands having type `short` would not increase the number of bits in the value representation and a program may have an implicit dependency on this behavior occurring. Moving to a processor where `sizeof(int) > sizeof(short)` may require modifications to explicitly enforce this

dependency. The issues involved in guideline recommendations that only deliver a benefit when a program is ported to a processor whose integer widths are different from the original processor are discussed elsewhere.

Example

```

1  unsigned char uc = 2;
2  signed char sc = -1;
3
4  void f(void)
5  {
6  /*
7   * Using two's complement notation the value 2 is represented
8   * in the uc object as the bit pattern 00000010
9   * If int is represented using 16 bits, this value is promoted
10  * to the representation 0000000000000010
11  * after being complemented 111111111111101 is the result
12  */
13      ~uc          ;
14  /*
15   * Using two's complement notation the value -1 is represented
16   * in the sc object as the bit pattern 11111110
17   * If int is represented using 16 bits, this value is promoted
18   * to the representation 1111111111111110
19   * after being complemented 0000000000000010 is the result
20  */
21      ~sc          ;
22  }
```

If the promoted type is an unsigned type, the expression `~E` is equivalent to the maximum value representable in that type minus `E`. 1110

Commentary

This is the standard pointing out an equivalence that holds for the binary representation of integer values.

C++

The C++ Standard does not point out this equivalence.

Coding Guidelines

The issues surrounding the use of bitwise operations to perform equivalent arithmetic operations is discussed elsewhere.

The result of the logical negation operator `!` is 0 if the value of its operand compares unequal to 0, 1 if the value of its operand compares equal to 0. 1111

Commentary

The term *not* (or *logical not*) is often used to denote this operator. The much less frequently used operator, bitwise complement, takes the longer name.

C++

5.3.1p8 *its value is **true** if the converted operand is **false** and **false** otherwise.*

This difference is only visible to the developer in one case. In all other situations the behavior is the same false and true will be converted to 0 and 1 as-needed.

Other Languages

Languages that support a boolean data type usually specify **true** and **false** return values for these operators.

Common Implementations

The implementation of this operator often depends on the context in which it occurs. The machine code generated can be very different if the result value is used to decide the control flow (e.g., it is the final operation in the evaluation of a controlling expression) than if the result value is the operand of further operators. In the control flow case an actual value of 0 or 1 is not usually required. On many processors loading a value from storage into a register will set various bits in a conditional flags register (these flag bit settings usually specify some relationship between the value loaded and zero— e.g., equal to, less than, etc.). A processor's conditional branch instructions use the current settings of combinations of these bits to decide whether to take the branch or not. When the result is used as an operand in further operations, a 0 or 1 value is needed; the generated machine code is often more complex. A common solution is the following pseudo machine code sequence (which leaves the result in REG_1):

```

1  load REG_1, 0
2  load REG_2, Operand
3  Branch_over_next_instr_if_last_load_not_zero
4  load REG_1, 1

```

The instruction `Branch_over_next_instr_if_last_load_not_zero` may be a single instruction, or several instructions.

Coding Guidelines

While the result is specified in numeric terms, most occurrences of this operator are as the top-level operator in a controlling expression (see Usage below). These contexts are usually considered in boolean rather than numeric terms.

if statement
operand compare
against 0

boolean role

1112 The result has type **int**.

Commentary

The C90 Standard did not include a boolean data type and C99 maintains compatibility with this existing definition.

C++

logical negation
result type

*The type of the result is **bool**.*

5.3.1p8

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.

Coding Guidelines

The possible treatment of the result of logical, and other operators as having a boolean type, is discussed elsewhere.

boolean role

1113 The expression **!E** is equivalent to **(0==E)**.

Commentary

In the case of pointers the 0 is equivalent to the null pointer constant. In the case of E having a floating-point type, the constant 0 will be converted to 0.0. The standard is being idiosyncratic in expressing this equivalence (the form $(E==0)$ is more frequent, by a factor of 28, in existing code).

!
equivalent to

null pointer
constant

C++

There is no explicit statement of equivalence given in the C++ Standard.

Common Implementations

logical
negation
result is

Both forms occur sufficiently frequently, in existing code, that translator implementors are likely to check for the context in which they occur in order to generate the appropriate machine code.

Coding Guidelines

Both forms are semantically identical, and it is very likely that identical machine code will be generated for both of them. (The sometimes-heard rationale of visual complexity of an expression being equated to inefficiency of program execution is discussed elsewhere.) Because of existing usage (the percentage occurrence of both operators in the visible source is approximately comparable) developers are going to have to learn to efficiently comprehend expressions containing both operators. Given this equivalence and existing practice, is there any benefit to be gained by a guideline recommending one form over the other? Both have their disadvantages:

- The ! character is not frequently encountered in formal education, and it may be easy to miss in a visual scan of source (no empirical studies using the ! character are known to your author).
- The equality operator, ==, is sometimes mistyped as an assignment operator, =.

Perhaps the most significant distinguishing feature of these operators is the conceptual usage associated with their operand. If this operand is primarily thought about in boolean terms, the conceptually closest operator is !. If the operand is thought of as being arithmetic, the conceptually closest operator is ==.

A number of studies have investigated the impact of negation in reasoning tasks. In natural languages negation comes in a variety of linguistic forms (e.g., “no boys go to class”, “few boys go to class”, “some boys go to class”) and while the results of these studies^[2] of human performance using these forms may be of interest to some researchers, they don’t have an obvious mapping to C language usage (apart from the obvious one that negating a sentence involves an additional operator, the negation, which itself needs cognitive resources to process).

Usage

The visible form of the .c files contain 95,024 instances of the operator ! (see Table ?? for information on punctuation frequencies) and 27,008 instances of the token sequence == 0 (plus 309 instances of the form == 0x0). Integer constants appearing as the operand of a binary operator occur 28 times more often as the right operand than as the left operand.

84) Thus, **&*E** is equivalent to **E** (even if **E** is a null pointer), and **&(E1[E2])** to **((E1)+(E2))**. 1114

Commentary

&* This footnote sentence should really have been referenced from a different paragraph, where these equivalences are discussed.

C90

This equivalence was not supported in C90, as discussed in the response to DR #012, #076, and #106.

C++

At the moment the C++ Standard specifies no such equivalence, explicitly or implicitly. However, this situation may be changed by the response to DR #232.

***&** It is always true that if **E** is a function designator or an lvalue that is a valid operand of the unary **&** operator, ***&E** is a function designator or an lvalue equal to **E**. 1115

Commentary

This statement can be deduced from the specifications of the two operators concerned.

visually com-
pact code
efficiency belief
punctuator
syntax

controlling
expression
if statement

boolean role

footnote
84

*&

-
- 1116 If $*P$ is an lvalue and T is the name of an object pointer type, $*(T)P$ is an lvalue that has a type compatible with that to which T points.

Commentary

The result of the cast operator is not an lvalue. However, if the operand is a pointer, the pointed-to object does not lose its lvalue-ness. This sentence simply points out the type of the result of the operations and its lvalue-ness; it does not give any additional semantics to the cast or dereference.

footnote
85**C++**

The C++ Standard makes no such observation.

- 1117 Among the invalid values for dereferencing a pointer by the unary $*$ operator are a null pointer, an address inappropriately aligned for the type of object pointed to, and the address of an object after the end of its lifetime.

Commentary

This list contains some examples of invalid values that may appear directly in the source; it is not exhaustive (another example is dereferencing a pointer-to function). The invalid values may also be the result of an operation that has undefined behavior. For instance, using pointer arithmetic to create an address that does not correspond to any physical memory location supported by a particular computing system. (In virtual memory systems this case would correspond to an unmapped address.)

C90

The wording in the C90 Standard only dealt with the address of objects having automatic storage duration.

C++

The C++ Standard does not call out a list of possible invalid values that might be dereferenced.

Other Languages

Most other languages do not get involved in specifying such low-level details, although their implementations invariably regard the above values as being invalid.

Common Implementations

A host's response to an attempt to use an invalid pointer value will usually depend on the characteristics of the processor executing the program image. In some cases an exception which can be caught by the program, may be signaled. The extent to which a signal handler can recover from the exception will depend on the application logic and the type of invalid valid dereference.

On many implementations the `offsetof` macro expands to an expression that dereferences the null pointer.

Coding Guidelines

One of the ways these guideline recommendations attempt to achieve their aim is to attempt to prevent invalid values from being created in the first place.

coding
guidelines
background to

References

1. D. Biber, S. Johansson, G. Leech, S. Conrad, and E. Finegan. *Longman Grammar of Spoken and Written English*. Pearson Education, 1999.
2. M. A. Just and P. A. Carpenter. Comprehension of negation with quantification. *Journal of Verbal Learning and Verbal Behavior*, 10:244–253, 1971.
3. B. W. Kernighan. Programming in C-A tutorial. Technical Report ???, Bell Laboratories, Aug. ???
4. Motorola, Inc. *PowerPC 601 RISC Microprocessor User's Manual*. Motorola, Inc, 1993.
5. Unisys Corporation. *C Programming Reference Manual, Volume 1: Basic Implementation*. Unisys Corporation, 8600 2268-203 edition, 1998.