

# **The New C Standard** (Excerpted material)

---

**An Economic and Cultural Commentary**

**Derek M. Jones**

derek@knosof.co.uk

### 6.3.1.8 Usual arithmetic conversions

operators cause conversions

Many operators that expect operands of arithmetic type cause conversions and yield result types in a similar way.

#### Commentary

The operators cause the conversions in the sense that the standard specifies that they do. There is no intrinsic reason why they have to occur. The Committee could have omitted any mention of operators causing conversions, but would then have had to enumerate the behavior for all possible pairs of operand types. The usual arithmetic conversions are the lesser of two evils (reducing the number of different arithmetic types did not get a look in).

FLT\_EVAL\_METHOD

The operators referred to here could be either unary or binary operators. In the case of the unary operators the standard specifies no conversion if the operand has a floating type (although the implementation may perform one, as indicated by the value of the FLT\_EVAL\_METHOD macro). However, there is a conversion (the integer promotions) if the operand has an integer type.

#### Other Languages

Java *Numeric promotion is applied to the operands of an arithmetic operator.*

All languages that support more than one type need to specify the behavior when an operator is given operands whose types are not the same. Invariably the type that can represent the widest range of values tends to be chosen.

#### Coding Guidelines

Readers needing to comprehend the evaluation of an expression in detail, need to work out which conversions, if any, are carried out as a result of the usual arithmetic conversions (and the integer promotions). This evaluation requires readers to apply a set of rules. Most developers are not exposed on a regular basis to a broad combination of operand type pairs. Without practice, developers' skill in deducing the result of the usual arithmetic conversions on the less commonly encountered combinations of types will fade away (training can only provide a temporary respite). The consequences of this restricted practice is that developers never progress past the stage of remembering a few specific cases.

If a single integer type were used, the need for developers to deduce the result of the usual arithmetic conversions would be removed. This is one of the primary rationales behind the guideline recommendation that only a single integer type be used.

object ??  
int type only  
operand  
convert automatically  
  
typedef  
is synonym

As with the integer promotions, there is a commonly seen, incorrect assumption made about the usual arithmetic conversions. This incorrect assumption is that they do not apply to objects defined using typedef names. In fact a typedef name is nothing more than a synonym. How an object is defined makes no difference to how the usual arithmetic conversions are applied. Pointing out this during developer training may help prevent developers from making this assumption.

common real type The purpose is to determine a *common real type* for the operands and result.

#### Commentary

This defines the term *common real type*. The basic idea is that both types are converted to the real type capable of holding the largest range of values (mixed signed/unsigned types are the fly in the ointment). In the case of integer types, the type with the largest rank is chosen. For types with the same rank, the conversions are unsigned preserving.

Many processor instructions operate on values that have the same type (or at least are treated as if they did). C recognizes this fact and specifies how the operands are to be converted. This process also removes the need to enumerate the behavior for all possible permutations of operand type pairs.

**C90**

The term *common real type* is new in C99; the equivalent C90 term was *common type*.

**Other Languages**

The need to convert the operands to the same type is common to languages that have a large number of different arithmetic types. There are simply too many permutations of different operand type pairs to want to enumerate, or expect developers to remember, the behavior for all cases. Some languages (e.g., PL/1) try very hard to implicitly convert the operands to the same type, without the developer having to specify any explicit conversions. Other languages (e.g., Pascal) enumerate a small number of implicit conversions and require the developer to explicitly specify how any other conversions are to be performed (by making it a constraint violation to mix operands having other type pairs).

**Common Implementations**

On some processors arithmetic operations can produce a result that is wider than the original operands—for instance multiplying two 16-bit values to give a 32-bit result. In these cases C requires that the result be converted back to the same width as the original operands. Such processor instructions also offer the potential optimization of not performing the widening to 32 bits before carrying out the multiplication. Other operand/result sizes include 8/16 bits.

**Coding Guidelines**

The C90 term *common type* is sometimes used by developers. Given that few developers will ever use complex types it is unlikely that there will be a general shift in terminology usage to the new C90 term.

Some developers make the incorrect assumption that if the two operands already have a common type, the usual arithmetic conversions are not applied. They forget about the integer promotions. For instance, two operands of type **short** are both promoted to **int** before the usual arithmetic conversions are applied.

Pointing out to developers that the integer promotions are always performed is a training issue (explicitly pointing out this case may help prevent developers from making this assumption). Following the guideline on using a single integer type ensures that incorrect developer assumptions about this promotion do not affect the intended behavior.

?? object  
int type only

**Example**

```

1  unsigned char c1, c2, c3;
2
3  int f(void)
4  {
5  /*
6   * Experience shows that many developers expect the following additional and
7   * relational operations to be performed on character types, rather than int.
8   */
9   if ((c1 + c2) > c3)
10     c1 = 3;
11 }
```

704 For the specified operands, each operand is converted, without change of type domain, to a type whose corresponding real type is the common real type.

arithmetic  
conversions  
type domain  
unchanged

**Commentary**

This requirement means that the usual arithmetic conversions leave operands that have complex types as complex types and operands that have real types remain real types for the purposes of performing the operation. As well as saving the execution-time overhead on the conversion and additional work for the operator, this behavior helps prevent some unexpected results from occurring. The following example first shows the effects of a multiplication using the C99 rules:

complex  
types  
real types

$$2.0 * (3.0 + \infty i) \Rightarrow 2.0 * 3.0 + 2.0 * \infty i \quad (704.1)$$

$$\Rightarrow 6.0 + \infty i \quad (704.2)$$

The result,  $6.0 + \infty i$ , is what the developer probably expected. Now assume that the usual arithmetic conversions were defined to change the type domain of the operands, a real type having  $0.0i$  added to it when converting to a complex type. In this case, we get:

$$2.0 * (3.0 + \infty i) \Rightarrow (2.0 + 0.0i) * (3.0 - \infty i) \quad (704.3)$$

$$(2.0 * 3.0 - 0.0 * \infty) + (2.0 * \infty + 0.0 * 3.0)i \Rightarrow NaN + \infty i \quad (704.4)$$

The result,  $NaN + \infty i$ , is probably a surprise to the developer. For imaginary types:

$$2.0i * (\infty + 3.0i) \quad (704.5)$$

leads to  $NaN + \infty i$  in one case and  $-6.0 + \infty i$  in the C99 case.

### C90

Support for type domains is new in C99.

### C++

The term *type domain* is new in C99 and is not defined in the C++ Standard.

The template class `complex` contain constructors that can be used to implicitly convert to the matching complex type. The operators defined in these templates all return the appropriate complex type.

C++ converts all operands to a complex type before performing the operation. In the above example the C result is  $6.0 + \infty i$ , while the C++ result is  $NaN + \infty i$ .

### Other Languages

Fortran converts the operand having the real type to a complex type before performing any operations.

### Coding Guidelines

Support for complex types is new in C99, and at the time of this writing there is very little practical experience available on the sort of mistakes that developers make with it. An obvious potential misunderstanding would be to assume that if one operand has a complex type then the other operand will also be converted to the corresponding complex type. This thinking fits the pattern of the other conversions, but would be incorrect. Based on the same rationale as that given in the previous two sentences, the solution is training, not a guideline recommendation.

---

Unless explicitly stated otherwise, the common real type is also the corresponding real type of the result, whose type domain is the type domain of the operands if they are the same, and complex otherwise. 705

### Commentary

The only place where the standard *explicitly stated otherwise* is in the discussion of imaginary types in annex G. Support for such types, by an implementation, is optional. When one operand has a complex type and the other operand does not, the latter operand is not converted to a different type domain (although its real type may be changed by a conversion), so there is no common arithmetic type, only a common real type.

### C++

The `complex` specializations (26.2.3) define conversions for **float**, **double** and **long double** to complex classes. A number of the constructors are defined as explicit, which means they do not happen implicitly, they can only be used explicitly. The effect is to create a different result type in some cases.

In C++, if the one operand does not have a complex type, it is converted to the corresponding complex type, and the result type is the same as the other operand having complex type. See footnote 51.

## Other Languages

The result type being the same as the final type of the operands is true in most languages.

706 This pattern is called the *usual arithmetic conversions*:

usual arithmetic conversions

### Commentary

This defines the term *usual arithmetic conversions*. There are variations on this term used by developers; however, *arithmetic conversions* is probably the most commonly heard.

### Other Languages

The equivalent operations in Java are known as the *Binary Numeric Promotion*.

### Common Implementations

In some cases the standard may specify two conversions— an integer promotion followed by an arithmetic conversion. An implementation need not perform two conversions. The as-if rule can be used to perform a single conversion (if the target processor has such an instruction available to it) provided the final result is the same.

### Coding Guidelines

The concept of usual arithmetic conversions is very important in any source that has operands of different types occurring together in the same expression. The guideline recommendation specifying use of a single integer type is an attempt to prevent this from occurring. The general issue of whether any operand that is converted should be explicitly cast, rather than implicitly converted, is discussed elsewhere.

?? object  
int type only  
operand  
convert automati-  
cally

707 First, if the corresponding real type of either operand is **long double**, the other operand is converted, without change of type domain, to a type whose corresponding real type is **long double**.

arithmetic conversions  
long double

### Commentary

If the other operand has an integer type, the integer promotions are not first applied to it.

integer pro-  
motions

### Other Languages

The operands having floating types takes precedence over those having integer types in all languages known to your author.

### Common Implementations

Processors often support an instruction for converting a value having a particular integer type to a floating representation. Implementations have to effectively promote integer values to this type before they can be converted to a floating type. Since the integer type is usually the widest one available, there is rarely an externally noticeable difference.

### Coding Guidelines

If both operands have a floating type, there is no loss of precision on the conversion.

When an integer is converted to a real type, there may be some loss of precision. The specific case of integer-to-floating conversion might be considered different from integer-to-integer and floating-to-floating conversions for a number of reasons:

float  
promoted to  
double or long  
double  
integer  
conversion to  
floating

- There is a significant change of representation.
- The mixing of operands having an integer and floating type is not common (the implication being that more information will be conveyed by an explicit cast in this context than in other contexts).
- The decision to declare an object to have a floating type is a much bigger one than giving it an integer type (experience suggests that, once this decision is made, it is much less likely to change than had an integer type has been chosen, for which there are many choices; so the maintenance issue of keeping explicit conversions up to date might be a smaller one than for integer conversions).

integer  
conversion to  
floating

arithmetic conversions double

Otherwise, if the corresponding real type of either operand is **double**, the other operand is converted, without change of type domain, to a type whose corresponding real type is **double**. 708

### Commentary

FLT\_EVAL\_METHOD

The operand may already be represented during expression evaluation to a greater precision than the type **double** (perhaps even type **long double**). However, from the point of view of type compatibility and other semantic rules, its type is still **double**.

### Coding Guidelines

FLT\_EVAL\_METHOD

The issues associated with the operands being represented to a greater precision than denoted by their type are discussed elsewhere.

If the other operand has an integer type and is very large there is a possibility that it will not be exactly represented in type **double**, particularly if this type only occupies 32 bits (a single-precision IEC representation, the minimum required by the C Standard). The issues associated with of exact representation are discussed elsewhere.

integer conversion to floating

arithmetic conversions float

Otherwise, if the corresponding real type of either operand is **float**, the other operand is converted, without change of type domain, to a type whose corresponding real type is **float**.<sup>51)</sup> 709

### Commentary

arithmetic conversions double 708

As was pointed out in the previous sentence, when one of the operands has type **double**, the operand may be represented to a greater precision during expression evaluation.

### Common Implementations

Many implementations perform operations on floating-point operands using the type **double** (either because the instruction that loads the value from storage converts them to this form, or because the translator generates an instruction to implicitly convert them). The Intel x86 processor<sup>[1]</sup> implicitly converts values to an internal 80-bit form when they are loaded from storage.

arithmetic conversions integer promotions

Otherwise, the integer promotions are performed on both operands. 710

### Commentary

integer promotions

At this point the usual arithmetic conversions become a two-stage process. Whether an implementation actually performs two conversions, or can merge everything into a single conversion, is an optimization issue that is not of concern to the developer. Performing the integer promotions reduces the number of permutations of operand types that the standard needs to specify behavior for.

### Other Languages

Most languages have a small number of arithmetic types, and it is practical to fully enumerate the behavior in all cases, making it unnecessary to define an intermediate step.

### Coding Guidelines

integer promotions operand same type no further conversion object ?? int type only 712

Developers sometimes forget about this step (i.e., they overlook the fact that the integer promotions are performed). The rule they jump to is often the one specifying that there need be no further conversions if the types are the same. There is no obvious guideline recommendation that can render this oversight (or lack of knowledge on the part of developers) harmless. If the guideline recommendation specifying use of a single integer type is followed these promotions will have no effect.

### Usage

Usage information on integer promotions is given elsewhere (see Table ??).

arithmetic conversions integer types

Then the following rules are applied to the promoted operands:

711

### Commentary

The C90 Standard enumerated each case based on type. Now that that concept of rank has been introduced, it is possible to specify rules in a more generalized form. This generalization also handles any extended integer types that an implementation may provide.

### C++

The rules in the C++ Standard appear in a bulleted list of types with an implied sequential application order.

### Common Implementations

In those implementations where the two types have the same representation no machine code, to perform the conversion at execution time, need be generated.

While the standard may specify a two-stage conversion process, some processors support instructions that enable some of the usual arithmetic conversions to be performed in a single instruction.

### Coding Guidelines

Developers have a lot of experience in dealing with the promotion and conversion of the standard integer types. In the C90 Standard the rules for these conversions were expressed using the names of the types, while C99 uses the newly created concept of rank. Not only is the existing use of extended integer types rare (and so developers are unlikely to be practiced in their use), but conversions involving them use rules based on their rank (which developers will have to deduce). At the time of this writing there is insufficient experience available with extended integer types to be able to estimate the extent to which the use of operands having some extended type will result in developers incorrectly deducing the result type. For this reason these coding guidelines sections say nothing more about the issue (although if the guideline recommendation specifying use of a single integer type is followed the promotion of extended integer types does not become an issue).

standard  
integer types  
conversion  
rank

?? object  
int type only

For one of the operands, these conversions can cause either its rank to be increased, its sign changed, or both of these things. An increase in rank is unlikely to have a surprising affect (unless performance is an issue, there can also be cascading consequences in that the result may need to be converted to a lesser rank later). A change of sign is more likely to cause a surprising result to be generated (i.e., a dramatic change in the magnitude of the value, or the reversal of its signedness). While 50% of the possible values an object can hold may produce a surprising result when it is converted between signed and unsigned, in practice the values that occur are often small and positive (see Table ??).

Experience has shown that mixing operands having signed and unsigned types in expressions has a benefit. Despite a great deal of effort, by a large number of people, no guideline recommendation having a cost less than that incurred by the occasional surprising results caused by the arithmetic conversions has been found.

### Usage

Usage information on implicit conversions is given elsewhere (see Table ??).

712 If both operands have the same type, then no further conversion is needed.

### Commentary

Both operands can have the same type because they were declared to have that type, the integer promotions converted them to that type, or because they were explicitly cast to those types.

### C90

For language lawyers only: A subtle difference in requirements exists between the C90 and C99 Standard (which in practice would have been optimized away by implementations). The rules in the C90 wording were ordered such that when two operands had the same type, except when both were type **int**, a conversion was required. So the type **unsigned long** needed to be converted to an **unsigned long**, or a **long** to a **long**, or an **unsigned int** to an **unsigned int**.

### Coding Guidelines

Many developers incorrectly assume this statement is true for all integer types, even those types whose rank is less than that of type **int**. They forget, or were never aware, that the integer promotions are always performed.

integer pro-  
motions

operand  
same type  
no further  
conversion

object ??  
int type only

The guideline recommendation dealing with a single integer type aims to ensure that both operands always have the same type.

Otherwise, if both operands have signed integer types or both have unsigned integer types, the operand with the type of lesser integer conversion rank is converted to the type of the operand with greater rank.

713

### Commentary

rank  
relative ranges

If the operands are both signed, or both unsigned, it is guaranteed that their value can be represented in the integer type of greater rank. This rule preserves the operand value and its signedness.

### Other Languages

Those languages that support different size integer types invariably also convert the *smaller* integer type to the *larger* integer type.

### Common Implementations

This conversion is usually implemented by sign-extending the more significant bits of the value for a signed operand, and by zero-filling for an unsigned operand. Many processors have instructions that will sign-extend or zero-fill a byte, or half word value, when it is loaded from storage into a register. Some processors have instructions that take operands of different widths; for instance, multiplying a 16-bit value by an 8-bit value. In these cases it may be possible to optimize away the conversion and to use specific instructions that return the expected result.

signed integer  
converted to  
unsigned

Otherwise, if the operand that has unsigned integer type has rank greater or equal to the rank of the type of the other operand, then the operand with signed integer type is converted to the type of the operand with unsigned integer type.

714

### Commentary

This rule could be said to be unsigned preserving. The behavior for conversion to an unsigned type is completely specified by the standard, while the conversion to a signed type is not.

### Other Languages

Languages that have an unsigned type either follow the same rules as C in this situation, or require an explicit conversion to be specified (e.g., Modula-2).

### Coding Guidelines

A signed operand having a negative value will be converted to a large unsigned value, which could be a surprising result. A guideline recommending against operands having negative values, in this case, might be considered equivalent to one recommending against faults— i.e., pointless. The real issue is one of operands of different signed'ness appearing together within an expression. This issue is discussed elsewhere.

### Example

The width of the integer types is irrelevant. The conversion is based on rank only. The unsigned type has greater rank and the matching rule has to be applied.

```

1  signed int si;
2  unsigned long ul;
3
4  void f(void)
5  {
6  /* ... */ si + ul;
7  }
```

signed integer  
represent all  
unsigned integer  
values

Otherwise, if the type of the operand with signed integer type can represent all of the values of the type of the operand with unsigned integer type, then the operand with unsigned integer type is converted to the type of the operand with signed integer type.

715

### Commentary

This rule is worded in terms of representable values, not rank. It preserves the value of the operand, but not its signedness. While a greater range of representable values does map to greater rank, the reverse is not true. It is possible for types that differ in their rank to have the same range of representable values. The case of same representation is covered in the following rule.

### Other Languages

Most languages have a single integer type. Those that support an unsigned type usually only have one of them. A rule like this one is very unlikely to be needed.

### Common Implementations

In this case an implementation need not generate any machine code. The translator can simply regard the value as having the appropriate signed integer type.

### Example

In the following addition two possible combinations of representation yield the conversions:

1. 16-bit unsigned int plus 32-bit long  $\Rightarrow$  32-bit long
2. 32-bit unsigned int plus 32-bit long  $\Rightarrow$  this rule not matched

```

1  unsigned int ui;
2  signed long sl;
3
4  void f(void)
5  {
6  /* ... */ ui + sl;
7  }
```

---

716 Otherwise, both operands are converted to the unsigned integer type corresponding to the type of the operand with signed integer type.

### Commentary

This rule is only reached when one operand is an unsigned integer type whose precision is greater than or equal to the precision of a signed type having a greater rank. This can occur when two standard types have the same representation, or when the unsigned type is an extended unsigned integer type (which must have a lesser rank than any standard integer type having the same precision). In this rule the result type is different from the type of either of the operands. The unsignedness is taken from one operand and the rank from another to create the final result type. This rule is the value-driven equivalent of the rank-driven rule specified previously.

714 signed  
integer  
converted to  
unsigned

### Coding Guidelines

This result type of this rule is likely to come as a surprise to many developers, particularly since it is dependent on the width and precision of the operands (because of the previous rule). Depending on the representation of integer types used, the previous rule may match on one implementation and this rule match on another. The result of the usual arithmetic conversions, in some cases, is thus dependent on the representation of integer types.

A previous rule dealt with operands being converted from a signed to an unsigned type. For the rule specified here, there is the added uncertainty of the behavior depending on the integer representations used by an implementation.

714 signed  
integer  
converted to  
unsigned

**Example**

```

1  #include <stdint.h>
2
3  uint_fast32_t fast_ui;
4  signed long sl;
5  unsigned int ui;
6
7  void f(void)
8  {
9  /*
10 * 16-bit unsigned int + 32-bit signed long -> 32-bit signed long
11 * 32-bit unsigned int + 32-bit signed long -> 32-bit unsigned long
12 */
13 ui + sl;
14
15 fast_ui + sl; /* unsigned long result. */
16 }

```

---

The values of floating operands and of the results of floating expressions may be represented in greater precision and range than that required by the type; 717

**Commentary**

[FLT\\_EVAL\\_METHOD](#) This issue is discussed under the `FLT_EVAL_METHOD` macro.

**Other Languages**

Most languages do not get involved in specifying this level of detail (although Ada explicitly requires the precision to be as defined by the types of the operands).

**Coding Guidelines**

[FLT\\_EVAL\\_METHOD](#) Additional precision does not necessarily make for more accurate results, or more consistent behavior (in fact often the reverse). The issues involved are discussed in more detail under the `FLT_EVAL_METHOD` macro.

---

the types are not changed thereby.<sup>52)</sup> 718

**Commentary**

Any extra precision that might be held by the implementation does not affect the semantic type of the result.

---

51) For example, addition of a `double _Complex` and a `float` entails just the conversion of the `float` operand to `double` (and yields a `double _Complex` result). 719

**Commentary**

Similarly, the addition of operands having types `float _Complex` and `double` entails conversion of the complex operand to `double _Complex`.

**C90**

Support for complex types is new in C99

**C++**

The conversion sequence is different in C++. In C++ the operand having type `float` will be converted to `complexfloat` prior to the addition operation.

```

1  #include <complex.h> // the equivalent C++ header
2
3  float complex fc; // std::complex<float> fc; this is the equivalent C++ declaration

```

footnote  
51

```
4 double d;
5
6 void f(void)
7 {
8 fc + d /* Result has type double complex. */
9        // Result has type complex<float>.
10      ;
11 }
```

---

720 52) The cast and assignment operators are still required to perform their specified conversions as described in 6.3.1.4 and 6.3.1.5.

footnote  
52

### Commentary

To be exact an implementation is required to generate code that behaves as if the specified conversions were performed. An implementation can optimize away any conversion if it knows that the current value representation is exactly the same as the type converted to (implicitly or explicitly). When an implementation carries out floating-point operations to a greater precision than required by the semantic type, the cast operator is guaranteed to return a result that contains only the precision required by the type converted to. An explicit cast provides a filter through which any additional precision cannot pass.

The assignment operator copies a value into an object. The object will have been allocated sufficient storage to be able to hold any value representable in its declared type. An implementation therefore has to scrape off any additional precision which may be being held in the value (i.e., if the register holding it has more precision), prior to storing a value into an object.

Passing an argument in a function call, where a visible prototype specifies a parameter type of less floating-point precision, is also required to perform these conversions.

### Other Languages

A universal feature of strongly typed languages is that the assignment operator is only able to store a value into an object that is representable in an object's declared type. Many of these languages do not get involved in representation details. They discuss the cast operator, if one is available, in terms of change of type and tend to say nothing about representation issues.

In some languages the type of the value being assigned becomes the type of the object assigned to. For such language objects are simply labels for values and have no identity of their own, apart from their name (and they may not even have that).

# References

*1: Basic Architecture*. Intel, Inc, 2000.

1. Intel. *IA-32 Intel Architecture Software Developer's Manual Volume*