

The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones

derek@knosof.co.uk

6.3 Conversions

Several operators convert operand values from one type to another automatically.

Commentary

The purpose of these implicit conversions is to reduce the number of type permutations that C operators have to deal with. Forcing developers to use explicit casts was considered unacceptable. There is also the practical issue that most processors only contain instructions that can operate on a small number of scalar types.

Other Languages

There are two main lines of thinking about binary operators whose operands have different types. One is to severely restrict the possible combination of types that are permitted, requiring the developer to explicitly insert casts to cause the types to match. The other (more developer-friendly, or the irresponsible approach, depending on your point of view) is to have the language define implicit conversions, allowing a wide range of differing types to be operated on together.

Common Implementations

Conversion of operand values on some processors can be a time-consuming operation. This is one area where the as-if rule can be applied to make savings—for instance, if the processor has instructions that support operations on mixed types.

Coding Guidelines

Experience suggests that implicit conversions are a common cause of developer miscomprehension and faults in code. Some guideline documents only consider the problems caused by these implicit conversions and simply recommend that all conversions be explicit. The extent to which these explicit conversions increase the cost of program comprehension and are themselves a source of faults is not considered.

The first issue that needs to be addressed is why the operands have different types in the first place. Why weren't they declared with the same type? C developers are often overly concerned with using what they consider to be the *right* integer type. This usually involves attempting to use the type with the smallest representable range needed for the job. Using a single integer type would remove many implicit conversions, and is the recommended practice.

How should those cases where the operands do have different types be handled? Unfortunately, there is no published research on the effects, on developer source code comprehension performance, of having all conversions either implicit or explicit in the source. Without such experimental data, it is not possible to make an effective case for any guideline recommendation.

The potential costs of using explicit conversions include:

- Ensuring that the types used in conversions continue to be the correct ones when code is modified. For instance, the expression `(unsigned int)x+y` may have been correct when the source was originally written, but changes to the declaration of `y` may need to be reflected in changes to the type used to cast `x`. Typedef names offer a partial solution to the dependence between the declaration of an object and its uses in that they provide a mechanism for dealing with changes in `x`'s type. However, handling changes to both `y`'s and `y`'s types is more complex.
- An explicit conversion has to be explicitly read and comprehended, potentially increasing a reader's cognitive load for that expression. Depending on the reason for reading the expression, information on conversions taking place may represent a level of detail below what is required (readers are then paying a cost for no benefit). For instance, in the expression `(unsigned int)x+y` knowing that `x` and `y` are referenced together may be the only information of interest to a reader. The fact that they are added together may be the next level of detail of interest. The exact sequence of operations carried out in the evaluation of the expression being the final level of detail of interest.

The potential costs of relying on implicit conversions include:

operand
convert auto-
matically

object ??
int type only

typedef name
syntax

- Modifications to the source may change the implicit conversions performed, resulting in a change of program behavior. While this situation is familiar to developers, it can also occur when explicit conversions are used. The implicit conversion case is more well known because most source relies on them rather than explicit conversions. It is not known whether the use of explicit conversions would result in a decrease or increase in faults whose root cause was conversion-related.
- In those cases where readers need to comprehend the exact behavior of an expression's evaluation, effort will need to be invested in deducing what conversions are performed. In the case of explicit conversions, it is generally assumed that the conversions specified are correct and it is often also assumed that they are the only ones that occur. No effort has to be expended in recalling operand types from (human) memory, or searching the source to obtain this information.

There is no experimental evidence showing that use of explicit conversions leads to fewer faults. Those cases where a potential benefit from using an explicit conversion might be found, include:

- Where it is necessary to comprehend the details of the evaluation of an expression, the cognitive effort needed to deduce the conversions performed will be reduced. It is generally assumed that the result of explicit conversions will not themselves be the subject of any implicitly conversions, and that the result type will be that of the explicit conversion.
- Experience has shown that developer beliefs about what implicit conversions will occur are not always the same as what the standard specifies. For instance, a common misconception is that no implicit conversion takes place if the operands have exactly the same type. Your author has heard several explanations for why a conversion is unnecessary, but has never recorded these (so no analysis is possible). A good example is the following:

```

1  unsigned char c1, c2, c3;
2
3  if (c1 + c2 > c3)

```

A developer who believes that no conversions take place will be under the impression that `c1+c2` will always return a result that is within the range supported by the type **unsigned char**. In fact, the integer promotions are applied to all the operands and the result of `c1+c2` can exceed the range supported by the type **unsigned char** (assuming the type **int** has a greater precision than the type **unsigned char**, which is by far the most common situation).

integer promotions

The following is one potential benefit of using implicit conversions:

- In those cases where exact details on the evaluation of an expression is not required, a reader of the source will not have to invest cognitive effort in reading and comprehending any explicit conversion operations.

The faults associated with mixing signed and unsigned operands are often perceived as being the most common conversion faults. Whether this perception is caused by the often-dramatic nature of these faults, or reflects actual occurrences, is not known.

Usage

Usage information on the cast operator is given elsewhere (see Table ??).

Table 653.1: Occurrence of implicit conversions (as a percentage of all implicit conversions; an `_` prefix indicates a literal operand). Based on the translated form of this book's benchmark programs.

Converted to	Converted from	%	Converted to	Converted from	%
(<code>unsigned char</code>)	<code>_int</code>	33.0	(<code>int</code>)	<code>unsigned short</code>	1.9
(<code>unsigned short</code>)	<code>_int</code>	17.7	(<code>unsigned long</code>)	<code>_int</code>	1.8
(other-types)	other-types	11.3	(<code>unsigned int</code>)	<code>int</code>	1.7
(<code>short</code>)	<code>_int</code>	7.6	(<code>short</code>)	<code>int</code>	1.7
(<code>unsigned int</code>)	<code>_int</code>	5.1	(<code>enum</code>)	<code>_int</code>	1.3
(ptr-to)	ptr-to	4.7	(<code>unsigned long</code>)	<code>int</code>	1.2
(<code>char</code>)	<code>_int</code>	3.6	(<code>int</code>)	<code>char</code>	1.2
(ptr-to)	<code>_ptr-to</code>	2.9	(<code>int</code>)	<code>enum</code>	1.0
(<code>int</code>)	<code>unsigned char</code>	2.3			

implicit conversion
explicit conversion

This subclause specifies the result required from such an *implicit conversion*, as well as those that result from a cast operation (an *explicit conversion*). 654

Commentary

This defines the terms *implicit conversion* and *explicit conversion*. The commonly used developer term for *implicit conversion* is *implicit cast* (a term that is not defined by the standard).

C++

The C++ Standard defines a set of implicit and explicit conversions. Declarations contained in library headers also contain constructs that can cause implicit conversions (through the declaration of constructors) and support additional explicit conversions— for instance, the complex class.

The C++ language differs from C in that the set of implicit conversions is not fixed. It is also possible for user-defined declarations to create additional implicit and explicit conversions.

Other Languages

Most languages have some form of implicit conversions. Strongly typed languages tend to minimize the number of implicit conversions. Other languages (e.g., APL, Basic, Perl, and PL/1) go out of their way to provide every possible form of implicit conversion. PL/1 became famous for the extent to which it would go to convert operands that had mismatched types. Some languages use the term *coercion*, not conversion. A value is said to be coerced from one type to another.

Coding Guidelines

Although translators do not treat implicit conversions any different from explicit conversions, developers and static analysis tools often treat them differently. The appearance of an explicit construct is given greater weight than its nonappearance (although an operation might still be performed). It is often assumed that an explicit conversion indicates that its consequences are fully known to the person who wrote it, and that subsequent readers will also comprehend its consequences (an implicit conversion is not usually afforded such a status). The possibility that subsequent modifications to the source may have changed the intended behavior of the explicit conversion, or that the developer may not have only had a limited grasp of the effects of the conversion (but just happened to work for an organization that enforces a *no implicit conversion* guideline), is not usually considered.

The issues involved in conversions between integer types is discussed elsewhere. The following example highlights some of the issues:

```

1  #if MACHINE_Z
2  typedef unsigned int X;

```

signed
integer
corresponding
unsigned integer

```

3  typedef signed int Y;
4  #else
5  typedef signed int X;
6  typedef unsigned int Y;
7  #endif
8
9  unsigned int ui;
10 signed int si;
11 X xi;
12 Y yi;
13
14 void f(void)
15 {
16 /*
17  * It is not apparent from the source that the type of any object might change.
18  */
19 unsigned int uloc;
20 signed int sloc;
21
22 uloc = ui + si;    uloc = ui + (unsigned int)si;
23 sloc = ui + si;    sloc = (signed int)(ui + si);
24 }
25
26 void g(void)
27 {
28 /*
29  * The visible source shows that it is possible for the type of an object to change.
30  */
31 X xloc;
32 Y yloc;
33
34 /*
35  * The following sequence of casts might not be equivalent to those used in f.
36  */
37 xloc = xi + yi;    xloc = xi + (X)yi;
38 yloc = xi + yi;    yloc = (Y)(xi + yi);
39 }

```

In the function `f`, the two assignments to `uloc` give different impressions. The one without an explicit cast of `si` raises the issue of its value always being positive. The presence of an explicit cast in the second cast gives the impression that the author of the code intended the conversion to take place and that there is no need to make further checks. In the second pair of assignments the result is being converted back to a signed quantity. The explicit cast in the second assignment gives the impression that the original author of the code intended the conversion to take place and that there is no need to make further checks.

In the function `g`, the issues are less clear-cut. The underlying types of the operands are not immediately obvious. The fact that typedef names have been used suggests some intent to hide implementation details. Is the replacement of a subset of the implicit conversions by explicit casts applicable in this situation? Having the definition of the typedef names conditional on the setting of a macro name only serves to reduce the possibility of being able to claim any kind of developer intent. Having a typedef name whose definition is conditional also creates problems for tool users. Such tools tend to work by tracing a single path of translation through the source code. In the function `g`, they are likely to give diagnostics applicable to one set of the `X` and `Y` definitions. Such diagnostics are likely to be different if the conditional inclusion results in a different pair of definitions for these typedef names.

Conversions between integer and floating types are special for several reasons:

- Although both represent numerical quantities, in the floating-point case literals visible in the source code need not be exact representations of the value used during program execution.
- The internal representations are usually significantly different.

- One of the representations is capable of representing very large and very small quantities (it has an exponent and fractional part),

These differences, and the resulting behavior, are sufficient to want to draw attention to the fact that a conversion is taking place. The issues involved are discussed in more detail elsewhere.

Example

```

1  extern short es_1, es_2, es_3;
2
3  void f(void)
4  {
5      es_1 = es_2 + es_3;
6      es_1 = (short)((int)es_2 + (int)es_3);
7      es_1 = (short)(es_2 + es_3);
8  }
```

The list in 6.3.1.8 summarizes the conversions performed by most ordinary operators;

655

Commentary

All operators might be said to be *ordinary* and these conversions apply to most of them (e.g., the shift operators are one example of where they are not applied).

C++

Clause 4 ‘Standard conversions’ and 5p9 define the conversions in the C++ Standard.

it is supplemented as required by the discussion of each operator in 6.5.

656

Commentary

Clause 6.5 deals with expressions and describes each operator in detail.

C++

There are fewer such supplements in the C++ Standard, partly due to the fact that C++ requires types to be the same and does not use the concept of compatible type.

C++ supports user-defined overloading of operators. Such overloading could change the behavior defined in the C++ Standard, however these definitions cannot appear in purely C source code.

Conversion of an operand value to a compatible type causes no change to the value or the representation.

657

Commentary

This might almost be viewed as a definition of compatible type. However, there are some conversions that cause no change to the value, or the representation, and yet are not compatible types; for instance, converting a value of type **int** to type **long** when both types have the same value representation.

C++

No such wording applied to the same types appears in the C++ Standard. Neither of the two uses of the C++ term *compatible* (layout-compatible, reference-compatible) discuss conversions.

Other Languages

More strongly typed languages, such as Pascal and Ada, allow types to be created that have the same representation as a particular integer type, but are not compatible with it. In such languages, conversions are about changes of type rather than changes of representation (which is usually only defined for a few special cases, such as between integer and floating-point).

floating-point
converted
to integer
integer
conversion
to floating

operators
cause conversions
shift operator
integer promotions

expressions

compatible type
conversion

Common Implementations

While an obvious optimization for a conversion of an operand value to a compatible type is to not generate any machine code, some translators have been known to generate code to perform this conversion. The generation of this conversion code is often a characteristic of translators implemented on a small budget.

Coding Guidelines

Conversion of an operand value to a compatible type can occur for a variety of reasons:

- The type of the operand has changed since the code was originally written— the conversion has become redundant.
- The type specified in the explicit conversion is a typedef name. The definition of this typedef name might vary depending on the host being targeted. For some hosts it happens to be compatible with the type of the operand value; in other cases it is not.
- The cast operation, or its operand is a parameter in a function-like macro definition. In some cases the arguments specified may result in the expanded body containing a conversion of a value having a compatible type.

macro
function-like

In only the first of these cases might there be a benefit in removing the explicit conversion. The issue of redundant code is discussed elsewhere.

redundant
code

658 **Forward references:** cast operators (6.5.4).

References