# The New C Standard (Excerpted material)

## An Economic and Cultural Commentary

**Derek M. Jones**
derek@knosof.co.uk

## 6.3.1.4 Real floating and integer

floating-point converted to integer

When a finite value of real floating type is converted to an integer type other than **_Bool**, the fractional part is discarded (i.e., the value is truncated toward zero). 686

### Commentary

NaNs are not finite values and neither are they infinities.

IEEE-754 6.3

*The Sign Bit. . . . and the sign of the result of the round floating-point number to integer operation is the sign of the operand. These rules shall apply even when operands or results are zero or infinite.*

negative zero only generated by

When a floating-point value in the range (-1.0, -0.0) is converted to an integer type, the result is required to be a positive zero.

### C90

Support for the type **_Bool** is new in C99.

### Other Languages

This behavior is common to most languages.

### Common Implementations

FLT_ROUNDS

Many processors include instructions that perform truncation when converting values of floating type to an integer type. On some processors the rounding mode, which is usually set to round-to-nearest, has to be changed to round-to-zero for this conversion, and then changed back after the operation. This is an execution-time overhead. Some implementations give developers the choice of faster execution provided they are willing to accept round-to-nearest behavior. In some applications the difference in behavior is significantly less than the error in the calculation, so it is acceptable.

### Coding Guidelines

integer constant expression

An expression consisting of a cast of a floating constant to an integer type is an integer constant expression. Such a constant can be evaluated at translation time. However, there is no requirement that the translation-time evaluation produce exactly the same results as the execution-time evaluation. Neither is there a requirement that the translation-time handling of floating-point constants be identical. In the following example it is possible that a call to `printf` will occur.

```
1  #include <stdio.h>
2
3  void f(void)
4  {
5  float fval = 123456789.0;
6  long lval = (long)123456789.0;
7
8  if (lval != (long)fval)
9     printf("(long)123456789.0 == %ld and %ld\n", lval, (long)fval);
10 }
```

There is a common, incorrect, developer assumption that floating constants whose fractional part is zero are always represented exactly by implementations (i.e., many developers have a mental model that such constants are really integers with the characters `.0` appended to them). While it is technically possible to convert many such constants exactly, experience shows that a surprising number of translators fail to achieve the required degree of accuracy (e.g., the floating constant 6.0 might be translated to the same internal representation as the floating constant 5.999999 and subsequently converted to the integer constant 5).

Rev 686.1

A program shall not depend on the value of a floating constant being converted to an integer constant having the same value.

A developer who has made the effort of typing a floating constant is probably expecting it to be used as a floating type. Based on this assumption a floating constant that is implicitly converted to an integer type is unexpected behavior. Such an implicit conversion can occur if the floating constant is the right operand of an assignment or the argument in a function call. Not only is the implicit conversion likely to be unexpected by the original author, but subsequent changes to the code that cause a function-like macro to be invoked, rather than a function call, to result in a significant change in behavior.

In the following example, a floating constant passed to `CALC_1` results in `glob` being converted to a floating type. If the value of `glob` contains more significant digits than supported by the floating type, the final result assigned to `loc` will not be the value expected. Using explicit casts, as in `CALC_2`, removes the problem caused by the macro argument having a floating type. However, as discussed elsewhere, other dependencies are introduced. Explicitly performing the cast, where the argument is passed, mimics the behavior of a function call and shows that the developer is aware of the type of the argument.

operand convert automatically

```
1  #define X_CONSTANT 123456789.0
2  #define Y_CONSTANT 2
3
4  #define CALC_1(a)  ((a) + (glob))
5  #define CALC_2(a)  ((long)(a) + (glob))
6  #define CALC_3(a)  ((a) + (glob))
7
8  extern long glob;
9
10 void f(void)
11 {
12 long loc;
13
14 loc = CALC_1(X_CONSTANT);
15 loc = CALC_1(Y_CONSTANT);
16
17 loc = CALC_2(X_CONSTANT);
18 loc = CALC_2(Y_CONSTANT);
19
20 loc = CALC_3((long)X_CONSTANT);
21 loc = CALC_3(Y_CONSTANT);
22 }
```

The previous discussion describes some of the unexpected behaviors that can occur when a floating constant is implicitly converted to an integer type. Some of the points raised also apply to objects having a floating type. The costs and benefits of relying on implicit conversions or using explicit casts are discussed, in general, elsewhere. That discussion did not reach a conclusion that resulted in a guideline recommendation being made. Literals differ from objects in that they are a single instance of a single value. As such developers have greater control over their use, on a case by case basis, and a guideline recommendation is considered to be more worthwhile. This guideline recommendation is similar to the one given for conversions of suffixed integer constants.

operand convert automatically

?? integer constant with suffix, not immediately converted

Cg 686.2 A floating constant shall not be implicitly converted to an integer type.

---

687 If the value of the integral part cannot be represented by the integer type, the behavior is undefined.[50)]

**Commentary**

The exponent part in a floating-point representation allows very large values to be created, these could significantly exceed the representable range supported by any integer type. The behavior specified by the standard reflects both the fact that there is no commonly seen processor behavior in this case and the execution-time overhead of performing some defined behavior.

### Other Languages

Other languages vary in their definition of behavior. Like integer values that are not representable in the destination type, some languages require an exception to be raise while others specify undefined behavior. In this case Java uses a two step-process. It first converts the real value to the most negative, or largest positive (depending on the sign of the floating-point number) value representable in a **long** or an **int**. In the second step, if the converted integer type is not **long** or **int**; the narrowing conversions are applied to the result of the first step.

### Common Implementations

Many processors have the option of raising an exception when the value cannot be represented in the integer type. Some allow these exceptions to be switched off, returning the least significant bytes of the value. The IEC 60559 Standard defines the behavior— raise invalid. Most current C99 implementations do not do this.

arithmetic saturated

In graphics applications saturated arithmetic is often required (see Figure 687.1). Some DSP processors[4] and the Intel MMX instructions[2] return the largest representable value (with the appropriate sign).

### Coding Guidelines

A naive analysis would suggest there is a high probability that an object having a floating type will hold a value that cannot be represented in an integer type. However, in many programs the range of floating-point values actually used is relatively small. It is this application-specific knowledge that needs to be taken into account by developers.

exponent

Those translators that perform some kind of flow analysis on object values often limit themselves to tracking the values of integer and pointer types. Because of the potential graininess in the values they represent and their less common usage, objects having floating types may have their set/unset status tracked but their possible numeric value is rarely tracked.

It might appear that, in many ways, this case is the same as that for integer conversions where the value cannot be represented. However, a major difference is processor behavior. There is greater execution overhead required for translators to handle this case independently of how the existing instructions behave. Also, a larger number of processors are capable of raising an exception in this case.

integer value not represented in signed integer

Given that instances of this undefined behavior are relatively rare and instances might be considered to be a fault, no guideline recommendation is made here.

---

integer conversion to floating

When a value of integer type is converted to a real floating type, if the value being converted can be represented exactly in the new type, it is unchanged. 688

### Commentary

The value may be unchanged, but its representation is likely to be completely changed.

There are the same number of representable floating-point values between every power of two (when `FLT_RADIX` has a value of two, the most common case). As the power of two increases, the numeric distance between representable values increases (see Figure **??**). The value of the `*_DIG` macros specify the number of digits in a decimal value that may be rounded into a floating-point number and back again without change

FLT_RADIX

*_DIG macros

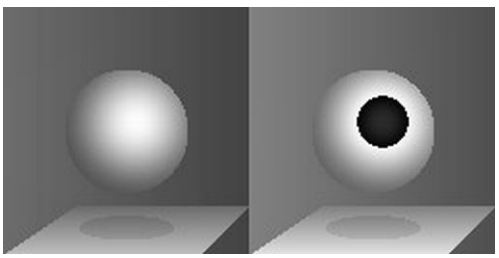

**Figure 687.1:** Illustration of the effect of integer addition wrapping rather than saturating. A value has been added to all of the pixels in the left image to increase the brightness, creating the image on the right. With permission from Jordán and Lotufo.[3]

of value. In the case of the single-precision IEC 60559 representation FLT_DIG is six, which is less than the number of representable digits in an object having type **long** (or 32-bit **int**).

**C++**

*An rvalue of an integer type or of an enumeration type can be converted to an rvalue of a floating point type. The result is exact if possible.* 4.9p2

Who decides what is possible or if it can be represented exactly? A friendly reading suggests that the meaning is the same as C99.

### Common Implementations

The Unisys A Series[5] uses the same representation for integer and floating-point types (an integer was a single precision value whose exponent was zero).

### Coding Guidelines

A common, incorrect belief held by developers is that because floating numbers can represent a much larger range of values than integers and include information on fractional parts, they must be able to exactly represent the complete range of values supported by any integer type. What is overlooked is that the support for an exponent value comes at the cost of graininess in the representation for large value (if objects of integer and floating type are represented in the same number of value bits).

The major conceptual difference between integer and floating types is that one is expected to hold an exact value and the other an approximate value. If developers are aware that approximations can begin at the point an integer value is converted, then it is possible for them to take this into account in designing algorithms. Developers who assume that inaccuracies don't occur until the floating value is operated on are in for a surprise.

Rev 688.1 Algorithms containing integer values that are converted to floating values shall be checked to ensure that any dependence on the accuracy of the conversion is documented and that any necessary execution-time checks against the *_DIG macros are made.

The rationale behind the guideline recommendations against converting floating constants to integer constants do not apply to conversions of integer constants to floating types. 686.2 floating constant implicitly converted

### Example

```
#include <limits.h>
#include <stdio.h>

static float max_short = (float)SHRT_MAX;
static float max_int = (float)INT_MAX;
static float max_long = (float)LONG_MAX;

int main(void)
{
float max_short_m1,
      max_int_m1,
      max_long_m1;

for (int i_f=1; i_f < 3; i_f++)
   {
   max_short_m1 = (float)(SHRT_MAX - i_f);
   if (max_short == max_short_m1)
      printf("float cannot represent all representable shorts\n");
```

```
19      max_short=max_short_m1;
20      max_int_m1 = (float)(INT_MAX - i_f);
21      if (max_int == max_int_m1)
22         printf("float cannot represent all representable ints\n");
23      max_int=max_int_m1;
24      max_long_m1 = (float)(LONG_MAX - i_f);
25      if (max_long == max_long_m1)
26         printf("float cannot represent all representable longs\n");
27      max_long=max_long_m1;
28      }
29   }
```

---

int to float nearest representable value

If the value being converted is in the range of values that can be represented but cannot be represented exactly, the result is either the nearest higher or nearest lower representable value, chosen in an implementation-defined manner. 689

### Commentary

There is no generally accepted behavior in this situation. The standard leaves it up to the implementation.

### Other Languages

Most languages are silent on this issue.

### Common Implementations

status flag floating-point
FLT_ROUNDS

Most processors contain a status flag that is used to control the rounding of all floating-point operations. Given that round-to-nearest is the most common rounding mode, the most likely implementation-defined behavior is round-to-nearest.

### Coding Guidelines

A consequence of this behavior is that it is possible for two unequal integer values to be converted to the same floating-point value. Any algorithm that depends on the relationships between integer values being maintained after conversion to a floating type may not work as expected.

---

footnote 48

48) The integer promotions are applied only: as part of the usual arithmetic conversions, to certain argument expressions, to the operands of the unary +, -, and ~ operators, and to both operands of the shift operators, as specified by their respective subclauses. 690

### Commentary

The certain argument expressions are function calls where there is no type information available. This happens for old-style function declarations and for prototype declarations where the ellipsis notation has been used and the argument corresponds to one of the parameters covered by this notation.

Another context where integer promotions are applied is the controlling expression in a **`switch`** statement. In all other cases the operand is being used in a context where its value is being operated on.

Contexts where the integer promotions are not applied are the expression specifying the number of elements in a VLA type definition (when there are no arithmetic or logical operators involved), function return values, the operands of the assignment operator, and arguments to a function where the parameter type is known. In the latter three cases the value is, potentially, implicitly cast directly to the destination type.

if statement operand compare against 0
iteration statement executed repeatedly

The standard does not specify that the implicit test against zero, in an **`if`** statement or iteration statement, will cause a single object (forming the complete controlling expression) to be promoted. A promotion would not affect the outcome in these contexts, and an implementation can use the as-if rule in selecting the best machine code to generate.

### C++

In C++, integral promotions are applied also as part of the usual arithmetic conversions, the operands of the unary +, -, and ~ operators, and to both operands of the shift operators. C++ also performs integer promotions in contexts not mentioned here, as does C.

### Coding Guidelines

There are a small number of cases where the integer promotions do not occur. Is anything to be gained by calling out these situations in coding guideline documents? Experience suggests that developers are more likely to forget that the integer promotions occur (or invent mythical special cases where they don't occur) rather than worry about additional conversions because of them. Coding guideline documents are no substitute for proper training.

---

691 49) The rules describe arithmetic on the mathematical value, not the value of a given type of expression. footnote 49

### Commentary

This means that there are no representational issues involving intermediate results being within range of the type of the values. The abstract machine must act as if the operation were performed using infinite precision arithmetic.

### C90

This observation was not made in the C90 Standard (but was deemed to be implicitly true).

### C++

The C++ Standard does not make this observation.

---

692 50) The remaindering operation performed when a value of integer type is converted to unsigned type need not be performed when a value of real floating type is converted to unsigned type. footnote 50

### Commentary

This permission reflects both differences in processor instruction behavior and the (in)ability to detect and catch those cases where the conversion might be performed in software. The behavior is essentially unspecified, although it is not explicitly specified as such (and it does not appear in the list of unspecified behaviors in Annex J.1).

### C++

*An rvalue of a floating point type can be converted to an rvalue of an integer type. The conversion truncates; that is, the fractional part is discarded. The behavior is undefined if the truncated value cannot be represented in the destination type.* 4.9p1

The conversion behavior, when the result cannot be represented in the destination type is undefined in C++ and unspecified in C.

### Common Implementations

The floating-point to integer conversion instructions on many processors are only capable of delivering signed integer results. An implementation may treat the value as a sequence of bits independent of whether a signed or unsigned value is expected. In this case the external behavior for two's complement notation is the same as if a remaindering operation had been performed. Converting values outside of the representable range of any integer type supported by an implementation requires that the processor conversion instruction either perform the remainder operation or raise some kind of range error signal that is caught by the implementation, which then performs the remainder operation in software.

### Coding Guidelines

This is one area where developers' expectations may not mirror the behavior that an implementation is required to support.

### Example

The following program may, or may not, output the values given in the comments.

```
#include <limits.h>
#include <stdio.h>

int main(void)
{
double d = UINT_MAX;
printf("%f, %u\n", d, (unsigned int)d); /* 4294967295.000000, 4294967295 */
d += 42;
printf("%f, %u\n", d, (unsigned int)d); /* 4294967337.000000, 41         */
d *= 20;
printf("%f, %u\n", d, (unsigned int)d); /* 85899346740.000000, 820       */
d = -1;
printf("%f, %u\n", d, (unsigned int)d); /* -1.000000, 4294967295         */
}
```

---

Thus, the range of portable real floating values is $(-1, \textit{Utype\_}\texttt{MAX} + 1)$. 693

### Commentary

The round brackets are being used in the mathematical sense; the bounds represent excluded limits (i.e., -1 is not in the portable range). This statement only applies to unsigned integer types.

### C++

The C++ Standard does not make this observation.

---

If the value being converted is outside the range of values that can be represented, the behavior is undefined. 694

### Commentary

*_MAX_10_EXP

An unsigned integer type represented in 123 bits, or more, could contain a value that would be outside the range of values representable in a minimum requirements **`float`** type (128 bits would be needed to exceed the range of the IEC 60559 single-precision).

### C++

4.9p2 *Otherwise, it is an implementation-defined choice of either the next lower or higher representable value.*

The conversion behavior, when the result is outside the range of values that can be represented in the destination type, is implementation-defined in C++ and undefined in C.

### Other Languages

Many other language standards were written in an age when floating-point types could always represent much larger values than could be represented in integer types and their specifications reflect this fact (by not mentioning this case). In Java values having integer type are always within the representable range of the floating-point types it defines.

### Common Implementations

Processors that support 128-bit integer types, in hardware, are starting to appear.[1]

### Coding Guidelines

Developers are likely to consider this issue to be similar to the year 2036 problem— address the issue when the course of history requires it to be addressed.

# References


1. AMD. *AMD64 Architecture Programmer's Manual Volume 1: Application programming*. Advanced Micro Devices, Inc, 3.09 edition, Sept. 2003.
2. Intel. *IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture*. Intel, Inc, 2000.
3. R. Jordan, R. Lotufo, and D. Argiro. *Khoros Pro 2001 Student Version*. Khoral Research, Inc, 2000.
4. Motorola, Inc. *DSP563CCC Motorola DSP56300 Family Optimizing C Compiler User's Manual*. Motorola, Inc, Austin, TX, USA, 19??
5. Unisys Corporation. *Architecture MCP/AS (Extended)*. Unisys Corporation, 3950 8932-100 edition, 1994.