The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

6.3.1.3 Signed and unsigned integers

When a value with integer type is converted to another integer type other than **_Bool**, if the value can be represented by the new type, it is unchanged. 682

Commentary

While it would very surprising to developers if the value was changed, the standard needs to be complete and specify the behavior of all conversions. For integer types this means that the value has to be within the range specified by the corresponding numerical limits macros.

numerical limits

The type of a bit-field is more than just the integer type used in its declaration. The width is also considered to be part of its type. This means that assignment, for instance, to a bit-field object may result in the value being assigned having its value changed.

bit-field interpreted as

```
1  void DR_120(void)
2  {
3  struct {
4          unsigned int mem : 1;
5         } x;
6  /*
7   * The value 3 can be represented in an unsigned int,
8   * but is changed by the assignment in this case.
9   */
10 x.mem = 3;
11 }
```

C90

Support for the type **_Bool** is new in C99, and the C90 Standard did not need to include it as an exception.

Other Languages

This general statement holds true for conversions in other languages.

Common Implementations

The value being in range is not usually relevant because most implementations do not perform any range checks on the value being converted. When converting to a type of lesser rank, the common implementation behavior is to ignore any bit values that are not significant in the destination type. (The sequence of bits in the value representation of the original type is truncated to the number of bits in the value representation of the destination type.) If the representation of a value does not have any bits set in these ignored bits, the converted value will be the same as the original value. In the case of conversions to value representations containing more bits, implementations simply sign-extend for signed values and zero-fill for unsigned values.

Coding Guidelines

One way of reducing the possibility that converted values are not representable in the converted type is to reduce the number of conversions. This is one of the rationales behind the general guideline on using a single integer type.

object ?? int type only

unsigned integer conversion to

Otherwise, if the new type is unsigned, the value is converted by repeatedly adding or subtracting one more than the maximum value that can be represented in the new type until the value is in the range of the new type.[49)] 683

Commentary

This behavior is what all known implementations do for operations on values having unsigned types. The standard is enshrining existing processor implementation practices in the language. As footnote 49 points out, this adding and subtracting is done on the abstract mathematical value, not on a value with a given C type. There is no need to think in terms of values wrapping (although this is a common way developers think about the process).

footnote 49

C90

Otherwise: if the unsigned integer has greater size, the signed integer is first promoted to the signed integer corresponding to the unsigned integer; the value is converted to unsigned by adding to it one greater than the largest number that can be represented in the unsigned integer type.[28)]

When a value with integral type is demoted to an unsigned integer with smaller size, the result is the nonnegative remainder on division by the number one greater than the largest unsigned number that can be represented in the type with smaller size.

The C99 wording is a simpler way of specifying the C90 behavior.

Common Implementations

For unsigned values and signed values represented using two's complement, the above algorithm can be implemented by simply chopping off the significant bits that are not available in the representation of the new type.

Coding Guidelines

The behavior for this conversion may be fully specified by the standard. The question is whether a conversion should be occurring in the first place.

unsigned computation modulo reduced

---

684 Otherwise, the new type is signed and the value cannot be represented in it;

integer value not represented in signed integer

Commentary

To be exact, the standard defines no algorithm for reducing the value to make it representable (because there is no universal agreement between different processors on what to do in this case).

Other Languages

The problem of what to do with a value that, when converted to a signed integer type, cannot be represented is universal to all languages supporting more than one signed integer type, or support an unsigned integer type (the overflow that can occur during an arithmetic operation is a different case).

Coding Guidelines

A guideline recommendation that the converted value always be representable might be thought to be equivalent to one requiring that a program not contain defects. However, while the standard may not specify an algorithm for this conversion, there is a commonly seen implementation behavior. Developers sometimes intentionally make use of this common behavior and the applicable guideline is the one dealing with the use of representation information.

?? representation information using

---

685 either the result is implementation-defined or an implementation-defined signal is raised.

signed integer conversion implementation-defined

Commentary

There is no universally agreed-on behavior (mathematical or processor) for the conversion of out-of-range signed values, so the C Standard's Committee could not simply define this behavior as being what happens in practice. The definition of implementation-defined behavior does not permit an implementation to raise a signal; hence, the additional permission to raise a signal is specified here.

implementation-defined behavior

C90

The specification in the C90 Standard did not explicitly specify that a signal might be raised. This is because the C90 definition of implementation-defined behavior did not rule out the possibility of an implementation raising a signal. The C99 wording does not permit this possibility, hence the additional permission given here.

C++

4.7p3 *. . . ; otherwise, the value is implementation-defined.*

The C++ Standard follows the wording in C90 and does not explicitly permit a signal from being raised in this context because this behavior is considered to be within the permissible range of implementation-defined behaviors.

Other Languages

Languages vary in how they classify the behavior of a value not being representable in the destination type. Java specifies that all the unavailable significant bits (in the destination type) are discarded. Ada requires that an exception be raised. Other languages tend to fall between these two extremes.

Common Implementations

The quest for performance and simplicity means that few translators generate machine code to check for nonrepresentable conversions. The usual behavior is for the appropriate number of least significant bits from the original value to be treated as the converted value. The most significant bit of this new value is treated as a sign bit, which is sign-extended to fill the available space if the value is being held in a register. If the conversion occurs immediately before a store (i.e., a right-hand side value is converted before being assigned into the left hand side object), there is often no conversion; the appropriate number of value bits are simply written into storage.

Some older processors[1] have the ability to raise a signal if a conversion operation on an integer value is not representable. On such processors an implementation can choose to use this instruction or alternatively use a sequence of instructions having the same effect, that do not raise a signal.

References

1. D. E. Corporation. *VAX11 780 Architecture Handbook*. Digital Equipment Corporation, 1977.