The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

6.3.1.1 Boolean, characters, and integers

conversion rank

Every integer type has an *integer conversion rank* defined as follows: 659

Commentary

This defines the term *integer conversion rank*. Although this is a new term, there already appears to be a common usage of the shorter term *rank*. C90 contained a small number of integer types. Their relative properties were enumerated by naming each type with its associated properties. With the introduction of two new integer types and the possibility of implementation-defined integer types, documenting the specification in this manner is no longer practical.

The importance of the rank of a type is its value relative to the rank of other types. The standard places no requirements on the absolute numerical value of any rank. The standard does not define any mechanism for a developer to control the rank assigned to any extended types. That is not to say that implementations cannot support such functionality, provided the requirements of this clause are met.

There is no rank defined for bit-fields.

C90

The concept of integer conversion rank is new in C99.

C++

The C++ Standard follows the style of documenting the requirements used in the C90 Standard. The conversions are called out explicitly rather than by rank (which was introduced in C99). C++ supports operator overloading, where the conversion rules are those of a function call. However, this functionality is not available in C.

Other Languages

Most languages have a single integer type, which means there are no other integer types to define a relationship against. Languages that support an unsigned type describe the specific relationship between the signed and unsigned integer type.

Common Implementations

Whether implementations use the new concept of rank or simply extend the existing way things are done internally is an engineering decision, which is invisible to the user of a translator product — the developer.

Coding Guidelines

Although integer conversion rank is a new term it has the capacity to greatly simplify discussions about the integer types. Previously, general discussions on integer types either needed to differentiate them based on some representation characteristic, such as their size or width, or by enumerating the types and their associated attributes one by one.

— No two signed integer types shall have the same rank, even if they have the same representation. 660

Commentary

types different even if same representation

This mirrors the concept that two types are different, even if they have the same representation.

rank signed integer vs less precision

— The rank of a signed integer type shall be greater than the rank of any signed integer type with less precision. 661

Commentary

The standard could equally well have defined things the other way around, with the rank being less than. On the basis that the precision of integral types continues to grow, over time, it seems more intuitive to also have the ranks grow. So the largest rank is as open-ended as the largest number of bits in an integral type. Given the requirement that follows this one, for the time being, this requirement, only really applies to extended integer types.

C++

The relative, promotion, ordering of signed integer types defined by the language is called out explicitly in clause 5p9.

---

662 — The rank of **long long int** shall be greater than the rank of **long int**, which shall be greater than the rank of **int**, which shall be greater than the rank of **short int**, which shall be greater than the rank of **signed char**.

rank
standard integer types

Commentary

This establishes a pecking order among the signed integer types defined by the standard.

C++

Clause 5p9 lists the pattern of the usual arithmetic conversions. This follows the relative orderings of rank given here (except that the types **short int** and **signed char** are not mentioned; nor would they be since the integral promotions would already have been applied to operands having these types).

---

663 — The rank of any unsigned integer type shall equal the rank of the corresponding signed integer type, if any.

rank
corresponding signed/unsigned

Commentary

The only unsigned integer type that does not have a corresponding signed integer type is **_Bool**.

standard unsigned integer
extended unsigned integer

Rank is not sufficient to handle all conversion cases. Information on the sign of the integer type is also needed.

---

664 — The rank of any standard integer type shall be greater than the rank of any extended integer type with the same width.

rank
standard integer relative to extended

Commentary

This is a requirement on the implementation. It gives preference to converting to the standard defined type rather than any extended integer type that shares the same representation.

Why would a vendor provide an extended type that is the same width as one of the standard integer types? The translator vendor may support a variety of different platforms and want to offer a common set of typedefs, across all supported platforms, in the **<stdint.h>** header. This could have the effect, on some platforms, of an extended integer type having the same width as one of the standard integer types. A vendor may also provide more than one representation of integer types. For instance, by providing support for extended integer types whose bytes have the opposite endianness to that of the standard integer types.

endian

C++

The C++ Standard specifies no requirements on how an implementation might extend the available integer types.

---

665 — The rank of **char** shall equal the rank of **signed char** and **unsigned char**.

char
rank

Commentary

This statement is needed because the type **char** is distinct from that of the types **signed char** and **unsigned char**.

char
separate type

---

666 — The rank of **_Bool** shall be less than the rank of all other standard integer types.

_Bool
rank

Commentary

This does not imply that the object representation of the type **_Bool** contains a smaller number of bits than any other integer type (although its value representation must).

unsigned integer types
object representation

C++

3.9.1p6 *As described below, **bool** values behave as integral types.*

4.5p4 *An rvalue of type **bool** can be converted to an rvalue of type **int**, with **false** becoming zero and **true** becoming one.*

The C++ Standard places no requirement on the relative size of the type **bool** with respect to the other integer types. An implementation may choose to hold the two possible values in a single byte, or it may hold those values in an object that has the same width as type **long**.

Other Languages

Boolean types, if supported, are usually viewed as the smallest type, irrespective of the amount of storage used to represent them.

---

rank enumerated type

— The rank of any enumerated type shall equal the rank of the compatible integer type (see 6.7.2.2). 667

Commentary

enumeration type compatible with enumerators type int

The compatible integer type can vary between different enumerated types. An enumeration constant has type **int**. There is no requirement preventing the rank of an enumerated type from being less than, or greater than, the rank of **int**.

Other Languages

Most languages that contain enumerated types treat them as being distinct from the integer types and an explicit cast is required to obtain their numeric value. So the C issues associated with rank do not occur.

---

rank extended integer relative to extended

— The rank of any extended signed integer type relative to another extended signed integer type with the same precision is implementation-defined, but still subject to the other rules for determining the integer conversion rank. 668

Commentary

The reasons why an implementation might provide two extended signed integer types of the same precision is the same as the reasons why it might provide such a type having the same precision as a standard integer type. Existing practice provides a ranking for the standard integer types (some or all of which may have the same precision).

rank 664 standard integer relative to extended

rank 662 standard integer types

C++

The C++ Standard does not specify any properties that must be given to user-defined classes that provide some form of extended integer type.

Coding Guidelines

rank 664 standard integer relative to extended

The same issues apply here as applied to the extended integer types in relation to the standard integer types.

---

rank transitive

— For all integer types **T1**, **T2**, and **T3**, if **T1** has greater rank than **T2** and **T2** has greater rank than **T3**, then **T1** has greater rank than **T3**. 669

Commentary

The rank property is transitive.

---

expression wherever an int may be used

The following may be used in an expression wherever an **int** or **unsigned int** may be used: 670

Commentary

An **int** can be thought of as the smallest functional unit of type for arithmetic operations (the types with greater rank being regarded as larger units). This observation is a consequence of the integer promotions. Any integer type can be used in an expression wherever an **int** or **unsigned int** may be used (this may involve them being implicitly converted). However, operands having one of the types specified in the following sentences will often return the same result if they also have the type **int** or **unsigned int**.

675 integer promotions

C90

The C90 Standard listed the types, while the C99 Standard bases the specification on the concept of rank.

*A **char**, a **short int**, or an **int** bit-field, or their signed or unsigned varieties, or an enumeration type, may be used in an expression wherever an **int** or **unsigned int** may be used.*

C++

C++ supports the overloading of operators; for instance, a developer-defined definition can be given to the binary + operator, when applied to operands having type **short**. Given this functionality, this C sentence cannot be said to universally apply to programs written in C++. It is not listed as a difference because it requires use of C++ functionality for it to be applicable. The implicit conversion sequences are specified in clause 13.3.3.1. When there are no overloaded operators visible (or to be exact no overloaded operators taking arithmetic operands, and no user-defined conversion involving arithmetic types), the behavior is the same as C.

Other Languages

Most other languages do not define integer types that have less precision than type **int**, so they do not contain an equivalent statement. The type **char** is usually a separate type and an explicit conversion is needed if an operand of this type is required in an **int** context.

Coding Guidelines

If the guideline recommendation specifying use of a single integer type is followed, this permission will never be used.

?? object int type only
675 integer promotions

Example

In the following:

```
1  #include <limits.h>
2
3  typedef unsigned int T;
4  T x;
5
6  int f(void)
7  {
8  if (sizeof(x) == 2)
9     return (x << CHAR_BIT) << CHAR_BIT;
10 else
11    return sizeof(x);
12 }
```

the first **return** statement will always return zero when the rank of type T is less than or equal to the rank of **int**. There is no guarantee that the second **return** statement will always deliver the same value for different types.

671 — An object or expression with an integer type whose integer conversion rank is less than or equal to the rank of **int** and **unsigned int**.

Commentary

rank 663
corresponding signed/unsigned integer promotions 675

The rank of **int** and **unsigned int** is the same. The integer promotions will be applied to these objects.

The wording was changed by the response to DR #230 and allows objects having enumeration type (whose rank may equal the rank of **int** and **unsigned int**) to appear in these contexts (as did C90).

C++

4.5p1 *An rvalue of type **char**, **signed char**, **unsigned char**, **short int**, or **unsigned short int** can be converted to an rvalue of type **int** if **int** can represent all the values of the source type; otherwise, the source rvalue can be converted to an rvalue of type **unsigned int**.*

4.5p2 *An rvalue of type **wchar_t** (3.9.1) or an enumeration type (7.2) can be converted to an rvalue of the first of the following types that can represent all the values of its underlying type: **int**, **unsigned int**, **long**, or **unsigned long**.*

4.5p4 *An rvalue of type **bool** can be converted to an rvalue of type **int**, with **false** becoming zero and **true** becoming one.*

The key phrase here is *can be*, which does not imply that they *shall be*. However, the situations where these conversions might not apply (e.g., operator overloading) do not involve constructs that are available in C. For binary operators the *can be* conversions quoted above become *shall be* requirements on the implementation (thus operands with rank less than the rank of **int** are supported in this context):

5p9 *Many binary operators that expect operands of arithmetic or enumeration type cause conversions and yield result types in a similar way. The purpose is to yield a common type, which is also the type of the result. This pattern is called the usual arithmetic conversions, which are defined as follows:*

— Otherwise, the integral promotions (4.5) shall be performed on both operands.[54)]

Footnote 54 *54) As a consequence, operands of type **bool**, **wchar_t**, or an enumerated type are converted to some integral type.*

The C++ Standard does not appear to contain explicit wording giving this permission for other occurrences of operands (e.g., to unary operators). However, it does not contain wording prohibiting the usage (the wording for the unary operators invariably requires the operand to have an arithmetic or scalar type).

Other Languages

The few languages that do support more than one integer type specify their own rules for when different types can occur in an expression at the same time.

---

bit-field in expression

— A bit-field of type **_Bool**, **int**, **signed int**, or **unsigned int**. 672

Commentary

bit-field maximum width

A bit-field is a method of specifying the number of bits to use in the representation of an integer type. The type used in a bit-field declaration specifies the set of possible values that might be available, while the constant value selects the subset (which can include all values) that can be represented by the member. Because the integer promotion rules are based on range of representable values, not underlying signedness of the type, it is possible for a member declared as a bit-field using an unsigned type to be promoted to the type **signed int**.

C90

Support for bit-fields of type **_Bool** is new in C99.

C++

An rvalue for an integral bit-field (9.6) can be converted to an rvalue of type ***int*** *if* ***int*** *can represent all the values of the bit-field; otherwise, it can be converted to* ***unsigned int*** *if* ***unsigned int*** *can represent all the values of the bit-field. If the bit-field is larger yet, no integral promotion applies to it. If the bit-field has an enumerated type, it is treated as any other value of that type for promotion purposes.* 4.5p3

C does not support the definition of bit-fields that are larger than type **int**, or bit-fields having an enumerated type.

Other Languages

Languages, such as Pascal and Ada, provide developers with the ability to specify the minimum and maximum values that need to be represented in an integer type (a bit-field specifies the number of bits in the representation, not the range of values). These languages contain rules that specify how objects defined to have these subrange types can be used anywhere that an object having integer type can appear.

Common Implementations

Obtaining the value of a member that is a bit-field usually involves several instructions. The storage unit holding the bit-field has to be loaded, invariably into a register. Those bits not associated with the bit-field being read then need to be removed. This can involve using a bitwise-and instruction to zero out bits and right shift the bit sequence. For signed bit-fields, it may then be necessary to sign extend the bit sequence. Storing a value into an object having a bit-field type can be even more complex. The new value has to be converted to a bit sequence that fits in the allocate storage, without changing the values of any adjacent objects.

Some CISC processors[6] have instructions designed to access bit-fields. Such relatively complex instructions went out of fashion when RISC design philosophy first took off, but they have started to make a come back.[1, 3] Li and Gupta[4] found that adding instructions to the ARM processor that operated (add, subtract, compare, move, and bitwise operations) on subwords reduced the cycle count of various multimedia benchmarks by between 0.39% and 8.67% (code size reductions were between 1.27% and 21.05%).

---

673 If an **int** can represent all values of the original type, the value is converted to an **int**; int can represent values converted to int

Commentary

Type conversions occur at translation time, when actual values are usually unknown. The standard requires the translator to assume that the value of the expression can be any one of the representable values supported by its type. While flow analysis could reduce the range of possible values, the standard does not require such analysis to be performed. (If it is performed, a translator cannot use it to change the external behavior of a program; that is, optimizations may be performed but the semantics specified by the standard is followed.)

Other Languages

Most languages have a single signed integer type, so there is rarely a smaller integer type that needs implicit conversion.

Coding Guidelines

Some developers incorrectly assume that objects declared using typedef names do not take part in the integer promotions. Incorrect assumptions by a developer are very difficult to deduce from an analysis of the source code. In some cases the misconception will be harmless, the actual program behavior being identical to the misconstrued behavior. In other cases the behavior is different. Guideline recommendations are not a substitute for proper developer training. typedef assumption of no integer promotions

Example

```
1  typedef short SHORT;
2
3  extern SHORT es_1,
4               es_2;
5
6  void f(void)
7  {
8  unsigned int ui = 3;    /* Value representable in a signed int.     */
9
10 if (es_1 == (es_2 + 1)) /* Operands converted to int.               */
11    ;
12 if (ui > es_1)          /* Right operand converted to unsigned int. */
13    ;
14 }
```

---

int cannot represent values converted to unsigned int

otherwise, it is converted to an **unsigned int**. 674

Commentary

This can occur for the types **unsigned short**, or **unsigned char**, if either of them has the same representation as an **unsigned int**. Depending on the type chosen to be compatible with an enumeration type, it is possible for an object that has an enumerated type to be promoted to the type **unsigned int**.

Common Implementations

On 16-bit processors the types **short** and **int** usually have the same representation, so **unsigned short** promotes to **unsigned int**. On 32-bit processors the type **short** usually has less precision than **int**, so the type **unsigned short** promotes to **int**. There are a few implementations, mostly on DSP-based processors, where the character types have the same width as the type **int**.[5]

Coding Guidelines

Existing source code ported, from an environment in which the type **int** has greater width than **short**, to an environment where they both have the same width may have its behavior changed. If the following is executed on a host where the width of type **int** is greater than the width of **short**:

```
1  #include <stdio.h>
2
3  extern unsigned short us;
4  extern signed int si; /* Can hold negative values. */
5
6  void f(void)
7  {
8  if (us > si)
9     printf("Pass\n");
10 else
11    printf("Fail\n");
12 }
```

the object `us` will be promoted to the type **int**. There will not be any change of values. On a host where the types **int** and **short** have the same width, an **unsigned short** will be promoted to **unsigned int**. This will lead to `si` being promoted to **unsigned int** (the usual arithmetic conversions) and a potential change in its value. (If it has a small negative value, it will convert to a large positive value.) The relational comparison will then return a different result than in the previous promotion case.

Cg 674.1

An object having an unsigned integer type shall not be implicitly converted to **unsigned int** through the application of the integer promotions.

The consequence of this guideline recommendation is that such conversions need to be made explicit, using a cast to an integer type whose rank is greater than or equal to **int**.

---

675 These are called the *integer promotions*.[48)]

integer promotions

Commentary

This defines the term *integer promotions*. Integer promotions occur when an object having a rank less than **int** appears in certain contexts. This behavior differs from arithmetic conversions where the type of a different object is involved. Integer promotions are affected by the relative widths of types (compared to the width of **int**). If the type **int** has greater width than **short** then, in general (the presence of extended integer types whose rank is also less than **int** can complicate the situation), all types of less rank will convert to **int**. If **short** has the same precision as **int**, an **unsigned short** will invariably promote to an **unsigned int**.

footnote 48

It is possible to design implementations where the integer conversions don't follow a simple pattern, such as the following:

```
signed short 16 bits including sign     unsigned short 24 bits
signed int   24 bits including sign     unsigned int   32 bits
```

Your author does not know of any implementation that uses this kind of unusual combination of bits for its integer type representation.

C90

These are called the integral promotions.[27)]

C++

The C++ Standard uses the C90 Standard terminology (and also points out, 3.9.1p7, "A synonym for integral type is *integer type*.").

Other Languages

The *unary numeric promotions* and *binary numeric promotions* in Java have the same effect.

Common Implementations

Many processors have load instructions that convert values having narrower types to a wider type. For instance, loading a byte into a register and either sign extending (**signed char**), or zero filling (**unsigned char**) the value to occupy 32 bits (promotion to **int**). On processors having instructions that operate on values having a type narrower than **int** more efficiently than type **int**, optimizers can make use of the as-if rule to improve efficiency. For instance, in some cases an analysis of the behavior of a program may find that operand values and the result value is always representable in the their unpromoted type. Implementations need only to act as if the object had been converted to the type **int**, or **unsigned int**.

Coding Guidelines

If the guideline recommendation specifying use of a single integer type is followed there would never be any integer promotions. The issue of implicit conversions versus explicit conversions might be a possible cause of a deviation from this recommendation and is discussed elsewhere.

?? object
int type only

operand
convert automatically

Example

```
1  signed short    s1,  s2,  s3;
2  unsigned short us1, us2, us3;
3
4  void f(void)
```

```
{
s1 = s2 + s3;  /*
                * The result of + may be undefined.
                * The conversion for the = may be undefined.
                */
               /* s1 = (short)((int)s2 + (int)s3);           */
s1 = us2 + s3; /* The conversion for the = may be undefined. */
               /*
                * The result of the binary + is always defined (unless
                * the type int is only one bit wider than a short; no
                * known implementations have this property).
                *
                * Either both shorts promote to a wider type:
                *
                *       s1 = (short)((int)us2 + (int)s3);
                *
                * or they both promote to an unsigned type of the same width:
                *
                *       s1 = (short)((unsigned int)us2 + (unsigned int)s3);
                */
 s1 = us2 + us3; /* The conversion for the = may be undefined. */
us1 = us2 + us3; /* Always defined */
us1 = us2 +  s3; /* Always defined */
us1 =  s2 +  s3; /* The result of + may undefined.             */
}
```

Table 675.1: Occurrence of integer promotions (as a percentage of all operands appearing in all expressions). Based on the translated form of this book's benchmark programs.

Original Type	%	Original Type	%
unsigned char	2.3	**char**	1.2
unsigned short	1.9	**short**	0.5

All other types are unchanged by the integer promotions. 676

Commentary

The integer promotions are only applied to values whose integer type has a rank less than that of the **int** type.

C++

This is not explicitly specified in the C++ Standard. However, clause 4.5, Integral promotions, discusses no other types, so the statement is also true in C++

value preserving

The integer promotions preserve value including sign. 677

Commentary

These rules are sometimes known as *value preserving promotions*. They were chosen by the Committee because they result in the least number of surprises to developers when applied to operands. The promoted value would remain unchanged whichever of the two rules used by implementations were used. However, in many cases this promoted value appears as an operand of a binary operator. If *unsigned preserving promotions* were used (see Common implementations below), the value of the operand could have its sign changed (e.g., if the operands had types **unsigned char** and **signed char**, both their final operand type would have been **unsigned int**), potentially leading to a change of that value (if it was negative). The *unsigned preserving promotions* (sometimes called rules rather than promotions) are sometimes also known as *sign preserving rules* because the form of the sign is preserved.

Most developers think in terms of values, not signedness. A rule that attempts to preserve sign can cause a change of value, something that is likely to be unexpected. Value preserving rules can also produce results that are unexpected, but these occur much less often.

Rationale

The *unsigned preserving* rules greatly increase the number of situations where **unsigned int** confronts **signed int** to yield a questionably signed result, whereas the *value preserving* rules minimize such confrontations. Thus, the value preserving rules were considered to be safer for the novice, or unwary, programmer. After much discussion, the C89 Committee decided in favor of value preserving rules, despite the fact that the UNIX C compilers had evolved in the direction of unsigned preserving.

Other Languages

This is only an issue for languages that contain more than one signed integer type and an unsigned integer type.

Common Implementations

base document

The base document specified unsigned preserving rules. If the type being promoted was either **unsigned char** or **unsigned short**, it was converted to an **unsigned int**. The corresponding signed types were promoted to **signed int**. Some implementations provide an option to change their default behavior to follow unsigned preserving rules.[2,7,8]

Coding Guidelines

Existing, very old, source code may rely on using the unsigned preserving rules. It can only do this if the translator is also running in such a mode, either because that is the only one available or because the translator is running in a compatibility mode to save on the porting (to the ISO rules) cost. Making developers aware of any of the issues involved in operating in a nonstandard C environment is outside the scope of these coding guidelines.

Example

```
1  extern unsigned char uc;
2
3  void f(void)
4  {
5  int si = -1;
6  /*
7   * Value preserving rules promote uc to an int -> comparison succeeds.
8   *
9   * Signed preserving rules promote uc to an unsigned int, usual arithmetic
10  * conversions then convert si to unsigned int -> comparison fails.
11  */
12 if (uc > si)
13    ;
14 }
```

678 As discussed earlier, whether a "plain" **char** is treated as signed is implementation-defined.

char
plain treated as

Commentary

char
range, representation and behavior

The implementation-defined treatment of "plain" char will only affect the result of the integer promotions if any of the character types can represent the same range of values as an object of type **int** or **unsigned int**.

679 **Forward references:** enumeration specifiers (6.7.2.2), structure and union specifiers (6.7.2.1).

References

1. Agere Systems. *DSP16000 Digital Signal Processor Core Information Manual*. Agere Systems, mn02-026winf edition, June 2002.
2. HP. *DEC C Language Reference Manual*. Compaq Computer Corporation, aa-rh9na-te edition, July 1999.
3. Intel, Inc. *Intel IA-64 Architecture Software Developer's Manual*, 2000. Instruction Set Reference.
4. B. Li and R. Gupta. Bit section instruction set extension of ARM for embedded applications. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems CASES 2002*, pages 69–78. ACM, Oct. 2002.
5. Motorola, Inc. *DSP563CCC Motorola DSP56300 Family Optimizing C Compiler User's Manual*. Motorola, Inc, Austin, TX, USA, 19??
6. Motorola, Inc. *MOTOROLA M68000 Family Programmer's Reference Manual*. Motorola, Inc, 1992.
7. Sun. *C User's Guide*. Sun Microsystems, Inc, Palo Alto, CA, USA, revision a edition, May 2000.
8. Texas Instruments. *TMS370 and TMS370C8 8-Bit Microcontroller Family Optimizing C Compiler Users' Guide*. Texas Instruments, spnu022c edition, Apr. 1996.