Unify string representation functions

Jens Gustedt

ISO/IEC JTC 1/SC 22/WG14 **N 2360**

March 30, 2019 **v 1**

Unify string representation functions
proposal for C2x

Jens Gustedt
INRIA and ICube, Université de Strasbourg, France

The recent integration of TS 18661-1 has added string representation functions for floating point types. The added functions are not entirely satisfactory, because they have names that do not fall into none of the established schemes, because they allow for security breaches via dynamic format arguments, and because they do not provide a type-generic interface. We propose to extend the function interface to all scalar types, to close the security gap and to provide a type-generic macro that combines all the functions.

1. INTRODUCTION

The integration of TS 18661-1 has added three functions that provide string representations for floating point types, namely **strfromd**, **strfromf** and **strfroml** by means of a call **snprintf**(buffer, len, lformat, x). This paper builds upon N2359 that already renames the functions to tostrd, tostrf and tostrl. Such a naming scheme is still not satisfactory, since it could lead to confusion about the source type: tostrd could also stand for **int**, tostrl for **long**. In particular, this scheme is not easy to extend to other arithmetic types. These functions also inherit the potential security problem from **snprintf** that allows the format string to be dynamic. Thus, incorrect format strings can not always be detected at compile time leading to possible stack exploits. For simple cases that print the representation of one single scalar data of a known type, all specification errors should be detected at translation time and diagnosed.
The new interfaces are also unnecessarily restrictive concerning the possible types. There is no apparent reason why such interfaces cannot be provided for all scalar types. In addition, the interfaces should be easy to use with convenient default formats. Therefore we propose to make the format parameter optional as the last of four arguments to the functions.
We propose:

(1) A naming scheme that is guided by the **printf** specifiers, to extend to all types that are provided by this function, and to determine a default specifier for each of the functions/types.
(2) To extend the scheme to complex types, if provided by the implementation.
(3) To extend the scheme to data pointer types.
(4) To regroup all these functions into a type-generic macro that allows a simple and normalized access to string representations of all arithmetic data types.

2. REAL TYPES

The **printf** functions provide conversion specifiers for all standard real types other than bool. Thus there is no reason *not* to provide functions for all of the supported types and thus to provide a unified interface to print string representations.
Seen like that, it is much easier to have a naming scheme for such functions that combines a default conversion specifier R with the necessary length specifier L, if any, for the type. We combine these into function names of the form tostrRL, similar as this is done for the other conversion direction, namely string to arithmetic type. In analogy of the **strtoull** function we use tostrull for a function that converts **unsigned long long** to a string representation. R is "i" for signed integer types, "u" for unsigned types and "g" for floating-point points. This should provide sufficient defaults for a decimal representation of all standard real types. We chose "i" for signed integers, because it is easier to distinguish than "d" which

also could refer to **double**. The new clause 7.22.1.5 (*cf.* appendix) has a comprehensive table of all possible combinations for R and L for real types.
In addition to the standard real types, we also expect implementations to provide functions for all mandatory real types that are not covered by the above and also to extend this to the types as described in Clause 7.20 (<stdint.h>).
The conversion specifier "c" is special, because it is not possible to distinguish plain and wide characters via their type. In particular, integer character constants ('A') have type **int** and wide character constants (L'A') have type **wchar_t**. So for implementations for which **wchar_t** is in fact **int**, such constants are not distinguishable. Since in general we can't know which integer type is used as **wchar_t** we allow "c" for all integer types. For these functions the width of the type decides if conversion "c" or "lc" is applied.
For implementations that define **wchar_t** to be **int**, and that have different encodings for the base source character set and for the extended source character set (**__STDC_MB_MIGHT_NEQ_WC__** defined) the usage of "c" might lead to surprising results. But we don't think that there would be an easy way out for that special case.
With the exception of **float**, all specifiers L from the function name specify the length modifier that is used together with the rest of the format. For **float** the situation is special, because it is not directly supported by **printf**. Here we introduce the artificial value 'H'. Thus, the latter is not used as a length modifier. The alternative would have been to have no function for **float**, but it seems that there in IEC 60559 is a need for a function that reduces excess precision in that case.
To avoid security gaps, the requirement is to use string literals for the format or to even leave it out if there is no ambiguity. The proposed text below shows simple means to implement and to check these properties during compilation. This requirement implies that all format and type information can be checked for consistency at translation time, and thus diagnostics can be enforced for all violations.

Examples

```
size_t const mlen = tostrull(0, 0, ULLONG_MAX, "o") + 2; // + sign or prefix
char buffer[mlen+1];
tostrull(buffer, mlen, 7, "#o");          // converts to unsigned, prints "07"
tostrill(buffer, mlen, LLONG_MIN);        // decimal, including the - sign
tostrill(buffer, mlen, ULLONG_MAX);       // invalid conversion, overflow
tostrull(buffer, mlen, LLONG_MIN);        // converts to unsigned, large

tostrc(buffer, mlen, 'A');                // prints "A"
tostrc(buffer, mlen, L'A');               // depends on __STDC_MB_MIGHT_NEQ_WC__
tostrcl(buffer, mlen, 'A');               // depends on __STDC_MB_MIGHT_NEQ_WC__
tostrcl(buffer, mlen, L'A');              // prints "A"

tostruj(buffer, mlen, 'A');               // prints the basic numerical code
tostruj(buffer, mlen, L'A');              // prints the extended numerical code
tostruj(buffer, mlen, L'A', "c");         // prints "A"
tostruj(buffer, mlen, 'A', "c");          // depends on __STDC_MB_MIGHT_NEQ_WC__

static char format[] = "#.37x";           // character array object
tostrul(buffer, mlen, LLONG_MAX, format); // diagnosis: not a string literal
```

3. COMPLEX TYPES

If the implementation supports complex types, we require that analogous functions for them are defined where the name has a tostrc prefix, so tostrcg, tostrcgH and tostrcgL. Complex number are then printed as 1.0+2.0i or 1.0-2.0i, where real and imaginary part are printed according to the format and the same rules as real floating-point types.

4. POINTER TYPES

With **snprintf**, data pointer values can be printed with a "p" conversion specifier if they are converted to **void***, first. This is to be distinguished from the "s" conversion specifier which views a pointer as the address of a string (plain or wide) and which then prints the string.

There is a particular difficulty to interface wide character strings, because **wchar_t** is not a separate type, but only a semantic type on top of the usual base types. Therefore a **wchar_t** pointer can be a legimitate integer array or represent a wide character string without a visible difference in the type system.

We propose two different functions that take pointer values, both after conversion to a **void** pointer.

	"p"	"s"
tostrs	pointer value	plain character string (default)
tostrp	pointer value (default)	wide character string

5. A TYPE GENERIC INTERFACE

For the user's comfort, all of these functions can be folded together in a type-generic macro that covers all scalar types. Because semantic types may or may not be covered by the basic types, the details which particular function to chose for a given type become a bit messy. We prefer the use of the functions for basic types over the functions that depend on semantic types (tostrij, tostrit, tostruj, tostruz).

The choices are such that "c" specifiers will always attempt to print a plain or wide character argument, and "s" will always interpret the pointer as a pointer to a string (plain or wide). Still, results for implementations that define **__STDC_MB_MIGHT_NEQ_WC__** can be a bit surprising.

Examples

```
size_t const mlen = tostr(0, 0, ULLONG_MAX, "o") + 2; // + sign or prefix
char buffer[mlen+1];
tostr(buffer, mlen, 7, "#o");           // invalid specifier for int
tostr(buffer, mlen, 7u, "#o");          // prints "07"
tostr(buffer, mlen, LLONG_MIN);         // decimal, including the - sign
tostr(buffer, mlen, ULLONG_MAX);        // decimal

tostr(buffer, mlen, 'A');               // prints the basic numerical code
tostr(buffer, mlen, L'A');              // prints the extended numerical code
tostr(buffer, mlen, L'A', "c");         // prints "A"
tostr(buffer, mlen, 'A', "c");          // depends on __STDC_MB_MIGHT_NEQ_WC__

tostr(buffer, mlen, "word");            // copies the string into buffer
tostr(buffer, mlen, L"wörd");           // prints a pointer value
tostr(buffer, mlen, u8"wörd");          // copies the utf8 string into buffer
tostr(buffer, mlen, "word", "p");       // prints a pointer value
tostr(buffer, mlen, L"wörd", "p");      // prints a pointer value
tostr(buffer, mlen, u8"wörd", "p");     // prints a pointer value
tostr(buffer, mlen, "word", "s");       // copies the string into buffer
tostr(buffer, mlen, L"wörd", "s");      // prints a multi-byte string
tostr(buffer, mlen, u8"wörd", "s");     // copies the utf8 string into buffer

static char format[] = "#.37x";         // character array object
tostr(buffer, mlen, ULLONG_MAX, format);// diagnosis: not a string literal
```

6. PROBLEMS

Besides the problems for character types mentioned above, the **printf** family suffers from the fact that the decimal point for floating-point representations depends on the locale. It would be good if we could get rid of that dependency at least for the "a" and "A" conversions.

Appendix: pages with diffmarks of the proposed changes against the N2359 proposed changes.

The following page numbers are from the particular snapshot and may vary once the changes are integrated.

outside external declarations or preceding all explicit declarations and statements inside a compound statement. When outside external declarations, the pragma takes effect from its occurrence until another **FENV_ROUND** pragma is encountered, or until the end of the translation unit. When inside a compound statement, the pragma takes effect from its occurrence until another **FENV_ROUND** pragma is encountered (including within a nested compound statement), or until the end of the compound statement; at the end of a compound statement the static rounding mode is restored to its condition just before the compound statement. If this pragma is used in any other context, its behavior is undefined.

3 *direction* shall be one of the rounding direction macro names defined in 7.6, or **FE_DYNAMIC**. If any other value is specified, the behavior is undefined. If no **FENV_ROUND** pragma is in effect, or the specified constant rounding mode is **FE_DYNAMIC**, rounding is according to the mode specified by the dynamic floating-point environment, which is the dynamic rounding mode that was established either at thread creation or by a call to **fesetround**, **fesetmode**, **fesetenv**, or **feupdateenv**. If the **FE_DYNAMIC** mode is specified and **FENV_ACCESS** is "off", the translator may assume that the default rounding mode is in effect.

4 Within the scope of an **FENV_ROUND** pragma establishing a mode other than **FE_DYNAMIC**, all floating-point operators, implicit conversions (including the conversion of a value represented in a format wider than its semantic types to its semantic type, as done by classification macros), and invocations of functions indicated in the table below, for which macro replacement has not been suppressed (7.1.4), shall be evaluated according to the specified constant rounding mode (as though no constant mode was specified and the corresponding dynamic rounding mode had been established by a call to **fesetround**). Invocations of functions for which macro replacement has been suppressed and invocations of functions other than those indicated in the table below shall not be affected by constant rounding modes – they are affected by (and affect) only the dynamic mode. Floating constants (6.4.4.2) that occur in the scope of a constant rounding mode shall be interpreted according to that mode.

Functions affected by constant rounding modes

Header	Function groups
<math.h>	**acos**, **asin**, **atan**, **atan2**
<math.h>	**cos**, **sin**, **tan**
<math.h>	**acosh**, **asinh**, **atanh**
<math.h>	**cosh**, **sinh**, **tanh**
<math.h>	**exp**, **exp2**, **expm1**
<math.h>	**log**, **log10**, **log1p**, **log2**
<math.h>	**scalbn**, **scalbln**, **ldexp**
<math.h>	**cbrt**, **hypot**, **pow**, **sqrt**
<math.h>	**erf**, **erfc**
<math.h>	**lgamma**, **tgamma**
<math.h>	**rint**, **nearbyint**, **lrint**, **llrint**
<math.h>	**fdim**
<math.h>	**fma**
<math.h>	**fadd**, **daddl**, **fsub**, **dsubl**, **fmul**, **dmull**, **fdiv**, **ddivl**, **ffma**, **dfmal**, **fsqrt**, **dsqrtl**
<stdlib.h>	**atof**, **strtod**, **strtof**, **strtold**, ~~**tostrd**~~, ~~**tostrf**~~, ~~**tostrl**~~ **tostrg**, **tostrgH**, **tostrgL**
<wchar.h>	**wcstod**, **wcstof**, **wcstold**
<stdio.h>	**printf** and **scanf** families
<wchar.h>	**wprintf** and **wscanf** families

Each <math.h> function listed in the table above indicates the family of functions of all supported types (for example, **acosf** and **acosl** as well as **acos**).

5 **NOTE** Constant rounding modes (other than **FE_DYNAMIC**) could be implemented using dynamic rounding modes as illustrated in the following example:

Returns

3 The **atof** function returns the converted value.

Forward references: the **strtod**, **strtof**, and **strtold** functions (7.22.1.3).

7.22.1.2 The atoi, atol, and atoll functions

Synopsis

1
```
#include <stdlib.h>
int atoi(const char *nptr);
long int atol(const char *nptr);
long long int atoll(const char *nptr);
```

Description

2 The **atoi**, **atol**, and **atoll** functions convert the initial portion of the string pointed to by nptr to **int**, **long int**, and **long long int** representation, respectively. Except for the behavior on error, they are equivalent to

```
atoi:  (int)strtol(nptr, (char **)NULL, 10)
atol:  strtol(nptr, (char **)NULL, 10)
atoll: strtoll(nptr, (char **)NULL, 10)
```

Returns

3 The **atoi**, **atol**, and **atoll** functions return the converted value.

Forward references: the **strtol**, **strtoll**, **strtoul**, and **strtoull** functions (7.22.1.4).

~~**Description**~~

~~The tostrd, tostrf, and tostrl functions are equivalent to **snprintf**(s, n, format, fp) (), except that the format string shall only contain the character %, an optional precision that does not contain an asterisk *, and one of the conversion specifiers a, A, e, E, f, F, g, or G, which applies to the type (**double**, **float**, or **long double**) indicated by the function suffix (rather than by a length modifier).~~

~~**Returns**~~

~~The tostrd, tostrf, and tostrl functions return the number of characters that would have been written had n been sufficiently large, not counting the terminating null character. Thus, the null-terminated output has been completely written if and only if the returned value is less than n.~~

7.22.1.3 The strtod, strtof, and strtold functions

Synopsis

1
```
#include <stdlib.h>
double strtod(const char *restrict nptr, char **restrict endptr);
float strtof(const char *restrict nptr, char **restrict endptr);
long double strtold(const char *restrict nptr, char **restrict endptr);
```

Description

2 The **strtod**, **strtof**, and **strtold** functions convert the initial portion of the string pointed to by nptr to **double**, **float**, and **long double** representation, respectively. First, they decompose the input string into three parts: an initial, possibly empty, sequence of white-space characters, a subject sequence resembling a floating-point constant or representing an infinity or NaN; and a final string of one or more unrecognized characters, including the terminating null character of the input string. Then, they attempt to convert the subject sequence to a floating-point number, and return the result.

3 The expected form of the subject sequence is an optional plus or minus sign, then one of the following:

returned (according to the return type and sign of the value, if any), and the value of the macro **ERANGE** is stored in **errno**.

7.22.1.5 String representation functions for real types

Synopsis

1
```
#include <stdlib.h>
int tostrRL(char*restrict s, size_t n, real-type x);
int tostrRL(char*restrict s, size_t n, real-type x, const char*restrict format);
```

Description

2 The string representation functions for real types are equivalent to **snprintf**(s, n, lformat, x) (7.21.6.5). Their name is composed of the prefix **tostr**, a conversion specifier *R* and a possibly empty length modifier *L* as given in the following table:

R	*L*	*real-type*	function	possible *C*
c		**char**	**tostrc**	c
c	l	**wint_t**	**tostrcl**	c
	hh	**signed char**	**tostrihh**	
	h	**signed short**	**tostrih**	
		signed int	**tostri**	
i	l	**signed long**	**tostril**	c, d, i
	ll	**signed long long**	**tostrill**	
	j	**uintmax_t**	**tostrij**	
	t	**ptrdiff_t**	**tostrit**	
	hh	**unsigned char**	**tostruhh**	
	h	**unsigned short**	**tostruh**	
		unsigned int	**tostru**	
u	l	**unsigned long**	**tostrul**	c, o, u, x, X
	ll	**unsigned long long**	**tostrull**	
	j	**intmax_t**	**tostruj**	
	z	**size_t**	**tostruz**	
	H	**float**	**tostrgH**	
g		**double**	**tostrg**	a, A, e, E, f, F, g, G
	L	**long double**	**tostrgL**	

format shall be a string literal that contains any of the following optional elements in the given order: a + character *S*, a # character *F*, a precision *P* composed of a . character followed by a sequence of decimal digits, and a conversion specifier *C* as indicated in the table above. If omitted, *C* defaults to *R*. For **tostrgH** the format string lformat is then "%*SFPC*", and "%*SFPLC*" otherwise.

3 It is implementation-defined if additional real types are supported, but the set of covered types includes at least all mandatory integer types other than **_Bool** and the supported among the optional integer types described in <stdint.h> . If provided, the additional functions follow analogous naming schemes and definitions. In particular, they use an *R* of *i*, u or g for signed, unsigned and floating point types, respectively.

4 The specifier c can be used with all integer types. If this specifier is used, *F* and *S* shall be empty and x shall not be negative. For **tostrc**, x shall not be greater than **CHAR_MAX**, for **tostrcl** it shall not be greater than **WINT_MAX**. Special rules apply for functions other than **tostrc** and **tostrcl** if called with *C* equal to c. If all positive values of **wchar_t** are representable in *real-type*, the call is as if replaced by a call to **tostrcl**, otherwise a call to **tostrc**.

5 Unless a macro definition is suppressed in order to access an actual function the following apply: format is optional and as if replaced by an empty string literal if it is omitted. A diagnostic shall be issued if a format argument is provided that is not a string literal, if format is not of the required form, or if lformat is not a valid **printf** format for *real-type*.

Returns

6 The string representation functions for real types return the number of characters that would have been written had `n` been sufficiently large, not counting the terminating null character. Thus, the null-terminated output has been completely written if and only if the returned value is less than `n`.

7 **NOTE 1** Enforcing a string literal as an argument to `format` ensures that decisions and diagnostics about the formatting may take place during translation and avoids that dynamically constructed **`snprintf`** format strings may breach security. Implementations can supply the optional argument `format` and force a string literal for it by providing additional macros for each of the functions analogous to the following.

```
#define tostrg(B, N, ...)             tostrg4(B, N, __VA_ARGS__, "", )
#define tostrg4(B, N, ...)            tostrg5(B, N, __VA_ARGS__)
#define tostrg5(B, N, X, FORMAT, ...) tostrg(B, N, X, "" FORMAT "")
```

8 **NOTE 2** If called with a floating point argument that has an evaluation format with a greater precision than its type, according to 5.2.4.2.2 the functions **`tostrg`**, **`tostrgH`** and **`tostrgL`** will convert such an argument to the precision of their respective parameter type before printing.

9 **EXAMPLE** The following shows examples of the use of the string representation functions:

```
size_t const mlen = tostrull(0, 0, ULLONG_MAX, "o") + 2; // + sign or prefix
char buffer[mlen+1];
tostrull(buffer, mlen, 7, "#o");          // converts to unsigned, prints "07"
tostrill(buffer, mlen, LLONG_MIN);        // decimal, including the - sign
tostrill(buffer, mlen, ULLONG_MAX);       // invalid conversion, overflow
tostrull(buffer, mlen, LLONG_MIN);        // converts to unsigned, large

tostrc(buffer, mlen, 'A');                // prints "A"
tostrc(buffer, mlen, L'A');               // depends on __STDC_MB_MIGHT_NEQ_WC__
tostrcl(buffer, mlen, 'A');               // depends on __STDC_MB_MIGHT_NEQ_WC__
tostrcl(buffer, mlen, L'A');              // prints "A"

tostruj(buffer, mlen, 'A');               // prints the basic source encoding of A
tostruj(buffer, mlen, L'A');              // prints the extended source encoding
tostruj(buffer, mlen, L'A', "c");         // prints "A"
tostruj(buffer, mlen, 'A', "c");          // depends on __STDC_MB_MIGHT_NEQ_WC__

static char format[] = "#.37x";           // character array object
tostrul(buffer, mlen, ULLONG_MAX, format);// diagnosis: not a string literal
```

7.22.1.6 String representation functions for complex types

Synopsis

1

```
#include <stdlib.h>
#ifndef __STDC_NO_COMPLEX__
int tostrcRL(char*restrict s, size_t n, complex-type x);
int tostrcRL(char*restrict s, size_t n, complex-type x, const char*restrict format);
#endif
```

Description

2 The string representation functions for complex types are only supported if the implementation supports complex types.

3 The string representation functions for complex types are equivalent to **`snprintf`**`(s, n, lformat, xreal, ximag)` (7.21.6.5), where `xreal` and `ximag` are the real and imaginary part of `x`, respectively. If a real type has a corresponding complex type and is supported by a function **`tostr`***RL* as of 7.22.1.5, there is a corresponding function **`tostrc`***RL* for the complex type. `format` shall follow the same rules as for **`tostr`***RL* to determine *S*, *F*, *P* and *C*. `lformat` is then composed of the corresponding format for **`tostr`***RL*, followed by a second copy of that format, but with a mandatory `+` for *S*, and then followed by the letter `i`.

4 Unless a macro definition is suppressed in order to access an actual function the following apply: `format` is optional and as if replaced by an empty string literal if it is omitted. A diagnostic shall be issued if a `format` argument is provided that is not a string literal or if `format` is not of the required form.

Returns

5 The same rules as for the functions in 7.22.1.5 apply.

6 **NOTE 1** Thus, for the three standard complex types, *complex-type*, function name and `lformat` used are then corresponding to the following table:

complex-type	function	`lformat`
`float _Complex`	`tostrcgH`	`"%`*SFPC*`%+`*FPC*`i"`
`double _Complex`	`tostrcg`	`"%`*SFPC*`%+`*FPC*`i"`
`long double _Complex`	`tostrcgL`	`"%`*SFPL*`C%+`*FPL*`Ci"`

7 **NOTE 2** If called with a complex floating point argument that has an evaluation format with a greater precision than its type, according to 5.2.4.2.2 the string representation functions for complex types will convert such an argument to the precision of their respective parameter type before printing.

7.22.1.7 String representation functions for pointer types

Synopsis

1
```
#include <stdlib.h>
int tostrR(char*restrict s, size_t n, void const volatile*restrict x);
int tostrR(char*restrict s, size_t n, void const volatile*restrict x,
           const char*restrict format);
```

Description

2 The string representation functions for pointer types are equivalent to **`snprintf`**`(s, n, lformat, x)` (7.21.6.5). Their name is composed of the prefix **`tostr`** and a conversion specifier *R* that is either `s` or `p`. `format` shall be a string literal that contains any of the following optional elements in the given order: a precision *P* composed of a `.` character followed by a sequence of decimal digits, and a conversion specifier *C* that is `s` or `p`. If omitted, *C* defaults to *R*.

3 If *C* is `p`, the format string `lformat` is `"%`*P*`p"`. Otherwise, *C* is `s`, `x` shall not be null and shall refer to a string (**`tostrs`**) or wide character string (**`tostrp`**), respectively. Then, for **`tostrs`**, `x` is converted to **`char const volatile`**`*` before the call to **`snprintf`** and `lformat` is `"%`*P*`s"`; for **`tostrp`**, `x` is converted to **`wchar_t const volatile`**`*` before the call to **`snprintf`** and `lformat` is `"%`*P*`ls"`.

4 Unless a macro definition is suppressed in order to access an actual function the following apply: `format` is optional and as if replaced by an empty string literal if it is omitted. A diagnostic shall be issued if a `format` argument is provided that is not a string literal or if `format` is not of the required form.

Returns

5 The same rules as for the functions in 7.22.1.5 apply.

7.22.1.8 The `tostr` type-generic macro

Synopsis

1
```
#include <stdlib.h>
int tostr(char*restrict s, size_t n, scalar-type x);
int tostr(char*restrict s, size_t n, scalar-type x, const char*restrict format);
```

Description

2 The **`tostr`** type-generic macro invokes a selected string representation function as mandated by 7.22.1.5, 7.22.1.6, and 7.22.1.7. The macro argument corresponding to `format` shall be omitted or a string literal. If *scalar-type* is **`_Bool`**, a pointer to a qualified or unqualified version of **`char`**, or a pointer to another data type, the selected function is **`tostri`**, **`tostrs`**, **`tostrp`**, respectively. Otherwise, a function that receives *scalar-type* as third argument is selected. If there are several

such functions the first function with *L* in the following list is selected: empty, `l`, `ll`, `j`, `z`, `t`. The selected function is then called with the same arguments.

3 A diagnostic shall be issued if `x` has a type that is not supported, if a `format` argument is provided that is not a string literal or if it is not of the required form for the selected function.

4 **NOTE 1** An implementation that does support complex types and covers all mandated and optional arithmetic types by the standard arithmetic types can implement **`tostr`** as:

```
#define tostr(B, N, ...)                tostr4(B, N, __VA_ARGS__, "", )
#define tostr4(B, N, ...)               tostr5(B, N, __VA_ARGS__)
#define tostr5(B, N, X, FORMAT, ...)                \
  _Generic(X,                                       \
    char:                   tostrc,                 \
    signed char:            tostrihh,               \
    signed short:           tostrih,                \
    signed:                 tostri,                 \
    signed long:            tostril,                \
    signed long long:       tostrill,               \
    _Bool:                  tostri,                 \
    unsigned char:          tostruhh,               \
    unsigned short:         tostruh,                \
    unsigned                tostru,                 \
    unsigned long:          tostrul,                \
    unsigned long long:     tostrull,               \
    float:                  tostrgH,                \
    double:                 tostrg,                 \
    long double:            tostrgL,                \
    double _Complex:        tostrcg,                \
    float _Complex:         tostrcgH,               \
    long double _Complex:   tostrcgL,               \
    char*:                  tostrs,                 \
    char const*:            tostrs,                 \
    char volatile*:         tostrs,                 \
    char const volatile*:   tostrs,                 \
    default:                tostrp)                 \
      (B, N, X, "" FORMAT "")
```

The behavior of the shown macro definition for function pointer arguments depends on the behavior of the implementation for implicit conversion of function pointers to **`void`**`*`.

5 **NOTE 2** The choices for the string representation functions are such that per default, **`char`** pointers are interpreted as strings and all other data pointer types, even to **`wchar_t`**, print the pointer value. The value of a pointer to **`char`** can be printed by using the `p` conversion specifier. Wide character strings can be printed by using the `s` conversion specifier. Thus, a call **`tostr`**`(buffer, n, str, "s")` can be used to represent `str` as multi-byte string regardless if `str` is a string or wide character string.

6 **NOTE 3** Similarly, only **`char`** values are interpreted as a character encoding. All other integer values, even if they are integer character constants (type **`int`**) or are of type **`wchar_t`**, are printed as numbers. Wide or plain characters may be printed by using the `c` conversion specifier. Then, if **`__STDC_MB_MIGHT_NEQ_WC__`** is not defined, a call **`tostr`**`(buffer, n, chr, "c")` can be used to represent `chr` as a multi-byte string regardless if `chr` is a plain character, an integer character constant or a wide character.

7 **NOTE 4** If called with a floating point argument that has an evaluation format with a greater precision than its type, according to 5.2.4.2.2 the type-generic macro **`tostr`** will convert such an argument to the precision of the argument type before printing.

8 **EXAMPLE** The following shows examples of the use of the generic string representation macro:

```
size_t const mlen = tostr(0, 0, ULLONG_MAX, "o") + 2; // + sign or prefix
char buffer[mlen+1];
tostr(buffer, mlen, 7, "#o");           // invalid specifier for int
tostr(buffer, mlen, 7u, "#o");          // prints "07"
tostr(buffer, mlen, LLONG_MIN);         // decimal, including the - sign
tostr(buffer, mlen, ULLONG_MAX);        // decimal
```

```
tostr(buffer, mlen, 'A');               // prints the basic source encoding
tostr(buffer, mlen, L'A');              // prints the extended source encoding
tostr(buffer, mlen, L'A', "c");         // prints "A"
tostr(buffer, mlen, 'A', "c");          // depends on __STDC_MB_MIGHT_NEQ_WC__

tostr(buffer, mlen, "word");            // copies the string into buffer
tostr(buffer, mlen, L"w\u0153rd");      // prints a pointer value
tostr(buffer, mlen, u8"w\u0153rd");     // copies the utf8 string into buffer
tostr(buffer, mlen, "word", "p");       // prints a pointer value
tostr(buffer, mlen, L"w\u0153rd", "p"); // prints a pointer value
tostr(buffer, mlen, u8"w\u0153rd", "p");// prints a pointer value
tostr(buffer, mlen, "word", "s");       // copies the string into buffer
tostr(buffer, mlen, L"w\u0153rd", "s"); // prints a correct multi-byte string
tostr(buffer, mlen, u8"w\u0153rd", "s");// copies the utf8 string into buffer

static char format[] = "#.37x";         // character array object
tostr(buffer, mlen, ULLONG_MAX, format);// diagnosis: not a string literal
```

7.22.2 Pseudo-random sequence generation functions

7.22.2.1 The `rand` function

Synopsis

1
```
#include <stdlib.h>
int rand(void);
```

Description

2 The **rand** function computes a sequence of pseudo-random integers in the range 0 to **RAND_MAX** inclusive.

3 The **rand** function is not required to avoid data races with other calls to pseudo-random sequence generation functions. The implementation shall behave as if no library function calls the **rand** function.

Recommended practice

4 There are no guarantees as to the quality of the random sequence produced and some implementations are known to produce sequences with distressingly non-random low-order bits. Applications with particular requirements should use a generator that is known to be sufficient for their needs.

Returns

5 The **rand** function returns a pseudo-random integer.

Environmental limits

6 The value of the **RAND_MAX** macro shall be at least 32767.

7.22.2.2 The `srand` function

Synopsis

1
```
#include <stdlib.h>
void srand(unsigned int seed);
```

Description

2 The **srand** function uses the argument as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to **rand**. If **srand** is then called with the same seed value, the sequence of pseudo-random numbers shall be repeated. If **rand** is called before any calls to **srand** have been made, the same sequence shall be generated as when **srand** is first called with a seed value of 1.

division	`/`, **`fdiv`**, **`fdivl`**, **`ddivl`**	6.5.5, 7.12.14.4, F.10.11
squareRoot	**`sqrt`**, **`fsqrt`**, **`fsqrtl`**, **`dsqrtl`**	7.12.7.5, F.10.4.5, 7.12.14.6, F.10.11
fusedMultiplyAdd	**`fma`**, **`ffma`**, **`ffmal`**, **`dfmal`**	7.12.13.1, F.10.10.1, 7.12.14.5, F.10.11
convertFromInt	cast and implicit conversion	6.3.1.4, 6.5.4
convertToIntegerTiesToEven convertToIntegerTowardZero convertToIntegerTowardPositive convertToIntegerTowardNegative	**`toint`**, **`touint`**	7.12.9.10, F.10.6.10
convertToIntegerTiesToAway	**`toint`**, **`touint`**, **`lround`**, **`llround`**	7.12.9.10, F.10.6.10, 7.12.9.7, F.10.6.7
convertToIntegerExactTiesToEven convertToIntegerExactTowardZero convertToIntegerExactTowardPositive convertToIntegerExactTowardNegative convertToIntegerExactTiesToAway	**`tointx`**, **`touintx`**	7.12.9.11, F.10.6.11
convertFormat - different formats	cast and implicit conversions	6.3.1.5, 6.5.4
convertFormat - same format	**`canonicalize`**	7.12.11.7, F.10.8.7
convertFromDecimalCharacter	**`strtod`**, **`wcstod`**, **`scanf`**, **`wscanf`**, decimal floating constants	7.22.1.3, 7.29.4.1.1, 7.21.6.4, 7.29.2.12, F.5
convertToDecimalCharacter	**`printf`**, **`wprintf`**, ~~`tostrd`~~ **`tostrg`**	7.21.6.3, 7.29.2.11, 7.22.1.5, F.5
convertFromHexCharacter	**`strtod`**, **`wcstod`**, **`scanf`**, **`wscanf`**, hexadecimal floating constants	7.22.1.3, 7.29.4.1.1, 7.21.6.4, 7.29.2.12, F.5
convertToHexCharacter	**`printf`**, **`wprintf`**, ~~`tostrd`~~ **`tostrg`**	7.21.6.3, 7.29.2.11, 7.22.1.5, F.5
copy	**`memcpy`**, **`memmove`**	7.24.2.1, 7.24.2.2
negate	`-(x)`	6.5.3.3
abs	**`fabs`**	7.12.7.2, F.10.4.2
copySign	**`copysign`**	7.12.11.1, F.10.8.1
compareQuietEqual	`==`	6.5.9, F.9.3
compareQuietNotEqual	`!=`	6.5.9, F.9.3
compareSignalingEqual	**`iseqsig`**	7.12.15.7, F.10.14.1
compareSignalingGreater	`>`	6.5.8, F.9.3
compareSignalingGreaterEqual	`>=`	6.5.8, F.9.3
compareSignalingLess	`<`	6.5.8, F.9.3
compareSignalingLessEqual	`<=`	6.5.8, F.9.3
compareSignalingNotEqual	`!` **`iseqsig`**`(x)`	7.12.15.7, F.10.14.1
compareSignalingNotGreater	`!(x > y)`	6.5.8, F.9.3
compareSignalingLessUnordered	`!(x >= y)`	6.5.8, F.9.3
compareSignalingNotLess	`!(x < y)`	6.5.8, F.9.3
compareSignalingGreaterUnordered	`!(x <= y)`	6.5.8, F.9.3
compareQuietGreater	**`isgreater`**	7.12.15.1
compareQuietGreaterEqual	**`isgreaterequal`**	7.12.15.2
compareQuietLess	**`isless`**	7.12.15.3
compareQuietLessEqual	**`islessequal`**	7.12.15.4
compareQuietUnordered	**`isunordered`**	7.12.15.6
compareQuietNotGreater	`!` **`isgreater`**`(x, y)`	7.12.15.1
compareQuietLessUnordered	`!` **`isgreaterequal`**`(x, y)`	7.12.15.2
compareQuietNotLess	`!` **`isless`**`(x, y)`	7.12.15.3