- 15 nesting levels of compound statements, iteration control structures, and selection control structures
- 8 nesting levels of conditional inclusion
- 12 pointer, array, and function declarators (in any combinations) modifying an arithmetic, a structure, a union, or an incomplete type in a declaration
- 31 declarators nested by parentheses within a full declarator
- 32 expressions nested by parentheses within a full expression
- 31 significant initial characters in an internal identifier or a macro name
- 6 significant initial characters in an external identifier
- 511 external identifiers in one translation unit
- 127 identifiers with block scope declared in one block
- 1024 macro identifiers simultaneously defined in one translation unit
- 31 parameters in one function definition
- 31 arguments in one function call
- 31 parameters in one macro definition
- 31 arguments in one macro invocation
- 509 characters in a logical source line
- 509 characters in a character string literal or wide string literal (after concatenation)
- 32767 bytes in an object (in a hosted environment only)
- 8 nesting levels for **#included** files
- 257 **case** labels for a **switch** statement (excluding those for any nested **switch** statements)
- 127 members in a single structure or union
- 127 enumeration constants in a single enumeration
- 15 levels of nested structure or union definitions in a single struct-declaration-list

2.2.4.2 Numerical limits

A conforming implementation shall document all the limits specified in this section, which shall be specified in the headers **<limits.h>** and **<float.h>**.

Sizes of integral types <limits.h>

The values given below shall be replaced by constant expressions suitable for use in **#if** preprocessing directives. Their implementation-defined values shall be equal or greater in magnitude (absolute value) to those shown, with the same sign.

- maximum number of bits for smallest object that is not a bit-field (byte)
  CHAR_BIT 8
- minimum value for an object of type **signed char**
  SCHAR_MIN -127
- maximum value for an object of type **signed char**
  SCHAR_MAX +127
- maximum value for an object of type **unsigned char**
  UCHAR_MAX 255

- minimum value for an object of type `char`
  `CHAR_MIN` *see below*
- maximum value for an object of type `char`
  `CHAR_MAX` *see below*
- maximum number of bytes in a multibyte character, for any supported locale
  `MB_LEN_MAX` `1`
- minimum value for an object of type `short int`
  `SHRT_MIN` `-32767`
- maximum value for an object of type `short int`
  `SHRT_MAX` `+32767`
- maximum value for an object of type `unsigned short int`
  `USHRT_MAX` `65535`
- minimum value for an object of type `int`
  `INT_MIN` `-32767`
- maximum value for an object of type `int`
  `INT_MAX` `+32767`
- maximum value for an object of type `unsigned int`
  `UINT_MAX` `65535`
- minimum value for an object of type `long int`
  `LONG_MIN` `-2147483647`
- maximum value for an object of type `long int`
  `LONG_MAX` `+2147483647`
- maximum value for an object of type `unsigned long int`
  `ULONG_MAX` `4294967295`

If the value of an object of type `char` sign-extends when used in an expression, the value of `CHAR_MIN` shall be the same as that of `SCHAR_MIN` and the value of `CHAR_MAX` shall be the same as that of `SCHAR_MAX`. If the value of an object of type `char` does not sign-extend when used in an expression, the value of `CHAR_MIN` shall be 0 and the value of `CHAR_MAX` shall be the same as that of `UCHAR_MAX`.[7]

Characteristics of floating types `<float.h>`

The characteristics of floating types are defined in terms of a model that describes a representation of floating-point numbers and values that provide information about an implementation's floating-point arithmetic. The following parameters are used to define the model for each floating-point type:

s	sign (± 1)
b	base or radix of exponent representation (an integer > 1)
e	exponent (an integer between a minimum e_{min} and a maximum e_{max})
p	precision (the number of base-b digits in the mantissa)
f_k	nonnegative integers less than b (the mantissa digits)

A normalized floating-point number x ($f_1 > 0$ if $x \neq 0$) is defined by the following model:[8]

7. See §3.1.2.5.

8. This model precludes floating-point representations other than sign-magnitude.

$$x = s \times b^e \times \sum_{k=1}^{p} f_k \times b^{-k}, \quad e_{\min} \le e \le e_{\max}$$

Of the values in the **<float.h>** header, **FLT_RADIX** shall be a constant expression suitable for use in **#if** preprocessing directives; all other values need not be constant expressions. All except **FLT_RADIX** and **FLT_ROUNDS** have separate names for all three floating-point types. The floating-point model representation is provided for all values except **FLT_ROUNDS**.

The rounding mode for floating-point addition is characterized by the value of **FLT_ROUNDS**:

-1	indeterminable
0	toward zero
1	to nearest
2	toward positive infinity
3	toward negative infinity

All other values for **FLT_ROUNDS** characterize implementation-defined rounding behavior.

The values given in the following list shall be replaced by implementation-defined expressions that shall be equal or greater in magnitude (absolute value) to those shown, with the same sign.

- radix of exponent representation, b

  FLT_RADIX 2

- number of base-**FLT_RADIX** digits in the floating-point mantissa, p

  FLT_MANT_DIG
  DBL_MANT_DIG
  LDBL_MANT_DIG

- number of decimal digits of precision, $\lfloor (p-1) \times \log_{10} b \rfloor + \begin{cases} 1 & \text{if } b \text{ is a power of 10} \\ 0 & \text{otherwise} \end{cases}$

  FLT_DIG 6
  DBL_DIG 10
  LDBL_DIG 10

- minimum negative integer such that **FLT_RADIX** raised to that power minus 1 is a normalized floating-point number, $e_{\min}$

  FLT_MIN_EXP
  DBL_MIN_EXP
  LDBL_MIN_EXP

- minimum negative integer such that 10 raised to that power is in the range of normalized floating-point numbers, $\lceil \log_{10} b^{e_{\min}-1} \rceil$

  FLT_MIN_10_EXP -37
  DBL_MIN_10_EXP -37
  LDBL_MIN_10_EXP -37

- maximum integer such that **FLT_RADIX** raised to that power minus 1 is a representable finite floating-point number, $e_{\max}$

  FLT_MAX_EXP
  DBL_MAX_EXP
  LDBL_MAX_EXP

- maximum integer such that 10 raised to that power is in the range of representable finite floating-point numbers, $\lfloor \log_{10}((1 - b^{-p}) \times b^{e_{\max}}) \rfloor$

FLT_MAX_10_EXP	+37
DBL_MAX_10_EXP	+37
LDBL_MAX_10_EXP	+37

The values given in the following list shall be replaced by implementation-defined expressions with values that shall be equal to or greater than those shown.

- maximum representable finite floating-point number, $(1 - b^{-p}) \times b^{e_{max}}$

FLT_MAX	1E+37
DBL_MAX	1E+37
LDBL_MAX	1E+37

The values given in the following list shall be replaced by implementation-defined expressions with values that shall be equal to or smaller than those shown.

- minimum positive floating-point number x such that $1.0 + x \neq 1.0$, b^{1-p}

FLT_EPSILON	1E-5
DBL_EPSILON	1E-9
LDBL_EPSILON	1E-9

- minimum normalized positive floating-point number, $b^{e_{min}-1}$

FLT_MIN	1E-37
DBL_MIN	1E-37
LDBL_MIN	1E-37

Examples

The following describes an artificial floating-point representation that meets the minimum requirements of the Standard, and the appropriate values in a **<float.h>** header for type **float**:

$$x = s \times 16^e \times \sum_{k=1}^{6} f_k \times 16^{-k}, \quad -31 \le e \le +32$$

FLT_RADIX	16
FLT_MANT_DIG	6
FLT_EPSILON	9.53674316E-07F
FLT_DIG	6
FLT_MIN_EXP	-31
FLT_MIN	2.93873588E-39F
FLT_MIN_10_EXP	-38
FLT_MAX_EXP	+32
FLT_MAX	3.40282347E+38F
FLT_MAX_10_EXP	+38

The following describes floating-point representations that also meet the requirements for single-precision and double-precision normalized numbers in the *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Std 754-1985),[9] and the appropriate values in a **<float.h>** header for types **float** and **double**:

9. The floating-point model in that standard sums powers of b from zero, so the values of the exponent limits are one less than shown here.

$$x_f = s \times 2^e \times \sum_{k=1}^{24} f_k \times 2^{-k}, \quad -125 \le e \le +128$$

$$x_d = s \times 2^e \times \sum_{k=1}^{53} f_k \times 2^{-k}, \quad -1021 \le e \le +1024$$

```
FLT_RADIX                          2
FLT_MANT_DIG                      24
FLT_EPSILON         1.19209290E-07F
FLT_DIG                            6
FLT_MIN_EXP                     -125
FLT_MIN             1.17549435E-38F
FLT_MIN_10_EXP                  -37
FLT_MAX_EXP                     +128
FLT_MAX             3.40282347E+38F
FLT_MAX_10_EXP                  +38
DBL_MANT_DIG                       53
DBL_EPSILON 2.2204460492503131E-16
DBL_DIG                           15
DBL_MIN_EXP                    -1021
DBL_MIN     2.2250738585072016E-308
DBL_MIN_10_EXP                  -307
DBL_MAX_EXP                    +1024
DBL_MAX     1.7976931348623157E+308
DBL_MAX_10_EXP                  +308
```

The values shown above for `FLT_EPSILON` and `DBL_EPSILON` are appropriate for the ANSI/IEEE Std 754-1985 default rounding mode (to nearest). Their values may differ for other rounding modes.

Forward references: conditional inclusion (§3.8.1).

function, even if that function is called within the signal handler.

No such guarantees can be extended to library functions, with the explicit exceptions of `longjmp` (§4.6.2.1) and `signal` (§4.7.1.1), since the library functions may be arbitrarily interrelated and since some of them have profound effect on the environment.

Calls to `longjmp` are problematic, despite the assurances of §4.6.2.1. The signal could have occurred during the execution of some library function which was in the process of updating external state and/or static variables.

A second signal for the same handler could occur before the first is processed, and the Standard makes no guarantees as to what happens to the second signal.

2.2.4 Environmental limits

The Committee agreed that the Standard must say something about certain capacities and limitations, but just how to enforce these treaty points was the topic of considerable debate.

2.2.4.1 Translation limits

The Standard requires that an implementation be able to translate and compile some program that meets each of the stated limits. This criterion was felt to give a useful latitude to the implementor in meeting these limits. While a deficient implementation could probably contrive a program that meets this requirement, yet still succeed in being useless, the Committee felt that such ingenuity would probably require more work than making something useful. The sense of the Committee is that implementors should not construe the translation limits as the values of hard-wired parameters, but rather as a set of criteria by which an implementation will be judged.

Some of the limits chosen represent interesting compromises. The goal was to allow reasonably large portable programs to be written, without placing excessive burdens on reasonably small implementations.

The minimum maximum limit of 257 cases in a switch statement allows coding of lexical routines which can branch on any character (one of at least 256 values) or on the value EOF.

2.2.4.2 Numerical limits

In addition to the discussion below, see §4.1.4.

Sizes of integral types `<limits.h>` Such a large body of C code has been developed for 8-bit byte machines that the integer sizes in such environments must be considered normative. The prescribed limits are minima: an implementation on a machine with 9-bit bytes can be conforming, as can an implementation that defines `int` to be the same width as `long`. The negative limits have been chosen to accommodate ones-complement or sign-magnitude implementations, as well as the

more usual twos-complement. The limits for the maxima and minima of unsigned types are specified as unsigned constants (e.g., 65535u) to avoid surprising widenings of expressions involving these extrema.

The macro `CHAR_BIT` makes available the number of bits in a `char` object. The Committee saw little utility in adding such macros for other data types.

Characteristics of floating types `<float.h>` The characterization of floating point follows, with minor changes, that of the FORTRAN standardization committee (X3J3).[1] The Committee chose to follow the FORTRAN model in some part out of a concern for FORTRAN-to-C translation, and in large part out of deference to the FORTRAN committee's greater experience with fine points of floating point usage.

Single precision (32-bit) floating point is considered adequate to support a conforming C implementation. Thus the minimum maxima constraining floating types are extremely permissive.

The Committee has also endeavored to accommodate the IEEE 754 floating point standard by not adopting any constraints on floating point which are contrary to this standard.

[1] See X3J3 working document S8-101.

by parentheses.

Implementation limits

The implementation shall allow the specification of types that have at least 12 pointer, array, and function declarators (in any valid combinations) modifying an arithmetic, a structure, a union, or an incomplete type, either directly or via one or more `typedefs`.

Forward references: type definitions (§3.5.6).

3.5.4.1 Pointer declarators

Semantics

If, in the declaration "`T D1`," `D1` has the form

`*` *type-qualifier-list*$_{opt}$ `D`

and the type specified for *ident* in the declaration "`T D`" is "*derived-declarator-type-list T*," then the type specified for *ident* is "*derived-declarator-type-list type-qualifier-list* pointer to *T*." For each type qualifier in the list, *ident* is a so-qualified pointer.

For two pointer types to be compatible, both shall be identically qualified and both shall be pointers to compatible types.

Examples

The following pair of declarations demonstrates the difference between a "variable pointer to a constant value" and a "constant pointer to a variable value."

```
const int *ptr_to_constant;
int *const constant_ptr;
```

The contents of the `const int` pointed to by `ptr_to_constant` shall not be modified, but `ptr_to_constant` itself may be changed to point to another `const int`. Similarly, the contents of the `int` pointed to by `constant_ptr` may be modified, but `constant_ptr` itself shall always point to the same location.

The declaration of the constant pointer `constant_ptr` may be clarified by including a definition for the type "pointer to `int`."

```
typedef int *int_ptr;
const int_ptr constant_ptr;
```

declares `constant_ptr` as an object that has type "const-qualified pointer to `int`."

3.5.4.2 Array declarators

Constraints

The expression that specifies the size of an array shall be an integral constant expression that has a value greater than zero.

Semantics

If, in the declaration "`T D1`," `D1` has the form

`D [`*constant-expression*$_{opt}$`]`

and the type specified for *ident* in the declaration "`T D`" is "*derived-declarator-type-list T*," then the type specified for *ident* is "*derived-declarator-type-list* array of *T*."[60] If the size is not

60. When several "array of" specifications are adjacent, a multi-dimensional array is declared.

In these declarations the `const` property is associated with the declarator `stype`, so `x` and `y` are both `const` objects.

The Committee considered making `const` and `volatile` storage classes, but this would have ruled out any number of desirable constructs, such as `const` members of structures and variable pointers to `const` types.

A cast of a value to a qualified type has no effect; the qualification (`volatile`, say) can have no effect on the access since it has occurred prior to the cast. If it is necessary to access a non-volatile object using volatile semantics, the technique is to cast the address of the object to the appropriate pointer-to-qualified type, then dereference that pointer.

3.5.4 Declarators

The function prototype syntax was adapted from C++. (See §3.3.2.2 and §3.5.4.3)

Some current implementations have a limit of six type modifiers (*function returning, array of, pointer to*), the limit used in Ritchie's original compiler. This limit has been raised to twelve since the original limit has proven insufficient in some cases; in particular, it did not allow for FORTRAN-to-C translation, since FORTRAN allows for seven subscripts. (Some users have reported using nine or ten levels, particularly in machine-generated C code.)

3.5.4.1 Pointer declarators

A pointer declarator may have its own type qualifiers, to specify the attributes of the pointer itself, as opposed to those of the reference type. The construct is adapted from C++.

`const int *` means *(variable) pointer to constant* `int`, and `int * const` means *constant pointer to (variable)* `int`, just as in C++, from which these constructs were adopted. (And *mutatis mutandis* for the other type qualifiers.) As with other aspects of C type declarators, judicious use of `typedef` statements can clarify the code.

3.5.4.2 Array declarators

The concept of *composite types* (§3.1.2.6) was introduced to provide for the accretion of information from incomplete declarations, such as array declarations with missing size, and function declarations with missing prototype (argument declarations). Type declarators are therefore said to specify *compatible types* if they agree except for the fact that one provides less information of this sort than the other.

The declaration of 0-length arrays is invalid, under the general principle of not providing for 0-length objects. The only common use of this construct has been in the declaration of dynamically allocated variable-size arrays, such as

```
struct segment {
```

```
    short int count;
    char c[N];
};

struct segment * new_segment( const int length );
{
    struct segment * result;
    result = malloc( sizeof segment + (length-N) );
    result->count = length;
    return result;
}
```

In such usage, N would be 0 and (length-N) would be written as length. But this paradigm works just as well, as written, if N is 1. ⌜(Note, by the by, an alternate way of specifying the size of result: Δ+

```
result = malloc( offsetof(struct segment,c) + length );
```

This illustrates one of the uses of the offsetof macro.)⌟

3.5.4.3 Function declarators (including prototypes)

The function prototype mechanism is one of the most useful additions to the C language. The feature, of course, has precedent in many of the Algol-derived languages of the past 25 years. The particular form adopted in the Standard is based in large part upon C++.

Function prototypes provide a powerful translation-time error detection capability. In traditional C practice without prototypes, it is extremely difficult for the translator to detect errors (wrong number or type of arguments) in calls to functions declared in another source file. Detection of such errors has either occurred at runtime, or through the use of auxiliary software tools.

In function calls not in scope of a function prototype, integral arguments have the ***integral widening conversions*** applied and float arguments are widened to double. It is thus impossible in such a call to pass an unconverted char or float argument. Function prototypes give the programmer explicit control over the function argument type conversions, so that the often inappropriate and sometimes inefficient default widening rules for arguments can be suppressed by the implementation. Modifications of function interfaces are easier in cases where the actual arguments are still assignment compatible with the new formal parameter type — only the function definition and its prototype need to be rewritten in this case; no function calls need be rewritten.

Allowing an optional identifier to appear in a function prototype serves two purposes:

- the programmer can associate a meaningful name with each argument position for documentation purposes, and

- a function declarator and a function prototype can use the same syntax. The consistent syntax makes it easier for new users of C to learn the language. Automatic generation of function prototype declarators from function definitions is also facilitated.

Optimizers can also take advantage of function prototype information. Consider this example:

```
extern int compare(const char * string1,
                   const char * string2) ;

void func2(int x)
{
        char * str1, * str2 ;
            /* ... */
        x = compare(str1, str2) ;
            /* ... */
}
```

The optimizer knows that the pointers passed to `compare` are not used to assign new values to any objects that the pointers reference. Hence the optimizer can make less conservative assumptions about the side effects of `compare` than would otherwise be necessary.

The Standard requires that calls to functions taking a variable number of arguments must occur in the presence of a prototype (using the trailing ellipsis notation , . . .). An implementation may thus assume that all other functions are called with a fixed argument list, and may therefore use possibly more efficient calling sequences.

3.5.5 Type names

Empty parentheses within a type name are always taken as meaning *function with unspecified arguments* and never as (unnecessary) parentheses around the elided identifier. This specification avoids an ambiguity by fiat.

3.5.6 Type definitions

A `typedef` may only be redeclared in an inner block with a declaration that explicitly contains a type name. This rule avoids the ambiguity about whether to take the `typedef` as the type name or the candidate for redeclaration.

Some implementations of C have allowed type specifiers to be added to a type defined using `typedef`. Thus

```
typedef short int small ;
unsigned small x ;
```

would give `x` the type `unsigned short int`. The Committee decided that since this interpretation may be difficult to provide in many implementations, and since

Forward references: diagnostics (§4.2).

4.1.3 Errors <errno.h>

The header **<errno.h>** defines several macros, all relating to the reporting of error conditions.

The macros are

EDOM
ERANGE

which expand to distinct nonzero integral constant expressions; and

errno

which expands to a modifiable lvalue[83] that has type **int**, the value of which is set to a positive error number by several library functions. It is unspecified whether **errno** is a macro or an identifier declared with external linkage. If a macro definition is suppressed in order to access an actual object, or a program defines an external identifier with the name **errno**, the behavior is undefined.

The value of **errno** is zero at program startup, but is never set to zero by any library function.[84] The value of **errno** may be set to nonzero by a library function call whether or not there is an error, provided the use of **errno** is not documented in the description of the function in the Standard.

Additional macro definitions, beginning with **E** and a digit or **E** and an upper-case letter,[85] may also be specified by the implementation.

4.1.4 Limits <float.h> and <limits.h>

The headers **<float.h>** and **<limits.h>** define several macros that expand to various limits and parameters.

The macros, their meanings, and their minimum magnitudes are listed in §2.2.4.2.

4.1.5 Common definitions <stddef.h>

The following types and macros are defined in the standard header **<stddef.h>**. Some are also defined in other headers, as noted in their respective sections.

The types are

ptrdiff_t

which is the signed integral type of the result of subtracting two pointers;

size_t

which is the unsigned integral type of the result of the **sizeof** operator; and

wchar_t

which is an integral type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locales; the null character shall have the code value zero and each member of the basic character set defined in §2.2.1 shall have

83. The macro **errno** need not be the identifier of an object. It might be a modifiable lvalue resulting from a function call (for example, ***errno**()).

84. Thus, a program that uses **errno** for error checking should set it to zero before a library function call, then inspect it before a subsequent library function call.

85. See "future library directions" (§4.13.1).

a code value equal to its value when used as the lone character in an integer character constant.

The macros are

```
NULL
```

which expands to an implementation-defined null pointer constant; and

```
offsetof(type, member-designator)
```

which expands to an integral constant expression that has type `size_t`, the value of which is the offset in bytes, to the structure member (designated by *member-designator*), from the beginning of its structure (designated by *type*). The *member-designator* shall be such that given

```
static type t;
```

then the expression `&(t.`*member-designator*`)` evaluates to an address constant. (If the specified member is a bit-field, the behavior is undefined.)

Forward references: localization (§4.4).

4.1.6 Use of library functions

Each of the following statements applies unless explicitly stated otherwise in the detailed descriptions that follow. If an argument to a function has an invalid value (such as a value outside the domain of the function, or a pointer outside the address space of the program, or a null pointer), the behavior is undefined. Any function declared in a header may be implemented as a macro defined in the header, so a library function should not be declared explicitly if its header is included. Any macro definition of a function can be suppressed locally by enclosing the name of the function in parentheses, because the name is then not followed by the left parenthesis that indicates expansion of a macro function name. For the same syntactic reason, it is permitted to take the address of a library function even if it is also defined as a macro.[86] The use of `#undef` to remove any macro definition will also ensure that an actual function is referred to. Any invocation of a library function that is implemented as a macro will expand to code that evaluates each of its arguments exactly once, fully protected by parentheses where necessary, so it is generally safe to use arbitrary expressions as arguments. Likewise, those function-like macros described in the following sections may be invoked in an expression anywhere a function with a compatible return type could be called.[87]

Provided that a library function can be declared without reference to any type defined in a header, it is also permissible to declare the function, either explicitly or implicitly, and use it without including its associated header. If a function that accepts a variable number of arguments is not declared (explicitly or by including its associated header), the behavior is undefined.

---

86. This means that an implementation must provide an actual function for each library function, even if it also provides a macro for that function.

87. Because external identifiers and some macro names beginning with an underscore are reserved, implementations may provide special semantics for such names. For example, the identifier `__BUILTIN_abs` could be used to indicate generation of in-line code for the `abs` function. Thus, the appropriate header could specify

```
#define abs(x) __BUILTIN_abs(x)
```

for a compiler whose code generator will accept it.

In this manner, a user desiring to guarantee that a given library function such as `abs` will be a genuine function may write

```
#undef abs
```

whether the implementation's header provides a macro implementation of `abs` or a builtin implementation. The prototype for the function, which precedes and is hidden by any macro definition, is thereby revealed also.

Examples

The function `atoi` may be used in any of several ways:

- by use of its associated header (possibly generating a macro expansion)

```
#include <stdlib.h>
const char *str;
/*...*/
i = atoi(str);
```

- by use of its associated header (assuredly generating a true function reference)

```
#include <stdlib.h>
#undef atoi
const char *str;
/*...*/
i = atoi(str);
```

or

```
#include <stdlib.h>
const char *str;
/*...*/
i = (atoi)(str);
```

- by explicit declaration

```
extern int atoi(const char *);
const char *str;
/*...*/
i = atoi(str);
```

- by implicit declaration

```
const char *str;
/*...*/
i = atoi(str);
```

The header is usable in an implementation of the Standard in the absence of a definition of __EXTENSIONS__, and the non-Standard implementation can provide the appropriate definitions to enable the extra declarations.

4.1.3 Errors
<errno.h>

<errno.h> is a header invented to encapsulate the error handling mechanism used by many of the library routines.[1]

The error reporting machinery centered about the setting of errno is generally regarded with tolerance at best. It requires a "pathological coupling" between library functions and makes use of a static writable memory cell, which interferes with the construction of shareable libraries. Nevertheless, the Committee preferred to standardize this existing, however deficient, machinery rather than invent something more ambitious.

The definition of errno as an lvalue macro grants implementors the license to expand it to something like *__errno_addr(), where the function returns a pointer to the (current) modifiable copy of errno.

4.1.4 Limits
<float.h> and <limits.h>

Both <float.h> and <limits.h> are inventions. Included in these headers are various parameters of the execution environment which are potentially useful at compile time, and which are difficult or impossible to determine by other means.

The availability of this information in headers provides a portable way of tuning a program to different environments. Another possible method of determining some of this information is to evaluate arithmetic expressions in the preprocessing statements. Requiring that preprocessing always yield the same results as run-time arithmetic, however, would cause problems for portable compilers (themselves written in C) or for cross compilers, which would then be required to implement the (possibly wildly different) arithmetic of the target machine on the host machine. (See §3.4.)

<float.h> makes available to programmers a set of useful quantities for numerical analysis. (See §2.2.4.2.) This set of quantities has seen widespread use for such analysis, in C and in other languages, and was recommended by the numerical analysts on the Committee. The set was chosen so as not to prejudice an implementation's selection of floating-point representation.

Most of the limits in <float.h> are specified to be general double expressions rather than restricted constant expressions

[1] In earlier drafts of the Standard, errno and related macros were defined in <stddef.h>. When the Committee decided that the other definitions in this header were of such general utility that they should be required even in freestanding environments, it created <errno.h>.

- to allow use of values which cannot readily (or, in some cases, cannot possibly) be constructed as manifest constants, and
- to allow for run-time selection of floating-point properties, as is possible, for instance, in IEEE-854 implementations.

4.1.5 Common definitions <stddef.h>

<stddef.h> is a header invented to provide definitions of several types and macros used widely in conjunction with the library: **ptrdiff_t** (see §3.3.6), **size_t** (see §3.3.3.4), **wchar_t** (see §3.1.3.4), and **NULL**. Including any header that references one of these macros will also define it, an exception to the usual library rule that each macro or function belongs to exactly one header.

NULL can be defined as any *null pointer constant*. Thus existing code can retain definitions of **NULL** as **0** or **0L**, but an implementation may choose to define it as **(void *)0**; this latter form of definition is convenient on architectures where the pointer size(s) do(es) not equal the size of any integer type. It has never been wise to use **NULL** in place of an arbitrary pointer as a function argument, however, since pointers to different types need not be the same size. The library avoids this problem by providing special macros for the arguments to **signal**, the one library function that might see a null function pointer.

The **offsetof** macro has been added to provide a portable means of determining
Δ+ the offset, in bytes, of a member within its structure. ⌈This capability is useful in programs, such as are typical in data-base implementations, which declare a large number of different data structures: it is desirable to provide "generic" routines that work from descriptions of the structures, rather than from the structure declarations themselves.[2]⌋

In many implementations, **offsetof** could be defined as one of

```
(size_t)&(((s_name*)0)->m_name)
```

or

```
(size_t)(char *)&(((s_name*)0)->m_name)
```

or, where **X** is some predeclared address (or 0) and **A(Z)** is defined as **((char*)&Z)**,

```
(size_t)( A( (s_name*)X->m_name ) - A( X ))
```

[2]Consider, for instance, a set of nodes (structures) which are to be dynamically allocated and garbage-collected, and which can contain pointers to other such nodes. A possible implementation is to have the first field in each node point to a descriptor for that node. The descriptor includes a table of the offsets of fields which are pointers to other nodes. A garbage-collector "mark" routine needs no further information about the content of the node (except, of course, where to put the mark). New node types can be added to the program without requiring the mark routine to be rewritten or even recompiled.

It was not feasible, however, to mandate any single one of these forms as a construct guaranteed to be portable.

Other implementations may choose to expand this macro as a call to a built-in function that interrogates the translator's symbol table.

4.1.6 Use of library functions

To make usage more uniform for both implementor and programmer, the Standard requires that every library function (unless specifically noted otherwise) must be represented as an actual function, in case a program wishes to pass its address as a parameter to another function. On the other hand, every library function is now a candidate for redefinition, in its associated header, as a macro, provided that the macro performs a "safe" evaluation of its arguments, i.e., it evaluates each of the arguments exactly once and parenthesizes them thoroughly, and provided that its top-level operator is such that the execution of the macro is not interleaved with other expressions. ⌈⌋ Two exceptions are the macros `getc` and `putc`, which may evaluate their arguments in an unsafe manner. (See §4.9.7.5.) Δ

If a program requires that a library facility be implemented as an actual function, not as a macro, then the macro name, if any, may be erased by using the `#undef` preprocessing directive (see §3.8.3).

All library prototypes are specified in terms of the "widened" types: an argument formerly declared as `char` is now written as `int`. This ensures that most library functions can be called with or without a prototype in scope (see §3.3.2.2), thus maintaining backwards compatibility with existing, pre-Standard, code. Note, however, that since functions like `printf` and `scanf` use variable-length argument lists, they must be called in the scope of a prototype.

The Standard contains an example showing how certain library functions may be "built in" in an implementation that remains *conforming*.

4.2 Diagnostics <assert.h>

4.2.1 Program diagnostics

4.2.1.1 The assert macro

Some implementations tolerate an arbitrary scalar expression as the argument to `assert`, but the Committee decided to require correct operation only for `int` expressions. For the sake of implementors, no hard and fast format for the output of a failing assertion is required; but the Standard mandates enough machinery to replicate the form shown in the footnote.

It can be difficult or impossible to make `assert` a true function, so it is restricted to macro form only.

4.5 MATHEMATICS <math.h>

The header <math.h> declares several mathematical functions and defines one macro. The functions take double-precision arguments and return double-precision values.[93] Integer arithmetic functions and conversion functions are discussed later.

The macro defined is

```
HUGE_VAL
```

which expands to a positive `double` expression, not necessarily representable as a `float`.

Forward references: integer arithmetic functions (§4.10.6), the `atof` function (§4.10.1.1), the `strtod` function (§4.10.1.4).

4.5.1 Treatment of error conditions

The behavior of each of these functions is defined for all representable values of its input arguments. Each function shall execute as if it were a single operation, without generating any externally visible exceptions.

For all functions, a *domain error* occurs if an input argument is outside the domain over which the mathematical function is defined. The description of each function lists any required + domain errors; an implementation may define additional domain errors, provided that such errors + are consistent with the mathematical definition of the function.[94] On a domain error, the function returns an implementation-defined value; the value of the macro `EDOM` is stored in `errno`.

Similarly, a *range error* occurs if the result of the function cannot be represented as a `double` value. If the result overflows (the magnitude of the result is so large that it cannot be represented in an object of the specified type), the function returns the value of the macro `HUGE_VAL`, with the same sign as the correct value of the function; the value of the macro `ERANGE` is stored in `errno`. If the result underflows (the magnitude of the result is so small that it cannot be represented in an object of the specified type), the function returns zero; whether the integer expression `errno` acquires the value of the macro `ERANGE` is implementation-defined.

4.5.2 Trigonometric functions

4.5.2.1 The `acos` function

Synopsis

```
#include <math.h>
double acos(double x);
```

Description

The `acos` function computes the principal value of the arc cosine of `x`. A domain error occurs for arguments not in the range $[-1, +1]$.

Returns

The `acos` function returns the arc cosine in the range $[0, \pi]$ radians.

93. See "future library directions" (§4.13.4).

94. In an implementation that supports infinities, this allows infinity as an argument to be a domain error if the mathematical domain of the function does not include infinity.

4.5.2.2 The asin function

Synopsis

```
#include <math.h>
double asin(double x);
```

Description

The **asin** function computes the principal value of the arc sine of **x**. A domain error occurs for arguments not in the range [−1, +1].

Returns

The **asin** function returns the arc sine in the range [−π/2, +π/2] radians.

4.5.2.3 The atan function

Synopsis

```
#include <math.h>
double atan(double x);
```

Description

The **atan** function computes the principal value of the arc tangent of **x**.

Returns

The **atan** function returns the arc tangent in the range [−π/2, +π/2] radians.

4.5.2.4 The atan2 function

Synopsis

```
#include <math.h>
double atan2(double y, double x);
```

Description

The **atan2** function computes the principal value of the arc tangent of **y/x**, using the signs of both arguments to determine the quadrant of the return value. A domain error occurs if both arguments are zero and **y/x** cannot be represented.

Returns

The **atan2** function returns the arc tangent of **y/x**, in the range [−π, +π] radians.

4.5.2.5 The cos function

Synopsis

```
#include <math.h>
double cos(double x);
```

Description

The **cos** function computes the cosine of **x** (measured radians). A large magnitude argument may yield a result with little or no significance.

Returns

The **cos** function returns the cosine value.

4.5.2.6 The sin function

Synopsis

```
#include <math.h>
double sin(double x);
```

Description

The **sin** function computes the sine of **x** (measured in radians). A large magnitude argument may yield a result with little or no significance.

Returns

The **sin** function returns the sine value.

4.5.2.7 The tan function

Synopsis

```
#include <math.h>
double tan(double x);
```

Description

The **tan** function returns the tangent of **x** (measured in radians). A large magnitude argument may yield a result with little or no significance.

Returns

The **tan** function returns the tangent value.

4.5.3 Hyperbolic functions

4.5.3.1 The cosh function

Synopsis

```
#include <math.h>
double cosh(double x);
```

Description

The **cosh** function computes the hyperbolic cosine of **x**. A range error occurs if the magnitude of **x** is too large.

Returns

The **cosh** function returns the hyperbolic cosine value.

4.5.3.2 The sinh function

Synopsis

```
#include <math.h>
double sinh(double x);
```

Description

The **sinh** function computes the hyperbolic sine of **x**. A range error occurs if the magnitude of **x** is too large.

Returns

The **sinh** function returns the hyperbolic sine value.

4.5.3.3 The tanh function

Synopsis

```
#include <math.h>
double tanh(double x);
```

Description

The **tanh** function computes the hyperbolic tangent of **x**.

Returns

The **tanh** function returns the hyperbolic tangent value.

4.5.4 Exponential and logarithmic functions

4.5.4.1 The exp function

Synopsis

```
#include <math.h>
double exp(double x);
```

Description

The **exp** function computes the exponential function of **x**. A range error occurs if the magnitude of **x** is too large.

Returns

The **exp** function returns the exponential value.

4.5.4.2 The frexp function

Synopsis

```
#include <math.h>
double frexp(double value, int *exp);
```

Description

The **frexp** function breaks a floating-point number into a normalized fraction and an integral power of 2. It stores the integer in the **int** object pointed to by **exp**.

Returns

The **frexp** function returns the value **x**, such that **x** is a **double** with magnitude in the interval [1/2, 1) or zero, and **value** equals **x** times 2 raised to the power ***exp**. If **value** is zero, both parts of the result are zero.

4.5.4.3 The ldexp function

Synopsis

```
#include <math.h>
double ldexp(double x, int exp);
```

Description

The **ldexp** function multiplies a floating-point number by an integral power of 2. A range error may occur.

Returns

The **ldexp** function returns the value of **x** times 2 raised to the power **exp**.

4.5.4.4 The log function

Synopsis

```
#include <math.h>
double log(double x);
```

Description

The **log** function computes the natural logarithm of **x**. A domain error occurs if the argument is negative. A range error occurs if the argument is zero and the logarithm of zero cannot be represented.

Returns

The **log** function returns the natural logarithm.

4.5.4.5 The log10 function

Synopsis

```
#include <math.h>
double log10(double x);
```

Description

The **log10** function computes the base-ten logarithm of **x**. A domain error occurs if the argument is negative. A range error occurs if the argument is zero and the logarithm of zero cannot be represented.

Returns

The **log10** function returns the base-ten logarithm.

4.5.4.6 The modf function

Synopsis

```
#include <math.h>
double modf(double value, double *iptr);
```

Description

The **modf** function breaks the argument **value** into integral and fractional parts, each of which has the same sign as the argument. It stores the integral part as a **double** in the object pointed to by **iptr**.

Returns

The **modf** function returns the signed fractional part of **value**.

4.5.5 Power functions

4.5.5.1 The pow function

Synopsis

```
#include <math.h>
double pow(double x, double y);
```

Description

The **pow** function computes **x** raised to the power **y**. A domain error occurs if **x** is negative and **y** is not an integer. A domain error occurs if the result cannot be represented when **x** is zero and **y** is less than or equal to zero. A range error may occur.

Returns

The **pow** function returns the value of **x** raised to the power **y**.

4.5.5.2 The sqrt function

Synopsis

```
#include <math.h>
double sqrt(double x);
```

Description

The **sqrt** function computes the nonnegative square root of **x**. A domain error occurs if the argument is negative.

Returns

The **sqrt** function returns the value of the square root.

4.5.6 Nearest integer, absolute value, and remainder functions

4.5.6.1 The ceil function

Synopsis

```
#include <math.h>
double ceil(double x);
```

Description

The **ceil** function computes the smallest integral value not less than **x**.

Returns

The **ceil** function returns the smallest integral value not less than **x**, expressed as a double.

4.5.6.2 The fabs function

Synopsis

```
#include <math.h>
double fabs(double x);
```

Description

The **fabs** function computes the absolute value of a floating-point number **x**.

Returns

The **fabs** function returns the absolute value of **x**.

4.5.6.3 The floor function

Synopsis

```
#include <math.h>
double floor(double x);
```

Description

The **floor** function computes the largest integral value not greater than **x**.

Returns

The **floor** function returns the largest integral value not greater than **x**, expressed as a double.

4.5.6.4 The fmod function

Synopsis

```
#include <math.h>
double fmod(double x, double y);
```

Description

The **fmod** function computes the floating-point remainder of **x/y**.

Returns

The **fmod** function returns the value **x** – *i* * **y**, for some integer *i* such that, if **y** is nonzero, the result has the same sign as **x** and magnitude less than the magnitude of **y**. If **y** is zero, whether a domain error occurs or the **fmod** function returns zero is implementation-defined.

4.4.1 Locale control

4.4.1.1 The `setlocale` function

`setlocale` provides the mechanism for controlling *locale-specific* features of the library. The `category` argument allows parts of the library to be localized as necessary without changing the entire locale-specific environment. Specifying the `locale` argument as a string gives an implementation maximum flexibility in providing a set of locales. For instance, an implementation could map the argument string into the name of a file containing appropriate localization parameters — these files could then be added and modified without requiring any recompilation of a localizable program.

4.4.2 Numeric formatting convention inquiry

4.4.2.1 The `localeconv` function

The `localeconv` function gives a programmer access to information about how to format numeric quantities (monetary or otherwise). This sort of interface was considered preferable to defining conversion functions directly: even with a specified locale, the set of distinct formats that can be constructed from these elements is large, and the ones desired very application-dependent.

4.5 Mathematics <math.h>

For historical reasons, the math library is only defined for the floating type `double`. All the names formed by appending `f` or `l` to a name in `<math.h>` are reserved to allow for the definition of `float` and `long double` libraries.

The functions `ecvt`, `fcvt`, and `gcvt` have been dropped since their capability is available through `sprintf`.

Traditionally, `HUGE_VAL` has been defined as a manifest constant that approximates the largest representable `double` value. As an approximation to *infinity* it is problematic. As a function return value indicating overflow, it can cause trouble if first assigned to a `float` before testing, since a `float` may not necessarily hold all values representable in a `double`.

After considering several alternatives, the Committee decided to generalize `HUGE_VAL` to a positive double expression, so that it could be expressed as an external identifier naming a location initialized precisely with hexadecimal bit patterns. It can even be a special encoding for *machine infinity*, on implementations that support such codes. It need not be representable as a `float`, however.

Similarly, domain errors in the past were typically indicated by a zero return, which is not necessarily distinguishable from a valid result. The Committee agreed to make the return value for domain errors *implementation-defined*, so that special machine codes can be used to advantage. This makes possible an implementation

of the math library in accordance with the IEEE P854 proposal on floating point representation and arithmetic.

4.5.1 Treatment of error conditions

Whether underflow should be considered a range error, and cause errno to be set, is specified as *implementation-defined* since detection of underflow is inefficient on some systems.

⌈The Standard has been crafted to neither require nor preclude any popular implementation of floating point. This principle affects the definition of *domain error*: an implementation may define extra domain errors to deal with floating-point arguments such as infinity or "not-a-number".⌋ Δ+

The Committee considered the adoption of the matherr capability from UNIX System V. In this feature of that system's math library, any error (such as overflow or underflow) results in a call from the library function to a user-defined exception handler named matherr. The Committee rejected this approach for several reasons:

- This style is incompatible with popular floating point implementations, such as IEEE 754 (with its special return codes), or that of VAX/VMS.
- It conflicts with the error-handling style of FORTRAN, thus making it more difficult to translate useful bodies of mathematical code from that language to C.
- It requires the math library to be reentrant (since math routines could be called from matherr), which may complicate some implementations.
- It introduces a new style of library interface: a user-defined library function with a library-defined name. Note, by way of comparison, the signal and exit handling mechanisms, which provide a way of "registering" user-defined functions.

4.5.2 Trigonometric functions

4.5.2.1 The acos function

4.5.2.2 The asin function

4.5.2.3 The atan function

4.5.2.4 The atan2 function

⌈The atan2 function is modelled after FORTRAN's. It is described in terms of arctan $\frac{y}{z}$ for simplicity; the Committee did not wish to complicate the descriptions by specifying in detail how the determine the appropriate quadrant, since that should be obvious from normal mathematical convention. atan2(y,x) is well-defined and Δ

finite, even when x is 0; the one ambiguity occurs when both arguments are 0, because at that point any value in the range of the function could logically be selected. Since valid reasons can be advanced for all the different choices that have been in this situation by various implements, the Standard preserves the implementor's freedom to return an arbitrary well-defined value such a 0, to report a domain error, or to return an IEEE *NaN* code.]

4.5.2.5 The cos function

4.5.2.6 The sin function

4.5.2.7 The tan function

4.5.3 Hyperbolic functions

4.5.3.1 The cosh function

4.5.3.2 The sinh function

4.5.3.3 The tanh function

4.5.4 Exponential and logarithmic functions

4.5.4.1 The exp function

4.5.4.2 The frexp function

The functions `frexp`, `ldexp`, and `modf` are primitives used by the remainder of the library. There was some sentiment for dropping them for the same reasons that `ecvt`, `fcvt`, and `gcvt` were dropped, but their adherents rescued them for general use.

4.5.4.3 The ldexp function

See §4.5.4.2.

4.5.4.4 The log function

Whether `log(0.)` is a domain error or a range error is arguable. The choice in the Standard, *range error*, is for compatibility with IEEE P854. Some such implementations would represent the result as $-\infty$, in which case no error is raised.

4.5.4.5 The log10 function

See §4.5.4.4.

4.5.4.6 The modf function

See §4.5.4.2.

4.5.5 Power functions

4.5.5.1 The pow function

4.5.5.2 The sqrt function

IEEE P854, unlike the Standard, requires `sqrt(-0.)` to return a negatively signed magnitude-zero result. This is an issue on implementations that support a negative floating zero. The Standard specifies that taking the square root of a negative number (in the mathematical sense: less than 0) is a domain error which requires the function to return an *implementation-defined* value. This rule permits implementations to support either the IEEE P854 or vendor-specific floating point representations.

4.5.6 Nearest integer, absolute value, and remainder functions

4.5.6.1 The ceil function

4.5.6.2 The fabs function

Adding an absolute value operator was rejected by the Committee. An implementation can provide a built-in function for efficiency.

4.5.6.3 The floor function

4.5.6.4 The fmod function

`fmod` is defined even if the quotient `x/y` is not representable — the implementation of this function is properly by scaled subtraction rather than division.

The result of `fmod(x,0.0)` is either a domain error or 0.0; the result always lies between 0.0 and `y`, so specifying the non-erroneous result as 0.0 simply recognizes the limit case.

The Committee considered and rejected a proposal to use the remainder operator `%` for this function; the operators in general correspond to hardware facilities, and `fmod` is not supported in hardware on most machines.

4.6 Non-local jumps <setjmp.h>

`jmp_buf` must be an array type for compatibility with existing practice: programs typically omit the address operator before a `jmp_buf` argument, even though a pointer to the argument is desired, not the value of the argument itself. Thus, a scalar or struct type is unsuitable. Note that a one-element array of the appropriate type is a valid definition.

`setjmp` is constrained to be a macro only: in some implementations the information necessary to restore context is only available while executing the function making the call to `setjmp`.

4.7 SIGNAL HANDLING <signal.h>

The header **<signal.h>** declares a type and two functions and defines several macros, for handling various *signals* (conditions that may be reported during program execution).

The type defined is

sig_atomic_t

which is the integral type of an object that can be accessed as an atomic entity, even in the presence of asynchronous interrupts.

The macros defined are

SIG_DFL
SIG_ERR
SIG_IGN

which expand to distinct constant expressions that have type compatible with the second argument to and the return value of the **signal** function, and whose value compares unequal to the address of any declarable function; and the following, each of which expands to a positive integral constant expression that is the signal number corresponding to the specified condition:

SIGABRT abnormal termination, such as is initiated by the **abort** function

SIGFPE an erroneous arithmetic operation, such as zero divide or an operation resulting in overflow

SIGILL detection of an invalid function image, such as an illegal instruction

SIGINT receipt of an interactive attention signal

SIGSEGV an invalid access to storage

SIGTERM a termination request sent to the program

An implementation need not generate any of these signals, except as a result of explicit calls to the **raise** function. Additional signals and pointers to undeclarable functions, with macro definitions beginning, respectively, with the letters **SIG** and an upper-case letter or with **SIG_** and an upper-case letter,[97] may also be specified by the implementation. The complete set of signals, their semantics, and their default handling is implementation-defined; all signal values shall be positive.

4.7.1 Specify signal handling

4.7.1.1 The signal function

Synopsis

```
#include <signal.h>
void (*signal(int sig, void (*func)(int)))(int);
```

Description

The **signal** function chooses one of three ways in which receipt of the signal number **sig** is to be subsequently handled. If the value of **func** is **SIG_DFL**, default handling for that signal will occur. If the value of **func** is **SIG_IGN**, the signal will be ignored. Otherwise, **func** shall point to a function to be called when that signal occurs. Such a function is called a

97. See "future library directions" (§4.13.5). The names of the signal numbers reflect the following terms (respectively): abort, floating-point exception, illegal instruction, interrupt, segmentation violation, and termination.

signal handler.

When a signal occurs, if **func** points to a function, first the equivalent of **signal(sig, SIG_DFL);** is executed or an implementation-defined blocking of the signal is performed. (If the value of **sig** is **SIGILL**, whether the reset to **SIG_DFL** occurs is implementation-defined.) Next the equivalent of **(*func)(sig);** is executed. The function **func** may terminate by executing a **return** statement or by calling the **abort**, **exit**, or **longjmp** function. If **func** executes a **return** statement and the value of **sig** was **SIGFPE** or any other implementation-defined value corresponding to a computational exception, the behavior is undefined. Otherwise, the program will resume execution at the point it was interrupted.

If the signal occurs other than as the result of calling the **abort** or **raise** function, the behavior is undefined if the signal handler calls any function in the standard library other than the **signal** function itself or refers to any object with static storage duration other than by assigning a value to a static storage duration variable of type **volatile sig_atomic_t**. Furthermore, if such a call to the **signal** function results in a **SIG_ERR** return, the value of **errno** is indeterminate.

At program startup, the equivalent of

```
signal(sig, SIG_IGN);
```

may be executed for some signals selected in an implementation-defined manner; the equivalent of

```
signal(sig, SIG_DFL);
```

is executed for all other signals defined by the implementation.

The implementation shall behave as if no library function calls the **signal** function.

Returns

If the request can be honored, the **signal** function returns the value of **func** for the most recent call to **signal** for the specified signal **sig**. Otherwise, a value of **SIG_ERR** is returned and a positive value is stored in **errno**.

Forward references: the **abort** function (§4.10.4.1).

4.7.2 Send signal

4.7.2.1 The raise function

Synopsis

```
#include <signal.h>
int raise(int sig);
```

Description

The **raise** function sends the signal **sig** to the executing program.

Returns

The **raise** function returns zero if successful, nonzero if unsuccessful.

longjmp to only one level of signal handling.

The longjmp function should not be called in an exit handler (i.e., a function registered with the atexit function (see §4.10.4.2)), since it might jump to some code which is no longer in scope.

4.7 Signal Handling <signal.h>

This facility has been retained from the Base Document since the Committee felt it important to provide some standard mechanism for dealing with exceptional program conditions. Thus a subset of the signals defined in UNIX were retained in the Standard, along with the basic mechanisms of declaring signal handlers and (with adaptations, see §4.7.2.1) raising signals. For a discussion of the problems created by including signals, see §2.2.3.

The signal machinery contains many misnomers: SIGFPE, SIGILL, and SIGSEGV have their roots in PDP-11 hardware terminology, but the names are too entrenched to change. A conforming implementation is not required to field *any* hardware interrupts.

The Committee has reserved the space of names beginning with SIG to permit implementations to add local names to <signal.h>. This implies that such names should not be otherwise used in a C source file which includes <signal.h>.

4.7.1 Specify signal handling

4.7.1.1 The signal function

When a signal occurs the normal flow of control of a program is interrupted. If a signal occurs that is being trapped by a signal handler, that handler is invoked. When it is finished, execution continues at the point at which the signal occurred. This arrangement could cause problems if the signal handler invokes a library function that was being executed at the time of the signal. Since library functions are not guaranteed to be re-entrant, they should not be called from a signal handler that returns. (See §2.2.3.) A specific exception to this rule has been granted for calls to signal from within the signal handler; otherwise, the handler could not reliably reset the signal.

The specification that some signals may be effectively set to SIG_IGN instead of SIG_DFL at program startup allows programs under UNIX systems to inherit this effective setting from parent processes.

For performance reasons, UNIX does not reset SIGILL to default handling when the handler is called (usually to emulate missing instructions). This treatment is sanctioned by specifying that whether reset occurs for SIGILL is *implementation-defined*.

Returns

The **setbuf** function returns no value.

Forward references: the **setvbuf** function (§4.9.5.6).

4.9.5.6 The setvbuf function

Synopsis

```
#include <stdio.h>
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

Description

The **setvbuf** function may be used after the stream pointed to by **stream** has been associated with an open file but before any other operation is performed on the stream. The argument **mode** determines how **stream** will be buffered, as follows: **_IOFBF** causes input/output to be fully buffered; **_IOLBF** causes output to be line buffered; **_IONBF** causes input/output to be unbuffered. If **buf** is not a null pointer, the array it points to may be used instead of a buffer allocated by the **setvbuf** function.[105] The argument **size** specifies the size of the array. The contents of the array at any time are indeterminate.

Returns

The **setvbuf** function returns zero on success, or nonzero if an invalid value is given for **mode** or if the request cannot be honored.

4.9.6 Formatted input/output functions

4.9.6.1 The fprintf function

Synopsis

```
#include <stdio.h>
int fprintf(FILE *stream, const char *format, ...);
```

Description

The **fprintf** function writes output to the stream pointed to by **stream**, under control of the string pointed to by **format** that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored. The **fprintf** function returns when the end of the format string is encountered.

The format shall be a multibyte character sequence, beginning and ending in its initial shift state. The format is composed of zero or more directives: ordinary multibyte characters (not %), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the character %. After the %, the following appear in sequence:

- Zero or more *flags* that modify the meaning of the conversion specification.
- An optional decimal integer specifying a minimum *field width*.[106] If the converted value has fewer characters than the field width, it will be padded with spaces on the left (or right, if the left adjustment flag, described later, has been given) to the field width.

105. The buffer must have a lifetime at least as great as the open stream, so the stream should be closed before a buffer that has automatic storage duration is deallocated upon block exit.

106. Note that 0 is taken as a flag, not as the beginning of a field width.

- An optional *precision* that gives the minimum number of digits to appear for the d, i, o, u, x, and X conversions, the number of digits to appear after the decimal-point character for e, E, and f conversions, the maximum number of significant digits for the g and G conversions, or the maximum number of characters to be written from a string in s conversion. The precision takes the form of a period (.) followed by an optional decimal integer; if the integer is omitted, it is treated as zero.

- An optional h specifying that a following d, i, o, u, x, or X conversion specifier applies to a **short int** or **unsigned short int** argument (the argument will have been promoted according to the integral promotions, and its value shall be converted to **short int** or **unsigned short int** before printing); an optional h specifying that a following n conversion specifier applies to a pointer to a **short int** argument; an optional l (ell) specifying that a following d, i, o, u, x, or X conversion specifier applies to a **long int** or **unsigned long int** argument; an optional l specifying that a following n conversion specifier applies to a pointer to a **long int** argument; or an optional L specifying that a following e, E, f, g, or G conversion specifier applies to a **long double** argument. If an h, l, or L appears with any other conversion specifier, the behavior is undefined.

- A character that specifies the type of conversion to be applied.

A field width or precision, or both, may be indicated by an asterisk * instead of a digit string. In this case, an **int** argument supplies the field width or precision. The arguments specifying field width or precision, or both, shall appear (in that order) before the argument (if any) to be converted. A negative field width argument is taken as a – flag followed by a positive field width. A negative precision argument is taken as if it were missing.

The flag characters and their meanings are

– The result of the conversion will be left-justified within the field.

\+ The result of a signed conversion will always begin with a plus or minus sign.

space If the first character of a signed conversion is not a sign, or if a signed conversion results in no characters, a space will be prepended to the result. If the *space* and + flags both appear, the *space* flag will be ignored.

\# The result is to be converted to an "alternate form." For o conversion, it increases the precision to force the first digit of the result to be a zero. For x (or X) conversion, a nonzero result will have 0x (or 0X) prepended to it. For e, E, f, g, and G conversions, the result will always contain a decimal-point character, even if no digits follow it (normally, a decimal-point character appears in the result of these conversions only if a digit follows it). For g and G conversions, trailing zeros will *not* be removed from the result. For other conversions, the behavior is undefined.

0 For d, i, o, u, x, X, e, E, f, g, and G conversions, leading zeros (following any indication of sign or base) are used to pad to the field width; no space padding is performed. If the 0 and – flags both appear, the 0 flag will be ignored. For d, i, o, u, x, and X conversions, if a precision is specified, the 0 flag will be ignored. For other conversions, the behavior is undefined.

The conversion specifiers and their meanings are

d,i,o,u,x,X The **int** argument is converted to signed decimal (d or i), unsigned octal (o), unsigned decimal (u), or unsigned hexadecimal notation (x or X); the letters **abcdef** are used for x conversion and the letters **ABCDEF** for X conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting a zero value with an explicit precision of zero is no characters.

`f` The `double` argument is converted to decimal notation in the style *[−]ddd.ddd*, where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is explicitly zero, no decimal-point character appears. If a decimal-point character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.

`e,E` The `double` argument is converted in the style *[−]d.ddde±dd*, where there is one digit before the decimal-point character (which is nonzero if the argument is nonzero) and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero, no decimal-point character appears. The value is rounded to the appropriate number of digits. The `E` conversion specifier will produce a number with `E` instead of `e` introducing the exponent. The exponent always contains at least two digits. If the value is zero, the exponent is zero.

`g,G` The `double` argument is converted in style `f` or `e` (or in style `E` in the case of a `G` conversion specifier), with the precision specifying the number of significant digits. If an explicit precision is zero, it is taken as 1. The style used depends on the value converted; style `e` will be used only if the exponent resulting from such a conversion is less than −4 or greater than or equal to the precision. Trailing zeros are removed from the fractional portion of the result; a decimal-point character appears only if it is followed by a digit.

`c` The `int` argument is converted to an `unsigned char`, and the resulting character is written.

`s` The argument shall be a pointer to an array of character type.[107] Characters from the array are written up to (but not including) a terminating null character; if the precision is specified, no more than that many characters are written. If the precision is not specified or is greater than the size of the array, the array shall contain a null character.

`p` The argument shall be a pointer to `void`. The value of the pointer is converted to a sequence of printable characters, in an implementation-defined manner.

`n` The argument shall be a pointer to an integer into which is *written* the number of characters written to the output stream so far by this call to `fprintf`. No argument is converted.

`%` A `%` is written. No argument is converted. The complete conversion specification shall be `%%`.

If a conversion specification is invalid, the behavior is undefined.[108]

If any argument is, or points to, a union or an aggregate (except for an array of character type using `%s` conversion, or a pointer cast to be a pointer to `void` using `%p` conversion), the behavior is undefined.

In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

107. No special provisions are made for multibyte characters.
108. See "future library directions" (§4.13.6).

Returns

The **fprintf** function returns the number of characters transmitted, or a negative value if an output error occurred.

Environmental limit

The minimum value for the maximum number of characters produced by any single conversion shall be 509.

Examples

To print a date and time in the form "Sunday, July 3, 10:02," where **weekday** and **month** are pointers to strings:

```
#include <stdio.h>
fprintf(stdout, "%s, %s %d, %.2d:%.2d\n",
        weekday, month, day, hour, min);
```

To print π to five decimal places:

```
#include <math.h>
#include <stdio.h>
fprintf(stdout, "pi = %.5f\n", 4 * atan(1.0));
```

4.9.6.2 The fscanf function

Synopsis

```
#include <stdio.h>
int fscanf(FILE *stream, const char *format, ...);
```

Description

The **fscanf** function reads input from the stream pointed to by **stream**, under control of the string pointed to by **format** that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored.

The format shall be a multibyte character sequence, beginning and ending in its initial shift state. The format is composed of zero or more directives: one or more white-space characters; an ordinary multibyte character (not **%**); or a conversion specification. Each conversion specification is introduced by the character **%**. After the **%**, the following appear in sequence:

- An optional assignment-suppressing character *****.
- An optional decimal integer that specifies the maximum field width.
- An optional **h**, **l** (ell) or **L** indicating the size of the receiving object. The conversion specifiers **d**, **i**, and **n** shall be preceded by **h** if the corresponding argument is a pointer to **short int** rather than a pointer to **int**, or by **l** if it is a pointer to **long int**. Similarly, the conversion specifiers **o**, **u**, and **x** shall be preceded by **h** if the corresponding argument is a pointer to **unsigned short int** rather than a pointer to **unsigned int**, or by **l** if it is a pointer to **unsigned long int**. Finally, the conversion specifiers **e**, **f**, and **g** shall be preceded by **l** if the corresponding argument is a pointer to **double** rather than a pointer to **float**, or by **L** if it is a pointer to **long double**. If an **h**, **l**, or **L** appears with any other conversion specifier, the behavior is undefined.
- A character that specifies the type of conversion to be applied. The valid conversion specifiers are described below.

The **fscanf** function executes each directive of the format in turn. If a directive fails, as detailed below, the **fscanf** function returns. Failures are described as input failures (due to the unavailability of input characters), or matching failures (due to inappropriate input).

A directive composed of white space is executed by reading input up to the first non-white-space character (which remains unread), or until no more characters can be read.

A directive that is an ordinary multibyte character is executed by reading the next characters of the stream. If one of the characters differs from one comprising the directive, the directive fails, and the differing and subsequent characters remain unread.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each specifier. A conversion specification is executed in the following steps:

Input white-space characters (as specified by the **isspace** function) are skipped, unless the specification includes a **[**, **c**, or **n** specifier.

An input item is read from the stream, unless the specification includes an **n** specifier. An input item is defined as the longest sequence of input characters (up to any specified maximum field width) which is an initial subsequence of a matching sequence. The first character, if any, after the input item remains unread. If the length of the input item is zero, the execution of the directive fails: this condition is a matching failure, unless an error prevented input from the stream, in which case it is an input failure.

Except in the case of a **%** specifier, the input item (or, in the case of a **%n** directive, the count of input characters) is converted to a type appropriate to the conversion specifier. If the input item is not a matching sequence, the execution of the directive fails: this condition is a matching failure. Unless assignment suppression was indicated by a *****, the result of the conversion is placed in the object pointed to by the first argument following the **format** argument that has not already received a conversion result. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the space provided, the behavior is undefined.

The following conversion specifiers are valid:

d Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the **strtol** function with the value 10 for the **base** argument. The corresponding argument shall be a pointer to integer.

i Matches an optionally signed integer, whose format is the same as expected for the subject sequence of the **strtol** function with the value 0 for the **base** argument. The corresponding argument shall be a pointer to integer.

o Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of the **strtoul** function with the value 8 for the **base** argument. The corresponding argument shall be a pointer to unsigned integer.

u Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the **strtoul** function with the value 10 for the **base** argument. The corresponding argument shall be a pointer to unsigned integer.

x Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of the **strtoul** function with the value 16 for the **base** argument. The corresponding argument shall be a pointer to unsigned integer.

e, f, g Matches an optionally signed floating-point number, whose format is the same as expected for the subject string of the **strtod** function. The corresponding argument shall be a pointer to floating.

s Matches a sequence of non-white-space characters.[109] The corresponding argument shall be a pointer to the initial character of an array large enough to accept the sequence and a terminating null character, which will be added automatically.

`[` Matches a nonempty sequence of characters[109] from a set of expected characters (the *scanset*). The corresponding argument shall be a pointer to the initial character of an array large enough to accept the sequence and a terminating null character, which will be added automatically. The conversion specifier includes all subsequent characters in the `format` string, up to and including the matching right bracket (`]`). The characters between the brackets (the *scanlist*) comprise the scanset, unless the character after the left bracket is a circumflex (`^`), in which case the scanset contains all characters that do not appear in the scanlist between the circumflex and the right bracket. As a special case, if the conversion specifier begins with `[]` or `[^]`, the right bracket character is in the scanlist and the next right bracket character is the matching right bracket that ends the specification. If a `-` character is in the scanlist and is not the first, nor the second where the first character is a `^`, nor the last character, the behavior is implementation-defined.

`c` Matches a sequence of characters[109] of the number specified by the field width (1 if no field width is present in the directive). The corresponding argument shall be a pointer to the initial character of an array large enough to accept the sequence. No null character is added.

`p` Matches an implementation-defined set of sequences, which should be the same as the set of sequences that may be produced by the `%p` conversion of the `fprintf` function. The corresponding argument shall be a pointer to a pointer to `void`. The interpretation of the input item is implementation-defined; however, for any input item other than a value converted earlier during the same program execution, the behavior of the `%p` conversion is undefined.

`n` No input is consumed. The corresponding argument shall be a pointer to integer into which is to be written the number of characters read from the input stream so far by this call to the `fscanf` function. Execution of a `%n` directive does not increment the assignment count returned at the completion of execution of the `fscanf` function.

`%` Matches a single `%`; no conversion or assignment occurs. The complete conversion specification shall be `%%`.

If a conversion specification is invalid, the behavior is undefined.[110]

The conversion specifiers `E`, `G`, and `X` are also valid and behave the same as, respectively, `e`, `g`, and `x`.

If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before any characters matching the current directive have been read (other than leading white space, where permitted), execution of the current directive terminates with an input failure; otherwise, unless execution of the current directive is terminated with a matching failure, execution of the following directive (if any) is terminated with an input failure.

If conversion terminates on a conflicting input character, the offending input character is left unread in the input stream. Trailing white space (including new-line characters) is left unread unless matched by a directive. The success of literal matches and suppressed assignments is not directly determinable other than via the `%n` directive.

109. No special provisions are made for multibyte characters.
110. See "future library directions" (§4.13.6).

Returns

The **fscanf** function returns the value of the macro **EOF** if an input failure occurs before any conversion. Otherwise, the **fscanf** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

Examples

The call:

```
#include <stdio.h>
int n, i; float x; char name[50];
n = fscanf(stdin, "%d%f%s", &i, &x, name);
```

with the input line:

```
25 54.32E-1 thompson
```

will assign to *n* the value 3, to *i* the value 25, to *x* the value 5.432, and *name* will contain **thompson\0**. Or:

```
#include <stdio.h>
int i; float x; char name[50];
fscanf(stdin, "%2d%f%*d %[0123456789]", &i, &x, name);
```

with input:

```
56789 0123 56a72
```

will assign to *i* the value 56 and to *x* the value 789.0, will skip 0123, and *name* will contain **56\0**. The next character read from the input stream will be a.

To accept repeatedly from **stdin** a quantity, a unit of measure and an item name:

```
#include <stdio.h>
int count; float quant; char units[21], item[21];
while (!feof(stdin) && !ferror(stdin)) {
      count = fscanf(stdin, "%f%20s of %20s",
            &quant, units, item);
      fscanf(stdin,"%*[^\n]");
}
```

If the **stdin** stream contains the following lines:

```
2 quarts of oil
-12.8degrees Celsius
lots of luck
10.0LBS     of     fertilizer
100ergs of energy
```

the execution of the above example will be equivalent to the following assignments:

```
quant = 2; strcpy(units, "quarts"); strcpy(item, "oil");
count = 3;
quant = -12.8; strcpy(units, "degrees");
count = 2; /* "C" fails to match "o" */
count = 0; /* "l" fails to match "%f" */
quant = 10.0; strcpy(units, "LBS"); strcpy(item, "fertilizer");
count = 3;
count = 0; /* "100e" fails to match "%f" */
count = EOF;
```

supporting additional file types that do truncate when written to, even when they are opened with the same sort of fopen call. Magnetic tape files are an example of a file type that must be handled this way. (On most tape hardware it is impossible to write to a tape without destroying immediately following data.) Hence tape files are not "binary files" within the meaning of the Standard. A conforming hosted implementation must provide (and document) at least one file type (on disk, most likely) that behaves exactly as specified in the Standard.

4.9.5.4 The freopen function

4.9.5.5 The setbuf function

setbuf is subsumed by setvbuf, but has been retained for compatibility with old code.

4.9.5.6 The setvbuf function

setvbuf has been adopted from UNIX System V, both to control the nature of stream buffering and to specify the size of I/O buffers. An implementation is not required to make actual use of a buffer provided for a stream, so a program must never expect the buffer's contents to reflect I/O operations. Further, the Standard does not require that the requested buffering be implemented; it merely mandates a standard mechanism for requesting whatever buffering services might be provided.

Although three types of buffering are defined, an implementation may choose to make one or more of them equivalent. For example, a library may choose to implement line-buffering for binary files as equivalent to unbuffered I/O or may choose to always implement full-buffering as equivalent to line-buffering.

The general principle is to provide portable code with a means of requesting the most appropriate popular buffering style, but not to require an implementation to support these styles.

4.9.6 Formatted input/output functions

4.9.6.1 The fprintf function

Use of the L modifier with floating conversions has been added to deal with formatted output of the new type long double.

Note that the %X and %x formats expect a corresponding int argument; %lX or %lx must be supplied with a long int argument.

The conversion specification %p has been added for pointer conversion, since the size of a pointer is not necessarily the same as the size of an int. Because an implementation may support more than one size of pointer, the corresponding argument is expected to be a (void *) pointer.

The %n format has been added to permit ascertaining the number of characters converted up to that point in the current invocation of the formatter.

Some pre-Standard implementations switch formats for %g at an exponent of −3 instead of (the Standard's) −4: existing code which requires the format switch at −3 will have to be changed.

Some existing implementations provide %D and %O as synonyms or replacements for %ld and %lo. The Committee considered the latter notation preferable.

The Committee has reserved lower case conversion specifiers for future standardization.

The use of leading zero in field widths to specify zero padding has been superseded by a precision field. The older mechanism has been retained.

Some implementations have provided the format %r as a means of indirectly passing a variable-length argument list. The functions `vfprintf`, etc., are considered to be a more controlled method of effecting this indirection, so %r was not adopted in the Standard. (See §4.9.6.7.)

4.9.6.2 The `fscanf` function

The specification of `fscanf` is based in part on these principles:

- As soon as one specified conversion fails, the whole function invocation fails.
- One-character pushback is sufficient for the implementation of `fscanf`;
- If a "flawed field" is detected, no value is stored for the corresponding argument.
- The conversions performed by `fscanf` are compatible with those performed by `strtod` and `strtol`.

Input pointer conversion with %p has been added, although it is obviously risky, for symmetry with `fprintf`. The %i format has been added to permit the scanner to determine the radix of the number in the input stream; the %n format has been added to make available the number of characters scanned thus far in the current invocation of the scanner.

White space is now defined by the `isspace` function. (See §4.3.1.9.)

An implementation must not use the `ungetc` function to perform the necessary one-character pushback. In particular, since the unmatched text is left "unread," the file position indicator as reported by the `ftell` function must be the position of the character remaining to be read. Furthermore, if the unread characters were themselves pushed back via `ungetc` calls, the pushback in `fscanf` must not affect the push-back stack in `ungetc`. A `scanf` call that matches N characters from a stream must leave the stream in the same state as if N consecutive `getc` calls had been issued.

4.9.6.3 The `printf` function

See comments of section §4.9.6.1 above.

4.10 GENERAL UTILITIES <stdlib.h>

The header **<stdlib.h>** declares four types and several functions of general utility, and defines several macros.[113]

The types declared are **size_t** and **wchar_t** (both described in §4.1.5),

div_t

which is a structure type that is the type of the value returned by the **div** function, and

ldiv_t

which is a structure type that is the type of the value returned by the **ldiv** function.

The macros defined are **NULL** (described in §4.1.5);

EXIT_FAILURE

and

EXIT_SUCCESS

which expand to integral expressions that may be used as the argument to the **exit** function to return unsuccessful or successful termination status, respectively, to the host environment;

RAND_MAX

which expands to an integral constant expression, the value of which is the maximum value returned by the **rand** function; and

MB_CUR_MAX

which expands to a positive integer expression whose value is the maximum number of bytes in a multibyte character for the extended character set specified by the current locale (category **LC_CTYPE**), and whose value is never greater than **MB_LEN_MAX**.

4.10.1 String conversion functions

The functions **atof**, **atoi**, and **atol** need not affect the value of the integer expression **errno** on an error. If the value of the result cannot be represented, the behavior is undefined.

4.10.1.1 The atof function

Synopsis

```
#include <stdlib.h>
double atof(const char *nptr);
```

Description

The **atof** function converts the initial portion of the string pointed to by **nptr** to **double** representation. Except for the behavior on error, it is equivalent to

```
strtod(nptr, (char **)NULL)
```

Returns

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

Forward references: the **strtod** function (§4.10.1.4).

113. See "future library directions" (§4.13.7).

4.10.1.2 The atoi function

Synopsis

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

Description

The **atoi** function converts the initial portion of the string pointed to by **nptr** to **int** representation. Except for the behavior on error, it is equivalent to

```
(int)strtol(nptr, (char **)NULL, 10)
```

Returns

The **atoi** function returns the converted value.

Forward references: the **strtol** function (§4.10.1.5).

4.10.1.3 The atol function

Synopsis

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

Description

The **atol** function converts the initial portion of the string pointed to by **nptr** to **long int** representation. Except for the behavior on error, it is equivalent to

```
strtol(nptr, (char **)NULL, 10)
```

Returns

The **atol** function returns the converted value.

Forward references: the **strtol** function (§4.10.1.5).

4.10.1.4 The strtod function

Synopsis

```
#include <stdlib.h>
double strtod(const char *nptr, char **endptr);
```

Description

The **strtod** function converts the initial portion of the string pointed to by **nptr** to **double** representation. First it decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by the **isspace** function), a subject sequence resembling a floating-point constant; and a final string of one or more unrecognized characters, including the terminating null character of the input string. Then it attempts to convert the subject sequence to a floating-point number, and returns the result.

The expected form of the subject sequence is an optional plus or minus sign, then a nonempty sequence of digits optionally containing a decimal-point character, then an optional exponent part as defined in §3.1.3.1, but no floating suffix. The subject sequence is defined as the longest subsequence of the input string, starting with the first non-white-space character, that is an initial subsequence of a sequence of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign, a digit, or a decimal-point character.

If the subject sequence has the expected form, the sequence of characters starting with the first digit or the decimal-point character (whichever occurs first) is interpreted as a floating constant according to the rules of §3.1.3.1, except that the decimal-point character is used in

place of a period, and that if neither an exponent part nor a decimal-point character appears, a decimal point is assumed to follow the last digit in the string. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final string is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

In other than the "C" locale, additional implementation-defined subject sequence forms may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of **nptr** is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

Returns

The **strtod** function returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value would cause overflow, plus or minus **HUGE_VAL** is returned (according to the sign of the value), and the value of the macro **ERANGE** is stored in **errno**. If the correct value would cause underflow, zero is returned and the value of the macro **ERANGE** is stored in **errno**.

4.10.1.5 The strtol function

Synopsis

```
#include <stdlib.h>
long int strtol(const char *nptr, char **endptr, int base);
```

Description

The **strtol** function converts the initial portion of the string pointed to by **nptr** to **long int** representation. First it decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by the **isspace** function), a subject sequence resembling an integer represented in some radix determined by the value of **base**, and a final string of one or more unrecognized characters, including the terminating null character of the input string. Then it attempts to convert the subject sequence to an integer, and returns the result.

If the value of **base** is zero, the expected form of the subject sequence is that of an integer constant as described in §3.1.3.2, optionally preceded by a plus or minus sign, but not including an integer suffix. If the value of **base** is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by **base**, optionally preceded by a plus or minus sign, but not including an integer suffix. The letters from **a** (or **A**) through **z** (or **Z**) are ascribed the values 10 to 35; only letters whose ascribed values are less than that of **base** are permitted. If the value of **base** is 16, the characters **0x** or **0X** may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest subsequence of the input string, starting with the first non-white-space character, that is an initial subsequence of a sequence of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of **base** is zero, the sequence of characters starting with the first digit is interpreted as an integer constant according to the rules of §3.1.3.2. If the subject sequence has the expected form and the value of **base** is between 2 and 36, it is used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final string is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

4.10.1 String conversion functions

4.10.1.1 The atof function

atof, atoi, and atol are subsumed by strtod and strtol, but have been retained because they are used extensively in existing code.

4.10.1.2 The atoi function

See §4.10.1.1.

4.10.1.3 The atol function

See §4.10.1.1.

4.10.1.4 The strtod function

strtod and strtol have been adopted (from UNIX System V) because they offer more control over the conversion process, and because they are required not to produce unexpected results on overflow during conversion.

The requirement that errno be set to EDOM when the argument string does not begin with a valid number string allows easy checking for invalid input.

4.10.1.5 The strtol function

See §4.10.1.4.

4.10.1.6 The strtoul function

strtoul was introduced by the Committee to provide a facility like strtol for unsigned long values. Simply using strtol in such cases could result in overflow upon conversion.

4.10.2 Pseudo-random sequence generation functions

4.10.2.1 The rand function

The Committee decided that an implementation should be allowed to provide a rand function which generates the best random sequence possible in that implementation, and therefore mandated no standard algorithm. It recognized the value, however, of being able to generate the same pseudo-random sequence in different implementations, and so it has published as an example in the Standard an algorithm that generates the same pseudo-random sequence in any conforming implementation, given the same seed.