The Red Hat newlib C Library

Full Configuration

libc 1.17.0
December 2008

Steve Chamberlain
Roland Pesch
Red Hat Support
Jeff Johnston
sac@cygnus.com, pesch@cygnus.com, jjohnstn@redhat.com  The Red Hat newlib C Library

`libc` includes software developed by the University of California, Berkeley and its contributors.

`libc` includes software developed by Martin Jackson, Graham Haley and Steve Chamberlain of Tadpole Technology and released to Cygnus.

`libc` uses floating-point conversion software developed at AT&T, which includes this copyright information:

```
The author of this software is David M. Gay.
Copyright (c) 1991 by AT&T.

Permission to use, copy, modify, and distribute this software for any purpose without fee is hereby granted, provided that this entire notice is included in all copies of any software which is or includes a copy or modification of this software and in all copies of the supporting documentation for such software.

THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED WARRANTY. IN PARTICULAR, NEITHER THE AUTHOR NOR AT&T MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
```

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, subject to the terms of the GNU General Public License, which includes the provision that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions.
1 Introduction

This reference manual describes the functions provided by the Red Hat “newlib” version of the standard ANSI C library. This document is not intended as an overview or a tutorial for the C library. Each library function is listed with a synopsis of its use, a brief description, return values (including error handling), and portability issues.

Some of the library functions depend on support from the underlying operating system and may not be available on every platform. For embedded systems in particular, many of these underlying operating system services may not be available or may not be fully functional. The specific operating system subroutines required for a particular library function are listed in the “Portability” section of the function description. See Chapter 12 [Syscalls], page 277, for a description of the relevant operating system calls.
2 Standard Utility Functions (‘stdlib.h’)

This chapter groups utility functions useful in a variety of programs. The corresponding declarations are in the header file ‘stdlib.h’.
2.1 _Exit—end program execution with no cleanup processing

Synopsis

```c
#include <stdlib.h>
void _Exit(int code);
```

Description

Use _Exit to return control from a program to the host operating environment. Use the argument `code` to pass an exit status to the operating environment: two particular values, `EXIT_SUCCESS` and `EXIT_FAILURE`, are defined in `stdlib.h` to indicate success or failure in a portable fashion.

.Exit differs from `exit` in that it does not run any application-defined cleanup functions registered with `atexit` and it does not clean up files and streams. It is identical to _exit.

Returns

.Exit does not return to its caller.

Portability

.Exit is defined by the C99 standard.

Supporting OS subroutines required: _exit.
2.2 a64l, l64a—convert between radix-64 ASCII string and long

Synopsis

```c
#include <stdlib.h>
long a64l(const char *input);
char *l64a(long input);
```

Description

Conversion is performed between long and radix-64 characters. The l64a routine transforms up to 32 bits of input value starting from least significant bits to the most significant bits. The input value is split up into a maximum of 5 groups of 6 bits and possibly one group of 2 bits (bits 31 and 30).

Each group of 6 bits forms a value from 0–63 which is translated into a character as follows:

- 0 = ‘.’
- 1 = ‘/’
- 2–11 = ’0’ to ’9’
- 12–37 = ’A’ to ’Z’
- 38–63 = ’a’ to ’z’

When the remaining bits are zero or all bits have been translated, a null terminator is appended to the string. An input value of 0 results in the empty string.

The a64l function performs the reverse translation. Each character is used to generate a 6-bit value for up to 30 bits and then a 2-bit value to complete a 32-bit result. The null terminator means that the remaining digits are 0. An empty input string or NULL string results in 0L. An invalid string results in undefined behavior. If the size of a long is greater than 32 bits, the result is sign-extended.

Returns

l64a returns a null-terminated string of 0 to 6 characters. a64l returns the 32-bit translated value from the input character string.

Portability

l64a and a64l are non-ANSI and are defined by the Single Unix Specification.

Supporting OS subroutines required: None.
2.3 abort—abnormal termination of a program

Synopsis

```c
#include <stdlib.h>
void abort(void);
```

Description
Use `abort` to signal that your program has detected a condition it cannot deal with. Normally, `abort` ends your program’s execution.

Before terminating your program, `abort` raises the exception `SIGABRT` (using `raise(SIGABRT)`). If you have used `signal` to register an exception handler for this condition, that handler has the opportunity to retain control, thereby avoiding program termination.

In this implementation, `abort` does not perform any stream- or file-related cleanup (the host environment may do so; if not, you can arrange for your program to do its own cleanup with a `SIGABRT` exception handler).

Returns
`abort` does not return to its caller.

Portability
ANSI C requires `abort`.
Supporting OS subroutines required: `_exit` and optionally, `write`. 
2.4 abs—integer absolute value (magnitude)

Synopsis

#include <stdlib.h>
int abs(int i);

Description

abs returns |x|, the absolute value of i (also called the magnitude of i). That is, if i is negative, the result is the opposite of i, but if i is nonnegative the result is i.

The similar function labs uses and returns long rather than int values.

Returns

The result is a nonnegative integer.

Portability

abs is ANSI.

No supporting OS subroutines are required.
2.5 assert—macro for debugging diagnostics

Synopsis

```c
#include <assert.h>
void assert(int expression);
```

Description

Use this macro to embed debugging diagnostic statements in your programs. The argument `expression` should be an expression which evaluates to true (nonzero) when your program is working as you intended.

When `expression` evaluates to false (zero), `assert` calls `abort`, after first printing a message showing what failed and where:

```
Assertion failed: expression, file filename, line lineno, function: func
```

If the name of the current function is not known (for example, when using a C89 compiler that does not understand `_func_`), the function location is omitted.

The macro is defined to permit you to turn off all uses of `assert` at compile time by defining `NDEBUG` as a preprocessor variable. If you do this, the `assert` macro expands to `(void(0))`

Returns

`assert` does not return a value.

Portability

The `assert` macro is required by ANSI, as is the behavior when `NDEBUG` is defined.

Supporting OS subroutines required (only if enabled): close, fstat, getpid, isatty, kill, lseek, read, sbrk, write.
2.6 `atexit`—request execution of functions at program exit

Synopsis

```c
#include <stdlib.h>
int atexit (void (*function)(void));
```

Description

You can use `atexit` to enroll functions in a list of functions that will be called when your program terminates normally. The argument is a pointer to a user-defined function (which must not require arguments and must not return a result).

The functions are kept in a LIFO stack; that is, the last function enrolled by `atexit` will be the first to execute when your program exits.

There is no built-in limit to the number of functions you can enroll in this list; however, after every group of 32 functions is enrolled, `atexit` will call `malloc` to get space for the next part of the list. The initial list of 32 functions is statically allocated, so you can always count on at least that many slots available.

Returns

`atexit` returns 0 if it succeeds in enrolling your function, -1 if it fails (possible only if no space was available for `malloc` to extend the list of functions).

Portability

`atexit` is required by the ANSI standard, which also specifies that implementations must support enrolling at least 32 functions.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`. 
2.7 atof, atoff—string to double or float

Synopsis

```c
#include <stdlib.h>
double atof(const char *s);
float atoff(const char *s);
```

Description

catof converts the initial portion of a string to a double. catoff converts the initial portion of a string to a float.

The functions parse the character string s, locating a substring which can be converted to a floating-point value. The substring must match the format:

```
[+|-]digits[.]digits[(e|E)[+|-]digits]
```

The substring converted is the longest initial fragment of s that has the expected format, beginning with the first non-whitespace character. The substring is empty if str is empty, consists entirely of whitespace, or if the first non-whitespace character is something other than +, -, ., or a digit.

atof(s) is implemented as `strtod(s, NULL)`. atoff(s) is implemented as `strtof(s, NULL)`.

Returns

atof returns the converted substring value, if any, as a double; or 0.0, if no conversion could be performed. If the correct value is out of the range of representable values, plus or minus HUGE_VAL is returned, and ERANGE is stored in errno. If the correct value would cause underflow, 0.0 is returned and ERANGE is stored in errno.

atoff obeys the same rules as atof, except that it returns a float.

Portability

atof is ANSI C. atof, atoi, and atol are subsumed by `strtol` and `strtol`, but are used extensively in existing code. These functions are less reliable, but may be faster if the argument is verified to be in a valid range.

Supporting OS subroutines required: close, fstat, isatty, lseek, read, sbrk, write.
2.8 atoi, atol—string to integer

Synopsis

```
#include <stdlib.h>
int atoi(const char *s);
long atol(const char *s);
int _atoi_r(struct _reent *ptr, const char *s);
long _atol_r(struct _reent *ptr, const char *s);
```

Description

`atoi` converts the initial portion of a string to an `int`. `atol` converts the initial portion of a string to a `long`.

`atoi(s)` is implemented as `(int)strtol(s, NULL, 10)`. `atol(s)` is implemented as `strtol(s, NULL, 10)`.

`_atoi_r` and `_atol_r` are reentrant versions of `atoi` and `atol` respectively, passing the reentrancy struct pointer.

Returns

The functions return the converted value, if any. If no conversion was made, 0 is returned.

Portability

`atoi`, `atol` are ANSI.

No supporting OS subroutines are required.
2.9 atoll—convert a string to a long long integer

Synopsis

```c
#include <stdlib.h>
long long atoll(const char *str);
long long _atoll_r(struct _reent *ptr, const char *str);
```

Description

The function atoll converts the initial portion of the string pointed to by *str to a type long long. A call to atoll(str) in this implementation is equivalent to strtoll(str, (char **)NULL, 10) including behavior on error.

The alternate function _atoll_r is a reentrant version. The extra argument reent is a pointer to a reentrancy structure.

Returns

The converted value.

Portability

atoll is ISO 9899 (C99) and POSIX 1003.1-2001 compatible.

No supporting OS subroutines are required.
2.10 **calloc**—allocate space for arrays

**Synopsis**

```
#include <stdlib.h>
void *calloc(size_t n, size_t s);
void *_calloc_r(void *reent, size_t n, size_t s);
```

**Description**

Use `calloc` to request a block of memory sufficient to hold an array of `n` elements, each of which has size `s`.

The memory allocated by `calloc` comes out of the same memory pool used by `malloc`, but the memory block is initialized to all zero bytes. (To avoid the overhead of initializing the space, use `malloc` instead.)

The alternate function `_calloc_r` is reentrant. The extra argument `reent` is a pointer to a reentrancy structure.

**Returns**

If successful, a pointer to the newly allocated space.

If unsuccessful, `NULL`.

**Portability**

calloc is ANSI.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.
2.11 div—divide two integers

Synopsis

```c
#include <stdlib.h>

div_t div(int n, int d);
```

Description

Divide \( n/d \), returning quotient and remainder as two integers in a structure `div_t`.

Returns

The result is represented with the structure

```c
typedef struct {
    int quot;
    int rem;
} div_t;
```

where the `quot` field represents the quotient, and `rem` the remainder. For nonzero `d`, if `'r = div(n, d);' then `n` equals `'r.rem + d*r.quot'`.

To divide `long` rather than `int` values, use the similar function `ldiv`.

Portability

div is ANSI.

No supporting OS subroutines are required.
2.12 ecvt, ecvtf, fcvt, fcvtf—double or float to string

Synopsis

```
#include <stdlib.h>

char *ecvt(double val, int chars, int *decpt, int *sgn);
char *ecvtf(float val, int chars, int *decpt, int *sgn);

char *fcvt(double val, int decimals,
       int *decpt, int *sgn);
char *fcvtf(float val, int decimals,
       int *decpt, int *sgn);
```

Description

`ecvt` and `fcvt` produce (null-terminated) strings of digits representing the double number `val`. `ecvtf` and `fcvtf` produce the corresponding character representations of float numbers.

(The stdlib functions `ecvtbuf` and `fcvtbuf` are reentrant versions of `ecvt` and `fcvt`.)

The only difference between `ecvt` and `fcvt` is the interpretation of the second argument (`chars` or `decimals`). For `ecvt`, the second argument `chars` specifies the total number of characters to write (which is also the number of significant digits in the formatted string, since these two functions write only digits). For `fcvt`, the second argument `decimals` specifies the number of characters to write after the decimal point; all digits for the integer part of `val` are always included.

Since `ecvt` and `fcvt` write only digits in the output string, they record the location of the decimal point in `*decpt`, and the sign of the number in `*sgn`. After formatting a number, `*decpt` contains the number of digits to the left of the decimal point. `*sgn` contains 0 if the number is positive, and 1 if it is negative.

Returns

All four functions return a pointer to the new string containing a character representation of `val`.

Portability

None of these functions are ANSI C.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`. 
2.13 `gcvt`, `gcvtf`—format double or float as string

Synopsis

```c
#include <stdlib.h>

char *gcvt(double val, int precision, char *buf);
char *gcvtf(float val, int precision, char *buf);
```

Description

`gcvt` writes a fully formatted number as a null-terminated string in the buffer `*buf`. `gcvtf` produces corresponding character representations of `float` numbers.

`gcvt` uses the same rules as the `printf` format `%.precisiong`—only negative values are signed (with `-'`), and either exponential or ordinary decimal-fraction format is chosen depending on the number of significant digits (specified by `precision`).

Returns

The result is a pointer to the formatted representation of `val` (the same as the argument `buf`).

Portability

Neither function is ANSI C.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`. 
2.14 `ecvtbuf`, `fcvtbuf`—double or float to string

Synopsis

```c
#include <stdio.h>

char *ecvtbuf(double val, int chars, int *decpt, 
               int *sgn, char *buf);

char *fcvtbuf(double val, int decimals, int *decpt, 
               int *sgn, char *buf);
```

Description

`ecvtbuf` and `fcvtbuf` produce (null-terminated) strings of digits representing the double number `val`.

The only difference between `ecvtbuf` and `fcvtbuf` is the interpretation of the second argument (`chars` or `decimals`). For `ecvtbuf`, the second argument `chars` specifies the total number of characters to write (which is also the number of significant digits in the formatted string, since these two functions write only digits). For `fcvtbuf`, the second argument `decimals` specifies the number of characters to write after the decimal point; all digits for the integer part of `val` are always included.

Since `ecvtbuf` and `fcvtbuf` write only digits in the output string, they record the location of the decimal point in `*decpt`, and the sign of the number in `*sgn`. After formatting a number, `*decpt` contains the number of digits to the left of the decimal point. `*sgn` contains 0 if the number is positive, and 1 if it is negative. For both functions, you supply a pointer `buf` to an area of memory to hold the converted string.

Returns

Both functions return a pointer to `buf`, the string containing a character representation of `val`.

Portability

Neither function is ANSI C.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`. 
2.15 __env_lock, __env_unlock—lock environ variable

Synopsis

```c
#include <envlock.h>
void __env_lock (struct _reent *reent);
void __env_unlock (struct _reent *reent);
```

Description

The `setenv` family of routines call these functions when they need to modify the `environ` variable. The version of these routines supplied in the library use the lock API defined in `sys/lock.h`. If multiple threads of execution can call `setenv`, or if `setenv` can be called reentrantly, then you need to define your own versions of these functions in order to safely lock the memory pool during a call. If you do not, the memory pool may become corrupted.

A call to `setenv` may call `__env_lock` recursively; that is, the sequence of calls may go `__env_lock, __env_lock, __env_unlock, __env_unlock`. Any implementation of these routines must be careful to avoid causing a thread to wait for a lock that it already holds.
2.16 exit—end program execution

Synopsis
#include <stdlib.h>
void exit(int code);

Description
Use exit to return control from a program to the host operating environment. Use the argument code to pass an exit status to the operating environment: two particular values, EXIT_SUCCESS and EXIT_FAILURE, are defined in 'stdlib.h' to indicate success or failure in a portable fashion.

exit does two kinds of cleanup before ending execution of your program. First, it calls all application-defined cleanup functions you have enrolled with atexit. Second, files and streams are cleaned up: any pending output is delivered to the host system, each open file or stream is closed, and files created by tmpfile are deleted.

Returns
exit does not return to its caller.

Portability
ANSI C requires exit, and specifies that EXIT_SUCCESS and EXIT_FAILURE must be defined. Supporting OS subroutines required: _exit.
2.17 getenv—look up environment variable

Synopsis

```c
#include <stdlib.h>
char *getenv(const char *name);
```

Description

getenv searches the list of environment variable names and values (using the global pointer “char **environ”) for a variable whose name matches the string at name. If a variable name matches, getenv returns a pointer to the associated value.

Returns

A pointer to the (string) value of the environment variable, or NULL if there is no such environment variable.

Portability

getenv is ANSI, but the rules for properly forming names of environment variables vary from one system to another.

getenv requires a global pointer environ.
2.18 labs—long integer absolute value

Synopsis

```
#include <stdlib.h>
long labs(long i);
```

Description

`labs` returns \(|x|\), the absolute value of \(i\) (also called the magnitude of \(i\)). That is, if \(i\) is negative, the result is the opposite of \(i\), but if \(i\) is nonnegative the result is \(i\).

The similar function `abs` uses and returns `int` rather than `long` values.

Returns

The result is a nonnegative long integer.

Portability

`labs` is ANSI.

No supporting OS subroutine calls are required.
2.19 ldiv—divide two long integers

Synopsis

```c
#include <stdlib.h>
ldiv_t ldiv(long n, long d);
```

Description

Divide \( n/d \), returning quotient and remainder as two long integers in a structure `ldiv_t`.

Returns

The result is represented with the structure

```c
typedef struct {
    long quot;
    long rem;
} ldiv_t;
```

where the `quot` field represents the quotient, and `rem` the remainder. For nonzero \( d \), if `r = ldiv(n, d);` then \( n \) equals `r.rem + d*r.quot`.

To divide `int` rather than `long` values, use the similar function `div`.

Portability

`ldiv` is ANSI.

No supporting OS subroutines are required.
2.20 `llabs`—compute the absolute value of an long long integer.

Synopsis

```c
#include <stdlib.h>
long long llabs(long long j);
```

Description

The `llabs` function computes the absolute value of the long long integer argument `j` (also called the magnitude of `j`).

The similar function `labs` uses and returns `long` rather than `long long` values.

Returns

A nonnegative long long integer.

Portability

`llabs` is ISO 9899 (C99) compatible.

No supporting OS subroutines are required.
2.21 lldiv—divide two long long integers

Synopsis

```c
#include <stdlib.h>
lldiv_t lldiv(long long n, long long d);
```

Description

Divide \( n/d \), returning quotient and remainder as two long long integers in a structure `lldiv_t`.

Returns

The result is represented with the structure

```c
typedef struct
{
  long long quot;
  long long rem;
} lldiv_t;
```

where the `quot` field represents the quotient, and `rem` the remainder. For nonzero \( d \), if `\( r = lldiv(n, d); \)` then \( n \) equals `\( r.rem + d*r.quot \)`.

To divide `long` rather than `long long` values, use the similar function `ldiv`.

Portability

`lldiv` is ISO 9899 (C99) compatible.

No supporting OS subroutines are required.
2.22 malloc, realloc, free—manage memory

Synopsis

```c
#include <stdlib.h>

void *malloc(size_t nbytes);
void *realloc(void *aptr, size_t nbytes);
void *reallocf(void *aptr, size_t nbytes);
void free(void *aptr);

void *memalign(size_t align, size_t nbytes);

size_t malloc_usable_size(void *aptr);

void *__malloc_r(void *reent, size_t nbytes);
void *__realloc_r(void *reent,
    void *aptr, size_t nbytes);
void *__reallocf_r(void *reent,
    void *aptr, size_t nbytes);
void __free_r(void *reent, void *aptr);

void *__memalign_r(void *reent,
    size_t align, size_t nbytes);

size_t __malloc_usable_size_r(void *reent, void *aptr);
```

Description

These functions manage a pool of system memory.

Use `malloc` to request allocation of an object with at least `nbytes` bytes of storage available. If the space is available, `malloc` returns a pointer to a newly allocated block as its result.

If you already have a block of storage allocated by `malloc`, but you no longer need all the space allocated to it, you can make it smaller by calling `realloc` with both the object pointer and the new desired size as arguments. `realloc` guarantees that the contents of the smaller object match the beginning of the original object.

Similarly, if you need more space for an object, use `realloc` to request the larger size; again, `realloc` guarantees that the beginning of the new, larger object matches the contents of the original object.

When you no longer need an object originally allocated by `malloc` or `realloc` (or the related function `calloc`), return it to the memory storage pool by calling `free` with the address of the object as the argument. You can also use `realloc` for this purpose by calling it with 0 as the `nbytes` argument.

The `reallocf` function behaves just like `realloc` except if the function is required to allocate new storage and this fails. In this case `reallocf` will free the original object passed in whereas `realloc` will not.

The `memalign` function returns a block of size `nbytes` aligned to a `align` boundary. The `align` argument must be a power of two.
The `malloc_usable_size` function takes a pointer to a block allocated by `malloc`. It returns the amount of space that is available in the block. This may or may not be more than the size requested from `malloc`, due to alignment or minimum size constraints.

The alternate functions `_malloc_r`, `_realloc_r`, `_reallocf_r`, `_free_r`, `_memalign_r`, and `_malloc_usable_size_r` are reentrant versions. The extra argument `reent` is a pointer to a reentrancy structure.

If you have multiple threads of execution which may call any of these routines, or if any of these routines may be called reentrantly, then you must provide implementations of the `__malloc_lock` and `__malloc_unlock` functions for your system. See the documentation for those functions.

These functions operate by calling the function `_sbrk_r` or `sbrk`, which allocates space. You may need to provide one of these functions for your system. `_sbrk_r` is called with a positive value to allocate more space, and with a negative value to release previously allocated space if it is no longer required. See Section 12.1 [Stubs], page 277.

**Returns**

`malloc` returns a pointer to the newly allocated space, if successful; otherwise it returns `NULL`. If your application needs to generate empty objects, you may use `malloc(0)` for this purpose.

`realloc` returns a pointer to the new block of memory, or `NULL` if a new block could not be allocated. `NULL` is also the result when you use `realloc(aptr,0)` (which has the same effect as `free(aptr)`). You should always check the result of `realloc`; successful reallocation is not guaranteed even when you request a smaller object.

`free` does not return a result.

`memalign` returns a pointer to the newly allocated space.

`malloc_usable_size` returns the usable size.

**Portability**

`malloc`, `realloc`, and `free` are specified by the ANSI C standard, but other conforming implementations of `malloc` may behave differently when `nbytes` is zero.

`memalign` is part of SVR4.

`malloc_usable_size` is not portable.

Supporting OS subroutines required: `sbrk`. 

---

Red Hat newlib C Library, Full
Chapter 2: Standard Utility Functions (‘stdlib.h’) 27

2.23 mallinfo, malloc_stats, mallopt—malloc support

Synopsis

#include <malloc.h>

struct mallinfo mallinfo(void);
void malloc_stats(void);
int mallopt(int parameter, value);

struct mallinfo _mallinfo_r(void *reent);
void _malloc_stats_r(void *reent);
int _mallopt_r(void *reent, int parameter, value);

Description

mallinfo returns a structure describing the current state of memory allocation. The structure is defined in malloc.h. The following fields are defined: arena is the total amount of space in the heap; ordblks is the number of chunks which are not in use; uordblks is the total amount of space allocated by malloc; fordblks is the total amount of space not in use; keepcost is the size of the top most memory block.

malloc_stats print some statistics about memory allocation on standard error.

mallopt takes a parameter and a value. The parameters are defined in malloc.h, and may be one of the following: M_TRIM_THRESHOLD sets the maximum amount of unused space in the top most block before releasing it back to the system in free (the space is released by calling _sbrk_r with a negative argument); M_TOP_PAD is the amount of padding to allocate whenever _sbrk_r is called to allocate more space.

The alternate functions _mallinfo_r, _malloc_stats_r, and _mallopt_r are reentrant versions. The extra argument reent is a pointer to a reentrancy structure.

Returns

mallinfo returns a mallinfo structure. The structure is defined in malloc.h.

malloc_stats does not return a result.

mallopt returns zero if the parameter could not be set, or non-zero if it could be set.

Portability

mallinfo and mallopt are provided by SVR4, but mallopt takes different parameters on different systems. malloc_stats is not portable.
2.24 __malloc_lock, __malloc_unlock—lock malloc pool

Synopsis

```c
#include <malloc.h>
void __malloc_lock (struct _reent *reent);
void __malloc_unlock (struct _reent *reent);
```

Description

The `malloc` family of routines call these functions when they need to lock the memory pool. The version of these routines supplied in the library use the lock API defined in `sys/lock.h`. If multiple threads of execution can call `malloc`, or if `malloc` can be called reentrantly, then you need to define your own versions of these functions in order to safely lock the memory pool during a call. If you do not, the memory pool may become corrupted.

A call to `malloc` may call `__malloc_lock` recursively; that is, the sequence of calls may go `__malloc_lock`, `__malloc_lock`, `__malloc_unlock`, `__malloc_unlock`. Any implementation of these routines must be careful to avoid causing a thread to wait for a lock that it already holds.
2.25 mblen—minimal multibyte length function

Synopsis

```c
#include <stdlib.h>
int mblen(const char *s, size_t n);
```

Description

When _MB_CAPABLE is not defined, this is a minimal ANSI-conforming implementation of mblen. In this case, the only “multi-byte character sequences” recognized are single bytes, and thus 1 is returned unless s is the null pointer or has a length of 0 or is the empty string.

When _MB_CAPABLE is defined, this routine calls _mbtowc_r to perform the conversion, passing a state variable to allow state dependent decoding. The result is based on the locale setting which may be restricted to a defined set of locales.

Returns

This implementation of mblen returns 0 if s is NULL or the empty string; it returns 1 if not _MB_CAPABLE or the character is a single-byte character; it returns -1 if the multi-byte character is invalid; otherwise it returns the number of bytes in the multibyte character.

Portability

mblen is required in the ANSI C standard. However, the precise effects vary with the locale. mblen requires no supporting OS subroutines.
2.26 mbsrtowcs, mbsnrtowcs—convert a character string to a wide-character string

Synopsis

```c
#include <wchar.h>
size_t mbsrtowcs(wchar_t *dst, const char **src, size_t len,
                  mbstate_t *ps);

#include <wchar.h>
size_t _mbsrtowcs_r(struct _reent *ptr, wchar_t *dst,
                    const char **src, size_t len,
                    mbstate_t *ps);

#include <wchar.h>
size_t mbsnrtowcs(wchar_t *dst, const char **src,
                  size_t nms, size_t len, mbstate_t *ps);

#include <wchar.h>
size_t _mbsnrtowcs_r(struct _reent *ptr, wchar_t *dst,
                     const char **src, size_t nms,
                     size_t len, mbstate_t *ps);
```

Description

The `mbsrtowcs` function converts a sequence of multibyte characters pointed to indirectly by `src` into a sequence of corresponding wide characters and stores at most `len` of them in the `wchar_t` array pointed to by `dst`, until it encounters a terminating null character (`\0`). If `dst` is NULL, no characters are stored.

If `dst` is not NULL, the pointer pointed to by `src` is updated to point to the character after the one that conversion stopped at. If conversion stops because a null character is encountered, `*src` is set to NULL.

The `mbstate_t` argument, `ps`, is used to keep track of the shift state. If it is NULL, `mbsrtowcs` uses an internal, static `mbstate_t` object, which is initialized to the initial conversion state at program startup.

The `mbsnrtowcs` function behaves identically to `mbsrtowcs`, except that conversion stops after reading at most `nms` bytes from the buffer pointed to by `src`.

Returns

The `mbsrtowcs` and `mbsnrtowcs` functions return the number of wide characters stored in the array pointed to by `dst` if successful, otherwise it returns `(size_t)-1`.

Portability

`mbsrtowcs` is defined by the C99 standard. `mbsnrtowcs` is defined by the POSIX.1-2008 standard.
2.27 mbstowcs—minimal multibyte string to wide char converter

Synopsis

```c
#include <stdlib.h>
int mbstowcs(wchar_t *pwc, const char *s, size_t n);
```

Description

When MB_CAPABLE is not defined, this is a minimal ANSI-conforming implementation of mbstowcs. In this case, the only “multi-byte character sequences” recognized are single bytes, and they are “converted” to wide-char versions simply by byte extension.

When MB_CAPABLE is defined, this routine calls _mbstowcs_r to perform the conversion, passing a state variable to allow state dependent decoding. The result is based on the locale setting which may be restricted to a defined set of locales.

Returns

This implementation of mbstowcs returns 0 if s is NULL or is the empty string; it returns -1 if MB_CAPABLE and one of the multi-byte characters is invalid or incomplete; otherwise it returns the minimum of: n or the number of multi-byte characters in s plus 1 (to compensate for the nul character). If the return value is -1, the state of the pwc string is indeterminate. If the input has a length of 0, the output string will be modified to contain a wchar_t nul terminator.

Portability

mbstowcs is required in the ANSI C standard. However, the precise effects vary with the locale.

mbstowcs requires no supporting OS subroutines.
2.28 mbtowc—minimal multibyte to wide char converter

Synopsis

```c
#include <stdlib.h>
int mbtowc(wchar_t *pwc, const char *s, size_t n);
```

Description

When _MB_CAPABLE is not defined, this is a minimal ANSI-conforming implementation of mbtowc. In this case, only “multi-byte character sequences” recognized are single bytes, and they are “converted” to themselves. Each call to mbtowc copies one character from *s to *pwc, unless s is a null pointer. The argument n is ignored.

When _MB_CAPABLE is defined, this routine calls _mbtowc_r to perform the conversion, passing a state variable to allow state dependent decoding. The result is based on the locale setting which may be restricted to a defined set of locales.

Returns

This implementation of mbtowc returns 0 if s is NULL or is the empty string; it returns 1 if not _MB_CAPABLE or the character is a single-byte character; it returns -1 if n is 0 or the multi-byte character is invalid; otherwise it returns the number of bytes in the multibyte character. If the return value is -1, no changes are made to the pwc output string. If the input is the empty string, a wchar_t nul is placed in the output string and 0 is returned. If the input has a length of 0, no changes are made to the pwc output string.

Portability

mbtowc is required in the ANSI C standard. However, the precise effects vary with the locale.

mbtowc requires no supporting OS subroutines.
2.29 on_exit—request execution of function with argument at program exit

Synopsis

    #include <stdlib.h>
    int on_exit (void (*function)(int, void *), void *arg);

Description

You can use on_exit to enroll functions in a list of functions that will be called when your program terminates normally. The argument is a pointer to a user-defined function which takes two arguments. The first is the status code passed to exit and the second argument is of type pointer to void. The function must not return a result. The value of arg is registered and passed as the argument to function.

The functions are kept in a LIFO stack; that is, the last function enrolled by atexit or on_exit will be the first to execute when your program exits. You can intermix functions using atexit and on_exit.

There is no built-in limit to the number of functions you can enroll in this list; however, after every group of 32 functions is enrolled, atexit/on_exit will call malloc to get space for the next part of the list. The initial list of 32 functions is statically allocated, so you can always count on at least that many slots available.

Returns

on_exit returns 0 if it succeeds in enrolling your function, -1 if it fails (possible only if no space was available for malloc to extend the list of functions).

Portability

on_exit is a non-standard glibc extension

Supporting OS subroutines required: None
# 2.30 rand, srand—pseudo-random numbers

**Synopsis**

```c
#include <stdlib.h>
int rand(void);
void srand(unsigned int seed);
int rand_r(unsigned int *seed);
```

**Description**

`rand` returns a different integer each time it is called; each integer is chosen by an algorithm designed to be unpredictable, so that you can use `rand` when you require a random number. The algorithm depends on a static variable called the “random seed”; starting with a given value of the random seed always produces the same sequence of numbers in successive calls to `rand`.

You can set the random seed using `srand`; it does nothing beyond storing its argument in the static variable used by `rand`. You can exploit this to make the pseudo-random sequence less predictable, if you wish, by using some other unpredictable value (often the least significant parts of a time-varying value) as the random seed before beginning a sequence of calls to `rand`; or, if you wish to ensure (for example, while debugging) that successive runs of your program use the same “random” numbers, you can use `srand` to set the same random seed at the outset.

**Returns**

`rand` returns the next pseudo-random integer in sequence; it is a number between 0 and `RAND_MAX` (inclusive).

`srand` does not return a result.

**Portability**

`rand` is required by ANSI, but the algorithm for pseudo-random number generation is not specified; therefore, even if you use the same random seed, you cannot expect the same sequence of results on two different systems.

`rand` requires no supporting OS subroutines.
Chapter 2: Standard Utility Functions (`stdlib.h')

2.3.1 rand48, drand48, erand48, lrand48, nrand48, mrand48, jrand48, srand48, seed48, lcong48—pseudo-random number generators and initialization routines

Synopsis

```c
#include <stdlib.h>
double drand48(void);
double erand48(unsigned short xseed[3]);
long lrand48(void);
long nrand48(unsigned short xseed[3]);
long mrand48(void);
long jrand48(unsigned short xseed[3]);
void srand48(long seed);
unsigned short *seed48(unsigned short xseed[3]);
void lcong48(unsigned short p[7]);
```

Description

The `rand48' family of functions generates pseudo-random numbers using a linear congruential algorithm working on integers 48 bits in size. The particular formula employed is \( r(n+1) = (a \times r(n) + c) \mod m \) where the default values are for the multiplicand \( a = 0xfdeece66d = 25214903917 \) and the addend \( c = 0xb = 11 \). The modulo is always fixed at \( m = 2^{48} \).

\( r(n) \) is called the seed of the random number generator.

For all the six generator routines described next, the first computational step is to perform a single iteration of the algorithm.

`drand48' and `erand48' return values of type double. The full 48 bits of \( r(n+1) \) are loaded into the mantissa of the returned value, with the exponent set such that the values produced lie in the interval \([0.0, 1.0]\).

`lrand48' and `nrand48' return values of type long in the range \([0, 2^{31}-1]\). The high-order (31) bits of \( r(n+1) \) are loaded into the lower bits of the returned value, with the topmost (sign) bit set to zero.

`mrand48' and `jrand48' return values of type long in the range \([-2^{31}, 2^{31}-1]\). The high-order (32) bits of \( r(n+1) \) are loaded into the returned value.

`drand48', `lrand48', and `mrand48' use an internal buffer to store \( r(n) \). For these functions, the initial value of \( r(0) = 0x1234abcd330e = 20017429951246 \).

On the other hand, `erand48', `nrand48', and `jrand48' use a user-supplied buffer to store the seed \( r(n) \), which consists of an array of 3 shorts, where the zeroth member holds the least significant bits.

All functions share the same multiplicand and addend.

`srand48' is used to initialize the internal buffer \( r(n) \) of `drand48', `lrand48', and `mrand48' such that the 32 bits of the seed value are copied into the upper 32 bits of \( r(n) \), with the lower 16 bits of \( r(n) \) arbitrarily being set to \( 0x330e \). Additionally, the constant multiplicand and addend of the algorithm are reset to the default values given above.

`seed48' also initializes the internal buffer \( r(n) \) of `drand48', `lrand48', and `mrand48', but here all 48 bits of the seed can be specified in an array of 3 shorts, where the zeroth member specifies the lowest bits. Again, the constant multiplicand and addend of the algorithm are reset to the default values given above. `seed48' returns a pointer to an array of 3 shorts.
which contains the old seed. This array is statically allocated, thus its contents are lost after each new call to seed48.

Finally, lcong48 allows full control over the multiplicand and addend used in drand48, erand48, lrand48, nrand48, mrand48, and jrand48, and the seed used in drand48, lrand48, and mrand48. An array of 7 shorts is passed as parameter; the first three shorts are used to initialize the seed; the second three are used to initialize the multiplicand; and the last short is used to initialize the addend. It is thus not possible to use values greater than 0xffff as the addend.

Note that all three methods of seeding the random number generator always also set the multiplicand and addend for any of the six generator calls.

For a more powerful random number generator, see random.

**Portability**

SUS requires these functions.

No supporting OS subroutines are required.
2.32 strtod, strtof—string to double or float

Synopsis

```
#include <stdlib.h>

double strtod(const char *str, char **tail);
float strtof(const char *str, char **tail);

double _strtod_r(void *reent,
                 const char *str, char **tail);
```

Description

The function `strtod` parses the character string `str`, producing a substring which can be converted to a double value. The substring converted is the longest initial subsequence of `str`, beginning with the first non-whitespace character, that has one of these formats:

- `[+-]digits[.digits][(e|E)[+-]digits]`
- `[+-].digits[(e|E)[+-]digits]`
- `[+-](i|I)(n|N)(f|F)[(i|I)(n|N)(i|I)(t|T)(y|Y)]`
- `[+-](n|N)(a|A)(n|N)[x<->[hexdigits]]<]`
- `[+-]0(x|X)hexdigits[.hexdigits][(p|P)[+-]digits]`
- `[+-]0(x|X).hexdigits[(p|P)[+-]digits]`

The substring contains no characters if `str` is empty, consists entirely of whitespace, or if the first non-whitespace character is something other than `+`, `-`, `.`, or a digit, and cannot be parsed as infinity or NaN. If the platform does not support NaN, then NaN is treated as an empty substring. If the substring is empty, no conversion is done, and the value of `str` is stored in `*tail`. Otherwise, the substring is converted, and a pointer to the final string (which will contain at least the terminating null character of `str`) is stored in `*tail`. If you want no assignment to `*tail`, pass a null pointer as `tail`. `strtof` is identical to `strtod` except for its return type.

This implementation returns the nearest machine number to the input decimal string. Ties are broken by using the IEEE round-even rule. However, `strtof` is currently subject to double rounding errors.

The alternate function `_strtod_r` is a reentrant version. The extra argument `reent` is a pointer to a reentrancy structure.

Returns

`strtod` returns the converted substring value, if any. If no conversion could be performed, 0 is returned. If the correct value is out of the range of representable values, plus or minus `HUGE_VAL` is returned, and `ERANGE` is stored in `errno`. If the correct value would cause underflow, 0 is returned and `ERANGE` is stored in `errno`.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`. 
2.33 strtol—string to long

Synopsis

```c
#include <stdlib.h>
long strtol(const char *s, char **ptr, int base);

long _strtol_r(void *reent,
    const char *s, char **ptr, int base);
```

Description

The function `strtol` converts the string `*s` to a `long`. First, it breaks down the string into three parts: leading whitespace, which is ignored; a subject string consisting of characters resembling an integer in the radix specified by `base`; and a trailing portion consisting of zero or more unparsable characters, and always including the terminating null character. Then, it attempts to convert the subject string into a `long` and returns the result.

If the value of `base` is 0, the subject string is expected to look like a normal C integer constant: an optional sign, a possible ‘0x’ indicating a hexadecimal base, and a number. If `base` is between 2 and 36, the expected form of the subject is a sequence of letters and digits representing an integer in the radix specified by `base`, with an optional plus or minus sign. The letters a–z (or, equivalently, A–Z) are used to signify values from 10 to 35; only letters whose ascribed values are less than `base` are permitted. If `base` is 16, a leading `0x` is permitted.

The subject sequence is the longest initial sequence of the input string that has the expected form, starting with the first non-whitespace character. If the string is empty or consists entirely of whitespace, or if the first non-whitespace character is not a permissible letter or digit, the subject string is empty.

If the subject string is acceptable, and the value of `base` is zero, `strtol` attempts to determine the radix from the input string. A string with a leading `0x` is treated as a hexadecimal value; a string with a leading `0` and no `x` is treated as octal; all other strings are treated as decimal. If `base` is between 2 and 36, it is used as the conversion radix, as described above. If the subject string begins with a minus sign, the value is negated. Finally, a pointer to the first character past the converted subject string is stored in `ptr`, if `ptr` is not `NULL`.

If the subject string is empty (or not in acceptable form), no conversion is performed and the value of `s` is stored in `ptr` (if `ptr` is not `NULL`).

The alternate function `_strtol_r` is a reentrant version. The extra argument `reent` is a pointer to a reentrancy structure.

Returns

`strtol` returns the converted value, if any. If no conversion was made, 0 is returned.

`strtol` returns `LONG_MAX` or `LONG_MIN` if the magnitude of the converted value is too large, and sets `errno` to `ERANGE`.

Portability

`strtol` is ANSI.

No supporting OS subroutines are required.
Chapter 2: Standard Utility Functions (‘stdlib.h’)
2.34 `strtoll`—string to long long

Synopsis

```c
#include <stdlib.h>
long long strtoll(const char *s, char **ptr, int base);

long long _strtoll_r(void *reent, const char *s, char **ptr, int base);
```

Description

The function `strtoll` converts the string `*s` to a `long long`. First, it breaks down the string into three parts: leading whitespace, which is ignored; a subject string consisting of characters resembling an integer in the radix specified by `base`; and a trailing portion consisting of zero or more unparsable characters, and always including the terminating null character. Then, it attempts to convert the subject string into a `long long` and returns the result.

If the value of `base` is 0, the subject string is expected to look like a normal C integer constant: an optional sign, a possible ‘0x’ indicating a hexadecimal base, and a number. If `base` is between 2 and 36, the expected form of the subject is a sequence of letters and digits representing an integer in the radix specified by `base`, with an optional plus or minus sign. The letters a–z (or, equivalently, A–Z) are used to signify values from 10 to 35; only letters whose ascribed values are less than `base` are permitted. If `base` is 16, a leading 0x is permitted.

The subject sequence is the longest initial sequence of the input string that has the expected form, starting with the first non-whitespace character. If the string is empty or consists entirely of whitespace, or if the first non-whitespace character is not a permissible letter or digit, the subject string is empty.

If the subject string is acceptable, and the value of `base` is zero, `strtoll` attempts to determine the radix from the input string. A string with a leading 0x is treated as a hexadecimal value; a string with a leading 0 and no x is treated as octal; all other strings are treated as decimal. If `base` is between 2 and 36, it is used as the conversion radix, as described above. If the subject string begins with a minus sign, the value is negated. Finally, a pointer to the first character past the converted subject string is stored in `ptr`, if `ptr` is not `NULL`.

If the subject string is empty (or not in acceptable form), no conversion is performed and the value of `s` is stored in `ptr` (if `ptr` is not `NULL`).

The alternate function `_strtoll_r` is a reentrant version. The extra argument `reent` is a pointer to a reentrancy structure.

Returns

`strtoll` returns the converted value, if any. If no conversion was made, 0 is returned. `strtoll` returns `LONG_LONG_MAX` or `LONG_LONG_MIN` if the magnitude of the converted value is too large, and sets `errno` to `ERANGE`.

Portability

`strtoll` is ANSI.
No supporting OS subroutines are required.
2.35 strtoul—string to unsigned long

Synopsis

```c
#include <stdlib.h>

unsigned long strtoul(const char *s, char **ptr, int base);

unsigned long _strtoul_r(void *reent, const char *s, char **ptr, int base);
```

Description

The function `strtoul` converts the string `*s` to an `unsigned long`. First, it breaks down the string into three parts: leading whitespace, which is ignored; a subject string consisting of the digits meaningful in the radix specified by `base` (for example, 0 through 7 if the value of `base` is 8); and a trailing portion consisting of one or more unparsable characters, which always includes the terminating null character. Then, it attempts to convert the subject string into an unsigned long integer, and returns the result.

If the value of `base` is zero, the subject string is expected to look like a normal C integer constant (save that no optional sign is permitted): a possible 0x indicating hexadecimal radix, and a number. If `base` is between 2 and 36, the expected form of the subject is a sequence of digits (which may include letters, depending on the base) representing an integer in the radix specified by `base`. The letters a–z (or A–Z) are used as digits valued from 10 to 35. If `base` is 16, a leading 0x is permitted.

The subject sequence is the longest initial sequence of the input string that has the expected form, starting with the first non-whitespace character. If the string is empty or consists entirely of whitespace, or if the first non-whitespace character is not a permissible digit, the subject string is empty.

If the subject string is acceptable, and the value of `base` is zero, `strtoul` attempts to determine the radix from the input string. A string with a leading 0x is treated as a hexadecimal value; a string with a leading 0 and no x is treated as octal; all other strings are treated as decimal. If `base` is between 2 and 36, it is used as the conversion radix, as described above. Finally, a pointer to the first character past the converted subject string is stored in `ptr`, if `ptr` is not NULL.

If the subject string is empty (that is, if `*s` does not start with a substring in acceptable form), no conversion is performed and the value of `s` is stored in `ptr` (if `ptr` is not NULL).

The alternate function `_strtoul_r` is a reentrant version. The extra argument `reent` is a pointer to a reentrancy structure.

Returns

`strtoul` returns the converted value, if any. If no conversion was made, 0 is returned.

`strtoul` returns ULONG_MAX if the magnitude of the converted value is too large, and sets `errno` to ERANGE.

Portability

`strtoul` is ANSI.
strtol requires no supporting OS subroutines.
2.36 **strtoull**—string to unsigned long long

**Synopsis**

```c
#include <stdlib.h>

unsigned long long strtoull(const char *s, char **ptr,
    int base);

unsigned long long _strtoull_r(void *reent, const char *s,
    char **ptr, int base);
```

**Description**

The function **strtoull** converts the string *s* to an **unsigned long long**. First, it breaks down the string into three parts: leading whitespace, which is ignored; a subject string consisting of the digits meaningful in the radix specified by **base** (for example, 0 through 7 if the value of **base** is 8); and a trailing portion consisting of one or more unparsable characters, which always includes the terminating null character. Then, it attempts to convert the subject string into an unsigned long long integer, and returns the result.

If the value of **base** is zero, the subject string is expected to look like a normal C integer constant (save that no optional sign is permitted): a possible 0x indicating hexadecimal radix, and a number. If **base** is between 2 and 36, the expected form of the subject is a sequence of digits (which may include letters, depending on the base) representing an integer in the radix specified by **base**. The letters a–z (or A–Z) are used as digits valued from 10 to 35. If **base** is 16, a leading 0x is permitted.

The subject sequence is the longest initial sequence of the input string that has the expected form, starting with the first non-whitespace character. If the string is empty or consists entirely of whitespace, or if the first non-whitespace character is not a permissible digit, the subject string is empty.

If the subject string is acceptable, and the value of **base** is zero, **strtoull** attempts to determine the radix from the input string. A string with a leading 0x is treated as a hexadecimal value; a string with a leading 0 and no x is treated as octal; all other strings are treated as decimal. If **base** is between 2 and 36, it is used as the conversion radix, as described above. Finally, a pointer to the first character past the converted subject string is stored in **ptr**, if **ptr** is not **NULL**.

If the subject string is empty (that is, if *s* does not start with a substring in acceptable form), no conversion is performed and the value of **s** is stored in **ptr** (if **ptr** is not **NULL**).

The alternate function **_strtoull_r** is a reentrant version. The extra argument **reent** is a pointer to a reentrancy structure.

**Returns**

**strtoull** returns the converted value, if any. If no conversion was made, 0 is returned. **strtoull** returns ULONG_LONG_MAX if the magnitude of the converted value is too large, and sets **errno** to ERANGE.

**Portability**

**strtoull** is ANSI.
\texttt{strtoull} requires no supporting OS subroutines.
2.37 wcsrtombs, wcnsrtombs—convert a wide-character string to a character string

Synopsis

```c
#include <wchar.h>
size_t wcsrtombs(char *dst, const wchar_t **src, size_t len, mbstate_t *ps);

#include <wchar.h>
size_t _wcsrtombs_r(struct _reent *ptr, char *dst, const wchar_t **src, size_t len, mbstate_t *ps);

#include <wchar.h>
size_t wcnsrtombs(char *dst, const wchar_t **src, size_t nwc, size_t len, mbstate_t *ps);

#include <wchar.h>
size_t _wcnsrtombs_r(struct _reent *ptr, char *dst, const wchar_t **src, size_t nwc, size_t len, mbstate_t *ps);
```

Description

The `wcsrtombs` function converts a string of wide characters indirectly pointed to by `src` to a corresponding multibyte character string stored in the array pointed to by `dst`. No more than `len` bytes are written to `dst`. If `dst` is NULL, no characters are stored.

If `dst` is not NULL, the pointer pointed to by `src` is updated to point to the character after the one that conversion stopped at. If conversion stops because a null character is encountered, *src is set to NULL.

The `mbstate_t` argument, `ps`, is used to keep track of the shift state. If it is NULL, `wcsrtombs` uses an internal, static `mbstate_t` object, which is initialized to the initial conversion state at program startup.

The `wcnsrtombs` function behaves identically to `wcsrtombs`, except that conversion stops after reading at most `nwc` characters from the buffer pointed to by `src`.

Returns

The `wcsrtombs` and `wcnsrtombs` functions return the number of bytes stored in the array pointed to by `dst` (not including any terminating null), if successful, otherwise it returns `(size_t)-1`.

Portability

`wcsrtombs` is defined by C99 standard. `wcnsrtombs` is defined by the POSIX.1-2008 standard.
2.38 wcstod, wcstof—wide char string to double or float

Synopsis

```c
#include <stdlib.h>

double wcstod(const wchar_t *str, wchar_t **tail);
float wcstof(const wchar_t *str, wchar_t **tail);

double _wcstod_r(void *reent,
    const wchar_t *str, wchar_t **tail);
float _wcstof_r(void *reent,
    const wchar_t *str, wchar_t **tail);
```

Description

The function `wcstod` parses the wide character string `str`, producing a substring which can be converted to a double value. The substring converted is the longest initial subsequence of `str`, beginning with the first non-whitespace character, that has one of these formats:

- `[+|-]digits[.[digits]][(e|E)[+|-]digits]`
- `[+|-].digits[.(e|E)[+|-]digits]`
- `[+|-](i|I)(n|N)(f|F)\[(i|I)(n|N)(i|I)(t|T)(y|Y)]`
- `[+|-](n|N)(a|A)(n|N)\[<&>[hexdigits]<]`
- `[+|-]0(x|X)\[.\[hexdigits]\][(p|P)[+|-]digits]`
- `[+|-]0(x|X).\[hexdigits\][(p|P)[+|-]digits]`

The substring contains no characters if `str` is empty, consists entirely of whitespace, or if the first non-whitespace character is something other than +, -, ., or a digit, and cannot be parsed as infinity or NaN. If the platform does not support NaN, then NaN is treated as an empty substring. If the substring is empty, no conversion is done, and the value of `str` is stored in `*tail`. Otherwise, the substring is converted, and a pointer to the final string (which will contain at least the terminating null character of `str`) is stored in `*tail`. If you want no assignment to `*tail`, pass a null pointer as `tail`. `wcstof` is identical to `wcstod` except for its return type.

This implementation returns the nearest machine number to the input decimal string. Ties are broken by using the IEEE round-even rule. However, `wcstof` is currently subject to double rounding errors.

The alternate functions `_wcstod_r` and `_wcstof_r` are reentrant versions of `wcstod` and `wcstof`, respectively. The extra argument `reent` is a pointer to a reentrancy structure.

Returns

Return the converted substring value, if any. If no conversion could be performed, 0 is returned. If the correct value is out of the range of representable values, plus or minus `HUGE_VAL` is returned, and `ERANGE` is stored in `errno`. If the correct value would cause underflow, 0 is returned and `ERANGE` is stored in `errno`.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`. 
2.39 wcstol—wide string to long

Synopsis

```
#include <wchar.h>
long wcstol(const wchar_t *s, wchar_t **ptr, int base);

long _wcstol_r(void *reent,
               const wchar_t *s, wchar_t **ptr, int base);
```

Description

The function `wcstol` converts the wide string `s` to a `long`. First, it breaks down the string into three parts: leading whitespace, which is ignored; a subject string consisting of characters resembling an integer in the radix specified by `base`; and a trailing portion consisting of zero or more unparsable characters, and always including the terminating null character. Then, it attempts to convert the subject string into a `long` and returns the result.

If the value of `base` is 0, the subject string is expected to look like a normal C integer constant: an optional sign, a possible ‘0x’ indicating a hexadecimal base, and a number. If `base` is between 2 and 36, the expected form of the subject is a sequence of letters and digits representing an integer in the radix specified by `base`, with an optional plus or minus sign. The letters `a`–`z` (or, equivalently, `A`–`Z`) are used to signify values from 10 to 35; only letters whose ascribed values are less than `base` are permitted. If `base` is 16, a leading `0x` is permitted.

The subject sequence is the longest initial sequence of the input string that has the expected form, starting with the first non-whitespace character. If the string is empty or consists entirely of whitespace, or if the first non-whitespace character is not a permissible letter or digit, the subject string is empty.

If the subject string is acceptable, and the value of `base` is zero, `wcstol` attempts to determine the radix from the input string. A string with a leading `0x` is treated as a hexadecimal value; a string with a leading 0 and no `x` is treated as octal; all other strings are treated as decimal. If `base` is between 2 and 36, it is used as the conversion radix, as described above. If the subject string begins with a minus sign, the value is negated. Finally, a pointer to the first character past the converted subject string is stored in `ptr`, if `ptr` is not `NULL`.

If the subject string is empty (or not in acceptable form), no conversion is performed and the value of `s` is stored in `ptr` (if `ptr` is not `NULL`).

The alternate function `_wcstol_r` is a reentrant version. The extra argument `reent` is a pointer to a reentrancy structure.

Returns

`wcstol` returns the converted value, if any. If no conversion was made, 0 is returned.

`wcstol` returns `LONG_MAX` or `LONG_MIN` if the magnitude of the converted value is too large, and sets `errno` to `ERANGE`.

Portability

`wcstol` is ANSI.
No supporting OS subroutines are required.
2.40 wcstoll—wide string to long long

Synopsis

#include <wchar.h>
long long wcstoll(const wchar_t *s, wchar_t **ptr, int base);

long long _wcstoll_r(void *reent,
const wchar_t *s, wchar_t **ptr, int base);

Description

The function wcstoll converts the wide string *s to a long long. First, it breaks down
the string into three parts: leading whitespace, which is ignored; a subject string consisting
of characters resembling an integer in the radix specified by base; and a trailing portion
consisting of zero or more unparsable characters, and always including the terminating null
character. Then, it attempts to convert the subject string into a long long and returns the
result.

If the value of base is 0, the subject string is expected to look like a normal C integer
constant: an optional sign, a possible ‘0x’ indicating a hexadecimal base, and a number.
If base is between 2 and 36, the expected form of the subject is a sequence of letters and
digits representing an integer in the radix specified by base, with an optional plus or minus
sign. The letters a–z (or, equivalently, A–Z) are used to signify values from 10 to 35; only
letters whose ascribed values are less than base are permitted. If base is 16, a leading 0x is
permitted.

The subject sequence is the longest initial sequence of the input string that has the expected
form, starting with the first non-whitespace character. If the string is empty or consists
entirely of whitespace, or if the first non-whitespace character is not a permissible letter or
digit, the subject string is empty.

If the subject string is acceptable, and the value of base is zero, wcstoll attempts to
determine the radix from the input string. A string with a leading 0x is treated as a
hexadecimal value; a string with a leading 0 and no x is treated as octal; all other strings
are treated as decimal. If base is between 2 and 36, it is used as the conversion radix,
as described above. If the subject string begins with a minus sign, the value is negated.
Finally, a pointer to the first character past the converted subject string is stored in ptr, if
ptr is not NULL.

If the subject string is empty (or not in acceptable form), no conversion is performed and
the value of s is stored in ptr (if ptr is not NULL).

The alternate function _wcstoll_r is a reentrant version. The extra argument reent is a
pointer to a reentrancy structure.

Returns

wcstoll returns the converted value, if any. If no conversion was made, 0 is returned.
wcstoll returns LONG_LONG_MAX or LONG_LONG_MIN if the magnitude of the converted value
is too large, and sets errno to ERANGE.

Portability

wcstoll is ANSI.
No supporting OS subroutines are required.
2.41 **wcstoul**—wide string to unsigned long

**Synopsis**

```c
#include <wchar.h>

unsigned long wcstoul(const wchar_t *s, wchar_t **ptr, int base);

unsigned long _wcstoul_r(void *reent, const wchar_t *s, wchar_t **ptr, int base);
```

**Description**

The function `wcstoul` converts the wide string `*s` to an **unsigned long**. First, it breaks down the string into three parts: leading whitespace, which is ignored; a subject string consisting of the digits meaningful in the radix specified by `base` (for example, 0 through 7 if the value of `base` is 8); and a trailing portion consisting of one or more unparsable characters, which always includes the terminating null character. Then, it attempts to convert the subject string into an unsigned long integer, and returns the result.

If the value of `base` is zero, the subject string is expected to look like a normal C integer constant (save that no optional sign is permitted): a possible 0x indicating hexadecimal radix, and a number. If `base` is between 2 and 36, the expected form of the subject is a sequence of digits (which may include letters, depending on the base) representing an integer in the radix specified by `base`. The letters a–z (or A–Z) are used as digits valued from 10 to 35. If `base` is 16, a leading 0x is permitted.

The subject sequence is the longest initial sequence of the input string that has the expected form, starting with the first non-whitespace character. If the string is empty or consists entirely of whitespace, or if the first non-whitespace character is not a permissible digit, the subject string is empty.

If the subject string is acceptable, and the value of `base` is zero, `wcstoul` attempts to determine the radix from the input string. A string with a leading 0x is treated as a hexadecimal value; a string with a leading 0 and no x is treated as octal; all other strings are treated as decimal. If `base` is between 2 and 36, it is used as the conversion radix, as described above. Finally, a pointer to the first character past the converted subject string is stored in `ptr`, if `ptr` is not `NULL`.

If the subject string is empty (that is, if `*s` does not start with a substring in acceptable form), no conversion is performed and the value of `s` is stored in `ptr` (if `ptr` is not `NULL`). The alternate function `_wcstoul_r` is a reentrant version. The extra argument `reent` is a pointer to a reentrancy structure.

**Returns**

`wcstoul` returns the converted value, if any. If no conversion was made, 0 is returned.

`wcstoul` returns `ULONG_MAX` if the magnitude of the converted value is too large, and sets `errno` to `ERANGE`.

**Portability**

`wcstoul` is ANSI.
`wcstoul` requires no supporting OS subroutines.
2.42 wcstoull—wide string to unsigned long long

Synopsis

```c
#include <wchar.h>

unsigned long long wcstoull(const wchar_t *s, wchar_t **ptr, int base);

unsigned long long _wcstoull_r(void *reent, const wchar_t *s, wchar_t **ptr, int base);
```

Description

The function `wcstoull` converts the wide string `*s` to an `unsigned long long`. First, it breaks down the string into three parts: leading whitespace, which is ignored; a subject string consisting of the digits meaningful in the radix specified by `base` (for example, 0 through 7 if the value of `base` is 8); and a trailing portion consisting of one or more unparsable characters, which always includes the terminating null character. Then, it attempts to convert the subject string into an unsigned long long integer, and returns the result.

If the value of `base` is zero, the subject string is expected to look like a normal C integer constant: an optional sign (+ or -), a possible 0x indicating hexadecimal radix or a possible <0> indicating octal radix, and a number. If `base` is between 2 and 36, the expected form of the subject is a sequence of digits (which may include letters, depending on the base) representing an integer in the radix specified by `base`. The letters a-z (or A-Z) are used as digits valued from 10 to 35. If `base` is 16, a leading 0x is permitted.

The subject sequence is the longest initial sequence of the input string that has the expected form, starting with the first non-whitespace character. If the string is empty or consists entirely of whitespace, or if the first non-whitespace character is not a permissible digit, the subject string is empty.

If the subject string is acceptable, and the value of `base` is zero, `wcstoull` attempts to determine the radix from the input string. A string with a leading 0x is treated as a hexadecimal value; a string with a leading 0 and no x is treated as octal; all other strings are treated as decimal. If `base` is between 2 and 36, it is used as the conversion radix, as described above. Finally, a pointer to the first character past the converted subject string is stored in `ptr`, if `ptr` is not NULL.

If the subject string is empty (that is, if `*s` does not start with a substring in acceptable form), no conversion is performed and the value of `s` is stored in `ptr` (if `ptr` is not NULL).

The alternate function `_wcstoull_r` is a reentrant version. The extra argument `reent` is a pointer to a reentrancy structure.

Returns

`wcstoull` returns 0 and sets `errno` to EINVAL if the value of `base` is not supported.

`wcstoull` returns the converted value, if any. If no conversion was made, 0 is returned.

`wcstoull` returns ULLONG_MAX if the magnitude of the converted value is too large, and sets `errno` to ERANGE.
Portability
wcstoull is ANSI.
wcestoull requires no supporting OS subroutines.
2.43 system—execute command string

Synopsis

```
#include <stdlib.h>
int system(char *s);

int _system_r(void *reent, char *s);
```

Description

Use **system** to pass a command string *s* to /bin/sh on your system, and wait for it to finish executing.

Use “**system(NULL)**” to test whether your system has /bin/sh available.

The alternate function **_system_r** is a reentrant version. The extra argument **reent** is a pointer to a reentrancy structure.

Returns

**system(NULL)** returns a non-zero value if /bin/sh is available, and 0 if it is not.

With a command argument, the result of **system** is the exit status returned by /bin/sh.

Portability

ANSI C requires **system**, but leaves the nature and effects of a command processor undefined. ANSI C does, however, specify that **system(NULL)** return zero or nonzero to report on the existence of a command processor.

POSIX.2 requires **system**, and requires that it invoke a **sh**. Where **sh** is found is left unspecified.

Supporting OS subroutines required: **_exit, _execve, _fork_r, _wait_r**.
2.44 wcstombs—minimal wide char string to multibyte string converter

Synopsis

```c
#include <stdlib.h>
int wcstombs(const char *s, wchar_t *pwc, size_t n);
```

Description

When `_MB_CAPABLE` is not defined, this is a minimal ANSI-conforming implementation of `wcstombs`. In this case, all wide-characters are expected to represent single bytes and so are converted simply by casting to char.

When `_MB_CAPABLE` is defined, this routine calls `_wcstombs_r` to perform the conversion, passing a state variable to allow state dependent decoding. The result is based on the locale setting which may be restricted to a defined set of locales.

Returns

This implementation of `wcstombs` returns 0 if `s` is NULL or is the empty string; it returns -1 if `_MB_CAPABLE` and one of the wide-char characters does not represent a valid multi-byte character; otherwise it returns the minimum of: `n` or the number of bytes that are transferred to `s`, not including the nul terminator.

If the return value is -1, the state of the `pwc` string is indeterminate. If the input has a length of 0, the output string will be modified to contain a wchar_t nul terminator if `n > 0`.

Portability

`wcstombs` is required in the ANSI C standard. However, the precise effects vary with the locale.

`wcstombs` requires no supporting OS subroutines.
2.45 wctomb—minimal wide char to multibyte converter

Synopsis

```c
#include <stdlib.h>
int wctomb(char *s, wchar_t wchar);
```

Description

When `MB_CAPABLE` is not defined, this is a minimal ANSI-conforming implementation of `wctomb`. The only “wide characters” recognized are single bytes, and they are “converted” to themselves.

When `MB_CAPABLE` is defined, this routine calls `_wctomb_r` to perform the conversion, passing a state variable to allow state dependent decoding. The result is based on the locale setting which may be restricted to a defined set of locales.

Each call to `wctomb` modifies `*s` unless `s` is a null pointer or `MB_CAPABLE` is defined and `wchar` is invalid.

Returns

This implementation of `wctomb` returns 0 if `s` is `NULL`; it returns -1 if `MB_CAPABLE` is enabled and the `wchar` is not a valid multi-byte character, it returns 1 if `MB_CAPABLE` is not defined or the `wchar` is in reality a single byte character, otherwise it returns the number of bytes in the multi-byte character.

Portability

`wctomb` is required in the ANSI C standard. However, the precise effects vary with the locale.

`wctomb` requires no supporting OS subroutines.
3 Character Type Macros and Functions
('ctype.h')

This chapter groups macros (which are also available as subroutines) to classify characters into several categories (alphabetic, numeric, control characters, whitespace, and so on), or to perform simple character mappings.

The header file 'ctype.h' defines the macros.
3.1 isalnum—alphanumeric character predicate

Synopsis

```c
#include <ctype.h>
int isalnum(int c);
```

Description

isalnum is a macro which classifies ASCII integer values by table lookup. It is a predicate returning non-zero for alphabetic or numeric ASCII characters, and 0 for other arguments. It is defined for all integer values.

You can use a compiled subroutine instead of the macro definition by undefining the macro using `#undef isalnum`.

Returns

isalnum returns non-zero if `c` is a letter (a–z or A–Z) or a digit (0–9).

Portability

isalnum is ANSI C.

No OS subroutines are required.
3.2 isalpha—alphabetic character predicate

Synopsis

```c
#include <ctype.h>
int isalpha(int c);
```

Description

isalpha is a macro which classifies ASCII integer values by table lookup. It is a predicate returning non-zero when `c` represents an alphabetic ASCII character, and 0 otherwise. It is defined only when `isascii(c)` is true or `c` is EOF.

You can use a compiled subroutine instead of the macro definition by undefining the macro using `#undef isalpha`.

Returns

isalpha returns non-zero if `c` is a letter (A–Z or a–z).

Portability

isalpha is ANSI C.

No supporting OS subroutines are required.
3.3 isascii—ASCII character predicate

Synopsis

```c
#include <ctype.h>
int isascii(int c);
```

Description

isascii is a macro which returns non-zero when \( c \) is an ASCII character, and 0 otherwise. It is defined for all integer values.

You can use a compiled subroutine instead of the macro definition by undefining the macro using `#undef isascii`.

Returns

isascii returns non-zero if the low order byte of \( c \) is in the range 0 to 127 (0x00–0x7F).

Portability

isascii is ANSI C.

No supporting OS subroutines are required.
3.4 iscntrl—control character predicate

Synopsis

```c
#include <ctype.h>
int iscntrl(int c);
```

Description

iscntrl is a macro which classifies ASCII integer values by table lookup. It is a predicate returning non-zero for control characters, and 0 for other characters. It is defined only when isascii(c) is true or c is EOF.

You can use a compiled subroutine instead of the macro definition by undefining the macro using `#undef iscntrl`.

Returns

iscntrl returns non-zero if c is a delete character or ordinary control character (0x7F or 0x00–0x1F).

Portability

iscntrl is ANSI C.

No supporting OS subroutines are required.
3.5 isdigit—decimal digit predicate

Synopsis

```c
#include <ctype.h>
int isdigit(int c);
```

Description

isdigit is a macro which classifies ASCII integer values by table lookup. It is a predicate returning non-zero for decimal digits, and 0 for other characters. It is defined only when isascii(c) is true or c is EOF.

You can use a compiled subroutine instead of the macro definition by undefining the macro using `#undef isdigit`.

Returns

isdigit returns non-zero if c is a decimal digit (0–9).

Portability

isdigit is ANSI C.

No supporting OS subroutines are required.
Chapter 3: Character Type Macros and Functions (`ctype.h`) 65

3.6 `islower`—lowercase character predicate

Synopsis

```c
#include <ctype.h>
int islower(int c);
```

Description

`islower` is a macro which classifies ASCII integer values by table lookup. It is a predicate returning non-zero for minuscules (lowercase alphabetic characters), and 0 for other characters. It is defined only when `isascii(c)` is true or `c` is EOF.

You can use a compiled subroutine instead of the macro definition by undefining the macro using `#undef islower`.

Returns

`islower` returns non-zero if `c` is a lowercase letter (a–z).

Portability

`islower` is ANSI C.

No supporting OS subroutines are required.
3.7 isprint, isgraph—printable character predicates

Synopsis

```c
#include <ctype.h>
int isprint(int c);
int isgraph(int c);
```

Description

isprint is a macro which classifies ASCII integer values by table lookup. It is a predicate returning non-zero for printable characters, and 0 for other character arguments. It is defined only when isascii(c) is true or c is EOF.

You can use a compiled subroutine instead of the macro definition by undefining either macro using ‘#undef isprint’ or ‘#undef isgraph’.

Returns

isprint returns non-zero if c is a printing character, (0x20–0x7E). isgraph behaves identically to isprint, except that the space character (0x20) is excluded.

Portability

isprint and isgraph are ANSI C.

No supporting OS subroutines are required.
3.8 ispunct—punctuation character predicate

Synopsis

```c
#include <ctype.h>
int ispunct(int c);
```

Description

`ispunct` is a macro which classifies ASCII integer values by table lookup. It is a predicate returning non-zero for printable punctuation characters, and 0 for other characters. It is defined only when `isascii(c)` is true or `c` is EOF.

You can use a compiled subroutine instead of the macro definition by undefining the macro using `#undef ispunct`.

Returns

`ispunct` returns non-zero if `c` is a printable punctuation character (`isgraph(c) && !isalnum(c)`).

Portability

`ispunct` is ANSI C.

No supporting OS subroutines are required.
3.9 isspace—whitespace character predicate

Synopsis

```c
#include <ctype.h>
int isspace(int c);
```

Description

`isspace` is a macro which classifies ASCII integer values by table lookup. It is a predicate returning non-zero for whitespace characters, and 0 for other characters. It is defined only when `isascii(c)` is true or `c` is EOF.

You can use a compiled subroutine instead of the macro definition by undefining the macro using `#undef isspace`.

Returns

`isspace` returns non-zero if `c` is a space, tab, carriage return, new line, vertical tab, or formfeed (0x09–0x0D, 0x20).

Portability

`isspace` is ANSI C.

No supporting OS subroutines are required.
3.10 **isupper—uppercase character predicate**

**Synopsis**
```c
#include <ctype.h>
int isupper(int c);
```

**Description**
isupper is a macro which classifies ASCII integer values by table lookup. It is a predicate returning non-zero for uppercase letters (A–Z), and 0 for other characters. It is defined only when isascii(c) is true or c is EOF.

You can use a compiled subroutine instead of the macro definition by undefining the macro using `#undef isupper`.

**Returns**
isupper returns non-zero if c is a uppercase letter (A–Z).

**Portability**
isupper is ANSI C.

No supporting OS subroutines are required.
3.11 isxdigit—hexadecimal digit predicate

Synopsis

```c
#include <ctype.h>
int isxdigit(int c);
```

Description

isxdigit is a macro which classifies ASCII integer values by table lookup. It is a predicate returning non-zero for hexadecimal digits, and 0 for other characters. It is defined only when isascii(c) is true or c is EOF.

You can use a compiled subroutine instead of the macro definition by undefining the macro using `#undef isxdigit`.

Returns

isxdigit returns non-zero if c is a hexadecimal digit (0–9, a–f, or A–F).

Portability

isxdigit is ANSI C.

No supporting OS subroutines are required.
3.12 toascii—force integers to ASCII range

Synopsis

#include <ctype.h>
int toascii(int c);

Description
toascii is a macro which coerces integers to the ASCII range (0–127) by zeroing any higher-order bits.
You can use a compiled subroutine instead of the macro definition by undefining this macro using ‘#undef toascii’.

Returns
toascii returns integers between 0 and 127.

Portability
toascii is not ANSI C.
No supporting OS subroutines are required.
3.13 `tolower`—translate characters to lowercase

Synopsis

```c
#include <ctype.h>
int tolower(int c);
int _tolower(int c);
```

Description

tolower is a macro which converts uppercase characters to lowercase, leaving all other characters unchanged. It is only defined when `c` is an integer in the range `EOF` to 255.

You can use a compiled subroutine instead of the macro definition by undefining this macro using `#undef tolower`.

tolua performs the same conversion as tolower, but should only be used when `c` is known to be an uppercase character (A–Z).

Returns

tolower returns the lowercase equivalent of `c` when it is a character between A and Z, and `c` otherwise.

tolower returns the lowercase equivalent of `c` when it is a character between A and Z. If `c` is not one of these characters, the behaviour of _tolower is undefined.

Portability

tolower is ANSI C. _tolower is not recommended for portable programs.

No supporting OS subroutines are required.
3.14 toupper—translate characters to uppercase

Synopsis

```c
#include <ctype.h>
int toupper(int c);
int _toupper(int c);
```

Description
toupper is a macro which converts lowercase characters to uppercase, leaving all other characters unchanged. It is only defined when \( c \) is an integer in the range EOF to 255.

You can use a compiled subroutine instead of the macro definition by undefining this macro using `#undef toupper`.

_toupper performs the same conversion as toupper, but should only be used when \( c \) is known to be a lowercase character (a–z).

Returns
toupper returns the uppercase equivalent of \( c \) when it is a character between a and z, and \( c \) otherwise.

_toupper returns the uppercase equivalent of \( c \) when it is a character between a and z. If \( c \) is not one of these characters, the behaviour of _toupper is undefined.

Portability
toupper is ANSI C. _toupper is not recommended for portable programs.

No supporting OS subroutines are required.
3.15 iswalnum—alphanumeric wide character test

Synopsis

```
#include <wctype.h>
int iswalnum(wint_t c);
```

Description

iswalnum is a function which classifies wide-character values that are alphanumeric.

Returns

iswalnum returns non-zero if c is a alphanumeric wide character.

Portability

iswalnum is C99.

No supporting OS subroutines are required.
3.16 iswalpha—alphabetic wide character test

Synopsis

    #include <wctype.h>
    int iswalpha(wint_t c);

Description

iswalpha is a function which classifies wide-character values that are alphabetic.

Returns

iswalpha returns non-zero if c is an alphabetic wide character.

Portability

iswalpha is C99.

No supporting OS subroutines are required.
3.17 iswcntrl—control wide character test

Synopsis

```c
#include <wctype.h>
int iswcntrl(wint_t c);
```

Description

`iswcntrl` is a function which classifies wide-character values that are categorized as control characters.

Returns

`iswcntrl` returns non-zero if `c` is a control wide character.

Portability

`iswcntrl` is C99.

No supporting OS subroutines are required.
3.18 iswblank—blank wide character test

Synopsis

```c
#include <wctype.h>
int iswblank(wint_t c);
```

Description

iswblank is a function which classifies wide-character values that are categorized as blank.

Returns

iswblank returns non-zero if c is a blank wide character.

Portability

iswblank is C99.

No supporting OS subroutines are required.
3.19 iswdigit—decimal digit wide character test

Synopsis

```c
#include <wctype.h>
int iswdigit(wint_t c);
```

Description

iswdigit is a function which classifies wide-character values that are decimal digits.

Returns

iswdigit returns non-zero if c is a decimal digit wide character.

Portability

iswdigit is C99.

No supporting OS subroutines are required.
3.20 iswgraph—graphic wide character test

Synopsis

#include <wctype.h>
int iswgraph(wint_t c);

Description

iswgraph is a function which classifies wide-character values that are graphic.

Returns

iswgraph returns non-zero if c is a graphic wide character.

Portability

iswgraph is C99.
No supporting OS subroutines are required.
3.21 iswlower—lowercase wide character test

Synopsis

```
#include <wctype.h>
int iswlower(wint_t c);
```

Description

iswlower is a function which classifies wide-character values that have uppercase translations.

Returns

iswlower returns non-zero if \( c \) is a lowercase wide character.

Portability

iswlower is C99.

No supporting OS subroutines are required.
3.22 iswprint—printable wide character test

Synopsis

```c
#include <wctype.h>
int iswprint(wint_t c);
```

Description

`iswprint` is a function which classifies wide-character values that are printable.

Returns

`iswprint` returns non-zero if `c` is a printable wide character.

Portability

`iswprint` is C99.

No supporting OS subroutines are required.
3.23 iswpunct—punctuation wide character test

Synopsis

```c
#include <wctype.h>
int iswpunct(wint_t c);
```

Description

iswpunct is a function which classifies wide-character values that are punctuation.

Returns

iswpunct returns non-zero if c is a punctuation wide character.

Portability

iswpunct is C99.

No supporting OS subroutines are required.
3.24 iswspace—whitespace wide character test

Synopsis

```c
#include <wctype.h>
int iswspace(wint_t c);
```

Description

iswspace is a function which classifies wide-character values that are categorized as whitespace.

Returns

iswspace returns non-zero if c is a whitespace wide character.

Portability

iswspace is C99.

No supporting OS subroutines are required.
3.25 iswupper—uppercase wide character test

Synopsis

```c
#include <wctype.h>
int iswupper(wint_t c);
```

Description

iswupper is a function which classifies wide-character values that have uppercase translations.

Returns

iswupper returns non-zero if \( c \) is a uppercase wide character.

Portability

iswupper is C99.

No supporting OS subroutines are required.
3.26 iswxdigit—hexadecimal digit wide character test

Synopsis

    #include <wctype.h>
    int iswxdigit(wint_t c);

Description

iswxdigit is a function which classifies wide character values that are hexadecimal digits.

Returns

iswxdigit returns non-zero if c is a hexadecimal digit wide character.

Portability

iswxdigit is C99.

No supporting OS subroutines are required.
3.27 iswctype—extensible wide-character test

Synopsis

#include <wctype.h>
int iswctype(wint_t c, wctype_t desc);

Description

iswctype is a function which classifies wide-character values using the wide-character test specified by desc.

Returns

iswctype returns non-zero if and only if c matches the test specified by desc. If desc is unknown, zero is returned.

Portability

iswctype is C99.

No supporting OS subroutines are required.
3.28 wctype—get wide-character classification type

Synopsis

#include <wctype.h>

wctype_t wctype(const char *c);

Description

wctype is a function which takes a string c and gives back the appropriate wctype_t type value associated with the string, if one exists. The following values are guaranteed to be recognized: "alnum", "alpha", "blank", "cntrl", "digit", "graph", "lower", "print", "punct", "space", "upper", and "xdigit".

Returns

wctype returns 0 and sets errno to EINVAL if the given name is invalid. Otherwise, it returns a valid non-zero wctype_t value.

Portability

wctype is C99.

No supporting OS subroutines are required.
3.29 towlower—translate wide characters to lowercase

Synopsis
#include <wctype.h>

wint_t towlower(wint_t c);

Description
towlower is a function which converts uppercase wide characters to lowercase, leaving all
other characters unchanged.

Returns
towlower returns the lowercase equivalent of c when it is a uppercase wide character;
otherwise, it returns the input character.

Portability
towlower is C99.
No supporting OS subroutines are required.
3.30 `towupper`—translate wide characters to uppercase

Synopsis
```
#include <wctype.h>
wint_t towupper(wint_t c);
```

Description
`towupper` is a function which converts lowercase wide characters to uppercase, leaving all other characters unchanged.

Returns
`towupper` returns the uppercase equivalent of `c` when it is a lowercase wide character, otherwise, it returns the input character.

Portability
`towupper` is C99.

No supporting OS subroutines are required.
3.31 towctrans—extensible wide-character translation

Synopsis

```c
#include <wctype.h>

wint_t towctrans(wint_t c, wctrans_t w);
```

Description
towctrans is a function which converts wide characters based on a specified translation type w. If the translation type is invalid or cannot be applied to the current character, no change to the character is made.

Returns
towctrans returns the translated equivalent of c when it is a valid for the given translation, otherwise, it returns the input character. When the translation type is invalid, `errno` is set `EINVAL`.

Portability
towctrans is C99.

No supporting OS subroutines are required.
3.32 wctrans—get wide-character translation type

Synopsis

```c
#include <wctype.h>
wctrans_t wctrans(const char *c);
```

Description

`wctrans` is a function which takes a string `c` and gives back the appropriate `wctrans_t` type value associated with the string, if one exists. The following values are guaranteed to be recognized: "tolower" and "toupper".

Returns

`wctrans` returns 0 and sets `errno` to `EINVAL` if the given name is invalid. Otherwise, it returns a valid non-zero `wctrans_t` value.

Portability

`wctrans` is C99.

No supporting OS subroutines are required.
4 Input and Output (‘stdio.h’)

This chapter comprises functions to manage files or other input/output streams. Among these functions are subroutines to generate or scan strings according to specifications from a format string.

The underlying facilities for input and output depend on the host system, but these functions provide a uniform interface.

The corresponding declarations are in ‘stdio.h’.

The reentrant versions of these functions use macros

```c
_stdin_r(reent)
_stdout_r(reent)
_stderr_r(reent)
```

instead of the globals stdin, stdout, and stderr. The argument `<reent>` is a pointer to a reentrancy structure.
4.1 clearerr—clear file or stream error indicator

Synopsis

```c
#include <stdio.h>
void clearerr(FILE *fp);
```

Description
The stdio functions maintain an error indicator with each file pointer fp, to record whether any read or write errors have occurred on the associated file or stream. Similarly, it maintains an end-of-file indicator to record whether there is no more data in the file.

Use clearerr to reset both of these indicators.

See ferror andfeof to query the two indicators.

Returns

clearerr does not return a result.

Portability

ANSI C requires clearerr.

No supporting OS subroutines are required.
4.2 `diprintf`, `vdiprintf`—print to a file descriptor (integer only)

**Synopsis**

```c
#include <stdio.h>
#include <stdarg.h>

int diprintf(int fd, const char *format, ...);
int vdiprintf(int fd, const char *format, va_list ap);
int _diprintf_r(struct _reent *ptr, int fd, const char *format, ...);
int _vdiprintf_r(struct _reent *ptr, int fd, const char *format, va_list ap);
```

**Description**

diprintf and vdiprintf are similar to dprintf and vdprintf, except that only integer format specifiers are processed.

The functions _diprintf_r and _vdiprintf_r are simply reentrant versions of the functions above.

**Returns**

Similar to dprintf and vdprintf.

**Portability**

This set of functions is an integer-only extension, and is not portable.

Supporting OS subroutines required: sbrk, write.
4.3 dprintf, vdprintf—print to a file descriptor

Synopsis

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

int dprintf(int fd, const char *format, ...);
int vdprintf(int fd, const char *format, va_list ap);
int _dprintf_r(struct __reent *ptr, int fd,
               const char *format, ...);
int _vdprintf_r(struct __reent *ptr, int fd,
               const char *format, va_list ap);
```

Description

dprintf and vdprintf allow printing a format, similarly to printf, but write to a file descriptor instead of to a FILE stream.

The functions _dprintf_r and _vdprintf_r are simply reentrant versions of the functions above.

Returns

The return value and errors are exactly as for write, except that errno may also be set to ENOMEM if the heap is exhausted.

Portability

This function is originally a GNU extension in glibc and is not portable.

Supporting OS subroutines required: sbrk, write.
4.4 fclose—close a file

Synopsis

```c
#include <stdio.h>
int fclose(FILE *fp);
int _fclose_r(struct _reent *reent, FILE *fp);
```

Description

If the file or stream identified by `fp` is open, `fclose` closes it, after first ensuring that any pending data is written (by calling `fflush(fp)`). The alternate function `_fclose_r` is a reentrant version. The extra argument `reent` is a pointer to a reentrancy structure.

Returns

`fclose` returns 0 if successful (including when `fp` is NULL or not an open file); otherwise, it returns EOF.

Portability

`fclose` is required by ANSI C.
Required OS subroutines: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.
4.5 fcloseall—close all files

Synopsis

```c
#include <stdio.h>
int fcloseall(void);
int _fcloseall_r (struct _reent *ptr);
```

Description
fcloseall closes all files in the current reentrancy struct’s domain. The function
_fcloseall_r is the same function, except the reentrancy struct is passed in as the `ptr`
argument.

This function is not recommended as it closes all streams, including the std streams.

Returns
fclose returns 0 if all closes are successful. Otherwise, EOF is returned.

Portability
fcloseall is a glibc extension.

Required OS subroutines: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.
4.6 fdopen—turn open file into a stream

Synopsis

```c
#include <stdio.h>
FILE *fdopen(int fd, const char *mode);
FILE *fdopen_r(struct _reent *reent, int fd, const char *mode);
```

Description

`fdopen` produces a file descriptor of type `FILE *`, from a descriptor for an already-open file (returned, for example, by the system subroutine `open` rather than by `fopen`). The `mode` argument has the same meanings as in `fopen`.

Returns

File pointer or NULL, as for `fopen`.

Portability

`fdopen` is ANSI.
4.7 **feof—test for end of file**

**Synopsis**

```c
#include <stdio.h>
int feof(FILE *fp);
```

**Description**

`feof` tests whether or not the end of the file identified by `fp` has been reached.

**Returns**

`feof` returns 0 if the end of file has not yet been reached; if at end of file, the result is nonzero.

**Portability**

`feof` is required by ANSI C.

No supporting OS subroutines are required.
4.8 ferror—test whether read/write error has occurred

Synopsis

```c
#include <stdio.h>
int ferror(FILE *fp);
```

Description

The `stdio` functions maintain an error indicator with each file pointer `fp`, to record whether any read or write errors have occurred on the associated file or stream. Use `ferror` to query this indicator.

See `clearerr` to reset the error indicator.

Returns

`ferror` returns 0 if no errors have occurred; it returns a nonzero value otherwise.

Portability

ANSI C requires `ferror`.

No supporting OS subroutines are required.
4.9 fflush—flush buffered file output

Synopsis

```c
#include <stdio.h>
int fflush(FILE *fp);
int _fflush_r(struct _reent *reent, FILE *fp);
```

Description

The `stdio` output functions can buffer output before delivering it to the host system, in order to minimize the overhead of system calls.

Use `fflush` to deliver any such pending output (for the file or stream identified by `fp`) to the host system.

If `fp` is `NULL`, `fflush` delivers pending output from all open files.

Additionally, if `fp` is a seekable input stream visiting a file descriptor, set the position of the file descriptor to match next unread byte, useful for obeying POSIX semantics when ending a process without consuming all input from the stream.

The alternate function `_fflush_r` is a reentrant version, where the extra argument `reent` is a pointer to a reentrancy structure, and `fp` must not be `NULL`.

Returns

`fflush` returns 0 unless it encounters a write error; in that situation, it returns `EOF`.

Portability

ANSI C requires `fflush`. The behavior on input streams is only specified by POSIX, and not all implementations follow POSIX rules.

No supporting OS subroutines are required.
4.10 fgetc—get a character from a file or stream

Synopsis

```c
#include <stdio.h>
int fgetc(FILE *fp);

#include <stdio.h>
int _fgetc_r(struct _reent *ptr, FILE *fp);
```

Description

Use `fgetc` to get the next single character from the file or stream identified by `fp`. As a side effect, `fgetc` advances the file’s current position indicator.

For a macro version of this function, see `getc`.

The function `_fgetc_r` is simply a reentrant version of `fgetc` that is passed the additional reentrant structure pointer argument: `ptr`.

Returns

The next character (read as an `unsigned char`, and cast to `int`), unless there is no more data, or the host system reports a read error; in either of these situations, `fgetc` returns `EOF`.

You can distinguish the two situations that cause an `EOF` result by using the `ferror` and `feof` functions.

Portability

ANSI C requires `fgetc`.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`. 
4.11 fgetpos—record position in a stream or file

Synopsis

```c
#include <stdio.h>
int fgetpos(FILE *fp, fpos_t *pos);
int _fgetpos_r(struct _reent *ptr, FILE *fp, fpos_t *pos);
```

Description

Objects of type FILE can have a “position” that records how much of the file your program has already read. Many of the stdio functions depend on this position, and many change it as a side effect.

You can use `fgetpos` to report on the current position for a file identified by `fp`; `fgetpos` will write a value representing that position at `*pos`. Later, you can use this value with `fsetpos` to return the file to this position.

In the current implementation, `fgetpos` simply uses a character count to represent the file position; this is the same number that would be returned by `ftell`.

Returns

`fgetpos` returns 0 when successful. If `fgetpos` fails, the result is 1. Failure occurs on streams that do not support positioning; the global `errno` indicates this condition with the value `ESPIPE`.

Portability

`fgetpos` is required by the ANSI C standard, but the meaning of the value it records is not specified beyond requiring that it be acceptable as an argument to `fsetpos`. In particular, other conforming C implementations may return a different result from `ftell` than what `fgetpos` writes at `*pos`.

No supporting OS subroutines are required.
4.12 fgets—get character string from a file or stream

Synopsis

```c
#include <stdio.h>
char *fgets(char *buf, int n, FILE *fp);
```

```c
#include <stdio.h>
char *fgets_r(struct _reent *ptr, char *buf, int n, FILE *fp);
```

Description

Reads at most \( n-1 \) characters from \( fp \) until a newline is found. The characters including to
the newline are stored in \( buf \). The buffer is terminated with a 0.
The \_fgets\_r function is simply the reentrant version of \texttt{fgets} and is passed an additional
reentrancy structure pointer: \texttt{ptr}.

Returns

\texttt{fgets} returns the buffer passed to it, with the data filled in. If end of file occurs with some
data already accumulated, the data is returned with no other indication. If no data are
read, NULL is returned instead.

Portability

\texttt{fgets} should replace all uses of \texttt{gets}. Note however that \texttt{fgets} returns all of the data,
while \texttt{gets} removes the trailing newline (with no indication that it has done so.)
Supporting OS subroutines required: \texttt{close}, \texttt{fstat}, \texttt{isatty}, \texttt{lseek}, \texttt{read}, \texttt{sbrk}, \texttt{write}.
4.13 fgetwc, getwc—get a wide character from a file or stream

Synopsis

```c
#include <stdio.h>
#include <wchar.h>
wint_t fgetwc(FILE *fp);

#include <stdio.h>
#include <wchar.h>
wint_t _fgetwc_r(struct _reent *ptr, FILE *fp);

#include <stdio.h>
#include <wchar.h>
wint_t getwc(FILE *fp);

#include <stdio.h>
#include <wchar.h>
wint_t _getwc_r(struct _reent *ptr, FILE *fp);
```

Description

Use `fgetwc` to get the next wide character from the file or stream identified by `fp`. As a side effect, `fgetwc` advances the file’s current position indicator.

The `getwc` function or macro functions identically to `fgetwc`. It may be implemented as a macro, and may evaluate its argument more than once. There is no reason ever to use it.

`_fgetwc_r` and `_getwc_r` are simply reentrant versions of `fgetwc` and `getwc` that are passed the additional reentrant structure pointer argument: `ptr`.

Returns

The next wide character cast to `wint_t`), unless there is no more data, or the host system reports a read error; in either of these situations, `fgetwc` and `getwc` return `WEOF`.

You can distinguish the two situations that cause an EOF result by using the `ferror` and `feof` functions.

Portability

C99, POSIX.1-2001
4.14 `fgetws`—get wide character string from a file or stream

**Synopsis**

```c
#include <wchar.h>
wchar_t *fgetws(wchar_t *ws, int n, FILE *fp);
```

```c
#include <wchar.h>
wchar_t *_fgetws_r(struct _reent *ptr, wchar_t *ws, int n, FILE *fp);
```

**Description**

Reads at most `n-1` wide characters from `fp` until a newline is found. The wide characters including to the newline are stored in `ws`. The buffer is terminated with a 0.

The `_fgetws_r` function is simply the reentrant version of `fgetws` and is passed an additional reentrancy structure pointer: `ptr`.

**Returns**

`fgetws` returns the buffer passed to it, with the data filled in. If end of file occurs with some data already accumulated, the data is returned with no other indication. If no data are read, NULL is returned instead.

**Portability**

C99, POSIX.1-2001
4.15 fileno—return file descriptor associated with stream

Synopsis

```c
#include <stdio.h>
int fileno(FILE *fp);
```

Description

You can use `fileno` to return the file descriptor identified by `fp`.

Returns

`fileno` returns a non-negative integer when successful. If `fp` is not an open stream, `fileno` returns -1.

Portability

`fileno` is not part of ANSI C. POSIX requires `fileno`.

Supporting OS subroutines required: none.
4.16 fmemopen—open a stream around a fixed-length string

Synopsis

```c
#include <stdio.h>
FILE *fmemopen(void *restrict buf, size_t size,
               const char *restrict mode);
```

Description

fmemopen creates a seekable FILE stream that wraps a fixed-length buffer of size bytes starting at buf. The stream is opened with mode treated as in fopen, where append mode starts writing at the first NUL byte. If buf is NULL, then size bytes are automatically provided as if by malloc, with the initial size of 0, and mode must contain + so that data can be read after it is written.

The stream maintains a current position, which moves according to bytes read or written, and which can be one past the end of the array. The stream also maintains a current file size, which is never greater than size. If mode starts with r, the position starts at 0, and file size starts at size if buf was provided. If mode starts with w, the position and file size start at 0, and if buf was provided, the first byte is set to NUL. If mode starts with a, the position and file size start at the location of the first NUL byte, or else size if buf was provided.

When reading, NUL bytes have no significance, and reads cannot exceed the current file size. When writing, the file size can increase up to size as needed, and NUL bytes may be embedded in the stream (see open_memstream for an alternative that automatically enlarges the buffer). When the stream is flushed or closed after a write that changed the file size, a NUL byte is written at the current position if there is still room; if the stream is not also open for reading, a NUL byte is additionally written at the last byte of buf when the stream has exceeded size, so that a write-only buf is always NUL-terminated when the stream is flushed or closed (and the initial size should take this into account). It is not possible to seek outside the bounds of size. A NUL byte written during a flush is restored to its previous value when seeking elsewhere in the string.

Returns

The return value is an open FILE pointer on success. On error, NULL is returned, and errno will be set to EINVAL if size is zero or mode is invalid, ENOMEM if buf was NULL and memory could not be allocated, or EMFILE if too many streams are already open.

Portability

This function is being added to POSIX 200x, but is not in POSIX 2001.

Supporting OS subroutines required: sbrk.
4.17 fopen—open a file

Synopsis

```c
#include <stdio.h>

FILE *fopen(const char *file, const char *mode);

FILE *_fopen_r(struct _reent *reent, const char *file, const char *mode);
```

Description

`fopen` initializes the data structures needed to read or write a file. Specify the file’s name as the string at `file`, and the kind of access you need to the file with the string at `mode`. The alternate function `_fopen_r` is a reentrant version. The extra argument `reent` is a pointer to a reentrancy structure.

Three fundamental kinds of access are available: read, write, and append. `*mode` must begin with one of the three characters ‘r’, ‘w’, or ‘a’, to select one of these:

- **r**: Open the file for reading; the operation will fail if the file does not exist, or if the host system does not permit you to read it.

- **w**: Open the file for writing from the beginning of the file: effectively, this always creates a new file. If the file whose name you specified already existed, its old contents are discarded.

- **a**: Open the file for appending data, that is writing from the end of file. When you open a file this way, all data always goes to the current end of file; you cannot change this using `fseek`.

Some host systems distinguish between “binary” and “text” files. Such systems may perform data transformations on data written to, or read from, files opened as “text”. If your system is one of these, then you can append a ‘b’ to any of the three modes above, to specify that you are opening the file as a binary file (the default is to open the file as a text file). ‘rb’, then, means “read binary”; ‘wb’, “write binary”; and ‘ab’, “append binary”.

To make C programs more portable, the ‘b’ is accepted on all systems, whether or not it makes a difference.

Finally, you might need to both read and write from the same file. You can also append a ‘+’ to any of the three modes, to permit this. (If you want to append both ‘b’ and ‘+’, you can do it in either order: for example, "rb+" means the same thing as "r+b" when used as a mode string.)

Use "r+" (or "rb+") to permit reading and writing anywhere in an existing file, without discarding any data; "w+" (or "wb+") to create a new file (or begin by discarding all data from an old one) that permits reading and writing anywhere in it; and "a+" (or "ab+") to permit reading anywhere in an existing file, but writing only at the end.

Returns

`fopen` returns a file pointer which you can use for other file operations, unless the file you requested could not be opened; in that situation, the result is `NULL`. If the reason for failure was an invalid string at `mode`, `errno` is set to `EINVAL`. 
Portability

fopen is required by ANSI C.

Supporting OS subroutines required: close, fstat, isatty, lseek, open, read, sbrk, write.
4.18 fopencookie—open a stream with custom callbacks

Synopsis

```c
#include <stdio.h>
FILE *fopencookie(const void *cookie, const char *mode,
                   cookie_io_functions_t functions);
```

Description

fopencookie creates a FILE stream where I/O is performed using custom callbacks. The callbacks are registered via the structure:

```c
typedef ssize_t (*cookie_read_function_t)(void *cookie, char *_buf, size_t _n);
typedef ssize_t (*cookie_write_function_t)(void *cookie, const char *_buf, size_t _n);
typedef int (*cookie_seek_function_t)(void *cookie, off_t *_off, int whence);
typedef int (*cookie_close_function_t)(void *cookie);

typedef struct
{
    cookie_read_function_t *read;
    cookie_write_function_t *write;
    cookie_seek_function_t *seek;
    cookie_close_function_t *close;
} cookie_io_functions_t;
```

The stream is opened with `mode` treated as in `fopen`. The callbacks `functions.read` and `functions.write` may only be NULL when `mode` does not require them.

`functions.read` should return -1 on failure, or else the number of bytes read (0 on EOF). It is similar to `read`, except that `cookie` will be passed as the first argument.

`functions.write` should return -1 on failure, or else the number of bytes written. It is similar to `write`, except that `cookie` will be passed as the first argument.

`functions.seek` should return -1 on failure, and 0 on success, with `_off` set to the current file position. It is a cross between `lseek` and `fseek`, with the `_whence` argument interpreted in the same manner. A NULL `functions.seek` makes the stream behave similarly to a pipe in relation to stdio functions that require positioning.

`functions.close` should return -1 on failure, or 0 on success. It is similar to `close`, except that `cookie` will be passed as the first argument. A NULL `functions.close` merely flushes all data then lets `fclose` succeed. A failed close will still invalidate the stream.

Read and write I/O functions are allowed to change the underlying buffer on fully buffered or line buffered streams by calling `setvbuf`. They are also not required to completely fill or empty the buffer. They are not, however, allowed to change streams from unbuffered to buffered or to change the state of the line buffering flag. They must also be prepared to have read or write calls occur on buffers other than the one most recently specified.

Returns

The return value is an open FILE pointer on success. On error, NULL is returned, and `errno` will be set to EINVAL if a function pointer is missing or `mode` is invalid, ENOMEM if the stream cannot be created, or EMFILE if too many streams are already open.

Portability
This function is a newlib extension, copying the prototype from Linux. It is not portable. See also the funopen interface from BSD.
Supporting OS subroutines required: sbrk.
4.19 `fputc`—write a character on a stream or file

Synopsis

```c
#include <stdio.h>
int fputc(int ch, FILE *fp);

#include <stdio.h>
int _fputc_r(struct _rent *ptr, int ch, FILE *fp);
```

Description

`fputc` converts the argument `ch` from an `int` to an `unsigned char`, then writes it to the file or stream identified by `fp`.

If the file was opened with append mode (or if the stream cannot support positioning), then the new character goes at the end of the file or stream. Otherwise, the new character is written at the current value of the position indicator, and the position indicator oadvances by one.

For a macro version of this function, see `putc`.

The `_fputc_r` function is simply a reentrant version of `fputc` that takes an additional reentrant structure argument: `ptr`.

Returns

If successful, `fputc` returns its argument `ch`. If an error intervenes, the result is `EOF`. You can use `ferror(fp)` to query for errors.

Portability

`fputc` is required by ANSI C.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.
4.20 fputs—write a character string in a file or stream

Synopsis

```c
#include <stdio.h>
int fputs(const char *s, FILE *fp);
```

```c
#include <stdio.h>
int _fputs_r(struct _reent *ptr, const char *s, FILE *fp);
```

Description

fputs writes the string at s (but without the trailing null) to the file or stream identified by fp.

_fputs_r is simply the reentrant version of fputs that takes an additional reentrant struct pointer argument: ptr.

Returns

If successful, the result is 0; otherwise, the result is EOF.

Portability

ANSI C requires fputs, but does not specify that the result on success must be 0; any non-negative value is permitted.

Supporting OS subroutines required: close, fstat, isatty, lseek, read, sbrk, write.
4.21 `fputwc`, `putwc`—write a wide character on a stream or file

Synopsis

```c
#include <stdio.h>
#include <wchar.h>
wint_t fputwc(wchar_t wc, FILE *fp);

#include <stdio.h>
#include <wchar.h>
wint_t _fputwc_r(struct _reent *ptr, wchar_t wc, FILE *fp);

#include <stdio.h>
#include <wchar.h>
wint_t putwc(wchar_t wc, FILE *fp);

#include <stdio.h>
#include <wchar.h>
wint_t _putwc_r(struct _reent *ptr, wchar_t wc, FILE *fp);
```

Description

`fputwc` writes the wide character argument `wc` to the file or stream identified by `fp`. If the file was opened with append mode (or if the stream cannot support positioning), then the new wide character goes at the end of the file or stream. Otherwise, the new wide character is written at the current value of the position indicator, and the position indicator advances by one.

The `putwc` function or macro functions identically to `fputwc`. It may be implemented as a macro, and may evaluate its argument more than once. There is no reason ever to use it. The `_fputwc_r` and `_putwc_r` functions are simply reentrant versions of `fputwc` and `putwc` that take an additional reentrant structure argument: `ptr`.

Returns

If successful, `fputwc` and `putwc` return their argument `wc`. If an error intervenes, the result is `EOF`. You can use `ferror(fp)` to query for errors.

Portability

C99, POSIX.1-2001
4.22 `fputws`—write a wide character string in a file or stream

**Synopsis**

```c
#include <wchar.h>
int fputws(const wchar_t *ws, FILE *fp);

#include <wchar.h>
int _fputws_r(struct _reent *ptr, const wchar_t *ws, FILE *fp);
```

**Description**

`fputws` writes the wide character string at `ws` (but without the trailing null) to the file or stream identified by `fp`.

`_fputws_r` is simply the reentrant version of `fputws` that takes an additional reentrant struct pointer argument: `ptr`.

**Returns**

If successful, the result is a non-negative integer; otherwise, the result is -1 to indicate an error.

**Portability**

C99, POSIX.1-2001
4.23 fread—read array elements from a file

Synopsis

```c
#include <stdio.h>
size_t fread(void *buf, size_t size, size_t count, FILE *fp);
```

```c
#include <stdio.h>
size_t _fread_r(struct _reent *ptr, void *buf, size_t size, size_t count, FILE *fp);
```

Description

fread attempts to copy, from the file or stream identified by fp, count elements (each of size size) into memory, starting at buf. fread may copy fewer elements than count if an error, or end of file, intervenes.

fread also advances the file position indicator (if any) for fp by the number of characters actually read.

_fread_r is simply the reentrant version of fread that takes an additional reentrant structure pointer argument: ptr.

Returns

The result of fread is the number of elements it succeeded in reading.

Portability

ANSI C requires fread.

Supporting OS subroutines required: close, fstat, isatty, lseek, read, sbrk, write.
4.24 `freopen`—open a file using an existing file descriptor

**Synopsis**

```c
#include <stdio.h>
FILE *freopen(const char *file, const char *mode, FILE *fp);
FILE *_freopen_r(struct _reent *ptr, const char *file, const char *mode, FILE *fp);
```

**Description**

Use this variant of `fopen` if you wish to specify a particular file descriptor `fp` (notably `stdin`, `stdout`, or `stderr`) for the file.

If `fp` was associated with another file or stream, `freopen` closes that other file or stream (but ignores any errors while closing it).

`file` and `mode` are used just as in `fopen`.

If `file` is `NULL`, the underlying stream is modified rather than closed. The file cannot be given a more permissive access mode (for example, a `mode` of "w" will fail on a read-only file descriptor), but can change status such as append or binary mode. If modification is not possible, failure occurs.

**Returns**

If successful, the result is the same as the argument `fp`. If the file cannot be opened as specified, the result is `NULL`.

**Portability**

ANSI C requires `freopen`.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `open`, `read`, `sbrk`, `write`. 
4.25 fseek, fseeko—set file position

Synopsis

```
#include <stdio.h>
int fseek(FILE *fp, long offset, int whence)
int fseeko(FILE *fp, off_t offset, int whence)
int _fseek_r(struct _reent *ptr, FILE *fp,
long offset, int whence)
int _fseeko_r(struct _reent *ptr, FILE *fp,
off_t offset, int whence)
```

Description

Objects of type FILE can have a “position” that records how much of the file your program has already read. Many of the stdio functions depend on this position, and many change it as a side effect.

You can use fseek/fseeko to set the position for the file identified by fp. The value of offset determines the new position, in one of three ways selected by the value of whence (defined as macros in ‘stdio.h’):

SEEK_SET—offset is the absolute file position (an offset from the beginning of the file) desired. offset must be positive.

SEEK_CUR—offset is relative to the current file position. offset can meaningfully be either positive or negative.

SEEK_END—offset is relative to the current end of file. offset can meaningfully be either positive (to increase the size of the file) or negative.

See ftell/ftello to determine the current file position.

Returns

fseek/fseeko return 0 when successful. On failure, the result is EOF. The reason for failure is indicated in errno: either ESPIPE (the stream identified by fp doesn’t support repositioning) or EINVAL (invalid file position).

Portability

ANSI C requires fseek.

fseeko is defined by the Single Unix specification.

Supporting OS subroutines required: close, fstat, isatty, lseek, read, sbrk, write.
4.26 fsetpos—restore position of a stream or file

Synopsis

```c
#include <stdio.h>
int fsetpos(FILE *fp, const fpos_t *pos);
int _fsetpos_r(struct _reent *ptr, FILE *fp, const fpos_t *pos);
```

Description

Objects of type `FILE` can have a “position” that records how much of the file your program has already read. Many of the `stdio` functions depend on this position, and many change it as a side effect.

You can use `fsetpos` to return the file identified by `fp` to a previous position `*pos` (after first recording it with `fgetpos`).

See `fseek` for a similar facility.

Returns

`fgetpos` returns 0 when successful. If `fgetpos` fails, the result is 1. The reason for failure is indicated in `errno`: either `ESPIPE` (the stream identified by `fp` doesn’t support repositioning) or `EINVAL` (invalid file position).

Portability

ANSI C requires `fsetpos`, but does not specify the nature of `*pos` beyond identifying it as written by `fgetpos`.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`. 
4.27 ftell, ftello—return position in a stream or file

Synopsis
#include <stdio.h>
long ftell(FILE *fp);
off_t ftello(FILE *fp);
long _ftell_r(struct _reent *ptr, FILE *fp);
off_t _ftello_r(struct _reent *ptr, FILE *fp);

Description
Objects of type FILE can have a “position” that records how much of the file your program has already read. Many of the stdio functions depend on this position, and many change it as a side effect.

The result of ftell/ftello is the current position for a file identified by fp. If you record this result, you can later use it with fseek/fseeko to return the file to this position. The difference between ftell and ftello is that ftell returns long and ftello returns off_t.

In the current implementation, ftell/ftello simply uses a character count to represent the file position; this is the same number that would be recorded by fgetpos.

Returns
ftell/ftello return the file position, if possible. If they cannot do this, they return -1L. Failure occurs on streams that do not support positioning; the global errno indicates this condition with the value ESPPIPE.

Portability
ftell is required by the ANSI C standard, but the meaning of its result (when successful) is not specified beyond requiring that it be acceptable as an argument to fseek. In particular, other conforming C implementations may return a different result from ftell than what fgetpos records.

ftello is defined by the Single Unix specification.
No supporting OS subroutines are required.
4.28 funopen, fropen, fwopen—open a stream with custom callbacks

Synopsis

#include <stdio.h>

FILE *funopen(const void *cookie,
               int (*readfn) (void *cookie, char *buf, int n),
               int (*writefn) (void *cookie, const char *buf, int n),
               fpos_t (*seekfn) (void *cookie, fpos_t off, int whence),
               int (*closefn) (void *cookie));

FILE *fropen(const void *cookie,
              int (*readfn) (void *cookie, char *buf, int n));

FILE *fwopen(const void *cookie,
              int (*writefn) (void *cookie, const char *buf, int n));

Description

funopen creates a FILE stream where I/O is performed using custom callbacks. At least one of readfn and writefn must be provided, which determines whether the stream behaves with mode <"r">, <"w">, or <"r+">.

readfn should return -1 on failure, or else the number of bytes read (0 on EOF). It is similar to read, except that <int> rather than <size_t> bounds a transaction size, and cookie will be passed as the first argument. A NULL readfn makes attempts to read the stream fail.

writefn should return -1 on failure, or else the number of bytes written. It is similar to write, except that <int> rather than <size_t> bounds a transaction size, and cookie will be passed as the first argument. A NULL writefn makes attempts to write the stream fail.

seekfn should return (fpos_t)-1 on failure, or else the current file position. It is similar to lseek, except that cookie will be passed as the first argument. A NULL seekfn makes the stream behave similarly to a pipe in relation to stdio functions that require positioning. This implementation assumes fpos_t and off_t are the same type.

closefn should return -1 on failure, or 0 on success. It is similar to close, except that cookie will be passed as the first argument. A NULL closefn merely flushes all data then lets fclose succeed. A failed close will still invalidate the stream.

Read and write I/O functions are allowed to change the underlying buffer on fully buffered or line buffered streams by calling setvbuf. They are also not required to completely fill or empty the buffer. They are not, however, allowed to change streams from unbuffered to buffered or to change the state of the line buffering flag. They must also be prepared to have read or write calls occur on buffers other than the one most recently specified.

The functions fropen and fwopen are convenience macros around funopen that only use the specified callback.

Returns

The return value is an open FILE pointer on success. On error, NULL is returned, and errno will be set to EINVAL if a function pointer is missing, ENOMEM if the stream cannot be created, or EMFILE if too many streams are already open.

Portability
This function is a newlib extension, copying the prototype from BSD. It is not portable. See also the `fopencookie` interface from Linux.
Supporting OS subroutines required: `sbrk`.
4.29 fwide—set and determine the orientation of a FILE stream

Synopsis

```c
#include <wchar.h>
int fwide(FILE *fp, int mode)

int _fwide_r(struct _reent *ptr, FILE *fp, int mode)
```

Description
When `mode` is zero, the `fwide` function determines the current orientation of `fp`. It returns a value > 0 if `fp` is wide-character oriented, i.e. if wide character I/O is permitted but char I/O is disallowed. It returns a value < 0 if `fp` is byte oriented, i.e. if char I/O is permitted but wide character I/O is disallowed. It returns zero if `fp` has no orientation yet; in this case the next I/O operation might change the orientation (to byte oriented if it is a char I/O operation, or to wide-character oriented if it is a wide character I/O operation).

Once a stream has an orientation, it cannot be changed and persists until the stream is closed, unless the stream is re-opened with freopen, which removes the orientation of the stream.

When `mode` is non-zero, the `fwide` function first attempts to set `fp`’s orientation (to wide-character oriented if `mode` > 0, or to byte oriented if `mode` < 0). It then returns a value denoting the current orientation, as above.

Returns
The `fwide` function returns `fp`’s orientation, after possibly changing it. A return value > 0 means wide-character oriented. A return value < 0 means byte oriented. A return value of zero means undecided.

Portability
4.30 fwrite—write array elements

Synopsis

```c
#include <stdio.h>
size_t fwrite(const void *buf, size_t size,
              size_t count, FILE *fp);

#include <stdio.h>
size_t _fwrite_r(struct _reent *ptr, const void *buf, size_t size,
                 size_t count, FILE *fp);
```

Description
fwrite attempts to copy, starting from the memory location buf, count elements (each of size size) into the file or stream identified by fp. fwrite may copy fewer elements than count if an error intervenes.
fwrite also advances the file position indicator (if any) for fp by the number of characters actually written.
_fwrite_r is simply the reentrant version of fwrite that takes an additional reentrant structure argument: ptr.

Returns
If fwrite succeeds in writing all the elements you specify, the result is the same as the argument count. In any event, the result is the number of complete elements that fwrite copied to the file.

Portability
ANSI C requires fwrite.
Supporting OS subroutines required: close, fstat, isatty, lseek, read, sbrk, write.
4.31 getc—read a character (macro)

Synopsis

```c
#include <stdio.h>
int getc(FILE *fp);
```

```c
#include <stdio.h>
int _getc_r(struct _reent *ptr, FILE *fp);
```

Description

getc is a macro, defined in `stdio.h`. You can use `getc` to get the next single character from the file or stream identified by `fp`. As a side effect, `getc` advances the file’s current position indicator.

For a subroutine version of this macro, see `fgetc`.

The `_getc_r` function is simply the reentrant version of `getc` which passes an additional reentrancy structure pointer argument: `ptr`.

Returns

The next character (read as an `unsigned char`, and cast to `int`), unless there is no more data, or the host system reports a read error; in either of these situations, `getc` returns `EOF`.

You can distinguish the two situations that cause an `EOF` result by using the `ferror` and `feof` functions.

Portability

ANSI C requires `getc`; it suggests, but does not require, that `getc` be implemented as a macro. The standard explicitly permits macro implementations of `getc` to use the argument more than once; therefore, in a portable program, you should not use an expression with side effects as the `getc` argument.

Supporting OS subroutines required: `close, fstat, isatty, lseek, read, sbrk, write`. 
4.32 getc_unlocked—non-thread-safe version of getc (macro)

Synopsis

```
#include <stdio.h>
int getc_unlocked(FILE *fp);

#include <stdio.h>
int _getc_unlocked_r(FILE *fp);
```

Description

getc_unlocked is a non-thread-safe version of getc declared in stdio.h. getc_unlocked may only safely be used within a scope protected by flockfile() (or ftrylockfile()) and funlockfile(). These functions may safely be used in a multi-threaded program if and only if they are called while the invoking thread owns the ( FILE *) object, as is the case after a successful call to the flockfile() or ftrylockfile() functions. If threads are disabled, then getc_unlocked is equivalent to getc.

The _getc_unlocked_r function is simply the reentrant version of getc_unlocked which passes an additional reentrancy structure pointer argument: ptr.

Returns

See getc.

Portability

POSIX 1003.1 requires getc_unlocked. getc_unlocked may be implemented as a macro, so arguments should not have side-effects.

Supporting OS subroutines required: close, fstat, isatty, lseek, read, sbrk, write.
4.33 `getchar`—read a character (macro)

**Synopsis**

```c
#include <stdio.h>
int getchar(void);

int _getchar_r(struct _reent *reent);
```

**Description**

`getchar` is a macro, defined in `stdio.h`. You can use `getchar` to get the next single character from the standard input stream. As a side effect, `getchar` advances the standard input’s current position indicator.

The alternate function `_getchar_r` is a reentrant version. The extra argument `reent` is a pointer to a reentrancy structure.

**Returns**

The next character (read as an `unsigned char`, and cast to `int`), unless there is no more data, or the host system reports a read error; in either of these situations, `getchar` returns `EOF`.

You can distinguish the two situations that cause an `EOF` result by using `ferror(stdin)` and `feof(stdin)`.

**Portability**

ANSI C requires `getchar`; it suggests, but does not require, that `getchar` be implemented as a macro.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.
4.34 getchar_unlocked—non-thread-safe version of getchar (macro)

Synopsis

```c
#include <stdio.h>
int getchar_unlocked();

#include <stdio.h>
int _getchar_unlocked_r(struct _reent *ptr);
```

Description

g getchar_unlocked is a non-thread-safe version of getchar declared in stdio.h. getchar_unlocked may only safely be used within a scope protected by flockfile() (or ftrylockfile()) and funlockfile(). These functions may safely be used in a multi-threaded program if and only if they are called while the invoking thread owns the (FILE *) object, as is the case after a successful call to the flockfile() or ftrylockfile() functions. If threads are disabled, then getchar_unlocked is equivalent to getchar.

The _getchar_unlocked_r function is simply the reentrant version of getchar_unlocked which passes an additional reentrancy structure pointer argument: ptr.

Returns

See getchar.

Portability

POSIX 1003.1 requires getchar_unlocked. getchar_unlocked may be implemented as a macro.

Supporting OS subroutines required: close, fstat, isatty, lseek, read, sbrk, write.
4.35 getdelim—read a line up to a specified line delimiter

Synopsis

```c
#include <stdio.h>
int getdelim(char **bufptr, size_t *n, int delim, FILE *fp);
```

Description
getdelim reads a file `fp` up to and possibly including a specified delimiter `delim`. The line is read into a buffer pointed to by `bufptr` and designated with size `*n`. If the buffer is not large enough, it will be dynamically grown by `getdelim`. As the buffer is grown, the pointer to the size `n` will be updated.

Returns
getdelim returns `-1` if no characters were successfully read; otherwise, it returns the number of bytes successfully read. At end of file, the result is nonzero.

Portability
getdelim is a glibc extension.
No supporting OS subroutines are directly required.
4.36 getline—read a line from a file

Synopsis

```c
#include <stdio.h>
ssize_t getline(char **bufptr, size_t *n, FILE *fp);
```

Description
gline reads a file `fp` up to and possibly including the newline character. The line is read into a buffer pointed to by `bufptr` and designated with size `*n`. If the buffer is not large enough, it will be dynamically grown by `getdelim`. As the buffer is grown, the pointer to the size `n` will be updated.

gline is equivalent to `getdelim(bufptr, n, '\n', fp);`

Returns
gline returns `-1` if no characters were successfully read, otherwise, it returns the number of bytes successfully read. At end of file, the result is nonzero.

Portability
gline is a glibc extension.

No supporting OS subroutines are directly required.
4.37 `gets`—get character string (obsolete, use `fgets` instead)

Synopsis

```c
#include <stdio.h>

char *gets(char *buf);

char *_gets_r(struct _reent *reent, char *buf);
```

Description
Reads characters from standard input until a newline is found. The characters up to the newline are stored in `buf`. The newline is discarded, and the buffer is terminated with a 0.

This is a dangerous function, as it has no way of checking the amount of space available in `buf`. One of the attacks used by the Internet Worm of 1988 used this to overrun a buffer allocated on the stack of the finger daemon and overwrite the return address, causing the daemon to execute code downloaded into it over the connection.

The alternate function `_gets_r` is a reentrant version. The extra argument `reent` is a pointer to a reentrancy structure.

Returns

`gets` returns the buffer passed to it, with the data filled in. If end of file occurs with some data already accumulated, the data is returned with no other indication. If end of file occurs with no data in the buffer, NULL is returned.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`. 
4.38 getw—read a word (int)

Synopsis

#include <stdio.h>
int getw(FILE *fp);

Description

getw is a function, defined in stdio.h. You can use getw to get the next word from the file or stream identified by fp. As a side effect, getw advances the file’s current position indicator.

Returns

The next word (read as an int), unless there is no more data or the host system reports a read error; in either of these situations, getw returns EOF. Since EOF is a valid int, you must use ferror or feof to distinguish these situations.

Portability

getw is a remnant of K&R C; it is not part of any ISO C Standard. fread should be used instead. In fact, this implementation of getw is based upon fread.

Supporting OS subroutines required: fread.
4.39 `getwchar`—read a wide character from standard input

Synopsis

```c
#include <wchar.h>

wint_t getwchar(void);

wint_t _getwchar_r(struct _reent *reent);
```

Description

`getwchar` function or macro is the wide character equivalent of the `getchar` function. You can use `getwchar` to get the next wide character from the standard input stream. As a side effect, `getwchar` advances the standard input’s current position indicator.

The alternate function `_getwchar_r` is a reentrant version. The extra argument `reent` is a pointer to a reentrancy structure.

Returns

The next wide character cast to `wint_t`, unless there is no more data, or the host system reports a read error; in either of these situations, `getwchar` returns `WEOF`.

You can distinguish the two situations that cause an `WEOF` result by using `ferror(stdin)` and `feof(stdin)`.

Portability

C99
4.40 `mktemp`, `mkstemp`—generate unused file name

Synopsis

```c
#include <stdio.h>
char *mktemp(char *path);
int mkstemp(char *path);

char *_mktemp_r(struct _reent *reent, char *path);
int *_mkstemp_r(struct _reent *reent, char *path);
```

Description

`mktemp` and `mkstemp` attempt to generate a file name that is not yet in use for any existing file. `mkstemp` creates the file and opens it for reading and writing; `mktemp` simply generates the file name.

You supply a simple pattern for the generated file name, as the string at `path`. The pattern should be a valid filename (including path information if you wish) ending with some number of ‘X’ characters. The generated filename will match the leading part of the name you supply, with the trailing ‘X’ characters replaced by some combination of digits and letters.

The alternate functions `_mktemp_r` and `_mkstemp_r` are reentrant versions. The extra argument `reent` is a pointer to a reentrancy structure.

Returns

`mktemp` returns the pointer `path` to the modified string representing an unused filename, unless it could not generate one, or the pattern you provided is not suitable for a filename; in that case, it returns `NULL`.

`mkstemp` returns a file descriptor to the newly created file, unless it could not generate an unused filename, or the pattern you provided is not suitable for a filename; in that case, it returns `-1`.

Portability

ANSI C does not require either `mktemp` or `mkstemp`; the System V Interface Definition requires `mktemp` as of Issue 2.

Supporting OS subroutines required: `getpid`, `open`, `stat`. 
4.41 open_memstream, open_wmemstream—open a write stream
around an arbitrary-length string

Synopsis

```c
#include <stdio.h>
FILE *open_memstream(char **restrict buf,
                      size_t *restrict size);

#include <wchar.h>
FILE *open_wmemstream(wchar_t **restrict buf,
                      size_t *restrict size);
```

Description

`open_memstream` creates a seekable, byte-oriented FILE stream that wraps an arbitrary-length buffer, created as if by `malloc`. The current contents of `*buf` are ignored; this implementation uses `*size` as a hint of the maximum size expected, but does not fail if the hint was wrong. The parameters `buf` and `size` are later stored through following any call to `fflush` or `fclose`, set to the current address and usable size of the allocated string; although after flush, the pointer is only valid until another stream operation that results in a write. Behavior is undefined if the user alters either `*buf` or `*size` prior to `fclose`.

`open_wmemstream` is like `open_memstream` just with the associated stream being wide-oriented. The size set in `size` in subsequent operations is the number of wide characters.

The stream is write-only, since the user can directly read `*buf` after a flush; see `fmemopen` for a way to wrap a string with a readable stream. The user is responsible for calling `free` on the final `*buf` after `fclose`.

Any time the stream is flushed, a NUL byte is written at the current position (but is not counted in the buffer length), so that the string is always NUL-terminated after at most `*size` bytes (or wide characters in case of `open_wmemstream`). However, data previously written beyond the current stream offset is not lost, and the NUL value written during a flush is restored to its previous value when seeking elsewhere in the string.

Returns

The return value is an open FILE pointer on success. On error, NULL is returned, and `errno` will be set to EINVAL if `buf` or `size` is NULL, ENOMEM if memory could not be allocated, or EMFILE if too many streams are already open.

Portability

POSIX.1-2008

Supporting OS subroutines required: `sbrk`.
4.42 perror—print an error message on standard error

Synopsis

```c
#include <stdio.h>
void perror(char *prefix);
void _perror_r(struct _reent *reent, char *prefix);
```

Description
Use perror to print (on standard error) an error message corresponding to the current value of the global variable errno. Unless you use NULL as the value of the argument prefix, the error message will begin with the string at prefix, followed by a colon and a space ( : ). The remainder of the error message is one of the strings described for strerror.

The alternate function _perror_r is a reentrant version. The extra argument reent is a pointer to a reentrancy structure.

Returns
perror returns no result.

Portability
ANSI C requires perror, but the strings issued vary from one implementation to another. Supporting OS subroutines required: close, fstat, isatty, lseek, read, sbrk, write.
4.43 putc—write a character (macro)

Synopsis

```c
#include <stdio.h>
int putc(int ch, FILE *fp);
```

```c
#include <stdio.h>
int _putc_r(struct _reent *ptr, int ch, FILE *fp);
```

Description

putc is a macro, defined in `stdio.h`. `putc` writes the argument `ch` to the file or stream identified by `fp`, after converting it from an `int` to an `unsigned char`.

If the file was opened with append mode (or if the stream cannot support positioning), then the new character goes at the end of the file or stream. Otherwise, the new character is written at the current value of the position indicator, and the position indicator advances by one.

For a subroutine version of this macro, see `fputc`.

The `_putc_r` function is simply the reentrant version of `putc` that takes an additional reentrant structure argument: `ptr`.

Returns

If successful, `putc` returns its argument `ch`. If an error intervenes, the result is `EOF`. You can use `ferror(fp)` to query for errors.

Portability

ANSI C requires `putc`; it suggests, but does not require, that `putc` be implemented as a macro. The standard explicitly permits macro implementations of `putc` to use the `fp` argument more than once; therefore, in a portable program, you should not use an expression with side effects as this argument.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`. 
4.44 `putc_unlocked`—non-thread-safe version of `putc` (macro)

**Synopsis**
```
#include <stdio.h>
int putc_unlocked(int ch, FILE *fp);

#include <stdio.h>
int _putc_unlocked_r(struct _reent *ptr, int ch, FILE *fp);
```

**Description**
`putc_unlocked` is a non-thread-safe version of `putc` declared in `<stdio.h>`. `putc_unlocked` may only safely be used within a scope protected by `flockfile()` (or `ftrylockfile()`) and `funlockfile()`. These functions may safely be used in a multi-threaded program if and only if they are called while the invoking thread owns the (FILE *) object, as is the case after a successful call to the `flockfile()` or `ftrylockfile()` functions. If threads are disabled, then `putc_unlocked` is equivalent to `putc`.

The function `_putc_unlocked_r` is simply the reentrant version of `putc_unlocked` that takes an additional reentrant structure pointer argument: `ptr`.

**Returns**
See `putc`.

**Portability**
POSIX 1003.1 requires `putc_unlocked`. `putc_unlocked` may be implemented as a macro, so arguments should not have side-effects.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`. 
4.45 putchar—write a character (macro)

Synopsis
#include <stdio.h>
int putchar(int ch);

int _putchar_r(struct _reent *reent, int ch);

Description
putchar is a macro, defined in stdio.h. putchar writes its argument to the standard output stream, after converting it from an int to an unsigned char.
The alternate function _putchar_r is a reentrant version. The extra argument reent is a pointer to a reentrancy structure.

Returns
If successful, putchar returns its argument ch. If an error intervenes, the result is EOF. You can use ‘ferror(stdin)’ to query for errors.

Portability
ANSI C requires putchar; it suggests, but does not require, that putchar be implemented as a macro.
Supporting OS subroutines required: close, fstat, isatty, lseek, read, sbrk, write.
4.46 putchar_unlocked—non-thread-safe version of putchar (macro)

Synopsis

```
#include <stdio.h>
int putchar_unlocked(int ch);
```

Description

putchar_unlocked is a non-thread-safe version of putchar declared in stdio.h. putchar_unlocked may only safely be used within a scope protected by flockfile() (or ftrylockfile()) and funlockfile(). These functions may safely be used in a multi-threaded program if and only if they are called while the invoking thread owns the ( FILE *) object, as is the case after a successful call to the flockfile() or ftrylockfile() functions. If threads are disabled, then putchar_unlocked is equivalent to putchar.

Returns

See putchar.

Portability

POSIX 1003.1 requires putchar_unlocked. putchar_unlocked may be implemented as a macro.

Supporting OS subroutines required: close, fstat, isatty, lseek, read, sbrk, write.
4.47 puts—write a character string

Synopsis

```c
#include <stdio.h>
int puts(const char *s);

int _puts_r(struct _reent *reent, const char *s);
```

Description

`puts` writes the string at `s` (followed by a newline, instead of the trailing null) to the standard output stream.

The alternate function `_puts_r` is a reentrant version. The extra argument `reent` is a pointer to a reentrancy structure.

Returns

If successful, the result is a nonnegative integer; otherwise, the result is `EOF`.

Portability

ANSI C requires `puts`, but does not specify that the result on success must be 0; any non-negative value is permitted.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`. 
4.48 putw—write a word (int)

Synopsis

```c
#include <stdio.h>
int putw(int w, FILE *fp);
```

Description

`putw` is a function, defined in `stdio.h`. You can use ` putw` to write a word to the file or stream identified by `fp`. As a side effect, `putw` advances the file's current position indicator.

Returns

Zero on success, EOF on failure.

Portability

`putw` is a remnant of K&R C; it is not part of any ISO C Standard. `fwrite` should be used instead. In fact, this implementation of `putw` is based upon `fwrite`.

Supporting OS subroutines required: `fwrite`. 
4.49 

**putwchar—write a wide character to standard output**

**Synopsis**

```c
#include <wchar.h>
wint_t putwchar(wchar_t wc);

wint_t _putwchar_r(struct _reent *reent, wchar_t wc);
```

**Description**

The `putwchar` function or macro is the wide-character equivalent of the `putchar` function. It writes the wide character wc to stdout.

The alternate function `_putwchar_r` is a reentrant version. The extra argument `reent` is a pointer to a reentrancy structure.

**Returns**

If successful, `putwchar` returns its argument wc. If an error intervenes, the result is EOF. You can use `ferror(stdin)` to query for errors.

**Portability**

C99
4.50 remove—delete a file’s name

Synopsis

```c
#include <stdio.h>
int remove(char *filename);

int _remove_r(struct _reent *reent, char *filename);
```

Description

Use `remove` to dissolve the association between a particular filename (the string at `filename`) and the file it represents. After calling `remove` with a particular filename, you will no longer be able to open the file by that name.

In this implementation, you may use `remove` on an open file without error; existing file descriptors for the file will continue to access the file’s data until the program using them closes the file.

The alternate function `_remove_r` is a reentrant version. The extra argument `reent` is a pointer to a reentrancy structure.

Returns

`remove` returns 0 if it succeeds, -1 if it fails.

Portability

ANSI C requires `remove`, but only specifies that the result on failure be nonzero. The behavior of `remove` when you call it on an open file may vary among implementations.

Supporting OS subroutine required: `unlink`. 
4.51 rename—rename a file

Synopsis

```
#include <stdio.h>
int rename(const char *old, const char *new);
```

Description
Use `rename` to establish a new name (the string at `new`) for a file now known by the string at `old`. After a successful `rename`, the file is no longer accessible by the string at `old`. If `rename` fails, the file named `*old` is unaffected. The conditions for failure depend on the host operating system.

Returns
The result is either 0 (when successful) or -1 (when the file could not be renamed).

Portability
ANSI C requires `rename`, but only specifies that the result on failure be nonzero. The effects of using the name of an existing file as `*new` may vary from one implementation to another. Supporting OS subroutines required: `link`, `unlink`, or `rename`. 
4.52 rewind—reinitialize a file or stream

Synopsis

```c
#include <stdio.h>
void rewind(FILE *fp);
void _rewind_r(struct _reent *ptr, FILE *fp);
```

Description

`rewind` returns the file position indicator (if any) for the file or stream identified by `fp` to
the beginning of the file. It also clears any error indicator and flushes any pending output.

Returns

`rewind` does not return a result.

Portability

ANSI C requires `rewind`.

No supporting OS subroutines are required.
4.53 setbuf—specify full buffering for a file or stream

Synopsis

```c
#include <stdio.h>
void setbuf(FILE *fp, char *buf);
```

Description

setbuf specifies that output to the file or stream identified by fp should be fully buffered. All output for this file will go to a buffer (of size BUFSIZ, specified in ‘stdio.h’). Output will be passed on to the host system only when the buffer is full, or when an input operation intervenes.

You may, if you wish, supply your own buffer by passing a pointer to it as the argument buf. It must have size BUFSIZ. You can also use NULL as the value of buf, to signal that the setbuf function is to allocate the buffer.

Warnings

You may only use setbuf before performing any file operation other than opening the file. If you supply a non-null buf, you must ensure that the associated storage continues to be available until you close the stream identified by fp.

Returns

setbuf does not return a result.

Portability

Both ANSI C and the System V Interface Definition (Issue 2) require setbuf. However, they differ on the meaning of a NULL buffer pointer: the SVID issue 2 specification says that a NULL buffer pointer requests unbuffered output. For maximum portability, avoid NULL buffer pointers.

Supporting OS subroutines required: close, fstat, isatty, lseek, read, sbrk, write.
4.54 setbuffer—specify full buffering for a file or stream with size

Synopsis

```c
#include <stdio.h>
void setbuffer(FILE *fp, char *buf, int size);
```

Description

setbuffer specifies that output to the file or stream identified by fp should be fully buffered. All output for this file will go to a buffer (of size size). Output will be passed on to the host system only when the buffer is full, or when an input operation intervenes.

You may, if you wish, supply your own buffer by passing a pointer to it as the argument buf. It must have size size. You can also use NULL as the value of buf, to signal that the setbuffer function is to allocate the buffer.

Warnings

You may only use setbuffer before performing any file operation other than opening the file.

If you supply a non-null buf, you must ensure that the associated storage continues to be available until you close the stream identified by fp.

Returns

setbuffer does not return a result.

Portability

This function comes from BSD not ANSI or POSIX.

Supporting OS subroutines required: close, fstat, isatty, lseek, read, sbrk, write.
4.55 setlinebuf—specify line buffering for a file or stream

Synopsis

```c
#include <stdio.h>
void setlinebuf(FILE *fp);
```

Description

`setlinebuf` specifies that output to the file or stream identified by `fp` should be line buffered. This causes the file or stream to pass on output to the host system at every newline, as well as when the buffer is full, or when an input operation intervenes.

Warnings

You may only use `setlinebuf` before performing any file operation other than opening the file.

Returns

`setlinebuf` returns as per `setvbuf`.

Portability

This function comes from BSD not ANSI or POSIX.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.
4.56 setvbuf—specify file or stream buffering

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

Description
Use setvbuf to specify what kind of buffering you want for the file or stream identified by fp, by using one of the following values (from stdio.h) as the mode argument:

_IONBF Do not use a buffer: send output directly to the host system for the file or stream identified by fp.

_IOFBF Use full output buffering: output will be passed on to the host system only when the buffer is full, or when an input operation intervenes.

_IOLBF Use line buffering: pass on output to the host system at every newline, as well as when the buffer is full, or when an input operation intervenes.

Use the size argument to specify how large a buffer you wish. You can supply the buffer itself, if you wish, by passing a pointer to a suitable area of memory as buf. Otherwise, you may pass NULL as the buf argument, and setvbuf will allocate the buffer.

Warnings
You may only use setvbuf before performing any file operation other than opening the file. If you supply a non-null buf, you must ensure that the associated storage continues to be available until you close the stream identified by fp.

Returns
A 0 result indicates success, EOF failure (invalid mode or size can cause failure).

Portability
Both ANSI C and the System V Interface Definition (Issue 2) require setvbuf. However, they differ on the meaning of a NULL buffer pointer: the SVID issue 2 specification says that a NULL buffer pointer requests unbuffered output. For maximum portability, avoid NULL buffer pointers.

Both specifications describe the result on failure only as a nonzero value.
Supporting OS subroutines required: close, fstat, isatty, lseek, read, sbrk, write.
4.57 `siprintf`, `fiprintf`, `iprintf`, `sniprintf`, `asiprintf`,
    `asniprintf`—format output (integer only)

Synopsis

```
#include <stdio.h>

int iprintf(const char *format, ...);
int fiprintf(FILE *fd, const char *format, ...);
int siprintf(char *str, const char *format, ...);
int sniprintf(char *str, size_t size, const char *format, ...);
int asiprintf(char **strp, const char *format, ...);
char *asniprintf(char *str, size_t *size,
                    const char *format, ...);

int _iprintf_r(struct _reent *ptr, const char *format, ...);
int _fiprintf_r(struct _reent *ptr, FILE *fd, const char *format, ...);
int _siprintf_r(struct _reent *ptr, char *str, const char *format, ...);
int _sniprintf_r(struct _reent *ptr, char *str, size_t size, const char *format, ...);
int _asiprintf_r(struct _reent *ptr, char **strp, const char *format, ...);
char *_asniprintf_r(struct _reent *ptr, char *str, size_t *size, const char *format, ...);
```

Description

`iprintf`, `fiprintf`, `siprintf`, `sniprintf`, `asiprintf`, and `asniprintf` are the same as
`printf`, `fprintf`, `sprintf`, `snprintf`, `asprintf`, and `asnprintf`, respectively, except that
they restrict usage to non-floating-point format specifiers.

 `_iprintf_r`, `_fiprintf_r`, `_asiprintf_r`, `_siprintf_r`, `_sniprintf_r`, `_asniprintf_r`
are simply reentrant versions of the functions above.

Returns

Similar to `printf`, `fprintf`, `sprintf`, `snprintf`, `asprintf`, and `asnprintf`.

Portability

`iprintf`, `fiprintf`, `siprintf`, `sniprintf`, `asiprintf`, and `asniprintf` are newlib exten-
sions.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.
4.58 siscanf, fiscanf, iscanf—scan and format non-floating input

Synopsis

```
#include <stdio.h>

int iscanf(const char *format, ...);
int fiscanf(FILE *fd, const char *format, ...);
int siscanf(const char *str, const char *format, ...);

int _iscanf_r(struct _reent *ptr, const char *format, ...);
int _fiscanf_r(struct _reent *ptr, FILE *fd, const char *format, ...);
int _siscanf_r(struct _reent *ptr, const char *str, const char *format, ...);
```

Description

iscanf, fiscanf, and siscanf are the same as scanf, fscanf, and sscanf respectively, only that they restrict the available formats to non-floating-point format specifiers.

The routines _iscanf_r, _fiscanf_r, and _siscanf_r are reentrant versions of iscanf, fscanf, and siscanf that take an additional first argument pointing to a reentrancy structure.

Returns

iscanf returns the number of input fields successfully scanned, converted and stored; the return value does not include scanned fields which were not stored.

If iscanf attempts to read at end-of-file, the return value is EOF.

If no fields were stored, the return value is 0.

Portability

iscanf, fscanf, and siscanf are newlib extensions.

Supporting OS subroutines required: close, fstat, isatty, lseek, read, sbrk, write.
4.59 sprintf, fprintf, printf, snprintf, asprintf, asnprintf—format output

Synopsis

```
#include <stdio.h>

int printf(const char *format, ...);
int fprintf(FILE *fd, const char *format, ...);
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...);
int asprintf(char **strp, const char *format, ...);
char *asprintf(char *str, size_t *size, const char *format, ...);
int _printf_r(struct _reent *ptr, const char *format, ...);
int _fprintf_r(struct _reent *ptr, FILE *fd, const char *format, ...);
int _sprintf_r(struct _reent *ptr, char *str, const char *format, ...);
int _snprintf_r(struct _reent *ptr, char *str, size_t size, const char *format, ...);
int _asprintf_r(struct _reent *ptr, char **strp, const char *format, ...);
char *_asnprintf_r(struct _reent *ptr, char *str, size_t *size, const char *format, ...);
```

Description

`printf` accepts a series of arguments, applies to each a format specifier from `*format`, and writes the formatted data to `stdout`, without a terminating NUL character. The behavior of `printf` is undefined if there are not enough arguments for the format. `printf` returns when it reaches the end of the format string. If there are more arguments than the format requires, excess arguments are ignored.

`fprintf` is like `printf`, except that output is directed to the stream `fd` rather than `stdout`.

`sprintf` is like `printf`, except that output is directed to the buffer `str`, and a terminating NUL is output. Behavior is undefined if more output is generated than the buffer can hold.

`snprintf` is like `sprintf`, except that output is limited to at most `size` bytes, including the terminating NUL. As a special case, if `size` is 0, `str` can be NULL, and `snprintf` merely calculates how many bytes would be printed.

`asprintf` is like `sprintf`, except that the output is stored in a dynamically allocated buffer, `pstr`, which should be freed later with `free`.

`asnprintf` is like `sprintf`, except that the return type is either the original `str` if it was large enough, or a dynamically allocated string if the output exceeds `*size`; the length of the result is returned in `*size`. When dynamic allocation occurs, the contents of the original `str` may have been modified.
For `sprintf`, `snprintf`, and `asprintf`, the behavior is undefined if the output *str* overlaps with one of the arguments. Behavior is also undefined if the argument for `%n` within *format* overlaps another argument.

`format` is a pointer to a character string containing two types of objects: ordinary characters (other than `%`), which are copied unchanged to the output, and conversion specifications, each of which is introduced by `%`. (To include `%` in the output, use `%%%` in the format string.) A conversion specification has the following form:

```
%[pos][flags][width][.prec][size]type
```

The fields of the conversion specification have the following meanings:

- **pos**
  Conversions normally consume arguments in the order that they are presented. However, it is possible to consume arguments out of order, and reuse an argument for more than one conversion specification (although the behavior is undefined if the same argument is requested with different types), by specifying `pos`, which is a decimal integer followed by `$`. The integer must be between 1 and `<NL_ARGMAX>` from limits.h, and if argument `%n` is requested, all earlier arguments must be requested somewhere within `format`. If positional parameters are used, then all conversion specifications except for `%%` must specify a position.

- **flags**
  `flags` is an optional sequence of characters which control output justification, numeric signs, decimal points, trailing zeros, and octal and hex prefixes. The flag characters are minus (`-`), plus (`+`), space (` `), zero (`0`), sharp (`#`), and quote (`'`). They can appear in any combination, although not all flags can be used for all conversion specification types.

  - Since newlib only supports the C locale, this flag has no effect in this implementation. But in other locales, when `type` is `i`, `d`, `u`, `o`, `x`, `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G`, the locale-dependent thousand’s separator is inserted prior to zero padding.
  - The result of the conversion is left justified, and the right is padded with blanks. If you do not use this flag, the result is right justified, and padded on the left.
  - The result of a signed conversion (as determined by `type` of `d`, `i`, `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G`) will always begin with a plus or minus sign. (If you do not use this flag, positive values do not begin with a plus sign.)

- **space**
  If the first character of a signed conversion specification is not a sign, or if a signed conversion results in no characters, the result will begin with a space. If the space ( ` `) flag and the plus ( `+`) flag both appear, the space flag is ignored.

0  If the `type` character is `d`, `i`, `o`, `u`, `x`, `X`, `a`, `A`, `e`, `E`, `f`, `g`, or `G`: leading zeros are used to pad the field width (following any indication of sign or base); no spaces are used for padding. If the zero ( `0`) and minus ( `-`) flags both appear, the zero ( `0`) flag will be ignored. For `d`, `i`, `o`, `u`, `x`, and `X` conversions, if a precision `prec` is specified, the zero ( `0`) flag is ignored.

Note that 0 is interpreted as a flag, not as the beginning of a field width.
The result is to be converted to an alternative form, according to the type character:

- **o**: Increases precision to force the first digit of the result to be a zero.
- **x**: A non-zero result will have a \( 0x \) prefix.
- **X**: A non-zero result will have a \( 0X \) prefix.
- **a**, **A**, **e**, **E**, **f**, or **F**: The result will always contain a decimal point even if no digits follow the point. (Normally, a decimal point appears only if a digit follows it.) Trailing zeros are removed.
- **g** or **G**: The result will always contain a decimal point even if no digits follow the point. Trailing zeros are not removed.
- **all others**: Undefined.

- **width**
  
  `width` is an optional minimum field width. You can either specify it directly as a decimal integer, or indirectly by using instead an asterisk (**\( * \)**), in which case an `int` argument is used as the field width. If positional arguments are used, then the width must also be specified positionally as `*m`, with \( m \) as a decimal integer. Negative field widths are treated as specifying the minus (\( - \)) flag for left justification, along with a positive field width. The resulting format may be wider than the specified width.

- **prec**

  `prec` is an optional field; if present, it is introduced with `'.'` (a period). You can specify the precision either directly as a decimal integer or indirectly by using an asterisk (**\( * \)**), in which case an `int` argument is used as the precision. If positional arguments are used, then the precision must also be specified positionally as `*m`, with \( m \) as a decimal integer. Supplying a negative precision is equivalent to omitting the precision. If only a period is specified the precision is zero. The effect depends on the conversion type.

  - **d**, **i**, **o**, **u**, **x**, or **X**: Minimum number of digits to appear. If no precision is given, defaults to 1.
  - **a** or **A**: Number of digits to appear after the decimal point. If no precision is given, the precision defaults to the minimum needed for an exact representation.
  - **e**, **E**, **f** or **F**: Number of digits to appear after the decimal point. If no precision is given, the precision defaults to 6.
  - **g** or **G**: Maximum number of significant digits. A precision of 0 is treated the same as a precision of 1. If no precision is given, the precision defaults to 6.
  - **s** or **S**: Maximum number of characters to print from the string. If no precision is given, the entire string is printed.
all others
    undefined.

• size

size is an optional modifier that changes the data type that the corresponding argument has. Behavior is unspecified if a size is given that does not match the type.

    hh  With d, i, o, u, x, or X, specifies that the argument should be converted to a signed char or unsigned char before printing.
        With n, specifies that the argument is a pointer to a signed char.
    h   With d, i, o, u, x, or X, specifies that the argument should be converted to a short or unsigned short before printing.
        With n, specifies that the argument is a pointer to a short.
    l   With d, i, o, u, x, or X, specifies that the argument is a long or unsigned long.
        With c, specifies that the argument has type wint_t.
        With s, specifies that the argument is a pointer to wchar_t.
        With n, specifies that the argument is a pointer to a long.
        With a, A, e, E, f, F, g, or G, has no effect (because of vararg promotion rules, there is no need to distinguish between float and double).
    ll  With d, i, o, u, x, or X, specifies that the argument is a long long or unsigned long long.
        With n, specifies that the argument is a pointer to a long long.
    j   With d, i, o, u, x, or X, specifies that the argument is an intmax_t or uintmax_t.
        With n, specifies that the argument is a pointer to an intmax_t.
    z   With d, i, o, u, x, or X, specifies that the argument is a ssize_t or size_t.
        With n, specifies that the argument is a pointer to a ssize_t.
    t   With d, i, o, u, x, or X, specifies that the argument is a ptrdiff_t.
        With n, specifies that the argument is a pointer to a ptrdiff_t.
    L   With a, A, e, E, f, F, g, or G, specifies that the argument is a long double.

• type

type specifies what kind of conversion printf performs. Here is a table of these:

    %  Prints the percent character (%).
    c  Prints arg as single character. If the l size specifier is in effect, a multibyte character is printed.
    C  Short for %lc.
    s  Prints the elements of a pointer to char until the precision or a null character is reached. If the l size specifier is in effect, the pointer is to an array of wchar_t, and the string is converted to multibyte characters before printing.
S  Short for `%ls`.

d or i  Prints a signed decimal integer; takes an `int`. Leading zeros are inserted as necessary to reach the precision. A precision of 0 produces an empty string.

D  Newlib extension, short for `%ld`.

o  Prints an unsigned octal integer; takes an `unsigned`. Leading zeros are inserted as necessary to reach the precision. A precision of 0 produces an empty string.

O  Newlib extension, short for `%lo`.

u  Prints an unsigned decimal integer; takes an `unsigned`. Leading zeros are inserted as necessary to reach the precision. A precision of 0 produces an empty string.

U  Newlib extension, short for `%lu`.

x  Prints an unsigned hexadecimal integer (using `abcdef` as digits beyond 9); takes an `unsigned`. Leading zeros are inserted as necessary to reach the precision. A precision of 0 produces an empty string.

X  Like x, but uses `ABCDEF` as digits beyond 9.

f  Prints a signed value of the form `[-]9999.9999`, with the precision determining how many digits follow the decimal point; takes a `double` (remember that `float` promotes to `double` as a vararg). The low order digit is rounded to even. If the precision results in at most DECIMAL_DIG digits, the result is rounded correctly; if more than DECIMAL_DIG digits are printed, the result is only guaranteed to round back to the original value.

If the value is infinite, the result is `inf`, and no zero padding is performed. If the value is not a number, the result is `nan`, and no zero padding is performed.

F  Like f, but uses `INF` and `NAN` for non-finite numbers.

e  Prints a signed value of the form `[-]9.9999e[+|−]999`; takes a `double`. The digit before the decimal point is non-zero if the value is non-zero. The precision determines how many digits appear between . and e, and the exponent always contains at least two digits. The value zero has an exponent of zero. If the value is not finite, it is printed like f.

E  Like e, but using E to introduce the exponent, and like F for non-finite values.

g  Prints a signed value in either f or e form, based on the given value and precision—an exponent less than -4 or greater than the precision selects the e form. Trailing zeros and the decimal point are printed only if necessary; takes a `double`.

G  Like g, except use F or E form.
a Prints a signed value of the form \([-\]0x1.ffffp[+\-]9; takes a double. The letters abcdef are used for digits beyond 9. The precision determines how many digits appear after the decimal point. The exponent contains at least one digit, and is a decimal value representing the power of 2; a value of 0 has an exponent of 0. Non-finite values are printed like f.

A Like a, except uses X, P, and ABCDEF instead of lower case.

n Takes a pointer to int, and stores a count of the number of bytes written so far. No output is created.

p Takes a pointer to void, and prints it in an implementation-defined format. This implementation is similar to \%#tx), except that 0x appears even for the NULL pointer.

Printf_r, fprintf_r, asprintf_r, sprintf_r, snprintf_r, asnprintf_r are simply reentrant versions of the functions above.

Returns
On success, sprintf and asprintf return the number of bytes in the output string, except the concluding NULL is not counted. snprintf returns the number of bytes that would be in the output string, except the concluding NULL is not counted. printf and fprintf return the number of characters transmitted. asprintf returns the original str if there was enough room, otherwise it returns an allocated string.

If an error occurs, the result of printf, fprintf, snprintf, and asprintf is a negative value, and the result of asprintf is NULL. No error returns occur for sprintf. For printf and fprintf, errno may be set according to fputc. For asprintf and asnprintf, errno may be set to ENOMEM if allocation fails, and for snprintf, errno may be set to EOVERFLOW if size or the output length exceeds INT_MAX.

Portability
ANSI C requires printf, fprintf, sprintf, and snprintf. asprintf and asnprintf are newlib extensions.

The ANSI C standard specifies that implementations must support at least formatted output of up to 509 characters. This implementation has no inherent limit.

Depending on how newlib was configured, not all format specifiers are supported.

Supporting OS subroutines required: close, fstat, isatty, lseek, read, sbrk, write.
4.60 sscanf, fscanf, scanf—scan and format input

Synopsis

```c
#include <stdio.h>

int scanf(const char *format, ...);
int fscanf(FILE *fd, const char *format, ...);
int sscanf(const char *str, const char *format, ...);

int _scanf_r(struct _reent *ptr, const char *format, ...);
int _fscanf_r(struct _reent *ptr, FILE *fd, const char *format, ...);
int _sscanf_r(struct _reent *ptr, const char *str, const char *format, ...);
```

Description

`scanf` scans a series of input fields from standard input, one character at a time. Each field is interpreted according to a format specifier passed to `scanf` in the format string at `*format`. `scanf` stores the interpreted input from each field at the address passed to it as the corresponding argument following `format`. You must supply the same number of format specifiers and address arguments as there are input fields.

There must be sufficient address arguments for the given format specifiers; if not the results are unpredictable and likely disastrous. Excess address arguments are merely ignored.

`scanf` often produces unexpected results if the input diverges from an expected pattern. Since the combination of `gets` or `fgets` followed by `sscanf` is safe and easy, that is the preferred way to be certain that a program is synchronized with input at the end of a line.

`fscanf` and `sscanf` are identical to `scanf`, other than the source of input: `fscanf` reads from a file, and `sscanf` from a string.

The routines `_scanf_r`, `_fscanf_r`, and `_sscanf_r` are reentrant versions of `scanf`, `fscanf`, and `sscanf` that take an additional first argument pointing to a reentrancy structure.

The string at `*format` is a character sequence composed of zero or more directives. Directives are composed of one or more whitespace characters, non-whitespace characters, and format specifications.

Whitespace characters are blank ( ), tab (\t), or newline (\n). When `scanf` encounters a whitespace character in the format string it will read (but not store) all consecutive whitespace characters up to the next non-whitespace character in the input.

Non-whitespace characters are all other ASCII characters except the percent sign (%). When `scanf` encounters a non-whitespace character in the format string it will read, but not store a matching non-whitespace character.

Format specifications tell `scanf` to read and convert characters from the input field into specific types of values, and store then in the locations specified by the address arguments.

Trailing whitespace is left unread unless explicitly matched in the format string.

The format specifiers must begin with a percent sign (%) and have the following form:

`%[+]*[width][size]type`

Each format specification begins with the percent character (%). The other fields are:
* an optional marker; if present, it suppresses interpretation and assignment of this input field.

**width**

an optional maximum field width: a decimal integer, which controls the maximum number of characters that will be read before converting the current input field. If the input field has fewer than width characters, scanf reads all the characters in the field, and then proceeds with the next field and its format specification.

If a whitespace or a non-convertable character occurs before width character are read, the characters up to that character are read, converted, and stored. Then scanf proceeds to the next format specification.

**size**

h, j, l, L, t, and z are optional size characters which override the default way that scanf interprets the data type of the corresponding argument.

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Type(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>hh</td>
<td>d, i, o, u, x, n convert input to char, store in char object</td>
</tr>
<tr>
<td>h</td>
<td>d, i, o, u, x, n convert input to short, store in short object</td>
</tr>
<tr>
<td>h</td>
<td>D, I, O, U, X no effect</td>
</tr>
<tr>
<td>e, f, c, s, p</td>
<td>j d, i, o, u, x, n convert input to intmax_t, store in intmax_t object</td>
</tr>
<tr>
<td>j</td>
<td>all others no effect</td>
</tr>
<tr>
<td>l</td>
<td>d, i, o, u, x, n convert input to long, store in long object</td>
</tr>
<tr>
<td>e, f, g</td>
<td>convert input to double store in a double object</td>
</tr>
<tr>
<td>l</td>
<td>D, I, O, U, X no effect</td>
</tr>
<tr>
<td>c, s, p</td>
<td>l d, i, o, u, x, n convert to long long, store in long long</td>
</tr>
<tr>
<td>ll</td>
<td>d, i, o, u, x, n convert to long long, store in long long</td>
</tr>
<tr>
<td>L</td>
<td>e, f, g, E, G convert to long double, store in long double</td>
</tr>
<tr>
<td>L</td>
<td>all others no effect</td>
</tr>
<tr>
<td>t</td>
<td>d, i, o, u, x, n convert input to ptrdiff_t, store in ptrdiff_t object</td>
</tr>
<tr>
<td>t</td>
<td>all others no effect</td>
</tr>
<tr>
<td>z</td>
<td>d, i, o, u, x, n convert input to size_t, store in size_t object</td>
</tr>
</tbody>
</table>
Chapter 4: Input and Output ('stdio.h')

z all others no effect

type
A character to specify what kind of conversion scanf performs. Here is a table of the conversion characters:

% No conversion is done; the percent character (%) is stored.
c Scans one character. Corresponding arg: (char *arg).
s Reads a character string into the array supplied. Corresponding arg: (char arg[]).
[pattern] Reads a non-empty character string into memory starting at arg. This area must be large enough to accept the sequence and a terminating null character which will be added automatically. (pattern is discussed in the paragraph following this table). Corresponding arg: (char *arg).
d Reads a decimal integer into the corresponding arg: (int *arg).
D Reads a decimal integer into the corresponding arg: (long *arg).
o Reads an octal integer into the corresponding arg: (int *arg).
O Reads an octal integer into the corresponding arg: (long *arg).
u Reads an unsigned decimal integer into the corresponding arg: (unsigned int *arg).
U Reads an unsigned decimal integer into the corresponding arg: (unsigned long *arg).
x, X Read a hexadecimal integer into the corresponding arg: (int *arg).
e, f, g Read a floating-point number into the corresponding arg: (float *arg).
E, F, G Read a floating-point number into the corresponding arg: (double *arg).
i Reads a decimal, octal or hexadecimal integer into the corresponding arg: (int *arg).
I Reads a decimal, octal or hexadecimal integer into the corresponding arg: (long *arg).
n Stores the number of characters read in the corresponding arg: (int *arg).
p Stores a scanned pointer. ANSI C leaves the details to each implementation; this implementation treats %p exactly the same as %U. Corresponding arg: (void **arg).
A pattern of characters surrounded by square brackets can be used instead of the s type character. pattern is a set of characters which define a search set of possible characters making up the scanf input field. If the first character in the brackets is a caret (^), the search set is inverted to include all ASCII characters except those between the brackets. There is also a range facility which you can use as a shortcut. %[0-9] matches all decimal digits. The hyphen must not be the first or last character in the set. The character prior to the hyphen must be lexically less than the character after it.

Here are some pattern examples:

- %[abcd] matches strings containing only a, b, c, and d.
- %[^abcd] matches strings containing any characters except a, b, c, or d.
- %[z-a] matches the characters z, -, and a.

Floating point numbers (for field types e, f, g, E, F, G) must correspond to the following general form:

```
[+/-] ddddd[.]ddd [E|e[+|-]ddd]
```

where objects inclosed in square brackets are optional, and ddd represents decimal, octal, or hexadecimal digits.

**Returns**

scanf returns the number of input fields successfully scanned, converted and stored; the return value does not include scanned fields which were not stored.

If scanf attempts to read at end-of-file, the return value is E0F.

If no fields were stored, the return value is 0.

scanf might stop scanning a particular field before reaching the normal field end character, or may terminate entirely.

scanf stops scanning and storing the current field and moves to the next input field (if any) in any of the following situations:

- The assignment suppressing character (*) appears after the % in the format specification; the current input field is scanned but not stored.
- width characters have been read (width is a width specification, a positive decimal integer).
- The next character read cannot be converted under the the current format (for example, if a Z is read when the format is decimal).
- The next character in the input field does not appear in the search set (or does appear in the inverted search set).

When scanf stops scanning the current input field for one of these reasons, the next character is considered unread and used as the first character of the following input field, or the first character in a subsequent read operation on the input.

scanf will terminate under the following circumstances:
• The next character in the input field conflicts with a corresponding non-whitespace character in the format string.
• The next character in the input field is EOF.
• The format string has been exhausted.

When the format string contains a character sequence that is not part of a format specification, the same character sequence must appear in the input; \texttt{scanf} will scan but not store the matched characters. If a conflict occurs, the first conflicting character remains in the input as if it had never been read.

\textbf{Portability}

\texttt{scanf} is ANSI C.

Supporting OS subroutines required: \texttt{close, fstat, isatty, lseek, read, sbrk, write}.
4.61 tmpfile—create a temporary file

Synopsis

```c
#include <stdio.h>
FILE *tmpfile(void);

FILE *tmpfile_r(struct _reent *reent);
```

Description

Create a temporary file (a file which will be deleted automatically), using a name generated by `tmpnam`. The temporary file is opened with the mode "wb+", permitting you to read and write anywhere in it as a binary file (without any data transformations the host system may perform for text files).

The alternate function `_tmpfile_r` is a reentrant version. The argument `reent` is a pointer to a reentrancy structure.

Returns

`tmpfile` normally returns a pointer to the temporary file. If no temporary file could be created, the result is NULL, and `errno` records the reason for failure.

Portability

Both ANSI C and the System V Interface Definition (Issue 2) require `tmpfile`.

Supporting OS subroutines required: `close`, `fstat`, `getpid`, `isatty`, `lseek`, `open`, `read`, `sbrk`, `write`.

`tmpfile` also requires the global pointer `environ`.
4.62 tmpnam, tempnam—name for a temporary file

Synopsis

```c
#include <stdio.h>
char *tmpnam(char *s);
char *tempnam(char *dir, char *pfx);
char *_tmpnam_r(struct _reent *reent, char *s);
char *_tempnam_r(struct _reent *reent, char *dir, char *pfx);
```

Description

Use either of these functions to generate a name for a temporary file. The generated name is guaranteed to avoid collision with other files (for up to TMP_MAX calls of either function). tmpnam generates file names with the value of P_tmpdir (defined in `stdio.h`) as the leading directory component of the path.

You can use the tmpnam argument `s` to specify a suitable area of memory for the generated filename; otherwise, you can call tmpnam(NULL) to use an internal static buffer.

tempnam allows you more control over the generated filename: you can use the argument `dir` to specify the path to a directory for temporary files, and you can use the argument `pfx` to specify a prefix for the base filename.

If `dir` is NULL, tempnam will attempt to use the value of environment variable TMPDIR instead; if there is no such value, tempnam uses the value of P_tmpdir (defined in `stdio.h`).

If you don’t need any particular prefix to the basename of temporary files, you can pass NULL as the `pfx` argument to tempnam.

_tempnam_r and _tempnam_r are reentrant versions of tmpnam and tempnam respectively. The extra argument `reent` is a pointer to a reentrancy structure.

Warnings

The generated filenames are suitable for temporary files, but do not in themselves make files temporary. Files with these names must still be explicitly removed when you no longer want them.

If you supply your own data area `s` for tmpnam, you must ensure that it has room for at least L_tmpnam elements of type char.

Returns

Both tmpnam and tempnam return a pointer to the newly generated filename.

Portability

ANSI C requires tmpnam, but does not specify the use of P_tmpdir. The System V Interface Definition (Issue 2) requires both tmpnam and tempnam.

Supporting OS subroutines required: close, fstat, getpid, isatty, lseek, open, read, sbrk, write.

The global pointer environ is also required.
4.63 `ungetc`—push data back into a stream

Synopsis

```c
#include <stdio.h>
int ungetc(int c, FILE *stream);

int _ungetc_r(struct _reent *reent, int c, FILE *stream);
```

Description

`ungetc` is used to return bytes back to `stream` to be read again. If `c` is EOF, the stream is unchanged. Otherwise, the unsigned char `c` is put back on the stream, and subsequent reads will see the bytes pushed back in reverse order. Pushed bytes are lost if the stream is repositioned, such as by `fseek`, `fsetpos`, or `rewind`.

The underlying file is not changed, but it is possible to push back something different than what was originally read. Ungetting a character will clear the end-of-stream marker, and decrement the file position indicator. Pushing back beyond the beginning of a file gives unspecified behavior.

The alternate function `_ungetc_r` is a reentrant version. The extra argument `reent` is a pointer to a reentrancy structure.

Returns

The character pushed back, or EOF on error.

Portability

ANSI C requires `ungetc`, but only requires a pushback buffer of one byte; although this implementation can handle multiple bytes, not all can. Pushing back a signed char is a common application bug.

Supporting OS subroutines required: `sbrk`. 
4.64 ungetwc—push wide character data back into a stream

Synopsis

```c
#include <stdio.h>
#include <wchar.h>
wint_t ungetwc(wint_t wc, FILE *stream);
wint_t _ungetwc_r(struct _reent *reent, wint_t wc, FILE *stream);
```

Description

ungetwc is used to return wide characters back to stream to be read again. If wc is WEOF, the stream is unchanged. Otherwise, the wide character wc is put back on the stream, and subsequent reads will see the wide chars pushed back in reverse order. Pushed wide chars are lost if the stream is repositioned, such as by fseek, fsetpos, or rewind.

The underlying file is not changed, but it is possible to push back something different than what was originally read. Ungetting a character will clear the end-of-stream marker, and decrement the file position indicator. Pushing back beyond the beginning of a file gives unspecified behavior.

The alternate function _ungetwc_r is a reentrant version. The extra argument reent is a pointer to a reentrancy structure.

Returns

The wide character pushed back, or WEOF on error.

Portability

C99
4.65 vfprintf, vprintf, vsprintf, vsnprintf, vasprintf, vasnprintf—format argument list

Synopsis

#include <stdio.h>
#include <stdarg.h>

int vprintf(const char *fmt, va_list list);
int vfprintf(FILE *fp, const char *fmt, va_list list);
int vsprintf(char *str, const char *fmt, va_list list);
int vsnprintf(char *str, size_t size, const char *fmt, va_list list);

int vasprintf(char **strp, const char *fmt, va_list list);
char *vasnprintf(char *str, size_t *size, const char *fmt, va_list list);

int _vprintf_r(struct _reent *reent, const char *fmt, va_list list);
int _vfprintf_r(struct _reent *reent, FILE *fp, const char *fmt, va_list list);
int _vsprintf_r(struct _reent *reent, char *str, const char *fmt, va_list list);
int _vsnprintf_r(struct _reent *reent, char *str, size_t size, const char *fmt, va_list list);

Description

vprintf, vfprintf, vasprintf, vsprintf, vsnprintf, and vasprintf are (respectively) variants of printf, fprintf, asprintf, sprintf, snprintf, and asnprintf. They differ only in allowing their caller to pass the variable argument list as a va_list object (initialized by va_start) rather than directly accepting a variable number of arguments. The caller is responsible for calling va_end.

_vprintf_r, _vfprintf_r, _vasprintf_r, _vsprintf_r, _vsnprintf_r, and _vasnprintf_r are reentrant versions of the above.

Returns

The return values are consistent with the corresponding functions.

Portability

ANSI C requires vprintf, vfprintf, vsprintf, and vsnprintf. The remaining functions are newlib extensions.

Supporting OS subroutines required: close, fstat, isatty, lseek, read, sbrk, write.
4.66 \texttt{vfscanf, vscanf, vsscanf}—format argument list

Synopsis

\begin{verbatim}
#include <stdio.h>
#include <stdarg.h>
int vscanf(const char *fmt, va_list list);
int vfscanf(FILE *fp, const char *fmt, va_list list);
int vsscanf(const char *str, const char *fmt, va_list list);

int _vscanf_r(struct _reent *reent, const char *fmt, va_list list);
int _vfscanf_r(struct _reent *reent, FILE *fp, const char *fmt, va_list list);
int _vsscanf_r(struct _reent *reent, const char *str, const char *fmt, va_list list);
\end{verbatim}

Description

\texttt{vscanf}, \texttt{vfscanf}, and \texttt{vsscanf} are (respectively) variants of \texttt{scanf}, \texttt{fscanf}, and \texttt{sscanf}. They differ only in allowing their caller to pass the variable argument list as a \texttt{va_list} object (initialized by \texttt{va_start}) rather than directly accepting a variable number of arguments.

Returns

The return values are consistent with the corresponding functions: \texttt{vscanf} returns the number of input fields successfully scanned, converted, and stored; the return value does not include scanned fields which were not stored.

If \texttt{vscanf} attempts to read at end-of-file, the return value is EOF.

If no fields were stored, the return value is 0.

The routines \texttt{_vscanf_r}, \texttt{_vfscanf_r}, and \texttt{_vsscanf_r} are reentrant versions which take an additional first parameter which points to the reentrancy structure.

Portability

These are GNU extensions.

Supporting OS subroutines required:
4.67viprintf, vfiprintf, vsiprintf, vsnprintf, vasiprintf, 
vsniprintf—format argument list (integer only)

Synopsis

```
#include <stdio.h>
#include <stdarg.h>
int viprintf(const char *fmt, va_list list);
int vfiprintf(FILE *fp, const char *fmt, va_list list);
int vsiprintf(char *str, const char *fmt, va_list list);
int vsnprintf(char *str, size_t size, const char *fmt, va_list list);
int vasiprintf(char **strp, const char *fmt, va_list list);
char *vasnpnprintf(char *str, size_t *size, const char *fmt, va_list list);
```

Description

viprintf, vfiprintf, vasiprintf, vsiprintf, vsnprintf, and vsnprintf are (respectively) variants of iprintf, fiprintf, asiprintf, siprintf, snprintf, and asnpnprintf. They differ only in allowing their caller to pass the variable argument list as a va_list object (initialized by va_start) rather than directly accepting a variable number of arguments. The caller is responsible for calling va_end.

_viprintf_r, _vfiprintf_r, _vasiprintf_r, _vsiprintf_r, _vsnprintf_r, and 
_vasnpnprintf_r are reentrant versions of the above.

Returns

The return values are consistent with the corresponding functions:

Portability

All of these functions are newlib extensions.

Supporting OS subroutines required: close, fstat, isatty, lseek, read, sbrk, write.
4.68 viscanf, vfiscanf, vsiscanf—format argument list

Synopsis

```c
#include <stdio.h>
#include <stdarg.h>
int viscanf(const char *fmt, va_list list);
int vfiscanf(FILE *fp, const char *fmt, va_list list);
int vsiscanf(const char *str, const char *fmt, va_list list);
int _viscanf_r(struct _reent *reent, const char *fmt, va_list list);
int _vfiscanf_r(struct _reent *reent, FILE *fp, const char *fmt, va_list list);
int _vsiscanf_r(struct _reent *reent, const char *str, const char *fmt, va_list list);
```

Description

viscanf, vfiscanf, and vsiscanf are (respectively) variants of iscanf, fiscanf, and siscanf. They differ only in allowing their caller to pass the variable argument list as a va_list object (initialized by va_start) rather than directly accepting a variable number of arguments.

Returns

The return values are consistent with the corresponding functions: viscanf returns the number of input fields successfully scanned, converted, and stored; the return value does not include scanned fields which were not stored.

If viscanf attempts to read at end-of-file, the return value is EOF.

If no fields were stored, the return value is 0.

The routines _viscanf_r, _vfiscanf_r, and _vsiscanf_r are reentrant versions which take an additional first parameter which points to the reentrancy structure.

Portability

These are newlib extensions.

Supporting OS subroutines required:
5 Strings and Memory (‘string.h’)

This chapter describes string-handling functions and functions for managing areas of memory. The corresponding declarations are in ‘string.h’.
5.1 **bcmp—compare two memory areas**

**Synopsis**

```c
#include <string.h>
int bcmp(const void *s1, const void *s2, size_t n);
```

**Description**

This function compares not more than \( n \) bytes of the object pointed to by \( s1 \) with the object pointed to by \( s2 \).

This function is identical to `memcmp`.

**Returns**

The function returns an integer greater than, equal to or less than zero according to whether the object pointed to by \( s1 \) is greater than, equal to or less than the object pointed to by \( s2 \).

**Portability**

`bcmp` requires no supporting OS subroutines.
5.2 `bcopy`—copy memory regions

Synopsis

```c
#include <string.h>
void bcopy(const void *in, void *out, size_t n);
```

Description

This function copies \(n\) bytes from the memory region pointed to by `in` to the memory region pointed to by `out`.

This function is implemented in term of `memmove`.

Portability

`bcopy` requires no supporting OS subroutines.
5.3 bzero—initialize memory to zero

Synopsis

```c
#include <string.h>
void bzero(void *b, size_t length);
```

Description

`bzero` initializes `length` bytes of memory, starting at address `b`, to zero.

Returns

`bzero` does not return a result.

Portability

`bzero` is in the Berkeley Software Distribution. Neither ANSI C nor the System V Interface Definition (Issue 2) require `bzero`. `bzero` requires no supporting OS subroutines.
5.4 index—search for character in string

Synopsis
#include <string.h>
char * index(const char *string, int c);

Description
This function finds the first occurrence of \texttt{c} (converted to a char) in the string pointed to by \texttt{string} (including the terminating null character).

This function is identical to \texttt{strstr}.

Returns
Returns a pointer to the located character, or a null pointer if \texttt{c} does not occur in \texttt{string}.

Portability
\texttt{index} requires no supporting OS subroutines.
5.5 memccpy—copy memory regions with end-token check

Synopsis

```c
#include <string.h>
void* memccpy(void *out, const void *in, int endchar, size_t n);
```

Description

This function copies up to \( n \) bytes from the memory region pointed to by \( in \) to the memory region pointed to by \( out \). If a byte matching the \( endchar \) is encountered, the byte is copied and copying stops.

If the regions overlap, the behavior is undefined.

Returns

\texttt{memccpy} returns a pointer to the first byte following the \textit{endchar} in the \textit{out} region. If no byte matching \textit{endchar} was copied, then \texttt{NULL} is returned.

Portability

\texttt{memccpy} is a GNU extension.

\texttt{memccpy} requires no supporting OS subroutines.
5.6 **memchr**—find character in memory

**Synopsis**
```
#include <string.h>
void *memchr(const void *src, int c, size_t length);
```

**Description**
This function searches memory starting at *src* for the character *c*. The search only ends with the first occurrence of *c*, or after *length* characters; in particular, NUL does not terminate the search.

**Returns**
If the character *c* is found within *length* characters of *src*, a pointer to the character is returned. If *c* is not found, then NULL is returned.

**Portability**
memchr is ANSI C.

memchr requires no supporting OS subroutines.
5.7 memcmp—compare two memory areas

Synopsis

```c
#include <string.h>
int memcmp(const void *s1, const void *s2, size_t n);
```

Description

This function compares not more than \( n \) characters of the object pointed to by \( s1 \) with the object pointed to by \( s2 \).

Returns

The function returns an integer greater than, equal to or less than zero according to whether the object pointed to by \( s1 \) is greater than, equal to or less than the object pointed to by \( s2 \).

Portability

`memcmp` is ANSI C.

`memcmp` requires no supporting OS subroutines.
5.8 memcpy—copy memory regions

Synopsis

```
#include <string.h>
void* memcpy(void *out, const void *in, size_t n);
```

Description

This function copies \( n \) bytes from the memory region pointed to by \( in \) to the memory region pointed to by \( out \).

If the regions overlap, the behavior is undefined.

Returns

\( \text{memcpy} \) returns a pointer to the first byte of the \( out \) region.

Portability

\( \text{memcpy} \) is ANSI C.

\( \text{memcpy} \) requires no supporting OS subroutines.
5.9 memmem—find memory segment

Synopsis

```
#include <string.h>
char *memmem(const void *s1, size_t l1, const void *s2, size_t l2);
```

Description

Locates the first occurrence in the memory region pointed to by s1 with length l1 of the sequence of bytes pointed to by s2 of length l2. If you already know the lengths of your haystack and needle, memmem can be much faster than strstr.

Returns

Returns a pointer to the located segment, or a null pointer if s2 is not found. If l2 is 0, s1 is returned.

Portability

memmem is a newlib extension.

memmem requires no supporting OS subroutines.
5.10 memmove—move possibly overlapping memory

Synopsis

```c
#include <string.h>

void *memmove(void *dst, const void *src, size_t length);
```

Description

This function moves `length` characters from the block of memory starting at `*src` to the memory starting at `*dst`. `memmove` reproduces the characters correctly at `*dst` even if the two areas overlap.

Returns

The function returns `dst` as passed.

Portability

`memmove` is ANSI C.

`memmove` requires no supporting OS subroutines.
5.11 `mempcpy`—copy memory regions and return end pointer

Synopsis

```c
#include <string.h>
void* mempcpy(void *out, const void *in, size_t n);
```

Description

This function copies `n` bytes from the memory region pointed to by `in` to the memory region pointed to by `out`.

If the regions overlap, the behavior is undefined.

Returns

`mempcpy` returns a pointer to the byte following the last byte copied to the `out` region.

Portability

`mempcpy` is a GNU extension.

`mempcpy` requires no supporting OS subroutines.
5.12 memset—set an area of memory

Synopsis

```
#include <string.h>
void *memset(void *dst, int c, size_t length);
```

Description

This function converts the argument `c` into an unsigned char and fills the first `length` characters of the array pointed to by `dst` to the value.

Returns

`memset` returns the value of `dst`.

Portability

`memset` is ANSI C.

`memset` requires no supporting OS subroutines.
5.13 rindex—reverse search for character in string

Synopsis

```
#include <string.h>
char * rindex(const char *string, int c);
```

Description
This function finds the last occurrence of \texttt{c} (converted to a char) in the string pointed to by \texttt{string} (including the terminating null character).

This function is identical to \texttt{strrchr}.

Returns
Returns a pointer to the located character, or a null pointer if \texttt{c} does not occur in \texttt{string}.

Portability
\texttt{rindex} requires no supporting OS subroutines.
5.14 stpcpy—copy string returning a pointer to its end

Synopsis

```
#include <string.h>
char *stpcpy(char *dst, const char *src);
```

Description

stpcpy copies the string pointed to by src (including the terminating null character) to the array pointed to by dst.

Returns

This function returns a pointer to the end of the destination string, thus pointing to the trailing '\0'.

Portability

stpcpy is a GNU extension, candidate for inclusion into POSIX/SUSv4.

stpcpy requires no supporting OS subroutines.
5.15 stpncpy—counted copy string returning a pointer to its end

Synopsis

```c
#include <string.h>
char *stpncpy(char *dst, const char *src, size_t length);
```

Description

stpncpy copies not more than length characters from the string pointed to by src (including the terminating null character) to the array pointed to by dst. If the string pointed to by src is shorter than length characters, null characters are appended to the destination array until a total of length characters have been written.

Returns

This function returns a pointer to the end of the destination string, thus pointing to the trailing '\0', or, if the destination string is not null-terminated, pointing to dst + n.

Portability

stpncpy is a GNU extension, candidate for inclusion into POSIX/SUSv4.

stpncpy requires no supporting OS subroutines.
5.16 strcasecmp—case-insensitive character string compare

Synopsis

```
#include <string.h>
int strcasecmp(const char *a, const char *b);
```

Description

strcasecmp compares the string at a to the string at b in a case-insensitive manner.

Returns

If *a sorts lexicographically after *b (after both are converted to uppercase), strcasecmp
returns a number greater than zero. If the two strings match, strcasecmp returns zero. If
*a sorts lexicographically before *b, strcasecmp returns a number less than zero.

Portability

strncasecmp is in the Berkeley Software Distribution.

strncasecmp requires no supporting OS subroutines. It uses tolower() from elsewhere in this
library.
5.17 strcasestr—case-insensitive character string search

Synopsis

```
#include <string.h>
char *strcasestr(const char *s, const char *find);
```

Description

strcasestr searches the string `s` for the first occurrence of the sequence `find`. strcasestr is identical to strstr except the search is case-insensitive.

Returns

A pointer to the first case-insensitive occurrence of the sequence `find` or NULL if no match was found.

Portability

strcasestr is in the Berkeley Software Distribution.

strcasestr requires no supporting OS subroutines. It uses tolower() from elsewhere in this library.
5.18 strcat—concatenate strings

Synopsis

```c
#include <string.h>
char *strcat(char *dst, const char *src);
```

Description

`strcat` appends a copy of the string pointed to by `src` (including the terminating null character) to the end of the string pointed to by `dst`. The initial character of `src` overwrites the null character at the end of `dst`.

Returns

This function returns the initial value of `dst`.

Portability

`strcat` is ANSI C.

`strcat` requires no supporting OS subroutines.
5.19 strchr—search for character in string

Synopsis

#include <string.h>
char * strchr(const char *string, int c);

Description
This function finds the first occurrence of c (converted to a char) in the string pointed to by string (including the terminating null character).

Returns
Returns a pointer to the located character, or a null pointer if c does not occur in string.

Portability
strchr is ANSI C.
strchr requires no supporting OS subroutines.
5.20 strcmp—character string compare

Synopsis

```c
#include <string.h>
int strcmp(const char *a, const char *b);
```

Description

`strcmp` compares the string at `a` to the string at `b`.

Returns

If `*a` sorts lexicographically after `*b`, `strcmp` returns a number greater than zero. If the two strings match, `strcmp` returns zero. If `*a` sorts lexicographically before `*b`, `strcmp` returns a number less than zero.

Portability

`strcmp` is ANSI C.

`strcmp` requires no supporting OS subroutines.
5.21 `strcoll’—locale-specific character string compare

Synopsis

```
#include <string.h>
int strcoll(const char *strA, const char *strB);
```

Description

`strcoll’ compares the string pointed to by `strA’ to the string pointed to by `strB’, using an interpretation appropriate to the current `LC_COLLATE’ state.

Returns

If the first string is greater than the second string, `strcoll’ returns a number greater than zero. If the two strings are equivalent, `strcoll’ returns zero. If the first string is less than the second string, `strcoll’ returns a number less than zero.

Portability

`strcoll’ is ANSI C.

`strcoll’ requires no supporting OS subroutines.
5.22 strcpy—copy string

Synopsis

```
#include <string.h>
char *strcpy(char *dst, const char *src);
```

Description

`strcpy` copies the string pointed to by `src` (including the terminating null character) to the array pointed to by `dst`.

Returns

This function returns the initial value of `dst`.

Portability

`strcpy` is ANSI C.

`strcpy` requires no supporting OS subroutines.
5.23 strcspn—count characters not in string

Synopsis

    size_t strcspn(const char *s1, const char *s2);

Description
This function computes the length of the initial part of the string pointed to by s1 which consists entirely of characters NOT from the string pointed to by s2 (excluding the terminating null character).

Returns
strcspn returns the length of the substring found.

Portability
strcspn is ANSI C.
strcspn requires no supporting OS subroutines.
5.24 `strerror`—convert error number to string

Synopsis

```c
#include <string.h>
char *strerror(int errnum);
```

Description

`strerror` converts the error number `errnum` into a string. The value of `errnum` is usually a copy of `errno`. If `errnum` is not a known error number, the result points to an empty string.

This implementation of `strerror` prints out the following strings for each of the values defined in `errno.h`:

- E2BIG: Arg list too long
- EACCES: Permission denied
- EADDRINUSE: Address already in use
- EADV: Advertise error
- EAFNOSUPPORT: Address family not supported by protocol family
- EAGAIN: No more processes
- EALREADY: Socket already connected
- EBADF: Bad file number
- EBADMSG: Bad message
- EBUSY: Device or resource busy
- ECHILD: No children
- ECOMM: Communication error
- ECONNABORTED: Software caused connection abort
- ECONNREFUSED: Connection refused
- EDEADLK: Deadlock
- EDESTADDRREQ: Destination address required
- EEXIST: File exists
- EDOM: Math argument
- EFAULT: Bad address
- EFBIG: File too large
- EHOSTDOWN: Host is down
EHOSTUNREACH
   Host is unreachable
EIDRM
   Identifier removed
EINPROGRESS
   Connection already in progress
EINTR
   Interrupted system call
EINVAL
   Invalid argument
EIO
   I/O error
EISCONN
   Socket is already connected
EISDIR
   Is a directory
ELIBACC
   Cannot access a needed shared library
ELIBBAD
   Accessing a corrupted shared library
ELIBEXEC
   Cannot exec a shared library directly
ELIBMAX
   Attempting to link in more shared libraries than system limit
ELIBSCN
   .lib section in a.out corrupted
EMFILE
   Too many open files
EMLINK
   Too many links
EMSGSIZE
   Message too long
EMULTIHOP
   Multihop attempted
ENAMETOOLONG
   File or path name too long
ENETDOWN
   Network interface not configured
ENETUNREACH
   Network is unreachable
ENFILE
   Too many open files in system
ENODEV
   No such device
ENOENT
   No such file or directory
ENOEXEC
   Exec format error
ENOLCK
   No lock
ENOLINK
   Virtual circuit is gone
ENOMEM
   Not enough space
ENOMEM
   Not enough space
ENOMSG
   No message of desired type
ENONET
   Machine is not on the network
ENOPKG      No package
ENOPROTOOPT Protocol not available
ENOSPC      No space left on device
ENOSR       No stream resources
ENOSTR      Not a stream
ENOSYS      Function not implemented
ENOTBLK     Block device required
ENOTCONN    Socket is not connected
ENOTDIR     Not a directory
ENOTEMPTY   Directory not empty
ENOTSOCK    Socket operation on non-socket
ENOTSUP     Not supported
ENOTTY      Not a character device
ENXIO       No such device or address
EPERM       Not owner
EPIPE       Broken pipe
EPROTO      Protocol error
EPROTOTYPE   Protocol wrong type for socket
EPROTONOSUPPORT Unknown protocol
ERANGE      Result too large
EREMOTE     Resource is remote
EROFS       Read-only file system
ESHUTDOWN    Can’t send after socket shutdown
ESOCKTNOSUPPORT Socket type not supported
ESPIPE      Illegal seek
ESRCH       No such process
ESRMNT      Srmount error
ETIME       Stream ioctl timeout
ETIMEDOUT  
Connection timed out
ETXTBSY  
Text file busy
EXDEV  
Cross-device link
ECANCELED  
Operation canceled
ENOTRECOVERABLE  
State not recoverable
EOWNERDEAD  
Previous owner died

Returns
This function returns a pointer to a string. Your application must not modify that string.

Portability
ANSI C requires `strerror`, but does not specify the strings used for each error number.
Although this implementation of `strerror` is reentrant, ANSI C declares that subsequent calls to `strerror` may overwrite the result string; therefore portable code cannot depend on the reentrancy of this subroutine.
This implementation of `strerror` provides for user-defined extensibility. `errno.h` defines `ELASTERROR`, which can be used as a base for user-defined error values. If the user supplies a routine named `user_strerror`, and `errno` passed to `strerror` does not match any of the supported values, `user_strerror` is called with `errno` as its argument.
`user_strerror` takes one argument of type `int`, and returns a character pointer. If `errno` is unknown to `user_strerror`, `user_strerror` returns `NULL`. The default `user_strerror` returns `NULL` for all input values.
`strerror` requires no supporting OS subroutines.
5.25 `strerror_r`—convert error number to string and copy to buffer

Synopsis

```
#include <string.h>
char *strerror_r(int errnum, char *buffer, size_t n);
```

Description

`strerror_r` converts the error number `errnum` into a string and copies the result into the supplied `buffer` for a length up to `n`, including the NUL terminator. The value of `errnum` is usually a copy of `errno`. If `errnum` is not a known error number, the result is the empty string.

See `strerror` for how strings are mapped to `errno`.

Returns

This function returns a pointer to a string. Your application must not modify that string.

Portability

`strerror_r` is a GNU extension.
`strerror_r` requires no supporting OS subroutines.
5.26 `strlen`—character string length

Synopsis

```c
#include <string.h>
size_t strlen(const char *str);
```

Description

The `strlen` function works out the length of the string starting at `*str` by counting characters until it reaches a NULL character.

Returns

`strlen` returns the character count.

Portability

`strlen` is ANSI C.

`strlen` requires no supporting OS subroutines.
5.27 `strlwr`—force string to lowercase

Synopsis

```c
#include <string.h>
char *strlwr(char *a);
```

Description

`strlwr` converts each character in the string at `a` to lowercase.

Returns

`strlwr` returns its argument, `a`.

Portability

`strlwr` is not widely portable.

`strlwr` requires no supporting OS subroutines.
5.28 `strncasecmp`—case-insensitive character string compare

Synopsis

```c
#include <string.h>
int strncasecmp(const char *a, const char * b, size_t length);
```

Description

`strncasecmp` compares up to `length` characters from the string at `a` to the string at `b` in a case-insensitive manner.

Returns

If `*a` sorts lexicographically after `*b` (after both are converted to uppercase), `strncasecmp` returns a number greater than zero. If the two strings are equivalent, `strncasecmp` returns zero. If `*a` sorts lexicographically before `*b`, `strncasecmp` returns a number less than zero.

Portability

`strncasecmp` is in the Berkeley Software Distribution.

`strncasecmp` requires no supporting OS subroutines. It uses tolower() from elsewhere in this library.
5.29 `strncat`—concatenate strings

Synopsis

```c
#include <string.h>
char *strncat(char *dst, const char *src, size_t length);
```

Description

`strncat` appends not more than `length` characters from the string pointed to by `src` (including the terminating null character) to the end of the string pointed to by `dst`. The initial character of `src` overwrites the null character at the end of `dst`. A terminating null character is always appended to the result.

Warnings

Note that a null is always appended, so that if the copy is limited by the `length` argument, the number of characters appended to `dst` is `n + 1`.

Returns

This function returns the initial value of `dst`.

Portability

`strncat` is ANSI C.

`strncat` requires no supporting OS subroutines.
5.30 strncmp—character string compare

Synopsis

```
#include <string.h>
int strncmp(const char *a, const char * b, size_t length);
```

Description

`strncmp` compares up to `length` characters from the string at `a` to the string at `b`.

Returns

If `*a` sorts lexicographically after `*b`, `strncmp` returns a number greater than zero. If the two strings are equivalent, `strncmp` returns zero. If `*a` sorts lexicographically before `*b`, `strncmp` returns a number less than zero.

Portability

`strncmp` is ANSI C.

`strncmp` requires no supporting OS subroutines.
5.31 strncpy—counted copy string

Synopsis

```c
#include <string.h>
char *strncpy(char *dst, const char *src, size_t length);
```

Description

`strncpy` copies not more than `length` characters from the the string pointed to by `src` (including the terminating null character) to the array pointed to by `dst`. If the string pointed to by `src` is shorter than `length` characters, null characters are appended to the destination array until a total of `length` characters have been written.

Returns

This function returns the initial value of `dst`.

Portability

`strncpy` is ANSI C.

`strncpy` requires no supporting OS subroutines.
5.32 strnlen—character string length

Synopsis

```c
#include <string.h>
size_t strnlen(const char *str, size_t n);
```

Description
The `strnlen` function works out the length of the string starting at `str` by counting characters until it reaches a NUL character or the maximum: `n` number of characters have been inspected.

Returns
`strnlen` returns the character count or `n`.

Portability
`strnlen` is a GNU extension.

`strnlen` requires no supporting OS subroutines.
5.33 strpbrk—find characters in string

Synopsis

```
#include <string.h>
char *strpbrk(const char *s1, const char *s2);
```

Description

This function locates the first occurrence in the string pointed to by s1 of any character in string pointed to by s2 (excluding the terminating null character).

Returns

strpbrk returns a pointer to the character found in s1, or a null pointer if no character from s2 occurs in s1.

Portability

strpbrk requires no supporting OS subroutines.
5.34 `strrchr`—reverse search for character in string

Synopsis

```c
#include <string.h>
char * strrchr(const char *string, int c);
```

Description

This function finds the last occurrence of `c` (converted to a char) in the string pointed to by `string` (including the terminating null character).

Returns

Returns a pointer to the located character, or a null pointer if `c` does not occur in `string`.

Portability

`strrchr` is ANSI C.

`strrchr` requires no supporting OS subroutines.
5.35 `strspn`—find initial match

**Synopsis**

```c
#include <string.h>
size_t strspn(const char *s1, const char *s2);
```

**Description**

This function computes the length of the initial segment of the string pointed to by `s1` which consists entirely of characters from the string pointed to by `s2` (excluding the terminating null character).

**Returns**

`strspn` returns the length of the segment found.

**Portability**

`strspn` is ANSI C.

`strspn` requires no supporting OS subroutines.
5.36 **strstr**—find string segment

**Synopsis**

```
#include <string.h>
char *strstr(const char *s1, const char *s2);
```

**Description**

Locates the first occurrence in the string pointed to by `s1` of the sequence of characters in the string pointed to by `s2` (excluding the terminating null character).

**Returns**

Returns a pointer to the located string segment, or a null pointer if the string `s2` is not found. If `s2` points to a string with zero length, `s1` is returned.

**Portability**

`strstr` is ANSI C.

`strstr` requires no supporting OS subroutines.
5.37 `strtok`, `strtok_r`, `strsep`—get next token from a string

Synopsis

```c
#include <string.h>
char *strtok(char *source, const char *delimiters)
char *strtok_r(char *source, const char *delimiters, char **lasts)
char *strsep(char **source_ptr, const char *delimiters)
```

Description

The `strtok` function is used to isolate sequential tokens in a null-terminated string, `*source`. These tokens are delimited in the string by at least one of the characters in `*delimiters`. The first time that `strtok` is called, `*source` should be specified; subsequent calls, wishing to obtain further tokens from the same string, should pass a null pointer instead. The separator string, `*delimiters`, must be supplied each time and may change between calls.

The `strtok` function returns a pointer to the beginning of each subsequent token in the string, after replacing the separator character itself with a null character. When no more tokens remain, a null pointer is returned.

The `strtok_r` function has the same behavior as `strtok`, except a pointer to placeholder `*lasts` must be supplied by the caller.

The `strsep` function is similar in behavior to `strtok`, except a pointer to the string pointer must be supplied `source_ptr` and the function does not skip leading delimiters. When the string starts with a delimiter, the delimiter is changed to the null character and the empty string is returned. Like `strtok_r` and `strtok`, the `*source_ptr` is updated to the next character following the last delimiter found or NULL if the end of string is reached with no more delimiters.

Returns

`strtok`, `strtok_r`, and `strsep` all return a pointer to the next token, or NULL if no more tokens can be found. For `strsep`, a token may be the empty string.

Portability

`strtok` is ANSI C. `strtok_r` is POSIX. `strsep` is a BSD extension.

`strtok`, `strtok_r`, and `strsep` require no supporting OS subroutines.
5.38 `strupr`—force string to uppercase

Synopsis
```
#include <string.h>
char *strupr(char *a);
```

Description
`strupr` converts each character in the string at `a` to uppercase.

Returns
`strupr` returns its argument, `a`.

Portability
`strupr` is not widely portable.
`strupr` requires no supporting OS subroutines.
Chapter 5: Strings and Memory ('string.h')

5.39 strxfrm—transform string

Synopsis

```
#include <string.h>
size_t strxfrm(char *s1, const char *s2, size_t n);
```

Description

This function transforms the string pointed to by `s2` and places the resulting string into the array pointed to by `s1`. The transformation is such that if the `strcmp` function is applied to the two transformed strings, it returns a value greater than, equal to, or less than zero, corresponding to the result of a `strcoll` function applied to the same two original strings. No more than `n` characters are placed into the resulting array pointed to by `s1`, including the terminating null character. If `n` is zero, `s1` may be a null pointer. If copying takes place between objects that overlap, the behavior is undefined. With a C locale, this function just copies.

Returns

The `strxfrm` function returns the length of the transformed string (not including the terminating null character). If the value returned is `n` or more, the contents of the array pointed to by `s1` are indeterminate.

Portability

`strxfrm` is ANSI C.

`strxfrm` requires no supporting OS subroutines.
5.40 swab—swap adjacent bytes

Synopsis

```c
#include <unistd.h>
void swab(const void *in, void *out, ssize_t n);
```

Description

This function copies \( n \) bytes from the memory region pointed to by \( \text{in} \) to the memory region pointed to by \( \text{out} \), exchanging adjacent even and odd bytes.

Portability

\text{swab} requires no supporting OS subroutines.
6 Wide Character Strings (‘wchar.h’)

This chapter describes wide-character string-handling functions and managing areas of memory containing wide characters. The corresponding declarations are in ‘wchar.h’.
6.1 wmemchr—find a wide character in memory

Synopsis

```c
#include <wchar.h>
wchar_t *wmemchr(const wchar_t *s, wchar_t c, size_t n);
```

Description

The `wmemchr` function locates the first occurrence of `c` in the initial `n` wide characters of the object pointed to be `s`. This function is not affected by locale and all wchar_t values are treated identically. The null wide character and wchar_t values not corresponding to valid characters are not treated specially.

If `n` is zero, `s` must be a valid pointer and the function behaves as if no valid occurrence of `c` is found.

Returns

The `wmemchr` function returns a pointer to the located wide character, or a null pointer if the wide character does not occur in the object.

Portability


No supporting OS subroutines are required.
6.2 wmemcmp—compare wide characters in memory

Synopsis

```c
#include <wchar.h>

int wmemcmp(const wchar_t *s1, const wchar_t *s2, size_t n);
```

Description

The `wmemcmp` function compares the first `n` wide characters of the object pointed to by `s1` to the first `n` wide characters of the object pointed to by `s2`. This function is not affected by locale and all `wchar_t` values are treated identically. The null wide character and `wchar_t` values not corresponding to valid characters are not treated specially.

If `n` is zero, `s1` and `s2` must be a valid pointers and the function behaves as if the two objects compare equal.

Returns

The `wmemcmp` function returns an integer greater than, equal to, or less than zero, accordingly as the object pointed to by `s1` is greater than, equal to, or less than the object pointed to by `s2`.

Portability


No supporting OS subroutines are required.
6.3 **wmemcpyp—copy wide characters in memory**

**Synopsis**

```c
#include <wchar.h>
wchar_t *wmemcpyp(wchar_t *d, const wchar_t *s, size_t n);
```

**Description**

The `wmemcpyp` function copies `n` wide characters from the object pointed to by `s` to the object pointed to be `d`. This function is not affected by locale and all `wchar_t` values are treated identically. The null wide character and `wchar_t` values not corresponding to valid characters are not treated specially.

If `n` is zero, `d` and `s` must be a valid pointers, and the function copies zero wide characters.

**Returns**

The `wmemcpyp` function returns the value of `d`.

**Portability**


No supporting OS subroutines are required.
6.4 wmemmove—copy wide characters in memory with overlapping areas

Synopsis

```
#include <wchar.h>

wchar_t *wmemmove(wchar_t *d, const wchar_t *s, size_t n);
```

Description

The `wmemmove` function copies `n` wide characters from the object pointed to by `s` to the object pointed to by `d`. Copying takes place as if the `n` wide characters from the object pointed to by `s` are first copied into a temporary array of `n` wide characters that does not overlap the objects pointed to by `d` or `s`, and then the `n` wide characters from the temporary array are copied into the object pointed to by `d`.

This function is not affected by locale and all `wchar_t` values are treated identically. The null wide character and `wchar_t` values not corresponding to valid characters are not treated specially.

If `n` is zero, `d` and `s` must be a valid pointers, and the function copies zero wide characters.

Returns

The `wmemmove` function returns the value of `d`.

Portability


No supporting OS subroutines are required.
6.5 `wmemset’—set wide characters in memory

Synopsis

```c
#include <wchar.h>
wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);
```

Description

The `wmemset` function copies the value of `c` into each of the first `n` wide characters of the object pointed to by `s`. This function is not affected by locale and all `wchar_t` values are treated identically. The null wide character and `wchar_t` values not corresponding to valid characters are not treated specially.

If `n` is zero, `s` must be a valid pointer and the function copies zero wide characters.

Returns

The `wmemset` function returns the value of `s`.

Portability


No supporting OS subroutines are required.
6.6 wcscat—concatenate two wide-character strings

Synopsis

```c
#include <wchar.h>
wchar_t *wcscat(wchar_t *s1, const wchar_t *s2);
```

Description

The `wcscat` function appends a copy of the wide-character string pointed to by `s2` (including the terminating null wide-character code) to the end of the wide-character string pointed to by `s1`. The initial wide-character code of `s2` overwrites the null wide-character code at the end of `s1`. If copying takes place between objects that overlap, the behaviour is undefined.

Returns

The `wcscat` function returns `s1`; no return value is reserved to indicate an error.

Portability

No supporting OS subroutines are required.
6.7 wcschr—wide-character string scanning operation

Synopsis

```c
#include <wchar.h>
wchar_t *wcschr(const wchar_t *s, wchar_t c);
```

Description

The `wcschr` function locates the first occurrence of `c` in the wide-character string pointed to by `s`. The value of `c` must be a character representable as a type `wchar_t` and must be a wide-character code corresponding to a valid character in the current locale. The terminating null wide-character string.

Returns

Upon completion, `wcschr` returns a pointer to the wide-character code, or a null pointer if the wide-character code is not found.

Portability


No supporting OS subroutines are required.
Chapter 6: Wide Character Strings (‘wchar.h’)

6.8 wcscmp—compare two wide-character strings

Synopsis

```
#include <wchar.h>
int wcscmp(const wchar_t *s1, *s2);
```

Description
The `wcscmp` function compares the wide-character string pointed to by `s1` to the wide-character string pointed to by `s2`.

The sign of a non-zero return value is determined by the sign of the difference between the values of the first pair of wide-character codes that differ in the objects being compared.

Returns
Upon completion, `wcscmp` returns an integer greater than, equal to or less than 0, if the wide-character string pointed to by `s1` is greater than, equal to or less than the wide-character string pointed to by `s2` respectively.

Portability

No supporting OS subroutines are required.
6.9 wcscoll—locale-specific wide-character string compare

Synopsis

```c
#include <wchar.h>
int wcscoll(const wchar_t *strA, const wchar_t *strB);
```

Description

`wcscoll` compares the wide-character string pointed to by `strA` to the wide-character string pointed to by `strB`, using an interpretation appropriate to the current `LC_COLLATE` state. The current implementation of `wcscoll` simply uses `wscmp` and does not support any language-specific sorting.

Returns

If the first string is greater than the second string, `wcscoll` returns a number greater than zero. If the two strings are equivalent, `wcscoll` returns zero. If the first string is less than the second string, `wcscoll` returns a number less than zero.

Portability

6.10 wcsncpy—copy a wide-character string

Synopsis

```c
#include <wchar.h>

wchar_t *wcsncpy(wchar_t *s1, const wchar_t *, s2);
```

Description

The `wcsncpy` function copies the wide-character string pointed to by `s2` (including the terminating null wide-character code) into the array pointed to by `s1`. If copying takes place between objects that overlap, the behaviour is undefined.

Returns

The `wcsncpy` function returns `s1`; no return value is reserved to indicate an error.

Portability


No supporting OS subroutines are required.
6.11 wcpcpy—copy a wide-character string returning a pointer to its end

Synopsis

```c
#include <wchar.h>
wchar_t *wcpcpy(wchar_t *s1, const wchar_t *s2);
```

Description

The `wcpcpy` function copies the wide-character string pointed to by `s2` (including the terminating null wide-character code) into the array pointed to by `s1`. If copying takes place between objects that overlap, the behaviour is undefined.

Returns

This function returns a pointer to the end of the destination string, thus pointing to the trailing '\0'.

Portability

`wcpcpy` is a GNU extension.

No supporting OS subroutines are required.
6.12 wcscspn—get length of a complementary wide substring

Synopsis

```c
#include <wchar.h>
size_t wcscspn(const wchar_t *s, wchar_t *set);
```

Description
The `wcscspn` function computes the length of the maximum initial segment of the wide-character string pointed to by `s` which consists entirely of wide-character codes not from the wide-character string pointed to by `set`.

Returns
The `wcscspn` function returns the length of the initial substring of `s`; no return value is reserved to indicate an error.

Portability
The `wcscspn` function is ISO/IEC 9899/AMD1:1995 (ISO C).
No supporting OS subroutines are required.
6.13 wcs1cat—concatenate wide-character strings to specified length

Synopsis

#include <wchar.h>
size_t wcs1cat(wchar_t *dst, const wchar_t *src, size_t siz);

Description
The wcs1cat function appends wide characters from src to end of the dst wide-character string so that the resultant wide-character string is not more than siz wide characters including the terminating null wide-character code. A terminating null wide character is always added unless siz is 0. Thus, the maximum number of wide characters that can be appended from src is siz - 1. If copying takes place between objects that overlap, the behaviour is undefined.

Returns
Wide-character string length of initial dst plus the wide-character string length of src (does not include terminating null wide-characters). If the return value is greater than or equal to siz, then truncation occurred and not all wide characters from src were appended.

Portability
No supporting OS subroutines are required.
6.14 wcs1cpy—copy a wide-character string to specified length

Synopsis

```c
#include <wchar.h>
size_t wcs1cpy(wchar_t *dst, const wchar_t *src, size_t siz);
```

Description

wcs1cpy copies wide characters from src to dst such that up to siz - 1 characters are copied. A terminating null is appended to the result, unless siz is zero.

Returns

wcs1cpy returns the number of wide characters in src, not including the terminating null wide character. If the return value is greater than or equal to siz, then not all wide characters were copied from src and truncation occurred.

Portability

No supporting OS subroutines are required.
6.15 wcslen—get wide-character string length

Synopsis

```
#include <wchar.h>
size_t wcslen(const wchar_t *s);
```

Description
The `wcslen` function computes the number of wide-character codes in the wide-character string to which `s` points, not including the terminating null wide-character code.

Returns
The `wcslen` function returns the length of `s`; no return value is reserved to indicate an error.

Portability
No supporting OS subroutines are required.
6.16 wcsncat—concatenate part of two wide-character strings

Synopsis

#include <wchar.h>

wchar_t *wcsncat(wchar_t *s1, const wchar_t *s2, size_t n);

Description

The wcsncat function appends not more than \( n \) wide-character codes (a null wide-character code and wide-character codes that follow it are not appended) from the array pointed to by \( s2 \) to the end of the wide-character string pointed to by \( s1 \). The initial wide-character code of \( s2 \) overwrites the null wide-character code at the end of \( s1 \). A terminating null wide-character code is always appended to the result. If copying takes place between objects that overlap, the behaviour is undefined.

Returns

The wcsncat function returns \( s1 \); no return value is reserved to indicate an error.

Portability


No supporting OS subroutines are required.
6.17 wcsncmp—compare part of two wide-character strings

Synopsis

```c
#include <wchar.h>
int wcsncmp(const wchar_t *s1, const wchar_t *s2, size_t n);
```

Description
The wcsncmp function compares not more than n wide-character codes (wide-character codes that follow a null wide-character code are not compared) from the array pointed to by s1 to the array pointed to by s2.
The sign of a non-zero return value is determined by the sign of the difference between the values of the first pair of wide-character codes that differ in the objects being compared.

Returns
Upon successful completion, wcsncmp returns an integer greater than, equal to or less than 0, if the possibly null-terminated array pointed to by s1 is greater than, equal to or less than the possibly null-terminated array pointed to by s2 respectively.

Portability
No supporting OS subroutines are required.
6.18 wcsncpy—copy part of a wide-character string

Synopsis

```c
#include <wchar.h>
wchar_t *wcsncpy(wchar_t *s1, const wchar_t *s2, size_t n);
```

Description

The `wcsncpy` function copies not more than `n` wide-character codes (wide-character codes that follow a null wide-character code are not copied) from the array pointed to by `s2` to the array pointed to by `s1`. If copying takes place between objects that overlap, the behaviour is undefined.

If the array pointed to by `s2` is a wide-character string that is shorter than `n` wide-character codes, null wide-character codes are appended to the copy in the array pointed to by `s1`, until `n` wide-character codes in all are written.

Returns

The `wcsncpy` function returns `s1`; no return value is reserved to indicate an error.

Portability


No supporting OS subroutines are required.
6.19 `wcpncpy`—copy part of a wide-character string returning a pointer to its end

Synopsis

```c
#include <wchar.h>
wchar_t *wcpncpy(wchar_t *s1, const wchar_t *s2, size_t n);
```

Description

The `wcpncpy` function copies not more than `n` wide-character codes (wide-character codes that follow a null wide-character code are not copied) from the array pointed to by `s2` to the array pointed to by `s1`. If copying takes place between objects that overlap, the behaviour is undefined.

If the array pointed to by `s2` is a wide-character string that is shorter than `n` wide-character codes, null wide-character codes are appended to the copy in the array pointed to by `s1`, until `n` wide-character codes in all are written.

Returns

The `wcpncpy` function returns `s1`; no return value is reserved to indicate an error.

Portability


No supporting OS subroutines are required.
6.20 wcsnlen—get fixed-size wide-character string length

Synopsis

```
#include <wchar.h>
size_t wcsnlen(const wchar_t *s, size_t maxlen);
```

Description

The `wcsnlen` function computes the number of wide-character codes in the wide-character string pointed to by `s` not including the terminating L'\0' wide character but at most `maxlen` wide characters.

Returns

`wcsnlen` returns the length of `s` if it is less then `maxlen`, or `maxlen` if there is no L'\0' wide character in first `maxlen` characters.

Portability

`wcsnlen` is a GNU extension.

`wcsnlen` requires no supporting OS subroutines.
6.21 \texttt{wcspbrk}—scan wide-character string for a wide-character code

Synopsis

\begin{verbatim}
#include <wchar.h>
wchar_t *wcspbrk(const wchar_t *s, const wchar_t *set);
\end{verbatim}

Description

The \texttt{wcspbrk} function locates the first occurrence in the wide-character string pointed to by \texttt{s} of any wide-character code from the wide-character string pointed to by \texttt{set}.

Returns

Upon successful completion, \texttt{wcspbrk} returns a pointer to the wide-character code or a null pointer if no wide-character code from \texttt{set} occurs in \texttt{s}.

Portability

\texttt{wcspbrk} is ISO/IEC 9899/AMD1:1995 (ISO C).

No supporting OS subroutines are required.
6.22 wcsrchr—wide-character string scanning operation

Synopsis

```c
#include <wchar.h>
wchar_t *wcsrchr(const wchar_t *s, wchar_t c);
```

Description

The `wcsrchr` function locates the last occurrence of `c` in the wide-character string pointed to by `s`. The value of `c` must be a character representable as a type `wchar_t` and must be a wide-character code corresponding to a valid character in the current locale. The terminating null wide-character code is considered to be part of the wide-character string.

Returns

Upon successful completion, `wcsrchr` returns a pointer to the wide-character code or a null pointer if `c` does not occur in the wide-character string.

Portability


No supporting OS subroutines are required.
6.23 wcsspn—get length of a wide substring

Synopsis

```c
#include <wchar.h>
size_t wcsspn(const wchar_t *s, const wchar_t *set);
```

Description

The wcsspn function computes the length of the maximum initial segment of the wide-character string pointed to by `s` which consists entirely of wide-character codes from the wide-character string pointed to by `set`.

Returns

The wcsspn() function returns the length `s1`; no return value is reserved to indicate an error.

Portability


No supporting OS subroutines are required.
6.24 wcsstr—find a wide-character substring

Synopsis

```c
#include <wchar.h>
wchar_t *wcsstr(const wchar_t *big, const wchar_t *little);
```

Description
The **wcsstr** function locates the first occurrence in the wide-character string pointed to by *big* of the sequence of wide characters (excluding the terminating null wide character) in the wide-character string pointed to by *little*.

Returns
On successful completion, **wcsstr** returns a pointer to the located wide-character string, or a null pointer if the wide-character string is not found.
If *little* points to a wide-character string with zero length, the function returns *big*.

Portability
**wcsstr** is ISO/IEC 9899/AMD1:1995 (ISO C).
6.25 wcstok—get next token from a string

Synopsis

```c
#include <wchar.h>
wchar_t *wcstok(wchar_t *source, const wchar_t *delimiters,
                wchar_t **lasts)
```

Description

The `wcstok` function is the wide-character equivalent of the `strtok_r` function (which in turn is the same as the `strtok` function with an added argument to make it thread-safe).

The `wcstok` function is used to isolate (one at a time) sequential tokens in a null-terminated wide-character string, `*source`. A token is defined as a substring not containing any wide-characters from `*delimiters`.

The first time that `wcstok` is called, `*source` should be specified with the wide-character string to be searched, and `*lasts`—but not `lasts`, which must be non-NULL—may be random; subsequent calls, wishing to obtain further tokens from the same string, should pass a null pointer for `*source` instead but must supply `*lasts` unchanged from the last call. The separator wide-character string, `*delimiters`, must be supplied each time and may change between calls. A pointer to placeholder `*lasts` must be supplied by the caller, and is set each time as needed to save the state by `wcstok`. Every call to `wcstok` with `*source == NULL` must pass the value of `*lasts` as last set by `wcstok`.

The `wcstok` function returns a pointer to the beginning of each subsequent token in the string, after replacing the separator wide-character itself with a null wide-character. When no more tokens remain, a null pointer is returned.

Returns

`wcstok` returns a pointer to the first wide character of a token, or NULL if there is no token.

Portability

`wcstok` is C99 and POSIX.1-2001.

`wcstok` requires no supporting OS subroutines.
6.26 wcswidth—number of column positions of a wide-character string

Synopsis

    #include <wchar.h>
    int wcswidth(const wchar_t *pwcs, size_t n);

Description

The wcswidth function shall determine the number of column positions required for \( n \) wide-character codes (or fewer than \( n \) wide-character codes if a null wide-character code is encountered before \( n \) wide-character codes are exhausted) in the string pointed to by \( pwcs \).

Returns

The wcswidth function either shall return 0 (if \( pwcs \) points to a null wide-character code), or return the number of column positions to be occupied by the wide-character string pointed to by \( pwcs \), or return -1 (if any of the first \( n \) wide-character codes in the wide-character string pointed to by \( pwcs \) is not a printable wide-character code).

Portability

wcswidth has been introduced in the Single UNIX Specification Volume 2. wcswidth has been marked as an extension in the Single UNIX Specification Volume 3.
6.27 wcsxfrm—locale-specific wide-character string transformation

Synopsis

#include <wchar.h>

int wcsxfrm(wchar_t *stra, const wchar_t *strb, size_t n);

Description

wcsxfrm transforms the wide-character string pointed to by strb to the wide-character string pointed to by stra, Comparing two transformed wide strings with wcsncmp should return the same result as comparing the original strings with wcsxcoll. No more than n wide characters are transformed, including the trailing null character.

If n is 0, stra may be a NULL pointer.

The current implementation of wcsxfrm simply uses wmemcpy and does not support any language-specific transformations.

Returns

wcsxfrm returns the length of the transformed wide character string. if the return value is greater or equal to n, the content of stra is undefined.

Portability

6.28 \texttt{wcwidth}—number of column positions of a wide-character code

Synopsis

\begin{verbatim}
#include <wchar.h>
int wcwidth(const wchar_t \texttt{wc});
\end{verbatim}

Description

The \texttt{wcwidth} function shall determine the number of column positions required for the wide character \texttt{wc}. The application shall ensure that the value of \texttt{wc} is a character representable as a wchar_t, and is a wide-character code corresponding to a valid character in the current locale.

Returns

The \texttt{wcwidth} function shall either return 0 (if \texttt{wc} is a null wide-character code), or return the number of column positions to be occupied by the wide-character code \texttt{wc}, or return -1 (if \texttt{wc} does not correspond to a printable wide-character code).

The current implementation of \texttt{wcwidth} simply sets the width of all printable characters to 1 since newlib has no character tables around.

Portability

\texttt{wcwidth} has been introduced in the Single UNIX Specification Volume 2. \texttt{wcwidth} has been marked as an extension in the Single UNIX Specification Volume 3.
7 Signal Handling (‘signal.h’)

A signal is an event that interrupts the normal flow of control in your program. Your operating environment normally defines the full set of signals available (see ‘sys/signal.h’), as well as the default means of dealing with them—typically, either printing an error message and aborting your program, or ignoring the signal.

All systems support at least the following signals:

- **SIGABRT**: Abnormal termination of a program; raised by the `<<abort>>` function.
- **SIGFPE**: A domain error in arithmetic, such as overflow, or division by zero.
- **SIGILL**: Attempt to execute as a function data that is not executable.
- **SIGINT**: Interrupt; an interactive attention signal.
- **SIGSEGV**: An attempt to access a memory location that is not available.
- **SIGTERM**: A request that your program end execution.

Two functions are available for dealing with asynchronous signals—one to allow your program to send signals to itself (this is called raising a signal), and one to specify subroutines (called handlers) to handle particular signals that you anticipate may occur—whether raised by your own program or the operating environment.

To support these functions, ‘signal.h’ defines three macros:

- **SIG_DFL**: Used with the `signal` function in place of a pointer to a handler subroutine, to select the operating environment’s default handling of a signal.
- **SIG_IGN**: Used with the `signal` function in place of a pointer to a handler, to ignore a particular signal.
- **SIG_ERR**: Returned by the `signal` function in place of a pointer to a handler, to indicate that your request to set up a handler could not be honored for some reason.

‘signal.h’ also defines an integral type, `sig_atomic_t`. This type is not used in any function declarations; it exists only to allow your signal handlers to declare a static storage location where they may store a signal value. (Static storage is not otherwise reliable from signal handlers.)
7.1 raise—send a signal

Synopsis

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

int _raise_r(void *reent, int sig);
```

Description

Send the signal `sig` (one of the macros from `sys/signal.h`). This interrupts your program’s normal flow of execution, and allows a signal handler (if you’ve defined one, using `signal`) to take control.

The alternate function `_raise_r` is a reentrant version. The extra argument `reent` is a pointer to a reentrancy structure.

Returns

The result is 0 if `sig` was successfully raised, 1 otherwise. However, the return value (since it depends on the normal flow of execution) may not be visible, unless the signal handler for `sig` terminates with a `return` or unless `SIG_IGN` is in effect for this signal.

Portability

ANSI C requires `raise`, but allows the full set of signal numbers to vary from one implementation to another.

Required OS subroutines: `getpid`, `kill`.
7.2 signal—specify handler subroutine for a signal

Synopsis

#include <signal.h>

void (*signal(int sig, void(*func)(int))) (int);

void (*_signal_r(void *reent, int sig, void(*func)(int))) (int);

Description

signal provides a simple signal-handling implementation for embedded targets.
signal allows you to request changed treatment for a particular signal sig. You can use one of the predefined macros SIG_DFL (select system default handling) or SIG_IGN (ignore this signal) as the value of func; otherwise, func is a function pointer that identifies a subroutine in your program as the handler for this signal.

Some of the execution environment for signal handlers is unpredictable; notably, the only library function required to work correctly from within a signal handler is signal itself, and only when used to redefine the handler for the current signal value.

Static storage is likewise unreliable for signal handlers, with one exception: if you declare a static storage location as 'volatile sig_atomic_t', then you may use that location in a signal handler to store signal values.

If your signal handler terminates using return (or implicit return), your program’s execution continues at the point where it was when the signal was raised (whether by your program itself, or by an external event). Signal handlers can also use functions such as exit and abort to avoid returning.

The alternate function _signal_r is the reentrant version. The extra argument reent is a pointer to a reentrancy structure.

Returns

If your request for a signal handler cannot be honored, the result is SIG_ERR; a specific error number is also recorded in errno.

Otherwise, the result is the previous handler (a function pointer or one of the predefined macros).

Portability

ANSI C requires signal.

No supporting OS subroutines are required to link with signal, but it will not have any useful effects, except for software generated signals, without an operating system that can actually raise exceptions.
8 Time Functions (‘time.h’)

This chapter groups functions used either for reporting on time (elapsed, current, or compute time) or to perform calculations based on time.

The header file ‘time.h’ defines three types. clock_t and time_t are both used for representations of time particularly suitable for arithmetic. (In this implementation, quantities of type clock_t have the highest resolution possible on your machine, and quantities of type time_t resolve to seconds.) size_t is also defined if necessary for quantities representing sizes.

‘time.h’ also defines the structure tm for the traditional representation of Gregorian calendar time as a series of numbers, with the following fields:

- **tm_sec** Seconds, between 0 and 60 inclusive (60 allows for leap seconds).
- **tm_min** Minutes, between 0 and 59 inclusive.
- **tm_hour** Hours, between 0 and 23 inclusive.
- **tm_mday** Day of the month, between 1 and 31 inclusive.
- **tm_mon** Month, between 0 (January) and 11 (December).
- **tm_year** Year (since 1900), can be negative for earlier years.
- **tm_wday** Day of week, between 0 (Sunday) and 6 (Saturday).
- **tm_yday** Number of days elapsed since last January 1, between 0 and 365 inclusive.
- **tm_isdst** Daylight Savings Time flag: positive means DST in effect, zero means DST not in effect, negative means no information about DST is available.
8.1 asctime—format time as string

Synopsis

```c
#include <time.h>
char *asctime(const struct tm *clock);
char *asctime_r(const struct tm *clock, char *buf);
```

Description

Format the time value at `clock` into a string of the form

```plaintext
```

The string is generated in a static buffer; each call to `asctime` overwrites the string generated by previous calls.

Returns

A pointer to the string containing a formatted timestamp.

Portability

ANSI C requires `asctime`.

`asctime` requires no supporting OS subroutines.
8.2 clock—cumulative processor time

Synopsis

```c
#include <time.h>
clock_t clock(void);
```

Description

Calculates the best available approximation of the cumulative amount of time used by your program since it started. To convert the result into seconds, divide by the macro `CLOCKS_PER_SEC`.

Returns

The amount of processor time used so far by your program, in units defined by the machine-dependent macro `CLOCKS_PER_SEC`. If no measurement is available, the result is `(clock_t)-1`.

Portability

ANSI C requires `clock` and `CLOCKS_PER_SEC`.
Supporting OS subroutine required: `times`.
8.3 ctime—convert time to local and format as string

Synopsis

```c
#include <time.h>
char *ctime(const time_t *clock);
char *ctime_r(const time_t *clock, char *buf);
```

Description
Convert the time value at `clock` to local time (like `localtime`) and format it into a string of the form

0`

(like `asctime`).

Returns
A pointer to the string containing a formatted timestamp.

Portability
ANSI C requires `ctime`.
`ctime` requires no supporting OS subroutines.
8.4 difftime—subtract two times

Synopsis

```c
#include <time.h>
double difftime(time_t tim1, time_t tim2);
```

Description

Subtracts the two times in the arguments: ‘tim1 − tim2’.

Returns

The difference (in seconds) between tim2 and tim1, as a double.

Portability

ANSI C requires difftime, and defines its result to be in seconds in all implementations. difftime requires no supporting OS subroutines.
8.5 gmtime—convert time to UTC traditional form

Synopsis

```c
#include <time.h>
struct tm *gmtime(const time_t *clock);
struct tm *gmtime_r(const time_t *clock, struct tm *res);
```

Description

gmtime takes the time at clock representing the number of elapsed seconds since 00:00:00 on January 1, 1970, Universal Coordinated Time (UTC, also known in some countries as GMT, Greenwich Mean time) and converts it to a `struct tm` representation.

gmtime constructs the traditional time representation in static storage; each call to gmtime or localtime will overwrite the information generated by previous calls to either function.

Returns

A pointer to the traditional time representation (`struct tm`).

Portability

ANSI C requires gmtime.

gmtime requires no supporting OS subroutines.
8.6 localtime—convert time to local representation

Synopsis

```c
#include <time.h>
struct tm *localtime(time_t *clock);
struct tm *localtime_r(time_t *clock, struct tm *res);
```

Description

`localtime` converts the time at `clock` into local time, then converts its representation from
the arithmetic representation to the traditional representation defined by `struct tm`.

`localtime` constructs the traditional time representation in static storage; each call to
`gmtime` or `localtime` will overwrite the information generated by previous calls to either
function.

`mktime` is the inverse of `localtime`.

Returns

A pointer to the traditional time representation (`struct tm`).

Portability

ANSI C requires `localtime`.

`localtime` requires no supporting OS subroutines.
8.7 mktime—convert time to arithmetic representation

Synopsis

```
#include <time.h>

    time_t mktime(struct tm *timp);
```

Description

`mktime` assumes the time at `timp` is a local time, and converts its representation from the traditional representation defined by `struct tm` into a representation suitable for arithmetic. `localtime` is the inverse of `mktime`.

Returns

If the contents of the structure at `timp` do not form a valid calendar time representation, the result is -1. Otherwise, the result is the time, converted to a `time_t` value.

Portability

ANSI C requires `mktime`.

`mktime` requires no supporting OS subroutines.
8.8 strftime—flexible calendar time formatter

Synopsis

```
#include <time.h>

size_t strftime(char *s, size_t maxsize, const char *format, const struct tm *timp);
```

Description

`strftime` converts a `struct tm` representation of the time (at `timp`) into a null-terminated string, starting at `s` and occupying no more than `maxsize` characters. You control the format of the output using the string at `format`. `*format` can contain two kinds of specifications: text to be copied literally into the formatted string, and time conversion specifications. Time conversion specifications are two- and three-character sequences beginning with ‘%’ (use ‘%%’ to include a percent sign in the output). Each defined conversion specification selects only the specified field(s) of calendar time data from `*timp`, and converts it to a string in one of the following ways:

- `%a` A three-letter abbreviation for the day of the week. [tm_wday]
- `%A` The full name for the day of the week, one of ‘Sunday’, ‘Monday’, ‘Tuesday’, ‘Wednesday’, ‘Thursday’, ‘Friday’, or ‘Saturday’. [tm_wday]
- `%b` A three-letter abbreviation for the month name. [tm_mon]
- `%c` A string representing the complete date and time, in the form "%a %b %e %H:%M:%S %Y" (example "Mon Apr 01 13:13:13 1992"). [tm_sec, tm_min, tm_hour, tm_mday, tm_mon, tm_year, tm_wday]
- `%C` The century, that is, the year divided by 100 then truncated. For 4-digit years, the result is zero-padded and exactly two characters; but for other years, there may a negative sign or more digits. In this way, `%C%y` is equivalent to `%Y`. [tm_year]
- `%d` The day of the month, formatted with two digits (from ‘01’ to ‘31’). [tm_mday]
- `%D` A string representing the date, in the form "%m/%d/%y". [tm_mday, tm_mon, tm_year]
- `%e` The day of the month, formatted with leading space if single digit (from ‘1’ to ‘31’). [tm_mday]
- `%Ex` In some locales, the E modifier selects alternative representations of certain modifiers x. But in the "C" locale supported by newlib, it is ignored, and treated as %x.
- `%F` A string representing the ISO 8601:2000 date format, in the form "%Y-%m-%d". [tm_mday, tm_mon, tm_year]
- `%g` The last two digits of the week-based year, see specifier %G (from ‘00’ to ‘99’). [tm_year, tm_wday, tm_yday]
The week-based year. In the ISO 8601:2000 calendar, week 1 of the year includes January 4th, and begin on Mondays. Therefore, if January 1st, 2nd, or 3rd falls on a Sunday, that day and earlier belong to the last week of the previous year; and if December 29th, 30th, or 31st falls on Monday, that day and later belong to week 1 of the next year. For consistency with %Y, it always has at least four characters. Example: "%G" for Saturday 2nd January 1999 gives "1998", and for Tuesday 30th December 1997 gives "1998". [tm_year, tm_wday, tm_yday]

%h A three-letter abbreviation for the month name (synonym for "%b"). [tm_mon]

%H The hour (on a 24-hour clock), formatted with two digits (from '00' to '23'). [tm_hour]

%I The hour (on a 12-hour clock), formatted with two digits (from '01' to '12'). [tm_hour]

%j The count of days in the year, formatted with three digits (from '001' to '366'). [tm_yday]

%k The hour (on a 24-hour clock), formatted with leading space if single digit (from '0' to '23'). Non-POSIX extension. [tm_hour]

%l The hour (on a 12-hour clock), formatted with leading space if single digit (from '1' to '12'). Non-POSIX extension. [tm_hour]

%m The month number, formatted with two digits (from '01' to '12'). [tm_mon]

%M The minute, formatted with two digits (from '00' to '59'). [tm_min]

%n A newline character ('\n').

%Ox In some locales, the O modifier selects alternative digit characters for certain modifiers x. But in the "C" locale supported by newlib, it is ignored, and treated as %x.

%p Either 'AM' or 'PM' as appropriate. [tm_hour]

%x The 12-hour time, to the second. Equivalent to "%I:%M:%S %p". [tm_sec, tm_min, tm_hour]

%r The 24-hour time, to the minute. Equivalent to "%H:%M". [tm_min, tm_hour]

%S The second, formatted with two digits (from '00' to '60'). The value 60 accounts for the occasional leap second. [tm_sec]

%c A tab character ('\t').

%t The 24-hour time, to the second. Equivalent to "%H:%M:%S". [tm_sec, tm_min, tm_hour]

%u The weekday as a number, 1-based from Monday (from '1' to '7'). [tm_wday]

%U The week number, where weeks start on Sunday, week 1 contains the first Sunday in a year, and earlier days are in week 0. Formatted with two digits (from '00' to '53'). See also %w. [tm_wday, tm_yday]

%V The week number, where weeks start on Monday, week 1 contains January 4th, and earlier days are in the previous year. Formatted with two digits (from '01' to '53'). See also %G. [tm_year, tm_wday, tm_yday]
%w
The weekday as a number, 0-based from Sunday (from ‘0’ to ‘6’). [tm_wday]

%W
The week number, where weeks start on Monday, week 1 contains the first
Monday in a year, and earlier days are in week 0. Formatted with two digits
(from ‘00’ to ‘53’). [tm_wday, tm_yday]

%x
A string representing the complete date, equivalent to “%m/%d/%Y”. [tm_mon, tm_mday, tm_year]

%X
A string representing the full time of day (hours, minutes, and seconds), equiva-
lent to “%H:%M:%S”. [tm_sec, tm_min, tm_hour]

%y
The last two digits of the year (from ‘00’ to ‘99’). [tm_year]

%Y
The full year, equivalent to %C%y. It will always have at least four characters,
but may have more. The year is accurate even when tm_year added to the
offset of 1900 overflows an int. [tm_year]

%z
The offset from UTC. The format consists of a sign (negative is west of
Greenwich), two characters for hour, then two characters for minutes (-hhmm
or +hhmm). If tm_isdst is negative, the offset is unknown and no output is
generated; if it is zero, the offset is the standard offset for the current time
zone; and if it is positive, the offset is the daylight savings offset for the current
timezone. The offset is determined from the TZ environment variable, as if by
calling tzset(). [tm_isdst]

%Z
The time zone name. If tm_isdst is negative, no output is generated. Otherwise,
the time zone name is based on the TZ environment variable, as if by calling
tzset(). [tm_isdst]

%%
A single character, ‘%’.

Returns
When the formatted time takes up no more than maxsize characters, the result is the length
of the formatted string. Otherwise, if the formatting operation was abandoned due to lack
of room, the result is 0, and the string starting at s corresponds to just those parts of
*format that could be completely filled in within the maxsize limit.

Portability
ANSI C requires strftime, but does not specify the contents of *s when the formatted
string would require more than maxsize characters. Unrecognized specifiers and fields of
timp that are out of range cause undefined results. Since some formats expand to 0 bytes,
it is wise to set *s to a nonzero value beforehand to distinguish between failure and an
empty string. This implementation does not support s being NULL, nor overlapping s and
format.

strftime requires no supporting OS subroutines.
8.9 time—get current calendar time (as single number)

Synopsis

```c
#include <time.h>

 time_t time(time_t *t);
```

Description

time looks up the best available representation of the current time and returns it, encoded as a time_t. It stores the same value at t unless the argument is NULL.

Returns

A -1 result means the current time is not available; otherwise the result represents the current time.

Portability

ANSI C requires time.

Supporting OS subroutine required: Some implementations require gettimeofday.
8.10 __tz_lock, __tz_unlock—lock time zone global variables

Synopsis

```c
#include "local.h"
void __tz_lock (void);
void __tz_unlock (void);
```

Description

The `tzset` facility functions call these functions when they need to ensure the values of global variables. The version of these routines supplied in the library use the lock API defined in `sys/lock.h`. If multiple threads of execution can call the time functions and give up scheduling in the middle, then you need to define your own versions of these functions in order to safely lock the time zone variables during a call. If you do not, the results of `localtime`, `mktime`, `ctime`, and `strftime` are undefined.

The lock `__tz_lock` may not be called recursively; that is, a call `__tz_lock` will always lock all subsequent `__tz_lock` calls until the corresponding `__tz_unlock` call on the same thread is made.
8.11 tzset—set timezone characteristics from TZ environment variable

Synopsis

```c
#include <time.h>
void tzset(void);
void _tzset_r (struct _reent *);
```

Description
tzset examines the TZ environment variable and sets up the three external variables: _timezone, _daylight, and tzname. The value of _timezone shall be the offset from the current time zone to GMT. The value of _daylight shall be 0 if there is no daylight savings time for the current time zone, otherwise it will be non-zero. The tzname array has two entries: the first is the name of the standard time zone, the second is the name of the daylight-savings time zone.

The TZ environment variable is expected to be in the following POSIX format:

```
stdoffset1[dst[offset2][start[/time1],end[/time2]]
```

where: 
- std is the name of the standard time-zone (minimum 3 chars)
- offset1 is the value to add to local time to arrive at Universal time (it has the form: hh:mm:ss)
- dst is the name of the alternate (daylight-savings) time-zone (min 3 chars)
- offset2 is the value to add to local time to arrive at Universal time (it has the same format as the std offset)
- start is the day that the alternate time-zone starts
- time1 is the optional time that the alternate time-zone starts (this is in local time and defaults to 02:00:00 if not specified)
- end is the day that the alternate time-zone ends
- time2 is the time that the alternate time-zone ends (it is in local time and defaults to 02:00:00 if not specified)

Note that there is no white-space padding between fields. Also note that if TZ is null, the default is Universal GMT which has no daylight-savings time. If TZ is empty, the default EST5EDT is used.

The function _tzset_r is identical to tzset only it is reentrant and is used for applications that use multiple threads.

Returns

There is no return value.

Portability
tzset is part of the POSIX standard.

Supporting OS subroutine required: None
9 Locale ('locale.h')

A locale is the name for a collection of parameters (affecting collating sequences and formatting conventions) that may be different depending on location or culture. The "C" locale is the only one defined in the ANSI C standard.

This is a minimal implementation, supporting only the required "C" value for locale; strings representing other locales are not honored. ("" is also accepted; it represents the default locale for an implementation, here equivalent to "C".

'locale.h' defines the structure lconv to collect the information on a locale, with the following fields:

char *decimal_point
The decimal point character used to format "ordinary" numbers (all numbers except those referring to amounts of money). "." in the C locale.

char *thousands_sep
The character (if any) used to separate groups of digits, when formatting ordinary numbers. "," in the C locale.

char *grouping
Specifications for how many digits to group (if any grouping is done at all) when formatting ordinary numbers. The numeric value of each character in the string represents the number of digits for the next group, and a value of 0 (that is, the string’s trailing NULL) means to continue grouping digits using the last value specified. Use CHAR_MAX to indicate that no further grouping is desired. "," in the C locale.

char *int_curr_symbol
The international currency symbol (first three characters), if any, and the character used to separate it from numbers. "," in the C locale.

char *currency_symbol
The local currency symbol, if any. "," in the C locale.

char *mon_decimal_point
The symbol used to delimit fractions in amounts of money. "," in the C locale.

char *mon_thousands_sep
Similar to thousands_sep, but used for amounts of money. "," in the C locale.

char *mon_grouping
Similar to grouping, but used for amounts of money. "," in the C locale.

char *positive_sign
A string to flag positive amounts of money when formatting. "," in the C locale.

char *negative_sign
A string to flag negative amounts of money when formatting. "," in the C locale.

char int_frac_digits
The number of digits to display when formatting amounts of money to international conventions. CHAR_MAX (the largest number representable as a char) in the C locale.
char frac_digits
The number of digits to display when formatting amounts of money to local conventions. CHAR_MAX in the C locale.

char p_cs_precedes
1 indicates the local currency symbol is used before a positive or zero formatted amount of money; 0 indicates the currency symbol is placed after the formatted number. CHAR_MAX in the C locale.

char p_sep_by_space
1 indicates the local currency symbol must be separated from positive or zero numbers by a space; 0 indicates that it is immediately adjacent to numbers. CHAR_MAX in the C locale.

char n_cs_precedes
1 indicates the local currency symbol is used before a negative formatted amount of money; 0 indicates the currency symbol is placed after the formatted number. CHAR_MAX in the C locale.

char n_sep_by_space
1 indicates the local currency symbol must be separated from negative numbers by a space; 0 indicates that it is immediately adjacent to numbers. CHAR_MAX in the C locale.

char p_sign_posn
Controls the position of the positive sign for numbers representing money. 0 means parentheses surround the number; 1 means the sign is placed before both the number and the currency symbol; 2 means the sign is placed after both the number and the currency symbol; 3 means the sign is placed just before the currency symbol; and 4 means the sign is placed just after the currency symbol. CHAR_MAX in the C locale.

char n_sign_posn
Controls the position of the negative sign for numbers representing money, using the same rules as p_sign_posn. CHAR_MAX in the C locale.
9.1 `setlocale`, `localeconv`—select or query locale

Synopsis

```c
#include <locale.h>

char *setlocale(int category, const char *locale);
lconv *localeconv(void);

char *setlocale_r(void *reent, int category, const char *locale);
lconv *localeconv_r(void *reent);
```

Description

`setlocale` is the facility defined by ANSI C to condition the execution environment for international collating and formatting information; `localeconv` reports on the settings of the current locale.

This is a minimal implementation, supporting only the required "C" value for `locale`; strings representing other locales are not honored unless `_MB_CAPABLE` is defined in which case three new extensions are allowed for LC_CTYPE or LC_MESSAGES only: "C-JIS", "C-EUCJP", "C-SJIS", or "C-ISO-8859-1". ("" is also accepted; it represents the default locale for an implementation, here equivalent to "C").

If you use `NULL` as the `locale` argument, `setlocale` returns a pointer to the string representing the current locale (always "C" in this implementation). The acceptable values for `category` are defined in `locale.h` as macros beginning with "LC_", but this implementation does not check the values you pass in the `category` argument.

`localeconv` returns a pointer to a structure (also defined in `locale.h`) describing the locale-specific conventions currently in effect.

`localeconv_r` and `setlocale_r` are reentrant versions of `localeconv` and `setlocale` respectively. The extra argument `reent` is a pointer to a reentrancy structure.

Returns

`setlocale` returns either a pointer to a string naming the locale currently in effect (always "C" for this implementation, or, if the locale request cannot be honored, `NULL`.

`localeconv` returns a pointer to a structure of type `lconv`, which describes the formatting and collating conventions in effect (in this implementation, always those of the C locale).

Portability

ANSI C requires `setlocale`, but the only locale required across all implementations is the C locale.

No supporting OS subroutines are required.
10 Reentrancy

Reentrancy is a characteristic of library functions which allows multiple processes to use the same address space with assurance that the values stored in those spaces will remain constant between calls. The Red Hat newlib implementation of the library functions ensures that whenever possible, these library functions are reentrant. However, there are some functions that can not be trivially made reentrant. Hooks have been provided to allow you to use these functions in a fully reentrant fashion.

These hooks use the structure `_reent` defined in `reent.h`. A variable defined as `struct _reent` is called a reentrancy structure. All functions which must manipulate global information are available in two versions. The first version has the usual name, and uses a single global instance of the reentrancy structure. The second has a different name, normally formed by prepending `_` and appending `'_r'`, and takes a pointer to the particular reentrancy structure to use.

For example, the function `fopen` takes two arguments, `file` and `mode`, and uses the global reentrancy structure. The function `_fopen_r` takes the arguments, `struct _reent`, which is a pointer to an instance of the reentrancy structure, `file` and `mode`.

There are two versions of `struct _reent`, a normal one and one for small memory systems, controlled by the `_REENT_SMALL` definition from the (automatically included) `<sys/config.h>`.

Each function which uses the global reentrancy structure uses the global variable `_impure_ptr`, which points to a reentrancy structure.

This means that you have two ways to achieve reentrancy. Both require that each thread of execution control initialize a unique global variable of type `struct _reent`:

1. Use the reentrant versions of the library functions, after initializing a global reentrancy structure for each process. Use the pointer to this structure as the extra argument for all library functions.
2. Ensure that each thread of execution control has a pointer to its own unique reentrancy structure in the global variable `_impure_ptr`, and call the standard library subroutines.

The following functions are provided in both reentrant and non-reentrant versions.

 Equivalent for errno variable:

```
_errno_r
```

Locale functions:

```
_localeconv_r _setlocale_r
```

Equivalents for stdio variables:

```
_stdin_r _stdout_r _stderr_r
```
Stdio functions:
  _fdopen_r _fopen_r _gets_r _iprintf_r _mkstemp_r _mktemp_t
  _perror_r _putchar_r _remove_r _rename_r _snprintf_r _sprintf_r
  _tempnam_r _tmpnam_r _tmpfile_r _vsprintf_r _tmpfile_r __vfprintf_r
  _vsnprintf_r __vfprintf_r

Signal functions:
  _init_signal_r _kill_r _raise_r
  __sigtramp_r

Stdlib functions:
  _calloc_r _calloc_r _malloc_r _malloc_stats_r
  _mblen_r _mbstowcs_r _memalign_r _mblen_r
  _setenv_r _mblen_r _mblen_r _mblen_r
  _strdup_r _strtok_r
  _setenv_r _mblen_r _mblen_r _mblen_r
  _setenv_r _mblen_r _mblen_r _mblen_r

String functions:
  _strdup_r _strtok_r

System functions:
  _close_r _execve_r _fork_r _fstat_r _gettimeofday_r
  _link_r _lseek_r _open_r _read_r _stat_r
  _unlink_r _wait_r _write_r _sbrk_r _times_r

Time function:
  _asctime_r
11 Miscellaneous Macros and Functions

This chapter describes miscellaneous routines not covered elsewhere.
11.1 ffs—find first bit set in a word

Synopsis

```c
int ffs(int word);
```

Description

ffs returns the first bit set in a word.

Returns

ffs returns 0 if c is 0, 1 if c is odd, 2 if c is a multiple of 2, etc.

Portability

ffs is not ANSI C.

No supporting OS subroutines are required.
11.2 unctrl—get printable representation of a character

Synopsis

```
#include <unctrl.h>
char *unctrl(int c);
int unctrl(len(int c));
```

Description
unctrl is a macro which returns the printable representation of \( c \) as a string. unctrl(len is a macro which returns the length of the printable representation of \( c \).

Returns
unctrl returns a string of the printable representation of \( c \).
unctrl(len returns the length of the string which is the printable representation of \( c \).

Portability
unctrl and unctrl(len are not ANSI C.
No supporting OS subroutines are required.
12 System Calls

The C subroutine library depends on a handful of subroutine calls for operating system services. If you use the C library on a system that complies with the POSIX.1 standard (also known as IEEE 1003.1), most of these subroutines are supplied with your operating system.

If some of these subroutines are not provided with your system—in the extreme case, if you are developing software for a “bare board” system, without an OS—you will at least need to provide do-nothing stubs (or subroutines with minimal functionality) to allow your programs to link with the subroutines in libc.a.

12.1 Definitions for OS interface

This is the complete set of system definitions (primarily subroutines) required; the examples shown implement the minimal functionality required to allow libc to link, and fail gracefully where OS services are not available.

Graceful failure is permitted by returning an error code. A minor complication arises here: the C library must be compatible with development environments that supply fully functional versions of these subroutines. Such environments usually return error codes in a global errno. However, the Red Hat newlib C library provides a macro definition for errno in the header file ‘errno.h’, as part of its support for reentrant routines (see Chapter 10 [Reentrancy], page 271).

The bridge between these two interpretations of errno is straightforward: the C library routines with OS interface calls capture the errno values returned globally, and record them in the appropriate field of the reentrancy structure (so that you can query them using the errno macro from ‘errno.h’).

This mechanism becomes visible when you write stub routines for OS interfaces. You must include ‘errno.h’, then disable the macro, like this:

```c
#include <errno.h>
#undef errno
extern int errno;
```

The examples in this chapter include this treatment of errno.

_exit Exit a program without cleaning up files. If your system doesn’t provide this, it is best to avoid linking with subroutines that require it (exit, system).

close Close a file. Minimal implementation:

```c
int close(int file) {
    return -1;
}
```

environ A pointer to a list of environment variables and their values. For a minimal environment, this empty list is adequate:

```c
char *__env[1] = { 0 };
char **environ = __env;
```

eexecve Transfer control to a new process. Minimal implementation (for a system without processes):
fork Create a new process. Minimal implementation (for a system without processes):

```c
#include <errno.h>
#undef errno
extern int errno;
int fork(void) {
    errno = EAGAIN;
    return -1;
}
```

fstat Status of an open file. For consistency with other minimal implementations in these examples, all files are regarded as character special devices. The ‘sys/stat.h’ header file required is distributed in the ‘include’ subdirectory for this C library.

```c
#include <sys/stat.h>
int fstat(int file, struct stat *st) {
    st->st_mode = S_IFCHR;
    return 0;
}
```

g getpid Process-ID; this is sometimes used to generate strings unlikely to conflict with other processes. Minimal implementation, for a system without processes:

```c
int getpid(void) {
    return 1;
}
```

isatty Query whether output stream is a terminal. For consistency with the other minimal implementations, which only support output to stdout, this minimal implementation is suggested:

```c
int isatty(int file) {
    return 1;
}
```

kill Send a signal. Minimal implementation:

```c
#include <errno.h>
#undef errno
extern int errno;
int kill(int pid, int sig) {
    errno = EINVAL;
    return -1;
}
```
link  Establish a new name for an existing file. Minimal implementation:

```c
#include <errno.h>
#undef errno
extern int errno;
int link(char *old, char *new) {
    errno = EMLINK;
    return -1;
}
```

lseek Set position in a file. Minimal implementation:

```c
int lseek(int file, int ptr, int dir) {
    return 0;
}
```

open Open a file. Minimal implementation:

```c
int open(const char *name, int flags, int mode) {
    return -1;
}
```

read Read from a file. Minimal implementation:

```c
int read(int file, char *ptr, int len) {
    return 0;
}
```

sbrk Increase program data space. As malloc and related functions depend on this, it is useful to have a working implementation. The following suffices for a standalone system; it exploits the symbol _end automatically defined by the GNU linker.

```c
caddr_t sbrk(int incr) {
    extern char _end; /* Defined by the linker */
    static char *heap_end;
    char *prev_heap_end;
    if (heap_end == 0) {
        heap_end = &_end;
    }
    prev_heap_end = heap_end;
    if (heap_end + incr > stack_ptr) {
        write (1, "Heap and stack collision\n", 25);
        abort ();
    }
    heap_end += incr;
    return (caddr_t) prev_heap_end;
}
```

stat Status of a file (by name). Minimal implementation:

```c
int stat(char *file, struct stat *st) {
```
st->st_mode = S_IFCHR;
    return 0;
}

times Timing information for current process. Minimal implementation:

    int times(struct tms *buf) {
        return -1;
    }

unlink Remove a file's directory entry. Minimal implementation:

    #include <errno.h>
    #undef errno
    extern int errno;
    int unlink(char *name) {
        errno = ENOENT;
        return -1;
    }

wait Wait for a child process. Minimal implementation:

    #include <errno.h>
    #undef errno
    extern int errno;
    int wait(int *status) {
        errno = ECHILD;
        return -1;
    }

write Write to a file. ‘libc’ subroutines will use this system routine for output to all files, including stdout—so if you need to generate any output, for example to a serial port for debugging, you should make your minimal write capable of doing this. The following minimal implementation is an incomplete example; it relies on a outbyte subroutine (not shown; typically, you must write this in assembler from examples provided by your hardware manufacturer) to actually perform the output.

    int write(int file, char *ptr, int len) {
        int todo;

        for (todo = 0; todo < len; todo++) {
            outbyte (*ptr++);
        }
        return len;
    }
12.2 Reentrant covers for OS subroutines

Since the system subroutines are used by other library routines that require reentrancy, ‘libc.a’ provides cover routines (for example, the reentrant version of fork is _fork_r). These cover routines are consistent with the other reentrant subroutines in this library, and achieve reentrancy by using a reserved global data block (see Chapter 10 [Reentrancy], page 271).

_open_r A reentrant version of open. It takes a pointer to the global data block, which holds errno.

    int _open_r(void *reent,
               const char *file, int flags, int mode);

_close_r A reentrant version of close. It takes a pointer to the global data block, which holds errno.

    int _close_r(void *reent, int fd);

_lseek_r A reentrant version of lseek. It takes a pointer to the global data block, which holds errno.

    off_t _lseek_r(void *reent,
                   int fd, off_t pos, int whence);

_read_r A reentrant version of read. It takes a pointer to the global data block, which holds errno.

    long _read_r(void *reent,
                 int fd, void *buf, size_t cnt);

_write_r A reentrant version of write. It takes a pointer to the global data block, which holds errno.

    long _write_r(void *reent,
                  int fd, const void *buf, size_t cnt);

_fork_r A reentrant version of fork. It takes a pointer to the global data block, which holds errno.

    int _fork_r(void *reent);

_wait_r A reentrant version of wait. It takes a pointer to the global data block, which holds errno.

    int _wait_r(void *reent, int *status);

_stat_r A reentrant version of stat. It takes a pointer to the global data block, which holds errno.

    int _stat_r(void *reent,
                const char *file, struct stat *pstat);

_fstat_r A reentrant version of fstat. It takes a pointer to the global data block, which holds errno.

    int _fstat_r(void *reent,
                 int fd, struct stat *pstat);

_link_r A reentrant version of link. It takes a pointer to the global data block, which holds errno.
int _link_r(void *reent,
    const char *old, const char *new);

unlink_r
A reentrant version of unlink. It takes a pointer to the global data block, which holds errno.

int _unlink_r(void *reent, const char *file);

_sbrk_r
A reentrant version of sbrk. It takes a pointer to the global data block, which holds errno.

char *_sbrk_r(void *reent, size_t incr);
13 Variable Argument Lists

The `printf` family of functions is defined to accept a variable number of arguments, rather than a fixed argument list. You can define your own functions with a variable argument list, by using macro definitions from either `stdarg.h` (for compatibility with ANSI C) or from `varargs.h` (for compatibility with a popular convention prior to ANSI C).

13.1 ANSI-standard macros, `stdarg.h`

In ANSI C, a function has a variable number of arguments when its parameter list ends in an ellipsis (...). The parameter list must also include at least one explicitly named argument; that argument is used to initialize the variable list data structure.

ANSI C defines three macros (`va_start`, `va_arg`, and `va_end`) to operate on variable argument lists. `stdarg.h` also defines a special type to represent variable argument lists: this type is called `va_list`. 
13.1.1 Initialize variable argument list

Synopsis

```c
#include <stdarg.h>
void va_start(va_list ap, rightmost);
```

Description

Use `va_start` to initialize the variable argument list `ap`, so that `va_arg` can extract values from it. `rightmost` is the name of the last explicit argument in the parameter list (the argument immediately preceding the ellipsis ‘...’ that flags variable arguments in an ANSI C function header). You can only use `va_start` in a function declared using this ellipsis notation (not, for example, in one of its subfunctions).

Returns

`va_start` does not return a result.

Portability

ANSI C requires `va_start`. 
13.1.2 Extract a value from argument list

Synopsis

```c
#include <stdarg.h>

type va_arg(va_list ap, type);
```

Description

`va_arg` returns the next unprocessed value from a variable argument list `ap` (which you must previously create with `va_start`). Specify the type for the value as the second parameter to the macro, `type`.

You may pass a `va_list` object `ap` to a subfunction, and use `va_arg` from the subfunction rather than from the function actually declared with an ellipsis in the header; however, in that case you may only use `va_arg` from the subfunction. ANSI C does not permit extracting successive values from a single variable-argument list from different levels of the calling stack.

There is no mechanism for testing whether there is actually a next argument available; you might instead pass an argument count (or some other data that implies an argument count) as one of the fixed arguments in your function call.

Returns

`va_arg` returns the next argument, an object of type `type`.

Portability

ANSI C requires `va_arg`.
13.1.3 Abandon a variable argument list

Synopsis

```c
#include <stdarg.h>
void va_end(va_list ap);
```

Description

Use `va_end` to declare that your program will not use the variable argument list `ap` any further.

Returns

`va_end` does not return a result.

Portability

ANSI C requires `va_end`.

13.2 Traditional macros, ‘varargs.h’

If your C compiler predates ANSI C, you may still be able to use variable argument lists using the macros from the ‘varargs.h’ header file. These macros resemble their ANSI counterparts, but have important differences in usage. In particular, since traditional C has no declaration mechanism for variable argument lists, two additional macros are provided simply for the purpose of defining functions with variable argument lists.

As with ‘stdarg.h’, the type `va_list` is used to hold a data structure representing a variable argument list.
13.2.1 Declare variable arguments

Synopsis

```c
#include <varargs.h>
function(va_alist)
  va_dcl
```

Description

To use the `varargs.h` version of variable argument lists, you must declare your function with a call to the macro `va_alist` as its argument list, and use `va_dcl` as the declaration. *Do not use a semicolon after va_dcl.*

Returns

These macros cannot be used in a context where a return is syntactically possible.

Portability

`va_alist` and `va_dcl` were the most widespread method of declaring variable argument lists prior to ANSI C.
13.2.2 Initialize variable argument list

Synopsis
   #include <varargs.h>
   va_list ap;
   va_start(ap);

Description
With the ‘varargs.h’ macros, use va_start to initialize a data structure ap to permit
manipulating a variable argument list. ap must have the type va_alist.

Returns
va_start does not return a result.

Portability
va_start is also defined as a macro in ANSI C, but the definitions are incompatible; the
ANSI version has another parameter besides ap.
13.2.3 Extract a value from argument list

Synopsis

```c
#include <varargs.h>
type va_arg(va_list ap, type);
```

Description
va_arg returns the next unprocessed value from a variable argument list `ap` (which you must previously create with va_start). Specify the type for the value as the second parameter to the macro, `type`.

Returns
va_arg returns the next argument, an object of type `type`.

Portability
The va_arg defined in ‘varargs.h’ has the same syntax and usage as the ANSI C version from ‘stdarg.h’.
13.2.4 Abandon a variable argument list

Synopsis

```c
#include <varargs.h>
va_end(va_list ap);
```

Description

Use `va_end` to declare that your program will not use the variable argument list `ap` any further.

Returns

`va_end` does not return a result.

Portability

The `va_end` defined in ‘varargs.h’ has the same syntax and usage as the ANSI C version from ‘stdarg.h’.
Index

-               _getchar_unlocked_r
__env_lock      _gets_r
__env_unlock    _getwc_r
__malloc_lock   _getwchar_r
__malloc_unlock _impure_ptr
__tz_lock       _fprintf_r
__tz_unlock     _iscnfn_r
_ascftime_r     __link_r
_asmprintf_r     __localeconv_r
_asmprintf_r     _lseek_r
_asmprintf_r     _localtime_r
_atoi_r         _localeconv_r
_atol_r         _malloc_r
_atoll_r        _malloc_stats_r
calloc_r        _malloc_usable_size_r
_close_r        _malloc_opt_r
_dprintf_r      _malloc_stats_r
_eexit          _malloc_opt_r
Exit            _malloc_opt_r
_fclose_r       _malloc_opt_r
_fcloseall_r    _ malloc_opt_r
_fflush_r       _malloc_opt_r
_fgetc_r        _malloc_opt_r
_fgetpos_r      _malloc_opt_r
_fgets_r        _malloc_opt_r
_fhgetwc_r      _malloc_opt_r
_fhgetws_r      _malloc_opt_r
_fiprintf_r     _malloc_opt_r
_fiscanf_r      _malloc_opt_r
_fopen_r        _malloc_opt_r
_fork_r         _malloc_opt_r
_fprintf_r      _malloc_opt_r
_fputc_r        _malloc_opt_r
_fputs_r        _malloc_opt_r
_fputwc_r       _malloc_opt_r
_fputws_r       _malloc_opt_r
_FREALIGN       _malloc_opt_r
_free_r         _malloc_opt_r
_freeopen_r     _malloc_opt_r
_fseek_r        _malloc_opt_r
_fseeko_r       _malloc_opt_r
_fsetpos_r      _malloc_opt_r
_fsetpos_r      _malloc_opt_r
_fsetpos_r      _malloc_opt_r
_fstat_r        _malloc_opt_r
_ftell_r        _malloc_opt_r
_ftello_r       _malloc_opt_r
_fwide_r        _malloc_opt_r
_fwrite_r       _malloc_opt_r
_getc_r         _malloc_opt_r
_getc_unlocked_r _malloc_opt_r
_getchar_r      _malloc_opt_r
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>atoll</td>
<td>277</td>
</tr>
<tr>
<td>atoff</td>
<td>10</td>
</tr>
<tr>
<td>atoi</td>
<td>11</td>
</tr>
<tr>
<td>atol</td>
<td>11</td>
</tr>
<tr>
<td>atof</td>
<td>10</td>
</tr>
<tr>
<td>assert</td>
<td>8</td>
</tr>
<tr>
<td>atexit</td>
<td>9</td>
</tr>
<tr>
<td>a64l</td>
<td>5</td>
</tr>
<tr>
<td>abort</td>
<td>6</td>
</tr>
<tr>
<td>abs</td>
<td>7</td>
</tr>
<tr>
<td>asctime</td>
<td>254</td>
</tr>
<tr>
<td>asiprintf</td>
<td>153</td>
</tr>
<tr>
<td>asniprintf</td>
<td>153</td>
</tr>
<tr>
<td>asnprintf</td>
<td>155</td>
</tr>
<tr>
<td>asprintf</td>
<td>155</td>
</tr>
<tr>
<td>bcmp</td>
<td>176</td>
</tr>
<tr>
<td>bzero</td>
<td>178</td>
</tr>
<tr>
<td>calloc</td>
<td>13</td>
</tr>
<tr>
<td>clearerr</td>
<td>94</td>
</tr>
<tr>
<td>clock</td>
<td>255</td>
</tr>
<tr>
<td>close</td>
<td>277</td>
</tr>
<tr>
<td>ctime</td>
<td>256</td>
</tr>
<tr>
<td>ctime_r</td>
<td>256</td>
</tr>
<tr>
<td>difftime</td>
<td>257</td>
</tr>
<tr>
<td>div</td>
<td>14</td>
</tr>
<tr>
<td>dprintf</td>
<td>96</td>
</tr>
<tr>
<td>drand48</td>
<td>35</td>
</tr>
<tr>
<td>execve</td>
<td>277</td>
</tr>
<tr>
<td>exit</td>
<td>19</td>
</tr>
<tr>
<td>extra argument, reentrant fns</td>
<td>271</td>
</tr>
<tr>
<td>fclose</td>
<td>97</td>
</tr>
<tr>
<td>fcloseall</td>
<td>98</td>
</tr>
<tr>
<td>fcvt</td>
<td>15</td>
</tr>
<tr>
<td>fcvtbuf</td>
<td>17</td>
</tr>
<tr>
<td>fcvtf</td>
<td>15</td>
</tr>
<tr>
<td>environ</td>
<td>20, 277</td>
</tr>
<tr>
<td>erand48</td>
<td>35</td>
</tr>
<tr>
<td>errno global vs macro</td>
<td>277</td>
</tr>
<tr>
<td>execve global vs macro</td>
<td>277</td>
</tr>
<tr>
<td>exit</td>
<td>19</td>
</tr>
<tr>
<td>extra argument, reentrant fns</td>
<td>271</td>
</tr>
<tr>
<td>fclose</td>
<td>97</td>
</tr>
<tr>
<td>fcloseall</td>
<td>98</td>
</tr>
<tr>
<td>fcvt</td>
<td>15</td>
</tr>
<tr>
<td>fcvtbuf</td>
<td>17</td>
</tr>
<tr>
<td>fcvtf</td>
<td>15</td>
</tr>
<tr>
<td>fdopen</td>
<td>99</td>
</tr>
<tr>
<td>feof</td>
<td>100</td>
</tr>
<tr>
<td>ferror</td>
<td>101</td>
</tr>
<tr>
<td>fflush</td>
<td>102</td>
</tr>
<tr>
<td>ffs</td>
<td>274</td>
</tr>
<tr>
<td>fgetc</td>
<td>103</td>
</tr>
<tr>
<td>fgetpos</td>
<td>104</td>
</tr>
<tr>
<td>fgets</td>
<td>105</td>
</tr>
<tr>
<td>fgetwc</td>
<td>106</td>
</tr>
<tr>
<td>fgetws</td>
<td>107</td>
</tr>
<tr>
<td>fileno</td>
<td>108</td>
</tr>
<tr>
<td>fprintf</td>
<td>153</td>
</tr>
<tr>
<td>fscanf</td>
<td>154</td>
</tr>
<tr>
<td>fmopen</td>
<td>109</td>
</tr>
<tr>
<td>fopen</td>
<td>110</td>
</tr>
<tr>
<td>fopencookie</td>
<td>112</td>
</tr>
<tr>
<td>fork</td>
<td>278</td>
</tr>
<tr>
<td>fprintf</td>
<td>155</td>
</tr>
</tbody>
</table>
### Index

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fputc</code></td>
<td>114</td>
</tr>
<tr>
<td><code>.puts</code></td>
<td>115</td>
</tr>
<tr>
<td><code>fputwc</code></td>
<td>116</td>
</tr>
<tr>
<td><code>fputws</code></td>
<td>117</td>
</tr>
<tr>
<td><code>fread</code></td>
<td>118</td>
</tr>
<tr>
<td><code>free</code></td>
<td>25</td>
</tr>
<tr>
<td><code>freopen</code></td>
<td>119</td>
</tr>
<tr>
<td><code>fopen</code></td>
<td>123</td>
</tr>
<tr>
<td><code>fscanf</code></td>
<td>161</td>
</tr>
<tr>
<td><code>fseek</code></td>
<td>120</td>
</tr>
<tr>
<td><code>fseeko</code></td>
<td>120</td>
</tr>
<tr>
<td><code>fsetpos</code></td>
<td>121</td>
</tr>
<tr>
<td><code>fstat</code></td>
<td>278</td>
</tr>
<tr>
<td><code>ftell</code></td>
<td>122</td>
</tr>
<tr>
<td><code>ftime</code></td>
<td>122</td>
</tr>
<tr>
<td><code>ftime</code></td>
<td>123</td>
</tr>
<tr>
<td><code>fwrite</code></td>
<td>125</td>
</tr>
<tr>
<td><code>fopen</code></td>
<td>123</td>
</tr>
<tr>
<td><code>fwrite</code></td>
<td>126</td>
</tr>
</tbody>
</table>

---

**G**
- `gcvt`     | 16   |
- `gcvtf`    | 16   |
- `getc`     | 127  |
- `getc_unlocked` | 128 |
- `getchar`  | 129  |
- `getchar_unlocked` | 130 |
- `getdelim` | 131  |
- `getenv`   | 20   |
- `getline`  | 132  |
- `getpid`   | 278  |
- `gets`     | 133  |
- `getw`     | 134  |
- `ggetwc`   | 106  |
- `getwchar` | 135  |
- `global reentrancy structure` | 271 |
- `ftime`    | 258  |
- `ftime`    | 258  |

---

**I**
- `index`    | 179  |
- `iprintf`  | 153  |
- `isalnum`  | 60   |
- `isalpha`  | 61   |
- `asci`     | 62   |
- `isatty`   | 278  |
- `iscanf`   | 154  |
- `iscntrl`  | 63   |
- `isdigit`  | 64   |
- `isgraph`  | 66   |
- `islower`  | 65   |
- `isprint`  | 66   |
- `ispunct`  | 67   |
- `isspace`  | 68   |
- `toupper`  | 69   |
- `iswalnum` | 74   |
- `iswalpha` | 75   |
- `iswblank` | 77   |
- `iswctrl`  | 76   |
- `iswctype` | 86   |
- `iswdigit` | 78   |
- `iswgraph` | 79   |
- `iswlower` | 80   |
- `iswprint` | 81   |
- `iswupper` | 82   |
- `isspace`  | 83   |
- `isupper`  | 84   |
- `isxdigit` | 85   |
- `isxdigit` | 70   |

---

**J**
- `jrand48`  | 35   |

---

**K**
- `kill`     | 278  |

---

**L**
- `164a`     | 5    |
- `labs`     | 21   |
- `lcong48`  | 35   |
- `ldiv`     | 22   |
- `link`     | 279  |
- `linking the C library` | 277 |
- `list of reentrant functions` | 271 |
- `llabs`    | 23   |
- `ldiv`     | 24   |
- `localeconv` | 269 |
- `localtime` | 259  |
- `localtime_r` | 259 |
- `lrand48`  | 35   |
- `lseek`    | 279  |

---

**M**
- `malloc`   | 25   |
- `malloc`   | 27   |
- `malloc_stats` | 27   |
- `malloc_usable_size` | 25 |
- `malloc`   | 27   |
- `mblen`    | 29   |
- `mbcsrtowcs` | 30  |
- `mbcsrtowcs` | 30  |
- `mbcsrtowcs` | 31  |
- `mbcsrtowcs` | 31  |
- `memalign` | 25   |
- `memchr`   | 181  |
- `memcmp`   | 182  |
- `memem`    | 184  |
- `memmove`  | 185  |
- `memset`   | 187  |
- `mkstemp`  | 136  |
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>mktemp</td>
<td>136</td>
</tr>
<tr>
<td>mktime</td>
<td>260</td>
</tr>
<tr>
<td>mrand48</td>
<td>35</td>
</tr>
<tr>
<td>N</td>
<td></td>
</tr>
<tr>
<td>nrand48</td>
<td>35</td>
</tr>
<tr>
<td>O</td>
<td></td>
</tr>
<tr>
<td>on_exit</td>
<td>33</td>
</tr>
<tr>
<td>open</td>
<td>279</td>
</tr>
<tr>
<td>open_memstream</td>
<td>137</td>
</tr>
<tr>
<td>open_wmemstream</td>
<td>137</td>
</tr>
<tr>
<td>OS interface subroutines</td>
<td>277</td>
</tr>
<tr>
<td>P</td>
<td></td>
</tr>
<tr>
<td>perror</td>
<td>138</td>
</tr>
<tr>
<td>printf</td>
<td>155</td>
</tr>
<tr>
<td>putc</td>
<td>139</td>
</tr>
<tr>
<td>putc_unlocked</td>
<td>140</td>
</tr>
<tr>
<td>putchar</td>
<td>141</td>
</tr>
<tr>
<td>putchar_unlocked</td>
<td>142</td>
</tr>
<tr>
<td>puts</td>
<td>143</td>
</tr>
<tr>
<td>putw</td>
<td>144</td>
</tr>
<tr>
<td>putwc</td>
<td>116</td>
</tr>
<tr>
<td>putwchar</td>
<td>145</td>
</tr>
<tr>
<td>R</td>
<td></td>
</tr>
<tr>
<td>raise</td>
<td>250</td>
</tr>
<tr>
<td>rand</td>
<td>34</td>
</tr>
<tr>
<td>rand_r</td>
<td>34</td>
</tr>
<tr>
<td>rand48</td>
<td>35</td>
</tr>
<tr>
<td>read</td>
<td>279</td>
</tr>
<tr>
<td>realloc</td>
<td>25</td>
</tr>
<tr>
<td>reallocf</td>
<td>25</td>
</tr>
<tr>
<td>reent.h</td>
<td>271</td>
</tr>
<tr>
<td>reentrancy</td>
<td>271</td>
</tr>
<tr>
<td>reentrancy structure</td>
<td>271</td>
</tr>
<tr>
<td>reentrant function list</td>
<td>271</td>
</tr>
<tr>
<td>remove</td>
<td>146</td>
</tr>
<tr>
<td>rename</td>
<td>147</td>
</tr>
<tr>
<td>rewind</td>
<td>148</td>
</tr>
<tr>
<td>rindex</td>
<td>188</td>
</tr>
<tr>
<td>S</td>
<td></td>
</tr>
<tr>
<td>sbrk</td>
<td>279</td>
</tr>
<tr>
<td>scanf</td>
<td>161</td>
</tr>
<tr>
<td>seed48</td>
<td>35</td>
</tr>
<tr>
<td>setbuf</td>
<td>149</td>
</tr>
<tr>
<td>setbuffer</td>
<td>150</td>
</tr>
<tr>
<td>setlinebuf</td>
<td>151</td>
</tr>
<tr>
<td>setlocale</td>
<td>269</td>
</tr>
<tr>
<td>setvbuf</td>
<td>152</td>
</tr>
<tr>
<td>signal</td>
<td>251</td>
</tr>
<tr>
<td>siprintf</td>
<td>153</td>
</tr>
<tr>
<td>sscanf</td>
<td>154</td>
</tr>
<tr>
<td>snprintf</td>
<td>153</td>
</tr>
<tr>
<td>snprintf</td>
<td>155</td>
</tr>
<tr>
<td>sprintf</td>
<td>155</td>
</tr>
<tr>
<td>srand</td>
<td>34</td>
</tr>
<tr>
<td>srand48</td>
<td>35</td>
</tr>
<tr>
<td>sscanf</td>
<td>161</td>
</tr>
<tr>
<td>stat</td>
<td>279</td>
</tr>
<tr>
<td>stpcpy</td>
<td>189</td>
</tr>
<tr>
<td>stpcpy</td>
<td>190</td>
</tr>
<tr>
<td>strcasecmp</td>
<td>191</td>
</tr>
<tr>
<td>strcasestr</td>
<td>192</td>
</tr>
<tr>
<td>strcat</td>
<td>193</td>
</tr>
<tr>
<td>strchr</td>
<td>194</td>
</tr>
<tr>
<td>strcmp</td>
<td>195</td>
</tr>
<tr>
<td>strcmp</td>
<td>196</td>
</tr>
<tr>
<td>strcoll</td>
<td>196</td>
</tr>
<tr>
<td>strcpy</td>
<td>197</td>
</tr>
<tr>
<td>strcsprintf</td>
<td>198</td>
</tr>
<tr>
<td>strerror</td>
<td>199</td>
</tr>
<tr>
<td>strerror_r</td>
<td>203</td>
</tr>
<tr>
<td>strftime</td>
<td>261</td>
</tr>
<tr>
<td>strlen</td>
<td>204</td>
</tr>
<tr>
<td>strlwr</td>
<td>205</td>
</tr>
<tr>
<td>strncasecmp</td>
<td>206</td>
</tr>
<tr>
<td>strncat</td>
<td>207</td>
</tr>
<tr>
<td>strncpy</td>
<td>208</td>
</tr>
<tr>
<td>strnlen</td>
<td>209</td>
</tr>
<tr>
<td>strpbrk</td>
<td>210</td>
</tr>
<tr>
<td>strrchr</td>
<td>211</td>
</tr>
<tr>
<td>strsep</td>
<td>212</td>
</tr>
<tr>
<td>strspn</td>
<td>215</td>
</tr>
<tr>
<td>strspn</td>
<td>213</td>
</tr>
<tr>
<td>strstr</td>
<td>214</td>
</tr>
<tr>
<td>strtof</td>
<td>37</td>
</tr>
<tr>
<td>strtok</td>
<td>37</td>
</tr>
<tr>
<td>strtok</td>
<td>38</td>
</tr>
<tr>
<td>strtok_r</td>
<td>40</td>
</tr>
<tr>
<td>strtok_r</td>
<td>42</td>
</tr>
<tr>
<td>strtoll</td>
<td>44</td>
</tr>
<tr>
<td>strupr</td>
<td>216</td>
</tr>
<tr>
<td>strtok</td>
<td>217</td>
</tr>
<tr>
<td>stubs</td>
<td>277</td>
</tr>
<tr>
<td>subroutines for OS interface</td>
<td>277</td>
</tr>
<tr>
<td>system</td>
<td>56</td>
</tr>
<tr>
<td>T</td>
<td></td>
</tr>
<tr>
<td>tempnam</td>
<td>167</td>
</tr>
<tr>
<td>time</td>
<td>264</td>
</tr>
<tr>
<td>times</td>
<td>280</td>
</tr>
<tr>
<td>tmpfile</td>
<td>166</td>
</tr>
<tr>
<td>tmpnam</td>
<td>167</td>
</tr>
<tr>
<td>toascii</td>
<td>71</td>
</tr>
<tr>
<td>tolower</td>
<td>72</td>
</tr>
<tr>
<td>toupper</td>
<td>73</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>-----------</td>
<td>-------</td>
</tr>
<tr>
<td>towtrans</td>
<td>90</td>
</tr>
<tr>
<td>towlower</td>
<td>88</td>
</tr>
<tr>
<td>towupper</td>
<td>89</td>
</tr>
<tr>
<td>tzset</td>
<td>266</td>
</tr>
<tr>
<td>U</td>
<td></td>
</tr>
<tr>
<td>unctrl</td>
<td>275</td>
</tr>
<tr>
<td>unctrlle</td>
<td>275</td>
</tr>
<tr>
<td>ungetc</td>
<td>168</td>
</tr>
<tr>
<td>ungetwc</td>
<td>169</td>
</tr>
<tr>
<td>unlinkwc</td>
<td>280</td>
</tr>
<tr>
<td>V</td>
<td></td>
</tr>
<tr>
<td>va_alist</td>
<td>287</td>
</tr>
<tr>
<td>va_arg</td>
<td>285, 289</td>
</tr>
<tr>
<td>va_end</td>
<td>287</td>
</tr>
<tr>
<td>va_start</td>
<td>284, 288</td>
</tr>
<tr>
<td>vasprintf</td>
<td>172</td>
</tr>
<tr>
<td>vasprintf</td>
<td>172</td>
</tr>
<tr>
<td>vsnprintf</td>
<td>170</td>
</tr>
<tr>
<td>vsnprintf</td>
<td>170</td>
</tr>
<tr>
<td>vasprintf</td>
<td>170</td>
</tr>
<tr>
<td>vsnprintf</td>
<td>170</td>
</tr>
<tr>
<td>vasprintf</td>
<td>170</td>
</tr>
<tr>
<td>vsnprintf</td>
<td>170</td>
</tr>
<tr>
<td>vsnprintf</td>
<td>170</td>
</tr>
<tr>
<td>vsnprintf</td>
<td>170</td>
</tr>
<tr>
<td>vsnprintf</td>
<td>170</td>
</tr>
<tr>
<td>vsnprintf</td>
<td>170</td>
</tr>
<tr>
<td>vsnprintf</td>
<td>170</td>
</tr>
<tr>
<td>vsscanf</td>
<td>171</td>
</tr>
<tr>
<td>vscanf</td>
<td>171</td>
</tr>
<tr>
<td>vscanf</td>
<td>171</td>
</tr>
<tr>
<td>vscanf</td>
<td>171</td>
</tr>
<tr>
<td>vscanf</td>
<td>171</td>
</tr>
<tr>
<td>vsscanf</td>
<td>171</td>
</tr>
<tr>
<td>vsscanf</td>
<td>171</td>
</tr>
<tr>
<td>vsscanf</td>
<td>171</td>
</tr>
<tr>
<td>vsscanf</td>
<td>171</td>
</tr>
<tr>
<td>vsscanf</td>
<td>171</td>
</tr>
<tr>
<td>vsscanf</td>
<td>171</td>
</tr>
<tr>
<td>vsscanf</td>
<td>171</td>
</tr>
<tr>
<td>vsscanf</td>
<td>171</td>
</tr>
<tr>
<td>vsscanf</td>
<td>171</td>
</tr>
<tr>
<td>vsscanf</td>
<td>171</td>
</tr>
<tr>
<td>vsscanf</td>
<td>171</td>
</tr>
<tr>
<td>vsscanf</td>
<td>171</td>
</tr>
<tr>
<td>vsscanf</td>
<td>171</td>
</tr>
<tr>
<td>vsscanf</td>
<td>171</td>
</tr>
<tr>
<td>vsscanf</td>
<td>171</td>
</tr>
<tr>
<td>vsscanf</td>
<td>171</td>
</tr>
<tr>
<td>vsscanf</td>
<td>171</td>
</tr>
<tr>
<td>vsscanf</td>
<td>171</td>
</tr>
<tr>
<td>vsscanf</td>
<td>171</td>
</tr>
<tr>
<td>vsscanf</td>
<td>171</td>
</tr>
<tr>
<td>vsscanf</td>
<td>171</td>
</tr>
<tr>
<td>vsscanf</td>
<td>171</td>
</tr>
<tr>
<td>vsscanf</td>
<td>171</td>
</tr>
<tr>
<td>vsscanf</td>
<td>171</td>
</tr>
<tr>
<td>vsscanf</td>
<td>171</td>
</tr>
<tr>
<td>vsscanf</td>
<td>171</td>
</tr>
<tr>
<td>vsscanf</td>
<td>171</td>
</tr>
<tr>
<td>vsscanf</td>
<td>171</td>
</tr>
<tr>
<td>vsscanf</td>
<td>171</td>
</tr>
<tr>
<td>vsscanf</td>
<td>171</td>
</tr>
<tr>
<td>vsscanf</td>
<td>171</td>
</tr>
<tr>
<td>vsscanf</td>
<td>171</td>
</tr>
<tr>
<td>vsscanf</td>
<td>171</td>
</tr>
<tr>
<td>vsscanf</td>
<td>171</td>
</tr>
<tr>
<td>vsscanf</td>
<td>171</td>
</tr>
<tr>
<td>vsscanf</td>
<td>171</td>
</tr>
<tr>
<td>vsscanf</td>
<td>171</td>
</tr>
<tr>
<td>vsscanf</td>
<td>171</td>
</tr>
<tr>
<td>vsscanf</td>
<td>171</td>
</tr>
<tr>
<td>vsscanf</td>
<td>171</td>
</tr>
<tr>
<td>vsscanf</td>
<td>171</td>
</tr>
</tbody>
</table>

W

wait ...................................... 280
wccat .................................. 225
wccoll .................................. 228
wcsln .................................. 239
wcsrtombs ................................ 46
wcsrtombs ................................ 46
wctod .................................. 47
wctof .................................. 47
wctok .................................. 244
wctol .................................. 48
wctoll .................................. 50
wctombs .................................. 57
wctou  .................................. 52
wctou .................................. 54
wcswidth ................................ 245
wcsxfrm .................................. 246
wctomb .................................. 58
wctrans .................................. 91
wctype .................................. 87
wcwidth .................................. 247
write .................................... 280
The body of this manual is set in \texttt{cmr10} at 10.95pt, with headings in \texttt{cmb10 at 10.95pt} and examples in \texttt{cmtt10 at 10.95pt}. \texttt{cmti10 at 10.95pt} and \texttt{cmsl10 at 10.95pt} are used for emphasis.
Table of Contents

1 Introduction ........................................... 1

2 Standard Utility Functions (‘stdlib.h’) .............. 3
  2.1 _Exit—end program execution with no cleanup processing .... 4
  2.2 a64l, l64a—convert between radix-64 ASCII string and long ... 5
  2.3 abort—abnormal termination of a program .................. 6
  2.4 abs—integer absolute value (magnitude) .................... 7
  2.5 assert—macro for debugging diagnostics .................. 8
  2.6 atexit—request execution of functions at program exit ........ 9
  2.7 atof, toff—string to double or float ...................... 10
  2.8 atoi, atol—string to integer .................................. 11
  2.9 atoll—convert a string to a long long integer .......... 12
  2.10 calloc—allocate space for arrays ......................... 13
  2.11 div—divide two integers ..................................... 14
  2.12 ecvt, ecvtf, fcvt, fcvtf—double or float to string .......... 15
  2.13 gcvt, gcvtf—format double or float as string ............. 16
  2.14 ecvtbuf, fcvtbuf—double or float to string ............... 17
  2.15 __environ_lock, __environ_unlock—lock environ variable .... 18
  2.16 exit—end program execution ................................. 19
  2.17 getenv—look up environment variable ....................... 20
  2.18 labs—long integer absolute value ............................ 21
  2.19 ldiv—divide two long integers ............................... 22
  2.20 llabs—compute the absolute value of a long long integer ... 23
  2.21 lldiv—divide two long long integers ......................... 24
  2.22 malloc, realloc, free—manage memory ....................... 25
  2.23 mallinfo, malloc_stats, mallocpt—malloc support .......... 27
  2.24 __malloc_lock, __malloc_unlock—lock malloc pool .......... 28
  2.25 mblen—minimal multibyte length function .................. 29
  2.26 mbsrtowcs, mbstrctowcs—convert a character string to a wide-character string ........................................ 30
  2.27 mbstowcs—minimal multibyte string to wide char converter .. 31
  2.28 mbtowc—minimal multibyte to wide char converter .......... 32
  2.29 on_exit—request execution of function with argument at program exit .................................................. 33
  2.30 rand, srand—pseudo-random numbers ......................... 34
  2.31 rand48, drand48, erand48, lrand48, nrand48, mrand48, 
  jrand48, srand48, seed48, lcong48—pseudo-random number 
  generators and initialization routines .......................... 35
  2.32 strtod, strtof—string to double or float ................. 37
  2.33 strto1—string to long .......................................... 38
  2.34 strto1l—string to long long ................................. 40
  2.35 strtonull—string to unsigned long ............................ 42
  2.36 strtonulll—string to unsigned long long ...................... 44
2.37 wcsrtombs, wcsnrtombs—convert a wide-character string to a character string........................................... 46
2.38 wcstod, wcstof—wide char string to double or float...... 47
2.39 wcstol—wide string to long.................................... 48
2.40 wcstoll—wide string to long long............................ 50
2.41 wcstoul—wide string to unsigned long...................... 52
2.42 wcstoull—wide string to unsigned long long.............. 54
2.43 system—execute command string................................ 56
2.44 wcsxmbms—minimal wide char string to multibyte string converter....................................................... 57
2.45 wcxmb—minimal wide char to multibyte converter......... 58

3 Character Type Macros and Functions
   ('ctype.h').................................................... 59
3.1 isalnum—alphanumeric character predicate................ 60
3.2 isalpha—alphabetic character predicate.................... 61
3.3 isascii—ASCII character predicate.......................... 62
3.4 iscntrl—control character predicate........................ 63
3.5 isdigit—decimal digit predicate.............................. 64
3.6 islower—lowercase character predicate...................... 65
3.7 isprint, isgraph—printable character predicates.......... 66
3.8 ispunct—punctuation character predicate................... 67
3.9 isspace—whitespace character predicate.................... 68
3.10 isupper—uppercase character predicate..................... 69
3.11 isxdigit—hexadecimal digit predicate...................... 70
3.12 toascii—force integers to ASCII range..................... 71
3.13 tolower—translate characters to lowercase............... 72
3.14 toupper—translate characters to uppercase............... 73
3.15 iswalnum—alphanumeric wide character test.............. 74
3.16 iswalpha—alphabetic wide character test................ 75
3.17 iswcntrl—control wide character test..................... 76
3.18 iswblank—blank wide character test......................... 77
3.19 iswdigit—decimal digit wide character test.............. 78
3.20 iswgraph—graphic wide character test..................... 79
3.21 iswlower—lowercase wide character test................... 80
3.22 iswprint—printable wide character test................... 81
3.23 iswpunct—punctuation wide character test............... 82
3.24 iswspace—whitespace wide character test................ 83
3.25 iswupper—uppercase wide character test.................. 84
3.26 iswxdigit—hexadecimal digit wide character test........ 85
3.27 iswctype—extensible wide-character test.................. 86
3.28 wctype—get wide-character classification type........... 87
3.29 towlower—translate wide characters to lowercase......... 88
3.30 towupper—translate wide characters to uppercase......... 89
3.31 towxtrans—extensible wide-character translation........ 90
3.32 wctrans—get wide-character translation type............ 91

(continued)
4 Input and Output (‘stdio.h’)......................... 93
4.1 clearerr—clear file or stream error indicator .......... 94
4.2 dprintf, vdprintf—print to a file descriptor (integer only) .. 95
4.3 dprintf, vdprintf—print to a file descriptor .......... 96
4.4 fclose—close a file .................................. 97
4.5 fcloseall—close all files ................................ 98
4.6 fopen—turn open file into a stream ......................... 99
4.7 ferror—test for end of file ................................ 100
4.8 perror—test whether read/write error has occurred ....... 101
4.9 fflush—flush buffered file output ......................... 102
4.10 fgetc—get a character from a file or stream ............... 103
4.11 fgetpos—record position in a stream or file ............. 104
4.12 fgets—get character string from a file or stream ....... 105
4.13 fgetwc, getwc—get a wide character from a file or stream 106
4.14 fgetws—get wide character string from a file or stream ... 107
4.15 fileno—return file descriptor associated with stream .... 108
4.16 fmemopen—open a stream around a fixed-length string .... 109
4.17 fopen—open a file ...................................... 110
4.18 fopencookie—open a stream with custom callbacks ......... 112
4.19 fprintf—write a character on a stream or file .......... 114
4.20 fputs—write a character string in a file or stream ....... 115
4.21 fputwc, putwc—write a wide character on a stream or file .. 116
4.22 fputws—write a wide character string in a file or stream .... 117
4.23 fread—read array elements from a file ..................... 118
4.24 freopen—open a file using an existing file descriptor .... 119
4.25 fseek, fseeko—set file position .......................... 120
4.26 fsetpos—restore position of a stream or file ............ 121
4.27 ftell, ftello—return position in a stream or file ........ 122
4.28 funopen, fopen, fwopen—open a stream with custom callbacks 123
4.29 fwrite—write array elements ................................ 126
4.30 getc—read a character (macro) ............................ 127
4.31 getc_unlocked—non-thread-safe version of getc (macro) ..... 128
4.32 getchar—read a character (macro) .......................... 129
4.33 getchar_unlocked—non-thread-safe version of getchar (macro) ........................................ 130
4.34 getdelim—read a line up to a specified line delimiter .... 131
4.35 getline—read a line from a file ............................ 132
4.36 gets—get character string (obsolete, use fgets instead) ... 133
4.37 getw—read a word (int) .................................... 134
4.38 getwchar—read a wide character from standard input ....... 135
4.39 mktemp, mkstemp—generate unused file name ............... 136
4.40 open_memstream, open_wmemstream—open a write stream around an arbitrary-length string .................. 137
4.41 perror—print an error message on standard error .......... 138
4.42 putc—write a character (macro) ............................ 139
4.43 putc_unlocked—non-thread-safe version of putc (macro) .... 140
5 Strings and Memory (‘string.h’)

5.1 bcmp—compare two memory areas .................................. 176
5.2 bcopy—copy memory regions .......................................... 177
5.3 bzero—initialize memory to zero .................................... 178
5.4 index—search for character in string ................................. 179
5.5 memccpy—copy memory regions with end-token check .......... 180
5.6 memchr—find character in memory .................................. 181
5.7 memcmp—compare two memory areas .................................. 182
5.8 memcpy—copy memory regions ......................................... 183
5.9 memmem—find memory segment ....................................... 184
5.10 memmove—move possibly overlapping memory .................... 185
5.11 mempcpy—copy memory regions and return end pointer ........ 186
5.12 memset—set an area of memory ...................................... 187
5.13 rindex—reverse search for character in string .................. 188
5.14 stpcpy—copy string returning a pointer to its end ............ 189
## 6 Wide Character Strings (‘wchar.h’) 219

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td><code>wmemchr</code>—find a wide character in memory 220</td>
</tr>
<tr>
<td>6.2</td>
<td><code>wmemcmp</code>—compare wide characters in memory 221</td>
</tr>
<tr>
<td>6.3</td>
<td><code>wmemcpn</code>—copy wide characters in memory 222</td>
</tr>
<tr>
<td>6.4</td>
<td><code>wmemmove</code>—copy wide characters in memory with overlapping areas 223</td>
</tr>
<tr>
<td>6.5</td>
<td><code>wmemset</code>—set wide characters in memory 224</td>
</tr>
<tr>
<td>6.6</td>
<td><code>wcscat</code>—concatenate two wide-character strings 225</td>
</tr>
<tr>
<td>6.7</td>
<td><code>wcschr</code>—wide-character string scanning operation 226</td>
</tr>
<tr>
<td>6.8</td>
<td><code>wcsncmp</code>—compare two wide-character strings 227</td>
</tr>
<tr>
<td>6.9</td>
<td><code>wcscoll</code>—locale-specific wide-character string compare 228</td>
</tr>
<tr>
<td>6.10</td>
<td><code>wcscpy</code>—copy a wide-character string 229</td>
</tr>
<tr>
<td>6.11</td>
<td><code>wcscpy</code>—copy a wide-character string returning a pointer to its end 230</td>
</tr>
<tr>
<td>6.12</td>
<td><code>wcscspn</code>—get length of a complementary wide substring 231</td>
</tr>
<tr>
<td>6.13</td>
<td><code>wcslen</code>—get wide-character string length 234</td>
</tr>
<tr>
<td>6.14</td>
<td><code>wcslen</code>—get wide-character string to specified length 233</td>
</tr>
<tr>
<td>6.15</td>
<td><code>wcslen</code>—get wide-character string length 234</td>
</tr>
</tbody>
</table>
6.16 wcsncat—concatenate part of two wide-character strings 235
6.17 wcsncmp—compare part of two wide-character strings 236
6.18 wcsncpy—copy part of a wide-character string 237
6.19 wcpcpy—copy part of a wide-character string returning a
   pointer to its end 238
6.20 wcsnlen—get fixed-size wide-character string length 239
6.21 wcspbrk—scan wide-character string for a wide-character code
   240
6.22 wcsrchr—wide-character string scanning operation 241
6.23 wcsspn—get length of a wide substring 242
6.24 wcsstr—find a wide-character substring 243
6.25 wcstok—get next token from a string 244
6.26 wcswidth—number of column positions of a wide-character string
   245
6.27 wcsxfrm—locale-specific wide-character string transformation
   246
6.28 wcwidth—number of column positions of a wide-character code
   247

7 Signal Handling (‘signal.h’) 249
7.1 raise—send a signal 250
7.2 signal—specify handler subroutine for a signal 251

8 Time Functions (‘time.h’) 253
8.1 asctime—format time as string 254
8.2 clock—cumulative processor time 255
8.3 ctime—convert time to local and format as string 256
8.4 difftime—subtract two times 257
8.5 gmtime—convert time to UTC traditional form 258
8.6 localtime—convert time to local representation 259
8.7 mktime—convert time to arithmetic representation 260
8.8 strftime—flexible calendar time formatter 261
8.9 time—get current calendar time (as single number) 264
8.10 __tz_lock, __tz_unlock—lock time zone global variables 265
8.11 tzset—set timezone characteristics from TZ environment
   variable 266

9 Locale (‘locale.h’) 267
9.1 setlocale, localeconv—select or query locale 269

10 Reentrancy 271

11 Miscellaneous Macros and Functions 273
11.1 ffs—find first bit set in a word 274
11.2 unctrl—get printable representation of a character 275
# 12 System Calls

12.1 Definitions for OS interface ........................................ 277  
12.2 Reentrant covers for OS subroutines ............................... 281

# 13 Variable Argument Lists

13.1 ANSI-standard macros, ‘stderr.h’ ................................. 283  
13.1.1 Initialize variable argument list ............................... 284  
13.1.2 Extract a value from argument list ............................ 285  
13.1.3 Abandon a variable argument list .............................. 286  
13.2 Traditional macros, ‘varargs.h’ .................................. 286  
13.2.1 Declare variable arguments ................................... 287  
13.2.2 Initialize variable argument list ............................... 288  
13.2.3 Extract a value from argument list ............................ 289  
13.2.4 Abandon a variable argument list .............................. 290

# Index

................................................................. 291