NAME

opendir — open directory

SYNOPSIS

```
#include <dirent.h>

DIR *opendir(const char *dirname);
```

DESCRIPTION

The *opendir*() function shall open a directory stream corresponding to the directory named by the *dirname* argument. The directory stream is positioned at the first entry. If the type **DIR** is implemented using a file descriptor, applications shall only be able to open up to a total of {OPEN_MAX} files and directories.

If the type **DIR** is implemented using a file descriptor, the descriptor shall be obtained as if the O_DIRECTORY flag was passed to *open*().

RETURN VALUE

Upon successful completion, the function shall return a pointer to an object of type **DIR**. Otherwise, the function shall return a null pointer and set *errno* to indicate the error.

ERRORS

The *opendir*() function shall fail if:

EACCES
Search permission is denied for the component of the path prefix of *dirname* or read permission is denied for *dirname*.

ELOOP
A loop exists in symbolic links encountered during resolution of the *dirname* argument.

ENAMETOOLONG
The length of a component of a pathname is longer than {NAME_MAX}.

ENOENT
A component of *dirname* does not name an existing directory or *dirname* is an empty string.

ENOTDIR
A component of *dirname* names an existing file that is neither a directory nor a symbolic link to a directory.

NAME

fprintf, printf, snprintf, sprintf — print formatted output

SYNOPSIS

```
#include <stdio.h>

int fprintf(FILE *restrict stream, const char *restrict format, ...);
int printf(const char *restrict format, ...);
int snprintf(char *restrict s, size_t n,
   const char *restrict format, ...);
int sprintf(char *restrict s, const char *restrict format, ...);
```

DESCRIPTION

The *fprintf*() function shall place output on the named output *stream*. The *printf*() function shall place output on the standard output stream *stdout*. The *sprintf*() function shall place output followed by the null byte, **'\0'**, in consecutive bytes starting at **s*; it is the user's responsibility to ensure that enough space is available.

The *snprintf*() function shall be equivalent to *sprintf*(), with the addition of the *n* argument which states the size of the buffer referred to by *s*. If *n* is zero, nothing shall be written and *s* may be a null pointer. Otherwise, output bytes beyond the *n*-1st shall be discarded instead of being written to the array, and a null byte is written at the end of the bytes actually written into the array.

If copying takes place between objects that overlap as a result of a call to *sprintf*() or *snprintf*(), the results are undefined.

Each of these functions converts, formats, and prints its arguments under control of the *format*. The *format* is a character string, beginning and ending in its initial shift state, if any. The *format* is composed of zero or more directives: *ordinary characters*, which are simply copied to the output stream, and *conversion specifications*, each of which shall result in the fetching of zero or more arguments. The results are undefined if there are insufficient arguments for the *format*. If the *format* is exhausted while arguments remain, the excess arguments shall be evaluated but are otherwise ignored.

Conversions can be applied to the *n*th argument after the *format* in the argument list, rather than to the next unused argument. In this case, the conversion specifier character **%** (see below) is replaced by the sequence "*%n*$", where *n* is a decimal integer in the range [1,{NL_ARGMAX}], giving the position of the argument in the argument list. This feature provides for the definition of format strings that select arguments in an order appropriate to specific languages (see the EXAMPLES section).

The *format* can contain either numbered argument conversion specifications (that is, "*%n*$" and "**m*$"), or unnumbered argument conversion specifications (that is, **%** and *), but not both. The only exception to this is that **%%** can be mixed with the "*%n*$" form. The results of mixing numbered and unnumbered argument specifications in a *format* string are undefined. When numbered argument specifications are used, specifying the *N*th argument requires that all the leading arguments, from the first to the (*N*−*1*)th, are specified in the format string.

In format strings containing the **%** form of conversion specification, each conversion specification uses the first unused argument in the argument list.

Each conversion specification is introduced by the **'%'** character or by the character sequence "*%n*$", after which the following appear in sequence:

* Zero or more *flags* (in any order), which modify the meaning of the conversion specification.
* An optional minimum *field width*.
* An optional *precision* that gives the minimum number of digits to appear for the **d**, **i**, **o**, **u**, **x**, and **X** conversion specifiers
* An optional length modifier that specifies the size of the argument.
* A *conversion specifier* character that indicates the type of conversion to be applied.

The conversion specifiers and their meanings are:

d, i
: The **int** argument shall be converted to a signed decimal in the style "[−]*dddd*". The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision is 1. The result of converting zero with an explicit precision of zero shall be no characters.

c
: The **int** argument shall be converted to an **unsigned char**, and the resulting byte shall be written.

s
: The argument shall be a pointer to an array of **char**. Bytes from the array shall be written up to (but not including) any terminating null byte. If the precision is specified, no more than that many bytes shall be written. If the precision is not specified or is greater than the size of the array, the application shall ensure that the array contains a null byte.

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

If a conversion specification does not match one of the above forms, the behavior is undefined. If any argument is not the correct type for the corresponding conversion specification, the behavior is undefined.

In no case shall a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field shall be expanded to contain the conversion result. Characters generated by *fprintf*() and *printf*() are printed as if *fputc*() had been called.

The last data modification and last file status change timestamps of the file shall be marked for update between the call to a successful execution of *fprintf*() or *printf*() and the next successful completion of a call to *fflush*() or *fclose*() on the same stream or a call to *exit*() or *abort*()

RETURN VALUE

Upon successful completion, the *dprintf*(), *fprintf*(), and *printf*() functions shall return the number of bytes transmitted.

Upon successful completion, the *sprintf*() function shall return the number of bytes written to *s*, excluding the terminating null byte.

Upon successful completion, the *snprintf*() function shall return the number of bytes that would be written to *s* had *n* been sufficiently large excluding the terminating null byte.

If an output error was encountered, these functions shall return a negative value and set *errno* to indicate the error.

If the value of *n* is zero on a call to *snprintf*(), nothing shall be written, the number of bytes that would have been written had *n* been sufficiently large excluding the terminating null shall be returned, and *s* may be a null pointer.

ERRORS

For the conditions under which *fprintf*(), and *printf*() fail and may fail, refer to *fputc*( ) or *fputwc*( ).

In addition, all forms of *fprintf*() shall fail if:

EILSEQ
: A wide-character code that does not correspond to a valid character has been detected.

EOVERFLOW
: The value to be returned is greater than {INT_MAX}.

The *fprintf*(), and *printf*() functions may fail if:

ENOMEM
: Insufficient storage space is available.

The *snprintf*() function shall fail if:

EOVERFLOW
: The value of *n* is greater than {INT_MAX}.

NAME

fstatat, lstat, stat — get file status

SYNOPSIS

```
#include <fcntl.h>
#include <sys/stat.h>

int fstatat(int fd, const char *restrict path,
    struct stat *restrict buf, int flag);
int lstat(const char *restrict path, struct stat *restrict buf);
int stat(const char *restrict path, struct stat *restrict buf);
```

DESCRIPTION

The *stat*() function shall obtain information about the named file and write it to the area pointed to by the *buf* argument. The *path* argument points to a pathname naming a file. Read, write, or execute permission of the named file is not required. An implementation that provides additional or alternate file access control mechanisms may, under implementation-defined conditions, cause *stat*() to fail. In particular, the system may deny the existence of the file specified by *path*.

If the named file is a symbolic link, the *stat*() function shall continue pathname resolution using the contents of the symbolic link, and shall return information pertaining to the resulting file if the file exists.

The *buf* argument is a pointer to a **stat** structure, as defined in the *<sys/stat.h>* header, into which information is placed concerning the file.

The *stat*() function shall update any time-related fields (as described in the Base Definitions volume of POSIX.1-2017, *Section 4.9*, *File Times Update*), before writing into the **stat** structure.

If the named file is a shared memory object, the implementation shall update in the **stat** structure pointed to by the *buf* argument the *st_uid*, *st_gid*, *st_size*, and *st_mode* fields, and only the S_IRUSR, S_IWUSR, S_IRGRP, S_IWGRP, S_IROTH, and S_IWOTH file permission bits need be valid. The implementation may update other fields and flags.

If the named file is a typed memory object, the implementation shall update in the **stat** structure pointed to by the *buf* argument the *st_uid*, *st_gid*, *st_size*, and *st_mode* fields, and only the S_IRUSR, S_IWUSR, S_IRGRP, S_IWGRP, S_IROTH, and S_IWOTH file permission bits need be valid. The implementation may update other fields and flags.

For all other file types defined in this volume of POSIX.1-2017, the structure members *st_mode*, *st_ino*, *st_dev*, *st_uid*, *st_gid*, *st_atim*, *st_ctim*, and *st_mtim* shall have meaningful values and the value of the member *st_nlink* shall be set to the number of links to the file.

The *lstat*() function shall be equivalent to *stat*(), except when *path* refers to a symbolic link. In that case *lstat*() shall return information about the link, while *stat*() shall return information about the file the link references.

For symbolic links, the *st_mode* member shall contain meaningful information when used with the file type macros. The file mode bits in *st_mode* are unspecified. The structure members *st_ino*, *st_dev*, *st_uid*, *st_gid*, *st_atim*, *st_ctim*, and *st_mtim* shall have meaningful values and the value of the *st_nlink* member shall be set to the number of (hard) links to the symbolic link. The value of the *st_size* member shall be set to the length of the pathname contained in the symbolic link not including any terminating null byte.

The *fstatat*() function shall be equivalent to the *stat*() or *lstat*() function, depending on the value of *flag* (see below), except in the case where *path* specifies a relative path. In this case the status shall be retrieved from a file relative to the directory associated with the file descriptor *fd* instead of the current working directory. If the access mode of the open file description associated with the file descriptor is not O_SEARCH, the function shall check whether directory searches are permitted using the current permissions of the directory underlying the file descriptor. If the access mode is O_SEARCH, the function shall not perform the check.

Values for *flag* are constructed by a bitwise-inclusive OR of flags from the following list, defined in *<fcntl.h>*:

AT_SYMLINK_NOFOLLOW
If *path* names a symbolic link, the status of the symbolic link is returned.

If *fstatat*() is passed the special value AT_FDCWD in the *fd* parameter, the current working directory shall be used and the behavior shall be identical to a call to *stat*() or *lstat*() respectively, depending on whether or not the AT_SYMLINK_NOFOLLOW bit is set in *flag*.

RETURN VALUE

Upon successful completion, these functions shall return 0. Otherwise, these functions shall return −1 and set *errno* to indicate the error.

ERRORS

These functions shall fail if:

EACCES
Search permission is denied for a component of the path prefix.

EIO An error occurred while reading from the file system.

ELOOP
A loop exists in symbolic links encountered during resolution of the *path* argument.

ENAMETOOLONG
The length of a component of a pathname is longer than {NAME_MAX}.

ENOENT
A component of *path* does not name an existing file or *path* is an empty string.

ENOTDIR
A component of the path prefix names an existing file that is neither a directory nor a symbolic link to a directory, or the *path* argument contains at least one non-<slash> character and ends with one or more trailing <slash> characters and the last pathname component names an existing file that is neither a directory nor a symbolic link to a directory.

EOVERFLOW
The file size in bytes or the number of blocks allocated to the file or the file serial number cannot be represented correctly in the structure pointed to by *buf*.

EXAMPLES

Obtaining File Status Information

The following example shows how to obtain file status information for a file named **/home/cnd/mod1**. The structure variable *buffer* is defined for the **stat** structure. Error handling is ommitted for brevity.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

struct stat buffer;
int         status;
...
status = stat("/home/cnd/mod1", &buffer);
```


NAME

malloc — a memory allocator

SYNOPSIS

```
#include <stdlib.h>

void *malloc(size_t size);
```

DESCRIPTION

The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2017 defers to the ISO C standard.

The *malloc*() function shall allocate unused space for an object whose size in bytes is specified by *size* and whose value is unspecified.

The order and contiguity of storage allocated by successive calls to *malloc*() is unspecified. The pointer returned if the allocation succeeds shall be suitably aligned so that it may be assigned to a pointer to any type of object and then used to access such an object in the space allocated (until the space is explicitly freed or reallocated). Each such allocation shall yield a pointer to an object disjoint from any other object. The pointer returned points to the start (lowest byte address) of the allocated space. If the space cannot be allocated, a null pointer shall be returned. If the size of the space requested is 0, the behavior is implementation-defined: either a null pointer shall be returned, or the behavior shall be as if the size were some nonzero value, except that the behavior is undefined if the returned pointer is used to access an object.

RETURN VALUE

Upon successful completion with *size* not equal to 0, *malloc*() shall return a pointer to the allocated space. If *size* is 0, either:

* A null pointer shall be returned and *errno* may be set to an implementation-defined value, or
* A pointer to the allocated space shall be returned. The application shall ensure that the pointer is not used to access an object.

Otherwise, it shall return a null pointer and set *errno* to indicate the error.

ERRORS

The *malloc*() function shall fail if:

ENOMEM
Insufficient storage space is available.

The following sections are informative.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

calloc(), *free*(), *getrlimit*(), *posix_memalign*(), *realloc*()

The Base Definitions volume of POSIX.1-2017, **<stdlib.h>**

NAME

memset — set bytes in memory

SYNOPSIS

```
#include <string.h>

void *memset(void *s, int c, size_t n);
```

DESCRIPTION

The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2017 defers to the ISO C standard.

The *memset*() function shall copy *c* (converted to an **unsigned char**) into each of the first *n* bytes of the object pointed to by *s*.

RETURN VALUE

The *memset*() function shall return *s*; no return value is reserved to indicate an error.

ERRORS

No errors are defined.

The following sections are informative.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of POSIX.1-2017, **<string.h>**

COPYRIGHT

NAME

pthread_create — thread creation

SYNOPSIS

```
#include <pthread.h>

int pthread_create(pthread_t *restrict thread,
    const pthread_attr_t *restrict attr,
    void *(*start_routine)(void*), void *restrict arg);
```

DESCRIPTION

The *pthread_create*() function shall create a new thread, with attributes specified by *attr*, within a process. If *attr* is NULL, the default attributes shall be used. If the attributes specified by *attr* are modified later, the thread's attributes shall not be affected. Upon successful completion, *pthread_create*() shall store the ID of the created thread in the location referenced by *thread*.

The thread is created executing *start_routine* with *arg* as its sole argument. If the *start_routine* returns, the effect shall be as if there was an implicit call to *pthread_exit*() using the return value of *start_routine* as the exit status. Note that the thread in which *main*() was originally invoked differs from this. When it returns from *main*(), the effect shall be as if there was an implicit call to *exit*() using the return value of *main*() as the exit status.

The signal state of the new thread shall be initialized as follows:

* The signal mask shall be inherited from the creating thread.
* The set of signals pending for the new thread shall be empty.

The thread-local current locale and the alternate stack shall not be inherited.

The floating-point environment shall be inherited from the creating thread.

If *pthread_create*() fails, no new thread is created and the contents of the location referenced by *thread* are undefined.

If _POSIX_THREAD_CPUTIME is defined, the new thread shall have a CPU-time clock accessible, and the initial value of this clock shall be set to zero.

The behavior is undefined if the value specified by the *attr* argument to *pthread_create*() does not refer to an initialized thread attributes object.

RETURN VALUE

If successful, the *pthread_create*() function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_create*() function shall fail if:

EAGAIN
: The system lacked the necessary resources to create another thread, or the system-imposed limit on the total number of threads in a process {PTHREAD_THREADS_MAX} would be exceeded.

EPERM
: The caller does not have appropriate privileges to set the required scheduling parameters or scheduling policy.

The *pthread_create*() function shall not return an error code of **[EINTR]**.

NAME

pthread_join — wait for thread termination

SYNOPSIS

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **value_ptr);
```

DESCRIPTION

The *pthread_join*() function shall suspend execution of the calling thread until the target *thread* terminates, unless the target *thread* has already terminated. On return from a successful *pthread_join*() call with a non-NULL *value_ptr* argument, the value passed to *pthread_exit*() by the terminating thread shall be made available in the location referenced by *value_ptr*. When a *pthread_join*() returns successfully, the target thread has been terminated. The results of multiple simultaneous calls to *pthread_join*() specifying the same target thread are undefined. If the thread calling *pthread_join*() is canceled, then the target thread shall not be detached.

The behavior is undefined if the value specified by the *thread* argument to *pthread_join*() does not refer to a joinable thread.

RETURN VALUE

If successful, the *pthread_join*() function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_join*() function may fail if:

EDEADLK
A deadlock was detected.

The *pthread_join*() function shall not return an error code of **[EINTR]**.

EXAMPLES

An example of thread creation and deletion follows. Some error handling code is omitted for brevity.

```
typedef struct { int *ar; long n; } subarray;

void * incer(void *arg) {
    long i;

    for (i = 0; i < ((subarray *)arg)->n; i++)
        ((subarray *)arg)->ar[i]++;
}

int main(void) {
    int       ar[1000000];
    pthread_t  th1, th2;
    subarray   sb1, sb2;

    sb1.ar = &ar[0];
    sb1.n  = 500000;
    (void) pthread_create(&th1, NULL, incer, &sb1);

    sb2.ar = &ar[500000];
    sb2.n  = 500000;
    (void) pthread_create(&th2, NULL, incer, &sb2);

    (void) pthread_join(th1, NULL);
    (void) pthread_join(th2, NULL);
    return 0;
}
```


NAME

qsort — sort a table of data

SYNOPSIS

```
#include <stdlib.h>

void qsort(void *base, size_t nel, size_t width,
    int (*compar)(const void *, const void *));
```

DESCRIPTION

The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2017 defers to the ISO C standard.

The *qsort*() function shall sort an array of *nel* objects, the initial element of which is pointed to by *base*. The size of each object, in bytes, is specified by the *width* argument. If the *nel* argument has the value zero, the comparison function pointed to by *compar* shall not be called and no rearrangement shall take place.

The application shall ensure that the comparison function pointed to by *compar* does not alter the contents of the array. The implementation may reorder elements of the array between calls to the comparison function, but shall not alter the contents of any individual element.

When the same objects (consisting of width bytes, irrespective of their current positions in the array) are passed more than once to the comparison function, the results shall be consistent with one another. That is, they shall define a total ordering on the array.

The contents of the array shall be sorted in ascending order according to a comparison function. The *compar* argument is a pointer to the comparison function, which is called with two arguments that point to the elements being compared. The application shall ensure that the function returns an integer less than, equal to, or greater than 0, if the first argument is considered respectively less than, equal to, or greater than the second. If two members compare as equal, their order in the sorted array is unspecified.

RETURN VALUE

The *qsort*() function shall not return a value.

ERRORS

No errors are defined.

The following sections are informative.

EXAMPLES

None.

APPLICATION USAGE

The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

RATIONALE

The requirement that each argument (hereafter referred to as *p)* to the comparison function is a pointer to elements of the array implies that for every call, for each argument separately, all of the following expressions are non-zero:

```
((char *)p - (char *)base) % width == 0
(char *)p >= (char *)base
(char *)p < (char *)base + nel * width
```

NAME

strcmp — compare two strings

SYNOPSIS

```
#include <string.h>

int strcmp(const char *s1, const char *s2);
```

DESCRIPTION

The *strcmp*() function shall compare the string pointed to by *s1* to the string pointed to by *s2*.

The sign of a non-zero return value shall be determined by the sign of the difference between the values of the first pair of bytes (both interpreted as type **unsigned char**) that differ in the strings being compared.

RETURN VALUE

Upon completion, *strcmp*() shall return an integer greater than, equal to, or less than 0, if the string pointed to by *s1* is greater than, equal to, or less than the string pointed to by *s2*, respectively.

ERRORS

No errors are defined.

EXAMPLES

Checking a Password Entry

The following example compares the information read from standard input to the value of the name of the user entry. If the *strcmp*() function returns 0 (indicating a match), a further check will be made to see if the user entered the proper old password. The *crypt*() function shall encrypt the old password entered by the user, using the value of the encrypted password in the **passwd** structure as the salt. If this value matches the value of the encrypted **passwd** in the structure, the entered password *oldpasswd* is the correct user's password. Finally, the program encrypts the new password so that it can store the information in the **passwd** structure.

```
#include <string.h>
#include <unistd.h>
#include <stdio.h>
...
int valid_change;
struct passwd *p;
char user[100];
char oldpasswd[100];
char newpasswd[100];
char savepasswd[100];
...
if (strcmp(p->pw_name, user) == 0) {
    if (strcmp(p->pw_passwd, crypt(oldpasswd, p->pw_passwd)) == 0) {
        strcpy(savepasswd, crypt(newpasswd, user));
        p->pw_passwd = savepasswd;
        valid_change = 1;
    }
    else {
        fprintf(stderr, "Old password is not valid\n");
    }
}
...
```


NAME

strlen, strnlen — get length of fixed size string

SYNOPSIS

```
#include <string.h>

size_t strlen(const char *s);
size_t strnlen(const char *s, size_t maxlen);
```

DESCRIPTION

For *strlen*(): The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2017 defers to the ISO C standard.

The *strlen*() function shall compute the number of bytes in the string to which *s* points, not including the terminating NUL character.

The *strnlen*() function shall compute the smaller of the number of bytes in the array to which *s* points, not including any terminating NUL character, or the value of the *maxlen* argument. The *strnlen*() function shall never examine more than *maxlen* bytes of the array pointed to by *s*.

RETURN VALUE

The *strlen*() function shall return the length of *s*; no return value shall be reserved to indicate an error.

The *strnlen*() function shall return the number of bytes preceding the first null byte in the array to which *s* points, if *s* contains a null byte within the first *maxlen* bytes; otherwise, it shall return *maxlen*.

ERRORS

No errors are defined.

The following sections are informative.

EXAMPLES

Getting String Lengths

The following example sets the maximum length of *key* and *data* by using *strlen*() to get the lengths of those strings.

```
#include <string.h>
...
struct element {
    char *key;
    char *data;
};
...
char *key, *data;
int len;

*keylength = *datalength = 0;
...
if ((len = strlen(key)) > *keylength)
    *keylength = len;
if ((len = strlen(data)) > *datalength)
    *datalength = len;
...
```