**NAME**

fdopendir, opendir — open directory associated with file descriptor

**SYNOPSIS**

#include <dirent.h>

DIR *fdopendir(int *fd*);
DIR *opendir(const char **dirname*);

**DESCRIPTION**

The *fdopendir*() function shall be equivalent to the *opendir*() function except that the directory is specified by a file descriptor rather than by a name. The file offset associated with the file descriptor at the time of the call determines which entries are returned.

Upon successful return from *fdopendir*(), the file descriptor is under the control of the system, and if any attempt is made to close the file descriptor, or to modify the state of the associated description, other than by means of *closedir*(), *readdir*(), *readdir_r*(), *rewinddir*(), or *seekdir*(), the behavior is undefined. Upon calling *closedir*() the file descriptor shall be closed.

It is unspecified whether the FD_CLOEXEC flag will be set on the file descriptor by a successful call to *fdopendir*().

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, these functions shall return a pointer to an object of type **DIR**. Otherwise, these functions shall return a null pointer and set *errno* to indicate the error.

**ERRORS**

The *fdopendir*() function shall fail if:

**EBADF**
The *fd* argument is not a valid file descriptor open for reading.

**ENOTDIR**
The descriptor *fd* is not associated with a directory.

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.

The *opendir*() function may fail if:

**ELOOP**
More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *dirname* argument.





**EMFILE**
All file descriptors available to the process are currently open.

**ENAMETOOLONG**
The length of a pathname exceeds {PATH_MAX}, or pathname resolution of a symbolic link produced an intermediate result with a length that exceeds {PATH_MAX}.

**ENFILE**
Too many files are currently open in the system.

*The following sections are informative.*

**EXAMPLES**

**Find And Open a File**

The following program searches through a given directory looking for files whose name does not begin with a dot and whose size is larger than 1 MiB. Some error handling code is omitted for brevity.

```
#include <stdio.h>
#include <dirent.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <stdint.h>
#include <stdlib.h>
#include <unistd.h>

int
main(int argc, char *argv[])
{
   struct stat statbuf;
   DIR *d;
   struct dirent *dp;
   int dfd, ffd;

   if ((d = fdopendir((dfd = open("./tmp", O_RDONLY)))) == NULL) {
      fprintf(stderr, "Cannot open ./tmp directory\n");
      exit(1);
   }
   while ((dp = readdir(d)) != NULL) {
      if (dp->d_name[0] == '.')
         continue;
      /* there is a possible race condition here as the file
       * could be renamed between the readdir and the open */
      if ((ffd = openat(dfd, dp->d_name, O_RDONLY)) == -1) {
         perror(dp->d_name);
         continue;
      }
      if (fstat(ffd, &statbuf) == 0 && statbuf.st_size > (1024*1024)) {
         /* found it ... */
         printf("%s: %jdK\n", dp->d_name,
            (intmax_t)(statbuf.st_size / 1024));
      }
      close(ffd);
   }
   closedir(d); // note this implicitly closes dfd
   return 0;
}
```

**NAME**

ferror — test error indicator on a stream

**SYNOPSIS**

#include <stdio.h>

int ferror(FILE *_stream_);

**DESCRIPTION**

The *ferror*() function shall test the error indicator for the stream pointed to by *stream*.

The *ferror*() function shall not change the setting of *errno* if *stream* is valid.

**RETURN VALUE**

The *ferror*() function shall return non-zero if and only if the error indicator is set for *stream*.

**ERRORS**

No errors are defined.





**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*$" and "**n*$"), 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**

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

readdir, readdir_r — read a directory

## SYNOPSIS

#include <dirent.h>

struct dirent *readdir(DIR **dirp*);
int readdir_r(DIR *restrict *dirp*, struct dirent *restrict *entry*,
struct dirent **restrict *result*);

## DESCRIPTION

The type **DIR**, which is defined in the <*dirent.h*> header, represents a *directory stream*, which is an ordered sequence of all the directory entries in a particular directory. Directory entries represent files; files may be removed from a directory or added to a directory asynchronously to the operation of *readdir*().

The *readdir*() function shall return a pointer to a structure representing the directory entry at the current position in the directory stream specified by the argument *dirp*, and position the directory stream at the next entry. It shall return a null pointer upon reaching the end of the directory stream. The structure **dirent** defined in the <*dirent.h*> header describes a directory entry. The value of the structure's *d_ino* member shall be set to the file serial number of the file named by the *d_name* member. If the *d_name* member names a symbolic link, the value of the *d_ino* member shall be set to the file serial number of the symbolic link itself.

The *readdir*() function shall not return directory entries containing empty names. If entries for dot or dot-dot exist, one entry shall be returned for dot and one entry shall be returned for dot-dot; otherwise, they shall not be returned.

The application shall not modify the structure to which the return value of *readdir*() points, nor any storage areas pointed to by pointers within the structure. The returned pointer, and pointers within the structure, might be invalidated or the structure or the storage areas might be overwritten by a subsequent call to *readdir*() on the same directory stream. They shall not be affected by a call to *readdir*() on a different directory stream. The returned pointer, and pointers within the structure, might also be invalidated if the calling thread is terminated.

The *readdir*() function may buffer several directory entries per actual read operation; *readdir*() shall mark for update the last data access timestamp of the directory each time the directory is actually read.

After a call to *fork*(), either the parent or child (but not both) may continue processing the directory stream using *readdir*(), *rewinddir*(), or *seekdir*(). If both the parent and child processes use these functions, the result is undefined.

The *readdir*() function need not be thread-safe.

Applications wishing to check for error situations should set *errno* to 0 before calling *readdir*(). If *errno* is set to non-zero on return, an error occurred.

The *readdir_r*() function shall initialize the **dirent** structure referenced by *entry* to represent the directory entry at the current position in the directory stream referred to by *dirp*, store a pointer to this structure at the location referenced by *result*, and position the directory stream at the next entry.

The storage pointed to by *entry* shall be large enough for a **dirent** with an array of **char** *d_name* members containing at least {NAME_MAX}+1 elements.

Upon successful return, the pointer returned at **result* shall have the same value as the argument *entry*. Upon reaching the end of the directory stream, this pointer shall have the value NULL.

The *readdir_r*() function shall not return directory entries containing empty names.

The *readdir_r*() function may buffer several directory entries per actual read operation; *readdir_r*() shall mark for update the last data access timestamp of the directory each time the directory is actually read.

## RETURN VALUE

Upon successful completion, *readdir*() shall return a pointer to an object of type **struct dirent**. When an error is encountered, a null pointer shall be returned and *errno* shall be set to indicate the error. When the

end of the directory is encountered, a null pointer shall be returned and *errno* is not changed.

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

**ERRORS**

These functions shall fail if:

**EOVERFLOW**

One of the values in the structure to be returned cannot be represented correctly.

These functions may fail if:

**EBADF**

The *dirp* argument does not refer to an open directory stream.

**ENOENT**

The current position of the directory stream is invalid.

**EXAMPLES**

The following sample program searches the current directory for each of the arguments supplied on the command line. Some error handling code is omitted for brevity.

```
#include <dirent.h>
#include <errno.h>
#include <stdio.h>
#include <string.h>

static void lookup(const char *arg) {
   DIR *dirp;
   struct dirent *dp;
   if ((dirp = opendir(".")) == NULL) {
      perror("couldn't open '.'");
      return;
   }
   do {
      errno = 0;
      if ((dp = readdir(dirp)) != NULL) {
         if (strcmp(dp->d_name, arg) != 0) continue;
         (void) printf("found %s\n", arg);
         (void) closedir(dirp);
         return;
      }
   } while (dp != NULL);
   if (errno != 0) perror("error reading directory");
   else (void) printf("failed to find %s\n", arg);
   (void) closedir(dirp);
   return;
}

int main(int argc, char *argv[]) {
   for (int i = 1; i < argc; i++) lookup(argv[i]);
   return (0);
}
```





**NAME**

strstr — find a substring

**SYNOPSIS**

#include <string.h>

char *strstr(const char *s1, const char *s2);

**DESCRIPTION**

The *strstr*() function shall locate the first occurrence in the string pointed to by *s1* of the sequence of bytes (excluding the terminating NUL character) in the string pointed to by *s2*.

**RETURN VALUE**

Upon successful completion, *strstr*() shall return a pointer to the located string or a null pointer if the string is not found.

If *s2* points to a string with zero length, the function shall return *s1*.

**ERRORS**

No errors are defined.