## NAME

gets, fgets – get a string from a stream
fputs, puts – output of strings

## SYNOPSIS

**#include <stdio.h>**

**char *gets(char ***_s_**);**

**char *fgets(char ***_s_**, int** _n_**, FILE ***_stream_**);**

**int fputs(const char ***_s_**, FILE ***_stream_**);**

**int puts(const char ***_s_**);**

## DESCRIPTION gets/fgets

The **gets**( ) function reads characters from the standard input stream (see **intro**(3)), **stdin**, into the array pointed to by _s_, until a newline character is read or an end-of-file condition is encountered. The newline character is discarded and the string is terminated with a null character.

The **fgets**( ) function reads characters from the _stream_ into the array pointed to by _s_, until _n_−1 characters are read, or a newline character is read and transferred to _s_, or an end-of-file condition is encountered. The string is then terminated with a null character.

When using **gets**( ), if the length of an input line exceeds the size of _s_, indeterminate behavior may result. For this reason, it is strongly recommended that **gets**( ) be avoided in favor of **fgets**( ).

## RETURN VALUES

If end-of-file is encountered and no characters have been read, no characters are transferred to _s_ and a null pointer is returned. If a read error occurs, such as trying to use these functions on a file that has not been opened for reading, a null pointer is returned and the error indicator for the stream is set. If end-of-file is encountered, the **EOF** indicator for the stream is set. Otherwise _s_ is returned.

## ERRORS

The **gets**( ) and **fgets**( ) functions will fail if data needs to be read and:

**EOVERFLOW** The file is a regular file and an attempt was made to read at or beyond the offset maximum associated with the corresponding _stream_.

## DESCRIPTION puts/fputs

**fputs**() writes the string _s_ to _stream_, without its trailing **'\0'**.

**puts**() writes the string _s_ and a trailing newline to _stdout_.

Calls to the functions described here can be mixed with each other and with calls to other output functions from the **stdio** library for the same output stream.

## RETURN VALUE

**puts**() and **fputs**() return a non - negative number on success, or **EOF** on error.

## NAME

calloc, malloc, free, realloc – Allocate and free dynamic memory

## SYNOPSIS

**#include <stdlib.h>**

**void *calloc(size_t** _nmemb_**, size_t** _size_**);**
**void *malloc(size_t** _size_**);**
**void free(void ***_ptr_**);**
**void *realloc(void ***_ptr_**, size_t** _size_**);**

## DESCRIPTION

**calloc()** allocates memory for an array of _nmemb_ elements of _size_ bytes each and returns a pointer to the allocated memory. The memory is set to zero.

**malloc()** allocates _size_ bytes and returns a pointer to the allocated memory. The memory is not cleared.

**free**() frees the memory space pointed to by _ptr_, which must have been returned by a previous call to **malloc()**, **calloc()** or **realloc()**. Otherwise, or if **free(**_ptr_**)** has already been called before, undefined behaviour occurs. If _ptr_ is **NULL**, no operation is performed.

**realloc()** changes the size of the memory block pointed to by _ptr_ to _size_ bytes. The contents will be unchanged to the minimum of the old and new sizes; newly allocated memory will be uninitialized. If _ptr_ is **NULL**, the call is equivalent to **malloc(size)**; if size is equal to zero, the call is equivalent to **free(**_ptr_**)**. Unless _ptr_ is **NULL**, it must have been returned by an earlier call to **malloc()**, **calloc()** or **realloc()**.

## RETURN VALUE

For **calloc()** and **malloc()**, the value returned is a pointer to the allocated memory, which is suitably aligned for any kind of variable, or **NULL** if the request fails.

**free()** returns no value.

**realloc()** returns a pointer to the newly allocated memory, which is suitably aligned for any kind of variable and may be different from _ptr_, or **NULL** if the request fails. If _size_ was equal to 0, either NULL or a pointer suitable to be passed to _free_() is returned. If **realloc()** fails the original block is left untouched - it is not freed or moved.

## CONFORMING TO

ANSI-C

## SEE ALSO

**brk**(2), **posix_memalign**(3)

## NAME

strcmp, strncmp – compare two strings

## SYNOPSIS

**#include <string.h>**

**int strcmp(const char ***_s1_**, const char ***_s2_**);**

**int strncmp(const char ***_s1_**, const char ***_s2_**, size_t** _n_**);**

## DESCRIPTION

The **strcmp**() function compares the two strings _s1_ and _s2_. It returns an integer less than, equal to, or greater than zero if _s1_ is found, respectively, to be less than, to match, or be greater than _s2_.

The **strncmp**() function is similar, except it only compares the first (at most) _n_ characters of _s1_ and _s2_.

## RETURN VALUE

The **strcmp**() and **strncmp**() functions return an integer less than, equal to, or greater than zero if _s1_ (or the first _n_ bytes thereof) is found, respectively, to be less than, to match, or be greater than _s2_.

## CONFORMING TO

SVr4, 4.3BSD, C89, C99.

## SEE ALSO

**bcmp**(3), **memcmp**(3), **strcasecmp**(3), **strcoll**(3), **strncasecmp**(3), **wcscmp**(3), **wcsncmp**(3)

## NAME

strtok, strtok_r – extract tokens from strings

## SYNOPSIS

**#include <string.h>**

**char *strtok(char ***_str_**, const char ***_delim_**);**

**char *strtok_r(char ***_str_**, const char ***_delim_**, char ****_saveptr_**);**

## DESCRIPTION

The **strtok**() function breaks a string into a sequence of zero or more nonempty tokens. On the first call to **strtok**() the string to be parsed should be specified in _str_. In each subsequent call that should parse the same string, _str_ must be NULL.

The _delim_ argument specifies a set of bytes that delimit the tokens in the parsed string. The caller may specify different strings in _delim_ in successive calls that parse the same string.

Each call to **strtok**() returns a pointer to a null-terminated string containing the next token. This string does not include the delimiting byte. If no more tokens are found, **strtok**() returns NULL.

A sequence of calls to **strtok**() that operate on the same string maintains a pointer that determines the point from which to start searching for the next token. The first call to **strtok**() sets this pointer to point to the first byte of the string. The start of the next token is determined by scanning forward for the next nondelimiter byte in _str_. If such a byte is found, it is taken as the start of the next token. If no such byte is found, then there are no more tokens, and **strtok**() returns NULL. (A string that is empty or that contains only delimiters will thus cause **strtok**() to return NULL on the first call.)

The end of each token is found by scanning forward until either the next delimiter byte is found or until the terminating null byte ('\0') is encountered. If a delimiter byte is found, it is overwritten with a null byte to terminate the current token, and **strtok**() saves a pointer to the following byte; that pointer will be used as the starting point when searching for the next token. In this case, **strtok**() returns a pointer to the start of the found token.

From the above description, it follows that a sequence of two or more contiguous delimiter bytes in the parsed string is considered to be a single delimiter, and that delimiter bytes at the start or end of the string are ignored. Put another way: the tokens returned by **strtok**() are always nonempty strings. Thus, for example, given the string "_aaa;;bbb,_", successive calls to **strtok**() that specify the delimiter string "_;,_" would return the strings "_aaa_" and "_bbb_", and then a null pointer.

The **strtok_r**() function is a reentrant version **strtok**(). The _saveptr_ argument is a pointer to a _char *_ variable that is used internally by **strtok_r**() in order to maintain context between successive calls that parse the same string. On the first call to **strtok_r**(), _str_ should point to the string to be parsed, and the value of _saveptr_ is ignored. In subsequent calls, _str_ should be NULL, and _saveptr_ should be unchanged since the previous call.

Different strings may be parsed concurrently using sequences of calls to **strtok_r**() that specify different _saveptr_ arguments.

## RETURN VALUE

**strtok**() and **strtok_r**() return a pointer to the next token, or NULL if there are no more tokens.

## ATTRIBUTES

### Multithreading (see pthreads(7))

The **strtok**() function is not thread-safe, the **strtok_r**() function is thread-safe.