function

<cstdio>

fopen

```
FILE * fopen ( const char * filename, const char * mode );
```

Open file

Opens the file whose name is specified in the parameter *filename* and associates it with a stream that can be identified in future operations by the FILE pointer returned.

The operations that are allowed on the stream and how these are performed are defined by the *mode* parameter.

The returned stream is *fully buffered* by default if it is known to not refer to an interactive device (see setbuf).

The returned pointer can be disassociated from the file by calling fclose or freopen. All opened files are automatically closed on normal program termination.

The running environment supports at least FOPEN_MAX files open simultaneously.

Parameters

`filename`

C string containing the name of the file to be opened.
Its value shall follow the file name specifications of the running environment and can include a path (if supported by the system).

`mode`

C string containing a file access mode. It can be:

"r"	**read:** Open file for input operations. The file must exist.
"w"	**write:** Create an empty file for output operations. If a file with the same name already exists, its contents are discarded and the file is treated as a new empty file.
"a"	**append:** Open file for output at the end of a file. Output operations always write data at the end of the file, expanding it. Repositioning operations (fseek, fsetpos, rewind) are ignored. The file is created if it does not exist.
"r+"	**read/update:** Open a file for update (both for input and output). The file must exist.
"w+"	**write/update:** Create an empty file and open it for update (both for input and output). If a file with the same name already exists its contents are discarded and the file is treated as a new empty file.
"a+"	**append/update:** Open a file for update (both for input and output) with all output operations writing data at the end of the file. Repositioning operations (fseek, fsetpos, rewind) affects the next input operations, but output operations move the position back to the end of file. The file is created if it does not exist.

With the *mode* specifiers above the file is open as a *text file*. In order to open a file as a *binary file*, a `"b"`character has to be included in the *mode* string. This additional `"b"` character can either be appended at the end of the string (thus making the following compound modes: "rb", "wb", "ab", "r+b", "w+b", "a+b") or be inserted between the letter and the `"+"` sign for the mixed modes ("rb+", "wb+", "ab+").

The new C standard (C2011, which is not part of C++) adds a new standard subspecifier (`"x"`), that can be appended to any `"w"` specifier (to form `"wx"`, `"wbx"`, `"w+x"` or `"w+bx"`/`"wb+x"`). This subspecifier forces the function to fail if the file exists, instead of overwriting it.

If additional characters follow the sequence, the behavior depends on the library implementation: some implementations may ignore additional characters so that for example an additional `"t"` (sometimes used to explicitly state a *text file*) is accepted.

On some library implementations, opening or creating a text file with update mode may treat the stream instead as a binary file.

Text files are files containing sequences of lines of text. Depending on the environment where the application runs, some special character conversion may occur in input/output operations in *text mode* to adapt them to a system-specific text file format. Although on some environments no conversions occur and both *text files* and *binary files* are treated the same way, using the appropriate mode improves portability.

For files open for update (those which include a `"+"` sign), on which both input and output operations are allowed, the stream should be flushed (fflush) or repositioned (fseek, fsetpos, rewind) between either a writing operation followed by a reading operation or a reading operation which did not reach the end-of-file followed by a writing operation.

Return Value

If the file is successfully opened, the function returns a pointer to a FILE object that can be used to identify the stream on future operations.

Otherwise, a null pointer is returned.

On most library implementations, the errno variable is also set to a system-specific error code on failure.

function

<cstdio>

fclose

```
int fclose ( FILE * stream );
```

Close file

Closes the file associated with the *stream* and disassociates it.

All internal buffers associated with the stream are disassociated from it and flushed: the content of any unwritten output buffer is written and the content of any unread input buffer is discarded.

Even if the call fails, the stream passed as parameter will no longer be associated with the file nor its buffers.

Parameters

`stream`

Pointer to a FILE object that specifies the stream to be closed.

Return Value

If the stream is successfully closed, a zero value is returned.
On failure, EOF is returned.

function

<cstdio>

feof

```
int feof ( FILE * stream );
```

Check end-of-file indicator

Checks whether the *end-of-File indicator* associated with *stream* is set, returning a value different from zero if it is.

This indicator is generally set by a previous operation on the *stream* that attempted to read at or past the end-of-file.

Notice that *stream*'s internal position indicator may point to the *end-of-file* for the next operation, but still, the *end-of-file* indicator may not be set until an operation attempts to read at that point.

This indicator is cleared by a call to clearerr, rewind, fseek, fsetpos or freopen. Although if the *position indicator* is not repositioned by such a call, the next i/o operation is likely to set the indicator again.

Parameters

`stream`

Pointer to a FILE object that identifies the stream.

Return Value

A non-zero value is returned in the case that the *end-of-file indicator* associated with the stream is set. Otherwise, zero is returned.

function

<cstdio>

fgetc

```
int fgetc ( FILE * stream );
```

Get character from stream

Returns the character currently pointed by the internal file position indicator of the specified *stream*. The internal file position indicator is then advanced to the next character.

If the stream is at the end-of-file when called, the function returns EOF and sets the *end-of-file indicator* for the stream (feof).

If a read error occurs, the function returns EOF and sets the *error indicator* for the stream (ferror).

`fgetc` and getc are equivalent, except that getc may be implemented as a macro in some libraries.

Parameters

`stream`

Pointer to a FILE object that identifies an input stream.

Return Value

On success, the character read is returned (promoted to an `int` value).
The return type is `int` to accommodate for the special value EOF, which indicates failure:
If the position indicator was at the *end-of-file*, the function returns EOF and sets the *eof indicator* (feof) of *stream*.
If some other reading error happens, the function also returns EOF, but sets its *error indicator* (ferror) instead.

function

<cstdio>

fputc

```
int fputc ( int character, FILE * stream );
```

Write character to stream

Writes a *character* to the *stream* and advances the position indicator.

The character is written at the position indicated by the *internal position indicator* of the *stream*, which is then automatically advanced by one.

Parameters

`character`

The `int` promotion of the character to be written.
The value is internally converted to an `unsigned char` when written.

`stream`

Pointer to a FILE object that identifies an output stream.

Return Value

On success, the *character* written is returned.
If a writing error occurs, EOF is returned and the *error indicator* (ferror) is set.

function

<cstdio>

fgets

```
char * fgets ( char * str, int num, FILE * stream );
```

Get string from stream

Reads characters from *stream* and stores them as a C string into *str* until (*num*-1) characters have been read or either a newline or the *end-of-file* is reached, whichever happens first.

A newline character makes `fgets` stop reading, but it is considered a valid character by the function and included in the string copied to *str*.

A terminating null character is automatically appended after the characters copied to *str*.

Notice that `fgets` is quite different from gets: not only `fgets` accepts a *stream* argument, but also allows to specify the maximum size of *str* and includes in the string any ending newline character.

Parameters

`str`

Pointer to an array of `char`s where the string read is copied.

`num`

Maximum number of characters to be copied into *str* (including the terminating null-character).

`stream`

Pointer to a FILE object that identifies an input stream.
stdin can be used as argument to read from the *standard input*.

Return Value

On success, the function returns *str*.
If the *end-of-file* is encountered while attempting to read a character, the *eof indicator* is set (feof). If this happens before any characters could be read, the pointer returned is a null pointer (and the contents of *str* remain unchanged).
If a read error occurs, the *error indicator* (ferror) is set and a null pointer is also returned (but the contents pointed by *str* may have changed).

function

<cstdio>

fputs

```
int fputs ( const char * str, FILE * stream );
```

Write string to stream

Writes the *C string* pointed by *str* to the *stream*.

The function begins copying from the address specified (*str*) until it reaches the terminating null character (`'\0'`). This terminating null-character is not copied to the stream.

Notice that `fputs` not only differs from puts in that the destination *stream* can be specified, but also `fputs` does not write additional characters, while puts appends a newline character at the end automatically.

Parameters

`str`

C string with the content to be written to *stream*.

`stream`

Pointer to a FILE object that identifies an output stream.

Return Value

On success, a non-negative value is returned.
On error, the function returns EOF and sets the *error indicator* (ferror).

function

<cstdio>

fprintf

```
int fprintf ( FILE * stream, const char * format, ... );
```

Write formatted data to stream

Writes the C string pointed by *format* to the *stream*. If *format* includes *format specifiers* (subsequences beginning with `%`), the additional arguments following *format* are formatted and inserted in the resulting string replacing their respective specifiers.

After the *format* parameter, the function expects at least as many additional arguments as specified by *format*.

Parameters

`stream`

Pointer to a FILE object that identifies an output stream.

`format`

C string that contains the text to be written to the stream.
It can optionally contain embedded *format specifiers* that are replaced by the values specified in subsequent additional arguments and formatted as requested.

A *format specifier* follows this prototype:

```
%[flags][width][.precision][length]specifier
```

Where the *specifier character* at the end is the most significant component, since it defines the type and the interpretation of its corresponding argument:

specifier	**Output**	**Example**
`d` *or* `i`	Signed decimal integer	`392`
`u`	Unsigned decimal integer	`7235`
`o`	Unsigned octal	`610`
`x`	Unsigned hexadecimal integer	`7fa`
`X`	Unsigned hexadecimal integer (uppercase)	`7FA`
`f`	Decimal floating point, lowercase	`392.65`
`F`	Decimal floating point, uppercase	`392.65`
`e`	Scientific notation (mantissa/exponent), lowercase	`3.9265e+2`
`E`	Scientific notation (mantissa/exponent), uppercase	`3.9265E+2`
`g`	Use the shortest representation: `%e` or `%f`	`392.65`
`G`	Use the shortest representation: `%E` or `%F`	`392.65`
`a`	Hexadecimal floating point, lowercase	`-0xc.90fep-2`
`A`	Hexadecimal floating point, uppercase	`-0XC.90FEP-2`
`c`	Character	`a`
`s`	String of characters	`sample`

p	Pointer address	b8000000
n	Nothing printed. The corresponding argument must be a pointer to a signed int. The number of characters written so far is stored in the pointed location.	
%	A % followed by another % character will write a single % to the stream.	%

The *format specifier* can also contain sub-specifiers: *flags*, *width*, *.precision* and *modifiers* (in that order), which are optional and follow these specifications:

flags	**description**
-	Left-justify within the given field width; Right justification is the default (see *width* sub-specifier).
+	Forces to preceed the result with a plus or minus sign (+ or −) even for positive numbers. By default, only negative numbers are preceded with a − sign.
(space)	If no sign is going to be written, a blank space is inserted before the value.
#	Used with o, x or X specifiers the value is preceeded with 0, 0x or 0X respectively for values different than zero. Used with a, A, e, E, f, F, g or G it forces the written output to contain a decimal point even if no more digits follow. By default, if no digits follow, no decimal point is written.
0	Left-pads the number with zeroes (0) instead of spaces when padding is specified (see *width* sub-specifier).

width	**description**
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	The *width* is not specified in the *format* string, but as an additional integer value argument preceding the argument that has to be formatted.

.precision	**description**
.number	For integer specifiers (d, i, o, u, x, X): *precision* specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A *precision* of 0 means that no character is written for the value 0. For a, A, e, E, f and F specifiers: this is the number of digits to be printed **after** the decimal point (by default, this is 6). For g and G specifiers: This is the maximum number of significant digits to be printed. For s: this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered. If the period is specified without an explicit value for *precision*, 0 is assumed.
.*	The *precision* is not specified in the *format* string, but as an additional integer value argument preceding the argument that has to be formatted.

The *length* sub-specifier modifies the length of the data type. This is a chart showing the types used to interpret the corresponding arguments with and without *length* specifier (if

a different type is used, the proper type promotion or conversion is performed, if allowed):

	specifiers						
length	**d i**	**u o x X**	**f F e E g G a A**	**c**	**s**	**p**	**n**
(none)	int	unsigned int	double	int	char*	void*	int*
hh	signed char	unsigned char					signed char*
h	short int	unsigned short int					short int*
l	long int	unsigned long int		wint_t	wchar_t*		long int*
ll	long long int	unsigned long long int					long long int*
j	intmax_t	uintmax_t					intmax_t*
z	size_t	size_t					size_t*
t	ptrdiff_t	ptrdiff_t					ptrdiff_t*
L			long double				

Note that the `c` specifier takes an `int` (or wint_t) as argument, but performs the proper conversion to a `char`value (or a `wchar_t`) before formatting it for output.

Note: Yellow rows indicate specifiers and sub-specifiers introduced by C99.
See `<cinttypes>` for the specifiers for extended types.

... (additional arguments)

Depending on the *format* string, the function may expect a sequence of additional arguments, each containing a value to be used to replace a *format specifier* in the *format* string (or a pointer to a storage location, for `n`).
There should be at least as many of these arguments as the number of values specified in the *format specifiers*. Additional arguments are ignored by the function.

Return Value

On success, the total number of characters written is returned.

If a writing error occurs, the *error indicator* (ferror) is set and a negative number is returned.

If a multibyte character encoding error occurs while writing wide characters, errno is set to `EILSEQ` and a negative number is returned.

function

<cstdio>

fscanf

```
int fscanf ( FILE * stream, const char * format, ... );
```

Read formatted data from stream

Reads data from the *stream* and stores them according to the parameter *format* into the locations pointed by the additional arguments.

The additional arguments should point to already allocated objects of the type specified by their corresponding format specifier within the *format* string.

Parameters

`stream`

Pointer to a FILE object that identifies the input stream to read data from.

`format`

C string that contains a sequence of characters that control how characters extracted from the stream are treated:

- **Whitespace character:** the function will read and ignore any whitespace characters encountered before the next non-whitespace character (whitespace characters include spaces, newline and tab characters -- see isspace). A single whitespace in the *format* string validates any quantity of whitespace characters extracted from the *stream* (including none).
- **Non-whitespace character, except format specifier (%):** Any character that is not either a whitespace character (blank, newline or tab) or part of a *format specifier* (which begin with a `%`character) causes the function to read the next character from the stream, compare it to this non-whitespace character and if it matches, it is discarded and the function continues with the next character of *format*. If the character does not match, the function fails, returning and leaving subsequent characters of the stream unread.
- **Format specifiers:** A sequence formed by an initial percentage sign (`%`) indicates a format specifier, which is used to specify the type and format of the data to be retrieved from the *stream* and stored into the locations pointed by the additional arguments.

A *format specifier* for `fscanf` follows this prototype:

`%[*][width][length]specifier`

Where the *specifier* character at the end is the most significant component, since it defines which characters are extracted, their interpretation and the type of its corresponding argument:

specifier	**Description**	**Characters extracted**
`i`, `u`	Integer	Any number of digits, optionally preceded by a sign (`+` or `-`). Decimal digits assumed by default (`0-9`), but a `0` prefix introduces octal digits (`0-7`), and `0x` hexadecimal digits (`0-f`).
`d`	Decimal integer	Any number of decimal digits (`0-9`), optionally preceded by a sign (`+` or `-`).
`o`	Octal integer	Any number of octal digits (`0-7`), optionally preceded by a sign (`+` or `-`).
`x`	Hexadecimal integer	Any number of hexadecimal digits (`0-9`, `a-f`, `A-F`), optionally preceded by `0x` or `0X`, and all optionally preceded by a sign (`+` or `-`).
`f`, `e`, `g`	Floating point number	A series of decimal digits, optionally containing a decimal point, optionally preceeded by a sign (`+` or `-`) and optionally followed by the `e` or `E` character and a decimal integer (or some of the other sequences supported by `strtod`). Implementations complying with C99 also support hexadecimal floating-point format when preceded by `0x` or `0X`.
`a`		
`c`	Character	The next character. If a *width* other than `1` is specified, the function reads exactly*width* characters and stores them in the successive locations of the array passed as argument. No null character is appended at the end.
`s`	String of characters	Any number of non-whitespace characters, stopping at the first whitespacecharacter found. A terminating null character is automatically added at the end of the stored sequence.
`p`	Pointer address	A sequence of characters representing a pointer. The particular format used depends on the system and library implementation, but it is the same as the one used to format `%p` in fprintf.
`[`*characters*`]`	Scanset	Any number of the characters specified between the brackets. A dash (`-`) that is not the first character may produce non-portable behavior in some library implementations.
`[^`*characters*`]`	Negated scanset	Any number of characters none of them specified as *characters* between the brackets.
`n`	Count	No input is consumed. The number of characters read so far from *stream* is stored in the pointed location.

%	%	A % followed by another % matches a single %.

Except for n, at least one character shall be consumed by any specifier. Otherwise the match fails, and the scan ends there.

The *format specifier* can also contain sub-specifiers: *asterisk* (*), *width* and *length* (in that order), which are optional and follow these specifications:

sub-specifier	**description**
*	An optional starting asterisk indicates that the data is to be read from the stream but ignored (i.e. it is not stored in the location pointed by an argument).
width	Specifies the maximum number of characters to be read in the current reading operation (optional).
length	One of hh, h, l, ll, j, z, t, L (optional). This alters the expected type of the storage pointed by the corresponding argument (see below).

This is a chart showing the types expected for the corresponding arguments where input is stored (both with and without a *length* sub-specifier):

	specifiers					
length	**d i**	**u o x**	**f e g a**	**c s [] [^]**	**p**	**n**
(none)	int*	unsigned int*	float*	char*	void**	int*
hh	signed char*	unsigned char*				signed char*
h	short int*	unsigned short int*				short int*
l	long int*	unsigned long int*	double*	wchar_t*		long int*
ll	long long int*	unsigned long long int*				long long int*
j	intmax_t*	uintmax_t*				intmax_t*
z	size_t*	size_t*				size_t*

t	ptrdiff_t*	ptrdiff_t*				ptrdiff_t*
L			long double*			

Note: Yellow rows indicate specifiers and sub-specifiers introduced by C99.

... (additional arguments)

Depending on the *format* string, the function may expect a sequence of additional arguments, each containing a pointer to allocated storage where the interpretation of the extracted characters is stored with the appropriate type.
There should be at least as many of these arguments as the number of values stored by the *format specifiers*. Additional arguments are ignored by the function.
These arguments are expected to be pointers: to store the result of a `fscanf` operation on a regular variable, its name should be preceded by the *reference operator* (`&`) (see example).

Return Value

On success, the function returns the number of items of the argument list successfully filled. This count can match the expected number of items or be less (even zero) due to a matching failure, a reading error, or the reach of the*end-of-file*.

If a reading error happens or the *end-of-file* is reached while reading, the proper indicator is set (feof or ferror). And, if either happens before any data could be successfully read, EOF is returned.

If an encoding error happens interpreting wide characters, the function sets errno to `EILSEQ`.

function

<cstdio>

fseek

```
int fseek ( FILE * stream, long int offset, int origin );
```

Reposition stream position indicator

Sets the position indicator associated with the *stream* to a new position.

For streams open in binary mode, the new position is defined by adding *offset* to a reference position specified by*origin*.

For streams open in text mode, *offset* shall either be zero or a value returned by a previous call to ftell, and *origin*shall necessarily be `SEEK_SET`.

If the function is called with other values for these arguments, support depends on the particular system and library implementation (non-portable).

The *end-of-file internal indicator* of the *stream* is cleared after a successful call to this function, and all effects from previous calls to ungetc on this *stream* are dropped.

On streams open for update (read+write), a call to `fseek` allows to switch between reading and writing.

Parameters

`stream`

Pointer to a FILE object that identifies the stream.

`offset`

Binary files: Number of bytes to offset from *origin*.
Text files: Either zero, or a value returned by ftell.

`origin`

Position used as reference for the *offset*. It is specified by one of the following constants defined in <cstdio>exclusively to be used as arguments for this function:

Constant	Reference position
SEEK_SET	Beginning of file
SEEK_CUR	Current position of the file pointer
SEEK_END	End of file *

* Library implementations are allowed to not meaningfully support `SEEK_END` (therefore, code using it has no real standard portability).

Return Value

If successful, the function returns zero.
Otherwise, it returns non-zero value.
If a read or write error occurs, the *error indicator* (ferror) is set.

function

<cstdio>

ftell

```
long int ftell ( FILE * stream );
```

Get current position in stream

Returns the current value of the position indicator of the *stream*.

For binary streams, this is the number of bytes from the beginning of the file.

For text streams, the numerical value may not be meaningful but can still be used to restore the position to the same position later using fseek (if there are characters put back using ungetc still pending of being read, the behavior is undefined).

Parameters

`stream`

Pointer to a FILE object that identifies the stream.

Return Value

On success, the current value of the position indicator is returned.
On failure, `-1L` is returned, and errno is set to a system-specific positive value.