High C™

Library Reference Manual

Version 1.2

by MetaWare™ Incorporated
High C ™
Library Reference Manual
Version 1.2

© 1985, MetaWare™ Incorporated, Santa Cruz, CA
All rights reserved

NOTICES

The software described in this manual is licensed, not sold. Use of the software constitutes agreement by the user with the terms and conditions of the End-User License Agreement packaged with the software. Read the Agreement carefully. Use in violation of the Agreement or without paying the license fee is unlawful.

Every effort has been made to make this manual as accurate as possible. However, MetaWare Incorporated shall have no liability or responsibility to any person or entity with respect to any liability, loss, or damage caused or alleged to be caused directly or indirectly by this manual, including but not limited to any interruption of service, loss of business or anticipated profits, and all direct, indirect, and consequential damages resulting from the use of this manual and the software that it describes.

MetaWare Incorporated reserves the right to change the specifications and characteristics of the software described in this manual, from time to time, without notice to users. Users of this manual should read the file named "README" contained on the distribution media for current information as to changes in files and characteristics, and bugs discovered in the software. Like all computer software this program is susceptible to unknown and undiscovered bugs. These will be corrected as soon as reasonably possible but cannot be anticipated or eliminated entirely. Use of the software is subject to the warranty provisions contained in the License Agreement.

A. M. D. G.

Trademark Acknowledgments

The term(s) is a trademark of
High C, MetaWare MetaWare Incorporated
MS-DOS Microsoft Corporation (registered tm.)
Professional Pascal MetaWare Incorporated
UNIX AT&T Bell Laboratories

v.10.15.85 © 1983–85 MetaWare Incorporated
Feedback, Please

(A Upon first reading.)

We would greatly appreciate your ideas regarding improvement of the language, its compiler, and its documentation. Please take time to mark up the manual on your first reading and make corresponding notes on this page (front and back) and on additional sheets as necessary. Then mail the results to:

MetaWare™ Incorporated
412 Liberty Street
Santa Cruz, CA 95060

MetaWare may use or distribute any information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use that information. If you wish a reply, please provide your name and address. Thank you in advance, The Authors.

Page Comment

v.09.15.85 © 1983-85 MetaWare Incorporated
Feedback, Please
## Contents


Cover, Title, Contents, Feedback ................... 10 pp.

Sections 1–13..................................... total 181 pp.

1 Introduction ......................................... 12 pp.
   Organization ......................................... 1-1
   Contents ............................................ 1-3
   Including Header Files ............................. 1-4
   Parameter Passing .................................. 1-5
   Macros ............................................. 1-6
   Naming Conventions ............................... 1-8
   Terms ............................................. 1-8
   Regular Expressions ................................ 1-10

2 assert.h ............................................. 2 pp.
   assert ............................................. 2-2

3 ctype.h ............................................. 5 pp.
   isalnum ............................................ 3-1
   isalpha .......................................... 3-1
   iscntrl .......................................... 3-2
   isdigit .......................................... 3-2
   isgraph .......................................... 3-2
   islower .......................................... 3-3
   isprint .......................................... 3-3
   ispunct .......................................... 3-3
   isspace .......................................... 3-4
   isupper .......................................... 3-4
   isxdigit .......................................... 3-4
   tolower .......................................... 3-5
   toupper .......................................... 3-5

4 limits.h ............................................. 4 pp.
   Characteristics of Integral Types .............. 4-2
   Characteristics of Floating-Point Types ........ 4-3
## Contents

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>math.h</td>
<td></td>
<td>14 pp.</td>
</tr>
<tr>
<td>EDOM</td>
<td></td>
<td>5-2</td>
</tr>
<tr>
<td>ERANGE</td>
<td></td>
<td>5-2</td>
</tr>
<tr>
<td>HUGE_VAL</td>
<td></td>
<td>5-3</td>
</tr>
<tr>
<td>abs</td>
<td></td>
<td>5-3</td>
</tr>
<tr>
<td>acos</td>
<td></td>
<td>5-4</td>
</tr>
<tr>
<td>asin</td>
<td></td>
<td>5-4</td>
</tr>
<tr>
<td>atan</td>
<td></td>
<td>5-5</td>
</tr>
<tr>
<td>atan2</td>
<td></td>
<td>5-6</td>
</tr>
<tr>
<td>ceil</td>
<td></td>
<td>5-7</td>
</tr>
<tr>
<td>cos</td>
<td></td>
<td>5-7</td>
</tr>
<tr>
<td>cosh</td>
<td></td>
<td>5-7</td>
</tr>
<tr>
<td>exp</td>
<td></td>
<td>5-8</td>
</tr>
<tr>
<td>fabs</td>
<td></td>
<td>5-8</td>
</tr>
<tr>
<td>floor</td>
<td></td>
<td>5-8</td>
</tr>
<tr>
<td>fmod</td>
<td></td>
<td>5-9</td>
</tr>
<tr>
<td>frexp</td>
<td></td>
<td>5-10</td>
</tr>
<tr>
<td>ldexp</td>
<td></td>
<td>5-11</td>
</tr>
<tr>
<td>log</td>
<td></td>
<td>5-11</td>
</tr>
<tr>
<td>log10</td>
<td></td>
<td>5-12</td>
</tr>
<tr>
<td>modf</td>
<td></td>
<td>5-12</td>
</tr>
<tr>
<td>pow</td>
<td></td>
<td>5-13</td>
</tr>
<tr>
<td>sin</td>
<td></td>
<td>5-13</td>
</tr>
<tr>
<td>sinh</td>
<td></td>
<td>5-13</td>
</tr>
<tr>
<td>sqrt</td>
<td></td>
<td>5-13</td>
</tr>
<tr>
<td>tan</td>
<td></td>
<td>5-14</td>
</tr>
<tr>
<td>tanh</td>
<td></td>
<td>5-14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>setjmp.h</td>
<td></td>
<td>3 pp.</td>
</tr>
<tr>
<td>jmp_buf</td>
<td></td>
<td>6-1</td>
</tr>
<tr>
<td>longjmp</td>
<td></td>
<td>6-1</td>
</tr>
<tr>
<td>setjmp</td>
<td></td>
<td>6-3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>signal.h</td>
<td></td>
<td>5 pp.</td>
</tr>
<tr>
<td>SIGABRT, SIGFPE</td>
<td></td>
<td>7-1</td>
</tr>
<tr>
<td>SIGILL, SIGINT, SIGSEGV</td>
<td></td>
<td>7-2</td>
</tr>
<tr>
<td>SIGTERM, SIGDFL, SIGERR</td>
<td></td>
<td>7-3</td>
</tr>
<tr>
<td>SIGIGN</td>
<td></td>
<td>7-4</td>
</tr>
<tr>
<td>kill</td>
<td></td>
<td>7-4</td>
</tr>
<tr>
<td>signal</td>
<td></td>
<td>7-5</td>
</tr>
<tr>
<td>Contents</td>
<td>page(s)</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>---------</td>
<td></td>
</tr>
<tr>
<td><strong>8</strong> stdarg.h</td>
<td>5 pp.</td>
<td></td>
</tr>
<tr>
<td>va_list</td>
<td>8-1</td>
<td></td>
</tr>
<tr>
<td>va_start</td>
<td>8-2</td>
<td></td>
</tr>
<tr>
<td>va_arg</td>
<td>8-3</td>
<td></td>
</tr>
<tr>
<td>va_end</td>
<td>8-5</td>
<td></td>
</tr>
<tr>
<td><strong>9</strong> stdefs.h</td>
<td>2 pp.</td>
<td></td>
</tr>
<tr>
<td>errno</td>
<td>9-1</td>
<td></td>
</tr>
<tr>
<td>NULL</td>
<td>9-1</td>
<td></td>
</tr>
<tr>
<td>ptrdiff_t</td>
<td>9-2</td>
<td></td>
</tr>
<tr>
<td>size_t</td>
<td>9-2</td>
<td></td>
</tr>
<tr>
<td><strong>10</strong> stdio.h</td>
<td>69 pp.</td>
<td></td>
</tr>
<tr>
<td>Streams</td>
<td>10-1</td>
<td></td>
</tr>
<tr>
<td>Errors</td>
<td>10-2</td>
<td></td>
</tr>
<tr>
<td>End of File</td>
<td>10-2</td>
<td></td>
</tr>
<tr>
<td>Text and Binary Streams</td>
<td>10-3</td>
<td></td>
</tr>
<tr>
<td>System Dependencies</td>
<td>10-4</td>
<td></td>
</tr>
<tr>
<td>FILE</td>
<td>10-5</td>
<td></td>
</tr>
<tr>
<td>BUFSIZ</td>
<td>10-5</td>
<td></td>
</tr>
<tr>
<td>EOF</td>
<td>10-5</td>
<td></td>
</tr>
<tr>
<td>L_tmpnam</td>
<td>10-5</td>
<td></td>
</tr>
<tr>
<td>NULL</td>
<td>10-6</td>
<td></td>
</tr>
<tr>
<td>SEEK_CUR, SEEK_END, SEEK_SET</td>
<td>10-6</td>
<td></td>
</tr>
<tr>
<td>SYS_OPEN</td>
<td>10-8</td>
<td></td>
</tr>
<tr>
<td>TMP_MAX</td>
<td>10-8</td>
<td></td>
</tr>
<tr>
<td>size_t</td>
<td>10-7</td>
<td></td>
</tr>
<tr>
<td>stdin, stdout, stderr</td>
<td>10-7</td>
<td></td>
</tr>
<tr>
<td>clearerr</td>
<td>10-9</td>
<td></td>
</tr>
<tr>
<td>fclose</td>
<td>10-9</td>
<td></td>
</tr>
<tr>
<td>feof</td>
<td>10-11</td>
<td></td>
</tr>
<tr>
<td>ferror</td>
<td>10-11</td>
<td></td>
</tr>
<tr>
<td>fflush</td>
<td>10-12</td>
<td></td>
</tr>
<tr>
<td>fgetc</td>
<td>10-13</td>
<td></td>
</tr>
<tr>
<td>fgets</td>
<td>10-15</td>
<td></td>
</tr>
<tr>
<td>fopen</td>
<td>10-17</td>
<td></td>
</tr>
<tr>
<td>fprintf</td>
<td>10-19</td>
<td></td>
</tr>
<tr>
<td>fputc</td>
<td>10-28</td>
<td></td>
</tr>
<tr>
<td>fputs</td>
<td>10-29</td>
<td></td>
</tr>
</tbody>
</table>
## Contents

<table>
<thead>
<tr>
<th>Function</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>getenv</td>
<td>11-10</td>
</tr>
<tr>
<td>malloc</td>
<td>11-10</td>
</tr>
<tr>
<td>onexit</td>
<td>11-11</td>
</tr>
<tr>
<td>rand</td>
<td>11-13</td>
</tr>
<tr>
<td>realloc</td>
<td>11-14</td>
</tr>
<tr>
<td>srand</td>
<td>11-14</td>
</tr>
<tr>
<td>strtod</td>
<td>11-15</td>
</tr>
<tr>
<td>strtol</td>
<td>11-17</td>
</tr>
<tr>
<td>system</td>
<td>11-19</td>
</tr>
</tbody>
</table>

### 12 string.h

<table>
<thead>
<tr>
<th>Function</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>size_t</td>
<td>12-1</td>
</tr>
<tr>
<td>_MAXSTRING</td>
<td>12-1</td>
</tr>
<tr>
<td>memchr</td>
<td>12-2</td>
</tr>
<tr>
<td>memcmp</td>
<td>12-4</td>
</tr>
<tr>
<td>memcpy</td>
<td>12-6</td>
</tr>
<tr>
<td>memset</td>
<td>12-8</td>
</tr>
<tr>
<td>strcat</td>
<td>12-10</td>
</tr>
<tr>
<td>strchr</td>
<td>12-12</td>
</tr>
<tr>
<td>strcmp</td>
<td>12-14</td>
</tr>
<tr>
<td>strcpy</td>
<td>12-16</td>
</tr>
<tr>
<td>strlen</td>
<td>12-18</td>
</tr>
<tr>
<td>strcsnspn</td>
<td>12-19</td>
</tr>
<tr>
<td>strncat</td>
<td>12-20</td>
</tr>
<tr>
<td>strncmp</td>
<td>12-21</td>
</tr>
<tr>
<td>strncpy</td>
<td>12-22</td>
</tr>
<tr>
<td>strpbrk</td>
<td>12-23</td>
</tr>
<tr>
<td>strrev</td>
<td>12-24</td>
</tr>
<tr>
<td>strspn</td>
<td>12-25</td>
</tr>
<tr>
<td>strtok</td>
<td>12-26</td>
</tr>
<tr>
<td>_memcpy</td>
<td>12-28</td>
</tr>
<tr>
<td>_rstrcat</td>
<td>12-30</td>
</tr>
<tr>
<td>_rstrncpy</td>
<td>12-32</td>
</tr>
<tr>
<td>_strcats</td>
<td>12-34</td>
</tr>
<tr>
<td>_strncat</td>
<td>12-35</td>
</tr>
</tbody>
</table>
1

Introduction

This manual describes the MetaWare™ High C™ Library, which is largely ANSI-standard, but which has a few additions.

This section briefly describes the organization, format, and contents of the library and the rest of this manual. It then describes: the effect of including the "h" header files in source programs, parameter passing conventions used in the library, macros, naming conventions used in the library, and finally some terms and the regular expression notation common to several subsequent sections of the manual.

Organization. The library is divided into several areas, such as mathematics and string handling, according to the ANSI C Library standard. For each area, there is a header file that contains definitions of macros and declarations of functions and types dealing with that area. For each header file, there is a section in this manual with the same name. The sections and their contents are arranged alphabetically.

Each section contains a general description of a header file. Following that are alphabetized individual descriptions of first the types declared in the header file, then the macros defined in the header file that embody manifest constants, and finally the functions declared in the file.

The individual descriptions appear in the format illustrated below.

All words from the C language, including names from the header files and names representing arguments to functions are set in this fixed-width type. Of those, keywords and preprocessor directives are set in boldface. Most such words (usually excepting macro names) must be in lower case, as the language is case sensitive. Due to the difficulty of picking out C words in the middle of an English paragraph, those that are
Introduction

not boldfaced nor CAPITALIZED are underlined in the context of English text, though not in program text.

Regular expressions (see below) are set in this fixed-width type, but are not underlined since the names used there always beginning with a capital letter.

Finally, occasional special terms not part of the C language nor of the header files, such as Infinity, NAN, and the names of the header files themselves, are set in italics.

The template below shows the format of the descriptions of each function. For any given function, there might appear only a subset of the paragraphs in this template. The description of a type or macro differs in that only the INTERFACE and DESCRIPTION paragraphs appear. The INTERFACE paragraph of a type or macro description contains the preprocessor directive that causes the header file containing the type declaration or macro definition to be included in a source file.

fcn — Short description of the type, macro, or function.
— If it is provided as a macro, that fact is stated here.

INTERFACE

The declaration for fcn appears here, providing information about fcn's return type and the type(s) of its argument(s). For example:

    double fcn(double argument);

If a header file must be included in order for fcn to be used, the line

    #include <appropriate header file name>

appears before the function declaration. Such a line must appear in the source line prior to the use of fcn. See the discussion of including header files, below.
Introduction

DESCRIPTION

`fcn` is described here; what it does, what it returns, what argument(s) are valid, what action(s) are taken when certain errors occur.

CAUTIONS

If `fcn` is potentially dangerous, the pitfalls are pointed out.

SURPRISES

Any unexpected or counter-intuitive aspects are mentioned here.

SYSTEM DEPENDENCIES

Any system-specific aspect(s) of `fcn` are discussed here.

SEE ALSO

Related functions are listed here.

EXAMPLE

An example appears here.

Contents. Here is a summary of the header files treated.

`assert.h` defines a macro that emits diagnostic messages when specified conditions are not met.

`ctype.h` defines macros for character handling.

`limits.h` defines macros specifying constraints on numeric representations.

`math.h` declares mathematical functions.

`setjmp.h` declares functions for setting up non-local jumps.

`stdarg.h` defines macros for accessing a variable number of parameters to a function.

`stddef.h` provides popular definitions for inclusion by other header files.

`stdio.h` declares input-output functions.
stdlib.h declares functions of general utility. Among them are functions for converting strings to other values, for communicating with the host environment, for managing memory, and for generating pseudo-random integers.

string.h declares functions that manipulate strings and character arrays.

time.h declares functions useful for determining and manipulating the date and time.

Including Header Files. The header files contain declarations for the functions provided by the High C™ Library. If a C program uses a library function, including the appropriate header file in the source file declares the function. (See the #include preprocessor directive in the High C Language Reference Manual.)

However, the standard C language permits references to functions that are not declared within the compilation unit containing the reference. This is possible because a function may be compiled separately from a compilation unit in which it is called, and a resolution between the function call and the function can be made at link time.

The return type of such a function is assumed to be int. The library has been carefully designed so that any library function taking arguments of scalar type and returning either nothing or a result of an integral type can be called without being declared.

A danger inherent in this capability is that functions that do not satisfy the above constraints can also be called without being declared, but since the constraints are assumed (by the compiler) to hold, unexpected behavior may result at run time. To correctly use a function that does not satisfy the above constraints, either include the header file that contains its declaration (safe) or duplicate the declaration (unreliable).
Despite C’s function-calling flexibility, it is a good idea to include the header file anyway. Such inclusion provides compile-time protection against passing the wrong number or type of arguments to these functions.

Unlike functions, references to types and macros must be resolved during compilation. If a type or macro provided by this library is used in a program, either the header that defines it must be included or the definition must be duplicated in the source code.

In the **INTERFACE** section of each function description, the line

```
#include <header file name>
```

appears if the function cannot be safely called without either including the header or duplicating some of the text from the header. This may be because the function does not satisfy the constraints mentioned above, or because the function takes or returns an argument of a type declared in a header file.

Header files may be included in any order, and they may be included more than once with no ill effects.

**Parameter Passing.** The ANSI standard C language provides a syntax for function declarations that allows the type and number of arguments to be checked against the type and number of declared parameters. Such a declaration is called a **prototype**.

Arguments to a function that is thus declared must be compatible with the types of the declared parameters, in the same sense that the right part of an assignment expression must be compatible with its left part. The arguments are passed with the same conversions that occur with assignment — i.e. no conversion if the types are the same, and otherwise some arithmetic conversion may occur, such as from ints to floats.

An argument to a function that is called without being declared or that is declared without a prototype has the
following conversion performed upon it: if the argument is a signed char or signed short, it is converted to signed int; if it is an unsigned char or unsigned short, it is coerced to unsigned int; if it is a float, it is coerced to double; otherwise no conversion occurs.

The header files use prototypes to declare library functions. The library is designed as much as possible to allow library functions to be called without being declared, which has an impact on the declarations of some functions.

A number of library functions take characters as arguments, but the corresponding parameters are declared as int rather than char. This is because chars are widened to ints when passed to non-prototype functions. If such a parameter c were declared of type char and the function were called without being declared, c would be passed as an int and the behavior would be undefined. For consistency, all such functions take ints rather than chars, even if they are likely to be used only if the appropriate header is included.

Likewise, functions that return character values are declared to return ints so they can be used without being declared, since undeclared functions are assumed to return int. In addition, a number of such functions must be able to return EOF, a macro declared in stdio.h that expands to an int literal.

Library functions that are implemented as macros are documented using the syntactic form of functions. The documentation uses the parameter and return types that would be appropriate if the macro were a true function, even though macros are not typed.

Macros. A number of macros are provided in the library. These fall into two categories: (1) those that implement a provided function, and (2) those that expand to constant or variable references. For one of these macros to be used, the
header file containing its definition must be included in the source file being compiled.

Macros in the second category require more discussion, since several functions are provided with two implementations: one as a macro and one as a function. This is mentioned in the description of each such function.

In the usual case the effect of a function is identical to the effect of the corresponding macro. A function call evaluates each argument exactly once, however, while a macro may evaluate its arguments more than once. Therefore, the effect of a function call may differ from the effect of the corresponding macro expansion if an argument has side effects. For each macro provided, if any arguments may be multiply evaluated, that fact is documented.

In a given instance, it may be desirable to choose a true function over a macro, or vice versa. To access a provided macro, it is merely necessary to include the header file. If the header file is not included, a reference to the function name references the true function.

It may be desirable to include the header file and still reference the function instead of the macro, since each header file typically provides a number of functions. Consider the line:

```
#undef <function_name>
```

It removes the definition of `<function_name>` as a macro, so that all subsequent instances of `<function_name>` refer to the function. The macro is irretrievable (short of duplicating the definition). Such a line must also appear in the source file any time a name defined as a macro (perhaps in an included file) must be redefined as a macro or declared as a name of an object in the program.

If a macro has parameters, substitution on that macro occurs only if it appears immediately followed by the parenthesized parameter list. Thus, surrounding the name with parentheses disables the substitution and forces a call to the function. For example, `getc(F)` invokes a macro but
Introduction

(qetc)(F) calls a function. This allows those functions with arguments to be called while the corresponding macros retain their definition.

Naming Conventions. This is an ANSI standard C library. The names of the functions, function parameters, types, and macros are those dictated by the standard. Any functions, types, and macros in the library that are not required by the standard have names that begin with underscore ("_”).

Terms. The following terms are used in several places in this manual. The final few are names from the C library that are referred to in several subsequent manual sections.

ANSI refers to the American National Standards Institute, which is responsible for the standard C language and library definition, currently in draft form.

null pointer
is any pointer that compares equal with 0.

NUL is not a macro, but a name for the character '\0', the string termination character, whose value is the integer zero. Note that NUL is different from NULL, a macro that expands to a representation of the null pointer; see stdefs.h.

string refers to a pointer to the first character of a sequence of characters terminated with NUL.

whitespace refers to a set of characters including tabs, spaces, and newlines. It is that set of characters tested for by the function isspace provided by header file ctype.h.

Sign refers to a plus or minus sign: '+ ' or ' - '.

Digit † refers to any decimal digit: '0' through '9'.

Odigit † refers to any octal digit, i.e. any Digit except '8' and '9'.

v.10.15.85 © 1985 MetaWare Incorporated
Hexdigit \( t \) refers to any hexadecimal digit, which is a Digit or any of the characters 'a' through 'f' (or 'A' through 'F' — casing is not significant in numbers) having values 10 through 15 respectively.

\( t \) Following Ada, sequences of digits may contain internal, non-consecutive underscores. Underscore ('_') is allowed as a separator to make large numbers more readable. It is not allowed as the first or last character.

\texttt{size t} is a macro that expands to the integral type of the result of the \texttt{sizeof} operator. It is defined in \texttt{stdefs.h}, \texttt{stdio.h}, \texttt{stdlib.h}, and \texttt{string.h}.

\texttt{errno} is a variable of type \texttt{int} that is set to a positive integer error code by some library functions when an error occurs during their execution. It is set to zero at program start up. It is never set to zero by any library function. A program that uses \texttt{errno} for error checking should set it to zero before a call to a library function and inspect it prior to a subsequent call to a library function. \texttt{errno} is declared in \texttt{stdefs.h}.

\texttt{NULL} is a macro that expands to a value that is assignment-compatible with any pointer type and compares equal with the constant zero. It is therefore suitable as a representation of the null pointer. Note that \texttt{NULL} is not appropriate as the terminating character of a string, as the size of a pointer is not necessarily the same as the size of the character \texttt{NUL}. \texttt{NULL} is defined in \texttt{stdefs.h}, \texttt{stdio.h}, \texttt{stdlib.h}.
Regular Expressions. The library contains functions that handle sequences of characters that must be of particular forms. A simple example is a digit sequence. There are an infinite number of such sequences, but the "regular expression" "Digit*" describes them all quite succinctly. Regular expressions are used in the manual to describe the forms sequences may take.

Regular expressions provide a way to specify the order and number of (in this case) characters in sequences. Within a regular expression, a quoted character ('a') represents the character (a). Names can be given to regular expressions, and names can be used in regular expressions that are either pre-defined (such as Digit and whitespace, defined above) or are themselves names of regular expressions.

The order in which characters appear in a regular expression is the order in which the characters must appear in the described sequence. When a name appears, it is equivalent to the regular expression it represents.

The descriptive power of these expressions comes from modifiers that act on the elements (characters and names) of an expression. The modifiers are *, ?, |, and -, where * means zero or more, ? means zero or one, | means alternation, and - means set subtraction. The modifiers are not quoted. Parentheses are used for grouping.

For example, "***" is a regular expression describing a sequence composed of zero or more asterisks.

A name N followed by a right arrow (->) followed by a regular expression E terminated with a semicolon defines N to represent E, as illustrated here:

Name -> Regular_expression;
Thus we can precisely define Digits, Odigits, and Hexdigits:

$$\text{Digit} \rightarrow '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';$$

$$\text{Digits} \rightarrow \text{Digit} ('-'? \text{Digit})*;$$

$$\text{Odigit} \rightarrow \text{Digit} - '8' - '9';$$

$$\text{Odigits} \rightarrow \text{Odigit} ('-'? \text{Odigit})*;$$

$$\text{Hexletter} \rightarrow 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F';$$

$$\text{Hexdigit} \rightarrow \text{Digit} \mid \text{Hexletter};$$

$$\text{Hexdigits} \rightarrow \text{Hexdigit} ('-'? \text{Hexdigit})*;$$

For example, a digit sequence optionally prefixed by a sign could be named Integer and described by

$$\text{Integer} \rightarrow ('+' | '-'? \text{Digit}+;$$

and the format sequence that is printf's first argument and fprintf's second argument can be described by

$$\text{Format} \rightarrow (\text{any}-'%' \mid \text{Conversion_spec})*;$$

Here, any refers to any character. Conversion specification (Conversion_spec) is described next by a copy of a small piece of Section stdio.h which describes the function fprintf.

$$\text{Conversion_spec} \rightarrow '%' \text{Flag}^* \text{Field_width}? \text{Precision}? \text{Size}? \text{C_char};$$

$$\text{Flag} \rightarrow '-' | '+' | '|' | '#' | 'O';$$

$$\text{Field_width} \rightarrow ((\text{Digit}-'0' '-'? \text{Digits}) \mid '^' \mid '(' \text{Digits} \mid '*')$$

$$\text{Precision} \rightarrow ';'.(\text{Digits} \mid '*')$$

$$\text{Size} \rightarrow 'h' | 'l' | 'L';$$

$$\text{C_char} \rightarrow 'd' | 'i' | 'o' | 'u' | 'x' | 'X' | 'f' | 'e' | 'E' | 'g' | 'G' | 'c' | 's' | 'p' | 'n' | '%';$$

The following are valid calls to printf: the string that is the first argument to printf is a valid format string.

```c
printf("Hello world.");
printf("Hello %s.", "world");
printf("%d * %hd = %ld", 2, 3, 6);
printf("%05d * %.5f = %+#07.*LG", 2, 3.14, 6.28);
```
printf("I am behind you %d%%.", 1000);

The following are not valid calls to printf: the string that is the first argument to printf is not a valid format string.

printf("Hello world.%");    /* Missing C_char. */
printf("Hello %k.", "world"); /* k not a C_char. */
printf("%d * %h = %hl", 2, 3, 6); /* h not a */
    /* C_char. Also, only one of h or */
    /* I-can be generated from Size. */
printf("%105d * %f.5 = %7+##.LG", 2, 3.14, 6.28);
    /* Size (1) must follow Field_width (05). */
    /* C_char (f) must follow Precision (.5). */
    /* Field width (7) must follow Flags(+##0). */
printf("I am behind you %d%%%.", 1000);
    /* Missing C_char. */
The header file `assert.h` defines one macro, `assert`, described on the next page, which puts diagnostics into a program. If a user-defined macro `NDEBUG` is defined at the point of inclusion of the header file `assert.h`, `assert` has no effect.
assert.h

assert — Abort if an assertion is false.
— Provided as a macro only.

INTERFACE

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

DESCRIPTION

Does nothing if expression evaluates to non-zero (true). If expression evaluates to zero and NDEBUG is not #defined, the name of the file and number of the source line where the assertion appears is written on standard error, and the abort function is called; see stdlib.h.

EXAMPLE

#include <stdio.h>
#include <assert.h>
main() {
#define C_ARRAY_END (25)
    int i = 0, no_z = 1;
    char c_array[C_ARRAY_END];
    ... 
    while (no_z) {
        assert(i < C_ARRAY_END); /* Line 64. */
        c_array[i] = getchar();
        no_z = (c_array[i++] != 'z');
    }
    ... 
}

Let us suppose the above program is in file /work/junk.c, and the assert function call appears on line 64. If during execution of the above while loop i becomes greater than 24, the program aborts after printing the following on the standard error file.

Assertion failed in file /work/junk.c at line 64.
The header file `ctype.h` provides a number of macros that can be used for character handling.

### isalnum
- **Test for alphanumeric character.**
- Provided as a macro only.

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

**DESCRIPTION**
Returns nonzero if `c` is a letter or digit; zero otherwise.

### isalpha
- **Test for alphabetic character.**
- Provided as a macro only.

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

**DESCRIPTION**
Returns nonzero if `c` is a letter; zero otherwise.
**iscntrl** — Test for control character.
— Provided as a macro only.

**INTERFACE**

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

**DESCRIPTION**

Returns nonzero if \( c \) is a control character; zero otherwise. A control character is any character in the ASCII character set whose hexadecimal value is between 0 and 1F inclusive, or 7F. If the value is greater than 7F, i.e. if the high bit is on, zero is returned.

**isdigit** — Test for numeric character.
— Provided as a macro only.

**INTERFACE**

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

**DESCRIPTION**

Returns nonzero if \( c \) is a Digit; zero otherwise.

**isgraph** — Test for visible character.
— Provided as a macro only.

**INTERFACE**

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

**DESCRIPTION**

Returns nonzero if \( c \) is a printing character other than space; zero otherwise.
islower — Test for lowercase alphabetic character.
— Provided as a macro only.

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

DESCRIPTION
Returns nonzero if \texttt{c} is a lowercase letter; zero otherwise.

isprint — Test for printing character.
— Provided as a macro only.

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

DESCRIPTION
Returns nonzero if \texttt{c} is a printing character including space; zero otherwise.

ispunct — Test for punctuation character.
— Provided as a macro only.

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

DESCRIPTION
Returns nonzero if \texttt{c} is a printing character that is not a Digit, letter, or space; zero otherwise.
isspace — Test for whitespace character.
— Provided as a macro only.

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

DESCRIPTION
Returns nonzero if \c is a space (' '), form feed ('\f'),
horizontal tab ('\h'), newline ('\n'), carriage return ('\r'),
or vertical tab ('\v'); zero otherwise.

isupper — Test for uppercase alphabetic character.
— Provided as a macro only.

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

DESCRIPTION
Returns nonzero if \c is an uppercase letter; zero otherwise.

isxdigit — Test for hexadecimal numeric character.
— Provided as a macro only.

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

DESCRIPTION
Returns nonzero if \c is a hexadecimal digit; zero otherwise.
tolower — Convert uppercase to lowercase.
— Provided as a macro only.

INTERFACE

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

DESCRIPTION

Converts an uppercase letter to a lowercase letter. If \( c \) is an uppercase letter, \texttt{tolower} returns the corresponding lowercase letter; otherwise \( c \).

\( c \) is evaluated more than once.

\texttt{tolower(c)} is a version of \texttt{tolower} that can be used when \( c \) is known to be an uppercase character. When \texttt{tolower} is used, \( c \) is evaluated exactly once.

toupper — Convert lowercase to uppercase.
— Provided as a macro only.

INTERFACE

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

DESCRIPTION

Converts a lowercase letter to an uppercase letter. If \( c \) is a lowercase letter, \texttt{toupper} returns the corresponding uppercase letter; otherwise \( c \).

\( c \) is evaluated more than once.

\texttt{toupper(x)} is a version of \texttt{toupper} that can be used when \( x \) is known to be a lowercase character. When \texttt{toupper} is used, \( c \) is evaluated exactly once.
The header file *limits.h* defines a number of macros that specify constraints on numerical representations. The macros, their values, and a description of their meanings follow.
limits.h

/* CHARACTERISTICS OF INTEGRAL TYPES. */

/* Maximum number of bits for smallest object (byte): */
#define CHAR_BIT 8

/* Maximum value for an object of type char: */
#define CHAR_MAX 255

/* Minimum value for an object of type char: */
#define CHAR_MIN 0

/* Maximum value for an object of type signed char: */
#define SCHAR_MAX +127

/* Minimum value for an object of type signed char: */
#define SCHAR_MIN -128

/* Maximum value for an object of type unsigned char: */
#define UCHAR_MAX 255

/* Maximum value for an object of type short: */
#define SHRT_MAX +32_767

/* Minimum value for an object of type short: */
#define SHRT_MIN -32_768

/* Maximum value for an object of type signed short: */
#define USHRT_MAX 65_535

/* Minimum value for an object of type int: */
#define INT_MAX +32_767

/* Minimum value for an object of type int: */
#define INT_MIN -32_768

/* Maximum value for an object of type unsigned int: */
#define UINT_MAX 65_535

/* Maximum value for an object of type signed long: */
#define LONG_MAX +2_147_483_647

/* Minimum value for an object of type signed long: */
#define LONG_MIN -2_147_483_648

/* Maximum value for an object of type unsigned long: */
#define ULONG_MAX 4_294_967_295

v.10.15.85 © 1985 MetaWare Incorporated
/* CHARACTERISTICS OF FLOATING-POINT TYPES. */

/* Note that the internal representation of a floating-point number in binary means that numbers such as LDBL_DIG, FLT_DIG, DBL_DIG, etc., are intrinsically inaccurate. For example, for double, between 15.6 and 16.9 significant digits can be represented, depending upon the value actually represented. The value of DBL_DIG should not be construed to mean that every DBL_DIG-digit floating-point number can be stored with complete accuracy in a double. */

/* Radix of double exponent representation: */
#define DBL_RADIX 2
/* double addition rounds (1) or chops (0): */
#define DBL_ROUNDS 1
/* Maximum exponent power of 10 that can be represented in a double: */
#define DBL_MAX_EXP +308
/* Minimum exponent power of 10 that can be represented in a double: */
#define DBL_MIN_EXP -307
/* Maximum number of decimal digits of floating-point precision in a double: */
#define DBL_DIG 16

/* Radix of float exponent representation: */
#define FLT_RADIX 2
/* float addition rounds (1) or chops (0): */
#define FLT_ROUNDS 1
/* Maximum exponent power of 10 that can be represented in a float: */
#define FLT_MAX_EXP +38
/* Minimum exponent power of 10 that can be represented in a float: */
#define FLT_MIN_EXP -38
limits.h

/* Maximum number of decimal digits of precision in a
   float: */
#define FLT_DIG 7

/* Radix of long double exponent representation: */
#define LDBL_RADIX 2

/* long double addition rounds (1) or chops (0): */
#define LDBL_ROUNDS 1

/* Maximum exponent power of 10 that can be represented
   in a long double: */
#define LDBL_MAX_EXP +4_932

/* Minimum exponent power of 10 that can be represented
   in a long double: */
#define LDBL_MIN_EXP -4_932

/* Maximum number of decimal digits of precision in a
   long double: */
#define LDBL_DIG 19
The header file *math.h* declares the mathematical functions described below.

**errno.** Some of the functions reference the int variable `errno`, which is declared in *stdefs.h*. `errno` is set to a positive integer error code by some library functions when an error occurs during their execution. It is set to zero on program start up. It is never set to zero by any library function. To reference `errno` from a program, either the header file *stdefs.h* must be included, or `errno` must be declared to be external (extern int errno;).

When an argument to a function is outside of the domain over which the function is (mathematically) defined, `errno` is set to EDOM (described below). When the result of a function is too large in magnitude to fit in a double, `errno` is set to ERANGE — described below. Each function description below specifies the return value when a domain or range error occurs. When appropriate, that value is HUGE_VAL — described below.

**SYSTEM DEPENDENCIES**

Under MS-DOS, when an 8087 is present, the value returned when a domain or range error occurs may not be as documented below. The 8087 uses the IEEE standard, generating the values *Infinity* and *NAN* (Not A Number) when error conditions occur. Furthermore, the behavior of a function when given an input of *Infinity* or *NAN* is undefined, but generally results in a *NAN*.

*Infinity* and *NAN* are special numbers that flag an error condition throughout a computation. If at any point during a computation an error condition causes *Infinity* or *NAN* to be generated, whether as intermediate or final results, the final
result is one of these special numbers. If \textit{NAN} is an operand in any arithmetic operation, \textit{NAN} is the result. If \textit{Infinity} is an operand in any arithmetic operation, \textit{Infinity} or \textit{NAN} is the result. The library function \texttt{printf} prints “\textit{Infinity}” and “\textit{NAN}” for those two numbers, but there is no explicit way to test for them (though with the 80(2)87 “x != x” is true if x is \textit{NAN}).

Future versions of the library may offer a choice between ANSI and IEEE standard error handling.

\textbf{EDOM} — A macro used to flag mathematical domain errors.

\texttt{#include <math.h>}

\texttt{DESCRIPTION}

Expands to an integer constant that is the error code for a domain error. Several functions described below set \texttt{errno} to \texttt{EDOM} when an argument is not in the mathematical domain over which the function is defined.

\textbf{ERANGE} — A macro used to flag mathematical range errors.

\texttt{#include <math.h>}

\texttt{DESCRIPTION}

Expands to an integer constant that is the error code for a range error. Several functions described below set \texttt{errno} to \texttt{ERANGE} when a result is not in the range of (too large in magnitude to fit in) the return type of the function.
HUGE_VAL — A macro used upon error as a return value from several mathematical functions.

INTERFACE
   #include <math.h>

DESCRIPTION
   Expands to a double constant that is the largest positive floating-point value that fits in a double. Several functions described below return HUGE_VAL when a valid result cannot be produced.

abs — Absolute value of an integer.
   — Provided as a macro and as a function.

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

DESCRIPTION
   Returns the absolute value of i.

CAUTIONS
   Arguments outside the range of int return undefined results.

SURPRISES
   abs(-32_768) returns -32_768, as there is no corresponding positive integer.

SEE ALSO
   fabs for the absolute value of reals.
acos — Arc cosine.

INTERFACE

    double acos(double x);

DESCRIPTION

    Returns the principal value of the arc cosine of \( x \) in the range \([0.0,\pi]\), where \( x \) is in radians. \( x \) must be in the range \([-1.0,1.0]\). If \( x \) is out of range, \texttt{errno} is set to \texttt{EDOM} and 0.0 is returned.

SEE ALSO

    \texttt{cos} for the cosine of an angle.

asin — Arc sine.

INTERFACE

    double asin(double x);

DESCRIPTION

    Returns the principal value of the arc sine of \( x \) in the range \([0.0,\pi]\), where \( x \) is in radians. \( x \) must be in the range \([-1.0,1.0]\). If \( x \) is out of range, \texttt{errno} is set to \texttt{EDOM} and 0.0 is returned.

SEE ALSO

    \texttt{sin} for the sine of an angle.
atan — Arc tangent.

**INTERFACE**

    double atan(double x);

**DESCRIPTION**

Returns the principal value of the arc tangent of \( x \) in the range \([-\pi/2.0, \pi/2.0]\), where \( x \) is in radians.

**SEE ALSO**

- tan for the tangent of an angle.
- atan2 for the arc tangent of the angle defined by a point.
atan2 — Arc tangent of the angle defined by a point.

**DESCRIPTION**

Returns the principal value of the arc tangent of \( y/x \) in the range \((-\pi,\pi]\), where \( y/x \) is in radians. The signs of \( y \) and \( x \) determine the quadrant of the point \( x,y \). A line is described by that point and the origin. The result of \( \text{atan2}(y,x) \) is the smallest angle from the \( x \) axis to that line, some or all of which angle lies in the first quadrant. The range of the result can be further broken down according to the signs of \( y \) and \( x \), as follows:

- For \( y \geq 0.0 \), \( x \geq 0.0 \), the result is in the range \([0.0,\pi/2.0]\).
- For \( y \geq 0.0 \), \( x < 0.0 \), the result is in the range \([\pi/2.0,\pi]\).
- For \( y < 0.0 \), \( x < 0.0 \), the result is in the range \([-\pi,-\pi/2.0]\).
- For \( y < 0.0 \), \( x \geq 0.0 \), the result is in the range \([-\pi/2.0,0.0]\).

If both arguments are 0.0, \texttt{errno} is set to \texttt{EDOM} and 0.0 is returned.

**SEE ALSO**

- \texttt{atan} for the arc tangent of an angle.
- \texttt{tan} for the tangent of an angle.

**EXAMPLE**

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

main() {
    double y, x;
    y = 1.55741;
    x = 1.0;
    printf("%e %e", atan2(y, x), atan2(-y, x));
    printf("%e %e", atan2(y,-x), atan2(-y,-x));
}
```

prints

\[ 1.000001e+00 \quad -1.000001e+00 \quad 2.141592e+00 \quad -2.141592e+00 \]
ceil — Ceiling function.

**INTERFACE**

double ceil(double x);

**DESCRIPTION**

Returns the smallest integer not less than \( x \).

**SEE ALSO**

floor for the floor function.

**COS** — Cosine.

**INTERFACE**

double cos(double x);

**DESCRIPTION**

Returns the cosine of \( x \) where \( x \) is in radians. \( \text{NAN} \) is returned if \( |x| > \text{HUGE}_\text{VAL} \).

**SEE ALSO**

acos for the arc cosine of an angle.

**cosh** — Hyperbolic cosine.

**INTERFACE**

double cosh(double x);

**DESCRIPTION**

Returns the hyperbolic cosine of \( x \). If the magnitude of \( x \) is too large, such that \( \text{cosh}(x) \) cannot be represented, \( \	ext{errno} \) is set to \( \text{ERANGE} \) and \( \text{cosh} \) returns \( \text{HUGE}_\text{VAL} \).
**exp** — Exponential ($e^x$).

**INTERFACE**

```c
double exp(double x);
```

**DESCRIPTION**

Returns $e^x$. If $x$ is too large, such that $e^x$ cannot be represented, *errno* is set to *ERANGE* and the return value of *exp* has magnitude *HUGE_VAL* and has the same sign as $x$. If $x$ is too small, such that $e^x$ cannot be represented, *errno* is set to *ERANGE* and 0.0 is returned.

**SEE ALSO**

`pow` for raising a number to a power.
`ldexp` for multiplying a number by a power of 2.

**fabs** — Absolute value of a double.

**INTERFACE**

```c
double fabs(double x);
```

**DESCRIPTION**

Returns the absolute value of $x$.

**SEE ALSO**

`abs` for the absolute value of an integer.

**floor** — Floor function.

**INTERFACE**

```c
double floor(double x);
```

**DESCRIPTION**

Returns the largest integer not greater than $x$.

**SEE ALSO**

`ceil` for the ceiling function.
fmod — Floating-point remainder.

**INTERFACE**

double fmod(double x, double y);

**DESCRIPTION**

Returns the floating-point remainder of \( x/y \). It returns a value \( r \) with the same sign as \( x \) such that \( x = i \times y + r \) for some integer \( i \), where \( |r| < |y| \).

**CAUTIONS**

If \( x/y \) cannot be represented, the result is undefined.
frexp — Break a double into fraction and exponent.

INTERFACE

double frexp(double value, int *exp);

DESCRIPTION

Returns a double \( x \) with magnitude in the range \([0.5 .. 1.0]\)
and stores an int \( \text{int} \) in the object referenced by \( \text{exp} \) such that
\( \text{value} \) equals \( x \) times 2.0 raised to the power stored in \( \text{*exp} \).
If \( \text{value} \) is 0.0, both the return value and the value stored in
\( \text{*exp} \) are set to 0.0.

SEE ALSO

modf for decomposing a double into integral and fractional parts.

EXAMPLE

```c
#include <stdio.h>
#include <math.h>
main() {
    int exp;
    double x;
    x = frexp(2.4, &exp);
    printf("%e %d", x, exp);
}
```

prints

\( 6.000000e-01 2 \)
math.h

Idexp — Multiply by a power of two.

INTERFACE

double ldexp(double x, int exp);

DESCRIPTION

Returns \( x \times 2^{\text{exp}} \). If the result would exceed \( \text{HUGE\_VAL} \) in magnitude, \( \text{errno} \) is set to \( \text{ERANGE} \) and the return value has magnitude \( \text{HUGE\_VAL} \) and has the same sign as \( x \).

SEE ALSO

pow for raising a number to a power.
exp for raising \( e \) to a power.

log — Natural logarithm: base \( e \).

INTERFACE

double log(double x);

DESCRIPTION

Returns the logarithm of \( x \) to the base \( e \). If \( x \leq 0.0 \), \( \text{errno} \) is set to \( \text{EDOM} \) and \( -\text{HUGE\_VAL} \) is returned.

SEE ALSO

log10 for the common (base 10) logarithm.
exp for raising \( e \) to a power.
log10 — Common logarithm: base ten.

INTERFACE

double log10(double x);

DESCRIPTION

Returns the logarithm of \( x \) to the base 10. If \( x \leq 0.0 \), \texttt{errno} is set to \texttt{EDOM} and \texttt{-HUGE\_VAL} is returned.

SEE ALSO

log for the natural (base e) logarithm.

modf — Break a double into integer and fractional parts.

INTERFACE

double modf(double value, double *iptr);

DESCRIPTION

Returns the fractional part of \texttt{value}. The integral part of \texttt{value} is stored in the object pointed to by \texttt{iptr}.

SEE ALSO

frexp for breaking a double into a fraction and exponent.

EXAMPLE

```c
#include <stdio.h>
#include <math.h>
main() {
    double x, i;
    x = modf(2.4, &i);
    printf("%e %e", x, i);
}

prints
4.000000e-01 2.000000e+00
```
pow — Raise a double to a power.

**INTERFACE**

double pow(double x, double y);

**DESCRIPTION**

Returns $x^y$. If $x = 0.0$ and $y = 0.0$, `errno` is set to `EDOM` and 0.0 is returned. If $x < 0.0$ and $y$ is not an integral number, `errno` is set to `EDOM` and `-HUGE_VAL` is returned. If $|x^y| > HUGE_VAL$, the return value of `pow` has magnitude `HUGE_VAL` and has the same sign as $x^y$.

**SEE ALSO**

`exp` for raising $e$ to a power.

`ldexp` for multiplying a number by a power of 2.

sin — Sine.

**INTERFACE**

double sin(double x);

**DESCRIPTION**

Returns the sine of $x$, where $x$ is in radians. `NAN` is returned when $|x| > HUGE_VAL$.

**SEE ALSO**

`asin` for the arc sine.

sinh — Hyperbolic sine.

**INTERFACE**

double sinh(double x);

**DESCRIPTION**

Returns the hyperbolic sine of $x$. If the magnitude of $x$ is too large, such that $\sinh(x)$ cannot be represented, `errno` is set to `ERANGE` and the return value of `sinh` has magnitude `HUGE_VAL` and has the same sign as $x$. 
**sqrt** — Square root.

**INTERFACE**

`double sqrt(double x);`

**DESCRIPTION**

Returns the non-negative square root of `x`. If `x < 0.0`, `errno` is set to `EDOM` and 0.0 is returned.

**tan** — Tangent.

**INTERFACE**

`double tan(double x);`

**DESCRIPTION**

Returns the tangent of `x`, where `x` is in radians. `NAN` is returned when `|x| > HUGE_VAL`.

**SEE ALSO**

`atan` for the arc tangent of an angle.

`atan2` for the arc tangent of the angle defined by a point.

**tanh** — Hyperbolic tangent.

**INTERFACE**

`double tanh(double x);`

**DESCRIPTION**

Returns the hyperbolic tangent of `x`: \( \frac{\sinh(x)}{\cosh(x)} \). If the magnitude of `x` is too large, such that \( \sinh(x) \) or \( \cosh(x) \) cannot be represented, `errno` is set to `ERANGE` and the return value of `tanh` has magnitude `HUGE_VAL` and has the same sign as `x`.

**SEE ALSO**

`sinh` for the hyperbolic sine of an angle.

`cosh` for the hyperbolic cosine of an angle.
setjmp.h

6

setjmp.h

The header file *setjmp.h* defines a type and two functions for setting up and executing non-local jumps.

**jmp_buf** — A type used by *setjmp* and *longjmp*.

**INTERFACE**

```c
#include <setjmp.h>
```

**DESCRIPTION**

An array type that can hold the information needed to restore a calling environment.

**longjmp** — Execute a non-local jump.

**INTERFACE**

```c
#include <setjmp.h>
void longjmp(jmp_buf env, int val);
```

**DESCRIPTION**

Restores (jumps to) the calling environment referenced by `env`. `env` must have been initialized by a previous call to *setjmp(env)*. `env` references the calling environment existing at the point of the most recent call to *setjmp(env)*.

*longjmp* does not return in the classic sense. Completion of a *longjmp* call appears to be a return of the corresponding call to *setjmp* (that is, the most recent call to *setjmp* with the same argument `env`). Such a “return” from *longjmp* (which appears to be a return from *setjmp*) returns non-zero. If `val` is zero, the “return” value from *longjmp* is one; otherwise it is `val`. As the return value from an actual call to *setjmp* is zero, the apparently equivalent kinds of “returns” from
setjmp.h

set jmp can be distinguished. (This protocol is patterned after that of the "fork" call of UNIX).

Calling long jmp has an indeterminate effect on any variables of storage class register declared within the function containing the corresponding set jmp, but has no effect on any other objects.

CAUTIONS

If env has not been initialized by set jmp, or if the routine in which the call to set jmp occurred has returned before the supposedly corresponding call to long jmp, the behavior is undefined. The use of an uninitialized or dangling environment reference will result in attempting to execute in an environment that is in a shambles.

SEE ALSO

set jmp to save a reference to the current calling environment for a subsequent non-local jump.

EXAMPLE

This example illustrates the use of long jmp and set jmp.

#include <setjmp.h>
jmp_buf Buf; int loopcnt;
static void loop(j) {
    if (++loopcnt < j) loop(j);
    else longjmp(Buf,1);
    /* Call to longjmp discards j invocations */
    /* of loop() and returns to the switch. */
}
void main () {
    int j;
    for (j = 1; j <= 10; j++) {
        loopcnt = 0;
        switch(setjmp(Buf)) {
            case 0: loop(j); break;
            case 1: printf("%d ",loopcnt); break;
            default: printf("ERROR!");
        }
    }
}
setjmp — Save a reference to the current calling environment for a subsequent non-local jump.

**INTERFACE**

```
#include <setjmp.h>
int setjmp(jmp_buf env);
```

**DESCRIPTION**

Saves the information required for a return to the current calling environment in `env` for use by `longjmp`. `setjmp` returns zero.

**CAUTIONS**

Correctly paired, `setjmp` and `longjmp` are fairly innocuous. If they are incorrectly paired, the behavior is undefined and may result in jumping to a non-existent environment.

**SURPRISES**

A call to `setjmp` may not appear as an argument to a function call.

**SEE ALSO**

`longjmp` to execute a non-local jump.

**EXAMPLE**

See the example for `longjmp` below.
signal.h

7

signal.h

Under MS-DOS, the High C Library does not provide implementations of the functions in header signal.h. The header file is provided so type checking may be done.

Header file signal.h declares two functions and defines several macros for handling signals.

SIGABRT — A macro used as an argument to signal and kill.

INTERFACE
#include <signal.h>

DESCRIPTION
Expands to the signal number corresponding to an abnormal termination. It may be used as the first argument to signal or the second argument to kill.

SIGFPE — A macro used as an argument to signal and kill.

INTERFACE
#include <signal.h>

DESCRIPTION
Expands to the signal number corresponding to an erroneous arithmetic operation. It may be used as the first argument to signal or the second argument to kill.
SIGILL — A macro used as an argument to `signal`
and `kill`.

INTERFACE

```
#include <signal.h>
```

DESCRIPTION

Expands to the signal number corresponding to detection
of an invalid function image. It may be used as the first
argument to `signal` or the second argument to `kill`.

SIGINT — A macro used as an argument to `signal`
and `kill`.

INTERFACE

```
#include <signal.h>
```

DESCRIPTION

Expands to the signal number corresponding to receipt of
an interactive attention signal such as Control C. It may be
used as the first argument to `signal` or the second
argument to `kill`.

SIGSEGV — A macro used as an argument to `signal`
and `kill`.

INTERFACE

```
#include <signal.h>
```

DESCRIPTION

Expands to the signal number corresponding to an invalid
access to a data object. It may be used as the first
argument to `signal` or the second argument to `kill`.
SIGTERM — A macro used as an argument to signal and kill.

**INTERFACE**

```c
#include <signal.h>
```

**DESCRIPTION**

Expands to the signal number corresponding to a termination request sent to the program. It may be used as the first argument to signal or the second argument to kill.

SIG_DFL — A macro used as an argument to signal.

**INTERFACE**

```c
#include <signal.h>
```

**DESCRIPTION**

Expands to a constant expression of type “void (*function)()” (pointer to a function returning void), that is distinct from all values obtainable by declaring such a function, and is distinct from SIG_IGN and SIG_ERR. It is used as the second argument to signal to specify that a given signal is to be handled in an implementation-defined default manner.

SIG_ERR — A macro used as an argument to signal.

**INTERFACE**

```c
#include <signal.h>
```

**DESCRIPTION**

Expands to a constant expression of type “void (*function)()” (pointer to function returning void), that is distinct from all values obtainable by declaring such a function, and is distinct from SIG_IGN and SIG_DFL. It is used as a return value from signal to indicate that signal was unable to correctly perform its function.
SIG_IGN — A macro used as an argument to `signal`.

**INTERFACE**

```c
#include <signal.h>
```

**DESCRIPTION**

SIG_IGN expands to a constant expression of type "void (*function)()" (pointer to function returning void), that is distinct from all values obtainable by declaring such a function, and is distinct from SIG_DFL and SIG_ERR. It is used as the second argument to `signal` to specify that a given signal is to be ignored.

**kill** — Send a signal.

**INTERFACE**

```c
#include <signal.h>
int kill(int pid, int sig);
```

**DESCRIPTION**

Sends the signal `sig` to the executing program specified by `pid`. If `pid` is zero, `sig` is sent to the program that called `kill`. See `SYSTEM DEPENDENCIES` for the meanings of other values for `pid`.

**CAUTIONS = SURPRISES**

In the current version of the library on MS-DOS, `kill` does not send a signal.

**SYSTEM DEPENDENCIES**

Under MS-DOS, `kill` prints a message on standard error and returns -1.

**SEE ALSO**

`signal` to set up a signal handler.
signal.h

signal — Set up a signal handler.

INTERFACE

#include <signal.h>
void (*signal(int sig, void (*fune)(int))(int);

DESCRIPTION

Arranges for fune to be used as the signal handler for the signal sig. If fune is SIG_IGN, the signal is ignored. If fune is SIG_DFL, the signal is handled in the default manner. (Since signal has not been implemented on MS-DOS, there is no default manner; see the header file for the default on other systems.) Otherwise, fune should point to a function to be called when sig is received.

When a signal sig is received, if the signal handler for sig is not SIG_IGN or SIG_DFL, it is handled as follows. First, the signal handler for sig is reset to the default — which is equivalent to executing signal(sig,SIG_DFL). Next, the signal handler for sig is called with sig as its argument — the equivalent of (*fune)(sig) is executed. If fune returns, i.e. if fune does not call abort, exit, or longjmp, then if sig was SIGFPE, the behavior is undefined; otherwise execution resumes at the point at which the signal was received.

How the various signals can arise is discussed under SYSTEM DEPENDENCIES below.

At program start-up time, the equivalent of signal(sig,SIG_DFL) is executed for each signal.

CAUTIONS = SURPRISES

In the current version of the library on MS-DOS, signal does not set up a signal handler.

SYSTEM DEPENDENCIES

Under MS-DOS, signal prints a message on standard error and returns a pointer to the library function exit.

SEE ALSO

kill to send a signal.
Header file *stdarg.h* provides a type and several macros for handling varying numbers and types of arguments to a given function.

A function may be called with varying numbers of arguments of varying types. The macros defined in *stdarg.h* assist in advancing through the actual sequence of arguments in order, first to last, one by one.

To use the variable argument macros, the function must have at least one declared argument whose name and type are known. For example, the functions

```c
void f(x), ff(x,y,z,...); int fff(x,y), *ffff(a,...);
```

meet the requirements, while

```c
char *g(), gg(...);
```

do not. The rightmost named argument is used as a starting place for advancing through the sequence.

**va_list** — A type used by the varying-argument macros.

**INTERFACE**

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

**DESCRIPTION**

Expands to a type suitable for holding the information needed by the varying-argument macros. The function using the macros must declare a variable of this type. See the example provided in the description of **va_arg**.
va_start — Initialize a va_list.
— Provided as a macro only.

INTERFACE

```
#include <stdarg.h>
void va_start(va_list ap, parmN);
```

DESCRIPTION

Initializes ap, using parmN, which must be the rightmost named argument of the function whose unnamed arguments are desired, such that the va_arg macro can return the first unnamed argument. va_start must be called before va_arg.

parmN is evaluated more than once.

SEE ALSO

va_arg to reference the next argument in sequence.
va_end to terminate va_list processing.

EXAMPLE

See the example for va_arg.
va_arg — Reference the next argument in a sequence of unnamed arguments.
— Provided as a macro only.

**INTERFACE**

```
#include <stdarg.h>
void va_arg(va_list ap,type); /* Substitute a type */
/* specifier for type. */
```

**DESCRIPTION**

Returns the value of the next, as yet unreferenced (by va_arg), argument of the function whose rightmost named argument was previously passed to `va_start`. `va_start` must be called before the first call to `va_arg`. The first call to `va_arg` returns the value of the argument following the rightmost named argument (that passed to `va_start`). Subsequent calls to `va_arg` return the values of corresponding subsequent arguments. `type` must be a type specifier such that a pointer to an object of that type can be obtained by appending a "**" to `type`. The type specifier should agree with the type of the corresponding argument (as widened by default function-call conversion).

`type` is evaluated more than once. This should be harmless, as a type specifier should not have side effects.

**CAUTIONS**

`va_arg` assumes that `ap` has been initialized by `va_start`. Calling `va_arg(ap,type)` when `va_start` has not been called correctly results in an indeterminant return value.

`va_arg` has no knowledge of the number of actual arguments available for referencing. Calling `va_arg` after all arguments have been referenced (by `va_arg`) results in an indeterminant return value.

**SEE ALSO**

`va_start` to initialize a `va_list`.

`va_end` to terminate `va_list` processing.
va_arg — Continued.

EXAMPLE

```
#include <stdarg.h>
define MAXARGS 100

void out(int n_ptrs, char *strings[]) {
  int i = 0;
  if (n_ptrs < 0) return;
  while (i < n_ptrs) printf("%s", strings[i++]);
}

void collect_args(int n_args, ...) {
  va_list ap;
  char *args[MAXARGS];
  int ptr_no = 0;
  if (n_args > MAXARGS) n_args = MAXARGS;
  va_start(ap, n_args);
  while (ptr_no < n_args)
    args[ptr_no++] = va_arg(ap, char *);
  va_end(ap);
  out(n_args, args);
}

main() {
  collect_args(3,"This ", "strikes me as ",
               "a little wierd.\n");
}
```

prints

This strikes me as a little wierd.
va_end — Terminate va_list processing.
— Provided as a macro only.

INTERFACE

#include <stdarg.h>
void va_end(va_list ap);

DESCRIPTION

Cleans up the processing of unnamed arguments so that a normal return may occur from a function using the variable argument macros. va_end must be called after all arguments have been accessed. Under some systems va_end expands to nothing, so it has zero cost; for portability it is best to always use va_end. See SYSTEM DEPENDENCIES.

SYSTEM DEPENDENCIES

Under MS-DOS va_end is an empty macro.

SEE ALSO

va_start to initiate va_list processing.
va_arg to reference the next argument in sequence.

EXAMPLE

See the example for va_arg.
The header file `stdefs.h` defines useful macros, some of which are defined in other header files as well, and declares a variable in which library errors are flagged.

**errno** — An int used to record error numbers.

**INTERFACE**
```
#include <stdefs.h>
```

**DESCRIPTION**
A variable of type int set to a positive integer error code by some library functions if an error occurs during their execution. It is set to zero on program start up. It is never set to zero by any library functions. A program that uses **errno** for error checking should set it to zero before a call to a library function and inspect it prior to a subsequent call to a library function.

**NULL** — A macro used to represent the null pointer.

**INTERFACE**
```
#include <stdefs.h>
```

**DESCRIPTION**
Expands to a value that is assignment-compatible with any pointer type and compares equal with the constant zero. It is therefore suitable as a representation of the null pointer. Note that **NULL** is not appropriate as the terminating character of a string, as the size of a pointer is not necessarily the same as the size of the character **NUL**.

**NULL** is also defined in `stdio.h` and `stdlib.h`. 
ptrdiff_t — A macro used as a type specifier.

INTERFACE

#include <stdefs.h>

DESCRIPTION
Expands to the integral type of the result of subtracting two pointers.

size_t — A macro used as a type specifier.

INTERFACE

#include <stdefs.h>

DESCRIPTION
Expands to the integral type of the result of the sizeof operator.

size_t is also defined in stdio.h, stdlib.h, and string.h.
The header file `stdio.h` declares functions, macros, a type, and a variable useful for performing input and output (I/O).

**Streams.** Input and output may be performed to and from various physical devices: the console, various disk drives, tape drives, the printer, etc. To provide a consistent interface to devices that have varying properties, input and output are mapped to logical data streams. These streams are associated with files, which may correspond to external storage (as in a disk file) or may not (as in the console or printer).

A stream is an ordered sequence of bytes. It may be buffered, which means that bytes are typically read (written) in standard amounts from (to) an external device into a buffer and read (written) in arbitrary amounts from (to) the buffer to (from) objects in a program. The buffer may be designated as an object of the program or it may be automatically allocated by the library.

If a file can support positioning requests, a file pointer associated with a stream maintains a record of the current position in the file through read, write, and positioning requests. If a file cannot support positioning requests, all input and output is done at the end of the stream. Aside from positioning, logical streams hide any differences among the various types of input and output devices.

Opening a file associates a stream with the file, positioning the file pointer at the beginning of the file (if the file can support a file pointer). It may involve creating the file, which causes any previous contents to be deleted.

Closing a file disassociates the stream from the file. If an output stream is buffered (see below), the buffer is flushed into
stdio.h

the stream when the file is closed. The file may be subsequently reopened. All files are closed if the function `main` returns or if the function `exit` is executed. At program startup three files are opened automatically: see the `stdin`, `stdout`, and `stderr` macros below.

Errors. A number of I/O functions return values that make it possible to distinguish failure from success. For example, `fputc` returns its input value if it succeeds in putting the character in the stream, and `EOF` if it fails.

In the case of a function that writes to a buffered output stream, a return value denoting success means that either it has successfully written to the external storage or that it has successfully written to the stream’s buffer. This means that such a function can return information to the effect that it has successfully written characters, but a subsequent write error may prevent those characters from being successfully written to external storage.

If a write error does occur, `errno` and the stream’s end of file flag (each discussed below) are set. The one case where it is impossible to return information about write failures is if the failure occurs when a stream’s buffer is being flushed at program termination.

If an application requires certainty about the success of a write, it should call `fflush` to explicitly flush the buffer. If no errors have been reported previously and `fflush` returns a value denoting success, all output has been successfully written to external storage, i.e. all output has been accepted by the operating system for writing — however, external events could cause the operating system to fail to perform the actual physical I/O.

End of file. Associated with each stream is an end-of-file flag. The end-of-file flag gets set when a read request occurs and the file pointer is already positioned at the end of the file. Thus it is not always the case that when the file pointer is positioned at the end of a file, the end-of-file flag has been set.
for the stream. If the end-of-file flag is set for a stream (see `feof`), a read request results in no characters being read (assuming `fseek` has not moved the file pointer from the end of the file — `fseek` does not clear the end-of-file flag).

An error flag is also associated with each stream. If a read error or write error has occurred on the stream, the error flag is set. Both the end-of-file and error flags are sticky, that is, they are not cleared by subsequent calls to library functions, except `clearerr` and `rewind`, whose functions are specifically to clear them.

Text and binary streams. A stream can be designated as a text stream or a binary stream.

Transformations may be performed upon the bytes making up a text stream (which are assumed to represent characters) upon input or output, in the case that the external representation of a character sequence does not match the C language representation of that sequence.

A binary stream has the characteristic that no conversions are performed on data upon input or output: a sequence of bytes stored on an external device matches exactly storage internal to a program containing the result of reading that sequence of bytes, which again exactly matches external storage containing the result of writing that sequence of bytes. See the System Dependencies paragraph below for specific information about text and binary streams on this system.

`errno`. Many of the functions declared in `stdio.h` reference the `int` variable `errno`, which is declared in `stdefs.h`. `errno` is set to a positive integer error code by some library functions when an error occurs during their execution. It is set to zero on program start up. It is never set to zero by any library function. To reference `errno` from a program, either the header file `stdefs.h` must be included, or `errno` must be declared to be external (`extern int errno;`).
System Dependencies. Under MS-DOS, line terminators are represented in files by a carriage return followed by a linefeed (\r\n). The C language expects a single character line terminator (\n). If a stream is designated as a text stream, \r\n is converted to \n upon input, and \n is converted to \r\n upon output. Single linefeeds must not occur in the external representation of a text stream. If a stream is designated as a binary stream, no such conversions occur.

Under MS-DOS, many editors and system utilities treat the character Control Z (ASCII 26) as an end-of-file indicator. Accordingly, if a Control Z is encountered in an input text stream, it indicates end-of-file: no further characters are read from the stream. The Control Z itself is ignored (in the sense that it is read by library input functions, but is not returned by them — fgetc(F) returns EOF if F is a text stream whose file pointer points at Control Z). Control Z is not treated in any special manner by binary streams.

A stream is designated as a text or binary stream when the file associated with the stream is opened. See fopen for a complete description. A function setmode is provided for changing the designation of a stream at any point. Streams not explicitly designated as text or binary by the type argument to fopen, and streams that are not explicitly created by a call to fopen or freopen (such as stdin), are designated as one or the other according to the default value in the int variable fmode. See the description of setmode for information about fmode.

If data are read from a binary stream and written to a text stream, the resulting file will contain an extra carriage return preceding each end-of-line. Conversely if data are read from a text stream and written to a binary stream, the resulting file will contain no carriage returns preceding each end-of-line.

An input text stream whose associated file is a device, such as the keyboard, that does not support file positioning converts carriage returns to linefeeds and ignores linefeeds.
FILE — A type used to control a stream.

INTERFACE
   \#include <stdio.h>

DESCRIPTION
   A struct type that can contain the information required to control a stream. Details are given in the header file. A variable of type FILE * is commonly used to designate a particular stream.

BUFSIZ — A macro used to specify the size of a buffer.

INTERFACE
   \#include <stdio.h>

DESCRIPTION
   Expands to the size (in bytes) of the buffers used by streams.

EOF — A macro used as an end-of-file indicator.

INTERFACE
   \#include <stdio.h>

DESCRIPTION
   Expands to a negative integer used to indicate end-of-file.

L_tmpnam — A macro that specifies the size of a temporary filename.

INTERFACE
   \#include <stdio.h>

DESCRIPTION
   Expands to the size (in bytes) of an array large enough to hold a temporary filename generated by a call to tmpnam.
NULL — A macro used to represent the null pointer.

INTERFACE
#include <stdio.h>

DESCRIPTION
Expands to a value that is assignment-compatible with any pointer type and compares equal with the constant zero. It is therefore suitable as a representation of the null pointer. Note that NULL is not appropriate as the terminating character of a string, as the size of a pointer is not necessarily the same as the size of the character NUL.

NULL is also defined in stdfs.h and stdlib.h.

SEEK_CUR, SEEK_END, SEEK_SET — Macros used as an argument to fseek.

INTERFACE
#include <stdio.h>

DESCRIPTION
Each expands to an integer to be used as the third argument to fseek to indicate from where fseek should seek.

SEEK_CUR indicates the current position of the file pointer.
SEEK_END indicates the end of the file.
SEEK_SET indicates the beginning of the file.
stdio.h

size_t — A macro used as a type specifier.

INTERFACE

#include <stdio.h>

DESCRIPTION

Expands to the integral type of the result of the sizeof operator.

size_t is also defined in stdiefs.h, stdlib.h, and string.h.

stdin, stdout, stderr — Macros that designate the standard input, standard output, and standard error streams, respectively.

INTERFACE

#include <stdio.h>

DESCRIPTION

Each expands to an expression that points to a FILE variable that is initialized at program start up to control a particular stream.

stderr is associated with the standard error file, which is initialized as write-only and writes to the screen.

stdin is associated with the standard input file, which is initialized as read-only and reads from the keyboard.

stdout is associated with the standard output file, which is initialized as write-only and writes to the screen.

stderr is by default an unbuffered file, while stdin and stdout are by default unbuffered if they reference the keyboard and screen respectively, otherwise they are by default buffered. The defaults can be overridden by calls to setbuf.

These streams are designated as text or binary streams according to the (link-time) value of _fmode. See fopen,
setmode, and the discussion under the System Dependencies paragraph for more complete information on text and binary streams.

SYS_OPEN — A macro that specifies the maximum number of open files.

INTERFACE
#include <stdio.h>

DESCRIPTION
Expands to the maximum number of files that may be open at one time, including the standard files mentioned below and two that are reserved for internal use. This is independent of any system limit on the maximum number of open files.

TMP_MAX — A macro that controls tmpnam.

INTERFACE
#include <stdio.h>

DESCRIPTION
Expands to the minimum number of unique file names that tmpnam generates.
clearerr — Clear end-of-file and error flags of a file.
— Provided as a macro and as a function.

**INTERFACE**

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

**DESCRIPTION**

Resets the end-of-file and error flags for the stream associated with `stream`. These flags are cleared only when the file is opened, or by explicit calls to `clearerr` and `rewind`.

**SEE ALSO**

- `feof` to test for end-of-file.
- `ferror` to test for a read or write error on a file.

fclose — Close a file.

**INTERFACE**

```c
#include <stdio.h>
int fclose(FILE *stream);
```

**DESCRIPTION**

Causes `stream` to be disassociated from the file it has controlled, and the file to be closed. If the file is open for writing and is buffered, any data in the buffer associated with `stream` is written to the file. If the buffer was automatically allocated, as opposed to being explicitly associated with `stream` via `setbuf`, it is deallocated. `fclose` returns zero if the operation is successful. If the `fclose` operation fails, `errno` is set such that a call to `perror` will result in an error message describing the reason for the failure, and the (non-zero) value of `errno` is returned. `fclose` also returns non-zero if the file was already closed.
If neither `fclose` nor `fflush` has been called on a buffered output stream and the program terminates abnormally (without returning from `main` or calling `exit`, as for example by an interrupt from the keyboard), the data in the associated buffer at program termination (if any) is not written to the file.

**SEE ALSO**
- `fopen` to open a file.
- `setbuf` to associate a particular buffer with a stream.
- `fflush` to flush a stream’s buffer without closing the file.

**EXAMPLE**
The program fragment below attempts to open `file1` for reading. It later attempts to close `file1` prior to opening `file2` for writing (using the same `FILE` variable).

```c
#include <stdio.h>
define FAILURE (-1)
FILE *FP;
if ((FP = fopen("file1", "r")) == NULL) {
    perror("fopen of file1");
    return FAILURE;
}

... 
if (fclose(FP) != 0)
    perror("fclose of file1");
if ((FP = fopen("file2", "w")) == NULL) {
    perror("fopen of file2");
    return FAILURE;
}
... 
```
feof — Test for end-of-file.
— Provided both as a macro and a function.

INTERFACE

#include <stdio.h>
int feof(FILE *stream);

DESCRIPTION

Returns non-zero if the end-of-file indicator for the file associated with stream is set, zero if it is clear.

SEE ALSO

ferror to test for a read or write error on a file.
clearerr to clear end-of-file and error flags of a file.

ferror — Test for a read or write error on a file.
— Provided both as a macro and a function.

INTERFACE

#include <stdio.h>
int ferror(FILE *stream);

DESCRIPTION

Returns non-zero if the error indicator for the file associated with stream is set, zero if it is clear.

SEE ALSO

feof to test for end-of-file.
clearerr to clear end-of-file and error flags of a file.
fflush — Flush a file's buffer.

INTERFACE

#include <stdio.h>
int fflush(FILE *stream);

DESCRIPTION

Causes any data in the buffer of the output stream associated with stream to be written to the associated file. (The buffer is automatically flushed when full and at normal program termination.) fflush returns zero when successful. If the file was not open for writing or a write error occurs, errno is set such that a call to perror will result in an error message, and the (non-zero) value of errno is returned.

CAUTIONS

If neither fflush nor fclose has been called on a buffered output stream and the program terminates abnormally (without returning from main or calling exit, as for example by an interrupt from the keyboard), the data in the associated buffer at program termination (if any) is not written to the file.

SEE ALSO

fclose to close a file.
fgetc — Get a character from a file.

INTERFACE

#include <stdio.h>
int fgetc(FILE *stream);

DESCRIPTION

Returns the next character from the input stream stream. The associated file pointer (if one is defined) is advanced one character. If the stream is at end-of-file or a read error occurs, EOF is returned.

SYSTEM DEPENDENCIES

If stream is a text stream, the MS-DOS line terminator (\r\n) is converted into the C line terminator (\n). In addition, a Control Z (\032) is interpreted as an end-of-file indicator, and EOF is returned. If stream is a binary stream, no such conversion occurs, and Control Z encountered in the input stream is not interpreted specially but returned as is.

SEE ALSO

getc to get a character from a file.
getchar to get a character from standard input.
fgets to get a line of text from a file.
gets to get a line of text from standard input.
putc to write a character to a file.
putc to write a character to a file.
putchar to write a character to standard output.
fputs to write a string to a file.
puts to write a string to standard output.
ungetc to push a character back into an input stream.
fgetc — Continued.

EXAMPLE

The program below copies `file1` character-by-character to `file2` and to standard output. This example illustrates the use of `fgetc`, `fputc`, and `putchar`. Similar examples are given under the `EXAMPLE` sections of `getc` and `getchar`, pointing out the differences between the functions.

```c
#include <stdio.h>
#define FAILURE (-1)

main() {
    FILE *FP1, *FP2;
    if ((FP1 = fopen("file1", "r")) == NULL) {
        perror("fopen of file1");
        return FAILURE;
    }
    if ((FP2 = fopen("file2", "w")) == NULL) {
        perror("fopen of file2");
        return FAILURE;
    }
    while (feof(FP1))
        fputc(putchar(fgetc(FP1), FP2);
}
```
fgets — Read a line of text from a file.

INTERFACE

```
#include <stdio.h>
char *fgets(char *s, int n, FILE *stream);
```

DESCRIPTION

Reads a line of text from the file associated with `stream` into the string `s`, appending a final NUL. No more than `n`-1 characters are read. No characters are read after a newline is encountered. The newline becomes the last (non-NUL) character stored. No characters are read after end-of-file is encountered.

`fgets` returns `s` if successful. If end-of-file is encountered before any characters have been read, the array is unchanged and `fgets` returns NULL. If a read error occurs, the array contents are indeterminate and `fgets` returns NULL. `feof` and `ferror` can be used to distinguish between end-of-file and error.

CAUTIONS

`fgets(s, n, stream)` should never be called with `n` greater than the length of the string `s`.

SURPRISES

`fgets` and `gets` are inconsistent: both functions return after encountering a newline, but `fgets` stores the newline while `gets` discards it.

SYSTEM DEPENDENCIES

If `stream` is a text stream, MS-DOS line terminators (`\r\n`) are converted into C terminators (`\n`). In addition, a Control Z (`\032`) is interpreted as an end-of-file indicator, and `EOF` is returned. If `stream` is a binary stream, no such conversion occurs, and Control Zs encountered in the input stream are not interpreted specially but returned as is.
fgets — Continued.

SEE ALSO

fgets, fgetc, getc, getchar, gets, fputs, putc, putchar, fputs, puts, ungetc.

For short descriptions of all of the listed functions, see the SEE ALSO section for fgetc.

EXAMPLE

The program below copies file1 line by line to file2. This example illustrates the use of fgets, fputs, and fopen.

```c
#include <stdio.h>
#define FAILURE (-1)
#define LINESIZE (80)
main() {
    FILE *FP1, *FP2;
    char line[LINESIZE];
    if ((FP1 = fopen("file1", "r")) == NULL) {
        perror("fopen of file1");
        return FAILURE;
    }
    if ((FP2 = fopen("file2", "w")) == NULL) {
        perror("fopen of file2");
        return FAILURE;
    }
    while (fgets(line, LINESIZE, FP1)) {
        /* fgets returns NULL (FALSE) when it */
        /* cannot get any more characters. */
        fputs(line, FP2);
    }
}
```
fopen — Open a file.

INTERFACE

#include <stdio.h>
FILE *fopen(char *pathname, char *type);

DESCRIPTION

Opens the file named by the string pathname. A FILE variable is associated with the file and a pointer to it is returned. This file variable controls the newly created input or output stream associated with the file. If the file cannot be opened a null pointer is returned.

The string type indicates how the file is to be opened. It must contain one (and only one) of "r", "w", or "a", and can also contain any combination of "+", "b", "t", and "u".

The meanings of the type strings are as follows:

"r" open existing file for reading.
"w" create file for writing. If file exists, truncate it.
"a" open or create file for appending to end of file.
"r+" open existing file for update.
"w+" create file for update. If file exists, truncate it.
"a+" open or create file for update, writing to end of file.

If the string contains "u", or if the file being opened references the keyboard or screen, the resulting stream is unbuffered, that is, each read or write request results in a system call to perform I/O. Otherwise, upon the first read or write request an automatically allocated buffer is associated with the stream and I/O is done in BUFSIZ-byte blocks. Prior to the first read or write request, setbuf can be called to associate a particular buffer with the stream and avoid any automatic allocation.

If the string contains "b" the resulting stream is binary, while if instead the string contains "t" the result is a text stream. See the SYSTEM DEPENDENCIES section on the distinction between text and binary streams. If the string...
contains both "t" and "b", the last one in the string takes precedence. If the string does not contain "t" or "b", a default is chosen. The default depends on the variable _fmode; for a description of _fmode, see _setmode.

If a non-existent file is opened with type "r" the open fails. If an existing file is opened with type "w", its contents are destroyed. If a file is opened with type "a", all writing is done at the end of the file, irrespective of the position of the file pointer at the time of the write request, and the buffer is flushed after each write. The file pointer can be positioned using fseek or rewind for read requests in the case of a file with type "a+".

If a file is open for update (type contains "+"), both reading and writing are allowed. When switching between reading and writing on a stream, a call to fseek or rewind is required on buffered files to clear or flush the buffer.

Cautions

If type contains "w" and pathname contains an existing file name, that file is destroyed.

System Dependencies

If stream is a text stream, MS-DOS line terminators (\r\n) are converted into C terminators (\n). In addition, a Control Z (\032) is interpreted as an end-of-file indicator, and EOF is returned. If stream is a binary stream, no such conversion occurs, and Control Zs encountered in the input stream are not interpreted specially but returned as is.

Due to these conversions, input from an unbuffered text stream is extremely slow. Where speed is important, we recommend that input text streams be buffered.

See Also

freopen to open a file using an existing FILE variable.
fclose to close a file.
**stdio.h**

`setbuf` to associate a particular buffer with a stream.

`setmode` to make a stream into a text stream or a binary stream, or to set the default for files opened without such a specification.

**EXAMPLE**

See the example for `fgets`.

**fprintf** — Print on a file.

**INTERFACE**

```
#include <stdio.h>
int fprintf(FILE *stream, char *format, ...);
```

**DESCRIPTION**

Writes to the file associated with `stream` according to the format string `format`.

`format` contains conversion specifications that specify how to represent subsequent arguments in print. Any characters that are not elements of conversion specifications are simply printed out unchanged. The representation of each argument following `format` is printed out at the point in the string where its conversion specification appears. The conversion specifications act as place-holders for a printable representation of the corresponding argument. `format` is printed with each specification replaced by the representation of its argument.

If the number of actual arguments is fewer than the number of arguments specified by `format`, `fprintf` traipses merrily through memory printing whatever it finds in its presumed argument list according to the specification in the format string. If the number of arguments passed in is greater than the number specified by `format`, the excess arguments are evaluated by the standard function call mechanism, but otherwise ignored.
Conversion specifications are of the following form:

'%' Flag* Field_width? Precision? Size? C_char

where

Flag -> '-' | '+' | ' ' | '#' | '0';
Field_width -> ((Digit - '0' '+')? Digits) | '*';
Precision -> '.' (Digits | 'x');
Size -> 'h' | 'l' | 'L';
C_char -> 'd' | 'i' | 'o' | 'u' | 'x' | 'X' |
         'f' | 'e' | 'E' | 'g' | 'G' | 'c' |
         's' | 'p' | 'n' | '%';

(See the introduction for an explanation of this regular expression notation.)

'%' sets off a conversion specification from ordinary characters.

Each conversion takes place in a field of some number of characters. The minimum size of the field can be specified by the Field_width. If no Field_width appears, the field is the size of the result of the conversion. If the result of the conversion contains fewer characters than the Field_width specifies, the extra space is padded with spaces or zeros. If Field_width is not an integer but is '*', the value of Field_width is to be taken from an int argument that precedes the argument to be converted. If there are more characters resulting from a conversion than specified by Field_width, the field is expanded so that they all get printed — Field_width never causes truncation.

The Precision specification's meaning varies from one conversion type to the next, but generally specifies a maximum or minimum number of significant characters to appear. Precision differs from Field_width in that Field_width can cause padding to occur, but can never affect the "value" of the result, while Precision affects the characters produced by converting some argument (for
example, Precision can cause string truncation, or affect the number of characters to appear after the decimal point in a double conversion). If Precision is not an integer but is "**", the value of Precision is to be taken from an int argument that precedes the argument to be converted. If both Field_width and Precision are specified by asterisks, the Field_width argument comes first, then the Precision argument, then the argument to be converted. A negative Precision is taken as if Precision were missing.

The Size specification is used to specify the size of an argument whose type comes in more than one size, i.e. int versus long int. The Size specification is mentioned below in the description of each conversion that it can affect. If a size is specified for a conversion that it cannot affect, it is ignored.

Finally the conversion character (C_char) specifies the type of conversion; both the type of the argument to be converted and the format of the converted result (modified, of course, by Flags, Field_width, Precision, and Size).

The meaning of the flag characters are:

'-' The result of the conversion is left-justified within the field. Any padding appears on the right. If '-' does not appear, the result is right justified.

'+ ' The result of a signed conversion begins with a plus or minus sign. Negative values are printed beginning with a minus sign, and positive values are printed beginning with a plus sign. If neither '+' nor `' ' appear, negative values begin with a minus sign, and positive values begin with the first digit of the result.

' ' The result of a signed conversion begins with a space or minus sign. Negative values are printed beginning with a minus sign, and positive values are printed beginning with a space. If both the ' ' flag and the '+' flag appear, the ' ' flag is ignored. If neither '+' nor ' ' appear, negative values begin with a minus sign, and
positive values begin with the first digit of the result.

fprintf — Continued.

'\#' The result is to be converted to an alternate form, specified below in the description of each conversion character. If no alternate form is mentioned in such a description, the flag has no effect on that conversion.

'O' Padding is by zeros. If the '0' flag does not appear, padding is by spaces. If padding is by zeros, the sign (or space, if the ' ' flag appears), if any, precedes the zeros. If padding is by spaces, the spaces precede the sign. If the result is left-justified ('-' appears), padding is by spaces on the right and the '0' flag has no effect.

The meanings of the conversion characters are:

'd', 'i' The argument A is an int that is printed out as a signed decimal number. Precision specifies the minimum number of digits to appear. If the value can be represented in fewer than Precision digits, it is expanded with leading zeros. The default precision is one. If Precision is zero and A is zero, the converted value consists of no characters. (This is independent of any padding specified by Field_width.) The 'h' size specification means A is a short int and 'l' means it is a long int.

'o' The argument A is an int that is printed out as an unsigned octal number. Precision specifies the minimum number of digits to appear. If the value can be represented in fewer than Precision digits, it is expanded with leading zeros. The default precision is one. If Precision is zero and A is zero, the converted value consists of no characters. (This is independent of any padding specified by Field_width.) The 'h' size specification means A is a short int and 'l' means it is a long int. If the '#' flag appears, '0' is prepended to the result if it is not zero.
The argument A is an int that is printed out as an unsigned decimal number. Precision specifies the minimum number of digits to appear. If the value can be represented in fewer than Precision digits, it is expanded with leading zeros. The default precision is one. If Precision is zero and A is zero, the converted value consists of no characters. (This is independent of any padding specified by Field_width.) The 'h' size specification means A is a short int and 'l' means it is a long int.

The argument A is an int that is printed out as an unsigned hexadecimal number. The letters abcdef are used for 'x' conversion, while ABCDEF are used for 'X' conversion. Precision specifies the minimum number of digits to appear. If the value can be represented in fewer than Precision digits, it is expanded with leading zeros. The default precision is one. If Precision is zero and A is zero, the converted value consists of no characters. (This is independent of any padding specified by Field_width.) The 'h' size specification means A is a short int and 'l' means it is a long int. If the '#' flag appears, Ox is prepended to the result (OX for 'x').

The argument A is a double (or a float, which is converted to double when passed as a parameter) that is printed out in decimal notation. If A is negative there is a leading minus. The integral portion of the number appears, then a decimal point, then Precision digits after the decimal point. If Precision is not specified, it defaults to six. If Precision is explicitly zero, no decimal point appears. If the number has no integral portion and Precision is not zero, a '0' is printed out before the decimal point. If the '#' flag appears, a decimal point is always printed, even if it is not followed by any digits. If Precision is zero and '#' appears, a decimal point is
stdio.h

printed. The 'L' size specification means A is a long
double.

`fprintf` — Continued.

'e', 'E' The argument A is a double (or a float, which is
converted to double when passed as a parameter) that
is printed out in decimal notation. If A is negative there
is a leading minus. One digit appears before the
decimal point, Precision digits after the decimal point,
then an e (or an E if the conversion char is 'E'), followed
by the sign of the exponent, followed by the exponent.
At least two digits are printed for the exponent. If
Precision is not specified, it defaults to six. If Precision
is explicitly zero, no decimal point appears. If the '#' flag
appears, a decimal point is always printed, even if it
is not followed by any digits. If Precision is zero and '#'
appears, a decimal point is printed. The 'L' size
specification means A is a long double.

g', 'G' The argument A is a double (or a float, which is
converted to double when passed as a parameter) that
is printed out in style 'f', 'e', or 'E' with precision
specifying the number of significant digits. Note that if
Precision is zero, the converted value consists of no
characters. (This is independent of any padding
specified by Field_width.) Style 'f' is used unless the
exponent to be printed is less than -4 or greater than
precision. In that case, style 'e' is used for 'g' and
style 'E' is used for 'G'. Trailing zeros are removed
from the result. A decimal point only appears if it is
followed by a digit. If the '#' flag appears, a decimal
point is always printed, even if it is not followed by any
digits, and trailing zeros are not removed. The 'L' size
specification means A is a long double.

c' The argument is an int, the least significant character
of which is printed.
fprintf — Continued.

's' The argument is a (char *) string. Characters from the string are printed until either a NUL is encountered (which is not printed), or Precision characters have been printed. If Precision is not explicitly set, characters are printed until a NUL is found.

'p' The argument is taken to be a pointer to an object. The action taken is system dependent — see the SYSTEM DEPENDENCIES section below.

'n' The argument is a pointer to an integer into which is written the number of characters printed thus far by the current call to fprintf.

'%' A % is printed.

If a conversion specification is invalid, that is, if the conversion character is not one of those listed above, errno is set and that conversion is ignored — no argument is converted and the would-have-been corresponding argument corresponds to the next conversion specification.

If the arguments passed in do not match those specified by format, the behavior is undefined.

fprintf returns the number of characters printed, or a negative number if a write error occurs.

CAUTIONS
The number of arguments passed to fprintf must equal (or exceed) the number specified by format, or garbage may be printed.

The types of the arguments to fprintf must match the types specified by format, or garbage may be printed.
For a '%p' conversion specification, the argument is taken to be a pointer to an object. The value of the pointer (which is the address of the object) is printed in hexadecimal; the segment is printed in four hex digits, then a colon, then the offset in four hex digits.

If stream is a text stream, C line terminators (\n) are converted into MS-DOS terminators (\r\n). If stream is a binary stream, no such conversion occurs.

SEE ALSO

fprintf to print on a file.
printf to print on standard output.
sprintf to print on a string.
vfprintf to print on a file using the variable argument macros.
vprintf to print on standard output using the variable argument macros.
vsprintf to print on a string using the variable argument macros.
fscanf to read values from a file.
scanf to read values from standard input.
sscanf to read values from a string.
EXAMPLE

```
#include <stdio.h>
#define FAILURE (-1)
main() {
    FILE *FP;
    char s[14] = "like this one";
    int i = 10, x = 255;
    double d = 3.1415927;
    if ((FP = fopen("example", "w")) == NULL) {
        perror("example");
        return FAILURE;
    }
    fprintf(FP,"fprintf prints strings (%s),\n", s);
    fprintf(FP,"decimal numbers(%d),", i);
    fprintf(FP," hex numbers(%04X),\n", x);
    fprintf(FP,"floating-point numbers (%+.4e)\n", d);
    fprintf(FP,"and percent signs(%%)."));
}
```

prints

fprintf prints strings (like this one),
decimal numbers (10), hex numbers (00FF),
floating point numbers (+3.1416e+00),
and percent signs (%).

on file example.
fputc — Write a character on a file.

**INTERFACE**

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

**DESCRIPTION**

Writes the character c (the least significant byte of the int c) on the file associated with stream at the current position of the file pointer (if one is defined), and advances the file pointer (if any). fputc returns the character c unless a write error occurs, in which case it returns EOF.

**SURPRISES**

fputc takes c as an int rather than a char parameter and returns an int rather than a char to conform to the conventions discussed in the Parameter Passing section of the introduction. If an object of type char is passed to fputc, it is automatically coerced to type int.

**SYSTEM DEPENDENCIES**

If stream is a text stream, a C line terminator (\n) is converted into an MS-DOS line terminator (\r\n). If stream is a binary stream, no such conversion occurs.

**SEE ALSO**

fgetc, getc, getchar, fgets, gets, putc, putchar, fputs, puts, ungetc.

For short descriptions of all of the listed functions, see the **SEE ALSO** section for fgetc.

**EXAMPLE**

See the example for fgetc.
fputs — Write a string to a file.

INTERFACE

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

DESCRIPTION

Writes the string `s` on the file associated with `stream` at the current position of the file pointer (if one is defined), and advances the file pointer (if any). The terminating null is not written. `fputs` returns zero if the operation is successful. If the `fputs` operation fails, `errno` is set such that a call to `perror` will result in an error message describing the reason for the failure, and the (non-zero) value of `errno` is returned.

SURPRISES

`fputs` and `puts` are inconsistent: `fputs` writes exactly the string `s`, while `puts` writes `s` followed by a newline.

SYSTEM DEPENDENCIES

If `stream` is a text stream, C line terminators (`\n`) are converted into MS-DOS terminators (`\r\n`). If `stream` is a binary stream, no such conversion occurs.

SEE ALSO

`fgetc`, `getc`, `getchar`, `fgets`, `gets`, `fputc`, `putc`, `putchar`, `puts`, `ungetc`.

For short descriptions of all of the listed functions, see the `SEE ALSO` section for `fgetc`.

EXAMPLE

See the example for `fgets`.
stdio.h

fread — Read from a file.

INTERFACE

#include <stdio.h>

int fread(void *ptr, size_t size, int nelm, FILE *stream);

DESCRIPTION

Reads nelm elements of size bytes each from the input stream associated with stream into the segment of memory beginning at the location referenced by ptr. The file pointer, if one is defined, is advanced by the number of bytes read. If a partial element is read (due either to a read error or reaching end-of-file), its value is indeterminate. fread returns the number of elements read, not counting any partial element.

If the return value of fread is less than nelm, either a read error has occurred or end-of-file has been encountered. If a read error has occurred, the position of the file pointer is indeterminate and the error flag is set for the stream. If end-of-file has been encountered and no bytes were read, the end-of-file flag is set for the stream. _feof and/or _ferror must be used to distinguish an error from end-of-file.

The first read or write request on a buffered stream causes a buffer to be allocated for the stream unless _setbuf has previously been called to associate a buffer with the stream.

CAUTIONS

fread assigns nelm * size bytes (or as many bytes as it was able to read) into the segment of memory that begins at the location referenced by ptr.
stdio.h

**fread** — Continued.

**SYSTEM DEPENDENCIES**

If *stream* is a text stream, MS-DOS line terminators (\r\n) are converted into C terminators (\n). In addition, a Control Z (\032) is interpreted as an end-of-file indicator, and EOF is returned. If *stream* is a binary stream, no such conversion occurs, and Control Zs encountered in the input stream are not interpreted specially but returned as is.

**SEE ALSO**

`fwrite` to write to a file.

`setbuf` to associate a particular buffer with a stream.

**EXAMPLE**

The program below copies `file1`, a block at a time, to `file2`. This example illustrates the use of `fread` and `fwrite`.

```c
#include <stdio.h>
#define FAILURE (-1)
#define BLOCKSIZE (512)
main() {
    FILE *FP1, *FP2;
    char buf[BLOCKSIZE];
    int i;
    if ((FP1 = fopen("file1", "r")) == NULL) {
        perror("fopen of file1");
        return FAILURE;
    }
    if ((FP2 = fopen("file2", "w")) == NULL) {
        perror("fopen of file2");
        return FAILURE;
    }
    while ((i = fread(buf, 1, BLOCKSIZE, FP1)) != 0)
        fwrite(buf, 1, i, FP2);
}
```

© 1985 MetaWare Incorporated
freopen — Open a file using an existing FILE variable.

INTERFACE

#include <stdio.h>
FILE *freopen(char *pathname,char *type,FILE *stream);

DESCRIPTION

Closes the file associated with stream, then opens the file named by the string pathname and associates stream with it. The type argument is used as in fopen. Failure to close the file originally associated with stream is ignored. If freopen is successful, it returns stream. If the open fails, it returns NULL.

The rationale behind freopen is that a new file can be associated with a stream without informing all those having copies of the stream value. For example, it provides a way to disassociate stdout from the console and associate it with a chosen file.

SEE ALSO

fopen to open a file.
setbuf to associate a particular buffer with a stream.
close to close a file.

EXAMPLE

The program below fragment first attempts to open file1 for writing. If the open succeeds, it associates a number of file pointers with file1. The subsequent call to freopen attempts to close file1 and open file2 using the same FILE variable. That causes all file pointers previously associated with file1 to reference file2. If the pair of calls fclose/fopen were used in the place of freopen, each file pointer would have to be individually updated.
freopen — Continued.

```c
#include <stdio.h>
#define FAILURE (-1)
main() {
  ...
  FILE *Current_file, *Log_file, *In_file,
      *Out_file, *Err_file;
  if ((Current_file = fopen("file1","w")) == NULL) {
    perror("fopen of file1");
    return FAILURE;
  }
  Log_file = Out_file = Err_file = Current_file;
  ...
  if ((Current_file =
       freopen("file2","w",Current_file) == NULL) {
    perror("freopen of file2");
    return FAILURE;
  }
```

fscanf — Read values from a file.

**INTERFACE**

```c
#include <stdio.h>
int fscanf(FILE *stream, char *format, ...);
```

**DESCRIPTION**

Reads from the file associated with stream according to the format string format, assigning values from the file into objects pointed to by subsequent arguments to fscanf.

format logically consists of an arbitrary sequence of three types of elements: conversion specifications, whitespace characters, and non-whitespace characters. A conversion specification causes characters in the input stream to be read and converted to a specified type, in the manner described below. In most cases there should be a corresponding argument to fscanf that is a pointer to an object of that type. Any whitespace character in format that is not a part of a conversion specification causes input to be
consumed up to the next non-whitespace character. Any non-whitespace character in _format_ that is not a part of a conversion specification must match the next character in the input stream. If such a character does not match the next input character, _fscanf_ returns.

In most cases a conversion specification in _format_ is associated with a subsequent argument to _fscanf_. The correspondence between conversion specifications and arguments is simple: the first conversion specification that expects an argument corresponds to the first argument following _format_, the second conversion specification that expects an argument corresponds to the second argument following _format_, etc. If the number of actual arguments is fewer than the number of arguments specified in _format_, _fscanf_ traipses merrily through memory assuming that whatever it finds is a pointer to the type of object required by the specification in _format_, and assigning into that object. If the number of arguments passed in is greater than the number specified by _format_, the excess arguments are evaluated by the standard function call mechanism but otherwise ignored.

Conversion specifications are of the following form.

'%' ' * ' Field_width? Size? C_char

where

Field_width -> Digits;
Size -> 'h' | 'l' | 'L';
C_char -> 'd' | 'i' | 'o' | 'u' | 'x' | 'X' | 'f' | 'e' | 'E' | 'g' | 'G' | 'c' | 's' | 'p' | 'n' | '%' | '[';

(See the introduction for an explanation of this regular expression notation.)

'%' sets off a conversion specification from ordinary characters.
fscanf — Continued.

'*' causes the conversion of the next value in the input stream to occur as normal, but the assignment of the resulting value is suppressed. All the computation (and concomitant reading of input) is done, but the result is ignored. No argument corresponds to a conversion specification containing '*'.

The input value to be converted generally extends from the next input character to the first character in the input stream that cannot be a part of that value. Except for the 'c' and '[' conversions, leading whitespace is skipped. For example, if format specifies that a decimal integer is to be found in the input stream, first all whitespace is skipped, then all characters that can be part of a decimal integer (all Digits) are read. The conversion ends with the first character that can not be part of a decimal integer (including whitespace characters), and that character remains unread, available as the first character of the next input value. Field_width can be used to impose a maximum on the number of characters that make up an input value.

It is possible for an input value to consist of zero characters for some conversions, e.g. a decimal integer is expected and the next character in the input is 'k'. The C library standard (as defined by ANSI) does not specify what should occur in this case. What does occur is this: an appropriate value (discussed in the description of each such conversion below) is assigned and fscanf goes on to the next element of the format string.

The Size specification is used to specify the size of the receiving object in the case of an argument that points to a type that comes in more than one size, i.e. int versus long int. The Size specification is mentioned in the description below of each conversion that it can affect. If Size is specified for a conversion that it cannot affect, it is ignored.
Finally, the conversion character (C_char) specifies the type of value expected in the input stream (implicitly specifying the type of the object pointed to by the corresponding argument).

The meanings of the conversion characters are:

'd' A decimal integer of the form
    ('+' | '-')? Digits
    is expected. The corresponding argument A should be
    a pointer to int. The 'h' size specification means A is a
    pointer to short int and 'l' means it is a pointer to
    long int. Following the sign (if any), if the next
    character in the input is not a decimal digit, zero is
    assigned to the object pointed to by A.

'i' An integer of the form
    ('+' | '-')? ('0' ('x' | 'X')? )? Hexdigits
    is expected. The corresponding argument A should be
    a pointer to int. Following the optional sign, if the input
    begins with 0x or 0X, the value is taken to be a
    hexadecimal number; otherwise, if the input begins with
    '0', the value is taken to be an octal number; other-
    wise, it is taken to be a decimal number. The 'h' size
    specification means A is a pointer to short int and 'l'
    means it is a pointer to long int. Following the sign (if
    any) and the prefix (0, 0x, or 0X) if present, if the next
    character in the input is not a digit of the appropriate
    base, zero is assigned to the object pointed to by A.

'o' An octal integer of the form
    Odigits
    is expected. The corresponding argument A should be
    a pointer to int. The 'h' size specification means A is a
    pointer to short int and 'l' means it is a pointer to
    long int. If the next character in the input is not an
    octal digit, zero is assigned to the object pointed to by A.
stdio.h

fscanf — Continued.

'u'
An unsigned decimal integer of the form
Digits
is expected. The corresponding argument A should be
a pointer to unsigned int. The 'h' size specification
means A is a pointer to unsigned short int and 'l'
means it is a pointer to unsigned long int. If the next
character in the input is not a decimal digit, zero is
assigned to the object pointed to by A.

'x', 'X'
A hexadecimal integer of the form
Hexdigits
is expected. The corresponding argument A should be
a pointer to int. The 'h' size specification means A is a
pointer to short int and 'l' means it is a pointer to
long int. If the next character in the input is not a
hexadecimal digit, zero is assigned to the object pointed
to by A.

'f', 'e', 'E', 'g', 'G'
A floating point number of the form
Sign? Digits (\'\.\) Digits)? ((\'E\'|\'e\') Sign? Digits)?
is expected. The corresponding argument A should be
a pointer to float. If the 'l' size specification is used,
A should be pointer to double. If the 'L' size
specification is used, A should be a pointer to long
double. If a conflicting character appears in the input
before a floating point value is found, zero is assigned to
A.

'c'
One character is expected. The corresponding
argument A should be a pointer to char. Whitespace is
not skipped in this case. To read the next
non-whitespace character use '%ls'. If a Field_width
is given, Field_width characters are read; A should point
to a character array large enough to hold them. No NUL
is appended.
A character string is expected. The corresponding argument should be a pointer to a character array large enough to accept the string and a NUL that is automatically appended to the result. The string is terminated in the input by any whitespace character. It is advisable to use Field_width with this specification, as otherwise the array may be overrun for some input. Field_width does not include the automatically appended NUL.

A pointer is expected. The corresponding argument should be a pointer to a pointer to void. See section SYSTEM_DEPENDENCIES below.

The argument is a pointer to int into which is written the number of characters read thus far by the current call to fscanf. No input is consumed.

A % is expected. No assignment occurs.

A string that is not delimited by whitespace is expected. The corresponding argument A should be a pointer to a character array. The left bracket is followed by a set of characters, then a right bracket. If the first character in the set is not a circumflex (\^), input characters are assigned to the array pointed to by A as long as they are in the set. If the first character is a circumflex, characters are assigned until a character from the set is encountered. A NUL is appended to the string. It is advisable to use Field_width with this specification, as otherwise the array may be overrun for some input.

If conversion terminates on a conflicting input character, that character is left unread in the input. Trailing whitespace is left unread unless matched in the format string.
fscanf — Continued.

If a conversion specification is invalid, that is, if the conversion character is not one of those listed above, errno is set and that conversion is ignored — no input is read and no argument is assigned into.

If the arguments passed do not match those specified by format, the behavior is undefined.

fscanf returns the number of input values assigned, or EOF if end-of-file is encountered before the first conflict or conversion. If no conflict occurs, fscanf returns when the end of the format string is encountered.

Cautions

If fewer arguments are passed to fscanf than specified by format, the behavior is undefined.

fscanf interprets each argument that follows format as a pointer to the type specified by the corresponding conversion specification. If an argument is not the expected pointer, it is nonetheless treated as if it were, potentially causing some arbitrary place in memory to be overwritten with the result of the conversion. If the incorrect argument is not the same size as a pointer, subsequent arguments may be misinterpreted as well.

System Dependencies

For a '%p' conversion, a pointer is expected. The corresponding argument A should be a pointer to a pointer to void. Four hexadecimal digits are read, which are taken to be the segment value, then a colon is read, then four hex digits, which are taken to be the offset. This address is then assigned into the pointer pointed to by A. This conversion is only guaranteed to work on a value that is output from one of the printf functions’ conversion of a pointer (also '%p') during the same run of a program.
stdio.h

fscanf — Continued.

If `stream` is a text stream, MS-DOS line terminators (`\r\n`) are converted into C terminators (`\n`), and Control Z ('032') is interpreted as an end-of-file indicator. If `stream` is a binary stream, no such conversion occurs, and Control Z is not interpreted specially.

**SEE ALSO**

`fprintf`, `printf`, `sprintf`, `vfprintf`, `vprintf`, `vsprintf`, `fscanf`, `scanf`, `sscanf`.

See the **SEE ALSO** section of `fprintf` for a description of the listed functions.

**EXAMPLE**

```c
#include <stdio.h>
#define FAILURE (-1)
main() {
    FILE *FP;
    char sl[ll], s2[19] = "but missed eleven.";
    int i, j;
    if ((FP = fopen("input", "r")) == NULL) {
        perror("fopen of input");
        return FAILURE;
    }
    fscanf(FP,"%llc%d %%%d %z", sl, &i, &j, s2);
    printf("lls%d, %s%s", sl, i,
        j == 11 ? "11, " : ", "sl, i, s2);
}
```

Given that file **input** contains
scanf read 10 %11 and then quit, z and garbage?
this program prints the following to standard output.
scanf read 10, 11, and then quit.
But if **input** contains
scanf read 10 11 who cares what is out here
(note: not %11), this program prints the following instead.
scanf read 10, but missed eleven.
**fseek** — Seek to somewhere in a file.

**INTERFACE**
```
#include <stdio.h>
int fseek(FILE *stream, long offset, int ptrname);
```

**DESCRIPTION**
Moves the file pointer for the file associated with `stream` to `offset` bytes from one of three positions specified by `ptrname`. If `ptrname` has the value of the macro `SEEK_SET`, the file pointer is moved to `offset` bytes from the beginning of the file. If `ptrname` equals `SEEK_CUR`, the file pointer is moved to `offset` bytes from the current position of the file pointer. If `ptrname` equals `SEEK_END`, the file pointer is moved to `offset` bytes from the end of the file. `offset` may be positive or negative. `fseek` returns zero unless the request is invalid.

A seek past the end of a file or a negative seek past the beginning of a file constitutes a valid request. The results are implementation defined: see `SYSTEM DEPENDENCIES`.

**SYSTEM DEPENDENCIES**
Under MS-DOS, after seeking outside of the limits of a file, whether past the end or negatively past the beginning, a read request reads no characters. After seeking negatively past the beginning of a file, a subsequent write request fails. After seeking past the end of a file, a subsequent write writes at the seek position. From the original end of the file (when the seek occurred) to the seek position (where the write occurred), the content of the file is arbitrary.

**SEE ALSO**
- `ftell` to obtain the value of a file pointer.
- `rewind` to seek to the beginning of a file.
EXAMPLE

The program below copies the second line of file1 to file2, then to file3. This example illustrates the use of fseek and ftell.

#include <stdio.h>
#define FAILURE (-1)
#define BLOCKSIZE (512)

main()
{
  FILE *FPl, *FP2, *FP3;
  char buf[BLOCKSIZE];
  int i;
  long place_mark;

  FILE *open(char *f, char *mode) {
    FILE *FP;
    if ((FP = fopen(f, mode)) == NULL) perror(f);
    return FP;
  }

  if (((FPl = open("file1", "r")) == NULL) ||
      ((FP2 = open("file2", "w")) == NULL) ||
      ((FP3 = open("file3", "w")) == NULL))
    return FAILURE;

  fgets(line, BLOCKSIZE, FPl);
  place_mark = ftell(FPl);
  if (fgets(line, BLOCKSIZE, FPl))
    fputs(line, FP2);
  fseek(FPl, place_mark, SEEK_SET);
  if (fgets(line, BLOCKSIZE, FPl))
    fputs(line, FP3);
}
ftell — Get the current value of a file pointer.

INTERFACE
#include <stdio.h>
long ftell(FILE *stream);

DESCRIPTION
Returns the value of the file pointer for the file associated with stream. The value is a byte offset from the beginning of the file.

SEE ALSO
fseek to move a file pointer to some position within a file.
rewind to seek to the beginning of a file.

EXAMPLE
See the example for fseek.
fwrite — Write to a file.

INTERFACE

#include <stdio.h>
int fwrite(void *ptr, size_t size, int nelem, FILE *stream);

DESCRIPTION

Writes nelem elements of size bytes each to the output stream associated with stream from the segment of memory beginning at the location referenced by ptr. If a write error occurs, fwrite returns having written fewer than nelem elements. The file pointer (if one is defined) is advanced by the number of bytes written. fwrite returns the number of elements actually written. If a write error occurs, the position of the file pointer (if any) is indeterminate.

The first read or write request on a buffered stream causes a buffer to be allocated for the stream unless setbuf has previously been called to associate a buffer with the stream.

CAUTIONS

fwrite writes nelem * size bytes from the segment of memory beginning at the location referenced by ptr.

SYSTEM DEPENDENCIES

If stream is a text stream, C line terminators (\n) are converted into MS-DOS terminators (\r\n). If stream is a binary stream, no such conversion occurs.

SEE ALSO

fread to read from a file.
setbuf to associate a particular buffer with a stream.

EXAMPLE

See the example for fread.
stdio.h

getc — Get a character from a file.
        — Provided both as a macro and a function.

INTERFACE

#include <stdio.h>
int getc(FILE *stream);

DESCRIPTION

Is equivalent to fgetc; see fgetc.
When used as a macro, stream is evaluated more than once.

SEE ALSO

fgetc, getchar, fgets, gets, fputc, putc, putchar, fputs,
puts, ungetc.

For short descriptions of all of the listed functions, see the
SEE ALSO section for fgetc.

EXAMPLE

The program below copies file1 character by character to
file2 and to standard output. This example illustrates the
use of getc and putc. Similar examples are given under the
EXAMPLE sections of fgetc and getchar, pointing out the
differences between the functions.

#include <stdio.h>
define FAILURE (-1)
main() {
    FILE *FP1, *FP2;
    if ((FP1 = fopen("file1", "r")) == NULL) {
        perror("fopen of file1");
        return FAILURE;
    }
    if ((FP2 = fopen("file2", "w")) == NULL) {
        perror("fopen of file2");
        return FAILURE;
    }
    while (feof(FP1) == 0)
        putc(putchar(getc(FP1)), FP2);
}
getchar — Get a character from standard input.
— Provided both as a macro and a function.

**INTERFACE**

    int getchar(void);

**DESCRIPTION**

    Is equivalent to `fgetc(stdin)`; see `fgetc`.

    The macro `getchar()` expands to `getc(stdin)`, which itself
    may be a macro.

**SEE ALSO**

    `fgetc`, `getc`, `fgets`, `gets`, `fputc`, `putc`, `putchar`, `puts`,
    `ungetc`.

    For short descriptions of all of the listed functions, see the
    **SEE ALSO** section for `fgetc`.

**EXAMPLE**

    The program below copies standard input character by
    character to `file1` and to standard output. Similar
    examples are given under the **EXAMPLE** sections of `fgetc` and
    `getc`, pointing out the differences between the functions.

```c
#include <stdio.h>
#undef getchar
#undef putchar
#define FAILURE (-1)

main() {
    FILE *FP1;
    if ((FP1 = fopen("file1", "w")) == NULL) {
        perror("fopen of file1");
        return FAILURE;
    }
    while (feof(stdin) == 0)
        fputc(putchar(getchar()), FP1);
}
```

gets — Get a line of text from standard input.

**INTERFACE**

```c
char *gets(char *s);
```

**DESCRIPTION**

Reads a line of text from the stream `stdin` into the string `s`, appending a final `NUL`. No characters are read after end-of-file or a newline is encountered. The newline, if any, is discarded.

`gets` returns `s` if successful. If end-of-file is encountered before any characters have been read, the array is unchanged and `gets` returns `NULL`. If a read error occurs, the array contents are indeterminate and `gets` returns `NULL`.

**CAUTIONS**

Care should be taken, when `gets` is called, that `s` is large enough to contain all input up to a newline.

**SURPRISES**

`gets` and `fgets` are inconsistent: both functions return after encountering a newline, but `fgets` stores the newline while `gets` discards it.

**SEE ALSO**

`fgets`, `fgetc`, `getchar`, `fgets`, `fputc`, `putc`, `putchar`, `fputs`, `puts`, `ungetc`.

For short descriptions of all of the listed functions, see the **SEE ALSO** section for `fgetc`.

**EXAMPLE**

The program below copies a line from standard input to standard output. It illustrates the use of `gets` and `puts`.

```c
#include <stdio.h>
#define LINESIZE (256)
main() {
    char line[LINESIZE];
    if (gets(line))
        puts(line);
}
```
stdio.h

perror — Print error message.

INTERFACE

char *perror(char *s);

DESCRIPTION

Prints the string $s$, then ": ", then an error message string associated with the current value of $errno$, and finally a newline, on the standard error file. perror returns the error message string. If $s$ is NULL, nothing is printed out.

EXAMPLE

The program fragment below attempts to open file1, and complains if the open fails.

```c
#include <stdio.h>
#define FAILURE (-1)
main() {
    FILE *FP1;
    if ((FP1 = fopen("file1", "w")) == NULL) {
        perror("file1");
        return FAILURE;
    }
}
```
stdio.h

**printf** — Print on standard output.

**INTERFACE**

```c
int printf(char *format, ...);
```

**DESCRIPTION**

Is equivalent to `fprintf(stdout, format, ...);`; see `fprintf`.

**CAUTIONS**

The number of arguments to `printf` must equal or exceed the number specified by `format`, or garbage may be printed.

The types of the arguments to `printf` must match the types specified by `format`, or garbage may be printed.

**SEE ALSO**

`fprintf`, `printf`, `sprintf`, `vfprintf`, `vprintf`, `vsprintf`, `fscanf`, `scanf`, `sscanf`.

See the **SEE ALSO** section of `fprintf` for a description of the listed functions.

**EXAMPLE**

A similar example is given in the **EXAMPLE** section of `fprintf`, pointing out the similarity between the two functions.

```c
#include <stdio.h>
main() {
  char s[14] = "like this one";
  int i = 10, x = 255;
  double d = 3.1415927;
  printf("printf prints strings (%s),\n", s);
  printf("decimal numbers (%d),", i);
  printf(" hex numbers (%04X),\n", x);
  printf("floating-point numbers (%+.4e)\n", d);
  printf("and percent signs (%%).");
}
```

prints the following on standard output:

printf prints strings (like this one),
decimal numbers (10), hex numbers (00FF),
floating-point numbers (+3.1416e+00),
and percent signs (%).
putc

--- Write a character to a file.
--- Provided both as a macro and a function.

**INTERFACE**

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

**DESCRIPTION**

Is equivalent to `fputc`; see `fputc`.

When `putc` is used as a macro, `stream` is evaluated more than once.

**SURPRISES**

`putc` takes `c` as an `int` rather than a `char` parameter and returns an `int` rather than a `char` to conform to the conventions discussed in the Parameter Passing section of the introduction. If an object of type `char` is passed to `putc`, it is automatically coerced to type `int`.

**SYSTEM DEPENDENCIES**

If `stream` is a text stream, a C line terminator (`\n`) is converted into an MS-DOS line terminator (`\r\n`). If `stream` is a binary stream, no such conversion occurs.

**SEE ALSO**

`fgetc`, `getc`, `getchar`, `fgets`, `gets`, `fputc`, `putchar`, `fputs`, `puts`, `ungetc`.

For short descriptions of all of the listed functions, see the `SEE ALSO` section for `fgetc`.

**EXAMPLE**

See the example for `getc`.

---
putchar — Write a character on standard output.
— Provided both as a macro and a function.

INTERFACE
int putchar(int c);

DESCRIPTION
Is equivalent to fputc(c, stdout; see fputc.

The macro putchar(c) expands to putc(c, stdout), which itself may be a macro.

SURPRISES
putchar takes c as an int rather than a char parameter and returns an int rather than a char to conform to the conventions discussed in the Parameter Passing section of the introduction. If an object of type char is passed to putchar, it is automatically coerced to type int.

SYSTEM DEPENDENCIES
If stream is a text stream, a C line terminator (\n) is converted into an MS-DOS line terminator (\r\n). If stream is a binary stream, no such conversion occurs.

SEE ALSO
fgetc, getc, getchar, fgets, gets, fputc, putc, fputs, puts, ungetc.

For short descriptions of all of the listed functions, see the SEE ALSO section for fgetc.

EXAMPLE
See the example for fgetc.
puts — Write a string to a file.

INTERFACE

int puts(char *s);

DESCRIPTION

Writes the string s on the stream associated with stdout, and appends a newline. The terminating NULL is not written. puts returns zero if the operation is successful. If the puts operation fails, errno is set such that a call to perror will result in an error message describing the reason for the failure, and the (non-zero) value of errno is returned.

SURPRISES

puts and fputs are inconsistent: fputs writes exactly the string s, while puts writes s followed by a newline.

SYSTEM DEPENDENCIES

If stream is a text stream, C line terminators (\n) are converted into MS-DOS terminators (\r\n). If stream is a binary stream, no such conversion occurs.

SEE ALSO

fgetc, getc, getchar, fgets, gets, fputc, putc, putchar, fputs, ungetc.

For short descriptions of all of the listed functions, see the SEE ALSO section for fgetc.

EXAMPLE

See the example for gets.
remove — Delete a file.

#include <stdio.h>
int remove(char *pathname);

DESCRIPTION

Removes the file named by the string pathname. remove returns zero if the operation is successful. If the remove operation fails, errno is set such that a call to perror will result in an error message describing the reason for the failure, and the (non-zero) value of errno is returned.

CAUTIONS

If remove is called on an open file, the behavior is implementation defined and the return value may be meaningless; see SYSTEM DEPENDENCIES.

SYSTEM DEPENDENCIES

Under MS-DOS if remove is called on an open file, the file is removed after it has been closed.

SEE ALSO

rename to change the name of a file.
fclose to close a file.

EXAMPLE

The program below opens a temporary file. Later, it closes it and removes it. This example illustrates the use of remove and tmpnam.

#include <stdio.h>
#define FAILURE (-1)
main() {
    FILE *FP1;
    char tmp[L_tmpnam];
    if (FP1 = fopen(tmpnam(tmp), "w+")) == NULL) {
        perror("fopen of temporary file");
        return FAILURE;
    }
    ... 
    fclose(FP1);
}
rename — Change the name of a file.

INTERFACE

    int rename(char *old, char *new);

DESCRIPTION

Changes the name of the file named by the string `old` to the name contained in the string `new`. There will no longer be a file whose name is contained in the string `old`.

rename returns zero if the operation is successful. If the rename operation fails, `errno` is set such that a call to `perror` will result in an error message describing the reason for the failure, and the (non-zero) value of `errno` is returned.

CAUTIONS

If rename is called on an open file, the behavior is implementation defined and the return value may be meaningless; see SYSTEM DEPENDENCIES.

SYSTEM DEPENDENCIES

If an existing file is named by `new`, rename fails.

rename should not be called on an open file. If there is no file named by `new` and rename is called when the file named by `old` is open, after program termination there will be no file `old`, and there will be a file `new` whose contents are what `old` contained before it was opened.

SEE ALSO

remove to delete a file.
rewind — Seek to the beginning of a file.

**INTERFACE**

```c
#include <stdio.h>
void rewind(FILE *stream);
```

**DESCRIPTION**

Is equivalent to `fseek(stream, 0, SEEK_SET)` followed by `clearerr(stream)` except that no value is returned. See `fseek` and `clearerr`.

**SEE ALSO**

`fseek` to move a file pointer to some position within a file.
`ftell` to obtain the value of a file pointer.
`clearerr` to clear end-of-file and error flags of a file.

**EXAMPLE**

The program below copies `file1`, a block at a time, to `file2`, then to `file3`.

```c
#include <stdio.h>
define FAILURE (-1)
define BLOCKSIZE (512)
main() {
    FILE *FP1, *FP2, *FP3;
    char buf[BLOCKSIZE];
    int i;

    FILE *open(char *f, char *mode) {
        FILE *FP;
        if ((FP = fopen(f, mode)) == NULL) perror(f);
        return FP;
    }

    if (!(FP1 = open("file1", "r")) ||
        !(FP2 = open("file2", "w")) ||
        !(FP3 = open("file3", "w"))) return FAILURE;

    while ((i = fread(buf, 1, BLOCKSIZE, FP1)) != 0) 
        fwrite(buf, 1, i, FP2);
    rewind(FP1);
    while ((i = fread(buf, 1, BLOCKSIZE, FP1)) != 0) 
        fwrite(buf, 1, i, FP3);
}
```
scanf — Read values from standard input.

INTERFACE

```c
int scanf(char *format, ...);
```

DESCRIPTION

Is equivalent to `fscanf(stdin, format, ...);` see `fscanf`.

CAUTIONS

If fewer arguments are passed to `scanf` than `format` specifies, arbitrary places in memory are overwritten with the results of the specified conversions.

`scanf` interprets each argument that follows `format` as a pointer to the type specified by the corresponding conversion specification. If an argument is not the expected pointer, it is nonetheless treated as if it were, potentially causing some arbitrary place in memory to be overwritten with the result of the conversion. If the incorrect argument is not the same size as a pointer, subsequent arguments may be misinterpreted as well.

SEE ALSO

`fprintf`, `printf`, `sprintf`, `vfprintf`, `vprintf`, `vsscanf`, `fscanf`, `sscanf`.

See the `SEE ALSO` section of `fprintf` for a description of the listed functions.

EXAMPLE

A similar example is given in the `EXAMPLE` section of `fscanf`, showing the slight difference between the two functions.

```c
#include <stdio.h>

main() {
    char sl[11], s2[19] = "but missed eleven."
    int i, j;
    scanf("%llc%d %%%d [% z]z", sl, &i, &j, s2);
    printf("lls%d, %s%s", sl, i,
             j == 11 ? "11, " : ", s2);
}
```
Given that the input from standard input is `scanf read 10 %11` and then quit.
z and garbage? this program prints the following to standard output.
`scanf read 10, 11, and then quit.`
But if the input is `scanf read 10 11` who cares what is out here
(note: not `%11`), this program prints the following instead.
`scanf read 10, but missed eleven.`
setbuf — Specify a buffer for a stream.

INTERFACE

#include <stdio.h>
void setbuf(FILE *stream, char *buf);

DESCRIPTION

Causes the BUFSIZ byte segment of memory beginning with the location referenced by buf to be used by the stream stream instead of an automatically allocated buffer. If buf is NULL, I/O on the stream is unbuffered.

If buffered I/O has been done on the stream at the time setbuf is called, the contents of the buffer in use are copied into buf, and the buffer in use is deallocated if it was automatically allocated.

CAUTIONS

Library functions that perform I/O assume that the buffer associated with a stream is BUFSIZ bytes in length. If setbuf is called with a buf argument that points to a smaller buffer, the behavior is undefined.

If the buffer that buf references is deallocated before the file is closed, havoc may be wreaked.

If buf points to non-static storage, fclose should be explicitly called on stream before the block containing the declaration of the buffer is exited. Normal program termination involves closing all open files, which in turn involves flushing the buffers of all buffered files that are open for writing or update. If the buffer referenced by buf is not static and the block in which it is declared is exited before fclose is called on stream, the buffer may be overwritten before it is flushed.

SEE ALSO

fopen to open a file.
fclose to close a file.
stdio.h

sprintf — Print on a string.

**INTERFACE**

```c
int sprintf(char *s, char *format, ...);
```

**DESCRIPTION**

Is equivalent to `fprintf(f, format,...)` except that the output is written on the string `s` rather than on the file associated with `f`; see `fprintf`.

**CAUTIONS**

The number of arguments to `sprintf` must equal or exceed the number specified by `format`, or garbage may be printed.

The types of the arguments to `sprintf` must match the types specified by `format`, or garbage may be printed.

**SEE ALSO**

`fprintf`, `printf`, `vfprintf`, `vprintf`, `vsprintf`, `fscanf`, `scanf`, `sscanf`.

See the **SEE ALSO** section of `fprintf` for a description of the listed functions.
A similar example is given in the `EXAMPLE` section of `fprintf`, showing the slight difference between the two functions.

```c
#include <stdio.h>

main() {
    char String[200];
    char *S = String;
    char s[14] = "like this one";
    int i = 10, x = 255, n;
    double d = 3.1415927;
    sprintf(S, "sprintf prints strings (%s),\n", s, &n);
    SP += n; /* The next call to sprintf must be */
    /* passed the address of the NUL */
    /* written by the last call. */
    sprintf(S, "decimal numbers (%d),\n", i, &n);
    SP += n;
    sprintf(S, " hex numbers (%04X),\n", x, &n);
    SP += n;
    sprintf(S, "percent signs (%), and\n", &n);
    SP += n;
    sprintf(S, "floating-point numbers (%.4e).\n", d);
}
```

The program above writes

`sprintf` prints strings (like this one),
decimal numbers (10), hex numbers (00FF),
percent signs (%), and
floating-point numbers (+3.1416e+00).

into `String`. What are here shown as separate lines are
demarcated by the character '\n' in `String`, and there is a
NUL after the final period.
sscanf — Read values from a string.

INTERFACE

int sscanf(char *s, char *format, ...);

DESCRIPTION

Is identical to fscanf(f, format, ...) except that the input is read from the string s rather than from the file associated with f; see fscanf.

CAUTIONS

If fewer arguments are passed to sscanf than specified by format, arbitrary places in memory are overwritten with the results of the specified conversions.

sscanf interprets each argument that follows format as a pointer to the type specified by the corresponding conversion specification. If an argument is not the expected pointer, it is nonetheless treated as if it were, potentially causing some arbitrary place in memory to be overwritten with the result of the conversion. If the incorrect argument is not the same size as a pointer, subsequent arguments may be misinterpreted as well.

SEE ALSO

fprintf, printf, sprintf, vfprintf, vprintf, vsprintf, fscanf, scanf, vfscanf, vscanf, vsscanf.

See the SEE ALSO section of fprintf for a description of the listed functions.

EXAMPLE

A similar example is given in the EXAMPLE section of fscanf, showing the slight difference between the two functions.
`stdio.h`

`sscanf` — Continued.

```c
#include <stdio.h>

main() {
    char s1[11];
    char s2[19] = "but missed eleven."
    char s3[19] = "but missed eleven."
    char input1[39] = "scanf read 10 %11 and quit.za";
    char input2[39] = "it read 9, 10 11 and quit.za";
    int i, j = 2, k = 2;
    fscanf(input1,"%11c%d %%%d [%^ z]z",s1,&i,&j,s2);
    printf("%11s%d, %s%s
", s1, i, j == 11 ? "11, " : ",", s2);
    fscanf(input2,"%11c%d %%%d [%^ z]z",s1,&i &k,s3);
    printf("%11s%d, %s%s
", s1, i, k == 11 ? "11, " : ",", s3);
}
```

prints the following on standard output:

```
scanf read 10, 11, and then quit.
it read 9, 10, but missed eleven.
```

`tmpfile` — Create a temporary file.

**INTERFACE**

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

**DESCRIPTION**

Creates a file and returns a pointer to its controlling `FILE` variable. The file is removed when if is closed or when the program terminates. The file is opened for update. If the file cannot be created, `NULL` is returned.

**CAUTIONS**

If the program terminates abnormally, the file may not be removed.

**SEE ALSO**

tmpnam to generate a temporary file name.
tmpnam — Generate a temporary file name.

INTERFACE

    char *tmpnam(char *s);

DESCRIPTION

Generates a string that can be used as a temporary file name. `tmpnam` generates a different file name each time it is called. Aside from being names that are unlikely to clash with file names generated by humans, there is nothing temporary about the files — they must still be opened, flushed, removed, etc., by standard library calls.

If `s` is not `NULL`, `tmpnam` writes on and returns the string `s`. If `s` is `NULL`, `tmpnam` writes the file name in an internal static string and returns it. Subsequent calls to `tmpnam` overwrite that string.

The macro `TMP_MAX` specifies the minimum number of names `tmpnam` must generate before repeating any.

CAUTIONS

If `s` is not `NULL`, `tmpnam` writes on whatever it points to, up to the number of bytes specified by `L_tmpnam`.

SYSTEM DEPENDENCIES

Under MS-DOS, `tmpnam` does not generate unique names across program executions. Two runs of a program that calls `tmpnam` generate identical file names.

SEE ALSO

rename to rename a file.

tmpfile to create and open a temporary file.

EXAMPLE

See the example for remove.
ungetc — Push character back into an input stream.

INTERFACE

```
#include <stdio.h>
int ungetc(int c, FILE *stream);
```

DESCRIPTION

Pushes the character \texttt{c} (the least significant byte of the \texttt{int} \texttt{c}) back into the input stream \texttt{stream}. The character is returned by the next read on \texttt{stream}, assuming that \texttt{fseek} (or \texttt{rewind}) is not called first. \texttt{fseek} and \texttt{rewind} erase all memory of pushed-back characters.

\texttt{ungetc} does not change the external storage associated with \texttt{stream}. \texttt{ungetc} works on buffered streams, pushing the character back into the buffer. The effect of calling \texttt{ungetc} on an unbuffered stream is to cause the stream to become buffered.

\texttt{ungetc} uses whatever space is available in the buffer to store pushed-back characters. The maximum number of characters that can be successfully pushed back is \texttt{BUFSIZ} (if the buffer was empty), the minimum is none (if the buffer was full). If any characters have been read or \texttt{fseek} or \texttt{rewind} has been called since a call to \texttt{ungetc}, at least one character can be successfully pushed back.

\texttt{ungetc} returns the character pushed if successful, or \texttt{EOF} if it could not push the character back into the stream.

SURPRISES

\texttt{ungetc} takes \texttt{c} as an \texttt{int} rather than a \texttt{char} parameter and returns an \texttt{int} rather than a \texttt{char} to conform to the conventions discussed in the Parameter Passing section of the introduction. If an object of type \texttt{char} is passed to \texttt{ungetc}, it is automatically coerced to type \texttt{int}.
ungetc — Continued.

CAUTIONS

Calling `ungetc` on an unbuffered file causes the file to become buffered.

SEE ALSO

`fgetc`, `getc`, `getchar`, `fgets`, `gets`, `fputc`, `putc`, `putchar`, `fputs`, `puts`.

For short descriptions of all of the listed functions, see the SEE ALSO section for `fgetc`.

EXAMPLE

The program below copies `file1` to `file2`, removing occurrences of "ab".

```c
#include <stdio.h>
define FAILURE (-1)
main() {
    FILE *FP1, *FP2;
    int c;

    FILE *open(char *f, char *mode) {
        FILE *FP;
        if ((FP = fopen(f, mode)) == NULL) perror(f);
        return FP;
    }

    if (!((FP1 = open("file1","r")) == NULL) ||
        (FP2 = open("file2","w")) == NULL))
        return FAILURE;

    while (!feof(FP1)) {
        if ((c = fgetc(FP1)) != 'a')
            putc(c, FP2);
        else /* c == 'a' */
            if ((c = fgetc(FP1)) != 'b') {
                ungetc(c, FP1);
                putc('a', FP2);
            } } }
```
vfprintf — Print on a file, using the variable argument macros.

INTERFACE

#include <stdio.h>
int vfprintf(FILE *stream, char *format, va_list arg);

DESCRIPTION

Is equivalent to fprintf(stream, format, arg) where arg has been initialized by the macro va_start (and possibly subsequent va_arg calls). See fprintf in this section on the effect of vfprintf and Section stdarg.h on how to use the variable argument macros.

SEE ALSO

fprintf, printf, sprintf, vprintf, vsprintf, fscanf, scanf, sscanf.

See the SEE ALSO section of fprintf for a description of the listed functions.

vprintf — Print on standard output, using the variable argument macros.

INTERFACE

int vprintf(char *format, va_list arg);

DESCRIPTION

Is equivalent to printf(format, arg) where arg has been initialized by the macro va_start (and possibly subsequent va_arg calls). See fprintf in this section on the effect of vprintf and Section stdarg.h on how to use the variable argument macros.

SEE ALSO

fprintf, printf, sprintf, vprintf, vsprintf, fscanf, scanf, sscanf.

See the SEE ALSO section of fprintf for a description of the listed functions.
vsprintf — Print on a string, using the variable argument macros.

**INTERFACE**

```c
int vsprintf(char *s, char *format, va_list arg);
```

**DESCRIPTION**

Is equivalent to `sprintf(s, format, arg)` where `arg` has been initialized by the macro `va_start` (and possibly subsequent `va_arg` calls). See `sprintf` in this section on the effect of `vsprintf` and Section `stdarg.h` on how to use the variable argument macros.

**SEE ALSO**

`fprintf`, `printf`, `sprintf`, `vfprintf`, `vprintf`, `fscanf`, `scanf`, `sscanf`.

See the **SEE ALSO** section of `fprintf` for a description of the listed functions.

__setmode — Make a stream into a text stream or a binary stream, or set the default for files opened without such a specification.

**INTERFACE**

```c
#include <stdio.h>
void _setmode(FILE *stream, int mode);
```

**DESCRIPTION**

Some systems make a distinction between text streams and binary streams. On such systems, character sequences in text streams may be converted to other sequences on input or output, while for binary streams, I/O performs no conversions. For information on the conversions performed on text streams, see the beginning of this section, under System Dependencies.
setmode — Continued.

_setmode has two distinct functions. It can be used to change the text/binary mode of stream, or it can be used to set the variable fmode. fmode is used to designate a stream as text or binary if no explicit designation is given when the stream is created, that is, when a file is opened. See fopen for how to explicitly designate the text/binary mode of a stream when it is created.

Several macros are provided as valid values for mode.

If stream is not NULL, it is assumed to point to a stream, which becomes a text stream if mode is _TEXT, or becomes a binary stream if mode is _BINARY.

If stream is NULL, the variable fmode is set to mode. If fopen or freopen is called without specifying that the stream being returned is text or binary, the text/binary mode of the stream is set according to the value of fmode. Changing the value of fmode has no effect on existing streams. The text/binary mode of the streams opened automatically at program startup (stdin, stdout, and stderr) can be specified by linking fmode in with an appropriate value. Values for mode and their usage are:

_ALL_FILES_TEXT all streams default to text.
_ALL_FILES_BINARY all streams default to binary.
_USER_FILES_TEXT all streams default to text except those automatically provided at start up (stdin, stdout, and stderr).
_USER_FILES_BINARY all streams default to binary except those automatically provided at start up (stdin, stdout, and stderr).
_STDERR_TEXT stderr defaults to text.
_STDERR_BINARY stderr defaults to binary.
_setmode — Continued.

_STDIN_AND_STDOUT_TEXT stdin, stdout default to text.
_STDIN_AND_STDOUT_BINARY stdin, stdout default to binary.

CAUTIONS

If stream is not NULL but is not a valid FILE pointer, some arbitrary place in memory is written on.

Buffered output streams cannot be successfully changed arbitrarily between text and binary modes. To change the mode of an output stream in midstream, call fflush on the stream prior to calling _setmode on it.

SEE ALSO

fopen to open a file.
freopen to open a file using an existing FILE variable.

EXAMPLE

The program below copies a line from standard input to standard output, without doing any text conversions.

#include <stdio.h>
main() {
    char line[256];
    _setmode(stdin, _BINARY);
    _setmode(stdout, _BINARY);
    if (gets(line))
        puts(line);
}
The header file `stdlib.h` declares a type, defines two macros, and declares a number of functions of general utility. Among them are functions for converting strings to numeric values, for communicating with the host environment, for managing memory, and for generating pseudo-random integers.

**errno.** `stdlib.h` contains a function `strtol` that references the variable `errno`, which is declared in `stdefs.h`. `errno` is set to a positive integer error code by some library functions and system calls when an error occurs during their execution. It is set to zero at program start up. It is never set to zero by any library function. To reference `errno` from a program, either the header file `stdefs.h` must be included, or `errno` must be declared to be external ("extern int errno;").

**onexit_t** — A type used by the function `onexit`.

**INTERFACE**

```
#include <stdlib.h>
```

**DESCRIPTION**

`onexit_t` is the type of both the argument to and the value returned by the function `onexit`.
NULL — A macro used to represent the null pointer.

**INTERFACE**

```c
#include <stdlib.h>
```

**DESCRIPTION**

NULL expands to a value that is assignment-compatible with any pointer type and compares equal with the constant zero. It is therefore suitable as a representation of the null pointer. Note that NULL is not appropriate as the terminating character of a string, as the size of a pointer is not necessarily the same as the size of the character NULL.

NULL is also defined in `stdefs.h` and `stdio.h`.

size_t — A macro used as a type specifier.

**INTERFACE**

```c
#include <stdlib.h>
```

**DESCRIPTION**

size_t expands to the integral type of the result of the sizeof operator.

size_t is also defined in `stdefs.h`, `stdio.h`, and `string.h`. 
stdlib.h

abort  — Terminate a program abnormally.

INTERFACE

    void abort(void);

DESCRIPTION

    Causes the program to terminate. Open streams are not closed and temporary files are not removed. -1 is returned to the calling environment of the program.

SEE ALSO

    exit to terminate a program normally.

EXAMPLE

    The program fragment below aborts the program with a message if all is not okay.

    #include <stdlib.h>
    ...
    if (all != okay) {
        fprintf(stderr, "Too messed up to continue.\n")
        abort();
    }
stdlib.h

atof — ASCII to floating-point.

INTERFACE
#include <stdlib.h>
double atof(char *nptr);

DESCRIPTION
Returns the result of converting a prefix of the string nptr to a floating-point number. Recognizes strings of the form
Whitespace? Sign? Digits ('.' Digits)?
 ('e' | 'E') Sign? Digits)?
(See the introduction for an explanation of this regular expression notation.)
The first character that is not in the above sequence ends the conversion. If the string is empty, if no Digits are found in the sequence, or if the only Digits found follow the 'e' or 'E', atof returns zero. If the value cannot be represented, the result is undefined.

SEE ALSO
atoi for ASCII to int.
atol for ASCII to long int.
strtof for string to double.
strtol for string to long int.

EXAMPLE
The program below computes the area of a circle with diameter 1.0. This example illustrates the use of atof. A similar example is given in the EXAMPLE section of strtod, pointing out the differences between the functions.

#include <stdlib.h>
main() {
    char pi[] = "3.141593";
    char diam[] = "1.0";
    double area;
    double diameter = atof(diam);
    area = (atof(pi)*diameter*diameter)/4.0;
}
stdlib.h

atoi — ASCII to int.

INTERFACE

int atoi(char *nptr);

DESCRIPTION

Converts a prefix of the string nptr to an integer, and returns that value. It recognizes strings of the form

Whitespace? Sign? Digits

(See the introduction for an explanation of this regular expression notation.)

The first character that is not in the above sequence ends the conversion. If the string is empty or no Digits are found in the sequence, atoi returns zero. If the value cannot be represented, the result is undefined.

SEE ALSO

atoi for ASCII to floating-point.
 atol for ASCII to long int.
 strtol for string to double.
 strlen for string to long int.

EXAMPLE

The program below computes the number of miles that would be covered if one went 60 miles per hour for a day. This example illustrates the use of atoi and atol. A similar example is given in the EXAMPLE section of strtol, pointing out the differences between the functions.

#include <stdlib.h>
main() {
#define secs_per_hr (3600)
    char seconds[] = "86400";
    char mph[] = "60";
    long miles;
    miles = atoi(mph)*atol(seconds)/secs_per_hr;
}
stdlib.h

atol — ASCII to long int.

INTERFACE

#include <stdlib.h>
long int atol(char *nptr);

DESCRIPTION

Converts a prefix of the string nptr to an integer, and returns that value. It recognizes strings of the form

Whitespace? Sign? Digits

(See the introduction for an explanation of this regular expression notation.)

The first character that is not in the above sequence ends the conversion. If the string is empty or no Digits are found in the sequence, atol returns zero. If the value cannot be represented, the result is undefined.

SEE ALSO

atoi for ASCII to int.
atof for ASCII to floating-point.
strtof for string to double.
strtol for string to long int.

EXAMPLE

See the example for atoi.
calloc — Dynamically allocate zero-initialized storage for objects.

**INTERFACE**

```c
#include <stdlib.h>
void *calloc(unsigned int nelem, size_t elsize);
```

**DESCRIPTION**

Returns a pointer to the lowest byte of newly allocated space for `nelem` objects of `elsize` bytes each. The space is initialized to all zeros. If the space cannot be allocated, `calloc` returns NULL.

**SEE ALSO**

- `malloc` to dynamically allocate uninitialized storage.
- `realloc` to change the size of previously dynamically allocated storage.
- `free` to free dynamically allocated storage.
exit — Terminate a program normally.

INTERFACE

void exit(int status);

DESCRIPTION

Causes the program to terminate. Any functions registered with onexit are called in reverse order of registration. Subsequently, open streams are closed and temporary files are removed. status is returned to the calling environment of the program.

SEE ALSO

abort to terminate a program abnormally.

onexit to register functions for execution at program termination time.

EXAMPLE

The program fragment below terminates the program if everything is finished.

```c
#include <stdlib.h>
...
if (everything == finished)
  exit(1);
...
```

© 1985 MetaWare Incorporated
stdlib.h

free — Free storage allocated by calloc, malloc, or realloc.

INTERFACE

void free(void *ptr);

DESCRIPTION

Takes a pointer to the lowest byte of dynamically allocated storage — a pointer that was returned by calloc, malloc, or realloc: the storage is made available again for allocation by one of these functions.

CAUTION

If the freed storage is referenced, the resulting behavior is undefined, and quite likely undesirable. Likewise, if the argument to free is not a pointer previously allocated by calloc, malloc, or realloc, the behavior is undefined.

SEE ALSO

malloc to dynamically allocate uninitialized storage.
realloc to change the size of previously dynamically allocated storage.
calloc to dynamically allocate zero-initialized storage.
getenv — Search for an environment variable.

**INTERFACE**

```c
char *getenv(char *name);
```

**DESCRIPTION**

Searches the host environment list for a character sequence of the form "variable=value", where "variable" matches `name`. If a match is made, a pointer to the character sequence "value" is returned, otherwise `NULL` is returned.

**EXAMPLE**

The following program fragment stamps the date (if defined) on the output.

```c
#include <stdlib.h>
char *s;
...
if ((s = getenv("DATE")) != NULL)
    printf("%s\n", s);
...
```

malloc — Dynamically allocate uninitialized storage.

**INTERFACE**

```c
#include <stdlib.h>
void *malloc(size_t size);
```

**DESCRIPTION**

Returns a pointer to the lowest byte of `size` bytes of newly allocated storage. If the space cannot be allocated, `malloc` returns `NULL`.

**SEE ALSO**

- `calloc` to dynamically allocate zero-initialized storage.
- `realloc` to change the size of previously dynamically allocated storage.
- `free` to free dynamically allocated storage.
onexit — Register functions for execution at program termination time.

INTERFACE

#include <stdlib.h>

onexit_t onexit(onexit_t (*func(void));

DESCRIPTION

Registers the function fcn pointed to by func to be called at program termination, in reverse order of registration. fcn takes no parameters and returns a value of type onexit_t. Because it is called after all functions (including main) have returned, fcn must reference only static objects.

There are some constraints that must be observed in order for onexit to function properly, and that cannot be checked by a compiler. There is no limit to the number of functions that may be registered with onexit. However: 1) For each function to be registered, a distinct static variable of type onexit_t must be declared. 2) Within the body of each function to be registered there must be a return statement that returns such a static object. 3) The return value of the call onexit(func) must be assigned into exactly that object that is returned by fcn. 4) No function may be registered more than once. 5) No two registered functions may return the same object. See the example below.

The requirement that fcn return the right value is an integral part of the implementation of onexit, as whatever fcn returns, right or wrong, will be called as a function, and the return value of that will be called, and so on, until the value returned by the first call to onexit is encountered. This allows for an arbitrary number of registered functions without wasting space.
CAUTION

If a function to be registered fails to return the required value (see above), or if the result of the call to `onexit` is not assigned into the appropriate object, undesirable behavior will occur (most likely, a series of arbitrary values will be taken to be pointers to functions, and those “functions” will be called).

If a function is registered more than once, or two registered functions return the same value, undesirable behavior will occur (most likely, the program will go into a loop endlessly calling registered functions).

SEE ALSO

`exit` to terminate a program normally.

EXAMPLE

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

onexit_t f_return;
onexit_t f() {  printf("f was called.\n");
                 return f_return;
               }
onexit_t g_return;
onexit_t g() {  printf("g was called.\n");
                 return g_return;
               }
onexit_t h_return;
onexit_t h() {  printf("h was called.\n");
                 return h_return;
               }

main() {
    f_return = onexit(f);
g_return = onexit(g);
h_return = onexit(h);
    printf("Last statement in main.\n");
    }
```
stdlib.h

prints
Last statement in main.
h was called.
g was called.
f was called.

rand — Pseudo-random number generator.

INTERFACE
int rand(void);

DESCRIPTION
Returns a pseudo-random integer. The sequence computed by rand has a period of $2^{32}$.

ANSI has specified the implementation, adhered to by this library, for rand and srand, as follows:

```c
static unsigned long int next = 1;
int rand() {
    next = next * 1_035_245 + 12_345;
    return (unsigned int)(next/65_536) % 32_768;
}
```

```c
void srand(unsigned int seed) {
    next = seed;
}
```

SEE ALSO
srand to seed the rand function.

EXAMPLE
```c
#include <stdlib.h>
main() {
    if (rand / 2)
        printf("Heads\n")
    else
        printf("Tails\n");
}
stdlib.h

realloc — Reallocate storage allocated by calloc or malloc.

INTERFACE

#include <stdlib.h>
void realloc(void *ptr, size_t size);

DESCRIPTION
Changes the size of the object pointed to by ptr, making it size bytes. ptr must be a pointer previously returned by calloc, malloc, or realloc. The contents of the storage are unchanged up to the smaller of size and the original size of the object. If the space cannot be allocated, realloc returns NULL, and the original object is unchanged. Newly allocated storage (if any) is uninitialized.

SEE ALSO
malloc to dynamically allocate uninitialized storage.
calloc to dynamically allocate zero-initialized storage.
free to free dynamically allocated storage.

srand — Seed the pseudo-random number generator.

INTERFACE
void srand(unsigned int seed);

DESCRIPTION
Uses its argument as a seed for the sequence of pseudo-random numbers generated by rand. If rand is called before any calls to srand, the seed used is 1.

SEE ALSO
rand to generate pseudo-random numbers.
stdlib.h

**strtod** — String to double.

**INTERFACE**

```c
#include <stdlib.h>
double strtod(char *nptr, char **endptr);
```

**DESCRIPTION**

Converts a prefix of the string *nptr* to a floating-point number, and returns that value. A pointer into *nptr*, pointing at the place where the conversion left off, is placed in the pointer referenced by *endptr*. **strtod** recognizes strings of the form

Whitespace? Sign? Digits ('.') Digits)?

(("e" | 'E') Sign? Digits)?

(See the introduction for an explanation of this regular expression notation.)

The first character that is not in the above sequence ends the conversion. If *endptr* is not NULL, a pointer to that character is stored in the object *endptr* points to. If the string is empty, no Digits are found in the sequence, or the only Digits found follow the 'e' or 'E', **strtod** returns zero. If the value cannot be represented, the result is undefined.

**SEE ALSO**

*atoi* for ASCII to int.

*atoi* for ASCII to long int.

*atof* for ASCII to floating-point.

*strtol* for string to long int.
The program below computes the area of circles with various diameters. A similar example is given in the EXAMPLE section of atof, pointing out the differences between the functions.

#include <stdlib.h>

main() {
    #define INPUTS (5)
    char diameters[] = "1.03 67.94 9.2032e27 4 8e-32";
    char *diameter = diameters;
    double pi = 3.141593;
    double diam;
    double area[INPUTS-1];
    int i = 0;
    while (*diameter) { /* while not at end of string */
        diam = strtod(diameter, &diameter);
        area[i++] = pi * diam * diam / 4.0;
    }
}
strconv — String to long.

**INTERFACE**

```
#include <stdlib.h>
long strtol(char *nptr, char **endptr, int base);
```

**DESCRIPTION**

Converts a prefix of the string `nptr` to a long int, and returns that value. A pointer into `nptr`, pointing at the place where the conversion left off, is placed in the pointer referenced by `endptr`. `strtol` recognizes strings of the form

Whitespace? Sign? ('0' ('x' | 'X'))? Digits

(See the introduction for an explanation of this regular expression notation.)

0x or 0X can appear in the string only if `base` is 16 or 0. The first character that is not in the above sequence ends the conversion. If `endptr` is not NULL, a pointer to that character is stored in the object `endptr` points to. If the string is empty or no Digits are found in the sequence, `strtol` returns zero. If the value cannot be represented, the result is undefined.

If `base` is between 2 and 36, it is used as the base for conversion. If `base` is 16, 0x or 0X can prefix the digit string. If `base` is 0, `nptr` itself determines which of 3 bases are used for conversion. Following the sign (if any), a leading 0x or 0X indicates a hexadecimal number: base 16 is used. Otherwise, a leading 0 indicates an octal number: base 8 is used. Otherwise base 10 is used. If `base` > 10 then alphabetic characters are available for use as Digits (as for hexadecimal numbers, carried to the logical limit). For example, in base 17, 16 is expressed as g or G, etc.

If the number is too large in magnitude to be expressed as a long int, `errno` is set to ERANGE and `LONG_MAX` or `LONG_MIN` is returned, depending on the sign of the input.
See also

atoi for ASCII to int.
atol for ASCII to long int.
atof for ASCII to floating-point.
strtol for string to double.

Example

The following program computes the number of miles that would be covered if one went 60 miles per hour for a week, a day, an hour, a minute, and a second. A similar example is given in the example section of atoi, pointing out the differences between the functions.

#include <stdlib.h>

main() {
#define secs_per_hr (3600)
#define INPUTS (5)
    char sec_list[] = "604800 86400 07020 0x3c 1"
    char *second_ptr = sec_list;
    long seconds;
    char mph[] = "60";
    long miles[INPUTS-1];
    int i = 0;
    while (*second_ptr) { /* while not at end of string */
        seconds = strtol(second_ptr, &second_ptr, 0);
        miles[i++] = seconds*atoi(mph)/secs_per_hr;
    }
}
system — Pass a command to the operating system.

**INTERFACE**

```c
int system(char *string);
```

**DESCRIPTION**

Passes the string `string` to be executed as a command by a command processor provided by the host environment. If `string` is NULL, a return value of zero indicates that there is no command processor.

**SYSTEM DEPENDENCIES**

Under MS-DOS, `system` has no effect and always returns zero. It is provided solely in the interest of compatibility with other systems.
The header file *string.h* defines two macros and declares a number of functions for manipulating strings and arbitrary areas of memory. The functions designed to manipulate strings take strings as parameters. The functions designed to manipulate arbitrary areas of memory take (void *) pointers to any type of object as parameters. Many of these functions are provided as macros as well.

**size_t** — A macro used as a type specifier.

**INTERFACE**

```
#include <string.h>
```

**DESCRIPTION**

Expands to the integral type of the result of the `sizeof` operator.

`size_t` is also defined in *stdefs.h*, *stdio.h*, and *stdlib.h*.

**_MAXSTRING** — A macro denoting the maximum length of a string.

**INTERFACE**

```
#include <string.h>
```

**DESCRIPTION**

Expands to the maximum length a string may attain.
string.h

---

memchr — Find a character in an area of memory.

**INTERFACE**

```
#include <string.h>
void *memchr(void *s, int c, size_t n);
```

**DESCRIPTION**

Finds the first occurrence of the character \(c\) (the least significant byte of the `int` \(c\)) in the \(n\) bytes starting at the location referenced by \(s\). A pointer to the character is returned if it is found; NULL is returned if it is not.

**SURPRISES**

`memchr` takes \(c\) as an `int` rather than a `char` parameter to conform to the conventions discussed in the Parameter Passing section of the introduction. If an object of type `char` is passed to `memchr`, it is automatically coerced to type `int`.

**SEE ALSO**

- `strchr` to find the first occurrence of a character in a string.
- `strrchr` to find the last occurrence of a character in a string.
- `strpbrk` to find the first occurrence of any character from one string in another.
- `strcspn` to determine the length of the prefix of a string not containing any characters from another string.
- `strspn` to determine the length of the prefix of a string composed of characters from another string.
The function below returns a pointer to the first instance of the string `s` in the `n` bytes starting at the location referenced by `ptr`.

```
#include <string.h>
char *find_string(char *s, void *ptr, int n) {
    void *start = ptr;
    void *char_one;
    int len = strlen(s);
    #define BYTES_LEFT ((n-len)-(start-ptr))
    do {
        char_one = memchr(start,*s,BYTES_LEFT);
        if (strcmp(s,char_one) == 0)
            return char_one;
        start = char_one + 1;
    } while (char_one != NULL);
    return NULL;
}
```
memcmp — Compare two areas of memory.
— Provided as a macro and as a function.

**INTERFACE**

```
#include <string.h>
int memcmp(void *s1, void *s2, size_t n);
```

**DESCRIPTION**

Compares $n$ bytes starting at the location pointed to by $s1$ to $n$ bytes starting at the location pointed to by $s2$. The bytes are treated as characters — the comparison is lexicographical. If $s1$’s bytes compare greater than $s2$’s, a value greater than zero is returned. If the two areas compare equal, zero is returned. If $s1$’s bytes compare less than $s2$’s, a value less than zero is returned.

**SEE ALSO**

`strcmp` to compare one string to another.

`strncmp` to compare some characters from one string to some characters from another.

**EXAMPLE**

The program below compares `file1` and `file2`.

```
#include <string.h>
#include <stdio.h>
#define EQUAL (1)
#define UNEQUAL (0)
#define FAILURE (-1)
#define TRUE (1)
#define CHUNK (5000)

main() {
    FILE *F1, *F2;
    char first[CHUNK], second[CHUNK];
    char *file1 = first, *file2 = second;
    size_t n, m;
    /* Open the files to be compared. If the files */
    /* cannot be opened, return an error code to the */
    /* calling environment. */
```
if ((F1 = fopen("file1","r")) == NULL) {
    perror("File1");
    return FAILURE;
}
if ((F2 = fopen("file2","r")) == NULL) {
    perror("File2");
    return FAILURE;
}
while (TRUE) {
    /* Try to read a CHUNK of characters from each file. If there are not that many left, fread returns the number it reads. 
    */
    n = fread(file1, 1, CHUNK, F1);
    m = fread(file2, 1, CHUNK, F2);
    if (n != m) {
        printf("Files are not of equal length.");
        return UNEQUAL;
    }
    if (n == 0) {
        /* End of file reached. All comparisons from previous iterations were equal. */
        printf("Files are identical.");
        return EQUAL;
    }
    /* Compare character sequences just read. Return if they are not the same. */
    if (memcmp(file1, file2, n)) {
        printf("Files are not identical.");
        return FAILURE;
    }
}
memcpy — Copy from one place in memory to another.
— Provided as a macro and as a function.

INTERFACE

```
#include <string.h>
void *memcpy(void *s1, void *s2, size_t n);
```

DESCRIPTION

Copies n bytes from the location referenced by s2 to the location referenced by s1. s2 is returned. memcpy copies from left to right (low addresses to high addresses). If s2 < s1 < s2 + n, that is, if the source and destination areas overlap, with the destination area beginning within the source area, use __memcpy.

When used as a macro, s1 is evaluated more than once.

CAUTIONS

memcpy writes on n bytes.

If the source and destination areas overlap such that s2 < s1 < s2 + n, part of the source area (that part from s1 to s2 + n) will have been overwritten by the copy before itself being copied. Use __memcpy to avoid that.

SURPRISES

memcpy copies the second argument to the first.

SEE ALSO

memcpy to copy from one place in memory to another, from left to right (low to high addresses).

__memcpy to copy from one place in memory to another, from right to left (high to low addresses).

strcpy to copy one string onto another, from left to right (low to high addresses).

__strncpy to copy one string onto another, from right to left (high to low addresses).
memcpy — Continued.

strncpy to copy some characters from one string onto another, from left to right (low to high addresses).

rstrncpy to copy some characters from one string onto another, from right to left (high to low addresses).

memset to copy one character into an area of memory.

EXAMPLE

```c
#include <string.h>
#include <stdio.h>
main () {
    char array1[50] =
        "1234567890abcdefgijklmnopqrstuvwxyz";
    char array2[50] =
        "1234567890abcdefgijklmnopqrstuvwxyz"
    char *source1 = &array1[10], *dest1 = array1;
    char *source2 = &array2[10], *dest2 = array2;
    int i;
    memcpy (dest2, source2, 27);
    rmemcpy(dest1, source1, 27);
    for (i = 0; i <=37; i++)
        putchar(array1[i] == 'O'? '@' : array1[i]);
    putchar('
');
    for (i = 0; i <= 37; i++)
        putchar(array2[i] == 'O'? '@' : array2[i]);
}
```

prints

```
abcdefghijklmnopqrstuvwxyz@rstuvwxyz@rstuvwxyz@rstuvwxyz@rstuvwxyz@rstuvwxyz@
```

v.10.15.85 © 1985 MetaWare Incorporated
memset — Duplicate one character across an area of memory.
— Provided as a macro and as a function.

INTERFACE

```c
#include <string.h>
void *memset(void *s, int c, size_t n);
```

DESCRIPTION

Copies the character c (the least significant byte of the int c) into each of n bytes starting at the location pointed to by s. s is returned.

When used as a macro, s is evaluated more than once.

CAUTIONS

memset writes on n bytes.

SURPRISES

memset copies the second argument to the first.

memset takes c as an int rather than a char parameter to conform to the conventions discussed in the Parameter Passing section of the introduction. If an object of type char is passed to memset, it is automatically coerced to type int.

SEE ALSO

memcpy to copy from one place in memory to another.

strcpy to copy the characters from one string into another.

strncpy to copy some characters from one string into another.

EXAMPLE

The function below writes NULS on each character in s1 that is not in s2. This example illustrates the use of memset, strspn, and strcspn.
#include <string.h>

void strip (char *s1, char *s2) {
    int i;
    while (*s1 != 0) { /* not the end of s1, so */
        /* Set s1 to point at the first character from s1 that is not in s2. */
        /* Set i to the length of the prefix of s1 containing no characters from s2. */
        /* Write NUL on each such character. */
        sl += strspn(s1, s2);
        /* Set sl to point at the character after the last NUL just written. */
        sl += i;
    }
}
**string.h**

**strcat**

— Concatenate two strings.
— Provided as a macro and as a function.

**INTERFACE**

```c
#include <string.h>
char *strcat(char *s1, char *s2);
```

**DESCRIPTION**

Appends a copy of the string `s2` to the string `s1`. The first character of `s2` writes over the NULL that ends `s1`. `s1` is returned.

When used as a macro, `s1` and `s2` are evaluated more than once.

**CAUTIONS**

`strcat` writes until the end of string `s2` is encountered, without regard for the size of `s1`.

**SEE ALSO**

`strncat` to append up to `n` characters from one string onto another.

`strncats` to append one string onto another, up to a limit of `n` characters total.

`strcats` to concatenate `n` strings, up to a limit of `m` characters total.

**EXAMPLE**

The program below collects all lines in `file1` that begin with "a", up to a 5000 character limit, into a buffer `lines` and prints the buffer. This example illustrates the use of `strcat` and `strncat`.

```c
v.lO.15.85 © 1985 MetaWare Incorporated
```
string.h

strcat — Continued.

#include <string.h>
#include <stdio.h>
#define LINESIZE (256)
#define OUTPUTLIMIT (5000)
#define TRUE (1)
#define FAILURE (-1)

main() {
    FILE *F1;
    char lines[OUTPUTLIMIT] = "", line[LINESIZE], *s;
    int i = OUTPUTLIMIT, j = 0;
    if ((F1 = fopen("file1","r")) == NULL) {
        perror("File1");
        return FAILURE;
    }
    while (TRUE) {
        /* Read a line and point s at it. */
        s = fgets(line, LINESIZE, F1);
        /* If no line was read, we're done. */
        if (s == NULL) break;
        /* if the line does not start */
        /* with a, read the next line. */
        if (line[0] == 'a')
            /* Append to lines as much of */
            /* the line as will fit. */
            if (i < (j = strlen(line))) {
                strncat(lines, line, i);
                break;
            }
        else {
            strcat(lines, line);
            i -= j;
        }
    }
    /* Print the lines */
    puts(lines);
}
Find the first occurrence of a character in a string.

**INTERFACE**

```c
char *strchr(char *s, int c);
```

**DESCRIPTION**

Finds the first occurrence of the character `c` (the least significant byte of the `int c`) in the string `s`. A pointer to the character is returned if it is found; `NULL` is returned if not. NUL is a valid value for `c` and results in a pointer to the end of the string (to the terminating NUL) being returned.

**SURPRISES**

`strchr` takes `c` as an `int` rather than a `char` parameter to conform to the conventions discussed in the Parameter Passing section of the introduction. If an object of type `char` is passed to `strchr`, it is automatically coerced to type `int`.

**SEE ALSO**

`memchr, strrchr, strpbrk, strcspn, strspn`.

For short descriptions of all of the listed functions, see the `SEE ALSO` section for `memchr`.
The function below returns a pointer to the first instance of the characters that make up `string1`, i.e. `string1` less its terminating NUL, in the string `string2`. This example illustrates the use of `memchr` and `strncmp`.

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

char *find_string(char *string1, void *string2) {
    void *start = string2; 
    void *first_char; 
    int len = strlen(string1); 
    do { 
        first_char = strchr(start,*string1); 
        if (strncmp(s,first_char,len) == 0) 
            return first_char; 
        start = first_char + 1; 
    } while (first_char != NULL); 
    return NULL; 
}
```
### strcmp

- Compare one string to another.
- Provided as a macro and as a function.

**INTERFACE**

```c
#include <string.h>
int strcmp(char *s1, char *s2);
```

**DESCRIPTION**

Compares the string `s1` to the string `s2`. If the first string compares greater than the second, a value greater than zero is returned. If the two strings compare equal, zero is returned. If the first string compares less than the second, a value less than zero is returned. If two strings of unequal length compare equal to the extent of the shorter string, the longer is lexicographically greater, e.g. "abc" compares greater than "ab".

**SEE ALSO**

- `memcmp` to compare two areas of memory.
- `strncmp` to compare some characters from one string with some characters from another.

**EXAMPLE**

The program below compares `file1` and `file2` line by line.

```c
#include <string.h>
#include <stdio.h>
main() {
    #define EQUAL 1
    #define UNEQUAL 0
    #define FAILURE -1
    #define TRUE 1
    #define LINESIZE 256
```
FILE *F1, *F2;
char line1[LINESIZE], line2[LINESIZE], *sl, *s2;
int i = 1;
    /* Open the files to be compared. If the */
    /* files cannot be opened, return an error */
    /* code to the calling environment. */
if ((F1 = fopen("file1","r")) == NULL) {
    perror("file1");
    return FAILURE;
}
if ((F2 = fopen("file2","r")) == NULL) {
    perror("file2");
    return FAILURE;
}
while (TRUE) {
    /* Read a line from each file. */
    sl = fgets(line1, LINESIZE, F1);
    s2 = fgets(line2, LINESIZE, F2);
    /* A NULL return from fgets means end-of- */
    /* file was found. If both strings are */
    /* NULL, the files are equal, if only one */
    /* is NULL, they are not equal. */
if ((sl == NULL) || (s2 == NULL)) {
    if (sl == s2) break;
    printf("Files diverge at line %d.", i);
    return UNEQUAL;
}
/* Compare the strings. */
if (strcmp(line1,line2)) {
    printf("Files diverge at line %d.", i);
    return UNEQUAL;
}
else i++;
}
printf("Files are identical.");
return EQUAL;
strcpy — Copy the characters of one string into another.
— Provided as a macro and as a function.

INTERFACE

#include <string.h>
char *strcpy(char *s1, char *s2);

DESCRIPTION

Copies the characters of string \texttt{s2} into string \texttt{s1}, including the terminating NUL. \texttt{s1} is returned. \texttt{strcpy} copies from left to right: low addresses to high addresses. If \texttt{s2 < s1 < s2 + n}, that is, if the source and destination areas overlap, with the destination area beginning within the source area, use \texttt{rstrcpy}.

When used as a macro, \texttt{s1} and \texttt{s2} are evaluated more than once.

CAUTIONS

\texttt{strcpy} writes until the end of string \texttt{s2} is encountered, without regard for the length of \texttt{s1}.

If the source and destination areas overlap such that \texttt{s2 < s1 < s2 + n}, part of the source area (that part from \texttt{s1} to \texttt{s2 + n}) will have been overwritten by the copy before itself being copied.

SURPRISES

\texttt{strcpy} copies the second argument to the first.

SEE ALSO

\texttt{memcpy}, \texttt{rmemcp}y, \texttt{rstrcpy}, \texttt{strncpy}, \texttt{rstrncpy}, \texttt{memset}.

For short descriptions of all of the listed functions, see the \texttt{SEE ALSO} section for \texttt{memcpy}.
EXAMPLE

The function below returns the last word of the string `line` in the string `word`. This example illustrates the use of `strcpy` and `strrchr`.

```
#include <string.h>
char *last_word (char *line, char *word) {
    char *s;
    /* Find the last space in line. */
    s = strrchr(line, ' ');
    /* If not found, line itself is the last word. */
    if (s == NULL) s = line;
    /* else the last word starts after the space. */
    else s++;
    /* Copy the word into word. */
    return strcpy(word, s);
}
```
**strcspn** — Determine the length of the prefix of a string not containing any characters from another string.

**INTERFACE**

```c
#include <string.h>

size_t strcspn(char *s1, char *s2);
```

**DESCRIPTION**

Returns the length of that segment of the string `s1` that begins at the beginning of the string and is made up entirely of characters that do not occur in the string `s2`. NUL is not considered a part of `s2` for this purpose.

**SEE ALSO**

`memchr`, `strchr`, `strrchr`, `strpbrk`, `strspn`.

For short descriptions of all of the listed functions, see the `SEE ALSO` section for `memchr`.

**EXAMPLE**

See the example for `memset`.
**strlen**

---

Find the length of a string.

---

Provided as a macro and as a function.

**INTERFACE**

```c
#include <string.h>
size_t strlen(char *s);
```

**DESCRIPTION**

Returns the number of characters in the string `s`, not counting the terminating `NULL`.

**EXAMPLE**

The function below copies the individual words of the line `line`, to a buffer, `NULL` terminating each copy. The words are indexed by an array `pword` of pointers. The number of words found is returned. This example illustrates the use of `strlen`, `strncpy`, and `strpbrk`.

```c
#include <string.h>
int *wordify(char *line, char *word, char **pword) {
    char *s = line;
    int i = 0, j;
    /* While not at the end of the string */
    while (s != NULL) {
        pword[i++] = word;
        /* Find the end of a word. */
        s = strpbrk(line, " ");
        /* Find the length of the word. */
        j = (s == NULL ? strlen(line) : s - line);
        /* Copy the word. */
        strncpy(word, line, j);
        word[j] = 0;
        word += j + 1;
        line += j + 1;
    }
    return i - 1;
}
```
strncat — Append characters of one string onto another, up to a limit.

INTERFACE

```
#include <string.h>
char *strncat(char *s1, char *s2, size_t n);
```

DESCRIPTION

Appends up to \( n-1 \) non-NUL characters from the string \( s2 \) to the string \( s1 \). The first character of \( s2 \) writes over the NUL that ends \( s1 \). The result is always NUL terminated.

SEE ALSO

`strncat`, `strcats`, `strcat`.

For short descriptions of all of the listed functions, see the `SEE ALSO` section for `strcat`.

EXAMPLE

See the example for `strcat`.
strncmp — Compare some characters of one string to some characters of another.

**INTERFACE**

```c
#include <string.h>
int strncmp(char *s1, char *s2, size_t n);
```

**DESCRIPTION**

Compares the string `s1` to the string `s2`. No more than `n` characters are compared. If the first `n` characters from the first string (or the entire string, if it contains less than `n` characters) compare greater than the corresponding characters from the second, a value greater than zero is returned. If the two sets of characters compare equal, zero is returned. If the first set of characters compares less than the second, a value less than zero is returned. If two strings of unequal length compare equal to the extent of the shorter string and the length of the shorter string is less than `n`, the longer is lexicographically greater, e.g. "abc" compares greater than "ab".

**SEE ALSO**

- memcmp to compare two areas of memory.
- strcmp to compare one string to another.

**EXAMPLE**

See the example for strchr.
strcpy — Copy a number of characters from one string into another.

INTERFACE

```
#include <string.h>
char *strncpy(char *s1, char *s2, size_t n);
```

DESCRIPTION

Copies exactly \( n \) characters from the string \( s2 \) to the string \( s1 \). If a NUL terminates \( s2 \) before \( n \) characters have been copied, \( s1 \) is NUL padded. If no NUL is encountered before \( n \) characters have been written into \( s1 \), the resulting character array is not NUL terminated. \( s1 \) is returned. \( strncpy \) copies from left to right (high addresses to low addresses). If \( s2 < s1 < s2 + n \), that is, if the source and destination areas overlap, with the destination area beginning within the source area, use \( rstrncpy \).

CAUTIONS

\( strncpy \) writes on \( n \) bytes.

After a call to \( strncpy \), it may not be safe to pass \( s1 \) to functions that expect a string, as it may not be terminated with a NUL.

If the source and destination areas overlap such that \( s2 < s1 < s2 + n \), part of the source area (that part from \( s1 \) to \( s2 + n \)) will have been overwritten by the copy before itself being copied.

SURPRISES

\( strncpy \) copies the second argument to the first.

SEE ALSO

\( memcpy \), \( rmemcpyp \), \( strcpy \), \( rstrncpy \), \( rstrncpy \), \( memset \).

For short descriptions of all of the listed functions, see the \( SEE\ ALSO \) section for \( memcpy \).

EXAMPLE

See the example for \( strlen \).
strpbrk — Find the first occurrence of any character of one string in another.

**INTERFACE**

```c
char *strpbrk(char *s1, char *s2);
```

**DESCRIPTION**

Finds the first occurrence of any character from the string `s2` in the string `s1`. A pointer to the character is returned if it is found; `NULL` is returned if it is not.

**SEE ALSO**

`memchr`, `strchr`, `strrchr`, `strcspn`, `strspn`.

For short descriptions of all of the listed functions, see the **SEE ALSO** section for `memchr`.

**EXAMPLE**

See the example for `strlen`.
**strrchr** — Find the last occurrence of a character in a string.

**INTERFACE**

```c
char *strrchr(char *s, int c);
```

**DESCRIPTION**

Finds the last occurrence of the character `c` (the least significant byte of the int `c`) in the string pointed to by `s`. A pointer to the character is returned if it is found; NULL is returned if it is not. NULL is a valid value for `c` and results in a pointer to the end of the string (pointing to the terminating NULL) being returned.

**SURPRISES**

`strrchr` takes `c` as an `int` rather than a `char` parameter to conform to the conventions discussed in the Parameter Passing section of the introduction. If an object of type `char` is passed to `strrchr`, it is automatically coerced to type `int`.

**SEE ALSO**

`memchr`, `strchr`, `strpbrk`, `strcspn`, `strspn`.

For short descriptions of all of the listed functions, see the **SEE ALSO** section for `memchr`.

**EXAMPLE**

See the example for `strcpy`.
string.h

strspn — Determine the length of the prefix of a string composed entirely of characters from another string.

INTERFACE

```
#include <string.h>
size_t strspn(char *s1, char *s2);
```

DESCRIPTION

Returns the length of that segment of the string `s1` that begins at the beginning of the string and is made up entirely of characters that occur in the string `s2`. NUL is not considered part of `s2` for this purpose.

SEE ALSO

memchr, strchr, strrchr, strpbrk, strcspn.

For short descriptions of all of the listed functions, see the SEE ALSO section for memchr.

EXAMPLE

See the example for memset.
**strtok** — Divide a string into tokens.

**INTERFACE**

```c
char *strtok(char *s1, char *s2);
```

**DESCRIPTION**

Treats the string `s1` as a series of tokens. The tokens are made up of characters not contained in the string `s2`, and are separated by one or more characters that are in `s2`, i.e. `s2` defines the token delimiters.

The first call to `strtok` finds the first token, writes a NUL into `s1` at the end of the token, and returns a pointer to the first character of the token — i.e. a substring consisting of the token is created and returned.

Each subsequent call to `strtok(NULL,s2)` (note that the first argument is the null pointer) causes a pointer to the next NUL terminated token to be returned in the same manner.

The token separator string `s2` may change from call to call.

If `strtok` is called and no token is found, NULL is returned.

**CAUTIONS**

`strtok` writes on `s1`. 

v.10.15.85 © 1985 MetaWare Incorporated
EXAMPLE

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

main()
{
    char sentence[80] = "This fruist b frabucomplete Osentence O by itself.\n";
    char separator[12] = "fruist b";
    printf("%s \", strtok(sentence,separator);
    printf("%s " , strtok(NULL,separator);
    printf("%s " , strtok(NULL,separator);
    printf("%s " , strtok(NULL,separator);
    printf("%s" , strtok(NULL,separator);
    printf("%s", strtok(NULL,"bits of icy leaf\n")
}
```

prints

This is a complete sentence.
_rmemcpy — Copy from one place in memory to another. (Extra-ANSI.)
— Provided as both macro and function.

INTERFACE

#include <string.h>
void * _rmemcpy(void * sl, void * s2, size_t n);

DESCRIPTION

Copies n bytes from the location referenced by s2 to the location referenced by sl. s2 is returned. _rmemcpy copies from right to left (high addresses to low addresses). If s1 < s2 < s1 + n, that is, if the source and destination areas overlap, with the source area beginning within the destination area, use memcpy.

When used as a macro, sl is evaluated more than once.

CAUTIONS

_rmemcpy writes on n bytes.

If the source and destination areas overlap such that s1 < s2 < s1 + n, part of the source area (that part from s2 to s1 + n) will have been overwritten by the copy before being itself copied.

SURPRISES

_rmemcpy copies the second argument to the first.

SEE ALSO

memcpy, strcpy, _rstrcpy, strncpy, _rstrncpy, memset.

For short descriptions of all of the listed functions, see the SEE ALSO section for memcpy.
EXAMPLE

```c
#include <string.h>
#include <stdio.h>
int main () {
    char array1[50] = "abcdefghijklmnopqrstuvwxyz";
    char array2[50] = "abcdefghijklmnopqrstuvwxyz";
    char *source1 = array1, *dest1 = &array1[9];
    char *source2 = array2, *dest2 = &array2[9];
    _rmemcpy(dest1, source1, 27);
    puts(array1);
    _fill_char(&array2[27], 23, 'O'); /* for puts */
    memcpy(dest2, source2, 27);
    puts(array2);
}
```

prints

```
abcdefghi
abcdefghi
```

```
abcdefghi
```
_rstrncpy — Copy one string onto another. (Extra-ANSI.)
— Provided as a macro and as a function.

INTERFACE
#include <string.h>
char *_rstrncpy(char *s1, char *s2);

DESCRIPTION
Copies the string s2 into the string s1, including the NULL that terminates s2. s1 is returned. _rstrncpy copies from right to left (high addresses to low addresses). If s1 < s2 < s1 + n, that is, if the source and destination areas overlap, with the source area beginning within the destination area, use strcpy.

When used as a macro, s1 and s2 are evaluated more than once.

CAUTIONS
_rstrncpy writes until the end of string s2 is encountered, without regard for the length of s1.

If the source and destination areas overlap such that s1 < s2 < s1 + n, part of the source area (that part from s2 to s1 + n) will have been overwritten by the copy before being itself copied.

SURPRISES
_rstrncpy copies the second argument to the first.

SEE ALSO
memcpy, rmemcpy, strcpy, strncpy, rstrncpy, memset.

For short descriptions of all of the listed functions, see the SEE ALSO section for memcpy.
EXAMPLE

#include <string.h>
#include <stdio.h>
main () {
    char string1[50] = "abcdefghijklmnopqrstuvwxyz";
    char string2[50] = "abcdefghijklmnopqrstuvwxyz";
    char *source1 = string1, *dest1 = &string1[9];
    char *source2 = string2, *dest2 = &string2[9];
    _rstrncpy(dest1, source1);
    puts(string1);
    _fill_char(&string2[27],23,'\0'); /* for puts */
    strcpy(dest2, source2);
    puts(string2);
}

prints

abcdefghijklmnopqrstuvwxyz
abcdefghijklmnopqrstuvwx

abcdefg

v.10.15.85 © 1985 MetaWare Incorporated
__rstrncpy — Copy a number of characters from one string into another. (Extra-ANSI.)

INTERFACE

#include <string.h>

char *__rstrncpy(char *sl, char *s2, size_t n);

DESCRIPTION

Copies exactly n characters from the string s2 into sl. If a NUL terminates s2 before n characters have been copied, sl is NUL padded. If no NUL is encountered before n characters have been written into sl, the resulting character array is not NUL terminated. sl is returned. __rstrncpy copies from right to left (low addresses to high addresses). If sl < s2 < sl + n, that is, if the source and destination areas overlap, with the source area beginning within the destination area, use strncpy.

CAUTIONS

__rstrncpy writes on n bytes.

After a call to __rstrncpy, it may not be safe to pass sl to functions that expect a string, as it may not be terminated with a NUL.

If the source and destination areas overlap such that sl < s2 < sl + n, part of the source area (that part from s2 to sl + n) will have been overwritten by the copy before being itself copied.

SURPRISES

__rstrncpy copies the second argument to the first.

SEE ALSO

memcpy, rmemcpyp, strcpy, rstrcpyp, strncpy, memset. For short descriptions of all of the listed functions, see the SEE ALSO section for memcpy.
EXAMPLE

#include <string.h>
#include <stdio.h>

main () {
    char string1[50] = "abcdefghijklmnopqrstuvwxyz";
    char string2[50] = "abcdefghijklmnopqrstuvwxyz";
    char *source1 = string1, *dest1 = &string1[9];
    char *source2 = string2, *dest2 = &string2[9];
    _rstrncpy(dest1, source1, 14);
    puts(string1);
    strncpy(dest2, source2, 14);
    puts(string2);
}

prints

abcdefg hijklm

abcdefg hijklmnxyz

abcdefghijklmnopg

abcdefghi abcdefghijklmnopqrstuvwxyz

abcdefghi abcdefghijklmnopqrstuvwxyz
_strcats — Concatenate strings, up to a limit of n characters. (Extra-ANSI.)

INTERFACE

#include <string.h>
char *strcats(size_t n, char *sl, char *s2, ...);

DESCRIPTION

Concatenates strings s2 and subsequent arguments to the end of string sl, appending up to n characters to sl. The final parameter must be NULL. The resulting string sl has a total of up to n-1 non-NUL characters. The first character of s2 writes over the NUL that ends sl, and similarly for each subsequent argument. The result is always NUL terminated.

SEE ALSO

strncat, _strncat, strcat.

For short descriptions of all of the listed functions, see the SEE ALSO section for strcat.

EXAMPLE

#include <string.h>
#include <stdio.h>

main () {
    char s1[65] = "This ";
    char s2[] = "string ";
    char s3[] = "contains ";
    char s4[] = "fifty ";
    char s5[] = "characters ";
    char s6[] = "more ";
    char s7[] = "or ";
    char s8[] = "less";
    char s9[] = ".\n"
    char s10[] = "More in fact than can be printed."
    _strcats(65,s1,s2,s3,s4,s5,s6,s7,s8,s9,s10,NULL);
    puts(s1);
}

prints

This string contains fifty characters more or less.
More in fact
__strncat — Append characters of one string into another, up to a total limit.

**INTERFACE**

```c
#include <string.h>
char *__strncat(char *s1, char *s2, size_t n);
```

**DESCRIPTION**

Appends characters from the string `s2` to the string `s1`. The resulting string `s1` has a total of up to `n-1` non-NULL characters. The first character of `s2` writes over the NULL that ends `s1`. The result is always NULL terminated.

__strncat__ differs from __strncat__ in that the third argument to __strncat__ is the maximum length for the result `s1`, while the third argument to __strncat__ is the maximum number of characters to be appended to `s1`.

**SEE ALSO**

strncat, __strcats, strcat.

For short descriptions of all of the listed functions, see the **SEE ALSO** section for strcat.

**EXAMPLE**

```c
#include <string.h>
#include <stdio.h>
main () {
    char string1[50] = "This string contains forty"
                     " characters. ";
    char string2[50] = "Plus ten. "
                        "This sentence will be ignored.";
    __strncat(string1, string2, 50);
    puts(string1);
}
```

prints

This string contains forty characters. Plus ten.
The header file *time.h* declares several types and defines a macro, as well as a number of functions for manipulating representations of time.

**clock_t** — A type used to represent a time.

**INTERFACE**

```
#include <time.h>
```

**DESCRIPTION**

An arithmetic type that can represent the time of day.

**time_t** — A type used to represent a date and time.

**INTERFACE**

```
#include <time.h>
```

**DESCRIPTION**

An arithmetic type that can represent time and date.

**tm** — A type used to represent the components of a date and time.

**INTERFACE**

```
#include <time.h>
```

**DESCRIPTION**

A struct type that holds individual components of time and date, such as hour, minute, year, and month. The components may differ from system to system; see *time.h* for details.
CLK_TCK — A macro used to convert from the time units provided by the operating system to seconds.

INTERFACE

#include <time.h>

DESCRIPTION

Expands to the number per second of the smallest time units available from the operating system.

asctime — Convert a tm struct to printable form.

INTERFACE

#include <time.h>
char *asctime(struct tm *timeptr);

DESCRIPTION

Converts the time represented by the contents of the struct referenced by timeptr into a static string of 26 characters, and returns the string. The form of the string is illustrated by

Tue Jun 09 04:23:19 1985

which is terminated with a newline followed by a NUL.

CAUTION

The same static string is overwritten by each call to asctime and ctime.

SEE ALSO

ctime to convert a time_t value to printable form.
asctime — Continued.

EXAMPLE

This example illustrates the use of all of the functions provided in time.h.

```
#include <time.h>
#include <stdio.h>
main () {
    time_t t1, t2;
    long int l;
    l = clock() / CLK_TCK;
    printf("%ld seconds = %ld hours and %ld minutes", 
        l, l/3600, (l%3600)/60);
    printf(" past midnight");
    time(&t1);
    printf("Greenwich time: %s","asctime(gmtime(&t1)))
    printf("Local time : %s",ctime(&t1));
    for (l = 0; l < 500_000; l++) ; /* Waste time. */
    time(&t2);
    printf("New local time: %s",
        asctime(localtime(&t1)));
    printf("Difference between the two local times ");
    printf("(in seconds): %.0f\n", difftime(t2,t1));
}
```

A sample run of the above produced the following output.

46596 seconds = 12 hours and 56 minutes past midnight
Greenwich time: Fri May 17 19:56:36 1985
Local time -- : Fri May 17 12:56:36 1985
New local time: Fri May 17 19:56:55 1985
Difference between the 2 local times (in seconds): 19
clock — Time since midnight.

INTERFACE

\#include <time.h>

clock_t clock(void);

DESCRIPTION

Returns CLK_TCK *, the number of seconds since midnight.

SEE ALSO

time for the time and date.

EXAMPLE

See the example for asctime.

ctime — Convert a time_t value to printable form.

INTERFACE

\#include <time.h>

char *ctime(time_t *timer);

DESCRIPTION

Converts the time represented by timer into a static string of 26 characters, and returns the string. The form of the string is the same as that for asctime.

CAUTION

The same static string is overwritten by each call to ctime and asctime.

SEE ALSO

asctime to convert a tm struct to printable form.

EXAMPLE

See the example for asctime.
difftime — Find the difference between two times.

**INTERFACE**

```
#include <time.h>

double difftime(time_t time2, time_t timel);
```

**DESCRIPTION**

Returns \( \text{time2} - \text{timel} \) expressed in seconds.

**EXAMPLE**

See the example for \texttt{asctime}.


gmtime — Convert a \texttt{time_t} value to a \texttt{tm} struct, adjusted to Greenwich Mean time.

**INTERFACE**

```
#include <time.h>

struct tm *gmtime(time_t *timer);
```

**DESCRIPTION**

Converts the time represented by \texttt{timer} into a struct containing the individual components of the time and date, expressed as Greenwich Mean time. A pointer to the struct is returned.

**CAUTION**

The struct is overwritten by each call to \texttt{gmtime} and \texttt{localtime}.

**SEE ALSO**

\texttt{localtime} to convert a \texttt{time_t} value to a \texttt{tm} struct.

**EXAMPLE**

See the example for \texttt{asctime}.
localtime — Convert a time_t value to a tm struct.

INTERFACE

#include <time.h>
struct tm *localtime(time_t *timer);

DESCRIPTION

Converts the time represented by timer into a struct containing the individual components of the time and date. A pointer to the struct is returned.

CAUTION

The struct is overwritten by each call to localtime and gmtime.

SEE ALSO

gmtime to convert a time_t value to a tm struct, adjusted to Greenwich Mean time.

EXAMPLE

See the example for asctime.

time — What time is it?

INTERFACE

#include <time.h>

DESCRIPTION

Returns the current time and date, to the best of the system’s knowledge and approximation, in a form that is understood by the other functions provided in time.h. If timer is not NULL, the return value is also stored in the object it points to.

SEE ALSO

clock to find out the time since midnight.

EXAMPLE

See the example for asctime.
More Feedback, Please

(After some use.)

We would greatly appreciate your ideas regarding improvement of the language, its compiler, and its documentation. Please take time to jot down your ideas on this page (front and back) and on additional sheets as necessary as you use the software. Then, after you have some significant experience with the software, please mail the results to:

MetaWare™ Incorporated
412 Liberty Street
Santa Cruz, CA 95060

MetaWare may use or distribute any information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use that information. If you wish a reply, please provide your name and address. Thank you in advance, The Authors.

Page Comment
More Feedback, Please

<table>
<thead>
<tr>
<th>Page</th>
<th>Comment</th>
</tr>
</thead>
</table>

© 1983-85 MeaWare Incorporated
abort ........................................ 11-3
abs ........................................ 5-3
acos ........................................ 5-4
ANSI ........................................ 1-8
asctime .................................... 13-2
asin ........................................ 5-4
assert ........................................ 2-2
assert.h ..................................... 2
atan ......................................... 5-5
atan2 ........................................ 5-6
atof ......................................... 11-4
atoi .......................................... 11-5
atol .......................................... 11-6
Binary Streams, Text and .................. 10-3
BUFSIZ ..................................... 10-5
(C Library Manual) Organization .......... 1-1
calloc ....................................... 11-7
ceil ......................................... 5-7
Characteristics of Integral Types ...... 4-2
Characteristics of Floating-Point Types .. 4-3
CHAR_BIT .................................... 4-2
CHAR_MAX ................................... 4-2
CHAR_MIN ................................... 4-2
clearerr ..................................... 10-9
CLK_TCK .................................... 13-2
clock .......................................... 13-4
clock_t ....................................... 13-1
Contents ..................................... 1-3
Conventions, Naming ...................... 1-8
cos .......................................... 5-7
cosh .......................................... 5-7
ctime ........................................ 13-4
ctype.h ....................................... 3
DBL_DIG ..................................... 4-3
DBL_MAX_EXP .................................. 4-3
DBL_MIN_EXP .................................. 4-3
DBL_RADIX ................................... 4-3
DBL_ROUNDS .................................. 4-3
Dependencies, System (I/O, stdio.h) ... 10-4
(math.h) ..................................... 5-1
difftime ..................................... 13-5
Digit ......................................... 1-8, 1-11
Digits ........................................ 1-11
EDOM ......................................... 5-2
End of File .................................. 10-2
EOF .......................................... 10-5
ERANGE ...................................... 5-2
Errors (I/O) .................................. 10-2
erro .......................................... 1-9, 9-1, 9-1, 10-2, 10-3, 11-1
erro may be set by the following functions:
acos, asin, atan2, cosh, exp, ldexp, log,
log10, pow, sinh, sqrt, tanh, strtol, fscanf,
fflush, fprintf, fputs, fscanf, printf, puts,
remove, rename, scanf, sprintf, sscanf,
vfprintf, vprintf, vsprintf.
<table>
<thead>
<tr>
<th>Function</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>exit</td>
<td>11-8</td>
</tr>
<tr>
<td>exp</td>
<td>5-8</td>
</tr>
<tr>
<td>Expressions, Regular</td>
<td>1-10</td>
</tr>
<tr>
<td>fabs</td>
<td>5-8</td>
</tr>
<tr>
<td>fclose</td>
<td>10-9</td>
</tr>
<tr>
<td>feof</td>
<td>10-11</td>
</tr>
<tr>
<td>ferror</td>
<td>10-11</td>
</tr>
<tr>
<td>fflush</td>
<td>10-12</td>
</tr>
<tr>
<td>fgetc</td>
<td>10-13</td>
</tr>
<tr>
<td>fgets</td>
<td>10-15</td>
</tr>
<tr>
<td>FILE</td>
<td>10-5</td>
</tr>
<tr>
<td>File, End of</td>
<td>10-2</td>
</tr>
<tr>
<td>File List, Header</td>
<td>1-3</td>
</tr>
<tr>
<td>Files, Including Header</td>
<td>1-4</td>
</tr>
<tr>
<td>files</td>
<td>10-1</td>
</tr>
<tr>
<td>Floating-Point Types, Characteristics of</td>
<td>4-3</td>
</tr>
<tr>
<td>floor</td>
<td>5-8</td>
</tr>
<tr>
<td>FLT_DIG</td>
<td>4-4</td>
</tr>
<tr>
<td>FLT_MAX_EXP</td>
<td>4-3</td>
</tr>
<tr>
<td>FLT_MIN_EXP</td>
<td>4-3</td>
</tr>
<tr>
<td>FLT_RADIX</td>
<td>4-3</td>
</tr>
<tr>
<td>FLT_ROUNDS</td>
<td>4-3</td>
</tr>
<tr>
<td>fmod</td>
<td>5-9</td>
</tr>
<tr>
<td>fopen</td>
<td>10-17</td>
</tr>
<tr>
<td>fprintf</td>
<td>10-19</td>
</tr>
<tr>
<td>fputc</td>
<td>10-28</td>
</tr>
<tr>
<td>fputs</td>
<td>10-29</td>
</tr>
<tr>
<td>fread</td>
<td>10-30</td>
</tr>
<tr>
<td>free</td>
<td>11-9</td>
</tr>
<tr>
<td>freopen</td>
<td>10-31</td>
</tr>
<tr>
<td>frexp</td>
<td>5-10</td>
</tr>
<tr>
<td>fscanf</td>
<td>10-33</td>
</tr>
<tr>
<td>fseek</td>
<td>10-41</td>
</tr>
<tr>
<td>ftell</td>
<td>10-43</td>
</tr>
<tr>
<td>fwrite</td>
<td>10-44</td>
</tr>
<tr>
<td>getc</td>
<td>10-45</td>
</tr>
<tr>
<td>getchar</td>
<td>10-46</td>
</tr>
<tr>
<td>getenv</td>
<td>11-10</td>
</tr>
<tr>
<td>gets</td>
<td>10-47</td>
</tr>
<tr>
<td>gmtime</td>
<td>13-5</td>
</tr>
<tr>
<td>Header File List</td>
<td>1-3</td>
</tr>
<tr>
<td>Header Files, Including</td>
<td>1-4</td>
</tr>
<tr>
<td>Hexdigit(s)</td>
<td>1-9, 1-11</td>
</tr>
<tr>
<td>Hexletter</td>
<td>1-11</td>
</tr>
<tr>
<td>HUGE_VAL</td>
<td>5-3</td>
</tr>
<tr>
<td>Including Header Files</td>
<td>1-4</td>
</tr>
<tr>
<td>Index</td>
<td>here</td>
</tr>
<tr>
<td>Integral Types, Characteristics of</td>
<td>4-2</td>
</tr>
<tr>
<td>Infinity</td>
<td>5-1</td>
</tr>
<tr>
<td>Introduction</td>
<td>1-1</td>
</tr>
<tr>
<td>INT_MAX</td>
<td>4-2</td>
</tr>
<tr>
<td>INT_MIN</td>
<td>4-2</td>
</tr>
<tr>
<td>isalnum</td>
<td>3-1</td>
</tr>
<tr>
<td>isalpha</td>
<td>3-1</td>
</tr>
<tr>
<td>iscntrl</td>
<td>3-2</td>
</tr>
<tr>
<td>Function</td>
<td>Page(s)</td>
</tr>
<tr>
<td>---------------</td>
<td>---------</td>
</tr>
<tr>
<td>isdigit</td>
<td>3-2</td>
</tr>
<tr>
<td>isgraph</td>
<td>3-2</td>
</tr>
<tr>
<td>islower</td>
<td>3-3</td>
</tr>
<tr>
<td>isprint</td>
<td>3-3</td>
</tr>
<tr>
<td>ispunct</td>
<td>3-3</td>
</tr>
<tr>
<td>isspace</td>
<td>3-4</td>
</tr>
<tr>
<td>isupper</td>
<td>3-4</td>
</tr>
<tr>
<td>isxdigit</td>
<td>3-4</td>
</tr>
<tr>
<td>(I/O) Errors</td>
<td>10-2</td>
</tr>
<tr>
<td>(I/O, stdio.h) System Dependencies</td>
<td>10-4</td>
</tr>
<tr>
<td>jmp buf</td>
<td>6-1</td>
</tr>
<tr>
<td>kill</td>
<td>7-4</td>
</tr>
<tr>
<td>LDBL_DIG</td>
<td>4-4</td>
</tr>
<tr>
<td>LDBL_MAX_EXP</td>
<td>4-4</td>
</tr>
<tr>
<td>LDBL_MIN_EXP</td>
<td>4-4</td>
</tr>
<tr>
<td>LDBL_RADIX</td>
<td>4-3</td>
</tr>
<tr>
<td>LDBL_ROUNDS</td>
<td>4-3</td>
</tr>
<tr>
<td>ldexp</td>
<td>5-11</td>
</tr>
<tr>
<td>Library Manual) Organization, (C</td>
<td>1-1</td>
</tr>
<tr>
<td>limits.h</td>
<td>4</td>
</tr>
<tr>
<td>List, Header File</td>
<td>1-3</td>
</tr>
<tr>
<td>localtime</td>
<td>13-6</td>
</tr>
<tr>
<td>log</td>
<td>5-11</td>
</tr>
<tr>
<td>log10</td>
<td>5-12</td>
</tr>
<tr>
<td>longjmp</td>
<td>6-1</td>
</tr>
<tr>
<td>LONG_MAX</td>
<td>4-2</td>
</tr>
<tr>
<td>LONG_MIN</td>
<td>4-2</td>
</tr>
<tr>
<td>L_tmpnam</td>
<td>10-5</td>
</tr>
<tr>
<td>Macros</td>
<td>1-6</td>
</tr>
<tr>
<td>malloc</td>
<td>11-10</td>
</tr>
<tr>
<td>Manual) Organization, (C Library</td>
<td>1-1</td>
</tr>
<tr>
<td>math.h</td>
<td>5</td>
</tr>
<tr>
<td>(math.h) System Dependencies</td>
<td>5-1</td>
</tr>
<tr>
<td>memchr</td>
<td>12-2</td>
</tr>
<tr>
<td>memcmp</td>
<td>12-4</td>
</tr>
<tr>
<td>memcmp</td>
<td>12-6</td>
</tr>
<tr>
<td>memset</td>
<td>12-8</td>
</tr>
<tr>
<td>modf</td>
<td>5-12</td>
</tr>
<tr>
<td>Naming Conventions</td>
<td>1-8</td>
</tr>
<tr>
<td>NAN</td>
<td>5-1</td>
</tr>
<tr>
<td>NDEBUG</td>
<td>2-1</td>
</tr>
<tr>
<td>NULL</td>
<td>1-8</td>
</tr>
<tr>
<td>NULL</td>
<td>1-9, 9-1, 10-6, 11-2</td>
</tr>
<tr>
<td>null pointer</td>
<td>1-8</td>
</tr>
<tr>
<td>Odigit(s)</td>
<td>1-8, 1-11</td>
</tr>
<tr>
<td>onexit</td>
<td>11-11</td>
</tr>
<tr>
<td>onexit_t</td>
<td>11-1</td>
</tr>
<tr>
<td>Organization, (C Library Manual)</td>
<td>1-1</td>
</tr>
<tr>
<td>Parameter Passing</td>
<td>1-5</td>
</tr>
<tr>
<td>Passing, Parameter</td>
<td>1-5</td>
</tr>
<tr>
<td>perror</td>
<td>10-48</td>
</tr>
<tr>
<td>pow</td>
<td>5-13</td>
</tr>
<tr>
<td>printf</td>
<td>10-49</td>
</tr>
<tr>
<td>ptrdiff_t</td>
<td>9-2</td>
</tr>
<tr>
<td>putc</td>
<td>10-50</td>
</tr>
</tbody>
</table>
Index: High C™ Library Reference  

<table>
<thead>
<tr>
<th>Function</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>putchar</td>
<td>10-51</td>
</tr>
<tr>
<td>puts</td>
<td>10-52</td>
</tr>
<tr>
<td>rand</td>
<td>11-13</td>
</tr>
<tr>
<td>realloc</td>
<td>11-14</td>
</tr>
<tr>
<td>Regular Expressions</td>
<td>1-10</td>
</tr>
<tr>
<td>remove</td>
<td>10-53</td>
</tr>
<tr>
<td>rename</td>
<td>10-54</td>
</tr>
<tr>
<td>rewind</td>
<td>10-55</td>
</tr>
<tr>
<td>scanf</td>
<td>10-56</td>
</tr>
<tr>
<td>SCHAR_MAX</td>
<td>4-2</td>
</tr>
<tr>
<td>SCHAR_MIN</td>
<td>4-2</td>
</tr>
<tr>
<td>SEEK_CUR</td>
<td>10-6</td>
</tr>
<tr>
<td>SEEK_END</td>
<td>10-6</td>
</tr>
<tr>
<td>SEEK_SET</td>
<td>10-6</td>
</tr>
<tr>
<td>setbuf</td>
<td>10-58</td>
</tr>
<tr>
<td>setjmp</td>
<td>6-3</td>
</tr>
<tr>
<td>setjmp.h</td>
<td>6</td>
</tr>
<tr>
<td>SHRT_MAX</td>
<td>4-2</td>
</tr>
<tr>
<td>SHRT_MIN</td>
<td>4-2</td>
</tr>
<tr>
<td>SIGABRT</td>
<td>7-1</td>
</tr>
<tr>
<td>SIGFPE</td>
<td>7-1</td>
</tr>
<tr>
<td>SIGILL</td>
<td>7-2</td>
</tr>
<tr>
<td>SIGINT</td>
<td>7-2</td>
</tr>
<tr>
<td>Sign</td>
<td>1-8</td>
</tr>
<tr>
<td>signal</td>
<td>7-5</td>
</tr>
<tr>
<td>signal.h</td>
<td>7</td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>7-2</td>
</tr>
<tr>
<td>SIGTERM</td>
<td>7-3</td>
</tr>
<tr>
<td>SIG_DFL</td>
<td>7-3</td>
</tr>
<tr>
<td>SIG_ERR</td>
<td>7-3</td>
</tr>
<tr>
<td>SIG_IGN</td>
<td>7-4</td>
</tr>
<tr>
<td>sin</td>
<td>5-13</td>
</tr>
<tr>
<td>sinh</td>
<td>5-13</td>
</tr>
<tr>
<td>size_t</td>
<td>1-9, 9-2, 10-7, 11-2, 12-1</td>
</tr>
<tr>
<td>sprintf</td>
<td>10-59</td>
</tr>
<tr>
<td>sqrt</td>
<td>5-14</td>
</tr>
<tr>
<td>srand</td>
<td>11-14</td>
</tr>
<tr>
<td>sscanf</td>
<td>10-61</td>
</tr>
<tr>
<td>stdarg.h</td>
<td>8</td>
</tr>
<tr>
<td>stddef.h</td>
<td>9</td>
</tr>
<tr>
<td>stderr</td>
<td>10-7</td>
</tr>
<tr>
<td>stdin</td>
<td>10-7</td>
</tr>
<tr>
<td>stdio.h</td>
<td>10</td>
</tr>
<tr>
<td>stdlib.h</td>
<td>11</td>
</tr>
<tr>
<td>stdio.h) System Dependencies, (I/O,</td>
<td>10-4</td>
</tr>
<tr>
<td>stdout</td>
<td>10-7</td>
</tr>
<tr>
<td>strcat</td>
<td>12-10</td>
</tr>
<tr>
<td>strchr</td>
<td>12-12</td>
</tr>
<tr>
<td>strcmp</td>
<td>12-14</td>
</tr>
<tr>
<td>strcpy</td>
<td>12-16</td>
</tr>
<tr>
<td>strcspn</td>
<td>12-18</td>
</tr>
<tr>
<td>Streams</td>
<td>10-1</td>
</tr>
<tr>
<td>Streams, Text and Binary</td>
<td>10-3</td>
</tr>
<tr>
<td>string</td>
<td>1-8</td>
</tr>
<tr>
<td>string.h</td>
<td>12</td>
</tr>
</tbody>
</table>
Index: High C™ Library Reference

<table>
<thead>
<tr>
<th>Function</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>strlen</td>
<td>12-19</td>
</tr>
<tr>
<td>strncat</td>
<td>12-20</td>
</tr>
<tr>
<td>strncmp</td>
<td>12-21</td>
</tr>
<tr>
<td>strncpy</td>
<td>12-22</td>
</tr>
<tr>
<td>strpbrk</td>
<td>12-23</td>
</tr>
<tr>
<td>strchr</td>
<td>12-24</td>
</tr>
<tr>
<td>strspn</td>
<td>12-25</td>
</tr>
<tr>
<td>strtok</td>
<td>12-26</td>
</tr>
<tr>
<td>strftime</td>
<td>11-17</td>
</tr>
<tr>
<td>system</td>
<td>11-18</td>
</tr>
<tr>
<td>System Dependencies</td>
<td>10-3</td>
</tr>
<tr>
<td>SYS_OPEN</td>
<td>10-8</td>
</tr>
<tr>
<td>tan</td>
<td>5-14</td>
</tr>
<tr>
<td>tanh</td>
<td>5-14</td>
</tr>
<tr>
<td>Terms</td>
<td>1-8</td>
</tr>
<tr>
<td>Text and Binary Streams</td>
<td>10-3</td>
</tr>
<tr>
<td>time</td>
<td>13-6</td>
</tr>
<tr>
<td>time.h</td>
<td>13</td>
</tr>
<tr>
<td>time_t</td>
<td>13-1</td>
</tr>
<tr>
<td>tm</td>
<td>13-1</td>
</tr>
<tr>
<td>tmpfile</td>
<td>10-62</td>
</tr>
<tr>
<td>tmpnam</td>
<td>10-63</td>
</tr>
<tr>
<td>TMP_MAX</td>
<td>10-8</td>
</tr>
<tr>
<td>tolower</td>
<td>3-5</td>
</tr>
<tr>
<td>toupper</td>
<td>3-5</td>
</tr>
<tr>
<td>Types, Characteristics of Integral</td>
<td>4-2</td>
</tr>
<tr>
<td>Types, Characteristics of Floating-Point</td>
<td>4-3</td>
</tr>
<tr>
<td>UCHAR_MAX</td>
<td>4-2</td>
</tr>
<tr>
<td>UINT_MAX</td>
<td>4-2</td>
</tr>
<tr>
<td>ULONG_MAX</td>
<td>4-2</td>
</tr>
<tr>
<td>ungetc</td>
<td>10-64</td>
</tr>
<tr>
<td>USHRT_MAX</td>
<td>4-2</td>
</tr>
<tr>
<td>va_arg</td>
<td>8-3</td>
</tr>
<tr>
<td>va_end</td>
<td>8-5</td>
</tr>
<tr>
<td>va_list</td>
<td>1-8</td>
</tr>
<tr>
<td>va_start</td>
<td>8-2</td>
</tr>
<tr>
<td>vfprintf</td>
<td>10-65</td>
</tr>
<tr>
<td>vprintf</td>
<td>10-66</td>
</tr>
<tr>
<td>vsprintf</td>
<td>10-67</td>
</tr>
<tr>
<td>whitespace</td>
<td>1-8</td>
</tr>
<tr>
<td>_MAXSTRING</td>
<td>12-1</td>
</tr>
<tr>
<td>_memcmp</td>
<td>12-28</td>
</tr>
<tr>
<td>_rstrcpyp</td>
<td>12-29</td>
</tr>
<tr>
<td>_rstrncpy</td>
<td>12-30</td>
</tr>
<tr>
<td>_setmode</td>
<td>10-67</td>
</tr>
<tr>
<td>_strcats</td>
<td>12-34</td>
</tr>
<tr>
<td>_strncat</td>
<td>12-35</td>
</tr>
</tbody>
</table>
Acknowledgments

The authors of these manuals and designers of the High C language would like to thank the C standards committee, whose drafts of the C standard helped illuminate many dark areas of the language and assisted greatly in “chunking” the language concepts.

Paul Redmond’s feedback was invaluable as he put dBase III through High C for Ashton-Tate. In the process he helped us polish the compiler in many ways.

David Shields’ efforts in working with us were also very beneficial. He put tens of thousands of lines of C source code through High C, transliterated from the SETL version of the Ada-Ed compiler at New York University.

Professor William McKeeman and his research group at the Wang Institute of Graduate Studies supplied us with a collection of “gray expressions” that helped us verify the compiler.

The support of others who must needs remain nameless at this time is also appreciated.

Most of all we acknowledge that we are not self-made, but God-made. And we thank God for building into us the talents that made it possible for us to create High C. Praise God, from whom all blessings flow.

Ad majorem Dei gloriam (A.M.D.G.).
This ends the

High C™

Library Reference Manual

© Copyright 1985 MetaWare™ Incorporated