UNPL
1446

Bell Laboratories

subject: **A Guide to the C Library for UNIX Users**
Case: **40125-003**

date: **August 15, 1979**

from: **C. D. Perez**
MH 3624
2C-260 x6041

3624-790815.02MF

MEMORANDUM FOR FILE

The attached paper describes some commonly used functions of the C Library.

C. D. Perez

C. D. Perez

MH-3624-CDP

Copy (with att.) to
Members of Dept. 3624
Members, C Standards Task Force
B. A. Tague
J. M. Scanlon

A Guide to the C Library for UNIX, Users

C. D. Perez

Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

The C language on the UNIX† system has been traditionally provided with a rich supply of often-used routines formed into libraries selectable at load time. When the interest in portability heightened, the C library kept pace with other software being modified, and the library known as */lib/libS.a* superseded the older attempts at portability [1]. This new library [2] concentrated on input-output functions that removed the user from close contact with operating-system features. It also introduced new string functions and some memory-allocation routines. The Bell Laboratories C Standards Task Force has been responsible for many of the ideas incorporated into the C library and reflected in this document. As the C compiler finds its way onto other hardware and becomes available on different operating systems, the C Task Force continues to provide a forum for the exchange of information on the needs of the various users.

It is to the advantage of the C programmer to become acquainted with the C library functions and to keep up-to-date with new versions. To select routines from the C library is to choose available code that has been tuned for portability and efficiency. This document is meant to acquaint the programmer with a selection of functions from the C library that are commonly used and to point out differences among functions, special features, and occasionally precautions about function usage. The veteran user of the C library will find in this compendium an update to previously published information about the library.

Section 2 describes the current changes and additions to the contents of */lib/libS.a* since the Ritchie document was published. The bulk of the information appears in Attachment A, which is intended to be a user's reference tool. Function descriptions appear alphabetically within logical groupings. Where it seems helpful, examples are supplied. The values returned by the functions are identified in a way that suggests their use in portable code. The content and form of the functions as described here match the C library at the time of the distribution of UNIX/TS Edition 1.1, unless otherwise specified. Attachment B contains the contents of the header files used by many of the C library functions. Attachments C and D, the *printf*(3S) and *scanf*(3S) manual entries from the *UNIX/TS User's Manual* [3], are included for convenience.

2. UPDATE INFORMATION

2.1 General

The standard library */lib/libS.a* no longer exists separate from the rest of the C library; these routines have been incorporated into the standard UNIX C library, */lib/libc.a* This library encompasses input-output functions, routines for character type recognition and translation, space allocation, file status and a few miscellaneous routines of general use as well as many functions specific to the operating system.

Three files exist with definitions of constants, and macros that are used by many of the C library functions. *Stdio.h* contains the definitions of `NULL`, `EOF`, `FILE`, and `BUFSIZ`. The standard input file (*stdin*), standard output file (*stdout*), and standard error file (*stderr*) are also defined there. These are included in a program with `#include <stdio.h>`. The file *ctype.h* provides the macro definitions for the variety of character classifications that is now possible. Any program using those facilities must contain the line `#include <ctype.h>`. The functions that handle signals need to include the signal definitions. This can be done with `#include <signal.h>`. See Attachment B.

† UNIX is a Trademark of Bell Laboratories.

2.2 Space Allocation.

Calloc was designed to be used for acquiring space initialized to zero; *malloc* is now available to allocate a chunk of uninitialized space, and *realloc* to change the size of an already allocated amount of space. *Cfree* has been renamed *free*, and returns space acquired by any of the above three functions.

2.3 Input-Output Functions

The function *fopen* may now be supplied with new options that allow updating of a file (`r+`, `w+`, `a+`). An added routine *fdopen* acts as a bridge between the low-level UNIX input-output functions and the "standard" technique of opening a stream. *Printf* provides more versatile formatting. For operating systems that support the concept of *pipes*, and the *shell*, the functions *popen* and *pclose* add a facility for creating a pipe between the calling process and a command supplied as the argument.

2.4 Status

To acquire information about a file, *feof*, *ferror*, and *fileno* have been available. Now a function named *clearerr* is added. It resets the error condition indicated by *ferror* while the stream remains open.

2.5 Character Types

New macros added to the collection in *ctype.h* are *isalnum* (alphanumeric test), *ispunct* (for recognizing punctuation characters), *iscntrl* (to identify certain control characters) *isascii* (to find ASCII characters) and *isgraph* (having visible graphic representation). *Toascii* can be used to translate characters into ASCII; *toupper* and *tolower* are useful in changing the case of a letter.

2.6 Some Conventions

When the overhead of a function call could be substantial, because the routine suggests repetitive use, it is likely to have been implemented as a macro. *Getchar* is an illustration of this. Any "function" coded as a macro is noted in its description. In these cases the user should beware of the hazards of macro expansion on complex arguments. Cases should be avoided where arguments are automatically incremented or decremented, are evaluated more than once, contain their own macros or function calls, or whose order of operations is unclear after expansion. In short, only simple arguments are safe to use with macros. In a few cases the C library provides both a function call and a faster macro version to perform a similar task.

Some function names have changed in order to follow the established convention. To insure that the uniqueness of function names is preserved even if truncation occurs on some systems, those functions dealing with entire strings are named *str*...; those functions that consider only the first *n* characters of a string are named *strn*....

2.7 Other Additions

Software signals are implemented by two functions, *gsignal* and *ssignal*, to generate and catch error conditions respectively [4]. This facility allows the user to raise signals to be handled in whatever way seems useful; the C Library code will eventually raise signals so that calling programs, such as UNIX commands, might be enhanced to respond to such signals.

Tmpnam can be used to create a name for a temporary file. *Ctermid* retrieves the terminal identifier from the system, while *cuserid* retrieves the user ID. In each of these three functions, the user may choose to supply space for the safe storage of that name, or accept an internal storage place of suitable size.

Tmpfile provides an unnamed temporary file that continues in existence until the termination of the process that requested it. This function is implemented in UNIX/TS Edition 1.2.

3. ACKNOWLEDGMENTS

The author is grateful to J. F. Maranzano and L. Rosler for their careful reading of this document, and to A. R. Koenig for his help during its preparation. T. A. Dolotta helped to format this document.

4. REFERENCES

[1] Lesk, M. E., *The Portable C Library*, Bell Laboratories (May 1975).

[2] Ritchie, D. M., *A New Input/Output Package*, Bell Laboratories (May 1977).

[3] Dolotta, T. A., and Olsson, S. B. (eds.), *UNIX/TS User's Manual—Edition 1.1*, Bell Laboratories (Jan. 1979).

[4] Koenig, A. R., "A Proposal for Software Signals," private communication (Apr. 14, 1978).

May 1979

Attachment A

COMMON C LIBRARY FUNCTIONS

FILE ACCESS

fclose

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

Fclose closes a file that was opened by *fopen*, frees any buffers after emptying them, and returns zero on success, non-zero on error. *Exit* calls *fclose* for all open files as part of its processing.

fdopen

#include <stdio.h>
*FILE *fdopen (fildes, type)*
int fildes;
*char *type;*

Fdopen is used strictly on UNIX systems and therefore is not a portable function. Its value is in providing a bridge between the low-level input-output (I/O) facilities of UNIX and the standard I/O functions. *Fdopen* associates a stream with a valid file descriptor obtained from a UNIX system call (e.g., *open*). `Type` is the same mode (`r`, `w`, `a`, `r+`, `w+`, `a+`) that was used in the original creation of a file identified by `fildes`. *Fdopen* returns a pointer to the associated stream, or NULL if unsuccessful. The update options (`r+`, `w+`, `a+`) are available in UNIX/TS Edition 1.2.

Example:

```
int fd;
char *name = "myfile";
FILE *strm;

fd = open(name,0);
        .
        .
        .
        if((strm = fdopen(fd,"r")) == NULL)
                fprintf(stderr,"Error on %d\n",fd);
```

fileno

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

Implemented as a macro on UNIX, (and contained in the file *stdio.h*), *fileno* returns an integer file descriptor associated with a valid `stream`. Any existing non-UNIX implementations may have different meanings for the integer which is returned. *Fileno* is used by many other standard functions in the C library.

fopen

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

Fopen opens a file named `filename` and returns a pointer to a structure (hereafter referred to as `stream`), containing the data necessary to handle a stream of data. `Type` is one of the following character strings:

- `r` used to open for reading.
- `w` used to open for writing, which truncates an existing file to zero length or creates a new file.
- `a` used to append, that is, open for writing at the end of a file, or create a new file.

r+ update reading, which means open for reading and allow writing, positions the file pointer at the beginning of the file.

w+ update writing, which means open for writing and allow reading, truncates an existing file to zero length or creates a new file.

a+ update appending, which means open for writing, positions to the end of the file, and allows for subsequent reads and writes. If the file does not exist, it will be created.

For the update options, *fseek* or *rewind* are necessary to trigger the change from reading to writing or vice versa. *Fopen* returns a NULL pointer if `filename` cannot be opened. On non-UNIX implementations, file names may be something different from a UNIX-like name. The update functions are particularly applicable to stream I/O and allow for the possibility of creating temporary files for both reading and writing. The GCOS and IBM implementations contain many options other than those mentioned above. The update options are available in UNIX/TS Edition 1.2.

Example:

```
FILE *fp;
char *file;

        if((fp = fopen(file,"r")) == NULL)
                fprintf(stderr, "Cannot open %s\n",file);
```

freopen

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

Freopen accepts a pointer, `stream`, to a previously opened file; the old file is closed, and then the new file is opened. The principal motivation for *freopen* is the desire to attach the names *stdin*, *stdout*, and *stderr* to specified files. On a successful *freopen*, the stream pointer is returned; otherwise NULL is returned, indicating that, while the file closing took place, the reopening failed. *Freopen* is of limited portability; it can not be implemented in all environments.

Example:

```
char *newfile;
FILE *nfile;

        if((nfile = freopen(newfile,"r",stdout)) == NULL)
                fprintf(stderr,"Cannot reopen %s\n",newfile);
```

fseek

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

Fseek positions a stream to a location `offset` distance from the beginning, current position or end of a file, depending on the values 0, 1, 2 respectively for `ptrname`. On UNIX the offset unit is bytes; other implementations are not necessarily the same. (For example, on GCOS the offset is three 12-bit fields of block, logical-record number, and offset-into-record number.) The return values are 0 on success and EOF on failure. Both buffered and unbuffered files may make use of *fseek*. As implemented, the function cannot be ported to the OS/370 environment.

Example:

To position to the end of a file:

```
FILE *stream;

fseek(stream,0L,2);
```

pclose

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

Pclose closes a stream opened by *popen*. It returns the exit status of the command from a corresponding *popen* or −1 if the stream was not opened by *popen*. The function name *pclose* means an entirely different thing in the OS/370 environment.

popen

#include <stdio.h>
*FILE *popen (command, type)*
*char *command, *type;*

Popen is used to create a pipe between the calling process and a command to be executed. The first argument is a shell command line; *type* is the I/O mode for the pipe, and may be either `r` for reading or `w` for writing. The function returns a stream pointer to be used for I/O on the standard input or output of the command. A NULL pointer is returned if an error occurs.

Example:

```
FILE *pstrm;
char store [MAX];

printf("PARENT READS ");
if((pstrm=popen("echo message from child","r"))== NULL)
        fprintf(stderr,"No child\n");
fgets(store,MAX,pstrm);
printf("%s\n",store);
if(pclose(pstrm) == -1)
        fprintf(stderr,"Pclose error\n");
```

results in:

```
PARENT READS message from child
```

rewind

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

Rewind sets the position of the next operation at the beginning of the file associated with `stream`, retaining the current mode of the file. It is the equivalent of *fseek (stream,0L,0);*.

setbuf

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

This function allows the user to choose his own buffer for I/O or to choose to have no buffering at all. Use it after opening and before reading or writing. The function is often used to eliminate the single character writes to a file that result from the execution of *putc* to standard output that is not redirected. The constant `BUFSIZ` in *stdio.h* tells how big the character array `buf` is. When `buf` is set to `NULL`, the I/O is completely unbuffered. (On GCOS the function is implemented as a null macro, because that environment does not need such a function.)

Example:

```
setbuf (stdout, malloc(BUFSIZ));
```

FILE STATUS

clearerr

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

Clearerr is used to reset the error condition on `stream`. The need for *clearerr* arises on UNIX implementations, where the error indicator is not reset after a query. This function is incorrectly referred to as *clrerr* in the UNIX User's Manual [3].

feof

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

Feof returns non-zero if an input operation on `stream` has reached end of file; otherwise a zero is returned. *Feof* should be used in conjunction with any I/O function whose return value is not a clear indicator of an end-of-file condition. Such functions are *fread* and *getw*.

Example:

```
int *x;
FILE *stream;

do
        *x++ = getw(stream);
while(!feof(stream));
```

ferror

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

Ferror tests for an indication of error on `stream`. It returns a non-zero value (true) when an error is found, and a zero otherwise. Calls to *ferror* do not clear the error condition, hence the *clearerr* function is needed for that purpose. On UNIX *ferror* is implemented as a macro.

Example:

```
FILE *stream;
int *x;

while(!ferror(stream))
        putw(*x++,stream);
```

ftell

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

Ftell is used to determine the current offset relative to the beginning of the file associated with `stream`. It returns the current value of the offset; in UNIX it returns the offset value in bytes. On error, a value of −1 is returned. This function is useful in obtaining an offset for subsequent *fseek* calls on some systems.

INPUT FUNCTIONS

fgetc

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

This is the function version of the macro *getc* and acts identically to *getc*. Because *fgetc* is a function and not a macro,it can be used in debugging to set breakpoints on *fgetc* and when the side effects of macro processing of the argument is a problem. Furthermore, it can be passed as an argument.

fgets

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

Fgets reads from `stream` into the area pointed to by `s` either `n-1` characters or an entire string including its new-line terminator. A final null character is affixed to the data read. It returns the pointer `s` on success, and `NULL` on end-of-file or error. *Fgets* differs from the function *gets* in that it can read from other than *stdin*, and that it appends the new-line at the end of input when the size of the string is longer than or equal to `n`. More importantly, it provides control over the size of the string to be read that is not available with *gets*. (See example under *strlen* below.)

Example:

```
char msg[MAX];
FILE *myfile;

        while(fgets(msg,MAX,myfile) != NULL)
                printf("%s\n",msg);
```

fread

#include <stdio.h>
*int fread((char *)ptr, sizeof (*ptr), nitems, stream)*
*FILE *stream;*

This function reads from `stream` the next `nitems` whose size is the same as the size of the item pointed to by `ptr`, into a sufficiently large area starting at `ptr`. It returns the number of items read. In UNIX, *fread* makes use of the function *getc*. It is often used in combination with *feof* and *ferror* to obtain a clear indication of the file status.

Example:

```
FILE *pstrm;
char mesg[100];

        while(fread((char *)mesg,sizeof(mesg),1,pstrm) == 1)
                printf("%s\n",mesg);
```

fscanf

#include <stdio.h>
int fscanf (stream, format[, argptr]...)
*char *format;*
*FILE *stream;*

Fscanf accepts input from the file associated with `stream`, and deposits it into the storage area pointed to by the respective argument pointers according to the specified formats. Format specifications are those that appear in Attachment D. *Fscanf* differs from *scanf* in that it can read from other than *stdin*. The function returns the number of successfully handled input arguments, or `EOF` on end-of-input.

Example:

```
FILE *file;
long pay;
char name[15];
char pan[7];

        fscanf(file,"%6s%14s%ld\n",pan,name,&pay);
        if(pay<50000)
                printf("$%ld raise for %s.\n",pay/10,name);
```

If the input data is:

```
020202MaryJones 15000
```

the resulting output is:

```
$1500 raise for MaryJones.
```

getc

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

Getc returns the next character from the named `stream`. On UNIX it is implemented as a macro to avoid the overhead of a function call. On error or end-of-file it returns an `EOF`. *Fgetc* should be used when it is necessary to avoid the side effects of argument processing by the macro *getc*.

getchar

#include < stdio.h>
int getchar()

This is identical to *getc (stdin)*.

gets

#include < stdio.h>
*char *gets(s)*
*char *s;*

Gets reads a string of characters up to a new-line from *stdin* and places them in the area pointed to by `s`. The new-line character which ended the string is replaced by the null character. The return values are `s` on success, `NULL` on error or end-of-file. The simple example below presumes the size of the string read into `msg` will not exceed `SIZE` in length. See *strlen* for a check on oversized input.

Example:

```
char msg[SIZE];
char *s;
        s = msg;
        while (gets(s) != NULL)
                printf("%s\n",s);
```

getw

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

Getw reads the next word from the file associated with `stream`. On success it returns the word; on error or end of file, it returns `EOF`. However, because `EOF` could be a valid word, this function is best used with *feof* and *ferror*.

Example:

```
FILE *stream;
int *x;
        do
                *x++ = getw(stream);
        while (!feof(stream));
```

scanf

#include < stdio.h>
int scanf (format[, argptr]...)
*char *format;*

Scanf reads input from *stdin*, delivers the input according to the specified formats, and deposits the input in the storage area pointed to by the respective argument pointers. The correct format specifications can be found in Attachment D. For input from other streams than *stdin* use *fscanf*; for input from a character array use *sscanf*. The return values are the number of successfully handled input arguments, or EOF on end-of-input.

Example:

```
long number;

        scanf("%ld",&number);
        (printf(number%2?"%ld is odd":"%ld is even",number));
```

sscanf

#include < stdio.h>
sscanf (s, format [, pointer]...)
*char *s;*
*char *format;*

Sscanf accepts input from a character string s, delivers the input according to the specified formats, and deposits it into the storage area pointed to by the respective argument pointers. Format specifications appear in Attachment D. This function returns the number of successfully handled input arguments.

Example:

```
char datestr[] = {"THU MAR 29 11:04:40 EST 1979"};
char month[4];
char year[5];

        sscanf(datestr,"%*3s%3s%*2s%*8s%*3s%4s",month,year);
        printf("%s, %s\n",month,year);
```

The result is:

```
MAR, 1979
```

ungetc

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

Ungetc puts the character c back on the file associated with stream. One character (but never EOF) is assured of being put back. If successful, the function returns c, otherwise EOF.

Example:

```
while(isspace (c = getc(stdin)))
        ;
ungetc(c,stdin);
```

This code puts the first character that is not white space back onto the standard input stream.

OUTPUT FUNCTIONS

fflush

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

Fflush takes action to guarantee that any data contained in file buffers and not yet written out will be written. It is used by *fclose* to flush a stream. No action is taken on files not open for writing. The return values are zero for success, EOF on error.

fprintf

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

Fprintf provides formatted output to a named stream. The function *printf* may be used if the destination is *stdout*. Specifications for formats are available in Attachment C. On error, *fprintf* returns non-zero, otherwise zero.

Example:

```
int *filename;
int c;

        if(c==EOF)
                fprintf(stderr,"EOF on %s\n",filename);
```

fputc

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

Fputc performs the same task as *putc;* that is, it writes the character `c` to the file associated with `stream`, but is implemented as a function rather than a macro. It is preferred to *putc* when the side effects of macro processing of arguments is a problem. On success, it returns the character written; on failure it returns EOF.

Example:

```
FILE *in, *out;
int c;

        while ((c = fgetc(in)) != EOF)
                fputc(c,out);
```

fputs

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

Fputs copies a string to the output file associated with `stream`. In UNIX it uses the function *putc* to do this. It is different from *puts* in two ways: it allows any output stream to be specified, and it does not affix a new-line to the output. On success it returns the last character copied, otherwise EOF. For an example, see *puts*.

fwrite

#include <stdio.h>
*int fwrite ((char *)ptr, sizeof (*ptr),nitems,stream)*
*FILE *stream;*

Beginning at `ptr`, this function writes up to `nitems` of data of the type pointed to by `ptr` into output `stream`. It returns the number of items actually written. For the GCOS implementation, `ptr` must be on a machine-word boundary. Like *fread* this function should be used in conjunction with *ferror* to detect the error condition.

Example:

```
char mesg[] ={"My message to write out\n"};
FILE *pstrm;

        if(fwrite(mesg,(sizeof(mesg)-1),1,pstrm) != 1)
                fprintf(stderr,"Output error\n");
```

printf

#include < stdio.h>
int printf(format[, arg]...)
*char *format;*

Printf provides formatted output on *stdout*. The many format specifications are available in Attachment C. *Fprintf* and *sprintf* are related functions that write output onto other than the standard output device. In case of error, implementations are not consistent in their output. On error, *printf* returns non-zero, otherwise zero.

Example:

```
int num = 10;
char msg[] = {"ten"};
printf("%d - %o - %s\n", num, num, msg);
```

results in the line:

```
10 - 12 - ten;
```

putc

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

Putc writes the character *c* to the file associated with *stream*. On success, it returns the character written; on error it returns EOF. Because it is implemented as a macro, side effects may result from argument processing. In such cases, the equivalent function *fputc* should be used.

Example:

```
#define PROMPT()        putc('\7',stderr)       /* BEL */
```

putchar

#include < stdio.h>
int putchar(c)
int c;

Putchar is defined as *putc (c, stdout)*. It returns the character written on success, or EOF on error.

Example:

```
char *cp;
char x[SIZE];

        for(cp=x;cp<(x+SIZE);cp++)
                putchar(*cp);
```

puts

#include < stdio.h>
int puts(s)
*char *s;*

The function copies the string pointed to by s without its terminating null character to *stdout*. A new-line character is appended. The UNIX implementation uses the macro *putchar* (which calls *putc*).

Example:

```
puts("I will append a new-line");
fputs("\tsome more data ", stdout);
puts("and now a new-line");
```

The resulting output is:

```
I will append a new-line
        some more data and now a new-line
```

putw

#include <stdio.h>
int putw(w,stream)
*FILE *stream;*
int w;

Putw appends word w to the output `stream`. As with *getw*, the proper way to check for an error or end-of-file is to use the *feof* and *ferror* functions.

Example:

```
int info;

while(!feof(stream))
        putw(info,stream);
```

sprintf

#include <stdio.h>
int sprintf(s, format, [, arg]...)
*char *s;*
*char *format;*

Sprintf allows for formatted output to be placed in a character array pointed to by s. *Sprintf* adds a null at the end of the string s. See Attachment C for the specification of formats. It is the user's responsibility to provide an array of sufficient length. Other related functions *printf* and *fprintf* handle similar kinds of formatted output. *Sprintf* can be used to build formatted arrays in memory, to be changed dynamically before output, or to be used to call other routines. The comparable input function is *sscanf*. On error, *sprintf* returns non-zero, otherwise zero.

Example:

```
char command[100];
char *doc = "/usr/src/cmd/cp.c"
int width = 50;
int length = 60;

        sprintf(command,"pr -w%d -l%d %s\n",width,length,doc);
        system(command);
```

The above code executes the *pr* command to print the source of the *cp* command.

STRING FUNCTIONS

strcat

*char *strcat(dst,src)*
*char *dst, *src;*

Strcat appends characters in the string pointed to by `src` to the end of the string pointed to by `dst`, and places a null character after the last character copied. It returns a pointer to `dst`. To concatenate strings up to a maximum number of characters, use *strncat*.

Example:

```
char *myfile;
char dir[16] = "/usr/";
        myfile = (strcat(dir,cuserid(0)));
```

The result is the concatenation of the login name onto the end of the string `dir`.

strchr *char *strchr(s,c)*
*char *s;*
int c;

Strchr searches a string pointed to by `s`, for the leftmost occurrence of the character `c`. It returns a pointer to the character found, or `NULL` if `c` does not occur in the string.

Example:

```
char *length;
char *a;
register char *b;

length = ((b=strchr(a,' ')) - a);
```

The resulting `length` is the number of characters up to the first blank in the string pointed to by `a`.

strcmp *char *strcmp(s1,s2)*
*char *s1, *s2;*

Strcmp compares the characters in the string `s1` and `s2`. It returns an integer value, greater than, equal to, or less than zero, depending on whether `s1` is lexicographically greater than, equal to, or less than `s2`.

Example:

```
long *clock;
if(strcmp(ctime(clock),"Sun") == 0)
        newweek();
```

strcpy *char *strcpy(dst, src)*
*char *dst, *src;*

Strcpy copies the characters (including the null terminator) from the string pointed to by `src` into the string pointed to by `dst`. A pointer to `dst` is returned.

Example:

```
char dst[];
char src[] = "this is lower case";

        printf("%s\n",strcpy(dst,src+8));
```

results in:

```
lower case
```

strlen *int strlen(s)*
*char *s;*

Strlen counts the number of characters starting at the character pointed to by `s` up to, but not including, the first null character. It returns the integer count.

Example:

```
char msg[MAX];

if(strlen (gets(msg)) >= MAX)
        fprintf(stderr,"String length exceeded!\n");
```

strncat *char *strncat(dst, src, n)*
*char *dst, *src;*
int n;

Strncat appends a maximum of n characters of the string pointed to by src and then a null character to the string pointed to by dst. It returns a pointer to dst.

Example:

```
char dst[] = "cover";
char src[] = "letter";

        printf("%s\n",strncat(dst,src,3));
```

The output is:

```
coverlet
```

strncmp

int strncmp(s1,s2,n)
*char *s1, *s2;*
int n;

Strncmp compares two strings for at most n characters and returns an integer greater than, equal to, or less than zero as s1 is lexicographically greater than, equal to or less than s2.

Example:

```
char filename [] = "/dev/ttyx";

if(strncmp (filename+5, "tty",3) == 0)
        printf("success\n");
```

strncpy

*char *strncpy(dst,src,n)*
*char *dst, *src;*
int n;

Strncpy copies n characters of the string pointed to by src into the string pointed to by dst. Null padding or truncation of src occurs as necessary. A pointer to dst is returned.

Example:

```
char date [29] = {"Fri Jun 29 09:35:44 EDT 1979"};
char *day;

        strncpy(day,date,3);
```

After executing this code, the contents of day is Fri.

strrchr

*char *strrchr(s,c)*
*char *s;*
int c;

Strrchr searches a string pointed to by s, for the rightmost occurrence of the character c. It returns a pointer to the character found, or NULL if c does not occur in the string.

Example:

```
char reverse[] = "NAME NO ONE MAN";

        printf(strrchr (reverse,'M'));
```

results in:

```
MAN
```

CHARACTER CLASSIFICATION

isalnum

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

This macro determines whether or not the character c is an alphanumeric character ([A-Za-z0-9]). It returns zero for false and non-zero for true.

isalpha

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

This macro determines whether or not the character c is an alphabetic character ([A-Za-z]). It returns zero for false and non-zero for true.

isascii

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

This macro determines whether or not the integer value supplied is an ASCII character; that is, a character whose octal value ranges from 000 to 177. It returns zero for false and non-zero for true.

iscntrl

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

This macro determines whether or not the character c when mapped to ASCII is a control character (that is, octal 177 or 000-037). It returns zero for false and non-zero for true.

isdigit

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

This macro determines whether or not the character c is a digit. It returns zero for false and non-zero for true.

isgraph

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

This macro determines whether or not the character c has a graphic representation (that is, is an ASCII code between octal 041 and 176 inclusive).

islower

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

This macro determines whether or not the character c is a lower-case letter. It returns zero for false and non-zero for true.

isprint

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

This macro determines whether or not the character c is a printable character. (This includes spaces.) It returns zero for false and non-zero for true.

ispunct

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

This macro determines whether or not the character c is a punctuation character (neither a control character nor an alphanumeric). It returns zero for false and non-zero for true.

isspace

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

This macro determines whether or not the character c is a form of white space (that is, a blank, horizontal or vertical tab, carriage return, form-feed or new-line). It returns zero for false and non-zero for true.

isupper

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

This macro determines whether or not the character c is an upper-case letter. It returns zero for false and non-zero for true.

CHARACTER TRANSLATION

toascii

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

The macro *toascii* usually does nothing in the UNIX environment. Its purpose is to map the input character into its ASCII equivalent. In a non-ASCII environment it is beneficial for users of character indices into tables ordered in ASCII collating sequence.

Example:

```
FILE *oddstrm;

        if(!isdigit (toascii(getw(oddstrm))))
                fprintf(stderr,"bad data\n");
```

tolower

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

If the argument c passed to the function *tolower* is an upper-case letter, the lower-case representation of c is returned, otherwise c is returned unchanged. For a faster routine, use *_tolower*, which is implemented as a macro; however, the argument *must* already be an upper-case letter.

Example:

```
if(tolower(getchar()) != 'y')
        exit(0);
```

toupper

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

If the argument c passed to the function *toupper* is a lower-case letter, the upper-case representation of c is returned, otherwise c is returned unchanged. For a faster routine, use *_toupper*, however, the argument *must* already be a lower-case letter.

Example:

```
if(toupper (getchar()) != 'Y')
        exit(0);
```

SPACE ALLOCATION

calloc *char *calloc(n, size)*
unsigned n, size;

Calloc allocates enough storage for an array of `n` items aligned for any use, each of `size` bytes. The space is initialized to zero. *Calloc* returns a pointer to the beginning of the allocated space, or a NULL pointer on failure.

Example:

```
char *t;
int n;
unsigned size;

        if(t=calloc((unsigned)n, size) == NULL)
                fprintf(stderr,"Out of space.\n");
```

free *free(ptr)*
*char *ptr;*

Free is used in conjunction with the space allocating functions *malloc*, *calloc*, or *realloc*. `Ptr` is a pointer supplied by one of these routines. The effect is to free the space previously allocated.

malloc *char *malloc(size)*
unsigned size;

Malloc allocates `size` bytes of storage beginning on a word boundary. It returns a pointer to the beginning of the allocated space, or a NULL pointer on failure to acquire space. For space initialized to zero, see *calloc*.

Example:

```
int n;
char *t;
unsigned size;

        if(t=malloc((unsigned)n) == NULL)
                fprintf(stderr,"Out of space.\n");
```

realloc *char *realloc (ptr, size)*
*char *ptr;*
unsigned size;

Given `ptr` which was supplied by a call to *malloc* or *calloc*, and a new byte size, `size`, *realloc* returns a pointer to the block of space of `size` bytes. This function is useful to do storage compacting along with *malloc* and *free*.

MISCELLANEOUS FUNCTIONS

ctermid *#include <stdio.h>*
*char *ctermid(s)*
*char *s;*

Ctermid provides a string that can be used as a file name, (`/dev/tty`), to identify the controlling terminal for the running process. Unlike the function *ttyname* it is disassociated from the machine-dependent concept of a file descriptor. If an argument of zero is supplied, the string is stored internally and will be overwritten on the next call to *ctermid*. A non-zero argument is treated as a pointer to a sufficiently large storage area where the string is placed.

cuserid

#include <stdio.h>
*char *cuserid(s)*
*char *s;*

Cuserid composes a string representation of the login name of the owner of the current process. A zero argument results in the string being stored in an internal area; in this case a pointer to that area is returned on success, and a NULL on failure. A non-zero argument is assumed to be a pointer to a repository of suitable size; on failure a null character will be inserted in place of a string and a NULL is returned.

Example:

```
puts (cuserid((char *) NULL));
```

gsignal

#include <signal.h>
int gsignal (sig)
int sig;

Along with *ssignal*, *gsignal* implements a facility for software signals. A software signal is raised by a call to *gsignal*. Raising a software signal causes the action established by *ssignal* to be taken. The argument `sig` identifies the signal to be set. If `sig` is a value defined in *signal.h*, then *gsignal* returns that value. If an action function was established for `sig`, then the action is reset to the default value, the action function is performed with argument `sig`, and the return value is the return value of the action function. In any abnormal case, *gsignal* returns the value 0 and takes no other action.

Example:

```
char *buf;
        if((buf = gets(string))==NULL) gsignal(2);
```

ssignal

#include <signal.h>
int (ssignal (sig,action))()*
*int sig, (*action)();*

Ssignal along with *gsignal* implements a software signal facility. An action for a software signal is established by a call to *ssignal*. `Sig` is the number identifying the type of signal for which an action is to be established. The numbers currently defined are found in *signal.h*. `Action` is either the name of a user-defined action function or one of the constants defined in *signal.h*. *Ssignal* returns the action previously established for that signal type; in abnormal circumstances *ssignal* returns a default of zero.

Example:

```
main() {
int error();
ssignal(2, error);
        .
        .
        .
}
error(x) {
int x;
printf("Software signal %d has been caught.\n",x);
}
```

system

#include <stdio.h>
system(string)
*char *string;*

System passes the argument `string` to the operating system as a command line. It returns the exit status of the command executed.

Example:

```
if(!system ("cmp -s file1 file2"))
        printf("The two files are identical.\n");
```

tmpnam

#include <stdio.h>
*char *tmpnam(s)*
*char *s;*

Tmpnam generates a file name that can be used for a temporary file. If s is zero, it returns a pointer to a character string containing that name in an internal storage area. For a non-zero value in s, the file name is stored in a sufficiently large area pointed to by s, and s is the return value as well.

tmpfile

#include <stdio.h>
*FILE *tmpfile ()*

Tmpfile creates a scratch file opened for update. It stays in existence only during the life of the process issuing the function call. It returns a pointer to the FILE associated with the opened stream. On error, it returns NULL. This function is implemented in UNIX/TS Edition 1.2.

Attachment B

ctype.h

```
#define _U      01
#define _L      02
#define _N      04
#define _S     010
#define _P     020
#define _C     040
#define _B    0100

extern  char    _ctype_[];

#define isalpha(c)      ((_ctype_+1)[c]&(_U|_L))
#define isupper(c)      ((_ctype_+1)[c]&_U)
#define islower(c)      ((_ctype_+1)[c]&_L)
#define isdigit(c)      ((_ctype_+1)[c]&_N)
#define isspace(c)      ((_ctype_+1)[c]&(_S|_B))
#define ispunct(c)      ((_ctype_+1)[c]&_P)
#define isalnum(c)      ((_ctype_+1)[c]&(_U|_L|_N))
#define isprint(c)      ((_ctype_+1)[c]&(_P|_U|_L|_N|_B))
#define isgraph(c)      ((_ctype_+1)[c]&(_P|_U|_L|_N))
#define iscntrl(c)      ((_ctype_+1)[c]&_C)
#define isascii(c)      ((unsigned)(c)<=0177)
#define _toupper(c)     ((c)-'a'+'A')
#define _tolower(c)     ((c)-'A'+'a')
#define toascii(c)      ((c)&0177)
```

signal.h

```
#define     NSIG        16

#define     SIGHUP       1    /* hangup */
#define     SIGINT       2    /* interrupt */
#define     SIGQUIT      3    /* quit */
#define     SIGILL       4    /* illegal instruction (not reset when caught) */
#define     SIGTRAP      5    /* trace trap (not reset when caught) */
#define     SIGIOT       6    /* IOT instruction */
#define     SIGEMT       7    /* EMT instruction */
#define     SIGFPE       8    /* floating point exception */
#define     SIGKILL      9    /* kill (cannot be caught or ignored) */
#define     SIGBUS      10    /* bus error */
#define     SIGSEGV     11    /* segmentation violation */
#define     SIGSYS      12    /* bad argument to system call */
#define     SIGPIPE     13    /* write on a pipe with no one to read it */
#define     SIGALRM     14    /* alarm clock */
#define     SIGTERM     15    /* software termination signal from kill */

int         (*signal())();
#define     SIG_DFL     (int (*)())0
#define     SIG_IGN     (int (*)())1
```

stdio.h

```
#define         BUFSIZ          512
#define         _NFILE           20
# ifndef FILE
extern  struct  _iobuf {
        char    *_ptr;
        int     _cnt;
        char    *_base;
        char    _flag;
        char    _file;
} _iob[_NFILE];
# endif

#define         _IOREAD          01
#define         _IOWRT           02
#define         _IONBF           04
#define         _IOMYBUF        010
#define         _IOEOF          020
#define         _IOERR          040
#define         _IOSTRG        0100
#define         _IORW          0200

#define         NULL             0
#define         FILE            struct  _iobuf
#define         EOF             (-1)

#define         L_ctermid        9
#define         L_cuserid        9
#define         L_tmpnam        19

#define         stdin           (&_iob[0])
#define         stdout          (&_iob[1])
#define         stderr          (&_iob[2])
#define         getc(p)         (--(p)->_cnt>=0?\
                                *(p)->_ptr++&0377:_filbuf(p))
#define         getchar()       getc(stdin)
#define         putc(x,p)       (--(p)->_cnt>=0?\
                                ((int)(*(p)->_ptr++=(unsigned)(x))):\
                                _flsbuf((unsigned)(x),p))
#define         putchar(x)      putc(x,stdout)
#define         feof(p)         (((p)->_flag&_IOEOF)!=0)
#define         ferror(p)       (((p)->_flag&_IOERR)!=0)
#define         fileno(p)       p->_file

FILE    *fopen();
FILE    *freopen();
FILE    *fdopen();
long    ftell();
char    *fgets();
```

Attachment C

NAME

printf, fprintf, sprintf − formatted output conversion

SYNOPSIS

```
#include <stdio.h>

printf (format [, arg ] ... )
char *format;

fprintf (stream, format [, arg ] ... )
FILE *stream;
char *format;

sprintf (s, format [, arg ] ... )
char *s, format;
```

DESCRIPTION

Printf places output on the standard output stream *stdout*. *Fprintf* places output on the named output *stream*. *Sprintf* places "output" in the string *s*, followed by the character \0. The string *s* must be long enough.

Each of these functions converts, formats, and prints each *arg* under control of the *format*. The *format* is a character string that contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which causes conversion and printing of the next successive *arg*.

Each conversion specification is introduced by the character %. After the %, the following appear in sequence:

- an optional minus sign (−), which specifies *left adjustment* of the converted value in the indicated field;
- an optional zero, which specifies that zero-padding will be done instead of blank-padding;
- an optional digit string specifying a *field width;* if the converted value has fewer characters than the field width, it will be padded on the left (or right, if the left-adjustment indicator has been given) to make up the field width;
- an optional period (.), which serves to separate the field width from the next digit string;
- an optional digit string specifying a *precision* that gives the number of digits to appear after the decimal point for the e and f conversions, the maximum number of significant digits for the g conversion, or the maximum number of characters to be printed from a string; it also serves as a modifier in the o and x conversions;
- an optional l or h, specifying that a following d, i, o, x, X, or u corresponds to a long integer (l) or a short integer (h) *arg*;
- a character that indicates the type of conversion to be applied.

A field width or precision may be indicated by an asterisk (*) instead of by a digit string. In this case, an integer *arg* supplies the field width or precision. If the integer corresponding to a precision has the value −1, the effect is as if the precision and its preceding decimal point were both absent.

If the end of the *format* occurs between a % and its following format code, that entire format item is ignored.

The conversion characters and their meanings are:

d,i,o,x,X The integer *arg* is converted to decimal (for either d or i), octal, or hexadecimal notation, respectively. The letters **abcdef** are used for x conversion and the letters **ABCDEF** for X conversion. If the *precision* is present, a single leading zero will be prepended to a non-zero value in o conversion, and the string 0x (0X) will be prepended to the value in x (X) conversion.

f The float or double *arg* is converted to decimal notation in the style "[−]ddd.ddd", where the number of digits after the decimal point is equal to the precision specification for the argument. If the precision is missing, 6 digits are given; if the precision is explicitly 0, no digits and no decimal point are printed, unless left-justification and zero-padding are both specified, and the field width is strictly larger than the minimum required.

e,E The float or double *arg* is converted in the style "[−]d.ddde±dd", where there is one digit before the decimal point and the number of digits after it is equal to the precision specification for the argument; when the precision is missing, 6 digits are produced. The E format code will produce a number with E instead of e introducing the exponent. If left-justification and zero-padding are both specified, any zeroes so generated will appear before the e (E). If the precision is zero and no padding zeroes are generated on the right, no decimal point will appear.

g,G The float or double *arg* is printed in style d, in style f, or in style e (or in style E in the case of a G format code), whichever gives the requested precision in minimum space.

c The character *arg* is printed if it is not \0.

s *Arg* is taken to be a string (character pointer) and characters from the string are printed until a null character or until the number of characters indicated by the precision specification is reached; however, if the precision is missing, all characters up to a null character are printed.

u The unsigned integer *arg* is converted to decimal and printed (the result will be in the range 0 to 65,535 for integer values, or 0 to 4,294,967,295 for long values).

% Print a %; no argument is converted.

In no case does a non-existent or small field width cause truncation of a field; padding takes place only if the specified field width exceeds the actual width. Characters generated by *printf* are printed by calling *putchar* (see *putc*(3S)).

EXAMPLES

To print a date and time in the form "Sunday, July 3, 10:02", where *weekday* and *month* are pointers to null-terminated strings:

```
printf("%s, %s %d, %02d:%02d", weekday, month, day, hour, min);
```

To print π to 5 decimal places:

```
printf("pi = %.5f", 4*atan(1.0));
```

SEE ALSO

ecvt(3C), putc(3S), scanf(3S), stdio(3S).

NOTES

For compatibility with earlier versions of *printf*, the format codes D, O, and U are currently implemented to mean the same as ld, lo, and lu. This usage should be avoided.

BUGS

Outrageous precision specifications on e, f, and g formats can cause failure.

Attachment D

NAME

scanf, fscanf, sscanf — formatted input conversion

SYNOPSIS

```
#include <stdio.h>

scanf (format [ , pointer ] ... )
char *format;

fscanf (stream, format [ , pointer ] ... )
FILE *stream;
char *format;

sscanf (s, format [ , pointer ] ... )
char *s, *format;
```

DESCRIPTION

Scanf reads from the standard input stream *stdin*. *Fscanf* reads from the named input *stream*. *Sscanf* reads from the character string *s*. Each function reads characters, interprets them according to a format, and stores the results in its arguments. Each expects, as arguments, a control string *format* described below, and a set of *pointer* arguments indicating where the converted input should be stored.

The control string usually contains conversion specifications, which are used to direct interpretation of input sequences. The control string may contain:

1. Blanks, tabs, or new-lines, which cause input to be read up to the next non-white-space character.
2. An ordinary character (not %), which must match the next character of the input stream.
3. Conversion specifications, consisting of the character %, an optional assignment suppressing character *, an optional numerical maximum field width, and a conversion character.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding argument, unless assignment suppression was indicated by *. An input field is defined as a string of non-space characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted.

The conversion character indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. The following conversion characters are legal:

- **%** a single % is expected in the input at this point; no assignment is done.
- **d** a decimal integer is expected; the corresponding argument should be an integer pointer.
- **o** an octal integer is expected; the corresponding argument should be an integer pointer.
- **x** a hexadecimal integer is expected; the corresponding argument should be an integer pointer.
- **s** a character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating \0, which will be added automatically. The input field is terminated by a space character or a new-line.
- **c** a character is expected; the corresponding argument should be a character pointer. The normal skip over space characters is suppressed in this case; to read the next non-space character, use %1s. If a field width is given, the corresponding argument should refer to a character array; the indicated number of characters is read.
- **e,f** a floating point number is expected; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a *float*. The input format for floating point numbers is an optionally signed string of digits, possibly containing a decimal point, followed by an optional exponent field consisting of an E or an e, followed by an optionally signed integer.

[indicates a string that is not to be delimited by space characters. The left bracket is followed by a set of characters and a right bracket; the characters between the brackets define a set of characters making up the string. If the first character is not a circumflex (^), the input field consists of all characters up to the first character that is not in the set between the brackets; if the first character after the left bracket is a ^, the input field consists of all characters up to the first character that is in the set of the remaining characters between the brackets. The corresponding argument must point to a character array.

The conversion characters **d**, **o**, and **x** may be capitalized and/or preceded by **l** to indicate that a pointer to **long** rather than to **int** is in the argument list. Similarly, the conversion characters e and f may be capitalized and/or preceded by **l** to indicate that a pointer to **double** rather than to **float** is in the argument list. The character **h** will, some time in the future, indicate **short** data items.

Scanf conversion terminates at **EOF**, at the end of the control string, or when an input character conflicts with the control string. In the latter case, the offending character is left unread in the input stream.

Scanf returns the number of successfully matched and assigned input items; this number can be zero in the event of an early conflict between an input character and the control string. If the input ends before the first conflict or conversion, **EOF** is returned.

EXAMPLES

The call:

```
int i; float x; char name[50];
scanf ("%d%f%s", &i, &x, name);
```

with the input line:

```
25 54.32E-1 thompson
```

will assign to *i* the value **25**, to *x* the value **5.432**, and *name* will contain **thompson\0**. Or:

```
int i; float x; char name[50];
scanf ("%2d%f%*d%[1234567890]", &i, &x, name);
```

with input:

```
56789 0123 56a72
```

will assign **56** to *i*, **789.0** to *x*, skip **0123**, and place the string **56\0** in *name*. The next call to *getchar* (see *getc*(3S)) will return **a**.

SEE ALSO

atof(3C), getc(3S), printf(3S).

NOTE

Trailing white space (including a new-line) is left unread unless matched in the control string.

DIAGNOSTICS

These functions return EOF on end of input and a short count for missing or illegal data items.

BUGS

The success of literal matches and suppressed assignments is not directly determinable.