S

sbrk() — Extended function (libc)

Increase a program's data space
char *sbrk(unsigned short *increment***);**

sbrk increases a program's data space by *increment* bytes. It increments the variable **__end**; this variable is set by the C runtime startup routine, and points to the end of the program's data space.

The memory-allocation function **malloc** calls **sbrk** should you attempt to allocate more space than is available in the program's data space.

sbrk returns a pointer to the previous setting of **__end** if the requested memory is available, or **((char *)-1)** if it is not.

See Also

__end, malloc, maxmem

Notes

sbrk will not increase the size of the program data area if the physical memory requested exceeds the physical memory allocated by MS-DOS, or if the requested memory exceeds the limit set in the user-defined variable **maxmem**. **sbrk** does not keep track of how space is used. Therefore, memory seized with **sbrk** cannot be freed. *Caveat utilitor*.

This function is not described in the ANSI Standard. Programs that use it do not conform strictly to the Standard, and may not be portable to other compilers or environments.

scanf() — STDIO (libc)

Read and interpret text from standard input stream
#include <stdio.h>
int scanf(const char **format***, ...);**

scanf reads characters from the standard input stream and uses the string *format* to interpret what it has read into the appropriate types of data.

format is a string that consists of one or more conversion specifications, each of which describes how a portion of text is to be interpreted. *format* is followed by zero or more arguments. There should be one argument for each conversion specification within *format*, and each should point to the data type that corresponds to the conversion specifier within its corresponding conversion specification. For example, if *format* contains three conversion specifications that convert text into, respectively, an **int**, a **float**, and a string, then *format* should be followed by three arguments that point, respectively, to an **int**, a **float**, and an array of **char**s that is large enough to hold the string being input. If there are fewer arguments than conversion specifications, then **scanf**'s behavior is undefined. If there are more, then every argument without a corresponding conversion specification is evaluated and then ignored. If an argument is not of the same type as its corresponding type specification, then **scanf** returns.

scanf organizes the text read into a series of tokens. Each token is delimited by white space. White space usually is thrown away, except in the case of the 'c' or '[' conversion specifiers, which are described below.

If an input error occurs during input or if **EOF** is read, **scanf** returns immediately. If it reads an inappropriate character (e.g., an alphabetic character where it expects a digit), it returns immediately. **scanf** returns the number of conversions it accomplished. If it could accomplish no conversions, it returns **EOF**.

Conversion Specifications

The percent sign character '%' marks the beginning of a conversion specification. The '%' will be followed by one or more of the following:

- An asterisk '*', which tells **scanf** to skip the next conversion; that is, read the next token but do not write it into the corresponding argument.
- A decimal integer, which tells **scanf** the maximum width of the next field being read. How the field width is used varies among conversion specifier. See the table of specifiers below for more information.
- One of the three modifiers **h**, **l**, or **L**, whose use is described below.
- A conversion specifier, whose use is described below.

Modifiers

The following three modifiers may be used before a conversion specifier:

h When used before the conversion specifiers **d**, **i**, **o**, **u**, **x**, or **X**, it specifies that the corresponding argument points to a **short int** or an **unsigned short int**. When used before **n**, it indicates that the corresponding argument points to a **short int**. In implementations where **short int** and **int** are synonymous, it is not needed. However, it is useful in writing portable code.

l When used before the conversion specifiers **d**, **i**, **o**, **u**, **x**, or **X**, it specifies that the corresponding argument points to a **long int** or an **unsigned long int**. When used before **n**, it indicates that the corresponding argument points to a **long int**. In implementations where **long int** and **int** are synonymous, it is not needed. However, it is useful in writing portable code.

L When used before the conversion specifiers **e**, **E**, **f**, **F**, or **G**, it indicates that the corresponding argument points to a **long double**.

If **h**, **l**, or **L** is used before a conversion specifier other than the ones mentioned above, it is ignored. In previous releases of **Let's C**, the modifier **L** meant that the corresponding argument pointed to a **long** rather than to a **long double**, as it does now. This has been changed to conform to the ANSI Standard, and may require that some code be rewritten.

Conversion Specifiers

The Standard describes the following conversion specifiers:

c Convert into **char**s the number of characters specified by the field width, and write them into the array pointed to by the corresponding argument. The default field width is one. **scanf** does not write a null character at the end of the array it creates. This specifier forces **scanf** to read and store white-space characters and numerals, as well as letters.

d Convert the token to a decimal integer. The format should be equivalent to that expected by the function **strtol** with a base argument of ten. The corresponding argument should point to an **int**.

D Convert the token to a **long**. This conversion specifier is not described in the ANSI Standard, and using it means that your program will not comply strictly with the Standard.

e Convert the token to a floating-point number. The format of the token should be that expected by the function **strtod** for a floating-point number that uses exponential notation. The corresponding argument should point to a **double**.

E Same as **e**. Under earlier releases of **Let's C**, this conversion specifier converted the token to a **double**. This change has been made to conform to the ANSI Standard, and may require that some code be rewritten.

f Convert the token to a floating-point number. The format of the token should be that expected by the function **strtod** for a floating-point number that uses decimal notation. The corresponding argument should point to a **double**.

F Same as **f**.

g Convert the token to a floating-point number. The format of the token should of that expected by the function **strtod** for a floating-point number that uses either exponential notation or decimal notation. The corresponding argument should point to a **double**.

G Same as **g**.

i Convert the token to a decimal integer. The format should be equivalent to that expected by the function **strtol** with a base argument of zero. The corresponding argument should point to an **int**.

n Do not read any text. Write into the corresponding argument the number of characters that **scanf** has read up to this point. The corresponding argument should point to an **int**.

o Convert the token to an octal integer. The format should be equivalent to that expected by the function **strtol** with a base argument of eight. The corresponding argument should point to an **int**.

O Same as **o**, except that the corresponding argument points to a **long**. This conversion specifier is not described in the ANSI Standard, and using it means that your program will not comply strictly with the Standard.

p Pointer format: read a sequence of implementation-defined characters, convert them in an implementation-defined way, and write them in an implementation-defined manner. The vagueness of this description is unavoidable, because the pointer format will vary between machines, and even on the same machine. The corresponding argument should point to a **void ***. The sequence of characters recognized should be identical with that written by **printf**'s **p** conversion specifier.

s Read a string of non-white space characters, copy them into the area pointed to by the corresponding argument, and append a null character to the end. The argument should be of type **char ***, and should point to enough allocated memory to hold the string being read plus its terminating null character.

u Convert the token to an unsigned integer. The format should be equivalent to that expected by the function **strtoul** with a base argument of ten. See **strtoul** for more information. The corresponding argument should point to an **unsigned int**.

x Convert the token from hexadecimal notation to a signed integer. The format should be equivalent to that expected by the function **strtol** with a base argument of 16. See **strtol** for more information. The corresponding argument should point to an **unsigned int**.

X Same as **x**. In previous releases of **Let's C**, the modifier **X** meant that the corresponding argument pointed to a **long** instead of an **int**. This has been changed to conform to the ANSI Standard, and may require that some code be rewritten.

% Match a single percent sign '%'. Make no conversion or assignment.

[/] Scan a *scanset*, which is a set of characters enclosed by brackets. A character that matches any member of the scanset is copied into the area pointed to by the corresponding argument, which should be a **char *** that points to enough allocated memory to hold the maximum number of characters that may be copied, plus the concluding null character. Appending a circumflex '^' to the scanset tells **scanf** to copy every character that does *not* match a member of the scanset (i.e., *complements* the scanset). If the format string begins with ']' or '^]', then ']' is included in the scanset, and the set specifier is terminated by the next ']' in the format string. If a hyphen appears within the scanset, the behavior is implementation-defined; often, it indicates a range of characters, as in **[a-z]**.

For example, passing the string **hello, world** to

```
char array[50];
scanf("[^abcd]", array);
```

writes the string **hello, worl** into **array**.

Cross-references

Standard, §4.9.6.4
The C Programming Language, ed. 2, p. 246

See Also

fscanf, printf, sscanf, STDIO

Notes

scanf will read up to, but not through, a newline character. The newline remains in the standard input device's buffer until you dispose of it. Programmers have been known to forget to empty the buffer before calling **scanf** a second time, which leads to unexpected results.

Experience has shown that **scanf** should not be used directly to obtain a string from the keyboard: use **gets** to obtain the string, and **sscanf** to format it.

The character that **scanf** recognizes as representing the decimal point is affected by the program's locale, as set by the function **setlocale**. For more information, see **localization**.

scope — Definition

The term *scope* describes the portion of the program in which a given identifier is recognized, or *visible*. Scope is similar to, but not identical to, linkage. Linkage refers to whether an identifier can be joined, or *linked*, across files. Scope refers to the portion of a program that can recognize an identifier.

There are four varieties of scope: *block*, *file*, *function*, and *function prototype*.

An identifier with block scope is visible only within the block of code where it is declared. When the program reaches the '}' that ends that block of code, then the identifier is no longer visible, and so no longer "within scope".

An identifier with file scope is visible throughout the translation unit within which it is declared. The only identifiers that have file scope are those that are declared globally, i.e., that are declared outside the braces that enclose any function. If a function in one file uses an identifier that is defined in another file, it must mark that identifier as being external, by using the storage-class specifier **extern**.

An identifier with function scope is visible throughout a function, no matter where in the function it is declared. A label is the only variety of identifier that has function scope.

An identifier with function-prototype scope is visible only within the function prototype where it is declared. For example, consider the following function prototype:

```
void va_end(va_list listptr);
```

The identifier **listptr** has function-prototype scope. It is recognized only within that prototype, and is used only for purposes of documentation.

If an identifier is redeclared but is within an enclosing scope, it "hides" the outermost identifier and renders it inaccessible. This condition is called "information hiding", and it holds true as long as the inner declaration is within scope.

Example

The following program demonstrates scope, and shows how to hide information.

```
/* global i */
int i = 13;

void
function1(void)
{
	/* local i; hides global i */
	int i = 23;

	for(;;) {
		/* block-scope i; hides local and global i's */
		int i = 33;
		/* print block-scope i */
		printf ("block-scope i: %d\n", i);
		break;
	}
	/* block-scope i has disappeared; print local i */
	printf ("local i: %d\n", i);
}

void
function2(void)
{
	/* local i has disappeared; print global i */
	printf("global i: %d\n", i);
}

main(void)
{
	function1();
	function2();
	return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §3.1.2.1
The C Programming Language, ed. 2, p. 227

See Also

extern, identifiers, storage duration

Notes

If an identifier is declared both within a block and with the storage-class identifier **extern**, it has block scope. An external declaration made within one block of code is not available outside that block. If an identifier that is declared external within one block is referenced within another, behavior is undefined.

A common extension to C automatically promotes to file scope all external identifiers that are declared within a block. Under such implementations, the following will work correctly:

```
/* non-ANSI code! */
function1()
{
      extern float example();
         . . .
}

function2()
{
      float variable;
        . . .
      variable = example();
        . . .
}
```

Under the Standard, however, this code will not work correctly: the declaration of the function **example** has block scope; therefore, it cannot be seen in **function2**. In **function2**, therefore, the translator properly assumes that **example** returns an **int**. The **float** that **example** actually returns is altered, causing undefined behavior. ANSI C causes this code to behave differently than expected, and an implementation may not issue a warning message. This is a quiet change that may break existing code.

sequence point — Definition

A sequence point is any point in a program where all side effects are resolved. At every sequence point, the environment of the actual machine must match that of the abstract machine. That is, whatever optimizations or short-cuts an implementation may take, at every sequence point it must be as if the machine executed every instruction as it appeared literally in the program. Sequence points cause the program's actual behavior to be synchronized with the abstract behavior that the source code describes.

The sequence points are as follows:

- When all arguments to a function call have been evaluated.
- When the *first* operand of the following operators has been evaluated: logical AND '&&', logical OR '||', conditional '?', and comma ','.
- When a variable is initialized.
- When the controlling expression or expressions are evaluated for the following statements: **do**, **for**, **if**, **return**, **switch**, and **while**.

Cross-reference

Standard, §2.1.2.3

See Also

side effect, translation units

setbuf() — STDIO (libc)

Set alternative stream buffer
#include <stdio.h>
void setbuf(FILE **fp*, **char** **buffer***);**

When the functions **fopen** and **freopen** open a stream, they automatically establish a buffer for it. The buffer is **BUFSIZ** bytes long. **BUFSIZ** is a macro that is defined in the header **stdio.h**.

setbuf changes the buffer for the stream pointed to by *fp* from its default buffer to *buffer*. It sets *buffer* to be **BUFSIZ** bytes long. To create a buffer of a size other than **BUFSIZ**, use **setvbuf**.

You should use **setbuf** after *fp* has been opened, but before any data have been read from or written to it.

If *buffer* is set to NULL, then *fp* will be unbuffered. For example, the call

```
setbuf(stdout, NULL);
```

ensures that all output to the standard output stream is unbuffered.

Cross-references

Standard, §4.9.5.5
The C Programming Language, ed. 2, p. 243

See Also

BUFSIZ, fclose, fflush, freopen, setbuf, setvbuf, STDIO

setjmp() — Non-local jump (setjmp.h)

Save environment for non-local jump
#include <setjmp.h>
int setjmp(jmp_buf *environment***);**

setjmp copies the current environment into the array **jump_buf**. The environment can then be restored by a call to the function **longjmp**.

environment is of type **jmp_buf**, which is defined in the header **setjmp.h**. **Let's C** defines **jmp_buf** to be an array of 11 **long**s.

setjmp returns zero if it is called directly. When it returns after a call to **longjmp**, however, it returns **longjmp**'s argument *rval*. If *rval* is set to zero, then **setjmp** returns one. See **longjmp** and **non-local jumps** for more information.

Cross-references

Standard, §4.6.1.1
The C Programming Language, ed. 2, p. 254

See Also

longjmp, jmp_buf, non-local jumps

Notes

Many user-level routines cannot be interrupted and reentered safely. For that reason, improper use of **setjmp** and **longjmp** will result in the creation of mysterious and irreproducible bugs. The use of **longjmp** to exit interrupt, exception, or signal handlers is particularly hazardous.

setjmp must be used as the controlling operand in a **switch** statement, as the controlling expression in an **if** statement, or as an operand in an equality expression. Any other use generates undefined behavior.

To conform with the Standard, **setjmp** is implemented as a macro.

setjmp.h — Header

Declarations for non-local jump
#include <setjmp.h>

setjmp.h is the header that contains declarations for the elements that perform a non-local jump. It contains the prototype for the function **longjmp**, and it defines the macro **setjmp** and the type **jmp_buf**.

Cross-references

Standard, §4.6
The C Programming Language, ed. 2, p. 254

See Also

header, jmp_buf, longjmp, non-local jump, setjmp

setlocale() — Localization (libc)

Set or query a program's locale
#include <locale.h>
char *setlocale(int *portion***, const char ****locale***);**

setlocale is a function that lets you set all or a portion of the locale information used by your program or query for information about the current locale.

portion is the portion of the locale that you wish to set or query. The Standard defines a number of manifest constants for this purpose, as follows:

LC_ALL
Set or query all locale-specific information. Setting the locale affects all of the following locale categories.

LC_COLLATE
Set or query information that affects collating functions. This affects the operation of the functions **strcoll** and **strxfrm**.

LC_CTYPE
Set or query information about character handling. This affects he operation of all character-handling functions, except for **isdigit** and **isxdigit**. It also affects the operation of the functions that handle multibyte characters, i.e., **mblen**, **mbtowc**, **mbstowcs**, and **wcstombs**, **wctomb**.

LC_MONETARY
Set or query all monetary-specific information as used in the structure **lconv**, which is initialized by the function **localeconv**.

LC_NUMERIC
Set or query information for formatting numeric strings. This may change the decimal-point character used by string conversion functions and functions that perform formatted input and output. This may also affect the contents of the structure **lconv**.

LC_TIME
Set or query information for formatting time strings. This changes the operation of the function **strftime**.

Setting *locale* to NULL tells **setlocale** that you wish to query information about the current locale rather than set a new locale.

setlocale returns a pointer to a string that contains the information needed to set or examine the locale. For example, the call

```
setlocale(LC_TIME, "");
```

returns a string that can be used to modify the time and date functions to conform to the requirements of the native locale. **setlocale** returns NULL if it does not recognize either *portion* or *locale*.

Cross-reference

Standard, §4.4.1.1

See Also

lconv, localeconv, localization

Notes

The Standard's section on compliance states that any program that uses locale-specific information does not strictly comply with the Standard. Therefore, any program that uses a locale other than the **C** locale *cannot* be assumed to be portable to every environment for which a conforming implementation of C has been written. *Caveat utilitor.*

setvbuf() — STDIO (libc)

Set alternative stream buffer

#include <stdio.h>

int setvbuf(FILE **fp*, **char** **buffer*, **int** *mode*, **size_t** *size***);**

When the functions **fopen** and **freopen** open a stream, they automatically establish a buffer for it. The buffer is **BUFSIZ** bytes long. **BUFSIZ** is a macro that is defined in the header **stdio.h**.

setvbuf alters the buffer used with the stream pointed to by *fp* from its default buffer to *buffer*. Unlike the related function **setbuf**, it also allows you set the size of the new buffer as well as the form of buffering.

buffer is the address of the new buffer. *size* is its size, in bytes. *mode* is the manner in which you wish the stream to be buffered, as follows:

_IOFBF	Fully buffered
_IOLBF	Line-buffered
_IONBF	No buffering

These macros are defined in the header **stdio.h**. For more information on what these terms mean, see **buffering**.

You should call **setvbuf** after a stream has been opened but before any data have been written to or read from the stream. For example, the following give *fp* a 50-byte buffer that is line-buffered:

```
char buffer[50];
FILE *fp;

fopen(fp, "r");
setvbuf(fp, buffer, _IOLBF, sizeof(buffer));
```

On the other hand, the following turns off buffering for the standard output stream:

```
setvbuf(stdout, NULL, _IONBF, 0);
```

setvbuf returns zero if the new buffer could be established correctly. It returns a number other than zero if something went wrong or if an invalid parameter is given for *mode* or *size*.

Example

This example uses **setvbuf** to turn off buffering and echo.

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

```
main(void)
{
      int c;

      if(setvbuf(stdin,  NULL, _IONBF, 0))
            fprintf(stderr, "Couldn't turn off stdin buffer\n");

      if(setvbuf(stdout,  NULL, _IONBF, 0))
            fprintf(stderr, "Couldn't turn off stdout buffer\n");

      while((c = getchar()) != EOF)
            putchar(c);
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.9.5.6
The C Programming Language, ed. 2, p. 243

See Also

BUFSIZ, fclose, fflush, fopen, freopen, setbuf, STDIO

shellsort() — Extended function (libc)

Sort arrays in memory
void shellsort(char **data*, **short** *n*, **short** *size*, **short (****comp*)**());**

shellsort is a generalized algorithm for sorting arrays of data in primary memory. It uses D. L. Shell's sorting algorithm. **shellsort** works with a sequential array of memory called *data*, which is divided into *n* parts of *size* bytes each. In practice, *data* is usually an array of pointers or structures, and *size* is the **sizeof** the pointer or structure.

Each routine compares pairs of items and exchanges them as required. The user-supplied routine to which *comp* points performs the comparison. It is called repeatedly, as follows:

```
(*comp)(p1, p2)
char *p1, *p2;
```

Here, *p1* and *p2* each point to a block of *size* bytes in the *data* array. In practice, they are usually pointers to pointers or pointers to structures. The comparison routine must return a negative, zero, or positive result, depending on whether *p1* is less than, equal to, or greater than *p2*, respectively.

See Also

general utilities, qsort
The Art of Computer Programming, vol. 3, pp. 84*ff*, 114*ff*

Notes

shellsort differs from the sort function **qsort** in that it uses an iterative algorithm that does not require much stack.

short int — Type

A **short int** is a signed integral type. This type can be no smaller than a **char**, and no larger than an **int**.

A **short int** can encode any number between **SHRT_MIN** and **SHRT_MAX**. These are macros that are defined in the header **limits.h**. The former equals -32,767, and the latter +32,767.

The types **short**, **signed short**, and **signed short int** are all synonyms for **short int**.

Cross-references

Standard, §2.2.4.2, §3.1.2.5, §3.2.1.1, §3.5.2
The C Programming Language, ed. 2, p. 211

See Also

int, long int, types

side effect — Definition

A *side effect* is any change to the execution environment that is caused by the program that accesses a volatile object, modifies an object, modifies a file, or calls a function that performs any of these tasks. An expression may generate side effects; a void expression exists just for the side effects it generates.

Cross-references

Standard, §2.1.2.3
The C Programming Language, ed. 2, p. 53

See Also

Environment, sequence point, translation phase

sig_atomic_t — Type

Type that can be updated despite signals

sig_atomic_t is an integral data type that is defined in the header **signal.h**. It defines the type of "atomic" object that can be accessed properly even if an asynchronous interrupt occurs.

Cross-reference

Standard, §4.7.1

See Also

signal handling, signal.h, volatile

Notes

When declaring objects of this type, you should use the type qualifier **volatile**; for example:

```
volatile sig_atomic_t save_state;
```

The **volatile** declaration tells the translator to re-read the object's value from memory each time it is used in an expression. When the program says to store the object, it should be stored immediately.

signal() — Signal handling (libc)

Set processing for a signal
#include <signal.h>
void (*signal(int *signame*, **void (****function***)(int)))(int);**

signal is a function that tells the environment what to do when it detects a given interrupt, or "signal." *signame* names the signal to be handled, and *function* points to the signal handler (the function to be executed when *signame* is detected). *signame* may be generated by the environment itself (when it detects an error condition, for example), by the hardware (to indicate a bus error, timer event, or other hardware error condition), or by the program itself (by using the function **raise**).

If **signal** is successful, it returns a pointer to the function that the environment previously used to handle *signame*. If an error occurred, **signal** returns **SIG_ERR** and the global variable **errno** is set to an appropriate value. For a list of the signals recognized, see **signal handling**.

signal can establish the following ways of handling a *signame*:

1. If it sets *function* to **SIG_DFL**, it tells the environment to execute the default signal-handling function for *signame*.
2. Then, the equivalent of

```
(*function)(signame)
```

   is executed, where function is the user-defined function installed with **signal** to handle *signame*.
3. If it sets *function* to point to a user-defined function, then it tells the environment to execute that function when it detects *signame*.

If **signal** is used to establish a user-defined *function* for a particular signal, then the following occurs when that signal is detected:

1. The equivalent of

```
signal(signame, SIG_DFL);
```

   is executed. If *signame* is equivalent to **SIGILL** (which indicates that an illegal instruction has been found), then this step is optional, depending upon the implementation.
2. Then, the equivalent of

```
(*function)(signame)
```

   is executed, where *function* points to a user-defined function. Some signals are reset to **STD_DFL**, some are not. The exception handler should be reset by another call to **signal** if subsequent signals are expected for that condition.
3. *function* can terminate either by returning to the calling function, or by calling **abort**, **exit**, or **longjmp**. If *function* returns and *signame* indicates that a computational exception had occurred (e.g., division by zero), then the behavior is undefined. Otherwise, the program which responded to the signal will continue to execute.

Cross-references

Standard, §4.7.1.1
The C Programming Language, ed. 2, p. 255

See Also

raise, signal handling, signal.h

Notes

The signal handler pointed to by *function* should not be another library function. Also, the signal handler should not attempt to modify external data other than those declared as type **volatile sig_atomic_t**.

signal.h — Header

Signal-handling routines
#include <signal.h>

signal.h is the header that defines or declares all elements used to handle asynchronous interrupts, or *signals*.

Signals vary from environment to environment. Therefore, the contents of **signal.h** will also vary greatly from environment to environment, and from implementation to implementation. The Standard mandates that it define the following elements to create a skeletal, portable suite of signal-handling routines:

Type

sig_atomic_t	Type that can be accessed atomically
SIG_DFL	Default signal-handling indicator
SIG_ERR	Indicate error in setting a signal
SIG_IGN	Indicate ignore a signal
SIGABRT	Abort signal
SIGFPE	Erroneous arithmetic signal
SIGILL	Illegal instruction
SIGINT	Interrupt signal
SIGSEGV	Invalid access to storage signal
SIGTERM	Program termination signal

Functions

raise	Generate a signal
signal	Set processing for a signal

Cross-references

Standard, §4.7
The C Programming Language, ed. 2, p. 255

See Also

signal handling

signal handling — Overview

#include <signal.h>

A *signal* is an asynchronous interrupt in a program. Its use allows a program to be notified of and react to external conditions, such as errors that would otherwise force it either to abort or to continue despite erroneous conditions.

To respond to a signal, a program uses a *signal handler*, which is a function that performs the actions appropriate to a given signal. A signal handler usually is installed early in a program. It is invoked either when the condition arises for which the signal handler was installed, or when the program uses the function **raise** to raise a signal explicitly. A signal handler can be thought of as a "daemon," or a process that lives in the background and waits for the right conditions to occur for it to spring to life. Once the signal has been handled, the program may continue to execute.

Every conforming implementation of C must include at least a skeletal facility for handling signals. The Standard describes two functions: **raise**, which generates (or "raises") a signal; and **signal**, which tells the environment what function to execute in response to a given signal.

The suite of signals that can be handled varies from environment to environment. At a minimum, the following signals must be recognized:

SIGABRT	Abort
SIGFPE	Erroneous arithmetic
SIGILL	Illegal instruction
SIGINT	Interrupt
SIGSEGV	Invalid access to storage
SIGTERM	Program termination request

All of these are positive integral expressions. An implementation is obliged to respond only if one of these signals is raised explicitly via the function **raise**. This limitation is imposed because in some environments it may be impossible for an implementation to "sense" the presence of such conditions.

signal tells the environment which function to execute in response to a signal by passing it a pointer to that function. The Standard describes three macros that expand to constant expressions that point to functions, as follows:

SIG_DFL	Default signal-handling indicator
SIG_ERR	Indicate error in setting a signal
SIG_IGN	Indicate ignore a signal

The Standard describes a new data type, called **sig_atomic_t**. An object of this type can be updated or read correctly, even if a signal occurs while it is being updated or read. Accesses to objects of this type are atomic, i.e., uninterruptable.

All of the above are defined or declared in the header **signal.h**.

Cross-references

Standard, §4.7, §2.2.3
The C Programming Language, ed. 2, p. 255

See Also

Library, sequence points, signal.h, signals/interrupts

Notes

The name *signal* is derived from the electrical model of having a wire connected to the central processing unit for an interrupt. When the level on the wire would rise, an interrupt would be generated and the central processing unit would service the device that "raised" its "signal."

signals/interrupts — Definition

The Standard mandates the following restrictions upon the manner in which functions are implemented. First, a signal must be able to interrupt a function at any time. Second, a signal handler must be able to call a function without affecting the value of any object with automatic duration created by any earlier invocation of the function. Third, the function image (that is, the set of instructions that constitutes the executable image of the function) cannot be altered in any way as it is executed. All variables must be kept outside of the function image.

MS-DOS Interrupts

MS-DOS makes available to the programmer a series of interrupts that can be used to perform all manner of useful tasks. These interrupts and their functions can be accessed directly through the C function **intcall**.

The header **dos.h** defines a set of manifest constants that use most MS-DOS interrupts. The following table lists these constants, the interrupt and function number they define, and gives a brief description of what each does. Some constants combine two interrupts to form one function. For example, **CLRIN** combines interrupts 0x0C and 0x01.

Interrupt 10 (text mode)

GCDM	0x0F00	Get current display mode
IWDOWN	0x0700	Initialize window or scroll window down
IWUP	0x0600	Initialize window or scroll window up
RACCUR	0x0800	Read attribute & character at cursor
RDCP	0x0300	Read cursor positon
RGRPIX	0x0D00	Read graphics pixel
RLPP	0x0400	Read light pen position
SDP	0x0500	Select display page
SETCLR	0x0B00	Set color palette
SETCP	0x0200	Set cursor position

SETCT	0x0100	Set cursor type
SPALREG	0x1000	Set palette registers
WACCUR	0x0900	Write attribute and character at cursor
WCONLY	0x0A00	Write character only at cursor
WGRPIX	0x0C00	Write graphics pixel
WSTRING	0x1300	Write string (AT only)
WTELE	0x0E00	Write text in teletype mode

Interrupt 10 (graphics mode)

VM1620JR	0x0008	160x200 16-color graphics (*PCjr*)
VM3220C	0x0004	320x200 four-color graphics
VM3220CB	0x0005	320x200 four-color graphics color burst off
VM3220EG	0x000D	320x200 16-color graphics (EGA)
VM3220JR	0x0009	320x200 16-color graphics (*PCjr*)
VM4025BW	0x0000	40x25 black & white text, color ad.
VM4025C	0x0001	40x25 color text
VM64202	0x0006	640x200 two-color graphics
VM6420EG	0x000E	640x200 16-color graphics (EGA)
VM6420JR	0x000A	640x200 16-color graphics (*PCjr*)
VM64354E	0x0010	640x350 four- or 16-color graphics (EGA)RAM
VM6435EG	0x000F	640x350 monochrome graphics (EGA)
VM8025BW	0x0002	80x25 black & white text
VM8025C	0x0003	80x25 color text
VMMONOAD	0x0007	Monochome adapter text display

Interrupt 13

FORMDT	0x0500	Format disk track
GFDSS	0x0100	Get disk system status
RDFD	0x0200	Read disk
RSTFDS	0x0000	Reset disk system
VERDS	0x0400	Verify disk sectors
WRTDSK	0x0300	Write to disk

Interrupt 14

ITCOMP	0x0000	Initialize communications port
RCCOMP	0x0200	Read character from communications port
WCCOMP	0x0100	Write character to communications port

Interrupt 16

RCKEYB	0x0000	Read character from keyboard
RKEYST	0x0100	Read keyboard status
RTKEYF	0x0200	Return keyboard flags

Interrupt 17

INITPP	0x0100	Initialize printer port
PRNSRQ	0x0200	Request printer status
WCPRN	0x0000	Write character to printer port

Interrupt 21

ALLOC	0x4800	Allocate memory
BUFCON	0x0A00	Read console, buffered
CHDIR	0x3B00	Change current directory
CHMOD	0x4300	Change file mode

CLOSEF*	0x1000	Close a file
CLOSEH	0x3E00	Close a file
CLR_E	0x0C08	Clear console, accept input without echo
CLRBUF	0x0C0A	Clear console, accept buffered input
CLRIN	0x0C01	Clear console, echo console input
CLRDIO	0x0C06	Clear console, perform direct console I/O
CLRRAW	0x0C07	Clear console, accept raw input
CONSTAT	0x0B00	Return console/s status
CREATH	0x3C00	Create a file
CTLBCHK	0x3300	Get/set Ctrl-Break flag
DELETE	0x4100	Delete a file
DELETEF*	0x1300	Delete a file
DUPH	0x4500	Duplicate a file handle
EXEC	0x4B00	Load or execute a program
FDUPH	0x4600	Force a duplicate of handle
FFIRST*	0x1100	Search for first match
FNEXT*	0x1200	Search for next match
FREE	0x4900	Free allocated memory
GETALTI	0x1B00	Get allocation table information
GETCDI	0x3800	Get country-dependent information
GETCDIR	0x4700	Get current directory
GETDATE	0x2A00	Get date
GETDISK	0x1900	Get default disk drive
GETDTA	0x2F00	Get address of disk transfer area
GETFREE	0x3600	Get free disk space
GETTIME	0x2C00	Get time
GETVEC	0x3500	Get interrupt vector
GETVER	0x3000	Get MS-DOS version number
GETVST	0x5400	Get verify state
GSDT	0x5700	Get/set a file's date and time
IOCTLH	0x4400	I/O control for devices
LSEEKH	0x4200	Move file read/write pointer
MAKEF*	0x1600	Create or truncate a file
MKDIR	0x3900	Create a sub-directory
NEXIT	0x4C00	Terminate a process
NFFIRST	0x4E00	Search for first match
NFNEXT	0x4F00	Search for next match
OPENF*	0x0F00	Open a file
OPENH	0x3D00	Open a file
PROGSEG	0x2600	Create program segment
PUTSTR	0x0900	Output string, terminated with '$'
READB*	0x2700	Block read, random
READH	0x3F00	Read from a file or device
READR*	0x2100	Read, random
READS*	0x1400	Read sequential
RENAME	0x5600	Rename a file
RENAMEF*	0x1700	Rename a file
RESDSK	0x0D00	Reset disk system
RMDIR	0x3A00	Remove a sub-directory
SELDSK	0x0E00	Set default disk drive
SETBLK	0x4A00	Modify allocated memory blocks
SETDATE	0x2B00	Set date
SETDMAO	0x1A00	Set disk transfer address
SETINT	0x2500	Set interrupt vector

SETRREC*	0x2400	Set random record number
SETTIME	0x2D00	Set time
SIZEF*	0x2300	Compute size of file
TERMRES	0x3100	Terminate and remain resident
VERIFY	0x2E00	Disk write verification
WAIT	0x4D00	Get return code of subprocess
WRITEB*	0x2800	Block write, random
WRITEH	0x4000	Write to a file or device
WRITER*	0x2200	Write, random
WRITES*	0x1500	Write sequential

The interrupts marked with an asterisk '*' use the file control block. These functions, in general, have been replaced by other, similarly named functions that are easier to use. The file control block is a structure, defined as follows:

```
typedef struct fcb_t {
      unsigned char f_drive;          /* drive code (A=1, etc.) */
      char f_name[8],                 /* file name */
           f_ext[3];                  /* file suffix */
      unsigned short f_block;         /* current block
                                         (=128 records) */
      unsigned short f_recsz;         /* record size in bytes
                                         (=1) */
      unsigned long f_size;           /* file size, bytes
                                         (system) */
      unsigned int f_date;            /* modif. date (system) */
      char f_sys[10];                 /* for system use */
      unsigned char f_rec;            /* current record in block */
      unsigned long f_seek;           /* random record position */
} fcb_t;
```

Calling DOS Interrupts

Let's C offers two ways to use MS-DOS interrupts in your C programs.

The first is through the function **intcall**. **intcall** gives a convenient way to call an MS-DOS interrupt directly from a C program. For more information and examples on how to use this function, see the entry for **intcall**.

The other method is by using the programs **int.c** and **intdis.m**, whose source code is included with **Let's C**. Unlike **intcall**, which is a tool for calling MS-DOS interrupts, these programs allow i8086 interrupts to call you. Thus, they are a tool for building interrupt handlers. They also demonstrate how to combine a C program with one written in assembly language.

The suffix '.m' is unique to Mark Williams Company. It is used with a file of assembly language that is first treated by **cpp**, a command that invokes the C preprocessor. Thus, a '.m' file can contain conditionalized code, manifest constants, and all other commands that are recognized by the preprocessor. To compile such a file, assemble it through the **cc** command. For example, to assemble **foo.m**, use the command:

```
cc foo.m
```

cc will automatically call **cpp**, and pass its output to the assembler **as**. The entry for **as** presents an example of a **.m** program. See the entry on **larges.h** for more information on the **.m** format in general.

Example

The following example, called **example.c**, uses routines in **int.c** and **intdis.m** to call several MS-DOS interrupts. You should enter it into the directory where you have stored **int.c** and **intdis.m**, and compile it with the following command line:

```
cc example.c int.c intdis.m
```

This program works in both LARGE and SMALL model. Compile it with the command line

```
cc -VLARGE example.c int.c intdis.m
```

to create a LARGE-model executable.

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

#define INT_BREAK 0x1B        /* keybd ctrl-break int */
#define INT_TICK 0x1C         /* system timer tick int */
#define STACKSIZE 0x100       /* small stack for locals */

int breakid;
int timerid;

#define TRUE 1
#define FALSE 0

int breakflag = FALSE;
int timerflag = FALSE;

/*
 *    Service routine for the Ctrl-Break Interrupt (0x1B).
 *    Simply sets the breakflag to TRUE.
 */

breaktrp(void)
{
      breakflag = TRUE;
      return(0);
}

/*
 *    Service routine for Timer-Tick Interrupt (0x1C).
 *    This comes from the 8253-5 Programmable Interval Timer
 *    at a rate of 18.2 Hz.  Thus every 91
 *    ( = 18.2 * 5 ) interrupts
 *    or 5 seconds, set the timerflag to TRUE.
 */

timertrp(void)
{
      static counter = 0;

      if(++counter == 91) {
            timerflag = TRUE;
            counter = 0;
      }

      /* Link in case interrupt 0x1C did something already */
      return(1);
}

void
fatal(char *message)
{
      fprintf(stderr, "%s\n", message);
      exit(EXIT_FAILURE);
}
```

```
main(void)
{
      int breaktrp();
      int timertrp();

      if ( (breakid=setint(INT_BREAK, breaktrp,
            STACKSIZE, 1)) == -1 )
            fatal("Error setting ctrl-break interrupt.");
      printf("Ctrl-Break Interrupt Set.\n");

      if((timerid=setint(INT_TICK, timertrp,
            STACKSIZE, 1)) == -1 )
            fatal("Error setting timer-tick interrupt.");
      printf("Timer-Tick Interrupt Set.0);

      for (;;) {
            if(breakflag == TRUE)
                  break;
            if(timerflag == FALSE)
                  continue;
            printf("Another 5 sec gone.\n");
            timerflag = FALSE;
      }
      printf("Got the Ctrl-Break Key.\n");

      if(clearint(breakid) != 0)
            fatal("Unable to reset interrupt.");
      printf("Ctrl-Break interrupt reset.\n");

      if ( clearint(timerid) != 0 ) {
            fatal("Unable to reset Timer-Tick Interrupt.");
      printf("Timer-Tick interrupt reset.\n");

      return EXIT_SUCCESS;
}
```

Cross-references

Standard, §2.2.3 *Advanced MS-DOS*, pp 208*ff*, 272*ff*

See Also

Environment, signal handling

signed — Definition

The modifier **signed** indicates that a data type can contain both positive and negative values. In some representations, the sign of a signed object is indicated by a bit set aside for the purpose. For this reason, a signed object can encode an absolute value only half that of its unsigned counterpart.

The four integral data types can be marked as signed: **char**, **short int**, **int**, and **long int**.

The implementation defines whether a **char** is signed or unsigned by default. The Standard describes the types **signed char** and **unsigned char**. These let the programmer use the type of **char** other than that supplied by the implementation. **short int**, **int**, and **long int** are signed by default. The declarations **signed short int**, **signed int**, and **signed long int** were created for the sake of symmetry.

For information about converting one type of integer to another, see **integral types**.

If **signed** is used by itself, it is a synonym for **int**.

Cross-references

Standard, §3.1.2.5, §3.2.1.2
The C Programming Language, ed. 2, p. 211

See Also

types, unsigned

signed char — Type

A **signed char** is a type that has the same size and the same alignment requirements as a plain **char**. The Standard created this type for implementations whose **char** type is unsigned by default.

A **signed char** can encode values from **SCHAR_MIN** to **SCHAR_MAX**. These are macros that are defined in the header **limits.h**. The former is set to -127, and the latter to +127.

Cross-references

Standard, §2.2.4.2, §3.1.2.5, §3.5.2
The C Programming Language, ed. 2, p. 44

See Also

char, types, unsigned char

sin() — Mathematics (libm)

Calculate sine
#include <math.h>
double sin(double *radian***);**

sin calculates and returns the sine of its argument *radian*, which must be in radian measure.

Example

This example verifies the identity **sin(2*x) == 2*sin(x)*cos(x)** over a range of values. Then, it scans the range of the worst error in smaller and smaller increments, until the precision of the floating point will not allow any more.

```
#include <float.h>
#include <math.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#define PI 0.31415926535897932e+01

main(void)
{
      int ct;
      double a, e, i, worstp;
      double worste=0.0;
      double f=-PI;
```

```
	printf("Verify sin(2*x) == 2*sin(x)*cos(x)\n");
	for(i = (PI / 100.0); (f + i) > f; i *= 0.01) {
		for(ct = 200, a = f; --ct; a += i) {
			e = fabs(sin(a+a)-(2.0*sin(a)*cos(a)));
			if(e > worste) {
				worste = e;
				worstp = a;
			}
		}
		f = worstp - i;
	}

	printf("Worst error %.17e at %.17e\n", worste, worstp);
	printf("sin(2x)=%.17e 2*sin(x)*cos(x)=%.17e\n",
		f=sin(worstp+worstp), 2.0*sin(worstp)*cos(worstp));
	printf("Epsilon is %.17e\n", fabs(f) * DBL_EPSILON);
	return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.5.2.6
The C Programming Language, ed. 2, p. 251

See Also

acos, asin, atan, atan2, cos, mathematics, tan

sinh() — Mathematics (libm)

Calculate hyperbolic sine
#include <math.h>
double sinh(double *value***);**

sinh calculates and returns the hyperbolic sine of *value*. A range error will occur if the argument is too large.

Cross-references

Standard, §4.5.3.2
The C Programming Language, ed. 2, p. 251

See Also

cosh, mathematics, tanh

size — Command

Print the size of an object module
size *file…*

size prints the size of each segment of each given *file*, which must be a relocatable object module or an executable file. The total size is given in decimal, and the size of each segment is given in both decimal and hexadecimal. All sizes are in bytes.

When it is used to size an executable file, **size** prints the size of the code segment and the data segment separately (in LARGE model), or the code segment plus the data segment (in SMALL model). Thus, **size** can help you to tell a SMALL-model program from one in LARGE model.

See Also

cc, commands, cpp, nm, strip

sizeof — C keyword

The operator **sizeof** yields the size of its argument, in bytes. Its argument can be the name of a type, an array, a function, a structure, or an expression that yields an object.

When the name of a type is used as the operand to **sizeof**, it must be enclosed within parentheses. If any of the types **char**, **signed char**, or **unsigned char** are used as the argument to **sizeof**, the result by definition is always one. When any complete type is used (i.e., a type whose size is known by the translator), the result is the size of that type, in bytes. For example,

```
sizeof (long double);
```

returns the size of a **long double** in bytes.

If **sizeof** is given the name of an array, it returns the size of the array. For example, the code

```
int example[5];
   . . . /* example[] is filled with some things */
sizeof example[] / sizeof int;
```

yields the number of members in **example[]**.

When **sizeof** is given the name of a structure or a **union**, it returns the size of that object, including padding used to align the objects within the structure, if any. This is especially useful when allocating memory for a linked list; for example:

```
struct example {
      int member1;
      example *member2;
};
struct example *variable;
variable=(struct example *)malloc(sizeof(struct example));
```

If **sizeof** is used to measure either a function or an array that has been passed as an argument to a function, it returns the size of a *pointer* to the appropriate object. This is because when an array name or function name is passed as an argument to a function, it is converted to a pointer. See **function definition** for more information.

sizeof always returns an object of type **size_t**; this type is defined in the header **stddef.h**. It is intended to be an unsigned integral type.

sizeof must not be used with a function, with an object whose type is incomplete, or a bit-field.

Example

For an example of using this operator in a program, see **bsearch**.

Cross-references

Standard, §3.3.3.4
The C Programming Language, ed. 2, p. 204

See Also

expressions, operators, size_t

SMALL model — Technical information

Intel single-segment memory model

The i8086/88 microprocessor uses a *segmented architecture*. This means that the memory is divided into segments of 64 kilobytes each; no program or data element can exceed that limit.

Intel Corporation has devised a number of memory models for handling segmented memory.

Let's C implements the two most useful of these: SMALL model and LARGE model.

SMALL model C programs use 16-bit pointers and NEAR calls. Because a 16-bit pointer can address 65,536 bytes (64 kilobytes) of memory, SMALL model programs are limited to 64 kilobytes (one segment) of code and 64 kilobytes of data.

The SMALL-model pointer consists only of the *offset* within a given segment, and does not include the *segment* itself. If you use a function that requires the full offset/segment pair, e.g., **_copy**, **peek**, or **poke**, you can supply the missing segment either by reading the contents of the DS segment register with the function **dsreg**, or by using the macro **PTR**. See the entries for **dsreg** and **PTR** for more information.

Note, too, that the SMALL-pointer is the same length as an **int**. This allows a programmer to use these data types interchangably. Most often, this happens when a programmer fails to declare properly a function that returns a pointer, so that the function is implicitly declared by the compiler as returning an **int**. Programs with this error will run correctly when compiled into SMALL model, but will fail to work when compiled into LARGE model. See the entry on **pun** for more information.

When **Let's C** compiles a program with the **-VSMALL** option, the resulting object module follows the rules of the *SMALL model*. This is the default setting for the compiler.

See Also

i8086 support, LARGE model, model, pun, technical information

source file — Definition

A *source file* is any file of C source text.

Cross-reference

Standard, §2.1.1.1

See Also

Environment, translation unit

sprintf() — STDIO (libc)

Print formatted text into a string
#include <stdio.h>
int sprintf(char ***string*, **const char ***format*, **...);

sprintf constructs a formatted string in the area pointed to by *string*, and appends a null character onto the end of what it constructs. It translates integers, floating-point numbers, and strings into a variety of text formats.

format points to a string that can contain text, character constants, and one or more *conversion specifications*. A conversion specification describes how to convert a particular data type into text. Each conversion specification is introduced with the percent sign '%'. (To print a literal percent sign, use the escape sequence "%%".) See **printf** for further discussion of the conversion specification, and for a table of the type specifiers that can be used with **sprintf**.

After *format* can come one or more arguments. There should be one argument for each conversion specification in *format*. The argument should be of the type appropriate to the conversion specification. For example, if *format* contains conversion specifications for an **int**, a **long**, and a string, then *format* should be followed by three arguments, respectively, an **int**, a **long**, and a **char** *.

If there are fewer arguments than conversion specifications, then **sprintf**'s behavior is undefined. If there are more, then every argument without a corresponding conversion specification is evaluated and then ignored. If an argument is not of the same type as its corresponding conversion specifier, then the behavior of **sprintf** is undefined. Thus, presenting an **int** where **sprintf** expects a **char** * may generate unwelcome results.

sprintf returns the number of characters written into *string*, not counting the terminating null character.

Cross-references

Standard, §4.9.6.5
The C Programming Language, ed. 2, p. 245

See Also

fprintf, printf, STDIO, vfprintf, vprintf, vsprintf

Notes

string must point to enough allocated memory to hold the string **sprintf** constructs, or you may overwrite unallocated memory.

The character that **sprintf** uses to represent the decimal point is affected by the program's locale, as set by the function **setlocale**. For more information, see **localization**.

Because the **printf** routines that print floating-point numbers are quite large, they are included only optionally. If you wish to have **printf** print **float**s or **double**s, you must compile your program with the **-f** option to the **cc** command. See **cc** for more details.

sqrt() — Mathematics (libm)

Calculate the square root of a number
#include <math.h>
double sqrt(double *z***);**

sqrt calculates and returns the square root of *z*.

Example

This example calculates the time an object takes to fall to the ground at sea level. It ignores air friction and the inverse square law.

```
#include <errno.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

double
fallingTime(double meters)
{
      double time;

      errno = 0;
      time = sqrt(meters * 2 / 9.8);
      /*
       * it would be simpler to test for (meters < 0) first,
       * but this way shows how sqrt() sets errno
       */
      if(errno) {
            printf("Sorry, but you can't fall up\n");
            return(HUGE_VAL);
      }
      return(time);
}
```

```
main(void)
{
	for(;;) {
		char buf[80];
		double height;

		printf("Enter height in meters ");
		fflush(stdout);
		if(gets(buf) == NULL || !strcmp(buf, "quit"))
			break;

		errno = 0;
		height = strtod(buf, (char **)NULL);

		if(errno) {
			printf("%s: invalid floating-point number\n");
			continue;
		}

		printf("It takes %3.2f sec. to fall %3.2f meters\n",
			fallingTime(height), height);

	}

	return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.5.5.2
The C Programming Language, ed. 2, p. 251

See Also

domain error, mathematics, pow

Notes

If *z* is negative, a domain error occurs.

srand() — General utility (libc)

Seed pseudo-random number generator
#include <stdlib>
void srand(unsigned int *seed***);**

srand uses *seed* to initialize the sequence of pseudo-random numbers returned by **rand**. Different values of *seed* produce different sequences.

Example

This example uses the random-number generator to encrypt or decrypt a file. This example is for illustration only. Do *not* use it if any serious attack is expected. This example also demonstrates a simple form of hashing.

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

/* Ask for a string and echo it. */
char *
ask(char *msg)
{
	static char reply[80];
```

```
	printf("Enter %s ", msg);
	fflush(stdout);

	if(gets(reply) == NULL)
		exit(EXIT_SUCCESS);
	return(reply);
}

main(void)
{
	register char *kp;
	register int c, seed;
	FILE *ifp, *ofp;

	if((ifp = fopen(ask("input filename"), "rb")) == NULL)
		exit(EXIT_FAILURE);
	if((ofp = fopen(ask("output filename"), "wb")) == NULL)
		exit(EXIT_FAILURE);

	/* hash encryption key into an int */
	seed = 0;
	for(kp = ask("encryption key"); c = *kp++; ) {
		/* don't lose any bits */
		if((seed <<= 1) < 0)
			/* a number picked at random */
			seed ^= 0xE51B;
		seed ^= c;
	}

	/* initialize random-number stream */
	srand(seed);

	while((c = fgetc(ifp)) != EOF)
		/*
		 * Use only the high byte of rand;
		 * its low-order bits are very non-random
		 */
		fputc(c ^ (rand() >> 8), ofp);

	return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.10.2.2
The C Programming Language, ed. 2, p. 252

See Also

general utilities, rand

sscanf() — STDIO (libc)

Read and interpret text from a string
#include <stdio.h>
int sscanf(const char **string*, **const char** **format*, **...);**

sscanf reads characters from *string* and uses the string pointed to by *format* to interpret what it has read into the appropriate type of data. *format* points to a string that contains one or more conversion specifications, each of which is introduced with the percent sign '%'. For a table of the conversion specifiers that can be used with **sscanf**, see **scanf**.

After *format* can come one or more arguments. There should be one argument for each conversion specification in *format*, and the argument should point to a data element of the type appropriate to the conversion specification. For example, if *format* contains conversion specifications for an **int**, a **long**, and a string, then *format* should be followed by three arguments, pointing, respectively, to an **int**, a **long**, and an array of **char**s.

If there are fewer arguments than conversion specifications, then **sscanf**'s behavior is undefined. If there are more, then every argument without a corresponding conversion specification is evaluated and then ignored. If an argument is not of the same type as its corresponding type specification, then **sscanf** returns.

sscanf returns the number of input elements it scanned and formatted. If an error occurs while **sscanf** is reading its input, it returns **EOF**.

Example

This example reads a list of hexadecimal numbers from the standard input and adds them.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

main(void)
{
	long h[5], total;
	char buf[80];
	int  count, i;

	printf("Enter a list of up to five hex numbers or quit\n");
	while(gets(buf) != NULL) {
		if(!strcmp("quit", buf))
			break;

		count = sscanf(buf, "%lx %lx %lx %lx %lx",
			h, h+1, h+2, h+3, h+4);

		for(i = total = 0; i < count; i++)
			total += h[i];
		printf("Total 0x%lx %ld\n", total, total);
	}

	return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.9.6.6
The C Programming Language, ed. 2, p. 246

See Also

fscanf, printf, STDIO, scanf

Notes

sscanf is best used to read data you are certain are in the correct format, such as data previously written with **sprintf**.

The character that **sscanf** recognizes as representing the decimal point is affected by the program's locale, as set by the function **setlocale**. For more information, see **localization**.

stack — Definition

The **stack** is the segment of memory that holds function arguments, local variables, function return addresses, and stack frame linkage information.

If your program uses recursive algorithms, or declares large amounts of automatic data, or simply contains many levels of functions calls, the stack may “overflow”, and overwrite the program data.

By default, **Let’s C** sets the default stack size to 2,048 bytes (two kilobytes). To increase the amount of stack available to your program, use the **-ys** option to the **cc** command. For example, to give the program **foo.c** 10,000 bytes of stack, use the following **cc** command:

```
cc -ys10000 foo.c
```

See Also

cc, Definitions

Standard — Overview

The *Standard* is the document written by the American National Standards Institute committee X3J11 to describe the programming language C. It is based on the following documents:

- Kernighan, B. W., Ritchie, D. M.: *The C Programming Language.* Englewood Cliffs, NJ: Prentice-Hall Inc., 1978. The Standard bases its description of C syntax upon Appendix A of this book.
- /usr/group Standard Committee: *1984 /usr/group Standard.* Santa Clara, Calif.: /usr/group, 1984. This document was the basis for the Standard’s description of the C library.
- *American National Dictionary for Information Processing Systems.* Information Processing Systems Technical Report ANSI X3/TR-1-82. 1982.
- ISO 646-1983 Invariant Code Set. This was used to help describe the C character set, and to select the characters that need to be represented by trigraphs.
- *IEEE Standard for Binary Floating-Point Arithmetic.* ANSI/IEEE Standard 754-1985. This is the basis for the Standard’s description of floating-point numbers.
- ISO 4217 Codes for Representation of Currency and Funds. This is the target for the Standard’s description of locale-specific ways to represent money.

The first two, due to their fundamental effect upon the Standard, are referred to as the “base documents”.

Cross-reference

Standard, §1.3, §1.5

See Also

Definitions, Environment, Language, Library, DOS-specific features

standard error — Definition

When a C program begins, it opens three text streams by default: the standard error, the standard input, and the standard output. The *standard error* is the stream into which error messages are written. In most implementations, the standard error stream is associated with the user’s terminal.

The macro **stderr** points to the **FILE** object through which the standard error device is accessed. It is defined in the header **stdio.h**.

Cross-references

Standard, §4.9.3
The C Programming Language, ed. 2, pp. 151*ff*

See Also

standard input, standard output, stderr, STDIO

standard input — Definition

When a C program begins execution, it opens three text streams by default: the standard error, the standard input, and the standard output. The *standard input* is the stream from which the program receives input by default. In most implementations, the standard input stream is associated with the user's terminal.

The macro **stdin** points to the **FILE** object that accesses the standard input stream. It is defined in the header **stdio.h**.

Cross-references

Standard, §4.9.3
The C Programming Language, ed. 2, pp. 151*ff*

See Also

standard error, standard output, stdin, STDIO

standard output — Definition

When a C program begins execution, it opens three text streams by default: the standard output, the standard input, and the standard error. The *standard output* is the stream into which a program's non-diagnostic output is written. In most implementations, the standard output stream is associated with the user's terminal.

The macro **stdout** points to the **FILE** object that accesses the standard output device. It is defined in the header **stdio.h**.

Cross-references

Standard, §4.9.3
The C Programming Language, ed. 2, pp. 151*ff*

See Also

standard error, standard input, STDIO, stdout

stat() — Access checking (libc)

Find file attributes
#include <stat.h>
short stat(char **file*, **struct stat** **statptr*);

stat returns a structure that contains the attributes of a file. This function is included to maintain compatibility with the UNIX and COHERENT operating systems.

file points to the path name of file, and *statptr* points to a structure of the type **stat**, as defined in the header file **stat.h**.

The following summarizes the structure **stat**:

```
struct stat {
      unsigned short st_mode;           /* mode */
      long st_size;                     /* size, in bytes */
      struct dostime st_dostime;        /* MS-DOS time and date */
      time_t st_mtime;                  /* modification time */
};
```

The structure **dostime** is defined in the header file **dosfind.h**. The following lists the legal values for **st_mode**, which sets the file's attributes:

S_IFMT	0x0300	type
S_IFDIR	0x0100	directory
S_IFREG	0x0200	regular file
S_IREAD	0x0400	read permission; always 1
S_IWRITE		0x0800write permission

The entry **st_size** gives the size of the file, in bytes.

stat returns -1 if an error occurs, e.g., the file cannot be found. Otherwise, it returns zero.

Example

The following example, called **test.c**, demonstrates **stat**. When compiled, it will take a file name as an argument; it will then search for the file and, if it is found, print a summary of its status.

```
#include <stat.h>
#include <stdio.h>
#include <stdlib.h>
char *_cmdname = "TEST";

void
fatal(char *error)
{
      fprintf(stderr, "Fatal Error: %s\n", error);
      exit(EXIT_FAILURE);
}

main(int argc, char *argv[])
{
      char *name;
      struct stat status;

      if ( argc != 2 )
            fatal("Usage: command filename");
      name = argv[1];
      if (stat(name, &status) != 0)
            fatal("Can’t find file");

      printf("File: {%s}\n", name);
      printf("st_mode: 0x%x\n", status.st_mode);
      printf("st_size: %D\n", status.st_size);
      printf("st_dostime: %02d-%02d-%02d %02d:%02d:%02d\n",
            status.st_dostime.dos_month,
            status.st_dostime.dos_day,
            status.st_dostime.dos_year+80,
            status.st_dostime.dos_hour,
            status.st_dostime.dos_minute,
            status.st_dostime.dos_twosec*2);
      printf("st_mtime: %s", ctime(&status.st_mtime));
      return EXIT_SUCCESS;
}
```

See Also

access checking, open, stat.h

stat.h — Header

Definitions and declarations to obtain file status
#include <stat.h>

stat.h is a header file that contains the declarations of several structures used by the routine **stat**, which returns information about a file's status.

See Also

access checking, header, stat

statements — Overview

A *statement* specifies an action to be performed. Unless otherwise specified, statements are executed in the order in which they appear in the program.

The actions of some statements may be controlled by a *full expression*; this is an expression that is not part of another expression. For example, **do**, **if**, **for**, **switch**, and **while** introduce statements that are controlled by one or more full expressions. The **return** statement may also use a full expression.

The Standard describes the following varieties of statements:

Compound statement

Expression statement

Iteration statements
- **do**
- **for**
- **while**

Jump statements
- **break**
- **continue**
- **goto**
- **return**

Labelled statements
- **case**
- **default**

Null statement

Selection statements
- **if**
- **else**
- **switch**

The set of compound, iteration, and selection statements is the foundation upon which many programming languages are based. From these alone, a programmer can construct many useful and interesting programs.

Let's C also includes the keyword **alien**, which marks a function that uses non-C calling conventions.

Cross-references

Standard, §3.6
The C Programming Language, ed. 2, pp. 222*ff*

See Also

alien, Language

static — C keyword

Internal linkage
static *type identifier*

The storage-class specifier **static** declares that *identifier* has internal linkage. This specifier may not be used to declare a function that has block scope.

Cross-references

Standard, §3.5.1
The C Programming Language, ed. 2, p. 83

See Also

linkage, storage-class identifiers

stdarg.h — Header

Header for variable numbers of arguments
#include <stdarg.h>

The header **stdarg.h** declares and defines routines that are used to traverse a variable-length argument list. It declares the type **va_list** and the function **va_end**, and it defines the macros **va_start** and **va_arg**.

Cross-references

Standard, §4.8
The C Programming Language, ed. 2, p. 254

See Also

header, variable arguments

stderr — Macro

Pointer to standard error stream
#include <stdio.h>

When a C program begins, it opens three text streams by default: the standard error, the standard input, and the standard output. **stderr** points to the **FILE** object through which the standard error stream is accessed; this is the stream into which error messages are written. In most implementations, the standard error stream is associated with the user's terminal.

stderr is defined in the header **stdio.h**.

stderr is not fully buffered when it is opened.

Example

For an example of **stderr** in a program, see **fprintf**.

Cross-references

Standard, §4.9.1, §4.9.3
The C Programming Language, ed. 2, p. 243

See Also

stdin, stdout, standard error, STDIO, stdio.h

stdin — Macro

Pointer to standard input stream
#include <stdio.h>

When a C program begins, it opens three text streams by default: the standard error, the standard input, and the standard output. **stdin** points to the **FILE** object that accesses the standard input stream; this is the stream from which the program receives input by default. In most implementations, the standard input stream is associated with the user's terminal.

stdin is defined in the header **stdio.h**.

Example

For an example of **stdin** in a program, see **setvbuf**.

Cross-references

Standard, §4.9.1, §4.9.3
The C Programming Language, ed. 2, p. 243

See Also

stderr, stdout, standard input, STDIO, stdio.h

STDIO — Overview

Standard input and output
#include <stdio.h>

STDIO is an acronym for *standard input and output*. Input-output can be performed on text files, binary files, or interactive devices. It can be either buffered or unbuffered.

The Standard describes 41 functions that perform input and output, as follows:

Error handling

clearerr	Clear a stream's error indicator
feof	Examine a stream's end-of-file indicator
ferror	Examine a stream's error indicator
perror	Write error message into standard error stream

File access

fclose	Close a stream
fflush	Flush an output stream's buffer
fopen	Open a stream
freopen	Close and reopen a stream
setbuf	Set an alternate buffer for a stream
setvbuf	Set an alternate buffer for a stream

File operations

remove	Remove a file
rename	Rename a file
tmpfile	Create a temporary file
tmpnam	Generate a unique name for a temporary file

File positioning

fgetpos	Get value of stream's file-position indicator (**fpos_t**)
fseek	Set stream's file-position indicator
fsetpos	Set stream's file-position indicator (**fpos_t**)
ftell	Get the value of the file-position indicator
rewind	Reset stream's file-position indicator

Input-output

By character

fgetc	Read a character from a stream
fgets	Read a line from a stream
fputc	Write a character into a stream
fputs	Write a string into a stream
getc	Read a character from a stream
getchar	Read a character from the standard input stream
gets	Read a string from the standard input stream
putc	Write character into a stream
putchar	Write a character into the standard output
puts	Write a string into the standard output
ungetc	Push a character back into the input stream

Direct

fread	Read data from a stream
fwrite	Write data into a stream

Formatted

fprintf	Print formatted text into a stream
fscanf	Read formatted text from a stream
printf	Format and print text into standard output stream
scanf	Read formatted text from standard input stream
sprintf	Print formatted text into a string
sscanf	Read formatted text from string
vfprintf	Format and print text into a stream
vprintf	Format and print text into standard output stream
vsprintf	Format and print text into a string

The prototypes for these functions appear in the header **stdio.h**, along with definitions for the types and macros they use.

All STDIO functions access a file or device through a *stream*. A stream is accessed via an object of type **FILE**; this object contains all of the information needed to access the file or device under the given environment. Because of the heterogeneous environments under which C has been implemented, the Standard does not describe the interior workings of the **FILE** object. It states only that this object contain all information needed to access a stream under the given environment.

Cross-references

Standard, §4.9
The C Programming Language, ed. 2, pp. 151*ff*, 241*ff*

See Also

close, create, extended STDIO, file, file-position indicator, Library, line, open, stdio.h, stream

Notes

Let's C also includes the following extended functions and macros that perform STDIO tasks:

_exit	Exit from a program without clean-up
close	Close a file
creat	Create a file
dup	Duplicate a file descriptor
dup2	Duplicate a file descriptor
execall	Pass arguments to a program
fdopen	Use a file descriptor to open a stream
fgetw	Read a word from a stream

fileno	Get a file descriptor
fputw	Write a word into a stream
getanb	Read unbuffered from auxiliary port
getcnb	Read unbuffered from the console
getw	Read a word from a stream
in	Read a word from a port
inb	Read a byte from a port
lseek	Set stream's file-position indicator
open	Open a file
out	Write a word to a port
outb	Write a byte to a port
putanb	Write unbuffered to auxiliary port
putcnb	Write unbuffered to the console
putw	Write a word into a stream
read	Read data from a stream
regtop	Convert register pair to pointer
tempnam	Generate a unique name for a temporary file
unlink	Remove a file
write	Write data into a stream

The ANSI Standard forbids any ANSI header to declare or define any function or macro that is not described within the Standard. Therefore, the routines **fdopen**, **fgetw**, **fileno**, **fputw**, **getanb**, **getcnb**, **getw**, **putanb**, **putcnb**, **putw**, and **regtop** have been moved from header **stdio.h** into a new header, **xstdio.h**.

Any programs that uses any of these extended functions will not comply strictly with the Standard, and may not be portable to other compilers or environments.

stdio.h — Header

Declarations and definitions for STDIO

stdio.h is the header that holds the definitions, declarations, and function prototypes used by the STDIO routines.

The following lists the types, macros, and manifest constants defined in **stdio.h**:

Types

FILE	Hold descriptor for a stream
fpos_t	Hold current position within a file

Cross-references

Standard, §4.9.1
The C Programming Language, ed. 2, pp. 151*ff*, 241*ff*

See Also

header, STDIO

stdlib.h — Header

General utilities
#include <stdlib.h>

stdlib.h is a header that declares the Standard's set of general utilities and defines attending macros and data types, as follows:

Types

div_t	Type of object returned by **div**
ldiv_t	Type of object returned by **ldiv**
EXIT_FAILURE	Value to indicate that program failed to execute properly
EXIT_SUCCESS	Value to indicate that program executed properly
MB_CUR_MAX	Largest size of multibyte character in current locale
MB_LEN_MAX	Largest overall size of multibyte character in any locale
RAND_MAX	Largest size of pseudo-random number

Functions

abort	End program immediately
abs	Compute the absolute value of an integer
atexit	Register a function to be executed at exit
atof	Convert string to floating-point number
atoi	Convert string to integer
atol	Convert string to long integer
bsearch	Search an array
calloc	Allocate dynamic memory
div	Perform integer division
exit	Terminate a program gracefully
free	De-allocate dynamic memory to free memory pool
getenv	Read environmental variable
labs	Compute the absolute value of a long integer
ldiv	Perform long integer division
malloc	Allocate dynamic memory
mblen	Compute length of a multibyte character
mbstowcs	Convert multibyte-character sequence to wide characters
mbtowc	Convert multibyte character to wide character
qsort	Sort an array
rand	Generate pseudo-random numbers
realloc	Reallocate dynamic memory
strtod	Convert string to floating-point number
strtol	Convert string to long integer
strtoul	Convert string to unsigned long integer
system	Suspend a program and execute another
wcstombs	Convert wide-character sequence to multibyte characters
wctomb	Convert wide character to multibyte character

Cross-references

Standard, §4.10.1
The C Programming Language, ed. 2, p. 251

See Also

general utilities

stdout — Macro

Pointer to standard output stream
#include <stdio.h>

When a C program begins, it opens three text streams by default: the standard error, the standard input, and the standard output. **stdout** points to the **FILE** object that accesses the standard output stream. This is the stream into which non-diagnostic output is written. Under **Let's C**, the standard output stream is associated with the user's terminal.

stdout is defined in the header **stdio.h**.

Example

For an example of **stdout** in a program, see **setvbuf**.

Cross-references

Standard, §4.9.1, §4.9.3
The C Programming Language, ed. 2, p. 243

See Also

stdin, stderr, standard output, STDIO, stdio.h

stime() — Extended function (libc)

Set the operating system time
#include <time.h>
#include <xtime.h>
int stime(time_t ****timep*);**

stime sets the operating system time, which **Let's C** defines as being the number of seconds since midnight of January 1, 1970, 0h00m00s UTC. The argument *timep* points to the new system time, which is of the type **time_t**. This is defined in the header file **time.h** as being equivalent to a **long**.

stime returns -1 on error, zero otherwise.

Example

The following example prints the time, then uses **stime** to reset the time by one hour.

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

main(void)
{
      time_t tp;

      /* print current time */
      time(&tp);
      printf("%s\n", ctime(&tp));

      /* subtract one hour (3600 seconds) from current time */
      tp -= 3600;
      if (stime(&tp) == -1) {
            printf("Cannot reset time.\n");
            exit(EXIT_FAILURE);
      }

      /* print altered time */
      time(&tp);
      printf("%s\n", ctime(&tp));
```

```
	/* add one hour to current time, to correct above */
	tp += 3600;
	if (stime(&tp) == -1) {
		printf("Cannot re-reset time.\n");
		exit(EXIT_FAILURE);
	}

	/* print fixed time, to confirm correction */
	time(&tp);
	printf("%s\n", ctime(&tp));
	return EXIT_SUCCESS;
}
```

See Also

extended time

Notes

To conform with the ANSI Standard, this function has been moved from the header **time.h** to the header **xtime.h**. This may require that some code be altered.

storage-class specifiers — Overview

A *storage-class specifier* specifies the manner in which an object is to be stored in memory. There are five such specifiers:

auto	Automatic storage duration
extern	External linkage
register	Quick access required
static	Internal linkage
typedef	Synonym for another type

Only one storage-class specifier is allowed per declaration. The Standard declares as “obsolescent” any declaration that does not have its storage class as the first specifier in a declaration.

Strictly speaking, **typedef** is not a storage-class specifier. The Standard bundles it into this group for the sake of convenience.

Cross-references

Standard, §3.5.1
The C Programming Language, ed. 2, p. 210

See Also

declarations, storage class, storage duration

storage duration — Definition

The term *storage duration* refers to how long a given object is retained within memory. There are two varieties of storage duration: *static* and *automatic*.

An object with static storage duration is retained throughout program execution. Its storage is reserved, and the object is initialized only when the program begins execution. All string literals have static duration, as do all objects that are declared globally — that is, declared outside of any function.

An object with automatic duration is declared within a block of code. It endures within memory only for the life of that block of code. Memory is allocated for the variable whenever that block is entered and deallocated when the block is terminated, either by encountering the ‘}’ that closes the block, or by exiting the block with **goto**, **longjmp**, or **return**.

A common practice is to declare all automatic variables at the beginning of a function. These variables endure as long as the function is operating. If the function calls another function, then these functions are stored away (usually in an special area of memory called the “stack”), but they cannot be accessed until the called function returns.

Cross-references

Standard, §3.1.2.4
The C Programming Language, ed. 2, p. 195

See Also

auto, identifiers, scope, static

strcat() — String handling (libc)

Append one string onto another
char *strcat(char ****string1*, const char ****string2***);**

strcat copies all characters in *string2*, including the terminating null character, onto the end of the string pointed to by *string1*. The null character at the end of *string1* is overwritten by the first character of *string2*.

strcat returns the pointer *string1*.

Example

The following example concatenates two strings.

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

char string1[80] = "The first string. ";
char string2[] = "The second string.";

main(void)
{
      printf("result = %s\n", strcat(string1, string2));
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.11.3.1
The C Programming Language, ed. 2, p. 250

See Also

string handling, strncat

Notes

string1 should point to enough reserved memory to hold itself and *string2*. Otherwise, data or code will be overwritten.

strchr() — String handling (libc)

Find a character in a string
#include <string.h>
char *strchr(const char ****string*, int** *character***);**

strchr searches for *character* within *string*. The null character at the end of *string* is included within the search. It is equivalent to the non-ANSI function **index**.

Internally, **strchr** converts *character* from an **int** to a **char** before searching for it within *string*.

strchr returns a pointer to the first occurrence of *character* within *string*. If *character* is not found, it returns NULL.

Having **strchr** search for a null character will always produce a pointer to the end of a string. For example,

```
	char *string;
	assert(strchr(string, '\0') == string + strlen(string));
```

will never fail.

Example

The following example creates functions called **replace** and **trim**. **replace** finds and replaces every occurrence of an item within a string and returns the altered string. **trim** removes all trailing spaces from a string, and returns a pointer to the altered string.

```
#include <stdlib.h>
#include <stddef.h>
#include <string.h>
#include <stdio.h>

char *
replace(char *string, char item, char newitem)
{
	char *start;

	/* replacing 0 is too dangerous */
	if ((start = string) == NULL || item == '\0')
		return(start);
	while ((string = strchr(string, item)) != NULL)
		*string = newitem;
	return(start);
}

char *
trim(char * str)
{
	register char *endp;

	if(str == NULL)
		return(str);

	/* start at end of string while in string and spaces */
	for(endp  = strchr(str, '\0');
	    endp != str && *--endp == ' '; )
		*endp = '\0';
	return(str);
}

char string1[] = "Remove trailing spaces         ";
char string2[] = "Spaces become dashes.";
main(void)
{
	printf("\"%s\"\n", trim(string1));
	printf("%s\n", replace(string2, ' ', '-'));
	return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.11.5.2
The C Programming Language, ed. 2, p. 249

See Also

index, memchr, strcspn, string handling, strpbrk, strrchr, strspn, strstr, strtok

strcmp() — String handling (libc)

Compare two strings
#include <string.h>
int strcmp(const char **string1*, **const char** **string2*);

strcmp lexicographically compares the string pointed to by *string1* with the one pointed to by *string2*. Comparison ends when a null character is encountered.

strcmp compares the two strings character by character until it finds a pair of characters that are not identical. It returns a number less than zero if the character in *string1* is less (i.e,. occurs earlier in the character table) than its counterpart in *string2*. It returns a number greater than zero if the character in *string1* is greater (i.e,. occurs later in the character table) than its counterpart in *string2*. If no characters are found to differ, then the strings are identical and **strcmp** returns zero.

Example

For an example of this function, see **fflush**.

Cross-references

Standard, §4.11.4.2
The C Programming Language, ed. 2, p. 250

See Also

memcmp, strcmp, strcoll, string handling, strncmp, strxfrm

Notes

strcmp differs from the memory-comparison routine **memcmp** in the following ways:

First, **strcmp** compares strings rather than areas of memory; therefore, it stops when it encounters a null character.

Second, **memcmp** takes two pointers to **void**, whereas **strcmp** takes two pointers to **char**. The following code illustrates how this difference affects these functions:

```
char carray[10];
int iarray[10];
char *s = "hi";
   . . .
strcmp(carray, s)          /* RIGHT */
memcmp(carray, s, 3)       /* RIGHT */
strcmp(iarray, s)          /* WRONG, 1st arg not char * */
memcmp(iarray, s, 3)       /* RIGHT, args cast to void * */
```

It is wrong to use **strcmp** to compare an **int** array with a **char** array, because this function compares strings. Using **memcmp** to compare an **int** array with a **char** array is permissible because **memcmp** simply compares areas of data.

strcoll() — String handling (libc)

Compare two strings, using locale-specific information
#include <string.h>
int strcoll(const char **string1*, **const char** **string2*);

strcoll lexicographically compares the string pointed to by *string1* with one pointed to by *string2*. Comparison ends when a null character is read. **strcoll** differs from *strcmp* in that it uses information concerning the program's locale, as set by the function **setlocale**, to help compare strings. It can be used to provide locale-specific collating. See **localization** for more information about setting a program's locale.

strcoll compares the two strings character by character until it finds a pair of characters that are not identical. It returns a number less than zero if the character in *string1* is less (i.e,. occurs earlier in the character table) than its counterpart in *string2*. It returns a number greater than zero if the character in *string1* is greater (i.e,. occurs later in the character table) than its counterpart in *string2*. If no characters are found to differ, then the strings are identical and **strcoll** returns zero.

Cross-references

Standard, §4.11.4.3
The C Programming Language, ed. 2, p. 250

See Also

localization, memcmp, strcmp, string handling, strncmp, strxfrm

Notes

The string-comparison routines **strcoll**, **strcmp**, and **strncmp** differ from the memory-comparison routine **memcmp** in that they compare strings rather than regions of memory. They stop when they encounter a null character, but **memcmp** does not.

strcpy() — String handling (libc)

Copy one string into another
#include <string.h>
char *strcpy(char ****string1*, const char ****string2***);**

strcpy copies the string pointed to by *string2*, including the null character, into the area pointed to by *string1*.

strcpy returns *string1*.

Example

For an example of this function, see **realloc**.

Cross-references

Standard, §4.11.2.3
The C Programming Language, ed. 2, p. 249

See Also

memcpy, memset, string handling, strncpy

Notes

If the region of memory pointed to by *string1* overlaps with the string pointed to by *string2*, the behavior of **strcpy** is undefined.

string1 should point to enough reserved memory to hold *string2*, or code or data will be overwritten.

strcspn() — String handling (libc)

Return length a string excludes characters in another
#include <string.h>
size_t strcspn(const char ****string1*, const char ****string2***);**

strcspn compares *string1* with *string2*. It then returns the length, in characters, for which *string1* consists of characters *not* found in *string2*.

Example

The following example returns a pointer to the first white-space character in a string. White space is defined as space, tab, or newline.

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

char *
nextwhite(char *string)
{
      size_t skipcount;

      if(string == NULL)
            return NULL;
      skipcount = strcspn(string, "\t \n");
      return(string + skipcount);
}

char string1[] = "My love is like a red, red, rose";

main(void)
{
      printf(nextwhite(string1));
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.11.5.3
The C Programming Language, ed. 2, p. 250

See Also

memchr, strchr, string handling, strpbrk, strrchr, strspn, strstr, strtok

stream — Definition

The term *stream* is a metaphor for the flow of data between a C program and either an external I/O device (e.g., a terminal) or a file stored on a semi-permanent medium (e.g., disk or tape). A program can read data from a stream, write data into it, or (in the case of a file) directly access any named portion of it.

The Standard describes two types of stream: the *binary* stream and the *text* stream.

A binary stream is simply a sequence of bytes. The Standard requires that once a program has written a sequence of bytes into a stream, it should be able to read back the same sequence of bytes unchanged from that stream — with the sole exception that, in some environments, one or more null characters may be appended to the end of the sequence.

A text stream, on the other hand, consists of characters that have been organized into lines. A *line* in turn, consists of zero or more characters terminated by a newline character. Under MS-DOS, a text stream is practically identical to a binary stream, with the exception that it cannot read or write characters other than alphanumeric characters, the null character, and the newline character.

The Standard mandates that when data are written into a binary file, the file is not truncated. Under **Let's C**, the same is true for text files.

The Standard also mandates that an implementation should be able to handle a line that is **BUFSIZ** characters long, which includes the terminating newline character. **BUFSIZ** is a macro that is defined in the header **stdio.h**, and must be defined to be equal to at least 256.

The maximum number of streams that can be opened at any one time is given by the macro

FOPEN_MAX. Under **Let's C**, this is 20, including **stdin**, **stdout**, and **stderr**.

Cross-references

Standard, §4.9.2
The C Programming Language, ed. 2, p. 241

See Also

buffer, file, line, STDIO, stdio.h

strerror() — String handling (libc)

Translate an error number into a string
#include <string.h>
char *strerror(int *error***);**

strerror helps to generate an error message. It takes the argument *error*, which presumably is an error code generated by an error condition in a program, and may return a pointer to the corresponding error message.

The error numbers recognized and the texts of the corresponding error messages all depend upon the implementation.

Example

This example prints the user's error message and the standard error message before exiting.

```
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stddef.h>

fatal(char * msg)
{
      int save;

      save = errno;
      /* this may clobber errno */
      fprintf(stderr, "%s", msg);
      if (save)
            fprintf(stderr, ": %s", strerror(save));
      fprintf(stderr, "\n");
      exit(save);
}

main(void)
{
      /* guaranteed wrong */
      sqrt(-1.0);
      fatal("What does sqrt say to -1?");
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.11.6.2
The C Programming Language, ed. 2, p. 250

See Also

error codes, errors, perror, string handling

Notes

strerror returns a pointer to a static array that may be overwritten by a subsequent call to **strerror**.

strerror differs from the related function **perror** in the following ways: **strerror** receives the error number through its argument *error*, whereas **perror** reads the global constant **errno**. Also, **strerror** returns a pointer to the error message, whereas **perror** writes the message directly into the standard error stream.

The error numbers recognized and the texts of the messages associated with each error number depend upon the implementation. However, **strerror** and **perror** return the same error message when handed the same error number.

strftime() — Time function (libc)

Format locale-specific time
#include <time.h>
size_t strftime(char **string*, **size_t** *maximum*, **const char** **format*,
const struct tm **brokentime*);

The function **strftime** provides a locale-specific way to print the current time and date. It also gives you an easy way to shuffle the elements of date and time into a string that suits your preferences.

strftime references the portion of the locale that is affected by the calls

```
setlocale(LC_TIME, locale);
```

or

```
setlocale(LC_ALL, locale);
```

For more information on setting locales, see the entry for **localization**.

string points to the region of memory into which **strftime** writes the date and time string it generates. *maximum* is the maximum number of characters that can be written into *string*. *string* should point to an area of allocated memory at least *maximum*+1 bytes long; if it does not, reserved portions of memory may be overwritten.

brokentime points to a structure of type **tm**, which contains the broken-down time. This structure must first be initialized by either of the functions **localtime** or **gmtime**.

Finally, *format* points to a string that contains one or more conversion specifications, which guide **strftime** in building its output string. Each conversion specification is introduced by the percent sign '%'. When the output string is built, each conversion specification is replaced by the appropriate time element. Characters within *format* that are not part of a conversion specification are copied into *string*; to write a literal percent sign, use "%%".

strftime recognizes the following conversion specifiers:

a The locale's abbreviated name for the day of the week.

A The locale's full name for the day of the week.

b The locale's abbreviated name for the month.

B The locale's full name for the month.

c The locale's default representation for the date and time.

d The day of the month as an integer (01 through 31).

H The hour as an integer (00 through 23).

I The hour as an integer (01 through 12).

j The day of the year as an integer (001 through 366).

m The month as an integer (01 through 12).

M The minute as an integer (00 through 59).

p The locale's way of indicating morning or afternoon (e.g, in the United States, "AM" or "PM").

S The second as an integer (00 through 59).

U The week of the year as an integer (00 through 53); regard Sunday as the first day of the week.

w The day of the week as an integer (0 through 6); regard Sunday as the first day of the week.

W The day of the week as an integer (0 through 6); regard Monday as the first day of the week.

x The locale's default representation of the date.

X The locale's default representation of the time.

y The year within the century (00 through 99).

Y The full year, including century.

Z The name of the locale's time zone. If no time zone can be determined, print a null string.

Use of any conversion specifier other than the ones listed above will result in undefined behavior.

If the number of characters written into *string* is less than or equal to *maximum*, then **strftime** returns the number of characters written. If, however, the number of characters to be written exceeds *maximum*, then **strftime** returns zero and the contents of the area pointed to by *string* are indeterminate.

Cross-references

Standard, §4.12.3.5
The C Programming Language, ed. 2, p. 256

See Also

asctime, ctime, date and time, gmtime, localtime, time_t, tm

Notes

strftime is modelled after the UNIX command **date**.

string.h — Header

#include <string.h>
string.h is the header that holds the declarations and definitions of all routines that handle strings and buffers. For a list of these routines, see **string handling**.

Cross-references

Standard, §4.11
The C Programming Language, ed. 2, p. 249

See Also

header, string handling

string handling — Overview

#include <string.h>

The Standard describes 22 routines for handling strings and regions of memory. All are declared in the header **string.h**.

String comparison	
memcmp	Compare two regions
strcmp	Compare two strings
strcoll	Compare two strings, using locale information
strncmp	Compare one string with first *n* bytes of another
strxfrm	Transform a string using locale information
String concatenation	
strcat	Concatenate two strings
strncat	Concatenate one string with *n* bytes of another
String copying	
memcpy	Copy one region into another
memmove	Copy one region into another with which it may overlap
strcpy	Copy one string into another
strncpy	Copy *n* bytes from one string into another
String miscellaneous	
memset	Fill a region with a character
strerror	Return the text of a pre-defined error message
strlen	Return the length of a string
String searching	
memchr	Find first occurrence of a character in a region
strchr	Find first occurrence of a character in a string
strcspn	Find how much of the initial portion of a string consists of characters *not* found in another string
strpbrk	Find first occurrence in one string of any character from another string
strrchr	Find *last* occurrence of a character within a string
strspn	Find how much of the initial portion of string consists only of characters from another string
strstr	Find one string within another string
strtok	Break a string into tokens

Cross-references

Standard, §4.11
The C Programming Language, ed. 2, p. 249

See Also

Library, string, string.h

Notes

Let's C includes three additional functions for string searching: **index**, **pnmatch**, and **rindex**.

index and **rindex** are synonymous with, respectively, **strchr** and **strrchr**. They are included only to support existing code, and it is recommended that they not be used in new code. **pnmatch** resembles **strstr**, except that it allows you to include wildcards in the search pattern. See their respective Lexicon entries for more information.

string literal — Definition

A *string literal* consists of zero or more characters that are enclosed by quotation marks “”. For example, the following is a string literal:

```
"This is a string literal."
```

Each character within a string literal is handled exactly as if it were within a character constant, with the following exceptions: The apostrophe ´ may be represented either by itself or by the escape sequence \´, and the quotation mark “” must be represented by the escape sequence \".

A string literal has **static** duration. Its type is array of **char** which is initialized to the string of characters enclosed within the quotation marks.

If string literals are adjacent, the translator will concatenate them. For example, the string literals

```
"Here’s a string literal" "Here’s another string literal"
```

are automatically concatenated into one string literal.

If a string literal is not followed by another string literal, then the translator appends a null character to the end of the string as a terminator.

If two or more string literals within the same scope are identical, then the translator may store only one of them in memory and redirect to that one copy all references to any of the duplicate literals. For this reason, a program’s behavior is undefined whenever it modifies a string literal.

A *wide-character literal* is a string literal that is formed of wide characters rather than ordinary, one-byte characters. It is marked by the prefix ‘L’. For example, the following

```
L"This is a wide-character literal"
```

is stored in the form of a string of wide characters. See **multibyte characters** for more information about wide characters.

Cross-references

Standard, §3.1.4
The C Programming Language, ed. 2, p. 194

See Also

", escape sequences, lexical elements, string, trigraphs

Notes

Because trigraph sequences are interpreted in translation phase 1, before string literals are parsed, a string literal that contains trigraph sequences will be translated to a different string. This is a quiet change that may break existing code.

strip — Command

Strip debug table from executable file
strip -drs *file ...*

strip removes the debug tables from a executable file that had been compiled with the **-VCSD** option. It makes the executable file noticeably smaller.

See Also

cc, commands, nm, size

Notes

strip can be used only on fully linked files.

strlen() — String handling (libc)

Measure the length of a string
size_t strlen(const char ***string*)**

strlen counts the number of characters in *string* up to the null character that ends it. It returns the number of characters in *string*, excluding the null character that ends it.

Example

The following example prints the length of an entered string.

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

main(void)
{
      char buf[132];

      printf("Enter something\n");
      if(gets(buf) != NULL)
            printf("You entered %lu characters\n",
                  (unsigned long)strlen(buf));
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.11.6.3
The C Programming Language, ed. 2, p. 250

See Also

string handling

strncat() — String handling (libc)

Append *n* characters of one string onto another
#include <string.h>
char *strncat(char ****string1*, const char ****string2***, size_t** ***n*****);**

strncat copies up to *n* characters from the string pointed to by *string2* onto the end of the one pointed to by *string1*. It stops when *n* characters have been copied or it encounters a null character in *string2*, whichever occurs first. The null character at the end of *string1* is overwritten by the first character of *string2*.

strncat returns the pointer *string1*.

Example

The following example concatenates two strings to make a file name. It works for an operating system in which a file name can have no more than eight characters, and a suffix of no more than three characters.

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

```
char *
dosfilen(char *dosname, char *filename, char *filetype)
{
      *dosname = '\0';
      /* strncpy() doesn't guarantee a NULL */
      strncat(dosname, filename, 8);
      strcat(dosname, ".");
      return(strncat(dosname, filetype, 3));
}

main(void)
{
      char dosname[13];

      puts(dosfilen(dosname, "A_LONG_FILENAME",
            "A_LONG_FILETYPE"));
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.11.3.2
The C Programming Language, ed. 2, p. 250

See Also

strcat, string handling

Notes

strncat always appends a null character onto the end of the concatenated string. Therefore, the number of characters appended to the end of *string1* could be as many as *n*+1. *string1* should point to enough allocated memory to hold itself plus *n*+1 characters; if it does not, data or code will be overwritten.

strncmp() — String handling (libc)

Compare one string with a portion of another
#include <string.h>
int strncmp(const char **string1*, **const char** **string2*, **size_t** *n*);

strncmp compares *string1* with *n* bytes of *string2*. Comparison ends when a null character is read.

strncmp compares the two strings character by character until it finds a pair of characters that are not identical. It returns a number less than zero if the character in *string1* is less (i.e,. occurs earlier in the character table) than its counterpart in *string2*. It returns a number greater than zero if the character in *string1* is greater (i.e,. occurs later in the character table) than its counterpart in *string2*. If no characters are found to differ, then the strings are identical and **strncmp** returns zero. Comparison ends either when *n* bytes have been compared or a null character has been encountered in either string. The null character is compared before **strncmp** terminates.

Example

The following example searches for a word within a string. It is a simple implementation of the function **strstr**.

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

```
void fatal(const char *string)
{
      fprintf(stderr, "%s\n", string);
      exit(EXIT_FAILURE);
}

main(int argc, char *argv[])
{
      int word, string, i;

      if (--argc != 2)
            fatal("Usage: example word string");

      word = strlen(argv[1]);
      string = strlen(argv[2]);
      if (word >= string)
            fatal("Word is longer than string being searched.");

      /* walk down "string" and search for "word" */
      for (i = 0; i < string - word; i++)
            if (strncmp(argv[2]+i, argv[1], word) == 0) {
                  printf("%s is in %s.\n", argv[1], argv[2]);
                  exit(EXIT_SUCCESS);
            }

      /* if we get this far, "word" isn't in "string" */
      printf("%s is not in %s.\n", argv[1], argv[2]);
      exit(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.11.4.4
The C Programming Language, ed. 2, p. 250

See Also

memcmp, strcmp, strcoll, string handling, strxfrm

Notes

The string-comparison routines **strcoll**, **strcmp**, and **strncmp** differ from the memory-comparison routine **memcmp** in that they compare strings rather than regions of memory. They stop when they encounter a null character, but **memcmp** does not.

strncpy() — String handling (libc)

Copy one string into another
#include <string.h>
char *strncpy(char ****string1*, const char ****string2***, size_t** *n***);**

strncpy copies *n* characters from the string pointed to by *string2* into the area pointed to by *string1*. Copying ends when *n* bytes have been copied or a null character is encountered in *string2*.

If *string2* is less than *n* characters long, **strncpy** pads *string1* with null characters until *n* characters have been deposited.

strncpy returns *string1*.

Example

This example reads a file of names and changes them from the format

```
first_name [middle_initial] last_name
```

to the format:

```
last_name, first_name [middle_initial]
```

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

#define NNAMES 512
#define MAXLEN 60
#define PERIOD '.'
#define SPACE ' '
#define COMMA ','
#define NEWLINE '\n'

char *array[NNAMES];
char gname[MAXLEN], lname[MAXLEN];

main(int argc, char *argv[])
{
      FILE *fp;
      int count, num;
      char  *name, string[MAXLEN], *cptr, *eptr;
      unsigned glength, length;

      /* check number of arguments */
      if (--argc != 1) {
            fprintf (stderr, "Usage: example filename\n");
            exit(EXIT_FAILURE);
      }

      /* open file */
      if ((fp = fopen(argv[1], "r")) == NULL) {
            fprintf(stderr, "Cannot open %s\n", argv[1]);
            exit(EXIT_FAILURE);
      }
      count = 0;

      /* get line and examine it */
      while (fgets(string, MAXLEN, fp) != NULL) {
            if ((cptr = strchr(string, PERIOD)) != NULL) {
                  cptr++;
                  cptr++;
            } else if ((cptr=strchr(string, SPACE))!=NULL)
                  cptr++;
            else continue;

            strcpy(lname, cptr);
            eptr = strchr(lname, NEWLINE);
            *eptr = COMMA;

            strcat(lname, " ");
            glength = (unsigned)(strlen(string)-strlen(cptr));
            strncpy(gname, string, glength);

            name = strncat(lname, gname, glength);
            length = (unsigned)strlen(name);
            array[count] = (char *)malloc(length + 1);

            strcpy(array[count],name);
            count++;
      }
```

```
	for (num = 0; num < count; num++)
		printf("%s\n", array[num]);
	return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.11.2.4
The C Programming Language, ed. 2, p. 249

See Also

memcpy, memset, strcpy, string handling

Notes

string1 should point to enough reserved memory to hold *n* characters. Otherwise, code or data will be overwritten.

If the region of memory pointed to by *string1* overlaps with the string pointed to by *string2*, then the behavior of **strncpy** is undefined.

strpbrk() — String handling (libc)

Find first occurrence of a character from another string
#include <string.h>
char *strpbrk(const char ****string1*, const char ****string2***);**

strpbrk returns a pointer to the first character in *string1* that matches any character in *string2*. It returns NULL if no character in *string1* matches a character in *string2*. The set of characters that *string2* points to is sometimes called the “break string”. For example,

```
	char *string = "To be, or not to be: that is the question.";
	char *brkset = ",;";
	strpbrk(string, brkset);
```

returns the value of the pointer **string** plus six. This points to the comma, which is the first character in the area pointed to by **string** that matches any character in the string pointed to by **brkset**.

Example

This example finds the first white-space character or punctuation character in a string and returns a pointer to it. White space is defined as tab, space, and newline. Punctuation is defined as the following characters:

```
	! @ # $ % ^ & * ( ) - + = ‘ ~
	{ } [ ] : ; ’ " | / , . ?
```

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

char *
findseparator(char *string)
{
	static char separators[] =
		" \n\t!@#$%^&*()-+=‘’~{}[]:;\"|\\/,.?";

	if(string == NULL)
		return(NULL);
```

```
    return strpbrk(string, separators);
}

char string1[]="I shall arise and go now/And go to Innisfree."

main(void)
{
    printf(findseparator(string1));
    return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.11.5.4
The C Programming Language, ed. 2, p. 250

See Also

memchr, strchr, strcspn, string handling, strpbrk, strrchr, strspn, strstr, strtok

Notes

strpbrk resembles the function **strtok** in functionality, but unlike **strtok**, it preserves the contents of the strings being compared. It also resembles the function **strchr**, but lets you search for any one of a group of characters, rather than for one character alone.

strrchr() — String handling (libc)

Search for rightmost occurrence of a character in a string
#include <string.h>
char *strrchr(const char ****string*, int** *character***);**

strrchr looks for the last, or rightmost, occurrence of *character* within *string*. *character* is declared to be an **int**, but is handled within the function as a **char**. Another way to describe this function is to say that it performs a reverse search for a character in a string. It is equivalent to the non-ANSI function **rindex**.

strrchr returns a pointer to the rightmost occurrence of *character*, or NULL if *character* could not be found within *string*.

Example

This example truncates a string by replacing the character after the last terminating character with a zero. It returns the truncated string.

```
#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

char *
truncate(char *string, char endat)
{
    char *endchr;

    if(string!=NULL && (endchr=strrchr(string, endat))!=NULL)
        *++endchr = '\0';
    return(string);
}

char string1[] = "Here we go gathering nuts in May.";
```

```
main(void)
{
      puts(truncate(string1, ','));
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.11.5.5
The C Programming Language, ed. 2, p. 249

See Also

memchr, rindex, strchr, strcspn, string handling, strpbrk, strspn, strstr, strtok

strspn() — String handling (libc)

Return length a string includes characters in another
#include <string.h>
size_t strspn(const char **string1*, **const char** **string2*);

strspn returns the length for which *string1* initially consists only of characters that are found in *string2*. For example,

```
      char *s1 = "hello, world";
      char *s2 = "kernighan & ritchie";
      strcspn(s1, s2);
```

returns two, which is the length for which the first string initially consists of characters found in the second.

Example

This example returns a pointer to the first non-white-space character in a string. White space is defined as a space, tab, or newline character.

```
#include <stdlib.h>
#include <string.h>
#include <stddef.h>
#include <stdio.h>

char *
skipwhite(char *string)
{
      size_t skipcount;

      if (string == NULL)
            return NULL;
      skipcount = strspn(string, "\t \n");
      return(string+skipcount);
}

char string1[] = "\t  Inventor: One who makes an intricate\n";
char string2[] = "arrangement of wheels, levers, and springs,\n;
char string3[] = "       and calls it civilization.\n";

main(void)
{
      printf("%s", skipwhite(string1));
      printf("%s", skipwhite(string2));
      printf("%s", skipwhite(string3));
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.11.5.6
The C Programming Language, ed. 2, p. 250

See Also

memchr, strchr, strcspn, string handling, strpbrk, strrchr, strstr, strtok

strstr() — String handling (libc)

Find one string within another
#include <string.h>
char *strstr(const char ***string1*, const char ***string2***);**

strstr looks for *string2* within *string1*. The terminating null character is not considered part of *string2*.

strstr returns a pointer to where *string2* begins within *string1*, or NULL if *string2* does not occur within *string1*.

For example,

```
char *string1 = "Hello, world";
char *string2 = "world";
strstr(string1, string2);
```

returns **string1** plus seven, which points to the beginning of **world** within **Hello, world**. On the other hand,

```
char *string1 = "Hello, world";
char *string2 = "worlds";
strstr(string1, string2);
```

returns NULL because **worlds** does not occur within **Hello, world**.

Example

This function counts the number of times a pattern appears in a string. The occurrences of the pattern can overlap.

```
#include <stdlib.h>
#include <string.h>
#include <stddef.h>
#include <stdio.h>

size_t
countpat(char *string, char *pattern)
{
	size_t found_count = 0;
	char *found;

	if((found = string)==NULL || pattern==NULL)
		return 0;

	while((found = strstr(found, pattern)) != NULL) {
		/* move past beginning of this one */
		found++;
		/* count it */
		found_count++;
	}
	return(found_count);
}
```

```
char string1[] = "Badges, Badges -- we need no stinking Badges.";
char string2[] = "Badges";

main(void)
{
      printf("%s occurs %d times in %s\n",
            string2, countpat(string1, string2), string1);
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.11.5.7
The C Programming Language, ed. 2, p. 250

See Also

memchr, strchr, strcspn, string handling, strpbrk, strrchr, strspn, strtok

strtod() — General utility (libc)

Convert string to floating-point number
#include <stdlib.h>
double strtod(const char **string*, **char** ***tailptr***);**

strtod converts the string pointed to by *string* to a double-precision floating-point number.

strtod reads the string pointed to by *string*, and parses it into three portions: beginning, subject sequence, and tail.

The *beginning* consists of zero or more white-space characters that begin the string.

The *subject sequence* is the portion of the string that will be converted into a floating-point number. It begins when **strtod** reads a sign character, a numeral, or a decimal-point character. It can include at least one numeral, at most one decimal point, and may end with an exponent marker (either 'e' or 'E') followed by an optional sign and at least one numeral. Reading continues until **strtod** reads either a second decimal-point character or exponent marker, or any other non-numeral.

The *tail* continues from the end of the subject sequence to the null character that ends the string.

strtod ignores the beginning portion of the string. It then converts the subject sequence to a double-precision number and returns it. Finally, it sets the pointer pointed to by *tailptr* to the address of the first character of the string's tail.

strtod returns the **double** generated from the subject sequence. If no subject sequence could be recognized, it returns zero. If the number represented by the subject sequence is too large to fit into a **double**, then **strtod** returns **HUGE_VAL** and sets the global constant **errno** to **ERANGE**. If the number represented by the subject sequence is too small to fit into a **double**, then **strtod** returns zero and again sets **errno** to **ERANGE**.

Example

For an example of using this function in a program, see **sqrt**.

Cross-references

Standard, §4.10.4
The C Programming Language, ed. 2, p. 251

See Also

atof, atoi, atol, errno, general utilities, strtol, strtoul

Notes

The character that **strtod** recognizes as representing the decimal point depends upon the program's locale, as set by the function **setlocale**. See **localization** for more information.

Initial white space in the string pointed to by *string* is ignored. White space is defined as being all characters so recognized by the function **isspace**; the current locale setting may affect the operation of **isspace**.

strtok() — String handling (libc)

Break a string into tokens
#include <string.h>
char *strtok(char ****string1*, const char ****string2***);**

strtok helps to divide a string into a set of tokens. *string1* points to the string to be divided, and *string2* points to the character or characters that delimit the tokens.

strtok divides a string into tokens by being called repeatedly.

On the first call to **strtok**, *string1* should point to the string being divided. **strtok** searches for a character that is *not* included within *string2*. If it finds one, then **strtok** regards it as the beginning of the first token within the string. If one cannot be found, then **strtok** returns NULL to signal that the string could not be divided into tokens. When the beginning of the first token is found, **strtok** then looks for a character that *is* included within *string2*. When one is found, **strtok** replaces it with a null character to mark the end of the first token, stores a pointer to the remainder of *string1* within a static buffer, and returns the address of the beginning of the first token.

On subsequent calls to **strtok**, set *string1* to NULL. **strtok** then looks for subsequent tokens, using the address that it saved from the first call. With each call to **strtok**, *string2* may point to a different delimiter or set of delimiters.

Example

The following example breaks **command_string** into individual tokens and puts pointers to the tokens into the array **tokenlist[]**. It then returns the number of tokens created. No more than **maxtoken** tokens will be created. **command_string** is modified to place '\0' over token separators. The token list points into **command_string**. Tokens are separated by spaces, tabs, commas, semicolons, and newlines.

```
#include <stdlib.h>
#include <string.h>
#include <stddef.h>
#include <stdio.h>

tokenize(char *command_string, char *tokenlist[],
      size_t maxtoken)
{
      static char tokensep[]="\t\n ,;";
      int tokencount;
      char *thistoken;

      if(command_string == NULL || !maxtoken)
            return 0;

      thistoken = strtok(command_string, tokensep);

      for(tokencount = 0; tokencount < maxtoken &&
                  thistoken != NULL;) {
            tokenlist[tokencount++] = thistoken;
            thistoken = strtok(NULL, tokensep);
      }
```

```
      tokenlist[tokencount] = NULL;
      return tokencount;
}

#define MAXTOKEN 100
char *tokens[MAXTOKEN];
char buf[80];

main(void)
{
      for(;;) {
            int i, j;

            printf("Enter string ");
            fflush(stdout);
            if(gets(buf) == NULL)
                  exit(EXIT_SUCCESS);

            i = tokenize(buf, tokens, MAXTOKEN);
            for(j = 0; j < i; j++)
                  printf("%s\n", tokens[j]);
      }
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.11.5.8
The C Programming Language, ed. 2, p. 250

See Also

memchr, strchr, strcspn, string handling, strpbrk, strrchr, strspn, strstr

strtol() — General utility (libc)

Convert string to long integer
#include <stdlib.h>
long strtol(const char **sptr*, **char** ***tailptr*, **int** *base*);

strtol converts the string pointed to by *sptr* into a **long**.

base gives the base of the number being read, from 0 to 36. This governs the form of the number that **strtol** expects. If *base* is zero, then **strtol** expects a number in the form of an integer constant. See **integer constant** for more information. If *base* is set to 16, then the string to be converted may be preceded by **0x** or **0X**.

strtol reads the string pointed to by *sptr* and parses it into three portions: beginning, subject sequence, and tail.

The *beginning* consists of zero or more white-space characters that begin the string.

The *subject sequence* is the portion of the string that will be converted into a **long**. It is introduced by a sign character, a numeral, or an alphabetic character appropriate to the base of the number being read. For example, if *base* is set to 16, then **strtol** will recognize the alphabetic characters 'A' through 'F' and 'a' to 'f' as indicating numbers. It continues to scan until it encounters any alphabetic character outside the set recognized for the setting of *base*, or the null character.

The *tail* continues from the end of the subject sequence to the null character that ends the string.

strtol ignores the beginning portion of the string. It then converts the subject sequence to a **long**. Finally, it sets the pointer pointed to by *tailptr* to the address of the first character of the string's tail.

strtol returns the **long** that it has built from the subject sequence. If it could not build a number, for whatever reason, it returns zero. If the number it builds is too large or too small to fit into a **long**, it returns, respectively, **LONG_MAX** or **LONG_MIN** and sets the global variable **errno** to the value of the macro **ERANGE**.

Cross-references

Standard, §4.10.1.5
The C Programming Language, ed. 2, p. 252

See Also

atof, atoi, atol, errno, general utility, strtod, strtoul

Notes

Initial white space in the string pointed to by *string* is ignored. White space is defined as being all characters so recognized by the function **isspace**; the current locale setting may affect the operation of **isspace**.

strtoul() — General utility (libc)

Convert string to unsigned long integer
#include <stdlib.h>
unsigned long strtoul(const char ****sptr*, char *****tailptr***, int** *base***);**

strtoul converts the string pointed to by *sptr* into an **unsigned long**.

base gives the base of the number being read, from 0 to 36. This governs the form of the number that **strtoul** expects. If *base* is zero, then **strtoul** expects a number in the form of an integer constant. See **integer constant** for more information. If *base* is set to 16, then the string to be converted may be preceded by **0x** or **0X**.

strtoul reads the string pointed to by *sptr* and parses it into three portions: beginning, subject sequence, and tail.

The *beginning* consists of zero or more white-space characters that begin the string.

The *subject sequence* is the portion of the string that will be converted into an **unsigned long**. It is introduced by a sign character, a numeral, or an alphabetic character appropriate to the base of the number being read. For example, if *base* is set to 16, then **strtoul** will recognize the alphabetic characters 'A' through 'F' and 'a' to 'f' as indicating numbers. It continues to scan until it encounters any alphabetic character outside the set recognized or the setting of *base*, or the null character.

The *tail* continues from the end of the subject sequence to the null character that ends the string.

strtoul ignores the beginning portion of the string. It then converts the subject sequence to an **unsigned long**. Finally, it sets the pointer pointed to by *tailptr* to the address of the first character of the string's tail.

strtoul returns the **unsigned long** that it has built from the subject sequence. If it could not build a number, for whatever reason, it returns zero. If the number it builds is too large to fit into an **unsigned long**, it returns **ULONG_MAX** and sets the global variable **errno** to the value of the macro **ERANGE**.

Example

This example uses **strtoul** as a hash function for table lookup. It demonstrates both hashing and linked lists. Hash-table lookup is the most efficient when used to look up entries in large tables; this is an example only.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/*
 * For fastest results, use a prime about 15% bigger
 * than the table. If short of space, use a smaller prime.
 */
#define HASHP 11
struct symbol {
      struct symbol *next;
      char *name;
      char *descr;
} *hasht[HASHP], codes[] = {

      NULL,       "a286",           "frogs togs",
      NULL,       "xy7800",         "doughnut holes",
      NULL,       "z678abc",        "used bits",
      NULL,       "xj781",          "black-hole varnish",
      NULL,       "h778a",          "table hash",
      NULL,       "q167",           "log(-5.2)",
      NULL,       "18888",          "quid pro quo",
      NULL,       NULL,             NULL   /* end marker */
};

void
buildTable(void)
{
      long h;
      register struct symbol *sym, **symp;

      for(symp = hasht; symp != (hasht + HASHP); symp++)
            *symp = NULL;

      for(sym = codes; sym->descr != NULL; sym++) {
            /*
             * hash by converting to base 36. There are
             * many ways to hash, but use all the data.
             */

            h = strtoul(sym->name, NULL, 36) % HASHP;
            sym->next = hasht[h];
            hasht[h] = sym;
      }
}

struct symbol *
lookup(char *s)
{
      long h;
      register struct symbol *sym;

      h = strtoul(s, NULL, 36) % HASHP;
      for(sym = hasht[h]; sym != NULL; sym = sym->next)
            if(!strcmp(sym->name, s))
                  return(sym);
      return(NULL);
}
```

```
main(void)
{
	char buf[80];
	struct symbol *sym;

	buildTable();
	for(;;) {
		printf("Enter name ");
		fflush(stdout);

		if(gets(buf) == NULL)
			exit(EXIT_SUCCESS);

		if((sym = lookup(buf)) == NULL)
			printf("%s not found\n", buf);

		else
			printf("%s is %s\n", buf, sym->descr);
	}
	return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.10.1.5
The C Programming Language, ed. 2, p. 252

See Also

atof, atoi, atol, general utilities, strtod, strtol

Notes

This function has no historical usage, but provides greater functionality than does **strtol**.

Initial white space in the string pointed to by *string* is ignored. White space is defined as being all characters so recognized by the function **isspace**. The current locale setting may affect the operation of **isspace**.

struct — C keyword

The keyword **struct** introduces a *structure*. This is an aggregate data type that consists of a number of fields, or *members*, each of which can have its own name and type.

The members of a structure are stored sequentially. Unlike the related type **union**, the elements of a **struct** do not overlap. Thus, the size of a **struct** is the total of the sizes of all of its members, plus any bytes used for alignment (if the implementation requires them). Aligning bytes may not be inserted at the beginning of a **struct**, but may appear in its middle, or at the end. For this reason, it is incorrect to assume that any two members of a structure abut each other in memory.

Any type may be used within a **struct**, including bit-fields. No incomplete type may be used; thus, a **struct** may not contain a copy of itself, but it may contain a pointer to itself. A **struct** is regarded as incomplete until its closing '}' is read.

The members of a **struct** are stored in the order in which they are declared. Thus, a pointer to a **struct** also points to the beginning of the **struct**'s first member.

The following is an example of a structure:

```
struct person {
      char name[30];
      char st_address[25];
      char city[20];
      char state[2];
      char zip[9];
      char id_number[9];
} MYSELF;
```

This example defines a structure type **person**, as well as an instance of this type, called **MYSELF**.

Cross-references

Standard, §3.1.2.5, §3.5.2.1
The C Programming Language, ed. 2, pp. 127*ff*

See Also

alignment, member name, tag, types, union

strxfrm() — String handling (libc)

Transform a string
#include <string.h>
size_t strxfrm(char **string1*, **const char** **string2*, **size_t** *n***);**

strxfrm transforms *string2* using information concerning the program's locale, as set by the function **setlocale**. See **localization** for more information about setting a program's locale.

strxfrm writes up to *n* bytes of the transformed result into the area pointed to by *string1*. It returns the length of the transformed string, not including the terminating null character. The transformation incorporates locale-specific material into *string2*.

If *n* is set to zero, **strxfrm** returns the length of the transformed string.

If two strings return a given result when compared by **strcoll** before transformation, they will return the same result when compared by **strcmp** after transformation.

Cross-references

Standard, §4.11.4.5
The C Programming Language, ed. 2, p. 250

See Also

localization, memcmp, strcmp, strcoll, string handling, strncmp

Notes

If **strxfrm** returns a value equal to or greater than *n*, the contents of the area pointed to by *string1* are indeterminate.

swab() — Extended function (libc)

Swap a pair of bytes
void swab(char **src*, **char** **dest*, **unsigned short** *nb***);**

The ordering of bytes within a word differs from machine to machine. This may cause problems when moving binary data between machines. **swab** interchanges each pair of bytes in the array *src* that is *n* bytes long, and writes the result into the array *dest*. The length *nb* should be an even number, or the last byte will not be touched. *src* and *dest* may be the same place.

Example

This example prompts for an integer; it then prints the integer both as you entered it, and as it appears with its bytes swapped.

```
#include <stdio.h>
#include <stdlib.h>
extern void swab(char *src, char *dest, unsigned short nb);

main(void)
{
	short word;

	printf("Enter an integer: \n");
	scanf("%d", &word);
	printf("The word is 0x%x\n", word);
	swab(&word, &word, 2);
	printf("The word with bytes swapped is 0x%x\n", word);
	return(EXIT_SUCCESS);
}
```

See Also

byte ordering, extended miscellaneous

switch — C keyword

Select an entry in a table
switch (*expression* **)** *statement*

switch evaluates *expression*, jumps to the **case** label whose expression is equal to *expression*, and continues execution from there. *expression* may evaluate to any integral type, not just an **int**. Every **case** label's *expression* is cast to the type of *conditional* before it is compared with *expression*.

If no **case** expression matches *expression*, **switch** jumps to the point marked by the **default** label. If there is no default label, then **switch** does not jump and no statement is executed; execution then continues from the '}' that marks the end of the **switch** statement.

The program continues its execution from the point to which **switch** jumps, either until a **break**, **continue**, **goto**, or **return** statement is read, or until the '}' that encloses all of the **case** statements is encountered.

All **case** labels are subordinate to the closest enclosing **switch** statement. No two **case** labels can have expressions with the same value. However, if a **case** label introduces a secondary **switch** statement, then that **switch** statement's suite of **case** labels may duplicate the values used by the **case** labels of the outer **switch** statement.

Example

For an example of this statement, see **printf**.

Cross-references

Standard, §3.6.4.2
The C Programming Language, ed. 2, pp. 58*ff*

See Also

break, case, default, if, statements

Notes

It is good programming practice always to use a **default** label with a **switch** statement. There may be only one **default** label with any **switch** statement.

The number of **case** labels that can be included with a **switch** statement may vary from implementation to implementation. The Standard requires that every conforming implementation allow a **switch** statement to have up to at least 257 **case** labels.

The first edition of *The C Programming Language* requires that *conditional* may evaluate to an **int**. The Standard lifts this requirement: *conditional* may now be any integral type, from **short** to **unsigned long**. Every *expression* associated with a **case** label will be altered to conform to the type of *conditional*. Therefore, if a program depends upon *conditional* or any *expression* being an **int**, it may work differently under a conforming translator. This is a quiet change that may break existing code.

system() — General utility (libc)

Suspend a program and execute another
#include <stdlib.h>
int system(const char **program***);**

system provides a way to execute another program from within a C program. It suspends the program currently being run, and passes the name pointed to by *program* to MS-DOS. When *program* has finished executing, MS-DOS returns to the current program, which then continues its operation.

If *program* is set to NULL, **system** checks to see if a command processor exists. In this case, **system** returns zero if a command processor does not exist and nonzero if it does. If *program* is set to any value other than NULL, then what **system** returns is defined by the implementation.

Example

This example execute system commands on request.

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

syscmds(char * prompt)
{
      for(;;) {
            char buf[80];

            printf(prompt);
            fflush(stdout);
            if(gets(buf) == NULL || !strcmp(buf, "exit"))
                  return;
            system(buf);
      }
}

main(void)
{
      printf("Enter system commands: ");
      syscmds(">");
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.10.4.5
The C Programming Language, ed. 2, p. 253

See Also

command processor, exit, general utilities