Prev | Contents | Next

25 <string.h> String Manipulation

Function	Description
memchr()	Find the first occurrence of a character in memory.
memcmp()	Compare two regions of memory.
memcpy()	Copy a region of memory to another.
memmove()	Move a (potentially overlapping) region of memory.
memset()	Set a region of memory to a value.
strcat()	Concatenate (join) two strings together.
strchr()	Find the first occurrence of a character in a string.
strcmp()	Compare two strings.
strcoll()	Compare two strings accounting for locale.
strcpy()	Copy a string.
strcspn()	Find length of a string not consisting of a set of characters.
strerror()	Return a human-readable error message for a given code.
strlen()	Return the length of a string.
strncat()	Concatenate (join) two strings, length-limited.
strncmp()	Compare two strings, length-limited.
strncpy()	Copy two strings, length-limited.
strpbrk()	Search a string for one of a set of character.
strrchr()	Find the last occurrence of a character in a string.
strspn()	Find length of a string consisting of a set

Function	Description
	of characters.
strstr()	Find a substring in a string.
strtok()	Tokenize a string.
strxfrm()	Prepare a string for comparison as if by strcoll().

As has been mentioned earlier in the guide, a string in C is a sequence of bytes in memory, terminated by a NUL character ('\0'). The NUL at the end is important, since it lets all these string functions (and printf() and puts() and everything else that deals with a string) know where the end of the string actually is.

Fortunately, when you operate on a string using one of these many functions available to you, they add the NUL terminator on for you, so you actually rarely have to keep track of it yourself. (Sometimes you do, especially if you're building a string from scratch a character at a time or something.)

In this section you'll find functions for pulling substrings out of strings, concatenating strings together, getting the length of a string, and so forth and so on.

25.1 memcpy(), memmove()

Copy bytes of memory from one location to another

Synopsis

```
#include <string.h>

void *memcpy(void * restrict s1, const void * restrict s2, size_t
n);

void *memmove(void *s1, const void *s2, size_t n);
```

Description

These functions copy memory—as many bytes as you want! From source to destination!

The main difference between the two is that memcpy() cannot safely copy overlapping memory regions, whereas memmove() can.

On the one hand, I'm not sure why you'd want to ever use memcpy() instead of memmove(), but I'll bet it's possibly more performant.

The parameters are in a particular order: destination first, then source. I remember this order because it behaves like an "=" assignment: the destination is on the left.

Return Value

Both functions return whatever you passed in for parameter `s1` for your convenience.

Example

```
#include <string.h>

int main(void)
{
    char s[100] = "Goats";
    char t[100];

    memcpy(t, s, 6);       // Copy non-overlapping memory

    memmove(s + 2, s, 6);  // Copy overlapping memory
}
```

See Also

`strcpy()`, `strncpy()`

25.2 `strcpy()`, `strncpy()`

Copy a string

Synopsis

```
#include <string.h>

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

char *strncpy(char *dest, char *src, size_t n);
```

Description

These functions copy a string from one address to another, stopping at the NUL terminator on the `src` string.

`strncpy()` is just like `strcpy()`, except only the first `n` characters are actually copied. Beware that if you hit the limit, `n` before you get a NUL terminator on the `src` string, your `dest` string won't be NUL-terminated. Beware! BEWARE!

(If the `src` string has fewer than `n` characters, it works just like `strcpy()`.)

You can terminate the string yourself by sticking the `'\0'` in there yourself:

```
char s[10];
char foo = "My hovercraft is full of eels."; // more than 10 chars

strncpy(s, foo, 9); // only copy 9 chars into positions 0-8
s[9] = '\0';        // position 9 gets the terminator
```

Return Value

Both functions return `dest` for your convenience, at no extra charge.

Example

```
#include <string.h>

int main(void)
{
    char *src = "hockey hockey hockey hockey hockey hockey hockey hockey";
    char dest[20];

    int len;

    strcpy(dest, "I like "); // dest is now "I like "

    len = strlen(dest);

    // tricky, but let's use some pointer arithmetic and math to append
    // as much of src as possible onto the end of dest, -1 on the length to
    // leave room for the terminator:
    strncpy(dest+len, src, sizeof(dest)-len-1);

    // remember that sizeof() returns the size of the array in bytes
    // and a char is a byte:
```

```
21      dest[sizeof(dest)-1] = '\0'; // terminate
22
23      // dest is now:       v null terminator
24      // I like hockey hocke
25      // 012345678901234567890123456
26  }
```

See Also

memcpy(), strcat(), strncat()

25.3 strcat(), strncat()

Concatenate two strings into a single string

Synopsis

```
#include <string.h>

int strcat(const char *dest, const char *src);

int strncat(const char *dest, const char *src, size_t n);
```

Description

“Concatenate”, for those not in the know, means to “stick together”. These functions take two strings, and stick them together, storing the result in the first string.

These functions don’t take the size of the first string into account when it does the concatenation. What this means in practical terms is that you can try to stick a 2 megabyte string into a 10 byte space. This will lead to unintended consequences, unless you intended to lead to unintended consequences, in which case it will lead to intended unintended consequences.

Technical banter aside, your boss and/or professor will be irate.

If you want to make sure you don’t overrun the first string, be sure to check the lengths of the strings first and use some highly technical subtraction to make sure things fit.

You can actually only concatenate the first `n` characters of the second string by using `strncat()` and specifying the maximum number of characters to copy.

Return Value

Both functions return a pointer to the destination string, like most of the string-oriented functions.

Example

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

int main(void)
{
    char dest[30] = "Hello";
    char *src = ", World!";
    char numbers[] = "12345678";

    printf("dest before strcat: \"%s\"\n", dest); // "Hello"

    strcat(dest, src);
    printf("dest after strcat:  \"%s\"\n", dest); // "Hello, world!"

    strncat(dest, numbers, 3); // strcat first 3 chars of numbers
    printf("dest after strncat: \"%s\"\n", dest); // "Hello, world!123"
}
```

Notice I mixed and matched pointer and array notation there with `src` and `numbers`; this is just fine with string functions.

See Also

`strlen()`

25.4 `strcmp()`, `strncmp()`, `memcmp()`

Compare two strings or memory regions and return a difference

Synopsis

```
#include <string.h>

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

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

int memcmp(const void *s1, const void *s2, size_t n);
```

Description

All these functions compare chunks of bytes in memory.

`strcmp()` and `strncmp()` operate on NUL-terminated strings, whereas `memcmp()` will compare the number of bytes you specify, brazenly ignoring any NUL characters it finds along the way.

`strcmp()` compares the entire string down to the end, while `strncmp()` only compares the first `n` characters of the strings.

It's a little funky what they return. Basically it's a difference of the strings, so if the strings are the same, it'll return zero (since the difference is zero). It'll return non-zero if the strings differ; basically it will find the first mismatched character and return less-than zero if that character in `s1` is less than the corresponding character in `s2`. It'll return greater-than zero if that character in `s1` is greater than that in `s2`.

So if they return `0`, the comparison was equal (i.e. the difference was `0`.)

These functions can be used as comparison functions for `qsort()` if you have an array of `char*`s you want to sort.

Return Value

Returns zero if the strings or memory are the same, less-than zero if the first different character in `s1` is less than that in `s2`, or greater-than zero if the first difference character in `s1` is greater than than in `s2`.

Example

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main(void)
5  {
```

```
 6      char *s1 = "Muffin";
 7      char *s2 = "Muffin Sandwich";
 8      char *s3 = "Muffin";
 9
10      int r1 = strcmp("Biscuits", "Kittens");
11      printf("%d\n", r1); // prints < 0 since 'B' < 'K'
12
13      int r2 = strcmp("Kittens", "Biscuits");
14      printf("%d\n", r2); // prints > 0 since 'K' > 'B'
15
16      if (strcmp(s1, s2) == 0)
17          printf("This won't get printed because the strings
differ\n");
18
19      if (strcmp(s1, s3) == 0)
20          printf("This will print because s1 and s3 are the
same\n");
21
22      // this is a little weird...but if the strings are the
same, it'll
23      // return zero, which can also be thought of as
"false". Not-false
24      // is "true", so (!strcmp()) will be true if the
strings are the
25      // same. yes, it's odd, but you see this all the time
in the wild
26      // so you might as well get used to it:
27
28      if (!strcmp(s1, s3))
29          printf("The strings are the same!\n");
30
31      if (!strncmp(s1, s2, 6))
32          printf("The first 6 characters of s1 and s2 are the
same\n");
33  }
```

See Also

memcmp(), qsort()

25.5 strcoll()

Compare two strings accounting for locale

Synopsis

```
#include <string.h>

int strcoll(const char *s1, const char *s2);
```

Description

This is basically `strcmp()`, except that it handles accented characters better depending on the locale.

For example, my `strcmp()` reports that the character “é” (with accent) is greater than “f”. But that’s hardly useful for alphabetizing.

By setting the `LC_COLLATE` locale value (either by name or via `LC_ALL`), you can have `strcoll()` sort in a way that’s more meaningful by the current locale. For example, by having “é” appear sanely *before* “f”.

It’s also a lot slower than `strcmp()` so use it only if you have to. See `strxfrm()` for a potential speedup.

Return Value

Like the other string comparison functions, `strcoll()` returns a negative value if `s1` is less than `s2`, or a positive value if `s1` is greater than `s2`. Or `0` if they are equal.

Example

```
 1  #include <stdio.h>
 2  #include <string.h>
 3  #include <locale.h>
 4
 5  int main(void)
 6  {
 7      setlocale(LC_ALL, "");
 8
 9      // If your source character set doesn't support "é" in
a string
10      // you can replace it with `\u00e9`, the Unicode code
point
11      // for "é".
12
```

```
13      printf("%d\n", strcmp("é", "f"));   // Reports é > f, yuck.
14      printf("%d\n", strcoll("é", "f"));  // Reports é < f, yay!
15  }
```

See Also

strcmp()

25.6 strxfrm()

Transform a string for comparing based on locale

Synopsis

```
#include <string.h>

size_t strxfrm(char * restrict s1, const char * restrict s2, size_t n);
```

Description

This is a strange little function, so bear with me.

Firstly, if you haven't done so, get familiar with strcoll() because this is closely related to that.

OK! Now that you're back, you can think of strxfrm() as the first part of the strcoll() internals. Basically, strcoll() has to transform a string into a form that can be compared with strcmp(). And it does this with strxfrm() for both strings every time you call it.

strxform() takes string s2 and transforms it (readies it for strcmp()) storing the result in s1. It writes no more than n bytes, protecting us from terrible buffer overflows.

But hang on—there's another mode! If you pass NULL for s1 and 0 for n, it will return the number of bytes that the transformed string *would have* used[64]. This is useful if you need to allocate some space to hold the transformed string before you strcmp() it against another.

What I'm getting at, not to be too blunt, is that `strcoll()` is slow compared to `strcmp()`. It does a lot of extra work running `strxfrm()` on all its strings.

In fact, we can see how it works by writing our own like this:

```
 1  int my_strcoll(char *s1, char *s2)
 2  {
 3      // Use n = 0 to just get the lengths of the transformed
strings
 4      int len1 = strxfrm(NULL, s1, 0) + 1;
 5      int len2 = strxfrm(NULL, s2, 0) + 1;
 6
 7      // Allocate enough room for each
 8      char *d1 = malloc(len1);
 9      char *d2 = malloc(len2);
10
11      // Transform the strings for comparison
12      strxfrm(d1, s1, len1);
13      strxfrm(d2, s2, len2);
14
15      // Compare the transformed strings
16      int result = strcmp(d1, d2);
17
18      // Free up the transformed strings
19      free(d2);
20      free(d1);
21
22      return result;
23  }
```

You see on lines 12, 13, and 16, above how we transform the two input strings and then call `strcmp()` on the result.

So why do we have this function? Can't we just call `strcoll()` and be done with it?

The idea is that if you have one string that you're going to be comparing against a whole lot of other ones, maybe you just want to transform that string one time, then use the faster `strcmp()` saving yourself a bunch of the work we had to do in the function, above.

We'll do that in the example.

Return Value

Returns the number of bytes in the transformed sequence. If the value is greater than `n`, the results in `s1` are meaningless.

Example

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

// Transform a string for comparison, returning a malloc'd
// result
char *get_xfrm_str(char *s)
{
    int len = strxfrm(NULL, s, 0) + 1;
    char *d = malloc(len);

    strxfrm(d, s, len);

    return d;
}

// Does half the work of a regular strcoll() because the
second
// string arrives already transformed.
int half_strcoll(char *s1, char *s2_transformed)
{
    char *s1_transformed = get_xfrm_str(s1);

    int result = strcmp(s1_transformed, s2_transformed);

    free(s1_transformed);

    return result;
}

int main(void)
{
    setlocale(LC_ALL, "");

    // Pre-transform the string to compare against
```

```
36      char *s = get_xfrm_str("éfg");
37
38      // Repeatedly compare against "éfg"
39      printf("%d\n", half_strcoll("fgh", s));  // "fgh" > "éfg"
40      printf("%d\n", half_strcoll("àbc", s));  // "àbc" < "éfg"
41      printf("%d\n", half_strcoll("ĥij", s));  // "ĥij" > "éfg"
42
43      free(s);
44  }
```

See Also

strcoll()

25.7 strchr(), strrchr(), memchr()

Find a character in a string

Synopsis

```
#include <string.h>

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

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

void *memchr(const void *s, int c, size_t n);
```

Description

The functions `strchr()` and `strrchr` find the first or last occurrence of a letter in a string, respectively. (The extra "r" in `strrchr()` stands for "reverse"–it looks starting at the end of the string and working backward.) Each function returns a pointer to the char in question, or `NULL` if the letter isn't found in the string.

`memchr()` is similar, except that instead of stopping on the first NUL character, it continues searching for however many bytes you specify.

Quite straightforward.

One thing you can do if you want to find the next occurrence of the letter after finding the first, is call the function again with the previous return value plus one. (Remember pointer arithmetic?) Or minus one if you're looking in reverse. Don't accidentally go off the end of the string!

Return Value

Returns a pointer to the occurrence of the letter in the string, or `NULL` if the letter is not found.

Example

```
 1  #include <stdio.h>
 2  #include <string.h>
 3
 4  int main(void)
 5  {
 6      // "Hello, world!"
 7      //       ^  ^   ^
 8      //       A  B   C
 9
10      char *str = "Hello, world!";
11      char *p;
12
13      p = strchr(str, ',');       // p now points at position A
14      p = strrchr(str, 'o');      // p now points at position B
15
16      p = memchr(str, '!', 13);   // p now points at position C
17
18      // repeatedly find all occurrences of the letter 'B'
19      str = "A BIG BROWN BAT BIT BEEJ";
20
21      for(p = strchr(str, 'B'); p != NULL; p = strchr(p + 1, 'B')) {
22          printf("Found a 'B' here: %s\n", p);
23      }
24  }
```

Output:

```
Found a 'B' here: BIG BROWN BAT BIT BEEJ
Found a 'B' here: BROWN BAT BIT BEEJ
Found a 'B' here: BAT BIT BEEJ
Found a 'B' here: BIT BEEJ
Found a 'B' here: BEEJ
```

25.8 `strspn()`, `strcspn()`

Return the length of a string consisting entirely of a set of characters, or of not a set of characters

Synopsis

```
#include <string.h>

size_t strspn(char *str, const char *accept);

size_t strcspn(char *str, const char *reject);
```

Description

`strspn()` will tell you the length of a string consisting entirely of the set of characters in `accept`. That is, it starts walking down `str` until it finds a character that is *not* in the set (that is, a character that is not to be accepted), and returns the length of the string so far.

`strcspn()` works much the same way, except that it walks down `str` until it finds a character in the `reject` set (that is, a character that is to be rejected.) It then returns the length of the string so far.

Return Value

The length of the string consisting of all characters in `accept` (for `strspn()`), or the length of the string consisting of all characters except `reject` (for `strcspn()`).

Example

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main(void)
5  {
```

```
 6      char str1[] = "a banana";
 7      char str2[] = "the bolivian navy on maenuvers in the
south pacific";
 8      int n;
 9
10      // how many letters in str1 until we reach something
that's not a vowel?
11      n = strspn(str1, "aeiou");
12      printf("%d\n", n);  // n == 1, just "a"
13
14      // how many letters in str1 until we reach something
that's not a, b,
15      // or space?
16      n = strspn(str1, "ab ");
17      printf("%d\n", n);  // n == 4, "a ba"
18
19      // how many letters in str2 before we get a "y"?
20      n = strcspn(str2, "y");
21      printf("%d\n", n);  // n = 16, "the bolivian nav"
22  }
```

See Also

strchr(), strrchr()

25.9 strpbrk()

Search a string for one of a set of characters

Synopsis

```
#include <string.h>

char *strpbrk(const char *s1, const char *s2);
```

Description

This function searches string `s1` for any of the characters that are found in string `s2`.

It's just like how `strchr()` searches for a specific character in a string, except it will match *any* of the characters found in `s2`.

Think of the power!

Return Value

Returns a pointer to the first character matched in `s1`, or NULL if the string isn't found.

Example

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

int main(void)
{
    //  p points here after strpbrk
    //              v
    char *s1 = "Hello, world!";
    char *s2 = "dow!";  // Match any of these chars

    char *p = strpbrk(s1, s2);  // p points to the o

    printf("%s\n", p);  // "o, world!"
}
```

See Also

`strchr()`, `memchr()`

25.10 `strstr()`

Find a string in another string

Synopsis

```
#include <string.h>

char *strstr(const char *str, const char *substr);
```

Description

Let's say you have a big long string, and you want to find a word, or whatever substring strikes your fancy, inside the first string. Then `strstr()` is for you! It'll return a pointer to the `substr` within the `str`!

Return Value

You get back a pointer to the occurrence of the `substr` inside the `str`, or `NULL` if the substring can't be found.

Example

```
 1  #include <stdio.h>
 2  #include <string.h>
 3
 4  int main(void)
 5  {
 6      char *str = "The quick brown fox jumped over the lazy
dogs.";
 7      char *p;
 8
 9      p = strstr(str, "lazy");
10      printf("%s\n", p == NULL? "null": p); // "lazy dogs."
11
12      // p is NULL after this, since the string "wombat"
isn't in str:
13      p = strstr(str, "wombat");
14      printf("%s\n", p == NULL? "null": p); // "null"
15  }
```

See Also

`strchr()`, `strrchr()`, `strspn()`, `strcspn()`

25.11 `strtok()`

Tokenize a string

Synopsis

```
#include <string.h>

char *strtok(char *str, const char *delim);
```

Description

If you have a string that has a bunch of separators in it, and you want to break that string up into individual pieces, this function can do it for you.

The usage is a little bit weird, but at least whenever you see the function in the wild, it's consistently weird.

Basically, the first time you call it, you pass the string, `str` that you want to break up in as the first argument. For each subsequent call to get more tokens out of the string, you pass `NULL`. This is a little weird, but `strtok()` remembers the string you originally passed in, and continues to strip tokens off for you.

Note that it does this by actually putting a NUL terminator after the token, and then returning a pointer to the start of the token. So the original string you pass in is destroyed, as it were. If you need to preserve the string, be sure to pass a copy of it to `strtok()` so the original isn't destroyed.

Return Value

A pointer to the next token. If you're out of tokens, `NULL` is returned.

Example

```
 1  #include <stdio.h>
 2  #include <string.h>
 3
 4  int main(void)
 5  {
 6      // break up the string into a series of space or
 7      // punctuation-separated words
 8      char str[] = "Where is my bacon, dude?";
 9      char *token;
10
11      // Note that the following if-do-while construct is
very very
12      // very very very common to see when using strtok().
13
```

```
14      // grab the first token (making sure there is a first
token!)
15      if ((token = strtok(str, ".,?! ")) != NULL) {
16          do {
17              printf("Word: \"%s\"\n", token);
18
19              // now, the while continuation condition grabs
the
20              // next token (by passing NULL as the first
param)
21              // and continues if the token's not NULL:
22          } while ((token = strtok(NULL, ".,?! ")) != NULL);
23      }
24  }
```

Output:

```
Word: "Where"
Word: "is"
Word: "my"
Word: "bacon"
Word: "dude"
```

See Also

`strchr()`, `strrchr()`, `strspn()`, `strcspn()`

25.12 `memset()`

Set a region of memory to a certain value

Synopsis

```
#include <string.h>

void *memset(void *s, int c, size_t n);
```

Description

This function is what you use to set a region of memory to a particular value, namely `c` converted into `unsigned char`.

The most common usage is to zero out an array or `struct`.

Return Value

`memset()` returns whatever you passed in as `s` for happy convenience.

Example

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

int main(void)
{
    struct banana {
        float ripeness;
        char *peel_color;
        int grams;
    };

    struct banana b;

    memset(&b, 0, sizeof b);

    printf("%d\n", b.ripeness == 0.0);     // True
    printf("%d\n", b.peel_color == NULL);  // True
    printf("%d\n", b.grams == 0);          // True
}
```

See Also

`memcpy()`, `memmove()`

25.13 `strerror()`

Get a string version of an error number

Synopsis

```
#include <string.h>

char *strerror(int errnum);
```

Description

This function ties closely into `perror()` (which prints a human-readable error message corresponding to `errno`). But instead of printing, `strerror()` returns a pointer to the locale-specific error message string.

So if you ever need that string back for some reason (e.g. you're going to `fprintf()` it to a file or something), this function will give it to you. All you need to do is pass in `errno` as an argument. (Recall that `errno` gets set as an error status by a variety of functions.)

You can actually pass in any integer for `errnum` you want. The function will return *some* message, even if the number doesn't correspond to any known value for `errno`.

The values of `errno` and the strings returned by `strerror()` are system-dependent.

Return Value

A string error message corresponding to the given error number.

You are not allowed to modify the returned string.

Example

```
 1  #include <stdio.h>
 2  #include <string.h>
 3  #include <errno.h>
 4
 5  int main(void)
 6  {
 7      FILE *fp = fopen("NONEXISTENT_FILE.TXT", "r");
 8
 9      if (fp == NULL) {
10          char *errmsg = strerror(errno);
11          printf("Error %d opening file: %s\n", errno,
errmsg);
12      }
13  }
```

Output:

```
Error 2 opening file: No such file or directory
```

See Also

`perror()`

25.14 `strlen()`

Returns the length of a string

Synopsis

```
#include <string.h>

size_t strlen(const char *s);
```

Description

This function returns the length of the passed null-terminated string (not counting the NUL character at the end). It does this by walking down the string and counting the bytes until the NUL character, so it's a little time consuming. If you have to get the length of the same string repeatedly, save it off in a variable somewhere.

Return Value

Returns the number of bytes in the string. Note that this might be different than the number of characters in a multibyte string.

Example

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main(void)
5  {
6      char *s = "Hello, world!"; // 13 characters
7
8      // prints "The string is 13 characters long.":
9
10     printf("The string is %zu characters long.\n",
strlen(s));
11 }
```

See Also

Prev | Contents | Next