

## Re: strtok ( ) help

---

*Source:* [http://coding.derkeiler.com/Archive/C\\_CPP/comp.lang.c/2006-01/msg03242.html](http://coding.derkeiler.com/Archive/C_CPP/comp.lang.c/2006-01/msg03242.html)

---

- *From:* "Gregory Pietsch" <GKPI@xxxxxxxxxx>
  - *Date:* 22 Jan 2006 12:57:42 -0800
- 

ern wrote:

```
> I'm using strtok( ) to capture lines of input. After I call
> "splitCommand", I call strtok( ) again to get the next line. Strtok( )
> returns NULL (but there is more in the file...). That didn't happen
> before 'splitCommands' entered the picture. The problem is in
> splitCommands( ) somehow modifying the pointer, but I HAVE to call that
> function. Is there a way to make a copy of it or something ?
>
> /* HERE IS MY CODE */
```

<code snipped; too easy to rewrite for debugging purposes>

First of all, you don't have to call any function if you know how to manipulate strings. ;-)

Here's an idea of how to use the strtok() function. Assuming that you don't mind trashing the contents of a string s, and t is your token pointer:

```
for (t = s; (t = strtok(t, delimiters)) != 0; t = 0)
```

will give you a loop that extracts the tokens one at a time from s.

If you need code to capture lines of input, just look at FreeDOS Edlin, available from either ibiblio or alt.sources. That code reads input a character at a time, taking advantage of stdio's file buffering mechanism, and starts a new line when it encounters '\n' in the text. Another way of doing it (but you have to be careful about buffering) is to use fgets() to read a long string and then testing the last character read for the newline. WARNING: Do not use gets(), for it is the tool of the Devil and can lead to buffer overruns, Satan's minions streaming out of your nose at very high speeds, and other delightful undefined behavior.

Here's what Wikipedia in its C book section says about strtok:

The strtok function

## Re: strtok ( ) help

```
char *strtok(char *restrict s1, const char *restrict delimiters);
```

A sequence of calls to `strtok()` breaks the string pointed to by `s1` into a sequence of tokens, each of which is delimited by a byte from the string pointed to by `delimiters`. The first call in the sequence has `s1` as its first argument, and is followed by calls with a null pointer as their first argument. The separator string pointed to by `delimiters` may be different from call to call.

The first call in the sequence searches the string pointed to by `s1` for the first byte that is not contained in the current separator string pointed to by `delimiters`. If no such byte is found, then there are no tokens in the string pointed to by `s1` and `strtok()` shall return a null pointer. If such a byte is found, it is the start of the first token.

The `strtok()` function then searches from there for a byte that is contained in the current separator string. If no such byte is found, the current token extends to the end of the string pointed to by `s1`, and subsequent searches for a token shall return a null pointer. If such a byte is found, it is overwritten by a null byte, which terminates the current token. The `strtok()` function saves a pointer to the following byte, from which the next search for a token shall start.

Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above.

The `strtok()` function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

Because the `strtok()` function must save state between calls, and you could not have two tokenizers going at the same time, the Single Unix Standard defined a similar function, `strtok_r()`, that does not need to save state. Its prototype is this:

```
char *strtok_r(char *s, const char *delimiters, char **lasts);
```

The `strtok_r()` function considers the null-terminated string `s` as a sequence of zero or more text tokens separated by spans of one or more characters from the separator string `delimiters`. The argument `lasts` points to a user-provided pointer which points to stored information necessary for `strtok_r()` to continue scanning the same string.

In the first call to `strtok_r()`, `s` points to a null-terminated string, `delimiters` to a null-terminated string of separator characters, and the value pointed to by `lasts` is ignored. The `strtok_r()` function shall return a pointer to the first character of the first token, write a null character into `s` immediately following the returned token, and update the pointer to which `lasts` points.

In subsequent calls, `s` is a null pointer and `lasts` shall be unchanged

Re: strtok ( ) help

## Re: strtok ( ) help

from the previous call so that subsequent calls shall move through the string s, returning successive tokens until no tokens remain. The separator string delimiters may be different from call to call. When no token remains in s, a NULL pointer shall be returned.

The following public-domain code for strtok and strtok\_r codes the former as a special case of the latter:

```
#include <string.h>
/* strtok_r */
char *(strtok_r)(char *s, const char *delimiters, char **lasts)
{
    char *sbegin, *send;
    sbegin = s ? s : *lasts;
    sbegin += strspn(sbegin, delimiters);
    if (*sbegin == '\0') {
        *lasts = "";
        return NULL;
    }
    send = strpbrk(sbegin, delimiters);
    if (*send != '\0')
        *send++ = '\0';
    *lasts = send;
    return sbegin;
}
/* strtok */
char *(strtok)(char *restrict s1, const char *restrict delimiters)
{
    static char *ssave = "";
    return strtok_r(s1, delimiters, &ssave);
}
```

HTH, Gregory Pietsch

.

---

- *Follow-Ups:*

- ◆ [Re: strtok \( \) help](#)  
◇ From: pete

- *References:*

- ◆ [strtok \( \) help](#)  
◇ From: ern

- Prev by Date: [Re: Does structure order matter?](#)
- Next by Date: [Re: How to determine the way data is stored in memory?](#)
- Previous by thread: [Re: strtok \( \) help](#)
- Next by thread: [Re: strtok \( \) help](#)

Re: strtok ( ) help

- Index(es):
  - ◆ *Date*
  - ◆ *Thread*