Copying and Concatenating C Strings with the str5 Functions

Eric Sanchis
University of Toulouse Capitole
France

ABSTRACT

The copy and the concatenation of strings constitute a recurring subject of polemics within the C programmers community. They generally relate to the respective advantages and disadvantages of the three principal couples of functions which are **strcpy()**/**strcat()**, **strncpy()**/**strncat()** and **strlcpy()**/**strlcat()**.

This article describes two new functions **str5cpy()** and **str5cat()** which were designed to replace those functions while bringing clearness, coherence, safety and facility of use which the preceding functions lack more or less.

The central point of the **str5cpy()** and **str5cat()** functions is their articulation around 5 parameters (and not 2 or 3) which will enable them to deal with the various checks, sources of many errors when they are not or badly done by the programmer, and which provide a truly significant return value indicating the action actually carried out.

General Terms

C Programming, secure coding.

Keywords

C strings, str5cpy, str5cat, strncpy, strncat, strlcpy, strlcat, buffer overflow.

1. INTRODUCTION

Few functions of the *C standard library* cumulate as many failures (trapped names, careless design, inconsistent behaviour, ambiguity, failing safety) as the functions dedicated (or supposedly dedicated) to the copy or concatenation of character strings.

For more than a decade, two points of view clash with each other: those who ask for the inclusion of two new functions **strlcpy()/strlcat()** (called *strl* functions) [1] into the (*GNU*) *C Standard Library* and those who are opposed to this addition, leading implicitly to a *status quo*.

When the remarks of both parties are followed, we must forcibly be struck with the personal animosities, big egos, mental block, answers which are partial, off topic or simply surrealist [2], [3].

Compared to this situation, the point of view which will be defended is that the two parties are wrong. The defenders of the *status quo* have forgotten that (1) the *Standard Library* is not solely used by gurus but also by students, teachers and simple programmers who need reliable functions with interfaces as clear and precise as those of system calls, (2) the use of arbitrary-sized strings is only a programming style and no philosophy of programming must be elevated to an ideology within the framework of such a general library. The other camp should finally take into account the fact that *strl* functions have technical flaws which make their quality insufficient to their integration. To conclude on this point, the *statu quo* must be broken: a reliable alternative to **strcpy()**/**strcat()** (called thereafter *str* functions) and **strncpy()**/**strncat()** (called *strn* functions) must be provided to programmers wishing to use fixed-sized strings.

The paper is broken down as follows. The first section will clarify the design flaw common to the *strn* and *strl* functions. The next section will detail the problems which result from it. Lastly, the last two sections will present the characteristics and qualities of the **str5cpy()/str5cat()** functions (called *str5* functions).

2. SOURCE OF THE PROBLEM: THE NUMBER OF PARAMETERS

As opposed to what one could think, the original *str* functions) of the *C standard library* only suffer from one but crippling defect: they do no checking so they are not safe. Creating safe code using these two functions quickly becomes difficult because the taking into account of the various possible cases is left to the programmer and these checks have to be done each time these two functions are used.

Presented sometimes as functions handling fixed length fields inside structures, *strn* functions of the *C standard library* are in fact interpreted and used as string manipulation functions safer than the *str* functions.

Indeed, *C* and *POSIX standards* reserve the names of functions starting with the *str* prefix to string manipulation functions. Compared to this naming convention, it would be contradictory to call *strn* functions which operate on structure fields: these names would then constitute a real trap for the user of these functions.

In terms of safety, the point of view which will be expressed here is that they create more problems than they solve because *strn* functions give a false sense of security.

The *strl* functions are an improvement compared to the *str* and *strn* functions but appropriately judged insufficient to be included into the *C standard library*.

A detailed analysis of the characteristics of *strn* and *strl* functions and their using modes made it possible to detect the central problem from which these various functions suffer: the definitely insufficient number of parameters which are provided to them compared to the services they should provide.

Prototypes of these functions are the following:

char * **strncpy(**char * *dst***,** const char * *src***,** size_t *n***);**

char * **strncat(**char * *dst***,** const char * *src***,** size_t *n***);**

size_t **strlcpy(**char * *dst***,** const char * *src***,** size_t *n***);**

size_t **strlcat(**char * *dst***,** const char * *src***,** size_t *n***);**

Indeed, a safe use of these functions requires that a certain number of checks be carried out by the programmer either before or after their execution. The three parameters passed to the function do not make it possible to delegate these checks to it. However, checking is often tedious and sometimes delicate to carry out, which leads a sloppy programmer to ignore them or to be mistaken.

Although the documentation of **strncpy()** and **strncat()** functions is perfectly clear as regards the meaning of third parameter *n* (number of characters of the *src* string to be copied or concatenated), it is sometimes used to specify the size of the destination buffer. According to the context, it results that the same parameter can indicate two completely different information: a number of characters or a size. This ambiguity about the meaning of *n* obviously brings about confusion and is the source of many errors.

Sometimes *n* means all the characters of *src* (for example, a full copy of the source string into the destination buffer), sometimes *n* means a copy of the first *n* characters of *src*, sometimes *n* means the size of the destination buffer pointed to by *dst*.

In the first case, the programmer has to count the terminating **nul** byte (**'\ 0'**) of *src*:

```
strncpy(dst,src, strlen(src)+1) ;
```

In the second case, the programmer has to explicitly add a **nul** byte at the end of *dst*:

```
strncpy(dst,src,n) ;
dst[n]='\0' ;
```

In the third case, the programmer copies the string *src* without calculating the length of *src*, but adds extra **nul** bytes if the destination buffer size is greater than `strlen(src)+1`:

```
strncpy(dst,src,sizeof(dst)) ;
```

In these three cases, we have supposed that *dst* was large enough. When *dst* is too small, the situation becomes more complicated.

Like *strn* functions, *strl* functions also use three parameters with the same ambiguities for *n* while introducing more checks.

Throughout this paper the weaknesses of the *strn*/*strl* functions will be highlighted then the solutions suggested by the *str5* functions explained [4].

3. STRN/STRL FUNCTIONS WEAKNESSES

As a reminder, in C language, a string is recorded in an array of characters. A “well-formed” string is an array of characters containing a **nul** byte. *A contrario*, a “bad-formed” string is an array of characters which does not contain a **nul** byte but which is interpreted by a function manipulating strings (e.g. **strlen()**, **strcpy()**, **strncat()**) or by other code as if it “were well-formed”. In the best case, the consequence of this is that characters external to the array may be wrongly considered as belonging to the string. But in certain circumstances, the calculation of the length of a bad-formed string or the search of the **nul** byte in this type of string can lead to a crash.

3.1 *Strn* functions

The *strn* functions suffer from many design problems:

(1) at the prototype level

¤ the number of parameters is insufficient, which leads to a dangerous ambiguity in the interpretation of the third parameter *n*

¤ the return value does not carry any useful information to the programmer

(2) at the functioning level

¤ elementary checks are not done: if one of the *dst* or *src* pointer is the **NULL** pointer, a crash occurs

¤ **strncpy()** and **strncat()** have opposite behaviors:

+ the first may never write a **nul** byte in the destination buffer, the second will always write a **nul** byte (even if it is outside the destination buffer)

+ the first writes as many **nul** bytes than necessary to copy *n* characters, the second stops character adding after writing the terminating **nul** byte of the string *src*

¤ the programmer must sometimes manage the terminating **nul** byte, sometimes he need not do so

¤ few checks being done, the verification of the result is left to the programmer; the ambiguity carried by the third parameter *n* added to the use of more or less complex checking arithmetic expressions produce a particularly error prone cocktail

¤ if the value of *n* is huge compared to the actual length of the string pointed to by *src*, **strncpy()** will unnecessarily copy a great number of **nul** characters into the space pointed to by *dst*, which can lead to a performance problem.

3.2 *Strl* functions

The *strl* functions suffer from several design problems too:

(1) at the prototype level

¤ the third parameter *n* also carries ambiguity

¤ the return value is only useful to detect if there were a truncation or to know the length of the created string. However this value can generate confusion because **strlcpy()** returns the length of *src* while **strlcat()** returns the initial length of *dst* plus the length of *src*

(2) at the functioning level

¤ the **NULL** value for one of the *dst* or *src* pointers causes a crash

¤ *Strl* functions have different behaviors: the **strlcpy()** function makes it possible to copy only the first *n* bytes of *src*, **strlcat()** does not allow to concatenate only the first *n* bytes of *src*

¤ There is a silent and systematic truncation of too long strings during their copy.

4. CHARACTERISTICS OF THE STR5 FUNCTIONS

Names of the **str5cpy()**/**str5cat()** functions reflect that they need 5 parameters:

```
int str5cpy( char *dst,
             size_t dstsize,
             const char * src,
             size_t nb,
             size_t mode ) ;
int str5cat( char * dst,
             size_t dstsize,
              const char * src,
              size_t nb,
              size_t mode ) ;
```

The two first parameters characterize the destination buffer:

`dst:` destination buffer

`dstsize:` size of the destination buffer

The two following parameters are related to the source string:

`src:` source string

`nb:` number of bytes of the source string to be copied/concatenated

The last parameter specifies if truncation is allowed:

`mode:` truncation allowed (`TRUNC`) or not allowed (`NOTRUNC`).

These functions are designed to force programmers to specify what they know (e.g. *dstsize*) and what they want (e.g. the number of characters of *src* to be copied, no truncation). They will behave the same manner whether the **nul** character is counted or not in *nb* when the string *src* must be entirely copied/concatenated: these functions make the implementation strings transparent.

Finally, they also integrate checking that programmers should explicitly do.

They return:

- a *non negative integer* after success:

`OKNOTRUNC`, no truncation was done during the copy/concatenation

`OKTRUNC`, an allowed truncation was done during the copy/concatenation

- and a *negative integer* after error:

`EDSTPAR`, a parameter related to the destination buffer is considered as incorrect

`ESRCPAR`, a parameter related to the source string is considered as incorrect

`EMODPAR`, `mode` is invalid

`ETRUNC`, `dstzise` is too small and truncation is not allowed.

The behavior of these functions is undefined if:

- *dst* and *src* overlap

- *dstsize* is different to **0** and does not correspond to the actual size of the destination buffer

- *src* is a bad-formed string.

¤ ***str5cpy***

The **str5cpy()** function copies up to the first *nb* characters from the source string pointed to by *src* to the destination buffer pointed to by *dst* and adds a terminating **nul** byte.

If *srclen* is the length of the string pointed to by *src* and *srclen* is less than *nb*, **str5cpy()** only writes the *srclen* characters of *src* and an additional **nul** byte to *dst*. Consequently, only `min(nb,srclen)` characters are considered, plus the terminating **nul** byte.

The copy is actually made if the size *dstsize* of the destination buffer is large enough (the returned value is *OKNOTRUNC*) or if the *TRUNC* mode is chosen (the returned value is *OKTRUNC*). The size *dstsize* of the destination buffer *dst* is large enough if

```
dstsize ≥ min(srclen,nb)+1
```

When *dstsize* is too small and the *TRUNC* mode is chosen, only `dstsize-1` characters are copied and the string is terminated with a **nul** byte.

The destination buffer remains unchanged if one of these conditions is realized:

- *dst* is a **NULL** pointer or *dstsize* is equal to **0**. The returned value is *EDSTPAR*

- *src* is a **NULL** pointer or *nb* (or *srclen*) is equal to **0**. The returned value is *ESRCPAR*

- *dstsize* is too small and the *NOTRUNC* mode is chosen. The returned value is *ETRUNC*

- the *mode* parameter is incorrect. The returned value is *EMODPAR*.

To sum up, **str5cpy()** stores a well-formed string into the buffer pointed to by *dst* and the return value is *OKxxx* or leaves this buffer unchanged and the return value is *Exxx*.

Examples:

String copy becomes easy. For example, to prevent truncation:

```
   char dst[DSTSIZE] ;
   ret= str5cpy(dst, sizeof(dst), src,
sizeof(dst), NOTRUNC) ;
   switch( ret )
   {
     case ETRUNC: /* appropriate
treatment */
        . . .
   }
```

Programmers do not need to calculate the length of the *src* string: they only need to indicate the size of the destination buffer *dst* as fourth parameter. No extra **nul** byte will be added.

Another way is to pass the length of *src* to the **str5cpy()** function, but it is counterproductive because **str5cpy()** calculates this length internally so this value will be calculated twice:

```
   ret = str5cpy(dst, sizeof(dst), src,
strlen(src), NOTRUNC) ;
```

If the possibility to lose data is not problematic:

```
   if ( (ret = str5cpy(dst, sizeof(dst),
src, sizeof(dst), TRUNC)) < 0 )
        { /* error */ }
```

To copy the first *n* bytes of *src*:

```
   ret = str5cpy(dst, sizeof(dst), src,
n, NOTRUNC) ; /* or TRUNC */
```

¤ ***str5cat***

The **str5cat()** function appends at most the first *nb* bytes from *src* string to the *dst* string, overwriting the terminating **nul** byte of *dst*, and then adds a terminating **nul** byte.

Similarly to **str5cpy()**, **str5cat()** appends `min(nb,srclen)` characters of *src* and an additional **nul** byte to *dst*.

The concatenation is actually made if the remaining space in the destination buffer *dst* is large enough (the returned value is *OKNOTRUNC*) or if the *TRUNC* mode is chosen (the returned value is *OKTRUNC*).

If *dstlen* is the length of a well-formed string initially memorized into the destination buffer, it is possible to append up to `dstsize-dstlen` characters (**nul** byte included) to *dst*.

Let `remain = dstsize-dstlen`. The *dst* destination buffer is large enough if

```
        remain > min(srclen,nb)
```

When there is not enough space and the *TRUNC* mode is chosen, only `remain-1` characters are appended and the string is terminated with a **nul** byte.

The destination buffer remains unchanged if one of these conditions is realized:

- *dst* is a **NULL** pointer or is a bad-formed string or *dstsize* is equal to **0**. The returned value is *EDSTPAR*

- *src* is a **NULL** pointer or *nb* (or *srclen*) is equal to **0**. The returned value is *ESRCPAR*

- the remaining space is too small and the *NOTRUNC* mode is chosen. The returned value is *ETRUNC*

- the *mode* parameter is incorrect. The returned value is *EMODPAR*

- there is no space left in the destination buffer (the string *dst* fully occupies the buffer) and the *TRUNC* mode is chosen. The returned value is *OKTRUNC*.

<u>*Examples*</u>:

Concatenating string is as simple as copying:

```
   char dst[DSTSIZE] ;

   if ( str5cat(dst, sizeof(dst), src,
sizeof(dst), TRUNC) == OKTRUNC )

   { /* truncation happened */ }
```

To concatenate the first *n* bytes of *src*:

```
   ret = str5cat(dst, sizeof(dst), src,
n, NOTRUNC) ; /* or TRUNC */
```

Copying and concatenating strings are often used to build pathnames piece by piece:

```
   char path[PATHSIZE] ;

   char * dir = "myHOME", * file =
"myFile" ;

   if ( str5cpy( path, sizeof(path), dir,
sizeof(path),NOTRUNC) == ETRUNC )

       goto toolong ; /* truncation
happened */

   if ( str5cat(path, sizeof(path), "/",
sizeof(path), NOTRUNC) == ETRUNC )

       goto toolong ; /* truncation
happened */

   if ( str5cat(path, sizeof(path), file,
sizeof(path), NOTRUNC) == ETRUNC )

       goto toolong ; /* truncation
happened */
```

5. SYNTHETIC SUMMARY

In short, *str5* functions have the following properties:

¤ the sufficient number of parameters allows the programmer not to carry out the tedious and delicate tasks of checking

¤ the return value informs the programmer of the quality of the obtained result

¤ if one of the *dst* or *src* pointers is the **NULL** pointer, a suitable error code is returned

¤ the programmer does not have to manage the terminating **nul** byte

¤ if the value of *nb* is bigger than the actual length of the string pointed to by *src*, no extra **nul** character will be copied into the buffer pointed to by *dst*

¤ as far as possible they take into account the presence of bad-formed strings

¤ truncations are directly controllable by the programmer.

Two more general properties enable to assess the quality of an API (*Application Programming Interface*): *compactness* and *transparency*.

¤ *Compactness*:

Compactness characterizes a tool which does not need a memorization of numerous details for the user to manipulate it appropriately. Compact software is pleasant to use because it makes programmers more productive. *Str5* functions are *compact*, *strn* and *strl* functions are not.

There is a link between *compactness* and *consistency*. *Strn and strl* functions are *inconsistent* because their copy function does not work in the same manner as the concatenation one. As a result, many details must be taken into account before using them.

¤ *Transparency*:

Lastly, the *str5* functions are *transparent* because they do not take any initiative on behalf of the programmer:

- if a parameter value is incorrect (except *dstsize*), the content of the destination buffer *dst* will not be modified. That guarantees that there will be no loss of data contained in *dst* after an incorrect call to **str5cpy()** or **str5cat()**

- truncation is only done if it is specified by the programmer.

6. CONCLUSION

C string handling is particularly tricky, prone to numerous kinds of programming errors which can lead to more or less serious vulnerabilities. Being more *transparent*, *compact* and *consistent* than *strn* and *strl* functions, **str5cpy()** and **str5cat()** are easier to understand and easier to use. As they do more checks, they are *safer*.

7. REFERENCES

[1] Todd C. Miller, Theo de Raadt, strlcpy and strlcat – Consistent, Safe String Copy and Concatenation, USENIX 1999 https://www.usenix.org/legacy/events/usenix99/full_papers/millert/millert.pdf

[2] libc-alpha archive, August 2000, PATCH: safe string copy and concetation, https://www.sourceware.org/ml/libc-alpha/2000-08/threads.html

[3] libc-alpha archive, September 2014, [PATCH] Implement strlcpy [BZ #178] https://sourceware.org/ml/libc-alpha/2014-09/threads.html

[4] Eric Sanchis, Copying and concatenating strings with the str5 functions, October 2014 http://aral.iut-rodez.fr/en/sanchis/miscellaneous/str5/str5_v09.1.pdf