

## (NeXT Tip #38) strcmp, strncmp & strlen

Christopher Lane (*lane[at]CAMIS.Stanford.EDU*)  
Wed, 14 Jul 1993 17:15:02 -0700 (PDT)

The standard C/Unix string library comparison function 'strcmp' compares its arguments and returns an integer greater than, equal to, or less than '0' (zero), if the first string is lexicographically greater than, equal to, or less than the second string. It returns three different values but 90% of the time it is used (understandably) as a boolean function, leading to code like:

```
if(!strcmp(string, "demos")) { ... };
```

A 'C' programmer will immediately recognize that the body of the 'if' clause will be executed if the two strings are the same, not if they are different as the uninitiated might expect from the boolean negation operator '!'. This is because '0' (zero) is the equality result and is also logical FALSE in 'C'. One way to fix this stylistic problem is to not use 'strcmp' as a boolean:

```
typedef enum { LT = -1, EQ, GT } Comparison;  
...  
if(strcmp(string, "demos") == EQ) { ... };
```

This fix can make 'strcmp' calls more readable -- however you still need to use '> GT' and '< LT' in your comparisons, not '==' as the only the sign of the 'strcmp' result is significant, not the value. (Also a '>= GT' comparison may be equally hard to decipher and not a stylistic improvement.)

However, since 'strcmp' is used as a boolean routine 90% of the time, let's optimize for that:

```
#define strequal(s1, s2) (strcmp(s1, s2) == EQ)  
...  
if(strequal(string, "demos")) { ... };
```

which should be a little more obvious to a casual reader of the program. (Or, to retain some C/Unix flavor, use a name like 'streq'. :-)

The 'strcasecmp' variant of 'strcmp' works the same but ignores the case of letters when doing a comparison. This routine is underutilized and should be considered when using a 'strcmp' comparison. It can avoid situations like:

```
if(!strncmp(extptr, ".sea", 4) ||  
!strncmp(extptr, ".Sea", 4) ||  
!strncmp(extptr, ".SEA", 4)) { ... };
```

This example also has a potential bug in its use of the 'strncmp' variant of 'strcmp' as the file extension ".sealion" might be an unintended valid match.

The 'strncmp' and 'strncasecmp' routines take a third argument which is the maximum number of characters to check. Here are a couple of cases I can think of where you would use these more specialized routines:

A) You just want to compare string prefixes:

```
if(strncasecmp(string, "Yes", 1) == EQ) /* test if 'Yes', 'YES', etc. */
```

B) You are comparing strings that are NOT null terminated. Most 'C' strings end in a null byte and the various 'strcmp' routines quit when that byte is reached on either string. In some situations, you have character arrays that lack a terminating null character in which case you should use 'strncmp' with the length of the shortest unterminated string argument:

```
char *string, array[ARRAYSIZE];  
....
```

```
if(strncmp(string, array, ARRAYSIZE) != EQ) { ... };
```

Otherwise, you might end up comparing memory beyond the end of the string and/or get a segmentation fault. However, DO NOT use 'strncmp' just because you are able to compute the length of a string:

```
#define PUBDIR "/usr/spool/uucppublic"  
...  
if(strncmp(string, PUBDIR, strlen(PUBDIR)) == EQ) { ... };
```

There is no real advantage to doing this over just using 'strcmp' and it could be a bug if you didn't intend for a match like "/usr/spool/uucppublic/lock".

While we're on the subject of 'strlen', the common 'C' code construct:

```
if(strlen(string) > 0) { ... };
```

walks through all the characters of a string just to determine if it has any characters. If the string is null terminated (as 'strlen' requires) then:

```
if(*string != '\0') { ... };
```

might be a better approach to determine if you have an empty, non-NULL string. Similarly, another not uncommon 'C' code construct:

```
for(i = 0; i < strlen(string); i++) if(!isprint(string[i])) { ... };
```

might be better written to compute 'strlen(string)' outside the loop as it will be evaluated for every character -- unless you have a very smart compiler. This approach is fine if the string is changing inside the loop, otherwise it wastes cycles. Of course, a more 'C-like' approach might be:

```
char *s = string;  
...  
while(*s) if(!isprint(*s++)) { ... };
```

which gets into a whole 'nother style issue altogether.

NeXT also provides their own general, table-driven string ordering function:

```
int NXOrderStrings(const unsigned char *s1,  
const unsigned char *s2,  
BOOL caseSensitive,  
int length,  
NXStringOrderTable *table)
```

```
NXStringOrderTable *NXDefaultStringOrderTable(void)
```

'NXOrderStrings' returns a signed result similar to 'strcmp' but as determined by the structure 'table', not necessarily a simple character comparison. The 'caseSensitive' argument incorporates 'strcasecmp' style comparisons in this same routine and like 'strncmp' the comparison considers at most the first 'length' characters but you can pass -1 for length if both strings are null-terminated. If 'table' is NULL, a default ordering table is used.

'NXOrderStrings' consults an NXStringOrderTable structure when comparing strings, indexing the characters into one of several arrays. The default ordering table works the same as 'strcmp' but provides additional ordering information for foreign and ligature characters (breaking ligatures apart for ordering). Thus, the two characters 'fi' and the equivalent ligature glyph are given equal rank. NeXT provides a default order table, which you can access by calling 'NXDefaultStringOrderTable' and you can create your own table by modifying a copy of this default table.

Thus when comparing strings that are visible to the user, NeXT recommends you use 'NXOrderStrings' as a replacement for 'strcmp' and its variants. What you choose to do depends on how much you want to trade off portability for

'localization' and the ability to function as a conforming NeXT application.

- Christopher