C String Reference Guide

Warren Toomey

This is only a reference guide for dealing with strings in the C language; it's not a tutorial. Search the web for C tutorials, or read through Steve Summit's C tutorial at `http://www.eskimo.com/~scs/cclass/notes/top.html`. I will provide some tips and warnings. Much of the reference material comes from the FreeBSD on-line manuals.

Warning: Throughout the guide, NUL means an 8-bit character whose value is binary 0000 0000, hexadecimal 0x00, and can be written as the character '\0'. Don't confuse this with NULL, which is the value a pointer takes when it is not pointing anywhere. One is the character used to terminate strings, the other indicates that a pointer has no pointee.

Working with Strings In-Place

In many cases you can work on strings in place:

- searching for characters
- replacing characters with other characters
- splitting a string into two or more strings
- truncating a string
- sometimes enlarging a string (if you have set aside extra space in a buffer)

It is important to remember that in C a string is represented as a "char" pointer, which points at the first character in the string, and that the string ends with the all-bits-zero character '\0'. In most cases, you can either walk a pointer along a string, or you can treat the pointer as a char array.

For example, to find the first 'F' in a given string, you might do:

```
char *string= "we want to Find the first F";
char *cptr;

for (cptr=string; *cptr && (*cptr!='F'); cptr++)
   /* Empty loop body */  ;
```

which translates as:

1. Start with cptr pointing at the first character in the string.
2. While cptr is non-zero, and while it isn't pointing to an 'F',
3. increment cptr

Once the loop exits, cptr either points at the first 'F', or it points as the '\0' character if no 'F' was found.

Replacing letters with other letters is just as simple. Here is the code to replace 'F's with 'G's:

```
for (cptr=string; *cptr; cptr++)
  if (*cptr=='F') *cptr='G';
```

Splitting a string into 2 or more smaller strings is done by replacing characters in the string with '\0' characters. Of course, the original characters will get lost. For example, if you have an input file which consists of two columns that are tab separated, this code will extract the two columns:

```
char *input_line;    /* whose value is read in somehow */
char *col1, *col2;

/* Walk the string to find the tab */
for (col1=col2=input_line; *col2 && (*col2!='\t'); col2++)
  ;

/* Replace tab with '\0' and move to next char */

if (*col2=='\t') {
  *col2= '\0'; col2++;
}

/* Now col1 points at the 1st column, col2 at the 2nd column */
```

You can use a char pointer (i.e. a string) as an array too. For example, after an fgets() you will have a string which ends in '\n'. To remove the newline, simply replace it with a '\0'. The strlen() (string length) function is useful here. For example, the string "Hello\n" consists of 5 printable characters (with array index values 0 to 4), a newline at array index 5, and a '\0' character at array index 6. The length of the string as reported by strlen() would be 6 (5 printable+1 newline). Thus, to remove the newline at position 5, we can do:

```
buf[strlen(buf)-1]= '\0';
```

If you are 100% sure that there is extra space past the end of a string (for example, it has been copied into a fixed-size buffer), you can enlarge the string. For example, take the original "Hello\n" string which is sitting in a 100-character buffer. If we wished to convert the linefeed into CR-LF, we might do:

```
int len;
len= strlen(buf);
buf[len-1]='\r'; buf[len]='\n'; buf[len+1]='\0';
```

Why didn't we use strlen() inside the [] this time. Two reasons: we would be recalculating the string's length 3 times, and the string's length would be changing, and so the position we would be overwriting would be wrong.

The rest of this guide lists the common C string functions with a brief description of their use.

Cloning A String

Sometimes you want to copy a string, but either you don't have an existing string buffer, or you can't predict the size of the string, so it may not fit into the existing string buffer. In other situations, a C function might return a pointer to a string which was one of its local variables (bad, bad, bad!) and you have to clone it quickly before it gets clobbered by a new function call. In these situations, use strdup() to clone the string. strdup() essentially malloc()s a string buffer which is the size of the string, and memcpy()s the string into the new buffer.

```
#include <string.h>

char * strdup(const char *str);
```

The strdup() function allocates sufficient memory for a copy of the string `str`, does the copy, and returns a pointer to it. The pointer may subsequently be used as an argument to the function free(3). If insufficient memory is available, NULL is returned.

Copying, Concatenation, Clearing

These functions assume an existing buffer exists to copy into.

```
#include <string.h>

char * strncpy(char *dst, const char *src, size_t len);
char * strncat(char *s, const char *append, size_t count);
void * memset(void *b, int c, size_t len);

size_t strlcpy(char *dst, const char *src, size_t size);
size_t strlcat(char *dst, const char *src, size_t size);
```

The strncpy() function copies not more than `len` characters from `src` into `dst`, appending '\0' characters if `src` is less than `len` characters long, and not terminating `dst` otherwise. The strncpy() function returns `dst`.

The strncat() functions appends a copy of the null-terminated string `append` to the end of the null-terminated string `s`, then add a terminating '\0'. The string `s` must have sufficient space to hold the result. The strncat() function appends not more than `count` characters from `append`, and then adds a terminating '\0'. The strncat() functions returns the pointer `s`.

The memset() function writes `len` bytes of value `c` (converted to an unsigned char) to the string `b`. The memset() function returns its first argument.

The strlcpy() and strlcat() functions copy and concatenate strings respectively. They are designed to be safer, more consistent, and less error prone replacements for strncpy(3) and strncat(3). Unlike those functions, strlcpy() and strlcat() take the full size of the buffer (not just the length) and guarantee to NUL-terminate (i.e '\0' terminate) the result (as long as `size` is larger than 0 or, in the case of strlcat(), as long as there is at least one byte free in `dst`). Note that you should include a byte for the NUL in `size`. Also note that strlcpy() and strlcat() only operate on true "C" strings. This means that for strlcpy() `src` must be NUL-terminated and for strlcat() both `src` and `dst` must be NUL-terminated.

The strlcpy() function copies up to `size-1` characters from the NUL-terminated string `src` to `dst`, NUL-terminating the result.

The strlcat() function appends the NUL-terminated string `src` to the end of `dst`. It will append at most `size-strlen(dst)-1` bytes, NUL-terminating the result.

The strlcpy() and strlcat() functions return the total length of the string they tried to create. For strlcpy() that means the length of `src`. For strlcat() that means the initial length of `dst` plus the length of `src`. While this may seem somewhat confusing it was done to make truncation detection simple.

Note however, that if strlcat() traverses `size` characters without finding a NUL, the length of the string is considered to be `size` and the destination string will not be NUL-terminated (since there was no space for the NUL). This keeps strlcat() from running off the end of a string. In

practice this should not happen (as it means that either size is incorrect or that dst is not a proper "C" string). The check exists to prevent potential security problems in incorrect code.

Comments: Don't ever, ever use strcat() nor strcpy(), as these functions may overflow the destination string and cause buffer overflows. Use strlcpy() and strlcat() if possible, but if your system doesn't support these functions, then fall back to using strncpy() and strncat().

String Comparisons

```
#include <string.h>

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

int strcasecmp(const char *s1, const char *s2);
int strncasecmp(const char *s1, const char *s2, size_t len);
```

The strcmp() and strncmp() functions lexicographically compare the NUL-terminated strings `s1` and `s2`. The strncmp() function compares not more than `len` characters. Because strncmp() is designed for comparing strings rather than binary data, characters that appear after a '\0' character are not compared.

The strcmp() and strncmp() return an integer greater than, equal to, or less than 0, according as the string `s1` is greater than, equal to, or less than the string `s2`. The comparison is done using unsigned characters, so that '0x80' is greater than '\0'.

Comment: These functions don't return true or false. In fact, they return 0 when there *is* a string match. Therefore you will find this common C style to see if two strings are the same:

```
if (!strcmp(string1, string2)) {
  printf("The 2 strings are equal\n";
}
```

The strcasecmp() and strncasecmp() return an integer greater than, equal to, or less than 0, according as `s1` is lexicographically greater than, equal to, or less than `s2` after translation of each corresponding character to lower-case. The strings themselves are not modified. The comparison is done using unsigned characters, so that '0x80' is greater than '\0'.

Finding Characters & Substrings

```
#include <string.h>

char * strchr(const char *s, int c);
char * strrchr(const char *s, int c);
```

The strchr() function locates the first occurrence of `c` (converted to a char) in the string pointed to by `s`. The terminating NUL character is considered part of the string; therefore if `c` is '\0', the functions locate the terminating '\0'. The strrchr() function is identical to strchr() except it locates the last occurrence of `c`.

The functions strchr() and strrchr() return a pointer to the located character, or NULL if the character does not appear in the string.

```
#include <string.h>

char * strpbrk(const char *s, const char *charset);
```

The strpbrk() function locates in the NUL-terminated string `s` the first occurrence of any character in the string `charset` and returns a pointer to this character. If no characters from charset occur anywhere in `s`, strpbrk() returns NULL.

```
#include <string.h>

char * strstr(const char *big, const char *little);
char * strcasestr(const char *big, const char *little);
```

The strstr() function locates the first occurrence of the NUL-terminated string `little` in the NUL-terminated string `big`. The strcasestr() function is similar to strstr(), but ignores the case of both strings.

If `little` is an empty string, `big` is returned; if `little` occurs nowhere in `big`, NULL is returned; otherwise a pointer to the first character of the first occurrence of `little` is returned.

Finding Substrings Separated by Certain Characters

```
#include <string.h>

char * strsep(char **stringp, const char *delim);
```

The strsep() function locates, in the string referenced by `*stringp`, the first occurrence of any character in the string `delim` (or the terminating '\0' character) and replaces it with a '\0'. The location of the next character after the delimiter character (or NULL, if the end of the string was reached) is stored in `*stringp`. The original value of `*stringp` is returned.

An "empty" field (i.e., a character in the string `delim` occurs as the first character of `*stringp`) can be detected by comparing the location referenced by the returned pointer to '\0'.

If `*stringp` is initially NULL, strsep() returns NULL.

The following uses strsep() to parse a string, which contains words delimited by white space (i.e. spaces or tabs, and save pointers to the words in an array:

```
char *inputstring;   /* Input string, e.g "hello there\t\tDave!" */
char *argv[10];      /* String pointers for up to 10 words */
char **ap;           /* Pointer used to store stuff into the array */

for (ap = argv; (*ap = strsep(&inputstring, " \t")) != NULL;)
  if (**ap != '\0')
    if (++ap >= &argv[10]) break;
```

which reads as follows:

1. Point ap (a character pointer pointer) at the first element of argv.
2. Loop while strsep() on the input string has found a word ending in space, tab or '\0'. This sets *ap to point at the string, which has the effect of saving the string pointer into argv[].
3. Inside the loop, if the character at **ap isn't a NUL, then increment ap.

4. If ap now points to the address of argv[10] (which doesn't exist, the array goes from 0 to 9), then break out of the loop.

Given the example input string, this should result in argv[0] pointing to "hello", argv[1] pointing to "there", argv[2] pointing to "Dave!" and argv[3] set to NULL.

Comment: There is an older function called strtok() which is in the ANSI C standard. Avoid it if possible, as strtok() cannot detect fields delimited by two adjacent delimiter characters like the "\t\t" in the example above.

Find Regular Expressions

Try to avoid doing this unless you really like C programming. If you really do want to do this, then read the manuals for regcomp(), regexec(), regerror() and regfree().

Variables to Strings

The best solution here is to use snprintf(), which acts like printf() but the result is stored into a string buffer. The usual printf() conversions like %d (decimal), %c (character), %x (hexadecimal) etc. exist. Read the snprintf() manual for more details. Here is the synopsis.

```
#include <stdio.h>

int snprintf(char *str, size_t size, const char *format, ...);
```

The printf() family of functions produces output according to a format as described in the on-line manual page. Snprintf() will write at most `size`-1 of the characters printed into the output string `str` (the size'th character then gets the terminating '\0'); if the return value is greater than or equal to the `size` argument, the string was too short and some of the printed characters were discarded.

Snprintf() returns the number of characters that would have been printed if the `size` were unlimited (again, not including the final '\0'), or a negative value if an output error occurs.

Strings to Variables

```
#include <stdlib.h>
int atoi(const char *nptr);
long atol(const char *nptr);
double atof(const char *nptr);
```

The atoi() function converts the initial portion of the string pointed to by `nptr` to integer representation. The atol() function converts the initial portion of the string pointed to by `nptr` to long integer representation. The atof() function converts the initial portion of the string pointed to by `nptr` to double representation.

For conversion of a string into an unsigned integer, read the strtoul() manual.

I would try to avoid scanf() and sscanf() to do string conversions. They really only work when the input string is always formatted exactly the right way, and this is often not true with user input. If you have to deal with arbitrary user input, then I would recommend using lex and yacc to parse the input and produce errors. But that's another reference guide in itself!