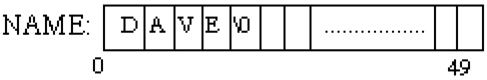
Subsections

String Handling: `<string.h>`

Recall from our discussion of arrays (Chapter 6) that strings are defined as an array of characters or a pointer to a portion of memory containing ASCII characters. A `string` in C is a sequence of zero or more characters followed by a `NULL` (\0)character:



It is important to preserve the `NULL` terminating character as it is how C defines and manages variable length strings. **All** the C standard library functions require this for successful operation.

In general, apart from some ***length-restricted functions*** (`strncat(), strncmp,()` and `strncpy()`), unless you create strings by hand you should not encounter any such problems, . You should use the many useful string handling functions and not really need to ***get your hands dirty*** dismantling and assembling strings.

Basic String Handling Functions

All the string handling functions are prototyped in:

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

The common functions are described below:

`char *stpcpy (const char *dest,const char *src)` -- Copy one string into another.
`int strcmp(const char *string1,const char *string2)` - Compare string1 and string2 to determine alphabetic order.
`char *strcpy(const char *string1,const char *string2)` -- Copy string2 to stringl.
`char *strerror(int errnum)` -- Get error message corresponding to specified error number.
`int strlen(const char *string)` -- Determine the length of a string.
`char *strncat(const char *string1, char *string2, size_t n)` -- Append n characters from string2 to stringl.
`int strncmp(const char *string1, char *string2, size_t n)` -- Compare first n characters of two strings.
`char *strncpy(const char *string1,const char *string2, size_t n)` -- Copy first n characters of string2 to stringl .
`int strcasecmp(const char *s1, const char *s2)` -- case insensitive version of `strcmp()`.

`int strncasecmp(const char *s1, const char *s2, int n)` -- case insensitive version of `strncmp()`.

The use of most of the functions is straightforward, for example:

```
char *str1 = "HELLO";
char *str2;
int length;

length = strlen("HELLO"); /* length = 5 */
(void) strcpy(str2,str1);
```

Note that both `strcat()` and `strcopy()` both return a copy of their first argument which is the destination array. Note the order of the arguments is ***destination array*** followed by ***source array*** which is sometimes easy to get the wrong around when programming.

The `strcmp()` function ***lexically*** compares the two input strings and returns:

Less than zero
    -- if `string1` is lexically less than `string2`
Zero
    -- if `string1` and `string2` are lexically equal
Greater than zero
    -- if `string1` is lexically greater than `string2`

This can also confuse beginners and experience programmers forget this too.

The `strncat(), strncmp,()` and `strncpy()` copy functions are string restricted version of their more general counterparts. They perform a similar task but only up to the first `n` characters. Note the the `NULL` terminated requirement may get violated when using these functions, for example:

```
char *str1 = "HELLO";
char *str2;
int length = 2;


(void) strcpy(str2,str1, length); /* str2 = "HE" */
```

`str2` is NOT NULL TERMINATED!! -- BEWARE

String Searching

The library also provides several string searching functions:

`char *strchr(const char *string, int c)` -- Find first occurrence of character `c` in string.
`char *strrchr(const char *string, int c)` -- Find last occurrence of character `c` in string.
`char *strstr(const char *s1, const char *s2)` -- locates the first occurrence of the string `s2` in string `s1`.
`char *strpbrk(const char *s1, const char *s2)` -- returns a pointer to the first occurrence in string s1 of any character from string `s2`, or a null pointer if no character from `s2` exists in `s1`
`size_t strspn(const char *s1, const char *s2)` -- returns the number of characters at the begining of `s1` that match `s2`.
`size_t strcspn(const char *s1, const char *s2)` -- returns the number of characters at the begining of `s1` that ***do not*** match `s2`.

`char *strtok(char *s1, const char *s2)` -- break the string pointed to by `s1` into a sequence of tokens, each of which is delimited by one or more characters from the string pointed to by `s2`.
`char *strtok_r(char *s1, const char *s2, char **lasts)` -- has the same functionality as strtok() except that a pointer to a string placeholder lasts must be supplied by the caller.

`strchr()` and `strrchr()` are the simplest to use, for example:

```
char *str1 = "Hello";
char *ans;

ans = strchr(str1,'l');
```

After this execution, `ans` points to the location `str1 + 2`

`strpbrk()` is a more general function that searches for the first occurrence of any of a group of characters, for example:

```
char *str1 = "Hello";
char *ans;

ans = strpbrk(str1,'aeiou');
```

Here, `ans` points to the location `str1 + 1`, the location of the first `e`.

strstr() returns a pointer to the specified search string or a null pointer if the string is not found. If s2 points to a string with zero length (that is, the string ""), the function returns s1. For example,

```
char *str1 = "Hello";
char *ans;

ans = strstr(str1,'lo');
```

will yield `ans` = `str + 3`.

`strtok()` is a little more complicated in operation. If the first argument is not NULL then the function finds the position of any of the second argument characters. However, the position is remembered and any subsequent calls to `strtok()` will start from this position if on these subsequent calls the first argument is `NULL`. For example, If we wish to break up the string `str1` at each space and print each token on a new line we could do:

```
char *str1 = "Hello Big Boy";
char *t1;


for ( t1 = strtok(str1," ");
      t1 != NULL;
      t1 = strtok(NULL, " ") )

printf("%s\n",t1);
```

Here we use the for loop in a non-standard counting fashion:

- The initialisation calls `strtok()` loads the function with the string `str1`
- We terminate when t1 is `NULL`
- We keep assigning tokens of `str1` to `t1` until termination by calling `strtok()` with a `NULL` first argument.

Character conversions and testing: `ctype.h`

We conclude this chapter with a related library `#include <ctype.h>` which contains many useful functions to convert and test ***single*** characters. The common functions are prototypes as follows:

Character testing:

`int isalnum(int c)` -- True if c is alphanumeric.
`int isalpha(int c)` -- True if c is a letter.
`int isascii(int c)` -- True if c is ASCII .
`int iscntrl(int c)` -- True if c is a control character.
`int isdigit(int c)` -- True if c is a decimal digit
`int isgraph(int c)` -- True if c is a graphical character.
`int islower(int c)` -- True if c is a lowercase letter
`int isprint(int c)` -- True if c is a printable character
`int ispunct (int c)` -- True if c is a punctuation character.
`int isspace(int c)` -- True if c is a space character.
`int isupper(int c)` -- True if c is an uppercase letter.
`int isxdigit(int c)` -- True if c is a hexadecimal digit

Character Conversion:

`int toascii(int c)` -- Convert `c` to ASCII .
`tolower(int c)` -- Convert c to lowercase.
`int toupper(int c)` -- Convert c to uppercase.

The use of these functions is straightforward and we do not give examples here.

Memory Operations: `<memory.h>`

Finally we briefly overview some basic memory operations. Although not strictly string functions the functions are prototyped in `#include <string.h>`:

`void *memchr (void *s, int c, size_t n)` -- Search for a character in a buffer .
`int memcmp (void *s1, void *s2, size_t n)` -- Compare two buffers.
`void *memcpy (void *dest, void *src, size_t n)` -- Copy one buffer into another .
`void *memmove (void *dest, void *src, size_t n)` -- Move a number of bytes from one buffer lo another.
`void *memset (void *s, int c, size_t n)` -- Set all bytes of a buffer to a given character.

Their use is fairly straightforward and not dissimilar to comparable string operations (except the exact length (`n`) of the operations must be specified as there is no natural termination here).

Note that in all case to **bytes** of memory are copied. The `sizeof()` function comes in handy again here, for example:

```
char src[SIZE],dest[SIZE];
int  isrc[SIZE],idest[SIZE];
```

```
memcpy(dest,src, SIZE); /* Copy chars (bytes) ok */

memcpy(idest,isrc, SIZE*sizeof(int)); /* Copy arrays of ints */
```

`memmove()` behaves in exactly the same way as `memcpy()` except that the source and destination locations may overlap.

`memcmp()` is similar to `strcmp()` except here ***unsigned bytes*** are compared and returns less than zero if `s1` is less than `s2` ***etc.***

Exercises

Exercise 12584

Write a function similar to strlen that can handle unterminated strings. Hint: you will need to know and pass in the length of the string.

Exercise 12585

Write a function that returns true if an input string is a palindrome of each other. A palindrome is a word that reads the same backwards as it does forwards ***e.g*** ABBA.

Exercise 12586

Suggest a possible implementation of the `strtok()` function:

1.
using other string handling functions.

2.
from first pointer principles

How is the storage of the tokenised string achieved?

Exercise 12587

Write a function that converts all characters of an input string to upper case characters.

Exercise 12591

Write a program that will reverse the contents stored in memory in bytes. That is to say if we have *n* bytes in memory byte *n* becomes byte 0, byte *n*-1 becomes byte 1 ***etc.***

Dave Marshall
1/5/1999