**Strings and Related**

Strings in C are represented by arrays of characters. The end of the string is marked with a special character, the null character, which is simply the character with the value 0. (The null character has no relation except in name to the null pointer. In the ASCII character set, the null character is named NUL.) The null or string-terminating character is represented by another character escape sequence, \0. To define say a buffer big enough for 80 chars, this definition works.  **char buffer[81];**

You can't assign a text string directly, so

**buffer = "a test string";**

Does not work, but a char pointer string initialization is allowed.

**char \* myName = "David";**

**Difference between string and character array**

* [Character](http://www.blurtit.com/q496124.html) array can be string or numbers have the same characters. Character array are variable but string is constant.
* array it is the collection of similer data types ex:int a[10]ten indicates the size of array. [ ] is index of array we can give only integer values to array of a. where as string mean colection of group of charaters in double Quotes( );
* Still not clear-explain with e.g

**String related library functions & programs**

* Strcpy/strncpy
* Strcat/strncat
* Strlen
* Strcmp/strncmp
* Strstr
* Strtok
* Memcpy
* Memmove
* Memset

Note: If using all the above string related library functions, then we have to use the #include <string.h>

**Strcpy/strncpy**

Both of these copy the string pointed to by s2 into the string pointed to by s1, including the trailing null. Strncpy will copy at most n characters, and pad with trailing nulls if s2 is shorter than n characters. If the strings overlap, the behavior is undefined. They return s1.

**Strcpy strncpy**

int main ()

{

char str1[]= "To be or not to be";

char str2[6];

strncpy (str2,str1,5);

str2[5]='\0';

puts (str2);

return 0;

}

Output:To be

|  |
| --- |
|  |

char string1[] = "Hello, world!";

char string2[20];

strcpy(string2, string1);

*How internally works otherway*

mystrcpy(char dest[], char src[])

{

int i = 0;

while(src[i] != '\0')

{

dest[i] = src[i];

i++;

}

dest[i] = '\0';

}

char \*strcpy(char \*dest, const char \*src)

{

char \*tmp = dest;

while ((\*dest++ = \*src++) != '\0')

return tmp;

}

H

**Strcat/strncat**

Both append the string in s2 to s1, overwriting the null at the end of s1. A final null is always written. At most n characters are copied from s2 by strncat, which means that for safety the destination string should have room for its original length (not counting the null) plus n + 1 characters. They return s1.

**strcat**

char string5[20] = "Hello, ";

char string6[] = "world!";

char string7[] = “Mapping”;

printf("%s\n", string5);

strcat(string5, string6);

printf("%s\n", string5);

strncat(string5,string7,4);

printf("%s\n", string5);

Output:Hello,Hello world!,Hello world Mapp

Output: Hello,

Hello, world!

char \*strcat(char \*dest, const char \*src)

{

char \*tmp = dest;

while (\*dest)

dest++;

while ((\*dest++ = \*src++) != '\0')

;

return tmp;

}

**Strlen**

Returns the length of the string s not counting the terminating null. This is a very widely used function.

int mystrlen(char str[])

{

int i;

for(i = 0; str[i] != '\0'; i++)

{}

return i;

}

char string7[] = "abc";

int len = strlen(string7);

printf("%d\n", len);

Output: 3

***Difference between sizeof() operator & strlen:***From the above program sizeof(string7) will result in 4 bytes since it includes ‘\0’ also whereas strlen doesn’t.

**Strcmp/Strncmp**

Compares the two strings. This is one of the most commonly used of the string-handling functions. As for strcmp, strncmp but compares at most n characters.

mystrcmp(char str1[], char str2[])

{

int i = 0;

while(1)

{

if(str1[i] != str2[i])

return str1[i] - str2[i];

if(str1[i] == '\0' || str2[i] == '\0')

return 0;

i++;

}}

char string3[] = "this is";

char string4[] = "a test";

if(strcmp(string3, string4) == 0)

printf("strings are equal\n");

else

printf("strings are different\n");

*Note: If fails what does it returns*

**Strstr strchr**

**strstr** will find a sub-string within a string. **strchr** will find the first matching character in a string.

char string1[]="red dwarf";

char string2[]="dwarf";

char \*pointer;

pointer = strstr(string1, string2);

Output : “dwarf” if printing pointer

int character='w';

char \*string="red dwarf";

if(strchr(string, character))

puts("Character found");

strstr returns a pointer to the beginning of the sub-string or NULL if not found.

**Strtok(still not clear)**

int x = 1;

char str[]="this:is:a:test:of:string:tokenizing";

char \*str1;

/\* print what we have so far \*/

printf("String: %s\n", str);

/\* extract first string from string sequence \*/

str1 = strtok(str, ":");

/\* print first string after tokenized \*/

printf("%i: %s\n", x, str1);

/\* loop until finishied \*/

while (1)

{

/\* extract string from string sequence \*/

str1 = strtok(NULL, ":");

/\* check if there is nothing else to extract \*/

if (str1 == NULL)

{

printf("Tokenizing complete\n");

exit(0);

}

/\* print string after tokenized \*/

printf("%i: %s\n", x, str1);

x++;

}

return 0;

**Memset :** Sets the n bytes pointed to by s to the value of (unsigned char) c & returns s. Basically it is being used to initialize the memory with zero after allocating memory using malloc.

**Memcpy :** This copies n bytes from the place pointed to by s2 to the place pointed to by s1. If the objects overlap, the result is undefined. The value of s1 is returned.

**Memmove :** Identical to memcpy, but works even for overlapping objects. It may be marginally slower, though.

***What is the difference between memcpy & memmove***

With memcpy, the destination cannot overlap the source at all. With memmove it can. This means that memmove might be very slightly slower than memcpy, as it cannot make the same assumptions.For example, memcpy might always copy addresses from low to high. If the destination overlaps after the source, this means some addresses will be overwritten before copied. memmove would detect this and copy in the other direction - from high to low - in this case. However, checking this and switching to another (possibly less efficient) algorithm takes time.

Now see the example for memset/memcpy/memmove

char msg[50] = "abcdefghijklmnopqrstuvwxyz";  
char temp[50];  
main()  
{  
strcpy(temp, msg);  
printf("Original Msg = %s\n",temp);  
  
memcpy(temp+4, temp+16, 10);  
printf("After memcpy without overlap = %s\n",temp);  
strcpy(temp , msg);  
memcpy(temp+6, temp+4, 10);  
printf("After memcpy with overlap = %s\n",temp);  
  
strcpy(temp, msg);  
printf("Original Msg = %s\n",temp);  
  
memmove(temp+4, temp+16, 10);  
printf("After memmove without overlap = %s\n",temp);  
strcpy(temp , msg);  
memmove(temp+6, temp+4, 10);  
printf("After memmove with overlap = %s\n",temp);  
}  
Original Msg = abcdefghijklmnopqrstuvwxyz  
After memcpy without overlap = abcdqrstuvwxyzopqrstuvwxyz   
After memcpy with overlap = abcdefefefefefefqrstuvwxyz   
  
Original Msg = abcdefghijklmnopqrstuvwxyz  
After memmove without overlap = abcdqrstuvwxyzopqrstuvwxyz   
After memmove with overlap = abcdefefghijklmnqrstuvwxyz

One more Example is, Say you have a list of 100 items in memory, taking 100 MBs of memory. You want to drop the 1st item so you only have 99.Memcpy will require the original 100 MBs and an additional 99 MBs for your *new* list of 99 items. Approximately 199 MBs total to perform the operation, but should be very fast. Memmove, in the worst scenario will require the original 100 MBs, and will move every item 1 memory address up at a time. This only requires the original 100 MBs, but will be significantly slower than Memcpy. Of course, creating a new pointer to point at the 2nd item in your list will achieve the same effect of "dropping" the first item from your list.

*Now let us see the implementation of memcpy & memmove*

*memcpy memmove*

void \*memmove(void \*dest, const void \*src, size\_t count){

char \*tmp;

const char \*s;

if (dest <= src) {

tmp = dest;

s = src;

while (count--)

\*tmp++ = \*s++;

} else {

tmp = dest;

tmp += count;

s = src;

s += count;

while (count--)

\*--tmp = \*--s;

}

return dest;

}

void\* memcpy(void\* dest, const void\* src, size\_t count) {  
        char\* dst8 = (char\*)dest;  
        char\* src8 = (char\*)src;  
 while (count--) {  
            \*dst8++ = \*src8++;  
         }  
        return dest;  
    }

**Some more important functions**

**Sprintf**

#include <stdio.h>

i = sprintf( s, control [, arg1, arg2, ...] );

char \*s; is the string to which the output should be written.

const char \*control; is a "printf" control string.

arg1, arg2, ... are the values to be output.

int i; is the number of characters that were output. If a write error occurred, a negative number is returned."sprintf" writes to the string "s". "sprintf" converts, formats, and prints its arguments under control of the string "control". For a full description of the formatting used by "sprintf".

char s[10];

int i = 10;

float f = 10.20;

sprintf(s,"%d %f",i,f);

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

Output: 10 10.200000

**Atoi/itoa**

atoi - convert a string to an integer

#include <[stdlib.h](http://www.opengroup.org/onlinepubs/000095399/basedefs/stdlib.h.html)>  
  
int atoi(const char \**str*);

char str[]= "234";  
  
int iVal;  
  
iVal = atoi(str);

printf("\niVal = %d\n",iVal); //234

Note: It basically subtracts '0' from each character to get you the numeric.

int a = atoi ("223asdf");

This will convert "223" to 223. The remainder of the string is not  
considered. If the string cannot be converted to a number, at all, atoi  
returns zero.