Memory Functions

- The string handling library also contains some functions for manipulating memory blocks. You will learn two of them: `memcpy` and `memmove`.

- These functions can be used with any type of pointers, but we will concantrate on their usage with character pointers, i.e., with strings.

- Both of them take three parameters: two strings (`s1` and `s2`) and an integer (`n`).

- The `memcpy` function copies `n` characters from the second string `s2` to the first string `s1`.

Example:

```
char s1[20], s2[] = "Copy this string";
memcpy(s1, s2, 17);
printf("%s\n%s\n", s1, s2);
```

Output:

```
Copy this string
Copy this string
```

- The terminating NULL character is not copied if `n` is not large enough. If `n` was 16 or smaller, `s1` would be followed with some nonsense characters.

- The `memmove` function copies `n` characters from the second string `s2` to the first string `s1`, as `memcpy` function. But, it performs this operation by the help of a temporary array. This allows to move characters from one part of a string to another part of the same string.

Example:

```
char x[] = "Ekin Kara";
printf("Before move x = %s\n", x);
printf("After  move x = %s\n", memmove(x, &x[5], 4));
```

Output:

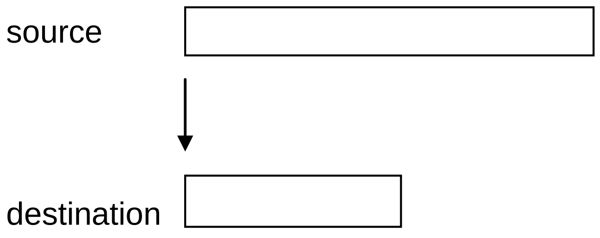
```
Before move x = Ekin Kara
After  move x = Kara Kara
```

- What about if `n` was **5**?

➢ *READ Sec. 8.9 (pg 333 – 337) from Deitel & Deitel.*

Protected Functions

- Using functions in <string.h> is sometimes DANGEROUS, because the space allocated for destination can be overflowed.



Example:

```
char y[8], x[] = "Good Morning";
strcpy(y, x);
```

- It does not give any error message, but overwrites the contents of the memory locations following the `y` array.
- Let's write a function to assign a string to another string, which is safer than `strcpy`, and use it in our programs instead of `strcpy` or `strncpy` from now on.

```
/*****************************************************************
   Copies into dest the string value of src, or as much as will fit
   given that dest has room for only dest_len characters, including \0.
   Extra characters are truncated.
*****************************************************************/
char * strassign(char dest[],      /* output - destination string      */
                 const char src[], /* input  - source string           */
                 int dest_len) {   /* input  - space available in dest */
   int new_len;              /* number of characters to copy into dest */
   int len = strlen(src);

   /* If the length of the source is less than the length of the
      destination, all characters in the source can be copied       */
   if (len < dest_len)
       new_len = len;
   else /* otherwise, it is possible to copy only dest_len - 1 characters */
       new_len = dest_len - 1;

   /* Copies new_len chars from src to dest using function that
      gives correct results even if src and dest overlap          */
   memmove(dest, src, new_len);

   /* Adds null character at end of string                        */
   dest[new_len] = '\0';

   return(dest);
}
```

- The keyword `const` in the second parameter tells to the computer that you don’t want the function to change the contents of that array. Thus, within the above function, if you try to make any change on the `src` array, it will cause a compilation error. We will use it for our input array parameters.

Example:

```
char y[8], x[] = "Good Morning";
strassign(y, x, 8);
printf("%s\n", y);
```

Output:

```
Good Mo
```

- To copy only 4 characters from x to y, we have to call `strassign` as

  ```
  strassign(y, x, 5);
  ```

  because one more position is needed for NULL character.

- Now, let’s write a function to concatenate two strings, which is safer than `strcat`, and use it in our programs instead of `strcat` or `strncat` from now on.

```
/*****************************************************************
   Concatenates str1 and str2, copying as much of the result as will
   fit into dest, which has space available for dest_len characters
   pre: dest_len <= MAX_STR_LEN for correct results
******************************************************************/
char * concat(
    char dest[],          /* output   - destination string        */
    const char str1[],    /* input    - strings                   */
    const char str2[],    /*              to concatenate         */
    int dest_len)         /* input    - available space in dest   */
{
   char result[MAX_STR_LEN]; /* local string space for result     */

   /* Checks if result will be wrong due to lack of local space   */
   if ( dest_len > MAX_STR_LEN  &&
        strlen(str1) + strlen(str2) >= MAX_STR_LEN )
      printf("\nInsufficent local storage causing loss of data!\n");

   /* Builds result in local storage to properly handle overlap of dest
      and str1 or str2                                                 */

   strassign(result, str1, MAX_STR_LEN);
   strncat(result, str2, MAX_STR_LEN - strlen(result) - 1);
   strassign(dest, result, dest_len);

   return(dest);
}
```

- The above protected `concat` function combines `str1` and `str2` copying as much of the result as will fit into `dest`, which has space available for `dest_len` characters. To obtain correct results, `MAX_STR_LEN` should be defined large enough (`MAX_STR_LEN >= dest_len`).

Example:

```
char s1[] = "Very Happy ";
char s2[] = "New Year", s3[30];
printf("%s\n", concat(s3, s1, s2, 30));
```

Output:

```
Very Happy New Year
```

- What would be the output, if the last parameter was 17?

- Another very useful operation with strings is the *substring* operation, i.e., taking a certain part of a string.

```
/*********************************************************************
   Extracts a substring from src and returns it in dest, truncating if
   the substring contains dest_len or more characters. The substring
   extracted starts with src[start] and contains characters including
   src[end] unless src[end] is beyond the end of src, in which case the
   substring returned is src[start] to the end of src. The empty string
   is returned if  start > end  or  if start >= strlen(src)
*********************************************************************/
char * substr(
       char dest[],       /* output - destination string             */
       char src[],        /* input  - source string                  */
       int start,         /* input  - subscript of first character   */
                          /*          of src to include              */
       int end,           /* input  - subscript of last char included*/
       int dest_len)      /* input  - space available in dest        */
```

```
{
    int sub_len;           /* length of substring returned           */
    int len = strlen(src);

    /* Adjusts start and end  if they are beyond the ends of src    */
    if (end > len - 1)
       end = len - 1;

    if (start < 0)
       start = 0;

    if (start > end)       /* if starting point is past ending point */
       dest[0] = '\0';
    else
    {   /* Find the length of the substring                          */
       sub_len = end - start + 1;

       /* sub_len can not be greater than dest_len - 1              */
       if (sub_len >= dest_len)
          sub_len = dest_len - 1;

       memmove(dest, src + start, sub_len);
       dest[sub_len] = '\0';
    }

    return(dest);
}
```

Example:

```
char result[10], str[] = "Jan. 30, 2007";
printf("%s\n", substr(result, str, 5, 6, 10));
```

Output:

```
30
```

- Notice that `gets` function may also cause memory overflow, if the input string does not fit to the destination.

Example:

```
char y[8];
gets(y);
```

- When the input string is more than 7 characters, it overwrites the contents of the memory locations following the `y` array.

- The following function gets one line of input data and stores only the part that fits into destination.

```
/*******************************************************************
   Gets one line of data from standard input. Returns an empty string
   on end of file.
*******************************************************************/
char * scanline(
         char dest[],        /* output  - destination string         */
         int  dest_len)      /* input   - space available in dest    */
{
   /* Uses library function fgets to get the next input line. if
      fgets returns 0, signals end of file by returning an empty
      string.                                                       */

   if (fgets(dest, dest_len, stdin) == 0)
      dest[0] = '\0';

   /* Removes new line character if it is present                   */
   else
      if (dest[strlen(dest) - 1] == '\n')
         dest[strlen(dest) - 1] = '\0';

   return(dest);
}
```

Example:

```
     char y[8];
     scanline(y, 8);
```

- As you know, it is possible to put user-defined functions in a library, create a header file for that library, and include it in any program that you need to use those functions.

- We have created a library containing those four protected string functions `strassign`, `concat`, `substr`, and `scanline`, and named it as `strman`. So, from now on, whenever you need to use them, you will include `strman.h` to your program.

- You need to copy it to the following folder:

  `C:\Program Files\Microsoft Visual Studio\VC98\Include`

- You know that parameters in any function may be a call to another function. We will frequently be doing this in string manipulation. Always make sure that the parentheses are matched, and the type and sequence (order) of the parameters are as required by the function.

Example: Find the output of the following program segment:

```
#include <stdio.h>
#include <strman.h>
#define MAX 15
…
char s1[] = "parameters";
char s2[] = "one and half kilo";
char s3[MAX], temp1[MAX], temp2[MAX];
concat(s3, substr(temp1, s2, strlen(s1)+3, strlen(s2)-1, MAX),
           substr(temp2, s1, 4, strlen(s1)-2, MAX), MAX);
printf("%s\n", s3);
```

- The strings to be joined are

  ```
  substr(temp1, s2, strlen(s1)+3, strlen(s2)-1, MAX)
  ```

  and

  ```
  substr(temp2, s1, 4, strlen(s1)-2, MAX)
  ```

- Now, let's analyze the first string. `s2` is the second parameter of `substr` function. Thus, the string we should look at is `s2`, starting at the point `strlen(s1)+3`, finishing at the point `strlen(s2)-1`. Substituting the values for `strlen(s1)` as **10**, and `strlen(s2)` as **17**, the second parameter becomes:

  ```
  substr(temp1, s2, 13, 16, MAX)
  ```

  Thus `temp1` is **"kilo"**. Analyzing the second string in the same way gives:

  ```
  substr(temp2, s1, 4, 8, MAX)
  ```

  Thus `temp2` is **"meter"**. Now we can easily see the parameters for the `concat` function:

  ```
  concat (s3, "kilo", "meter", MAX)
  ```

  which stores **"kilometer"** into the string `s3`. So the output of the above program segment is **kilometer**.

- Notice that, it would be better to define an integer to store the result of `strlen(s1)` so that it wouldn't be called twice.

Example: Write a function to insert a string after the nth character of another string.

```
void insert_string(char *str1, char *str2, int n)
{
     char temp1[MAX], temp2[MAX]; // temporary strings
     int l1 = strlen(str1); // length of the first string
     if (n < 0 || n > l1)
        printf("Error, Position is out of bounds!\n");
     else
     {  // Copy the first n characters of the first string
        substr(temp1, str1, 0, n-1, MAX);
        // Concatenate with the second string
        concat(temp1, temp1, str2, MAX);
        // Copy the part after the nth character
        substr(temp2, str1, n, l1-1, MAX);
        concat(str1, temp1, temp2, MAX);
     }
}
```

- Let's write a main that inputs a sentence and a word and inserts the word at the beginning of the sentence.

```
int main(void)
{
     char sent[MAX];
     char word[15];
     printf("Enter a sentence: ");
     scanline (sent, MAX);
     printf("Enter a word to insert in the sentence: ");
     scanf("%s", word);
     concat(word, word, " ", 15); // Concat. a blank
     insert_string(sent, word, 0);
     printf("\nThe new sentence: %s\n", sent);
     return(0);
}
```

Output:

```
Enter a sentence: baby is sleeping
Enter a word to insert in the sentence: little

The new sentence: little baby is sleeping
```