VI
CHAPTER

Working with Strings

This chapter focuses on manipulating strings—copying strings, copying portions of strings, performing string comparisons, right-justifying, removing trailing and leading spaces, executing string conversions, and more. The standard C library provides many functions that aid in string manipulation, and several of these functions will be covered in this chapter.

You will probably find yourself using string manipulation techniques often when writing your C programs. The examples in this chapter are provided to help you get a jump-start on utilizing the functions you need in order to be productive right away. Pay close attention to the examples provided here—several of them could save you valuable time when you're writing your programs.

VI.1: What is the difference between a string copy (*strcpy*) and a memory copy (*memcpy*)? When should each be used?

Answer:

The `strcpy()` function is designed to work exclusively with strings. It copies each byte of the source string to the destination string and stops when the terminating *null character*

(`\0`) has been moved. On the other hand, the `memcpy()` function is designed to work with any type of data. Because not all data ends with a null character, you must provide the `memcpy()` function with the number of bytes you want to copy from the source to the destination. The following program shows examples of both the `strcpy()` and the `memcpy()` functions:

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

typedef struct cust_str {
     int  id;
     char last_name[20];
     char first_name[15];
} CUSTREC;

void main(void);

void main(void)
{

     char*   src_string = "This is the source string";
     char    dest_string[50];
     CUSTREC src_cust;
     CUSTREC dest_cust;

     printf("Hello!  I'm going to copy src_string into dest_string!\n");

     /* Copy src_string into dest_string. Notice that the destination
        string is the first argument. Notice also that the strcpy()
        function returns a pointer to the destination string. */

     printf("Done! dest_string is: %s\n",
            strcpy(dest_string, src_string));

     printf("Encore! Let's copy one CUSTREC to another.\n");
     printf("I'll copy src_cust into dest_cust.\n");

     /* First, initialize the src_cust data members. */

     src_cust.id = 1;

     strcpy(src_cust.last_name, "Strahan");
     strcpy(src_cust.first_name, "Troy");

     /* Now, use the memcpy() function to copy the src_cust structure to
        the dest_cust structure. Notice that, just as with strcpy(), the
        destination comes first. */

     memcpy(&dest_cust, &src_cust, sizeof(CUSTREC));

     printf("Done! I just copied customer number #%d (%s %s).",
               dest_cust.id, dest_cust.first_name, dest_cust.last_name);

}
```

When dealing with strings, you generally should use the `strcpy()` function, because it is easier to use with strings. When dealing with abstract data other than strings (such as structures), you should use the `memcpy()` function.

Cross Reference:

VI.6: How can I copy just a portion of a string?

VI.9: How do you print only part of a string?

VI.2: How can I remove the trailing spaces from a string?

Answer:

The C language does not provide a standard function that removes trailing spaces from a string. It is easy, however, to build your own function to do just this. The following program uses a custom function named `rtrim()` to remove the trailing spaces from a string. It carries out this action by iterating through the string backward, starting at the character before the terminating null character (`\0`) and ending when it finds the first nonspace character. When the program finds a nonspace character, it sets the next character in the string to the terminating null character (`\0`), thereby effectively eliminating all the trailing blanks. Here is how this task is performed:

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

void main(void);
char* rtrim(char*);

void main(void)
{

     char* trail_str = "This string has trailing spaces in it.               ";

     /* Show the status of the string before calling the rtrim()
        function. */

     printf("Before calling rtrim(), trail_str is '%s'\n", trail_str);
     printf("and has a length of %d.\n", strlen(trail_str));

     /* Call the rtrim() function to remove the trailing blanks. */

     rtrim(trail_str);

     /* Show the status of the string
        after calling the rtrim() function. */

     printf("After calling rtrim(), trail_str is '%s'\n", trail_str);
     printf("and has a length of %d.\n", strlen(trail_str));

}

/* The rtrim() function removes trailing spaces from a string. */

char* rtrim(char* str)
{
     int n = strlen(str) - 1;     /* Start at the character BEFORE
                                     the null character (\0). */

     while (n>0)            /* Make sure we don’t go out of bounds... */
     {
          if (*(str+n) != ‘ ‘)    /*  If we find a nonspace character: */
          {
               *(str+n+1) = ‘\0’; /* Put the null character at one
                                     character past our current
                                     position. */
               break;             /* Break out of the loop. */
          }
          else      /* Otherwise, keep moving backward in the string. */
               n--;
     }

     return str;                  /* Return a pointer to the string. */

}
```

Notice that the `rtrim()` function works because in C, strings are terminated by the null character. With the insertion of a null character after the last nonspace character, the string is considered terminated at that point, and all characters beyond the null character are ignored.

Cross Reference:

VI.3: How can I remove the leading spaces from a string?

VI.5: How can I pad a string to a known length?

VI.3: How can I remove the leading spaces from a string?

Answer:

The C language does not provide a standard function that removes leading spaces from a string. It is easy, however, to build your own function to do just this. Using the example from FAQ VI.2, you can easily construct a custom function that uses the `rtrim()` function in conjunction with the standard C library function `strrev()` to remove the leading spaces from a string. Look at how this task is performed:

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

void main(void);
char* ltrim(char*);
char* rtrim(char*);

void main(void)
{

     char* lead_str = “          This string has leading spaces in it.”;

     /* Show the status of the string before calling the ltrim()
        function. */
     printf("Before calling ltrim(), lead_str is '%s'\n", lead_str);
     printf("and has a length of %d.\n", strlen(lead_str));

     /* Call the ltrim() function to remove the leading blanks. */

     ltrim(lead_str);

     /* Show the status of the string
        after calling the ltrim() function. */

     printf("After calling ltrim(), lead_str is '%s'\n", lead_str);
     printf("and has a length of %d.\n", strlen(lead_str));

}

/* The ltrim() function removes leading spaces from a string. */

char* ltrim(char* str)
{

     strrev(str);    /* Call strrev() to reverse the string. */

     rtrim(str);     /* Call rtrim() to remove the "trailing" spaces. */

     strrev(str);    /* Restore the string's original order. */

     return str;     /* Return a pointer to the string. */

}

/* The rtrim() function removes trailing spaces from a string. */

char* rtrim(char* str)
{

     int n = strlen(str) - 1;     /* Start at the character BEFORE
                                     the null character (\0). */

     while (n>0)            /* Make sure we don't go out of bounds... */
     {
          if (*(str+n) != ' ')    /* If we find a nonspace character: */
          {
               *(str+n+1) = '\0'; /* Put the null character at one
                                     character past our current
                                     position. */
               break;             /* Break out of the loop. */
          }
          else      /* Otherwise, keep moving backward in the string. */
               n--;
     }

     return str;                  /* Return a pointer to the string. */

}
```

Notice that the `ltrim()` function performs the following tasks: First, it calls the standard C library function `strrev()`, which reverses the string that is passed to it. This action puts the original string in reverse order, thereby creating "trailing spaces" rather than leading spaces. Now, the `rtrim()` function (that was created in the example from FAQ VI.2) is used to remove the “trailing spaces” from the string. After this task is done, the `strrev()` function is called again to “reverse” the string, thereby putting it back in its original order. See FAQ VI.2 for an explanation of the `rtrim()` function.

Cross Reference:

VI.2: How can I remove the trailing spaces from a string?

VI.5: How can I pad a string to a known length?

VI.4: How can I right-justify a string?

Answer:

Even though the C language does not provide a standard function that right-justifies a string, you can easily build your own function to perform this action. Using the `rtrim()` function (introduced in the example for FAQ VI.2), you can create your own function to take a string and right-justify it. Here is how this task is accomplished:

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

void main(void);
char* rjust(char*);
char* rtrim(char*);

void main(void)
{

     char* rjust_str = “This string is not right-justified.                   “;

     /* Show the status of the string before calling the rjust()
        function. */

     printf(“Before calling rjust(), rjust_str is ‘%s’\n.”, rjust_str);

     /* Call the rjust() function to right-justify this string. */

     rjust(rjust_str);

     /* Show the status of the string
        after calling the rjust() function. */

     printf(“After calling rjust(), rjust_str is ‘%s’\n.”, rjust_str);

}
/* The rjust() function right-justifies a string. */

char* rjust(char* str)
{

     int n = strlen(str);   /* Save the original length of the string. */
     char* dup_str;
     dup_str = strdup(str);  /* Make an exact duplicate of the string. */

     rtrim(dup_str);         /* Trim off the trailing spaces. */

     /* Call sprintf() to do a virtual "printf" back into the original
        string. By passing sprintf() the length of the original string,
        we force the output to be the same size as the original, and by
        default the sprintf() right-justifies the output. The sprintf()
        function fills the beginning of the string with spaces to make
        it the same size as the original string. */

     sprintf(str, "%*.*s", n, n, dup_str);

     free(dup_str);    /* Free the memory taken by
                          the duplicated string. */

     return str;       /* Return a pointer to the string. */

}

/* The rtrim() function removes trailing spaces from a string. */

char* rtrim(char* str)
{

     int n = strlen(str) - 1;  /* Start at the character BEFORE the null
                                  character (\0). */

     while (n>0)            /* Make sure we don't go out of bounds... */
     {
          if (*(str+n) != ' ')    /* If we find a nonspace character: */
          {
               *(str+n+1) = '\0'; /* Put the null character at one
                                     character past our current
                                     position. */
               break;             /* Break out of the loop. */
          }
          else      /* Otherwise, keep moving backward in the string. */
               n--;
     }

     return str;                  /* Return a pointer to the string. */

}
```

The `rjust()` function first saves the length of the original string in a variable named `n`. This step is needed because the output string must be the same length as the input string. Next, the `rjust()` function calls the standard C library function named `strdup()` to create a duplicate of the original string. A duplicate of the string is required because the original version of the string is going to be overwritten with a right-justified version. After the duplicate string is created, a call to the `rtrim()` function is invoked (using the duplicate string, not the original), which eliminates all trailing spaces from the duplicate string.

Next, the standard C library function `sprintf()` is called to rewrite the new string to its original place in memory. The `sprintf()` function is passed the original length of the string (stored in `n`), thereby forcing the output string to be the same length as the original. Because `sprintf()` by default right-justifies string output, the output string is filled with leading spaces to make it the same size as the original string. This has the effect of right-justifying the input string. Finally, because the `strdup()` function dynamically allocates memory, the `free()` function is called to free up the memory taken by the duplicate string.

Cross Reference:

VI.5: How can I pad a string to a known length?

VI.5: How can I pad a string to a known length?

Answer:

Padding strings to a fixed length can be handy when you are printing fixed-length data such as tables or spreadsheets. You can easily perform this task using the `printf()` function. The following example program shows how to accomplish this task:

```
#include <stdio.h>

char *data[25] = {
     "REGION", "--Q1--",    "--Q2--",   "--Q3--", "  --Q4--",
     "North", "10090.50", "12200.10", "26653.12", "62634.32",
     "South", "21662.37", "95843.23", "23788.23", "48279.28",
     "East", "23889.38", "23789.05", "89432.84", "29874.48",
     "West", "85933.82", "74373.23", "78457.23", "28799.84" };

void main(void);

void main(void)
{

     int x;

     for (x=0; x<25; x++)
     {

          if ((x % 5) == 0 && (x != 0))
               printf("\n");

          printf("%-10.10s", data[x]);

     }

}
```

In this example, a character array (`char* data[]`) is filled with this year's sales data for four regions. Of course, you would want to print this data in an orderly fashion, not just print one figure after the other with no formatting. This being the case, the following statement is used to print the data:

```
printf("%-10.10s", data[x]);
```

The `"%-10.10s"` argument tells the `printf()` function that you are printing a string and you want to force it to be 10 characters long. By default, the string is right-justified, but by including the minus sign (–) before the first `10`, you tell the `printf()` function to left-justify your string. This action forces the `printf()` function to pad the string with spaces to make it 10 characters long. The result is a clean, formatted spreadsheet-like output:

```
REGION      --Q1--    --Q2--     --Q3--     --Q4--
North      10090.50  12200.10  26653.12  62634.32
South      21662.37  95843.23  23788.23  48279.28
East       23889.38  23789.05  89432.84  29874.48
West       85933.82  74373.23  78457.23  28799.84
```

Cross Reference:

VI.4: How can I right-justify a string?

VI.6: How can I copy just a portion of a string?

Answer:

You can use the standard C library function `strncpy()` to copy one portion of a string into another string. The `strncpy()` function takes three arguments: the first argument is the destination string, the second argument is the source string, and the third argument is an integer representing the number of characters you want to copy from the source string to the destination string. For example, consider the following program, which uses the `strncpy()` function to copy portions of one string to another:

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

void main(void);

void main(void)
{

     char* source_str = "THIS IS THE SOURCE STRING";
     char dest_str1[40] = {0}, dest_str2[40] = {0};

     /* Use strncpy() to copy only the first 11 characters. */

     strncpy(dest_str1, source_str, 11);

     printf("How about that! dest_str1 is now: '%s'!!!\n", dest_str1);

     /* Now, use strncpy() to copy only the last 13 characters. */

     strncpy(dest_str2, source_str + (strlen(source_str) - 13), 13);

     printf("Whoa! dest_str2 is now: '%s'!!!\n", dest_str2);

}
```

The first call to `strncpy()` in this example program copies the first 11 characters of the source string into `dest_str1`. This example is fairly straightforward, one you might use often. The second call is a bit more complicated and deserves some explanation. In the second argument to the strncpy() function call, the total length of the source_str string is calculated (using the strlen() function). Then, 13 (the number of characters you want to print) is subtracted from the total length of source_str. This gives the number of remaining characters in source_str. This number is then added to the address of source_str to give a pointer to an address in the source string that is 13 characters from the end of source_str. Then, for the last argument, the number 13 is specified to denote that 13 characters are to be copied out of the string. The combination of these three arguments in the second call to strncpy() sets dest_str2 equal to the last 13 characters of source_str.

The example program prints the following output:

```
How about that! dest_str1 is now: 'THIS IS THE'!!!
Whoa! dest_str2 is now: 'SOURCE STRING'!!!
```

Notice that before source_str was copied to dest_str1 and dest_st2, dest_str1 and dest_str2 had to be initialized to null characters (\0). This is because the strncpy() function does not automatically append a null character to the string you are copying to. Therefore, you must ensure that you have put the null character after the string you have copied, or else you might wind up with garbage being printed.

Cross Reference:

VI.1: What is the difference between a string copy (strcpy) and a memory copy (memcpy)? When should each be used?

VI.9: How do you print only part of a string?

VI.7: How can I convert a number to a string?

Answer:

The standard C library provides several functions for converting numbers of all formats (integers, longs, floats, and so on) to strings and vice versa. One of these functions, itoa(), is used here to illustrate how an integer is converted to a string:

```
#include <stdio.h>
#include <stdlib.h>

void main(void);

void main(void)
{

     int num = 100;
     char str[25];

     itoa(num, str, 10);

     printf("The number 'num' is %d and the string 'str' is %s.\n",
               num, str);

}
```

Notice that the `itoa()` function takes three arguments: the first argument is the number you want to convert to the string, the second is the destination string to put the converted number into, and the third is the base, or radix, to be used when converting the number. The preceding example uses the common base 10 to convert the number to the string.

The following functions can be used to convert integers to strings:

Function Name	*Purpose*
`itoa()`	Converts an integer value to a string.
`ltoa()`	Converts a long integer value to a string.
`ultoa()`	Converts an unsigned long integer value to a string.

Note that the `itoa()`, `ltoa()`, and `ultoa()` functions are not ANSI compatible. An alternative way to convert an integer to a string (that is ANSI compatible) is to use the `sprintf()` function, as in the following example:

```
#include <stdio.h>
#include <stdlib.h>

void main(void);

void main(void)
{

     int num = 100;
     char str[25];

     sprintf(str, "%d", num);

     printf("The number 'num' is %d and the string 'str' is %s.\n",
              num, str);

}
```

When floating-point numbers are being converted, a different set of functions must be used. Here is an example of a program that uses the standard C library function `fcvt()` to convert a floating-point value to a string:

```
#include <stdio.h>
#include <stdlib.h>

void main(void);

void main(void)
{

     double num = 12345.678;
     char* str;
     int dec_pl, sign, ndigits = 3;    /* Keep 3 digits of precision. */

     str = fcvt(num, ndigits, &dec_pl, &sign);  /* Convert the float
                                                   to a string. */

     printf("Original number:  %f\n", num);     /* Print the original
                                                   floating-point
                                                   value. */
     printf("Converted string: %s\n", str);     /* Print the converted
```

```
                                                     string's value. */
     printf("Decimal place:    %d\n", dec_pl);  /* Print the location of
                                                     the decimal point. */
     printf("Sign:             %d\n", sign);    /* Print the sign.
                                                     0 = positive,
                                                     1 = negative. */

}
```

Notice that the `fcvt()` function is quite different from the `itoa()` function used previously. The `fcvt()` function takes four arguments. The first argument is the floating-point value you want to convert. The second argument is the number of digits to be stored to the right of the decimal point. The third argument is a pointer to an integer that is used to return the position of the decimal point in the converted string. The fourth argument is a pointer to an integer that is used to return the sign of the converted number (0 is positive, 1 is negative).

Note that the converted string does not contain the actual decimal point. Instead, the `fcvt()` returns the *position* of the decimal point as it would have been if it were in the string. In the preceding example, the `dec_pl` integer variable contains the number 5 because the decimal point is located after the fifth digit in the resulting string. If you wanted the resulting string to include the decimal point, you could use the `gcvt()` function (described in the following table).

The following functions can be used to convert floating-point values to strings:

Function Name	*Purpose*
`ecvt()`	Converts a double-precision floating-point value to a string without an embedded decimal point.
`fcvt()`	Same as `ecvt()`, but forces the precision to a specified number of digits.
`gcvt()`	Converts a double-precision floating-point value to a string with an embedded decimal point.

See FAQ VI.8 for an explanation of how you can convert strings to numbers.

Cross Reference:

VI.8: How can I convert a string to a number?

VI.8: How can I convert a string to a number?

Answer:

The standard C library provides several functions for converting strings to numbers of all formats (integers, longs, floats, and so on) and vice versa. One of these functions, `atoi()`, is used here to illustrate how a string is converted to an integer:

```
#include <stdio.h>
#include <stdlib.h>

void main(void);
void main(void)
{

     int num;
     char* str = "100";

     num = atoi(str);

     printf("The string 'str' is %s and the number 'num' is %d.\n",
               str, num);

}
```

To use the `atoi()` function, you simply pass it the string containing the number you want to convert. The return value from the `atoi()` function is the converted integer value.

The following functions can be used to convert strings to numbers:

Function Name	*Purpose*
`atof()`	Converts a string to a double-precision floating-point value.
`atoi()`	Converts a string to an integer.
`atol()`	Converts a string to a long integer.
`strtod()`	Converts a string to a double-precision floating-point value and reports any "leftover" numbers that could not be converted.
`strtol()`	Converts a string to a long integer and reports any "leftover" numbers that could not be converted.
`strtoul()`	Converts a string to an unsigned long integer and reports any "leftover" numbers that could not be converted.

Sometimes, you might want to trap overflow errors that can occur when converting a string to a number that results in an overflow condition. The following program shows an example of the `strtoul()` function, which traps this overflow condition:

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

void main(void);

void main(void)
{

     char* str  = "1234567891011121314151617181920";
     unsigned long num;
     char* leftover;

     num = strtoul(str, &leftover, 10);

     printf("Original string:      %s\n", str);
     printf("Converted number:     %lu\n", num);
     printf("Leftover characters:  %s\n", leftover);

}
```

In this example, the string to be converted is much too large to fit into an unsigned long integer variable. The `strtoul()` function therefore returns `ULONG_MAX` (4294967295) and sets the `char* leftover` to point to the character in the string that caused it to overflow. It also sets the global variable `errno` to `ERANGE` to notify the caller of the function that an overflow condition has occurred. The `strtod()` and `strtol()` functions work exactly the same way as the `strtoul()` function shown above. Refer to your C compiler documentation for more information regarding the syntax of these functions.

Cross Reference:

VI.7: How can I convert a number to a string?

VI.9: How do you print only part of a string?

Answer:

FAQ VI.6 showed you how to copy only part of a string. But how do you *print* a portion of a string? The answer is to use some of the same techniques as in the example for FAQ VI.6, except this time, rather than the `strncpy()` function, the `printf()` function is used. The following program is a modified version of the example from FAQ VI.6 that shows how to print only part of a string using the `printf()` function:

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

void main(void);

void main(void)
{

     char* source_str = “THIS IS THE SOURCE STRING”;

     /* Use printf() to print the first 11 characters of source_str. */

     printf(“First 11 characters: ‘%11.11s’\n”, source_str);

     /* Use printf() to print only the
        last 13 characters of source_str. */

     printf(“Last 13 characters: ‘%13.13s’\n”,
                 source_str + (strlen(source_str) - 13));

}
```

This example program produces the following output:

```
First 11 characters: ‘THIS IS THE’
Last 13 characters: ‘SOURCE STRING’
```

The first call to `printf()` uses the argument `“%11.11s”` to force the `printf()` function to make the output exactly 11 characters long. Because the source string is longer than 11 characters, it is truncated, and only the first 11 characters are printed. The second call to `printf()` is a bit more tricky. The total length of the `source_str` string is calculated (using the `strlen()` function). Then, 13 (the number of characters you want to print) is subtracted from the total length of `source_str`. This gives the number of remaining characters in `source_str`. This number is then added to the address of `source_str` to give a pointer to an address in the source string that is 13 characters from the end of `source_str`. By using the argument `“%13.13s”`, the program forces the output to be exactly 13 characters long, and thus the last 13 characters of the string are printed.

See FAQ VI.6 for a similar example of extracting a portion of a string using the `strncpy()` function rather than the `printf()` function.

Cross Reference:

VI.1: What is the difference between a string copy (`strcpy`) and a memory copy (`memcpy`)? When should each be used?

VI.6: How can I copy just a portion of a string?

VI.10: How do you remove spaces from the end of a string?

Answer:

See the answer to FAQ VI.2.

Cross Reference:

VI.2: How can I remove the trailing spaces from a string?

VI.11: How can you tell whether two strings are the same?

Answer:

The standard C library provides several functions to compare two strings to see whether they are the same. One of these functions, `strcmp()`, is used here to show how this task is accomplished:

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

void main(void);

void main(void)
{

     char* str_1 = “abc”;
     char* str_2 = “abc”;
     char* str_3 = “ABC”;

     if (strcmp(str_1, str_2) == 0)
          printf(“str_1 is equal to str_2.\n”);
     else
          printf(“str_1 is not equal to str_2.\n”);
     if (strcmp(str_1, str_3) == 0)
          printf("str_1 is equal to str_3.\n");
     else
          printf("str_1 is not equal to str_3.\n");

}
```

This program produces the following output:

```
str_1 is equal to str_2.
str_1 is not equal to str_3.
```

Notice that the `strcmp()` function is passed two arguments that correspond to the two strings you want to compare. It performs a case-sensitive lexicographic comparison of the two strings and returns one of the following values:

Return Value	*Meaning*
< 0	The first string is less than the second string.
0	The two strings are equal.
> 0	The first string is greater than the second string.

In the preceding example code, `strcmp()` returns 0 when comparing `str_1` (which is `"abc"`) and `str_2` (which is `"abc"`). However, when comparing `str_1` (which is `"abc"`) with `str_3` (which is `"ABC"`), `strcmp()` returns a value greater than 0, because the string `"ABC"` is greater than (in ASCII order) the string `"abc"`.

Many variations of the `strcmp()` function perform the same basic function (comparing two strings), but with slight differences. The following table lists some of the functions available that are similar to `strcmp()`:

Function Name	*Description*
`strcmp()`	Case-sensitive comparison of two strings
`strcmpi()`	Case-insensitive comparison of two strings
`stricmp()`	Same as `strcmpi()`
`strncmp()`	Case-sensitive comparison of a portion of two strings
`strnicmp()`	Case-insensitive comparison of a portion of two strings

Looking at the example provided previously, if you were to replace the call to `strcmp()` with a call to `strcmpi()` (a case-insensitive version of `strcmp()`), the two strings `"abc"` and `"ABC"` would be reported as being equal.

Cross Reference:

VI.1: What is the difference between a string copy (`strcpy`) and a memory copy (`memcpy`)? When should each be used?