Lesson 9

STRING PROCESSING

In C, a string is a one-dimensional array of type char. A character in a string can be accessed either as an element in an array or by making use of a pointer to char. The flexibility this provides makes C especially useful in writing string processing programs. The standard library provides many useful string handling functions.

While string processing can be viewed as a special case of array processing, it has characteristics that give it a unique flavor. Important to this is the use of the character value \0 to terminate a string. This lesson includes a number of example programs that illustrate string processing ideas. Again, as in the previous lesson, the important relationship between pointer and array is shown. The type pointer to char is conceptually a string. An examples will illustrate the pointer arithmetic and dereferencing needed to properly process string data.

By convention, a string is terminated by the end-of-string sentinel \0, or null character. A constant string such as "abc" is stored in memory as four characters, the last one being \0. Thus the length of the string "abc" is 3, but the size of the string is 4. Notice that the string uses the " character and the char uses the ' character. To allocate storage in memory for a string, you could write

```
#define MAXWORD 100
main(   )
{
char w[MAXWORD]:
```

After storage has been allocated, there are a number of ways of getting character values into the string w. First, you can do it character by character as follows:

```
w[0] = 'A';
w[1] = 'B';
w[2] = 'C':
w[3] = '\0';
```

Notice that the string ended with the null character.

Another way to get character values into w is to make use of scanf( ). The format %s is used to read in a string. The statement

```
scanf("%s", w)
```


University of Houston Clear Lake

causes all white space characters in the input stream to be skipped. Then non-white space characters (characters that use ink or toner when you send them to the printer) are read in and placed in memory beginning at the base address of w. The process stops when a white space character or EOF is encountered. At that point a null character is placed in memory to end the string. Notice that the address operator &w was not use as the second argument to scanf( ). To do so would be an error. Since an array name is a pointer to the base address of the array, the expression w is equivalent to &w[0]. Because the size of w is 100, up to 99 characters can be entered into this array. If more are entered, the bounds of the array w will overrun.

The sentinel \0, also called a delimiter, allows a simple test to detect the end of a string. It is useful to think of strings as having a variable length delimited by the null character, but with a maximum length determined by the size of the string. The size of a string must include the storage needed for the null character. As with all arrays it is the job of the programmer to make sure that string bounds are not overrun.

Note carefully that 'a' and "a" are different. The first is a character constant, and the second is a string constant. The string "a" is an array of characters with two elements, the first with value 'a' and the second with value \0.

Recall that external and static arrays can be initialized. This feature works with character arrays as well. However, the compiler allows for an initialization of the form

```
static char s[ ] = "abc";
```

which is taken to be equivalent to

```
static char s[ ] = {'a', 'b', 'c', '\0'};
```

Automatic arrays, including character arrays, cannot be initialized under K&R compilers (they can be initialized under ANSI rules). However, it is possible to initialize a pointer to char of any storage class to a constant string. Consider, for example,

```
char *p = "abc";
```

The effect of this is for the string "abc" to be placed in memory and for the pointer p to be initialized with the base address of the string.

Since a string is an array of characters, one way to process a string is to use array notation with subscripts. An interactive program to illustrate this will read a

line of characters typed by the user into a string, print the string in reverse order, and then sum the elements of the string.

```
 /* Have a nice day! */
#define MAXLINE 100
main(   )
{
   char c, line [MAXLINE];
   int  i, sum = 0;
   printf("\nHi! What is your name? ");
   for (i=0; (c= getcharO) l= '\n'; ++i)
     line [i] = c;
   line [i] = '\0';
   printf( "\n%s%s%s\n",  "Nice to meet you ", line,".");
   printf( "Your name spelled backwards is ");
   while (i > 0)
      putchar( line [--i] ) ;
   putchar ( '.' ) ;
   for (i = 0; line [i] /= '\0'f ++i)
     sum += line[ i ];
   printf( "\n%s%d%s\n\n%s\n\n",
   "Your character sum is ", sum, ". ",
   "Have a nice day! " ) ;
}
```

Execute this program and enter C. B. Diligent when prompted. Here is what appears on the screen:

Hi! What is your name? C. B. Diligent

Nice to meet you C. B. Diligent.

Your name spelled backwards is tnegiliD .B .C.

Your character sum is 1105.

Have a nice day!

PROGRAM ANALYSIS

```
#define MAXUNE 100

main(   )
{

   char c, line[MAXLINE];

   int i, sum = 0;
```

The character array line has size 100. Since the null character is always used to delimit a string, this means that the array can hold strings of length up to 99. The length of a string is a count of all characters up to, but not including, the null character. In this program you are making the assumption that the user will not type in more than 99 characters.

```
printf("\nHi! What is your name? ");
```

This is a prompt to the user. The program now expects a name to be typed in, followed by a carriage return.

```
for (i = 0;  (c = getchar(   ) ) != '\n'; + +i)
   line[ i ] = c;
```

In this for loop the variable 1 is initialized to 0. The function getchar( ) gets a character and assigns it to c. If c is not a newline character, then it is assigned to the array element line[i] and 1 is incremented. The loop is repeatedly executed until a newline character is received.

```
line[ i ] = '\0';
```

After the for loop is finished, the null character '\0' is assigned to the element line[i]. By convention, all strings end with a null character. Functions that process strings, such as printf( ), use the null character as an end-of-string sentinel.

```
printf("\n%s%s%s\n", "Nice to meet you ", line, ".");
```

The format %s is used to print a string. Here, the array line is one of three string arguments that are being printed. The effect of this statement is to print:

```
Nice to meet you C. B. Diligent,
```

on the screen.

```
printf ("Your name spelled backwards is ");
while (i > 0)
   putchar(line[ --i ]);
putcharC.');
```

At the start of this while loop the value of i is 14 and the value of line[i] is a null character. The expression i first causes the stored value of i to be decremented by 1, and then the expression takes on this value. The effect of this loop is to print backwards the characters stored in the array. After the loop is finished, a period is printed to end the sentence.

```
for(i = 0; line[i]  != '\0'; ++i)
   sum  += line[i];
```

Characters in C have integer value, namely the value of their ASCII code representation. This for loop is summing the values of the characters in the array. The loop stops when a newline character is reached.

```
printf("\n%s%d%s\n\n%s\n\n",
        "Your character sum is",sum,".",
        "Have a nice day!"):
```

The control string

```
 "\n%s%d%s\n\n%s\n\n"
```

is telling printf( ) to print its remaining arguments in turn as a string, a decimal integer, a string, and a string interspersed with newline characters.

String Processing Using Subscripts

In the last part of this lesson string processing with the use of subscripts is illustrated. In this section you want to use pointers to process a string. Also, you want to show how strings can be used as arguments to functions. Let us write a small interactive program that reads into a string a line of characters input by the user. Then the program will use this to create a new string and print it.

```
/* Character processing; change a line. */
 #define MAXLINE 100
#include <stdio.h>
main( )
{
   char line[MAXLINE], *change ( );
   void read_in(char );
   printf("\nWha-t is your read_in (line);
```

```
    printf( "\n%s\n\n%s\n\n",
"Here it is after being changed: change ( line) );
}
```

After prompting the user, this program uses read_in( ) to put characters into line. Then line is passed as an argument to change( ), which returns a pointer to char. The returned pointer value is printed by printf( ) in the format of a string. Here is the function read_in( ).

```
void read_in(s)
char s [ ];
{
   int c, i = 0;
   while ((c = getchar( )) != EOF && c ! '\n')
         s[i++] = c;
    s[i] = '\0';
 }
```

The parameter s is of type pointer to char. It could just as well have been declared as

```
char *s;
```

In the while loop successive characters are obtained from the input stream and placed one after another into the array with base address s. When a newline character is received, the loop is exited and a null character is put into the array to act as the end-of-string sentinel. Notice that this function allocates no space. In main( ) space is allocated with the declaration of line. You are making the assumption that the user will not type in more than MAXLINE - 1 characters. When line is passed as an argument to read_in( ), a copy of the base address of the array is made, and this value is taken on by the parameter s. The array elements themselves are not copied, but they are accessible in read_in( ) through this base address.

```
char *change(s);
char *s;
{
   static char new_string[MAXLINE];
   char *p = new_string;
   *p++ = '\t';
   for  ( ; *s != '\0'; ++s)
      if (*s == 'e')
         *p++ = 'E';
      else
          if (*s == ' ') {
               *p++ = '\n';
```

```
                *p++ = '\t';
            }
            else
                *p++ = *s
        *p = '\0';
         return (new_string);
    }
```

This function takes a string and copies it, changing every e to E and replacing every blank by a newline and a tab. Suppose you run the program and type in the line

she sells sea shells

after receiving the prompt. Here is what appears on the screen:

What is your favorite line? she sells sea shells

Here it is after being changed:

shE

sElls

sEa

shElls

ANALYSIS OF THE change( ) FUNCTION

The change( ) function will be explained in detail to understand how the it works.

```
char *change(s)
        char *s;
        {

           static char new_string[MAXLINE];

           char *p == new_string:
```

The first char * tells the compiler that this function returns a value of type pointer to char. The parameter s and the local variable p are both declared to be of type

pointer to char. Since s is a parameter in a function header, it could just as well have been declared as

```
char s[  ];
```

However, since p is a local variable and not a parameter, a similar declaration for p would be wrong. The array new_string is declared to have static storage class, and space is allocated for MAXLINE characters. The reason for static rather than automatic is explained below. The pointer p is initialized to the base address of new_string.

```
*p++= '\t';
```

This one line of code is equivalent to

```
*p = '\t';

p++;
```

The expression is analyzed as follows. Since the operators * and ++ are both unary and associate "right-to-left," the expression *p++ is equivalent to *(p++). Thus the ++ operator is causing p to be indexed. In contrast, the expression (*p) ++ would cause the value of what is pointed to by p to be indexed, which is something quite different. Since the ++ operator occurs on the right side of p rather than the left, the indexing of p occurs after the total expression *p++ = '\t' has been evaluated. Assignment is part of the evaluation process, and this causes a tab character to be assigned to what is pointed to by p. Since p points to the base address of new_string, a tab character is assigned to new_string[0]. After the indexing of p occurs, p points to new_string [1].

```
for ( ; *s !  = '\0'; ++s)
```

Each time through the for loop, a test is made to see if the value of what is pointed to by s is the end-of-string sentinel. If not. then the body of the for loop is executed and s is indexed. The effect of indexing a pointer to char is to cause it to point at the next character in the string.

```
if (*s  =  'e')

    *p++= 'E';
```

In the body of the for loop a test is made to see if s is pointing to the character e. If it is, then the character E is assigned to what p is pointing at, and then p is indexed.

```
else if (*s == ' ') {
    *p++= '\n';
    *p++ = '\t',
}
```

Otherwise, a test is made to see if s is pointing to a blank character. If it is, then a newline character is assigned to what p is pointing at, followed by the indexing of p, followed by the assignment of a tab character to what p is pointing at, followed by the indexing of p.

```
else

    *p++ = *s;
```

Finally, if the character to which s is pointing is neither an e or a blank, then what p is pointing at is assigned the value of what s is pointing at, followed by the indexing of p. The effect of this for loop is to copy the string passed as an argument to change( ) into the string with base address &new_string[1], except that each e is replaced by an E and each blank is replaced by a newline and a tab.

```
*p = '\0';
```

When the for loop is exited, what p is pointing at is assigned an end-of-string sentinel.

```
return (new_string);
```

The array name new_string is a pointer to char, and this value is returned. If the storage class for new_string were automatic instead of static, then the memory allocated to new_string would not need to be preserved on exit from change( ). If the memory is overwritten, then the final printf( ) statement in main( ) will not work properly.

Pointers and Pointer Arithmetic Example

The example in this section illustrated the use of pointers and pointer arithmetic to process a string. A function will be written that counts the number of words in a string. For the purposes of this function a maximal sequence of nonwhite space characters will constitute a word.

```
/* Count the number of words in a string. */
#include <ctype.h>
word_cnt(s)
```

```
char *s;

{
   int cnt = 0;
   while (*s ! = '\0') {
      while (isspace(*s))             /* skip white space */
           ++s;
           if (*s !=  '\0')  {          /* found a word */
              + + cnt;
               while (!isspace(*s)  &&  *s  !=  '\0') /* skip the word */
                    ++s;
            }/* end for loop  */
   }/* end while loop */
 return (cnt);
}
```

This is a typical string processing function. Pointer arithmetic and dereferencing are used to search for various patterns or characters.

ARGUMENTS TO MAIN

C provides for arrays of any type, including arrays of pointers. Although this is an advanced topic and will not be treated in detail, you need to use arrays of pointers to char to write programs that use command line arguments. Two arguments, conventionally called argc and argv, can be used with main( ) to communicate with the operating system. Here is a program that prints its command line arguments. It is a variant of the echo command in UNIX.

```
/* Echo the command line arguments. */
main(argc, argv)
int argc;
char *argv[];
{
   int i;
   printf("argc = %d\n", argc);
   for ( i = 0; i < argc; ++i)
      printf("argv[%d] = %s\n", i, argv[ i ] ) );
}
```

The variable argc provides a count of the command line arguments. The array argv is an array of pointers to char, and can be thought of as an array of strings. Since the element argv[0] always contains the name of the command itself, the value of argc is always 1 or more. Suppose that the above program is in the file myecho.c. If we compile the program with the command

```
cc   -o   myecho   myecho.c
```

and then give the command myecho, the following is printed on the screen:

```
argc = 1
argv[0] = myecho
```

Now suppose that you give the command:

myecho try this

Here is what appears on the screen:

```
argc = 3

argv[0] = myecho

argv[1] = try

argv[2] = this
```

Finally, suppose that you give the command:

```
myecho big sky country
```

The following is printed on the screen:

```
argc = 4

argv[0] = myecho

argv[1] = big

argv[2] = sky

argv[3] = country
```

The parameter argv could just as well have been declared

```
char **argv;
```

It is a pointer to pointer to char that can be thought of as an array of pointers to char, which in turn can be thought of as an array of strings. Notice that there was no allocated space for the strings. The C compiler and the operating system do this for this type declaration and passes information to main( ) via the two arguments argc and argv.

STANDARD STRING FUNCTIONS

The standard library contains many useful string handling functions. Although these functions are not part of the C language, they are available on most systems. There is nothing special about these functions. They are all written in C and are quite short. Variables in them are often declared with storage class register in an attempt to make them execute more quickly. They all require that strings passed as arguments be null terminated, and they all return either an int or a pointer to char. The following table describes some of the available functions. The reader should consult a manual to learn about others.

Some string handling functions in the standard library

strlen(s);

char *s;

A count of the number of characters that occur before \0 is returned.

strcmp(s1 ,s2);

char *s1, *s2;

An integer is returned that is less than, equal to, or greater than zero depending on whether s1 is lexicographically less than, equal to, or greater than s2.

strncmp(s1, s2, n);

char *s1, *s2;

Similar to strcmp( ) except that at most n characters are compared.

char *strcat(s1, s2)

char *s1, *s2:

The two strings s1 and s2 are concatenated and the resulting string is placed in s1. The pointer value s1 is returned. The programmer must ensure that enough memory has been allocated so that s1 can hold the result.

char *strcpy(s1, s2);

char *s1, *s2;

The string s2 is copied into memory beginning at the base address pointed to by s1. Whatever might have been in the string s1 is lost. The pointer value s1 is returned. The programmer is responsible for ensuring that array bounds are not overrun.

```
char *strchr(s, c);

char *s, c;
```

A pointer to the first occurrence of c in the string s is returned, or NULL is returned if c is not in the string.

To demonstrate that there is nothing special about these functions, lconsider strlen( ). Here is one way the function strlen( ) could be written:

```
strlen(s)

register char *s;
{
   register int n;
     for( n = 0;   *s  != '\0';   ++s)
         ++n;
   return (n);
}
```

Before making use of string functions in the standard library, the programmer must property declare the functions in the program. Even though the code for the functions is provided by the system, the compiler needs to know the type of the value returned by these functions. The programmer can use the control line:

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

to include the header file <string.h>. This header file contains all the declarations needed to use the string functions in the standard library.

There are two styles of programming that can be used to process strings. Namely, one can use array notation with subscripts, or one can use pointers and pointer arithmetic. Although both styles are common, there is a tendency for experienced programmers to favor the use of pointers. In some C systems the pointer versions may execute faster.

Since the null character is always used to delimit a string, it is a common programming style to explicitly test for \0 when processing a string. However, it is not necessary to do so. The alternative is to use the length of the string. As an example of this:

```
n  =  strlen(s);

for( i =0;  i < n;   ++i)

if (islower( s[ i ] )

s[ i ] = toupper( s[ i ] );
```

to capitalize all the letters in the string s. This style of string processing is certainly acceptable. Notice, however, that a for loop of the form:

```
for ( i = 0;  i <= strien(s); ++i )
```

is inefficient. This code causes the length of s to be recomputed every time through the loop.

The Complete Set of STRING FUNCTIONS

```
        // Using strcpy and strncpy
#include <stdio.h>
#include <string.h>

        main()
        {
          char x[] = "Happy Birthday to You";
          char y[25], z[15];


           printf("The string in array x is: %s\n",x);
           printf("The string in array y is: %s\n",strcpy(y, x));


           strncpy(z, x, 14);
           z[14] = '\0';
           printf("The string in array z is: %s\n", z);
           return 0;

        }

/*
The string in array x is: Happy Birthday to You
The string in array y is: Happy Birthday to You
The string in array z is: Happy Birthday
```

```
*/


        // Using strcat and strncat
#include <stdio.h>
#include <string.h>

        main()
        {
          char s1[20] = "Happy ";
          char s2[] = "New Year ";
          char s3[40] = "";
          printf("s1 = %s\ns2 = %s\n",s1,s2);
         printf("strcat(s1, s2) = %s\n",strcat(s1, s2));
         printf("strncat(s3, s1, 6) = %s\n",strncat(s3, s1, 6));
         printf("strcat(s3, s1) = %s\n",strcat(s3, s1));
          return 0;
        }

/*
s1 = Happy
s2 = New Year
strcat(s1, s2) = Happy New Year
strncat(s3, s1, 6) = Happy
strcat(s3, s1) = Happy Happy New Year
*/




        // Using strcmp and strncmp
#include <string.h>
#include <stdio.h>
        main()
        {
          char *s1 = "Happy New Year";
          char *s2 = "Happy New Year";
          char *s3 = "Happy Holidays";


            printf("\ns1 = %s\ns2 = %s\ns3 = ",s1,s2,s3);
            printf("strcmp(s1, s2) = %d\n",strcmp(s1, s2));
            printf("strcmp(s1, s3) = %d\n",strcmp(s1, s3));
            printf("strcmp(s3, s1) = %d\n\n",strcmp(s3, s1));
```

```
        printf("strncmp(s1, s3, 6) = %d\n",strncmp(s1, s3, 6));
        printf("strncmp(s1, s3, 6) = %d\n",strncmp(s1, s3, 7));
        printf("strncmp(s3, s1, 7) = %d\n",strncmp(s3, s1, 7));

      return 0;
    }
    /*
    s1 = Happy New Year
    s2 = Happy New Year
    s3 = strcmp(s1, s2) = 0
    strcmp(s1, s3) = 1
    strcmp(s3, s1) = -1

    strncmp(s1, s3, 6) = 0
    strncmp(s1, s3, 6) = 1
    strncmp(s3, s1, 7) = -1
    */



// Using strtok
#include <iostream.h>
#include <string.h>
/*Multiple calls to strtok are required to break a string into tokens.
  The first call to strtok contains two arguments, a string to be
  tokenized, and a string containing characters that separate the tokens.
        tokenPtr = strtok(string, " ")
  assigns tokenPtr a pointer to the first token in string.  The second
  argument of  strtok,  " ", indicates that tokens in string are separated
  by spaces.  Function strtok searches for the first character in string
  that is not a delimiting character (space).  This begins the first token.
  The function then finds the next delimiting character in the string and r
  replaces it wiht the null ('\0') character.  This terminates the current
  token.  Function strtok saves a pointer to the next characterr following
  the token in string, and returns a pointer to the current token.

  Subsequent calls to strtok to continue tokenizing string contain NULL as
  the first argument.  The NULL argument indicates that the call to strtok
  should continue tokenizing from the location in string saved by the last
  call to strtok.  If no tokens remain when strtok is called, strtok return
  NULL.
*/


    main()
    {
```

```
   char string[] = "This is a sentence with 7 tokens";
   char *tokenPtr;
    printf("The string to be tokenized is:\n\n%s\n",string);
    printf("\n\nThe tokens are:\n");

    tokenPtr = strtok(string, " ");
     while (tokenPtr != NULL)
      {
       printf("%s\n",tokenPtr);
       tokenPtr = strtok(NULL, " ");
      }
  return 0;
}
/*
The string to be tokenized is:

This is a sentence with 7 tokens


The tokens are:
This
is
a
sentence
with
7
tokens
*/



// Using strlen
#include <stdio.h>
#include <string.h>

main()
{
  char *string1 = "abcdefghijklmnopqrstuvwxyz";
  char *string2 = "four";
  char *string3 = "Boston";
   printf("The length of \"%s\" is %d\n",string1,strlen(string1));
   printf("The length of \"%s\" is %d\n",string2,strlen(string2));
   printf("The length of \"%s\" is %d\n",string3,strlen(string3));
   return 0;
}
/*
```

```
        The length of "abcdefghijklmnopqrstuvwxyz" is 26
        The length of "four" is 4
        The length of "Boston" is 6
        */



//String Conversion Functions
#include <iostream.h>
#include <stdlib.h>  //General utilities library

//nPtr is a pointer to a character constant.
//const declares that the argument value will not be modified.

//      Function prototype          Function Description

//double atof(const char *nPtr)       Converts the string nPtr to double.
//int atof(const char *nPtr)          Converts the string nPtr to int.
//long atof(const char *nPtr)         Converts the string nPtr to long int.
//double strtod(const char *nPtr, char **endPtr)
//                                    Converts the string nPtr to double.
//long strtol(const char *nPtr, char **endPtr, int base)
//                                    Converts the string nPtr to long.
//                                    base is number base, octal, decimal etc.
//unsigned long strtoul(const char *nPtr, char **endPtr, int base)
//                                    Converts the string nPtr to unsigned long..

main()
{
// Using atof
  double d = atof("99.0");

  printf("The string \"99.0\" converted to double is %g\n",d);
  printf("The converted value divided by 2 is %g\n", d/2.0);
  printf("\n");
/*
The string "99.0" converted to double is 99
The converted value divided by 2 is 49.5
*/



// Using atoi
  int i = atoi("2593");
  printf("The string \"2593\" converted to int is %d\n",i);
```

```
   printf("The converted value minus 593 is %d\n", i - 593);
   printf("\n");
/*
The string "2593" converted to int is 2593
The converted value minus 593 is 2000
*/
```

```
// Using atol
   long l = atol("1000000");
   printf("The string \"1000000\" converted to long is %lu\n",l);
   printf("The converted value divided by 2 is  %lu\n", l/2);
   printf("\n");
/*
The string "1000000" converted to long is 1000000
The converted value divided by 2 is 500000
*/
```

```
// Using strtod
   char *string = "51.2% are admitted", *stringPtr;
   d = strtod(string, &stringPtr);
   printf("The string \"%s\" is converted to the\n",string);
   printf("double value %g and the string \"%s\"\n",d,stringPtr);
   printf("\n");
/*
The string "51.2% are admitted" is converted to the
double value 51.2 and the string "% are admitted"
*/
```

```
// Using strtol
   long x;
   char *remainderPtr;
   string = "-1234567abc";
   x = strtol(string, &remainderPtr, 0);
   printf("The original string is \"%s\"\n",string);
   printf("The converted value is %d\n",x);
   printf("The remainder of the original string is \"%s\"\n",
               remainderPtr);
   printf("The converted value plus 567 is %d\n",x+567);
   printf("\n");
/*
The original string is "-1234567abc"
The converted value is -1234567
The remainder of the original string is "abc"
The converted value plus 567 is -1234000
*/




// Using strtoul
   unsigned long xx;
   string = "1234567abc";
   xx = strtoul(string, &remainderPtr, 0);

   printf("The original string is \"%s\"\n",string);
   printf("The converted value is %lu\n",xx);
   printf("The remainder of the original string is \"%s\"\n",
               remainderPtr);
   printf("The converted value minus 567 is %lu\n",xx-567);
   printf("\n");

/*
The original string is "1234567abc"
The converted value is 1234567
The remainder of the original string is "abc"
The converted value minus 567 is 1234000
*/

  return 0;
}
```

```
// Using strchr
#include <stdio.h>
#include <string.h>
/*      char *strchr(const *s, int c);
Locates the first occurrence of character c in string s.  if
c is found, a pointer to c in s is returned.  Otherwise, a
NULL pointer is returned.
*/
main()
{
   char *string = "This is a test";
   char character1 = 'a', character2 = 'z';

   if (strchr(string, character1) != NULL)
      printf("\'%c\' was found in \"%s\".\n",character1,string);
   else
      printf("\'%c\' was not found in \"%s\".\n",character1,string);

   if (strchr(string, character2) != NULL)
      printf("\'%c\' was found in \"%s\".\n",character2,string);

   else
      printf("\'%c\' was not found in \"%s\".\n",character2,string);

   return 0;
}

/*  Output
'a' was found in "This is a test".
'z' was not found in "This is a test".
*/


// Using strcspn
#include <stdio.h>
#include <string.h>
/* size_t strcspn(const char *s1, const char *s2);
  Determines and returns the length of the initial segment
  of string s1 consisting of characters not contained in s2.
*/

main()
{
   char *string1 = "The value is 3.14159";
```

```
   char *string2 = "1234567890";

   printf("string1 = %s\n",string1);
   printf("string2 = %s\n\n",string2);
   printf("The length of the initial segment of string1\n");
   printf("containing no characters from string2 = %d\n",
            strcspn(string1, string2));
   return 0;
}
/*
string1 = The value is 3.14159
string2 = 1234567890

The length of the initial segment of string1
containing no characters from string2 = 13
*/


#include <stdio.h>
#include <string.h>
/* char *strpbrk(const char *s1, const char *s2);
  Locates the first occurrence in string s1 of any character
  in string s2.  If a character from string s2 is found, a pointer
  to the character in string s1 is returned.  Otherwise a NULL
  pointer is returned.
*/


main()
{
   char *string1 = "This is a test";
   char *string2 = "beware";
   printf("Of the characters in \"%s\"\n",string2);
   printf("'%c' is the first character to appear in\n%s\n",
          *strpbrk(string1,string2),string1);
   return 0;
}
/*
Of the characters in "beware"
'a' is the first character to appear in
"This is a test"
*/


// Using strrchr
#include <stdio.h>
```

```
#include <string.h>
/* char *strrchr(const *s, int c);
  Locates the last occurrence of c in string s.  If c is found,
  a pointer to c in string s is returned.  Otherwise, a NULL
  pointer is returned.
*/


main()
{
   char *string1 = "A zoo has many animals including zebras";
   int c = 'z';
   printf("The remainder of string1 beginning with the\n");
   printf("last occurrence of character '%c' is: %s\n",
              c , strrchr(string1, c));
   return 0;
}
/*
The remainder of string1 beginning with the
last occurrence of character 'z' is: "zebras"
*/

// Using strspn
#include <stdio.h>
#include <string.h>
/*  size_t strspn(const char *s1, const char *s2);
    Determines and returns the length of the initial segment of
    string s1 consisting only of characters contained in string s2.
*/

main()
{
   char *string1 = "The value is 3.14159";
   char *string2 = "aehilsTuv ";

   printf("string1 = %s\n",string1);
   printf("string2 = %s\n\n",string2);
   printf("The length of the initial segment of string1\n");
   printf("containing only characters from string2 = %d\n",
              strspn(string1, string2));
   return 0;
}
/*
string1 = The value is 3.14159
string2 = aehilsTuv
```

```
The length of the initial segment of string1
containing only characters from string2 = 13
*/


// Using strstr
#include <stdio.h>
#include <string.h>
/* char *strstr(const char *s1, const char *s2);
   Locates the first occurrence in string s1 of string s2.  If
   the string is found, a pointer to the string in s1 is
   returned.  Otherwise, a NULL pointer is returned.
*/

main()
{
   char *string1 = "abcdefabcdef";
   char *string2 = "def";
   printf("string1 = %s\n",string1);
   printf("string2 = %s\n\n",string2);
   printf("The remainder of string1 beginning with the\n");
   printf("first occurrence of string2 is: %s\n",
              strstr(string1, string2));
   return 0;
}

/*
string1 = abcdefabcdef
string2 = def

The remainder of string1 beginning with the
first occurrence of string2 is: defabcdef
*/


// Using memcpy
#include <stdio.h>
#include <string.h>

/*  void *memcpy(void *s1, const void *s2, size_t n);
    Copies n characters from the object pointed to by s2 into the
    object pointed to by s1.  A pointer to the resulting object is
    returned.
*/

main()
```

```
{
   char s1[17], s2[]  = "Copy this string";

   memcpy(s1, s2, 17);
   printf("After s2 is copied into s1 with memcpy,\n");
   printf("s1 contains \"%s\"\n",s1);
   return 0;
}
/*
After s2 is copied into s1 with memcpy,
s1 contains "Copy this string"
*/

// Using memmove
#include <stdio.h>
#include <string.h>
/* void *memmove(void *s1, const void *s2, size_t n);
   Copies n characters from the object pointed to by s2 into
   the object pointed to by s1.  The copy is performed as if
   the characters are first copied from the object pointed to
   by s2 into a temporary array, then from the temporary array
   into the object pointed to by s1.  A pointer to the
   resulting object is returned.
*/



main()
{
   char x[] = "Home Sweet Home";
   printf("The string in array x before memmove is: %s\n",x);
   printf("The string in array x after  memmove is: %s\n",
          (char *) memmove(x, &x[5], 10));

   return 0;
}
/*
The string in array x before memmove is: Home Sweet Home
The string in array x after memmove is:  Sweet Home Home
*/


// Using memcmp
#include <stdio.h>
#include <string.h>
/* int memcmp(const void *s1, const void *s2, size_t n);
```

```
   Compares the first n characters of the object pointed to by
   s1 and s2.  The function returns 0, less than 0, or greater
   than 0 if s1 is equalto, less than, or greater than s2.
*/
main()
{
   char s1[] = "ABCDEFG", s2[] = "ABCDXYZ";
   printf("s1 = %s\n",s1);
   printf("s2 = %s\n\n",s2);
   printf("memcmp(s1, s2, 4) = %d\n",memcmp(s1, s2, 4));
   printf("memcmp(s1, s2, 7) = %d\n",memcmp(s1, s2, 7));
   printf("memcmp(s2, s1, 7) = %d\n",memcmp(s2, s1, 7));
   return 0;

}
/*
s1 = ABCDEFG
s2 = ABCDXYZ

memcmp(s1, s2, 4) =   0
memcmp(s1, s2, 7) = -224
memcmp(s2, s1, 7) =  32
*/

// Using memchr
#include <stdio.h>
#include <string.h>

/*        void *memchr(const void *s, int c, size_t n);
   Locates the first occurrence of c (converted to unsigned char)
   in the first n characters of the object pointed to by s.  If c is
   found, a pointer ot c in the object is returned.
   Otherwise, NULL is returned.
*/
main()
{
   char *s = "This is a string";
   printf("The remainder of s after character 'r' is found is \"%s\"\n",
    (char *) memchr(s, 'r', 16));
//   cout << "The remainder of s after character 'r' is found is \""
//      << (char *) memchr(s, 'r', 16) << '\"' << endl;
   return 0;
}
/*
The remainder of s after character 'r' is found is "ring"
*/
```

```
// Using memset
#include <stdio.h>
#include <string.h>

/*     void *memset(void *s, int c, size_t n);
   Copies c (converted to unsigned char) into the first
   n characters of the object pointed to by s.
   A pointer to the result is returned.
*/


main()
{
   char string1[15] = "BBBBBBBBBBBBBB";
   printf("string1 = %s\n",string1);
   printf("string1 after memset = %s\n",(char *) memset(string1, 'b', 7));
   return 0;
}
/*
string1 = BBBBBBBBBBBBBB
string1 after memset = bbbbbbbBBBBBBB
*/
```