مبانی برنامه نویسی
به زبان سی

۷، ۹ و ۱۱ دی ۱۳۹۹

جلسه های ۲۳، ۲۴ و ۲۵

ملکی مجد

مباحث این هفته:

- اشاره گر به تابع
- رشته و کارکتر

7.12 Pointers to Functions

- A pointer to a function contains the address of the function in memory.
  - we saw that an array name is really the address in memory of the first element of the array.
  - Similarly, a function name is really the starting address in memory of the code that performs the function's task.

- Pointers to functions can **be passed to functions**, **returned from functions**, stored in arrays and **assigned to other function pointers**.

7.12 Pointers to Functions (Cont.)

- To illustrate the use of pointers to functions,

  consider a modified version of the bubble sort program
- The new version consists of `main` and functions `bubble`, `swap`, `ascending` and `descending`.

- Function `bubbleSort` receives a pointer to a function
  - either function `ascending` or
  - function `descending`
- as an argument, in addition to an integer array and the size of the array.

7.12 Pointers to Functions (Cont.)

- The program prompts the user to choose whether the array should be sorted in ascending or in descending order.
  - If the user enters `1`, a pointer to function `ascending` is passed to function `bubble`, causing the array to be sorted into increasing order.
  - If the user enters `2`, a pointer to function `descending` is passed to function `bubble`, causing the array to be sorted into decreasing order.

```c
/* Fig. 7.26: fig07_26.c
   Multipurpose sorting program using function pointers */
#include <stdio.h>
#define SIZE 10

/* prototypes */
void bubble( int work[], const int size, int (*compare)( int a, int b ) );
int ascending( int a, int b );
int descending( int a, int b );

int main( void )
{
   int order; /* 1 for ascending order or 2 for descending order */
   int counter; /* counter */

   /* initialize array a */
   int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };

   printf( "Enter 1 to sort in ascending order,\n"
           "Enter 2 to sort in descending order: " );
   scanf( "%d",  &order );

   printf( "\nData items in original order\n" );
```

Fig. 7.26 | Multipurpose sorting program using function pointers. (Part 1 of 4.)

```
24
25      /* output original array */
26      for ( counter = 0; counter < SIZE; counter++ ) {
27         printf( "%5d", a[ counter ] );
28      } /* end for */
29
30      /* sort array in ascending order; pass function ascending as an
31        argument to specify ascending sorting order */
32      if ( order == 1 ) {
33         bubble( a, SIZE, ascending );
34         printf( "\nData items in ascending order\n" );
35      } /* end if */
36      else { /* pass function descending */
37         bubble( a, SIZE, descending );
38         printf( "\nData items in descending order\n" );
39      } /* end else */
40
41      /* output sorted array */
42      for ( counter = 0; counter < SIZE; counter++ ) {
43         printf( "%5d", a[ counter ] );
44      } /* end for */
45
46      printf( "\n" );
47      return 0; /* indicates successful termination */
48   } /* end main */
```

Fig. 7.26 | Multipurpose sorting program using function pointers. (Part 2 of 4.)

```
49
50  /* multipurpose bubble sort; parameter compare is a pointer to
51     the comparison function that determines sorting order */
52  void bubble( int work[], const int size, int (*compare)( int a, int b ) )
53  {
54     int pass; /* pass counter */
55     int count; /* comparison counter */
56
57     void swap( int *element1Ptr, int *element2ptr ); /* prototype */
58
59     /* loop to control passes */
60     for ( pass = 1; pass < size; pass++ ) {
61
62        /* loop to control number of comparisons per pass */
63        for ( count = 0; count < size - 1; count++ ) {
64
65           /* if adjacent elements are out of order, swap them */
66           if ( (*compare)( work[ count ], work[ count + 1 ] ) ) {
67              swap( &work[ count ], &work[ count + 1 ] );
68           } /* end if */
69        } /* end for */
70     } /* end for */
71  } /* end function bubble */
72
```

Fig. 7.26 | Multipurpose sorting program using function pointers. (Part 3 of 4.)

```
73  /* swap values at memory locations to which element1Ptr and
74     element2Ptr point */
75  void swap( int *element1Ptr, int *element2Ptr )
76  {
77     int hold; /* temporary holding variable */
78
79     hold = *element1Ptr;
80     *element1Ptr = *element2Ptr;
81     *element2Ptr = hold;
82  } /* end function swap */
83
84  /* determine whether elements are out of order for an ascending
85     order sort */
86  int ascending( int a, int b )
87  {
88     return b < a; /* swap if b is less than a */
89  } /* end function ascending */
90
91  /* determine whether elements are out of order for a descending
92     order sort */
93  int descending( int a, int b )
94  {
95     return b > a; /* swap if b is greater than a */
96  } /* end function descending */
```

Fig. 7.26 | Multipurpose sorting program using function pointers. (Part 4 of 4.)

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 1

Data items in original order
    2    6    4    8   10   12   89   68   45   37
Data items in ascending order
    2    4    6    8   10   12   37   45   68   89
```

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 2

Data items in original order
    2    6    4    8   10   12   89   68   45   37
Data items in descending order
   89   68   45   37   12   10    8    6    4    2
```

Fig. 7.27 | The outputs of the bubble sort program in Fig. 7.26.

7.12 Pointers to Functions (Cont.)

- The following parameter appears in the function header for `bubble` (line 52)
  - `int (*compare)( int a, int b )`
  - This tells `bubble` to expect a parameter (`compare`) that is a pointer to a function that receives two integer parameters and returns an integer result.

- Parentheses are needed around `*compare` to group `*` with `compare` to indicate that `compare` is a pointer.
- If we had not included the parentheses, the declaration would have been
  - `int *compare( int a, int b )`

  which declares a function that receives two integers as parameters and returns a pointer to an integer.

7.12 Pointers to Functions (Cont.)

- The prototype for `bubble` could have been written as
  - `int (*)( int, int );`

  without the function-pointer name and parameter names.

- The function passed to `bubble` is called in an `if` statement (line 66) as follows:
  - `if ((*compare)( work[count], work[count + 1]))`
- Just as a pointer to a variable is dereferenced to access the value of the variable, a pointer to a function is dereferenced to use the function.

7.12 Pointers to Functions (Cont.)

- The call to the function **could have been made without dereferencing** the pointer as in
  - `if (compare(work[count], work[count + 1]))`

  which uses the pointer directly as the function name.
- calling a function through a pointer

  explicitly illustrates that `compare` is a pointer to a function that is dereferenced to call the function.

  - The second method of calling a function through a pointer makes it appear as though `compare` is an actual function.
  - This may be confusing to a user of the program who would like to see the definition of function `compare` and finds that it's never defined in the file.

7.12 Pointers to Functions (Cont.)

- A common use of function pointers is in text-based menu-driven systems.
- A user is prompted to select an option from a menu (possibly from 1 to 5) by typing the menu item's number.
- Each option is serviced by a different function.
- Pointers to each function are stored in an array of pointers to functions.
- The user's choice is used as a subscript in the array, and the pointer in the array is used to call the function.

```c
/* Fig. 7.28: fig07_28.c
   Demonstrating an array of pointers to functions */
#include <stdio.h>

/* prototypes */
void function1( int a );
void function2( int b );
void function3( int c );

int main( void )
{
   /* initialize array of 3 pointers to functions that each take an
      int argument and return void */
   void (*f[ 3 ])( int ) = { function1, function2, function3 };

   int choice; /* variable to hold user's choice */

   printf( "Enter a number between 0 and 2, 3 to end: " );
   scanf( "%d", &choice );

```

The definition is read beginning in the leftmost set of parentheses, "`f` is an array of `3` pointers to functions that each take an `int` as an argument and return `void`."

Fig. 7.28 | Demonstrating an array of pointers to functions. (Part 1 of 3.)

```
21      /* process user's choice */
22      while ( choice >= 0 && choice < 3 ) {
23
24         /* invoke function at location choice in array f and pass
25            choice as an argument */
26         (*f[ choice ])( choice );
27
28         printf( "Enter a number between 0 and 2, 3 to end: ");
29         scanf( "%d", &choice );
30      } /* end while */
31
32      printf( "Program execution completed.\n" );
33      return 0; /* indicates successful termination */
34   } /* end main */
35
36   void function1( int a )
37   {
38      printf( "You entered %d so function1 was called\n\n", a );
39   } /* end function1 */
40
41   void function2( int b )
42   {
43      printf( "You entered %d so function2 was called\n\n", b );
44   } /* end function2 */
```

Fig. 7.28 | Demonstrating an array of pointers to functions. (Part 2 of 3.)

```
45
46  void function3( int c )
47  {
48     printf( "You entered %d so function3 was called\n\n", c );
49  } /* end function3 */
```

```
Enter a number between 0 and 2, 3 to end: 0
You entered 0 so function1 was called

Enter a number between 0 and 2, 3 to end: 1
You entered 1 so function2 was called

Enter a number between 0 and 2, 3 to end: 2
You entered 2 so function3 was called

Enter a number between 0 and 2, 3 to end: 3
Program execution completed.
```

Fig. 7.28 | Demonstrating an array of pointers to functions. (Part 3 of 3.)

8.2 Fundamentals of Strings and Characters

- Character constants.
  - A character constant is an `int` value represented as a character in single quotes.
  - The value of a character constant is the integer value of the character in the machine's character set.
  - For example, `'z'` represents the integer value of `z`, and `'\n'` the integer value of newline (122 and 10 in ASCII, respectively).

8.2 Fundamentals of Strings and Characters (Cont.)

- A string is a series of characters treated as a single unit.
  - A string may include letters, digits and various special characters such as +, -, *, / and $.
  - String literals, or string constants, in C are written in double quotation marks.
  - A string in C is an array of characters ending in the null character (`'\0'`).
  - A string is accessed via a pointer to the first character in the string.
  - The value of a string is the address of its first character.
  - Thus, in C, it is appropriate to say that a string is a pointer—in fact, a pointer to the string's first character. In this sense, strings are like arrays, because an array is also a pointer to its first element.
  - A character array or a variable of type `char *` can be initialized with a string in a definition.

8.2 Fundamentals of Strings and Characters (Cont.)

- The definitions
  - ```
    char color[] = "blue";
    const char *colorPtr = "blue";
    ```

  each initialize a variable to the string `"blue"`.
  - The first definition creates a 5-element array `color` containing the characters `'b'`, `'l'`, `'u'`, `'e'` and `'\0'`.
  - The second definition creates pointer variable `colorPtr` that points to the string `"blue"` somewhere in memory.
    ```

Portability Tip 8.1

When a variable of type `char *` *is initialized with a string literal, some compilers may place the string in a location in memory where the string cannot be modified. If you might need to modify a string literal, it should be stored in a character array to ensure modifiability on all systems.*

8.2 Fundamentals of Strings and Characters (Cont.)

- The preceding array definition could also have been written
  - `char color[] = { 'b', 'l', 'u', 'e', '\0' };`
- When defining a character array to contain a string, the array must be large enough to store the string and its terminating null character.
- The preceding definition automatically determines the size of the array based on the number of initializers in the initializer list.

Common Programming Error 8.2

Printing a "string" that does not contain a terminating null character is an error.

Error-Prevention Tip 8.1

When storing a string of characters in a character array, be sure that the array is large enough to hold the largest string that will be stored. C allows strings of any length to be stored. If a string is longer than the character array in which it is to be stored, characters beyond the end of the array will overwrite data in memory following the array.

8.2 Fundamentals of Strings and Characters (Cont.)

- A string can be stored in an array using `scanf`.
- For example, the following statement stores a string in character array `word[20]`:
  - `scanf( "%s", word );`
  - Variable `word` is an array, which is, of course, a pointer, so the `&` is not needed with argument `word`.
  - `scanf` will read characters until a space, tab, newline or end-of-file indicator is encountered.
  - So, it is possible that the user input could exceed 19 characters and that your program might crash!
  - For this reason, use the conversion specifier `%19s` so that `scanf` reads up to 19 characters and saves the last character for the terminating null character.

8.3 Character-Handling Library

- The character-handling library (`<ctype.h>`) includes several functions that perform useful tests and manipulations of character data.
  - Each function receives a character—represented as an `int`—or `EOF` as an argument.
  - `EOF` normally has the value –1

Prototype	Function description
`int isdigit( int c );`	Returns a true value if `c` is a digit and 0 (false) otherwise.
`int isalpha( int c );`	Returns a true value if `c` is a letter and 0 otherwise.
`int isalnum( int c );`	Returns a true value if `c` is a digit or a letter and 0 otherwise.
`int isxdigit( int c );`	Returns a true value if `c` is a hexadecimal digit character and 0 otherwise. (See Appendix C, Number Systems, for a detailed explanation of binary numbers, octal numbers, decimal numbers and hexadecimal numbers.)
`int islower( int c );`	Returns a true value if `c` is a lowercase letter and 0 otherwise.
`int isupper( int c );`	Returns a true value if `c` is an uppercase letter and 0 otherwise.
`int tolower( int c );`	If `c` is an uppercase letter, `tolower` returns `c` as a lowercase letter. Otherwise, `tolower` returns the argument unchanged.
`int toupper( int c );`	If `c` is a lowercase letter, `toupper` returns `c` as an uppercase letter. Otherwise, `toupper` returns the argument unchanged.
`int isspace( int c );`	Returns a true value if `c` is a white-space character—newline (`'\n'`), space (`' '`), form feed (`'\f'`), carriage return (`'\r'`), horizontal tab (`'\t'`) or vertical tab (`'\v'`)—and 0 otherwise.

Fig. 8.1 | Character-handling library (`<ctype.h>`) functions. (Part 1 of 2.)

Prototype	Function description
`int iscntrl( int c );`	Returns a true value if `c` is a control character and 0 otherwise.
`int ispunct( int c );`	Returns a true value if `c` is a printing character other than a space, a digit, or a letter and returns 0 otherwise.
`int isprint( int c );`	Returns a true value if `c` is a printing character including a space (`' '`) and returns 0 otherwise.
`int isgraph( int c );`	Returns a true value if `c` is a printing character other than a space (`' '`) and returns 0 otherwise.

Fig. 8.1 | Character-handling library (`<ctype.h>`) functions. (Part 2 of 2.)

```c
/* Fig. 8.2: fig08_02.c
   Using functions isdigit, isalpha, isalnum, and isxdigit */
#include <stdio.h>
#include <ctype.h>

int main( void )
{
   printf( "%s\n%s%s\n%s%s\n\n", "According to isdigit: ",
      isdigit( '8' ) ? "8 is a " : "8 is not a ", "digit",
      isdigit( '#' ) ? "# is a " : "# is not a ", "digit" );

   printf( "%s\n%s%s\n%s%s\n%s%s\n%s%s\n\n",
      "According to isalpha:",
      isalpha( 'A' ) ? "A is a " : "A is not a ", "letter",
      isalpha( 'b' ) ? "b is a " : "b is not a ", "letter",
      isalpha( '&' ) ? "& is a " : "& is not a ", "letter",
      isalpha( '4' ) ? "4 is a " : "4 is not a ", "letter" );

```

Fig. 8.2 | Using `isdigit`, `isalpha`, `isalnum` and `isxdigit`. (Part 1 of 3.)

```
19      printf( "%s\n%s%s\n%s%s\n%s%s\n\n",
20         "According to isalnum:",
21         isalnum( 'A' ) ? "A is a " : "A is not a ",
22         "digit or a letter",
23         isalnum( '8' ) ? "8 is a " : "8 is not a ",
24         "digit or a letter",
25         isalnum( '#' ) ? "# is a " : "# is not a ",
26         "digit or a letter" );
27
28      printf( "%s\n%s%s\n%s%s\n%s%s\n%s%s\n%s%s\n",
29         "According to isxdigit:",
30         isxdigit( 'F' ) ? "F is a " : "F is not a ",
31         "hexadecimal digit",
32         isxdigit( 'J' ) ? "J is a " : "J is not a ",
33         "hexadecimal digit",
34         isxdigit( '7' ) ? "7 is a " : "7 is not a ",
35         "hexadecimal digit",
36         isxdigit( '$' ) ? "$ is a " : "$ is not a ",
37         "hexadecimal digit",
38         isxdigit( 'f' ) ? "f is a " : "f is not a ",
39         "hexadecimal digit" );
40      return 0; /* indicates successful termination */
41   } /* end main */
```

Fig. 8.2 | Using `isdigit`, `isalpha`, `isalnum` and `isxdigit`. (Part 2 of 3.)

```
According to isdigit:
8 is a digit
# is not a digit

According to isalpha:
A is a letter
b is a letter
& is not a letter
4 is not a letter

According to isalnum:
A is a digit or a letter
8 is a digit or a letter
# is not a digit or a letter

According to isxdigit:
F is a hexadecimal digit
J is not a hexadecimal digit
7 is a hexadecimal digit
$ is not a hexadecimal digit
f is a hexadecimal digit
```

Fig. 8.2 | Using `isdigit`, `isalpha`, `isalnum` and `isxdigit`. (Part 3 of 3.)

```c
/* Fig. 8.3: fig08_03.c
   Using functions islower, isupper, tolower, toupper */
#include <stdio.h>
#include <ctype.h>

int main( void )
{
   printf( "%s\n%s%s\n%s%s\n%s%s\n%s%s\n\n",
      "According to islower:",
      islower( 'p' ) ? "p is a " : "p is not a ",
      "lowercase letter",
      islower( 'P' ) ? "P is a " : "P is not a ",
      "lowercase letter",
      islower( '5' ) ? "5 is a " : "5 is not a ",
      "lowercase letter",
      islower( '!' ) ? "! is a " : "! is not a ",
      "lowercase letter" );

```

Fig. 8.3 | Using functions `islower`, `isupper`, `tolower` and `toupper`. (Part 1 of 3.)

```
19      printf( "%s\n%s%s\n%s%s\n%s%s\n%s%s\n\n",
20         "According to isupper:",
21         isupper( 'D' ) ? "D is an " : "D is not an ",
22         "uppercase letter",
23         isupper( 'd' ) ? "d is an " : "d is not an ",
24         "uppercase letter",
25         isupper( '8' ) ? "8 is an " : "8 is not an ",
26         "uppercase letter",
27         isupper( '$' ) ? "$ is an " : "$ is not an ",
28         "uppercase letter" );
29
30      printf( "%s%c\n%s%c\n%s%c\n%s%c\n",
31         "u converted to uppercase is ", toupper( 'u' ),
32         "7 converted to uppercase is ", toupper( '7' ),
33         "$ converted to uppercase is ", toupper( '$' ),
34         "L converted to lowercase is ", tolower( 'L' ) );
35      return 0; /* indicates successful termination */
36   } /* end main */
```

Fig. 8.3 | Using functions `islower`, `isupper`, `tolower` and `toupper`. (Part 2 of 3.)

```
According to islower:
p is a lowercase letter
P is not a lowercase letter
5 is not a lowercase letter
! is not a lowercase letter

According to isupper:
D is an uppercase letter
d is not an uppercase letter
8 is not an uppercase letter
$ is not an uppercase letter

u converted to uppercase is U
7 converted to uppercase is 7
$ converted to uppercase is $
L converted to lowercase is l
```

Fig. 8.3 | Using functions `islower`, `isupper`, `tolower` and `toupper`. (Part 3 of 3.)

```c
/* Fig. 8.4: fig08_04.c
   Using functions isspace, iscntrl, ispunct, isprint, isgraph */
#include <stdio.h>
#include <ctype.h>

int main( void )
{
   printf( "%s\n%s%s%s\n%s%s%s\n%s%s\n\n",
      "According to isspace:",
      "Newline", isspace( '\n' ) ? " is a " : " is not a ",
      "whitespace character", "Horizontal tab",
      isspace( '\t' ) ? " is a " : " is not a ",
      "whitespace character",
      isspace( '%' ) ? "% is a " : "% is not a ",
      "whitespace character" );

   printf( "%s\n%s%s%s\n%s%s\n\n", "According to iscntrl:",
      "Newline", iscntrl( '\n' ) ? " is a " : " is not a ",
      "control character", iscntrl( '$' ) ? "$ is a " :
      "$ is not a ", "control character" );

```

Fig. 8.4 | Using `isspace`, `iscntrl`, `ispunct`, `isprint` and `isgraph`. (Part 1 of 3.)

```
22      printf( "%s\n%s%s\n%s%s\n%s%s\n\n",
23          "According to ispunct:",
24          ispunct( ';' ) ? "; is a " : "; is not a ",
25          "punctuation character",
26          ispunct( 'Y' ) ? "Y is a " : "Y is not a ",
27          "punctuation character",
28          ispunct( '#' ) ? "# is a " : "# is not a ",
29          "punctuation character" );
30
31      printf( "%s\n%s%s\n%s%s%s\n\n", "According to isprint:",
32          isprint( '$' ) ? "$ is a " : "$ is not a ",
33          "printing character",
34          "Alert", isprint( '\a' ) ? " is a " : " is not a ",
35          "printing character" );
36
37      printf( "%s\n%s%s\n%s%s%s\n",  "According to isgraph:",
38          isgraph( 'Q' ) ? "Q is a " : "Q is not a ",
39          "printing character other than a space",
40          "Space", isgraph( ' ' ) ? " is a " : " is not a ",
41          "printing character other than a space" );
42      return 0; /* indicates successful termination */
43  } /* end main */
```

Fig. 8.4 | Using `isspace`, `iscntrl`, `ispunct`, `isprint` and `isgraph`. (Part 2 of 3.)

```
According to isspace:
Newline is a whitespace character
Horizontal tab is a whitespace character
% is not a whitespace character

According to iscntrl:
Newline is a control character
$ is not a control character

According to ispunct:
; is a punctuation character
Y is not a punctuation character
# is a punctuation character

According to isprint:
$ is a printing character
Alert is not a printing character

According to isgraph:
Q is a printing character other than a space
Space is not a printing character other than a space
```

Fig. 8.4 | Using `isspace`, `iscntrl`, `ispunct`, `isprint` and `isgraph`. (Part 3 of 3.)

8.4 String-Conversion Functions

- This section presents the string-conversion functions from the general utilities library (`<stdlib.h>`).
- These functions convert strings of digits to integer and floating-point values.
- Figure 8.5 summarizes the string-conversion functions.
- Note the use of `const` to declare variable `nPtr` in the function headers (read from right to left as "`nPtr` is a pointer to a character constant"); `const` specifies that the argument value will not be modified.

Function prototype	Function description
`double atof( const char *nPtr );`	Converts the string `nPtr` to `double`.
`int atoi( const char *nPtr );`	Converts the string `nPtr` to `int`.
`long atol( 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`.
`unsigned long strtoul( const char *nPtr, char **endPtr, int base );`	Converts the string `nPtr` to `unsigned long`.

Fig. 8.5 | String-conversion functions of the general utilities library.

8.4 String-Conversion Functions (Cont.)

- Function `atof` (Fig. 8.6) converts its argument—a string that represents a floating-point number—to a `double` value.
- The function returns the `double` value.
- If the converted value cannot be represented—for example, if the first character of the string is a letter—the behavior of function `atof` is undefined.

```c
/* Fig. 8.6: fig08_06.c
   Using atof */
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
   double d; /* variable to hold converted string */

   d = atof( "99.0" );

   printf( "%s%.3f\n%s%.3f\n",
      "The string \"99.0\" converted to double is ", d,
      "The converted value divided by 2 is ", d / 2.0 );
   return 0; /* indicates successful termination */
} /* end main */
```

```
The string "99.0" converted to double is 99.000
The converted value divided by 2 is 49.500
```

Fig. 8.6 | Using atof.

8.4 String-Conversion Functions (Cont.)

- Function `atoi` (Fig. 8.7) converts its argument—a string of digits that represents an integer—to an `int` value.
- The function returns the `int` value.
- If the converted value cannot be represented, the behavior of function `atoi` is undefined.

```c
/* Fig. 8.7: fig08_07.c
   Using atoi */
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
   int i; /* variable to hold converted string */

   i = atoi( "2593" );

   printf( "%s%d\n%s%d\n",
      "The string \"2593\" converted to int is ", i,
      "The converted value minus 593 is ", i - 593 );
   return 0; /* indicates successful termination */
} /* end main */
```

```
The string "2593" converted to int is 2593
The converted value minus 593 is 2000
```

Fig. 8.7 | Using atoi.

8.4 String-Conversion Functions (Cont.)

- Function `atol` (Fig. 8.8) converts its argument—a string of digits representing a long integer—to a `long` value.
- The function returns the `long` value.
- If the converted value cannot be represented, the behavior of function `atol` is undefined.
- If `int` and `long` are both stored in 4 bytes, function `atoi` and function `atol` work identically.

```c
/* Fig. 8.8: fig08_08.c
   Using atol */
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
   long l; /* variable to hold converted string */

   l = atol( "1000000" );

   printf( "%s%ld\n%s%ld\n",
      "The string \"1000000\" converted to long int is ", l,
      "The converted value divided by 2 is ", l / 2 );
   return 0; /* indicates successful termination */
} /* end main */
```

```
The string "1000000" converted to long int is 1000000
The converted value divided by 2 is 500000
```

Fig. 8.8 | Using `atol`.

8.4 String-Conversion Functions (Cont.)

- Function `strtod` (Fig. 8.9) converts a sequence of characters representing a floating-point value to `double`.
- The function receives two arguments—a string (`char *`) and a pointer to a string (`char **`).
- The string contains the character sequence to be converted.
- The pointer is assigned the location of the first character after the converted portion of the string. Line 14
  - `d = strtod( string, &stringPtr );`
- indicates that `d` is assigned the `double` value converted from `string`, and `stringPtr` is assigned the location of the first character after the converted value (`51.2`) in `string`.

```c
/* Fig. 8.9: fig08_09.c
   Using strtod */
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
   /* initialize string pointer */
   const char *string = "51.2% are admitted"; /* initialize string */

   double d; /* variable to hold converted sequence */
   char *stringPtr; /* create char pointer */

   d = strtod( string, &stringPtr );

   printf( "The string \"%s\" is converted to the\n", string );
   printf( "double value %.2f and the string \"%s\"\n", d, stringPtr );
   return 0; /* indicates successful termination */
} /* end main */
```

```
The string "51.2% are admitted" is converted to the
double value 51.20 and the string "% are admitted"
```

Fig. 8.9 | Using `strtod`.

8.4 String-Conversion Functions (Cont.)

- Function `strtol`
  - converts to `long` a sequence of characters representing an integer.
  - receives three arguments
    - a string (`char *`), a pointer to a string and an integer.
  - The string contains the character sequence to be converted.
  - The pointer is assigned the location of the first character after the converted portion of the string.
    - Using `NULL` for the second argument causes the remainder of the string to be ignored.
  - The integer specifies the base of the value being converted.
    - The third argument, `0`, indicates that the value to be converted can be in octal (base 8), decimal (base 10) or hexadecimal (base 16) format.

```c
/* Fig. 8.10: fig08_10.c
   Using strtol */
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
   const char *string = "-1234567abc"; /* initialize string pointer */

   char *remainderPtr; /* create char pointer */
   long x; /* variable to hold converted sequence */

   x = strtol( string, &remainderPtr, 0 );

   printf( "%s\"%s\"\n%s%ld\n%s\"%s\"\n%s%ld\n",
      "The original string is ", string,
      "The converted value is ", x,
      "The remainder of the original string is ",
      remainderPtr,
      "The converted value plus 567 is ", x + 567 );
   return 0; /* indicates successful termination */
} /* end main */
```

```
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
```

Fig. 8.10 | Using `strtol`. (Part 1 of 2.)

8.4 String-Conversion Functions (Cont.)

- Function `strtoul` (Fig. 8.11) converts to `unsigned long` characters representing an `unsigned long` integer.
  - The function works identically to function `strtol`.

```c
/* Fig. 8.11: fig08_11.c
   Using strtoul */
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
   const char *string = "1234567abc"; /* initialize string pointer */
   unsigned long x; /* variable to hold converted sequence */
   char *remainderPtr; /* create char pointer */

   x = strtoul( string, &remainderPtr, 0 );

   printf( "%s\"%s\"\n%s%lu\n%s\"%s\"\n%s%lu\n",
      "The original string is ", string,
      "The converted value is ", x,
      "The remainder of the original string is ",
      remainderPtr,
      "The converted value minus 567 is ", x - 567 );
   return 0; /* indicates successful termination */
} /* end main */
```

Fig. 8.11 | Using `strtoul`. (Part 1 of 2.)

```
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
```

8.5 Standard Input/Output Library Functions

- This section presents several functions from the standard input/output library (`<stdio.h>`) specifically for manipulating character and string data.
- .

Function prototype	Function description
`int getchar( void );`	Inputs the next character from the standard input and returns it as an integer.
`char *fgets( char *s, int n, FILE *stream);`	Inputs characters from the specified stream into the array `s` until a newline or end-of-file character is encountered, or until `n - 1` bytes are read. In this chapter, we specify the stream as `stdin`—the standard input stream, which is typically used to read characters from the keyboard. A terminating null character is appended to the array. Returns the string that was read into `s`.
`int putchar( int c );`	Prints the character stored in `c` and returns it as an integer.
`int puts( const char *s );`	Prints the string `s` followed by a newline character. Returns a non-zero integer if successful, or `EOF` if an error occurs.
`int sprintf( char *s, const char *format, ... );`	Equivalent to `printf`, except the output is stored in the array `s` instead of printed on the screen. Returns the number of characters written to `s`, or `EOF` if an error occurs.

Fig. 8.12 | Standard input/output library character and string functions. (Part 1 of 2.)

Function prototype	Function description
`int sscanf( char *s, const char *format, ... );`	Equivalent to `scanf`, except the input is read from the array `s` rather than from the keyboard. Returns the number of items successfully read by the function, or `EOF` if an error occurs.

Fig. 8.12 | Standard input/output library character and string functions. (Part 2 of 2.)

8.5 Standard Input/Output Library Functions (Cont.)

- Function `fgets`
  - reads characters from the standard input into its first argument—an array of `char`s—until a newline or the end-of-file indicator is encountered, or until the maximum number of characters is read.
  - The maximum number of characters is one fewer than the value specified in `fgets`'s second argument.
  - The third argument specifies the stream from which to read characters—in this case, we use the standard input stream (`stdin`).
  - A null character (`'\0'`) is appended to the array when reading terminates.

8.5 Standard Input/Output Library Functions (Cont.)

- Function `putchar`
  - prints its character argument.

```c
/* Fig. 8.13: fig08_13.c
   Using gets and putchar */
#include <stdio.h>

void reverse( const char * const sPtr ); /* prototype */

int main( void )
{
   char sentence[ 80 ]; /* create char array */

   printf( "Enter a line of text:\n" );

   /* use fgets to read line of text */
   fgets( sentence, 80, stdin );

   printf( "\nThe line printed backward is:\n" );
   reverse( sentence );
   return 0; /* indicates successful termination */
} /* end main */

```

Fig. 8.13 | Using `fgets` and `putchar`. (Part 1 of 3.)

```
21  /* recursively outputs characters in string in reverse order */
22  void reverse( const char * const sPtr )
23  {
24     /* if end of the string */
25     if ( sPtr[ 0 ] == '\0' ) { /* base case */
26        return;
27     } /* end if */
28     else { /* if not end of the string */
29        reverse( &sPtr[ 1 ] ); /* recursion step */
30        putchar( sPtr[ 0 ] ); /* use putchar to display character */
31     } /* end else */
32  } /* end function reverse */
```

```
Enter a line of text:
Characters and Strings

The line printed backward is:
sgnirtS dna sretcarahC
```

```
Enter a line of text:
able was I ere I saw elba

The line printed backward is:
able was I ere I saw elba
```

8.5 Standard Input/Output Library Functions (Cont.)

- Function `getchar`
  - reads a character from the standard input and returns the character as an integer.
- Function `puts`
  - takes a string (`char *`) as an argument and prints the string followed by a newline character.

```c
/* Fig. 8.14: fig08_14.c
   Using getchar and puts */
#include <stdio.h>

int main( void )
{
   char c; /* variable to hold character input by user */
   char sentence[ 80 ]; /* create char array */
   int i = 0; /* initialize counter i */

   /* prompt user to enter line of text */
   puts( "Enter a line of text:" );

   /* use getchar to read each character */
   while ( ( c = getchar() ) != '\n') {
      sentence[ i++ ] = c;
   } /* end while */

   sentence[ i ] = '\0'; /* terminate string */

   /* use puts to display sentence */
   puts( "\nThe line entered was:" );
   puts( sentence );
   return 0; /* indicates successful termination */
} /* end main */
```

```
Enter a line of text:
This is a test.

The line entered was:
This is a test.
```

Fig. 8.14 | Using getchar and puts. (Part 1 of 2.)

```c
/* Fig. 8.15: fig08_15.c
   Using sprintf */
#include <stdio.h>

int main( void )
{
   char s[ 80 ]; /* create char array */
   int x; /* x value to be input */
   double y; /* y value to be input */

   printf( "Enter an integer and a double:\n" );
   scanf( "%d%lf", &x, &y );

   sprintf( s, "integer:%6d\ndouble:%8.2f", x, y );

   printf( "%s\n%s\n",
           "The formatted output stored in array s is:", s );
   return 0; /* indicates successful termination */
} /* end main */
```

uses function `sprintf` to print formatted data into array `s`—an array of characters.

```
Enter an integer and a double:
298 87.375
The formatted output stored in array s is:
integer:   298
double:   87.38
```

```c
/* Fig. 8.16: fig08_16.c
   Using sscanf */
#include <stdio.h>

int main( void )
{
   char s[] = "31298 87.375"; /* initialize arra
   int x; /* x value to be input */
   double y; /* y value to be input */

   sscanf( s, "%d%lf", &x, &y );
   printf( "%s\n%s%6d\n%s%8.3f\n",
           "The values stored in character array s are:",
           "integer:", x, "double:", y );
   return 0; /* indicates successful termination */
} /* end main */
```

uses function `sscanf` to read formatted data from character array `s`.

```
The values stored in character array s are:
integer: 31298
double:  87.375
```

Fig. 8.16 | Using `sscanf`.

8.6 String-Manipulation Functions of the String-Handling Library

- The string-handling library (`<string.h>`) provides many useful functions for
  - manipulating string data (copying strings and concatenating strings),
  - comparing strings,
  - searching strings for characters and other strings,
  - tokenizing strings (separating strings into logical pieces) and
  - determining the length of strings.

- Every function—except for `strncpy`—appends the null character to its result.

Function prototype	Function description
`char *strcpy( char *s1, const char *s2 )`	Copies string `s2` into array `s1`. The value of `s1` is returned.
`char *strncpy( char *s1, const char *s2, size_t n )`	Copies at most `n` characters of string `s2` into array `s1`. The value of `s1` is returned.
`char *strcat( char *s1, const char *s2 )`	Appends string `s2` to array `s1`. The first character of `s2` overwrites the terminating null character of `s1`. The value of `s1` is returned.
`char *strncat( char *s1, const char *s2, size_t n )`	Appends at most `n` characters of string `s2` to array `s1`. The first character of `s2` overwrites the terminating null character of `s1`. The value of `s1` is returned.

Fig. 8.17 | String-manipulation functions of the string-handling library.

8.6 String-Manipulation Functions of the String-Handling Library (Cont.)

- Functions `strncpy` and `strncat` specify a parameter of type `size_t`, which is a type defined by the C standard as the integral type of the value returned by operator `sizeof`.

Portability Tip 8.2

Type `size_t` is a system-dependent synonym for either type `unsigned long` or type `unsigned int`.

8.6 String-Manipulation Functions of the String-Handling Library (Cont.)

- Function `strcpy` copies its second argument (a string) into its first argument (a character array that must be large enough to store the string and its terminating null character, which is also copied).
- Function `strncpy` is equivalent to `strcpy`, except that `strncpy` specifies the number of characters to be copied from the string into the array.
  - Function `strncpy` does not necessarily copy the terminating null character of its second argument.

8.6 String-Manipulation Functions of the String-Handling Library (Cont.)

- A terminating null character is written only if the number of characters to be copied is at least one more than the length of the string.
- For example, if `"test"` is the second argument, a terminating null character is written only if the third argument to `strncpy` is at least `5` (four characters in `"test"` plus a terminating null character).
  - If the third argument is larger than `5`, null characters are appended to the array until the total number of characters specified by the third argument are written.

```c
/* Fig. 8.18: fig08_18.c
   Using strcpy and strncpy */
#include <stdio.h>
#include <string.h>

int main( void )
{
   char x[] = "Happy Birthday to You"; /* initialize char array x */
   char y[ 25 ]; /* create char array y */
   char z[ 15 ]; /* create char array z */

   /* copy contents of x into y */
   printf( "%s%s\n%s%s\n",
      "The string in array x is: ", x,
      "The string in array y is: ", strcpy( y, x ) );

   /* copy first 14 characters of x into z. Does not copy null
      character */
   strncpy( z, x, 14 );

   z[ 14 ] = '\0'; /* terminate string in z */
   printf( "The string in array z is: %s\n", z );
   return 0; /* indicates successful termination */
} /* end main */
```

Fig. 8.18

```
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
```

8.6 String-Manipulation Functions of the String-Handling Library (Cont.)

- Function `strcat` appends its second argument (a string) to its first argument (a character array containing a string).
- The first character of the second argument replaces the null (`'\0'`) that terminates the string in the first argument.
  - You must ensure that the array used to store the first string is large enough to store the first string, the second string and the terminating null character copied from the second string.
- Function `strncat` appends a specified number of characters from the second string to the first string.
  - A terminating null character is appended to the result.

```c
/* Fig. 8.19: fig08_19.c
   Using strcat and strncat */
#include <stdio.h>
#include <string.h>

int main( void )
{
   char s1[ 20 ] = "Happy "; /* initialize char array s1 */
   char s2[] = "New Year "; /* initialize char array s2 */
   char s3[ 40 ] = ""; /* initialize char array s3 to empty */

   printf( "s1 = %s\ns2 = %s\n", s1, s2 );

   /* concatenate s2 to s1 */
   printf( "strcat( s1, s2 ) = %s\n", strcat( s1, s2 ) );

   /* concatenate first 6 characters of s1 to s3. Place '\0'
      after last character */
   printf( "strncat( s3, s1, 6 ) = %s\n", strncat( s3, s1, 6 ) );

   /* concatenate s1 to s3 */
   printf( "strcat( s3, s1 ) = %s\n", strcat( s3, s1 ) );
   return 0; /* indicates successful termination */
} /* end main */
```

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

Fig. 8.19 | Using `strcat` and `strncat`.

8.7 Comparison Functions of the String-Handling Library

- This section presents the string-handling library's string-comparison functions, `strcmp` and `strncmp`.

Function prototype	Function description
`int strcmp( const char *s1, const char *s2 );`	Compares the string `s1` with the string `s2`. The function returns `0`, less than `0` or greater than `0` if `s1` is equal to, less than or greater than `s2`, respectively.
`int strncmp( const char *s1, const char *s2, size_t n );`	Compares up to `n` characters of the string `s1` with the string `s2`. The function returns `0`, less than `0` or greater than `0` if `s1` is equal to, less than or greater than `s2`, respectively.

Fig. 8.20 | String-comparison functions of the string-handling library.

```c
/* Fig. 8.21: fig08_21.c
   Using strcmp and strncmp */
#include <stdio.h>
#include <string.h>

int main( void )
{
   const char *s1 = "Happy New Year"; /* initialize char pointer */
   const char *s2 = "Happy New Year"; /* initialize char pointer */
   const char *s3 = "Happy Holidays"; /* initialize char pointer */

   printf("%s%s\n%s%s\n%s%s\n\n%s%2d\n%s%2d\n%s%2d\n\n",
      "s1 = ", s1, "s2 = ", s2, "s3 = ", s3,
      "strcmp(s1, s2) = ", strcmp( s1, s2 ),
      "strcmp(s1, s3) = ", strcmp( s1, s3 ),
      "strcmp(s3, s1) = ", strcmp( s3, s1 ) );

   printf("%s%2d\n%s%2d\n%s%2d\n",
      "strncmp(s1, s3, 6) = ", strncmp( s1, s3, 6 ),
      "strncmp(s1, s3, 7) = ", strncmp( s1, s3, 7 ),
      "strncmp(s3, s1, 7) = ", strncmp( s3, s1, 7 ) );
   return 0; /* indicates successful termination */
} /* end main */
```

```
s1 = Happy New Year
s2 = Happy New Year
s3 = Happy Holidays

strcmp(s1, s2) =  0
strcmp(s1, s3) =  1
strcmp(s3, s1) = -1

strncmp(s1, s3, 6) =  0
strncmp(s1, s3, 7) =  6
strncmp(s3, s1, 7) = -6
```

Fig. 8.21 | Using `strcmp` and `strncmp`. (Part 1 of 2.)

8.8 Search Functions of the String-Handling Library

functions of the string-handling library used to search strings for characters and other strings

Function prototype and description
`char *strchr( const char *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.
`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 string `s2`.
`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`.
`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.
`char *strrchr( const char *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.

Fig. 8.22 | String-manipulation functions of the string-handling library. (Part 1 of 2.)

Function prototype and description
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.
char `*strtok( char *s1, const char *s2 );` A sequence of calls to `strtok` breaks string `s1` into "tokens"—logical pieces such as words in a line of text—separated by characters contained in string `s2`. The first call contains `s1` as the first argument, and subsequent calls to continue tokenizing the same string contain `NULL` as the first argument. A pointer to the current token is returned by each call. If there are no more tokens when the function is called, `NULL` is returned.

Fig. 8.22 | String-manipulation functions of the string-handling library. (Part 2 of 2.)

```c
/* Fig. 8.23: fig08_23.c
   Using strchr */
#include <stdio.h>
#include <string.h>

int main( void )
{
   const char *string = "This is a test"; /* initialize char pointer */
   char character1 = 'a'; /* initialize character1 */
   char character2 = 'z'; /* initialize character2 */

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

   /* if character2 was found in string */
   if ( strchr( string, character2 ) != NULL ) {
      printf( "\'%c\' was found in \"%s\".\n",
         character2, string );
   } /* end if */
   else { /* if character2 was not found */
      printf( "\'%c\' was not found in \"%s\".\n",
         character2, string );
   } /* end else */

   return 0; /* indicates successful termination */
} /* end main */
```

```
'a' was found in "This is a test".
'z' was not found in "This is a test".
```

8.8 Search Functions of the String-Handling Library (Cont.)

- Function `strcspn` determines the length of the initial part of the string in its first argument that does not contain any characters from the string in its second argument.
- The function returns the length of the segment.

```c
/* Fig. 8.24: fig08_24.c
   Using strcspn */
#include <stdio.h>
#include <string.h>

int main( void )
{
   /* initialize two char pointers */
   const char *string1 = "The value is 3.14159";
   const char *string2 = "1234567890";

   printf( "%s%s\n%s%s\n\n%s\n%s%u\n",
      "string1 = ", string1, "string2 = ", string2,
      "The length of the initial segment of string1",
      "containing no characters from string2 = ",
      strcspn( string1, string2 ) );
   return 0; /* indicates successful termination */
} /* end main */
```

Fig. 8.24 |

```
string1 = The value is 3.14159
string2 = 1234567890

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

Fig. 8.24 | Using `strcspn`. (Part 2 of 2.)

8.8 Search Functions of the String-Handling Library (Cont.)

- Function `strpbrk` searches its first string argument for the first occurrence of any character in its second string argument.
- If a character from the second argument is found, `strpbrk` returns a pointer to the character in the first argument; otherwise, `strpbrk` returns `NULL`.

```c
/* Fig. 8.25: fig08_25.c
   Using strpbrk */
#include <stdio.h>
#include <string.h>

int main( void )
{
   const char *string1 = "This is a test"; /* initialize char pointer */
   const char *string2 = "beware"; /* initialize char pointer */

   printf( "%s\"%s\"\n'%c'%s\n\"%s\"\n",
      "Of the characters in ", string2,
      *strpbrk( string1, string2 ),
      " appears earliest in ", string1 );
   return 0; /* indicates successful termination */
} /* end main */
```

```
Of the characters in "beware"
'a' appears earliest in
"This is a test"
```

Fig. 8.25 | Using `strpbrk`.

8.8 Search Functions of the String-Handling Library (Cont.)

- Function `strrchr` searches for the last occurrence of the specified character in a string.
- If the character is found, `strrchr` returns a pointer to the character in the string; otherwise, `strrchr` returns `NULL`.

```c
/* Fig. 8.26: fig08_26.c
   Using strrchr */
#include <stdio.h>
#include <string.h>

int main( void )
{
   /* initialize char pointer */
   const char *string1 = "A zoo has many animals including zebras";

   int c = 'z'; /* character to search for */

   printf( "%s\n%s'%c'%s\"%s\"\n",
      "The remainder of string1 beginning with the",
      "last occurrence of character ", c,
      " is: ", strrchr( string1, c ) );
   return 0; /* indicates successful termination */
} /* end main */
```

```
The remainder of string1 beginning with the
last occurrence of character 'z' is: "zebras"
```

Fig. 8.26 | Using `strrchr`.

8.8 Search Functions of the String-Handling Library (Cont.)

- Function `strspn` determines the length of the initial part of the string in its first argument that contains only characters from the string in its second argument.
- The function returns the length of the segment.

```c
/* Fig. 8.27: fig08_27.c
   Using strspn */
#include <stdio.h>
#include <string.h>

int main( void )
{
   /* initialize two char pointers */
   const char *string1 = "The value is 3.14159";
   const char *string2 = "aehi lsTuv";

   printf( "%s%s\n%s%s\n\n%s\n%s%u\n",
      "string1 = ", string1, "string2 = ", string2,
      "The length of the initial segment of string1",
      "containing only characters from string2 = ",
      strspn( string1, string2 ) );
   return 0; /* indicates successful termination */
} /* end main */
```

Fig. 8.27 | Using `strspn`.

```
string1 = The value is 3.14159
string2 = aehi lsTuv

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

Fig. 8.27 | Using `strspn`. (Part 2 of 2.)

8.8 Search Functions of the String-Handling Library (Cont.)

- Function `strstr` searches for the first occurrence of its second string argument in its first string argument.
- If the second string is found in the first string, a pointer to the location of the string in the first argument is returned.

```c
/* Fig. 8.28: fig08_28.c
   Using strstr */
#include <stdio.h>
#include <string.h>

int main( void )
{
   const char *string1 = "abcdefabcdef"; /* string to search */
   const char *string2 = "def"; /* string to search for */

   printf( "%s%s\n%s%s\n\n%s\n%s%s\n",
      "string1 = ", string1, "string2 = ", string2,
      "The remainder of string1 beginning with the",
      "first occurrence of string2 is: ",
      strstr( string1, string2 ) );
   return 0; /* indicates successful termination */
} /* end main */
```

```
string1 = abcdefabcdef
string2 = def

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

Fig. 8.28 | Using `strstr`.

8.8 Search Functions of the String-Handling Library (Cont.)

- Function `strtok` (Fig. 8.29) is used to break a string into a series of tokens.
- A token is a sequence of characters separated by delimiters
  - (usually spaces or punctuation marks, but a delimiter can be any character).
  - For example, in a line of text, each word can be considered a token, and the spaces and punctuation separating the words can be considered delimiters.

8.8 Search Functions of the String-Handling Library (Cont.)

- Multiple calls to `strtok` are required to tokenize a string
  - i.e., break it into tokens (assuming that the string contains more than one token).
- The first call to `strtok` contains two arguments: **a string to be tokenized**, and **a string containing characters that separate the tokens**.
  - ```
    /* begin tokenizing sentence */
    tokenPtr = strtok( string, " " );
    ```

  assigns `tokenPtr` a pointer to the first token in `string`.
    ```

8.8 Search Functions of the String-Handling Library (Cont.)

- Function `strtok` searches for the first character in `string` that is not a delimiting character (e.g., space).
  - This begins the first token.
- The function then finds the next delimiting character in the string and replaces it with a null (`'\0'`) character to terminate the current token.
- Function `strtok` saves a pointer to the next character following the token in `string` and returns a pointer to the current token.
- Subsequent `strtok` calls continue tokenizing `string`.
  - These calls contain `NULL` as their 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` returns `NULL`.
- Function `strtok` modifies the input string by placing `\0` at the end of each token; therefore, a copy of the string should be made if the string will be used again in the program after the calls to `strtok`.

```c
/* Fig. 8.29: fig08_29.c
   Using strtok */
#include <stdio.h>
#include <string.h>

int main( void )
{
   /* initialize array string */
   char string[] = "This is a sentence with 7 tokens";
   char *tokenPtr; /* create char pointer */

   printf( "%s\n%s\n\n%s\n",
      "The string to be tokenized is:", string,
      "The tokens are:" );

   tokenPtr = strtok( string, " " ); /* begin tokenizing sentence */

   /* continue tokenizing sentence until tokenPtr becomes NULL */
   while ( tokenPtr != NULL ) {
      printf( "%s\n", tokenPtr );
      tokenPtr = strtok( NULL, " " ); /* get next token */
   } /* end while */

   return 0; /* indicates successful termination */
} /* end main */
```

```
The string to be tokenized is:
This is a sentence with 7 tokens

The tokens are:
This
is
a
sentence
with
7
tokens
```

Fig. 8.29 | Using `strtok`. (Part 1 of 2.)

Fig. 8.29 | Using `strtok`. (Part 2 of 2.)

8.10 Other Functions of the String-Handling Library

- Function `strlen` takes a string as an argument and returns the number of characters in the string—the terminating null character is not included in the length.

```
size_t strlen( const char *s );
```

Determines the length of string `s`. The number of characters preceding the terminating null character is returned.

```c
/* Fig. 8.38: fig08_38.c
   Using strlen */
#include <stdio.h>
#include <string.h>

int main( void )
{
   /* initialize 3 char pointers */
   const char *string1 = "abcdefghijklmnopqrstuvwxyz";
   const char *string2 = "four";
   const char *string3 = "Boston";

   printf("%s\"%s\"%s%lu\n%s\"%s\"%s%lu\n%s\"%s\"%s%lu\n",
      "The length of ", string1, " is ",
      ( unsigned long ) strlen( string1 ),
      "The length of ", string2, " is ",
      ( unsigned long ) strlen( string2 ),
      "The length of ", string3, " is ",
      ( unsigned long ) strlen( string3 ) );
   return 0; /* indicates successful termination */
} /* end main */
```

Fig. 8.38 | Using `strlen`.

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

Fig. 8.38 | Using `strlen`. (Part 2 of 2.)

Structures

- Structures—sometimes referred to as aggregates—are collections of related variables under one name.
- Structures may contain variables of many different data types
  - in contrast to arrays that contain only elements of the same data type.
- Pointers and structures facilitate the formation of more complex data structures
  - such as linked lists, queues, stacks and trees.

10.2 Structure Definitions

- Structures are derived data types
  - they are constructed using objects of other types.
- Consider the following structure definition:

```
struct card {
   char *face;
   char *suit;
};
```

- Keyword `struct` introduces the structure definition.
  - The identifier `card` is the structure tag, which names the structure definition and is used with the keyword `struct` to declare variables of the structure type.

10.2 Structure Definitions (Cont.)

- In this example, the structure type is `struct card`.
- Variables declared within the braces of the structure definition are the structure's members.
- Members of the same structure type must have unique names, but two different structure types may contain members of the same name without conflict.
- Each structure definition must end with a semicolon.

10.2 Structure Definitions (Cont.)

- The definition of `struct card` contains members `face` and `suit` of type `char *`.
- Structure members can be variables of the primitive data types (e.g., `int`, `float`, etc.), or aggregates, such as arrays and other structures.
- Structure members can be of many types.

10.2 Structure Definitions (Cont.)

- For example, the following `struct` contains character array members for an employee's first and last names, an `int` member for the employee's age, a `char` member that would contain `'M'` or `'F'` for the employee's gender and a `double` member for the employee's hourly salary:,
  - ```
    struct employee {
       char firstName[ 20 ];
       char lastName[ 20 ];
       int age;
       char gender;
       double hourlySalary;
    };
    ```
    ```

10.2 Structure Definitions (Cont.)

- A structure cannot contain an instance of itself.
- For example, a variable of type `struct employee` cannot be declared in the definition for `struct employee`.
- A pointer to `struct employee`, however, may be included.
- For example,

```
struct employee2 {
   char firstName[ 20 ];
   char lastName[ 20 ];
   int age;
   char gender;
   double hourlySalary;
   struct employee2 person; /* ERROR */
   struct employee2 *ePtr; /* pointer */
};
```

- `struct employee2` contains an instance of itself (`person`), which is an error.

10.2 Structure Definitions (Cont.)

- Because `ePtr` is a pointer (to type `struct employee2`), it is permitted in the definition.
- A structure containing a member that is a pointer to the same structure type is referred to as a self-referential structure.
- Self-referential structures can be used to build linked data structures.

10.2 Structure Definitions (Cont.)

- Structure definitions do not reserve any space in memory; rather, each definition creates a new data type that is used to define variables.
- Structure variables are defined like variables of other types.
- The definition
  - `struct card aCard, deck[ 52 ], *cardPtr;`

  declares `aCard` to be a variable of type `struct card`, declares `deck` to be an array with 52 elements of type `struct card` and declares `cardPtr` to be a pointer to `struct card`.

10.2 Structure Definitions (Cont.)

- Variables of a given structure type may also be declared by placing a comma-separated list of the variable names between the closing brace of the structure definition and the semicolon that ends the structure definition.
- For example, the preceding definition could have been incorporated into the `struct card` structure definition as follows:

```
struct card {
   char *face;
   char *suit;
} aCard, deck[ 52 ], *cardPtr;
```

10.2 Structure Definitions (Cont.)

- The structure tag name is optional.
- If a structure definition does not contain a structure tag name, variables of the structure type may be declared only in the structure definition—not in a separate declaration.

10.2 Structure Definitions (Cont.)

- The only valid operations that may be performed on structures are:
  - assigning structure variables to structure variables of the same type,
  - taking the address (&) of a structure variable,
  - accessing the members of a structure variable and
  - using the `sizeof` operator to determine the size of a structure variable.

10.2 Structure Definitions (Cont.)

- Structures may **not be compared using operators ==** and **!=**, because structure members are not necessarily stored in consecutive bytes of memory.
- Sometimes there are "holes" in a structure, because computers may store specific data types only on certain memory boundaries such as half word, word or double word boundaries.

10.2 Structure Definitions (Cont.)

- ```
  struct example {
     char c;
     int i;
  } sample;
  ```
- has been assigned the character `'a'` and the integer `97` (the bit representations of the values are shown).
- A computer with 2-byte words may require that each member of `struct example` be aligned on a word boundary, i.e., at the beginning of a word (this is machine dependent).

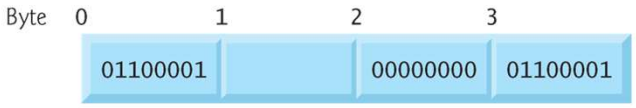

Fig. 10.1 | Possible storage alignment for a variable of type `struct` example showing an undefined area in memory.
  ```

10.3 Initializing Structures

- Structures can be initialized using initializer lists as with arrays.
- To initialize a structure, follow the variable name in the definition with an equals sign and a brace-enclosed, comma-separated list of initializers.
- For example, the declaration
  - `struct card aCard = { "Three", "Hearts" };`

  creates variable `aCard` to be of type `struct card` and initializes member `face` to `"Three"` and member `suit` to `"Hearts"`.

10.3 Initializing Structures (Cont.)

- If there are fewer initializers in the list than members in the structure, the remaining members are automatically initialized to 0 (or NULL if the member is a pointer).
- Structure variables defined outside a function definition (i.e., externally) are initialized to 0 or NULL if they are not explicitly initialized in the external definition.
- Structure variables may also be initialized in assignment statements by assigning a structure variable of the same type, or by assigning values to the individual members of the structure.

10.4 Accessing Structure Members

- Two operators are used to access members of structures:
  - the structure member operator (.)—also called the dot operator and
  - the structure pointer operator (->)—also called the arrow operator.
    - consisting of a minus (-) sign and a greater than (>) sign with no intervening spaces
- The structure member operator accesses a structure member via the structure variable name.

- For example, to print member `suit` of structure variable `aCard` defined, use the statement
  - `printf( "%s", aCard.suit ); /* displays Hearts */`

10.4 Accessing Structure Members (Cont.)

- Assume that the pointer `cardPtr` has been declared to point to `struct card` and that the address of structure `aCard` has been assigned to `cardPtr`.
- To print member `suit` of structure `aCard` with pointer `cardPtr`, use the statement
  - `printf( "%s", cardPtr->suit ); /* displays Hearts */`

10.4 Accessing Structure Members (Cont.)

- The expression `cardPtr->suit` is equivalent to `(*cardPtr).suit`, which dereferences the pointer and accesses the member `suit` using the structure member operator.
- The parentheses are needed here because the structure member operator (`.`) has a higher precedence than the pointer dereferencing operator (`*`).

```c
/* Fig. 10.2: fig10_02.c
   Using the structure member and
   structure pointer operators */
#include <stdio.h>

/* card structure definition */
struct card {
   char *face; /* define pointer face */
   char *suit; /* define pointer suit */
}; /* end structure card */

int main( void )
{
   struct card aCard; /* define one struct card variable */
   struct card *cardPtr; /* define a pointer to a struct card */

   /* place strings into aCard */
   aCard.face = "Ace";
   aCard.suit = "Spades";

   cardPtr = &aCard; /* assign address of aCard to cardPtr */

   printf( "%s%s%s\n%s%s%s\n%s%s%s\n", aCard.face, " of ", aCard.suit,
      cardPtr->face, " of ", cardPtr->suit,
      ( *cardPtr ).face, " of ", ( *cardPtr ).suit );
   return 0; /* indicates successful termination */
} /* end main */
```

```
Ace of Spades
Ace of Spades
Ace of Spades
```

10.5 Using Structures with Functions

- Structures may be passed to functions by passing individual structure members, by passing an entire structure or by passing a pointer to a structure.
- When structures or individual structure members are passed to a function, they are passed by value.
  - Therefore, the members of a caller's structure cannot be modified by the called function.
- To pass a structure by reference, pass the address of the structure variable.

10.5 Using Structures with Functions (Cont.)

- Arrays of structures—like all other arrays—are automatically passed by reference.
- **To pass an array by value, create a structure with the array as a member.**
- Structures are passed by value, so the array is passed by value.

Common Programming Error 10.6

Assuming that structures, like arrays, are automatically passed by reference and trying to modify the caller's structure values in the called function is a logic error.

Performance Tip 10.1

Passing structures by reference is more efficient than passing structures by value (which requires the entire structure to be copied).