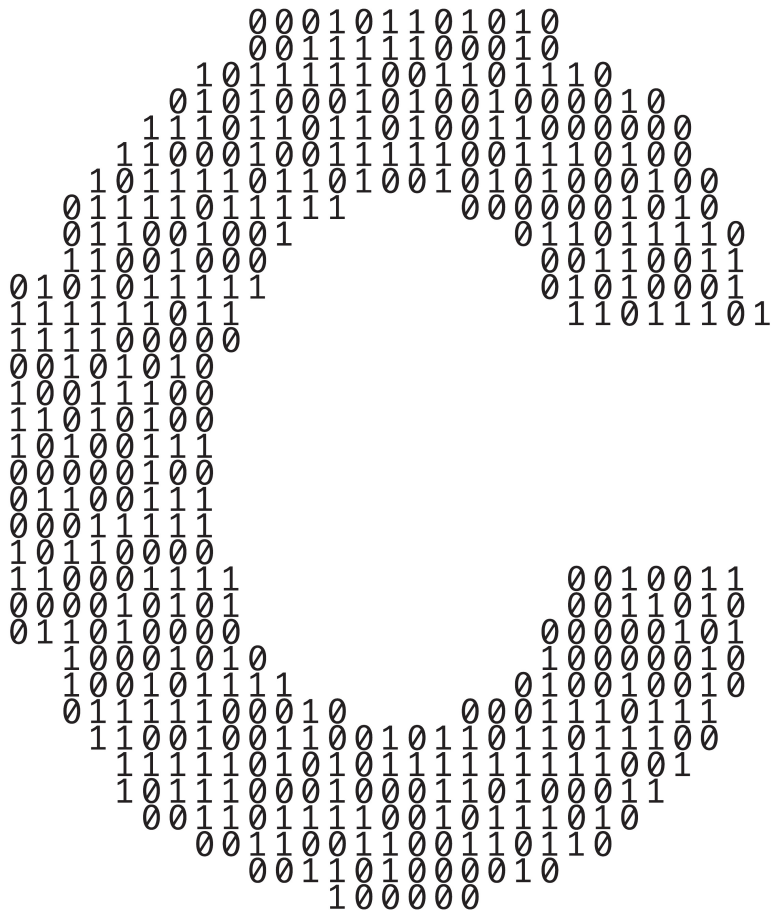
# 8

# Characters and Strings



## Objectives

In this chapter, you'll:

- Use the functions of the character-handling library (`<ctype.h>`).
- Use the string-conversion functions of the general utilities library (`<stdlib.h>`).
- Use the string and character input/output functions of the standard input/output library (`<stdio.h>`).
- Use the string-processing functions of the string-handling library (`<string.h>`).
- Use the memory-processing functions of the string-handling library (`<string.h>`).

Outline



## 8.1 Introduction

This chapter introduces the C standard library functions that help you process characters, strings, lines of text and blocks of memory. The chapter discusses the techniques used to develop editors, word processors, page-layout software and other kinds of text-processing software. The text manipulations performed by formatted input/output functions like `printf` and `scanf` can be implemented using the functions this chapter presents.

## 8.2 Fundamentals of Strings and Characters

Characters are the fundamental building blocks of your programs. Every program is composed of characters that—when grouped together meaningfully—the computer interprets as a series of instructions used to accomplish a task. A program may contain **character constants**—each is an `int` value represented as a character in single quotes. A character constant's value is that character's integer value in the machine's **character**

set. For example, `'z'` represents the letter z's integer value, and `'\n'` represents a newline's integer value.

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**, are written in double quotation marks as follows:

```
"John Q. Doe"              (a name)
"99999 Main Street"        (a street address)
"Waltham, Massachusetts"   (a city and state)
"(201) 555-1212"           (a telephone number)
```

### Strings Are Null Terminated

Every string must end with the **null character** (`'\0'`). Printing a "string" that does not contain a terminating null character is a logic error. The results of this are undefined. On some systems, printing will continue past the end of the "string" until a null character is encountered. On others, your program will terminate prematurely (i.e., "crash") and indicate a "segmentation fault" or "access violation" error. ⊗ ERR

### Strings and Pointers

You access a string via a *pointer* to its first character. A string's "value" is the address of its first character. Thus, in C, it's appropriate to say that a string is a pointer to the string's first character. This is just like arrays, because strings are simply arrays of characters.

### Initializing char Arrays and char * Pointers

You can initialize a character array or a `char *` variable with a string. The definitions

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

initialize `color` and `colorPtr` to the string `"blue"`. The first definition creates a 5-element array `color` containing the *modifiable* characters `'b'`, `'l'`, `'u'`, `'e'` and `'\0'`. The second definition creates the pointer variable `colorPtr` that points to the letter `'b'` in `"blue"`, which is *not modifiable*.

The `color` array definition also can be written as

```
char color[] = {'b', 'l', 'u', 'e', '\0'};
```

The preceding definition automatically determines the array's size based on its number of initializers (5). When storing a string in a `char` array, the array must be large enough to store the string *and* its terminating null character. Not allocating sufficient space in a character array to store the null character that terminates a string is an error. C allows you to store strings of any length. If a string is longer than the `char` array in which you store it, characters beyond the array's end may overwrite other data in memory. ⊗ ERR

### String Literals Should Not Be Modified

SE The C standard indicates that a string literal is immutable—that is, not modifiable. If you might need to modify a string, it must be stored in a character array.

### Reading a String with `scanf`

Function `scanf` can read a string and store it in a `char` array. Assume we have a `char` array `word` containing 20 elements. You can read a string into the array with

```
scanf("%19s", word);
```

Since `word` is an array, the array name is a pointer to the array's first element. So, the `&` that we typically use with `scanf`'s arguments is not required.

Recall from Section 6.5.4 that `scanf` reads characters until it encounters a space, tab, newline or end-of-file indicator. The field width `19` in the preceding statement ensures that `scanf` reads a *maximum* of 19 characters, saving the last array element for the string's terminating null character. This prevents `scanf` from writing characters into memory beyond the array's last element.

Without the field width `19` in the conversion specification `%19s`, the user input could exceed 19 characters and overwrite other data in memory. If so, your program might crash, or overwrite other data in memory. So, always use a field width when reading strings with `scanf`. (For reading input lines of arbitrary length, there's a non-standard—yet widely supported—function `readline`, usually included in `stdio.h`.)

## Self Check

**1** ***(Fill-In)*** A string is accessed via a _____ to the string's first character.
**Answer:** pointer.

**2** ***(True/False)*** The following definition initializes the `color` array to the character string `"blue"`:

```
char color[] = {'b', 'l', 'u', 'e'};
```

**Answer:** *False*. Actually, to be a character string, the `color` array must end with the null character, as in

```
char color[] = {'b', 'l', 'u', 'e', '\0'};
```

**3** ***(True/False)*** Printing a string that does not contain a terminating null character is a logic error—program execution terminates immediately.
**Answer:** *False*. Actually, printing will continue past the end of the string until a null character is encountered.

## 8.3 Character-Handling Library

The character-handling library (`<ctype.h>`) contains functions that test and manipulate character data. Each function receives an `unsigned char` (represented as an `int`) or `EOF` as an argument. As we discussed in Chapter 4, characters are often manipulated as integers because a character in C is a one-byte integer. `EOF`'s value is typically –1. The following table summarizes the character-handling library functions.

| Prototype | Function description |
|---|---|
| `int isblank(int c);` | Returns a true value if `c` is a blank character that separates words in a line of text; otherwise, it returns 0 (false). |
| `int isdigit(int c);` | Returns a true value if `c` is a digit; otherwise, it returns 0 (false). |
| `int isalpha(int c);` | Returns a true value if `c` is a letter; otherwise, it returns 0 (false). |
| `int isalnum(int c);` | Returns a true value if `c` is a digit or a letter; otherwise, it returns 0 (false). |
| `int isxdigit(int c);` | Returns a true value if `c` is a hexadecimal digit character; otherwise, it returns 0 (false). (See Online Appendix E 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; otherwise, it returns 0 (false). |
| `int isupper(int c);` | Returns a true value if `c` is an uppercase letter; otherwise, it returns 0 (false). |
| `int tolower(int c);` | If `c` is an uppercase letter, `tolower` returns `c` as a lowercase letter; otherwise, it returns the argument unchanged. |
| `int toupper(int c);` | If `c` is a lowercase letter, `toupper` returns `c` as an uppercase letter; otherwise, it returns the argument unchanged. |
| `int isspace(int c);` | Returns a true value if `c` is a whitespace character—newline (`'\n'`), space (`' '`), form feed (`'\f'`), carriage return (`'\r'`), horizontal tab (`'\t'`) or vertical tab (`'\v'`)—otherwise, it returns 0 (false). |
| `int iscntrl(int c);` | Returns a true value if `c` is a control character—horizontal tab (`'\t'`), vertical tab (`'\v'`), form feed (`'\f'`), alert (`'\a'`), backspace (`'\b'`), carriage return (`'\r'`), newline (`'\n'`) and others—otherwise, it returns 0 (false). |
| `int ispunct(int c);` | Returns a true value if `c` is a printing character other than a space, a digit, or a letter—such as `$`, `#`, `(`, `)`, `[`, `]`, `{`, `}`, `;`, `:` or `%`—otherwise, it returns 0 (false). |
| `int isprint(int c);` | Returns a true value if `c` is a printing character (i.e., a character that's visible on the screen) including a space; otherwise, it returns 0 (false). |
| `int isgraph(int c);` | Returns a true value if `c` is a printing character other than a space; otherwise, it returns 0 (false). |

### 8.3.1 Functions `isdigit`, `isalpha`, `isalnum` and `isxdigit`

Figure 8.1 demonstrates functions `isdigit`, `isalpha`, `isalnum` and `isxdigit`. Function `isdigit` determines whether its argument is a digit (`0`–`9`). Function `isalpha` determines whether its argument is an uppercase (`A`–`Z`) or lowercase letter (`a`–`z`). Function `isalnum` determines whether its argument is an uppercase letter, a lowercase letter or a digit. Function `isxdigit` determines whether its argument is a hexadecimal digit (`A`–`F`, `a`–`f`, `0`–`9`).

```c
// fig08_01.c
// Using functions isdigit, isalpha, isalnum, and isxdigit
#include <ctype.h>
#include <stdio.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");

   printf("%s\n%s%s\n%s%s\n%s%s\n\n", "According to isalnum:",
      isalnum('A') ? "A is a " : "A is not a ", "digit or a letter",
      isalnum('8') ? "8 is a " : "8 is not a ", "digit or a letter",
      isalnum('#') ? "# is a " : "# is not a ", "digit or a letter");

   printf("%s\n%s%s\n%s%s\n%s%s\n%s%s\n%s%s\n", "According to isxdigit:",
      isxdigit('F') ? "F is a " : "F is not a ", "hexadecimal digit",
      isxdigit('J') ? "J is a " : "J is not a ", "hexadecimal digit",
      isxdigit('7') ? "7 is a " : "7 is not a ", "hexadecimal digit",
      isxdigit('$') ? "$ is a " : "$ is not a ", "hexadecimal digit",
      isxdigit('f') ? "f is a " : "f is not a ", "hexadecimal digit");
}
```

```
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. 8.1** | Using functions `isdigit`, `isalpha`, `isalnum` and `isxdigit`.

Figure 8.1 uses the conditional operator (`?:`) to determine whether the string `" is a "` or the string `" is not a "` should be printed in the output for each character tested. For example, the expression

```
isdigit('8') ? "8 is a " : "8 is not a "
```

indicates that if `'8'` is a digit, the string `"8 is a "` is printed, and if `'8'` is not a digit (i.e., `isdigit` returns 0), the string `"8 is not a "` is printed.

### 8.3.2 Functions `islower`, `isupper`, `tolower` and `toupper`

Figure 8.2 demonstrates functions **`islower`**, **`isupper`**, **`tolower`** and **`toupper`**. Function `islower` determines whether its argument is a lowercase letter (a–z). Function `isupper` determines whether its argument is an uppercase letter (A–Z). Function `tolower` converts an uppercase letter to a lowercase letter and returns the lowercase letter. If the argument is not an uppercase letter, `tolower` returns the argument unchanged. Function `toupper` converts a lowercase letter to an uppercase letter and returns the uppercase letter. If the argument is not a lowercase letter, `toupper` returns the argument unchanged.

```
1  // fig08_02.c
2  // Using functions islower, isupper, tolower and toupper
3  #include <ctype.h>
4  #include <stdio.h>
5
6  int main(void) {
7     printf("%s\n%s%s\n%s%s\n%s%s\n%s%s\n\n", "According to islower:",
8        islower('p') ? "p is a " : "p is not a ", "lowercase letter",
9        islower('P') ? "P is a " : "P is not a ", "lowercase letter",
10       islower('5') ? "5 is a " : "5 is not a ", "lowercase letter",
11       islower('!') ? "! is a " : "! is not a ", "lowercase letter");
12
13    printf("%s\n%s%s\n%s%s\n%s%s\n%s%s\n\n", "According to isupper:",
14       isupper('D') ? "D is an " : "D is not an ", "uppercase letter",
15       isupper('d') ? "d is an " : "d is not an ", "uppercase letter",
16       isupper('8') ? "8 is an " : "8 is not an ", "uppercase letter",
17       isupper('$') ? "$ is an " : "$ is not an ", "uppercase letter");
18
19    printf("%s%c\n%s%c\n%s%c\n%s%c\n",
20       "u converted to uppercase is ", toupper('u'),
21       "7 converted to uppercase is ", toupper('7'),
22       "$ converted to uppercase is ", toupper('$'),
23       "L converted to lowercase is ", tolower('L'));
24 }
```

```
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
```

**Fig. 8.2** | Using functions `islower`, `isupper`, `tolower` and `toupper`. (Part 1 of 2.)

```
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. 8.2** | Using functions `islower`, `isupper`, `tolower` and `toupper`. (Part 2 of 2.)

### 8.3.3 Functions `isspace`, `iscntrl`, `ispunct`, `isprint` and `isgraph`

Figure 8.3 demonstrates functions **`isspace`**, **`iscntrl`**, **`ispunct`**, **`isprint`** and **`isgraph`**. Function `isspace` determines whether a character is one of the following whitespace characters: space (`' '`), form feed (`'\f'`), newline (`'\n'`), carriage return (`'\r'`), horizontal tab (`'\t'`) or vertical tab (`'\v'`). Function `iscntrl` determines whether a character is one of the following **control characters**: horizontal tab (`'\t'`), vertical tab (`'\v'`), form feed (`'\f'`), alert (`'\a'`), backspace (`'\b'`), carriage return (`'\r'`) or newline (`'\n'`). Function `ispunct` determines whether a character is a **printing character** other than a space, a digit or a letter, such as `$`, `#`, `(`, `)`, `[`, `]`, `{`, `}`, `;`, `:` or `%`. Function `isprint` determines whether a character can be displayed on the screen (including the space character). Function `isgraph` is the same as `isprint`, except that the space character is not included.

```
1  // fig08_03.c
2  // Using functions isspace, iscntrl, ispunct, isprint and isgraph
3  #include <ctype.h>
4  #include <stdio.h>
5
6  int main(void) {
7     printf("%s\n%s%s%s\n%s%s%s\n%s%s\n\n", "According to isspace:",
8         "Newline", isspace('\n') ? " is a " : " is not a ",
9         "whitespace character",
10        "Horizontal tab", isspace('\t') ? " is a " : " is not a ",
11        "whitespace character",
12        isspace('%') ? "% is a " : "% is not a ", "whitespace character");
13
14    printf("%s\n%s%s%s\n%s%s\n\n", "According to iscntrl:",
15        "Newline", iscntrl('\n') ? " is a " : " is not a ",
16        "control character",
17        iscntrl('$') ? "$ is a " : "$ is not a ", "control character");
18
19    printf("%s\n%s%s\n%s%s\n%s%s\n\n", "According to ispunct:",
20        ispunct(';') ? "; is a " : "; is not a ", "punctuation character",
21        ispunct('Y') ? "Y is a " : "Y is not a ", "punctuation character",
22        ispunct('#') ? "# is a " : "# is not a ", "punctuation character");
```

**Fig. 8.3** | Using functions `isspace`, `iscntrl`, `ispunct`, `isprint` and `isgraph`. (Part 1 of 2.)

```
23
24      printf("%s\n%s%s\n%s%s%s\n\n", "According to isprint:",
25          isprint('$') ? "$ is a " : "$ is not a ", "printing character",
26          "Alert", isprint('\a') ? " is a " : " is not a ",
27          "printing character");
28
29      printf("%s\n%s%s\n%s%s%s\n",  "According to isgraph:",
30          isgraph('Q') ? "Q is a " : "Q is not a ",
31          "printing character other than a space",
32          "Space", isgraph(' ') ? " is a " : " is not a ",
33          "printing character other than a space");
34  }
```

```
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. 8.3** | Using functions `isspace`, `iscntrl`, `ispunct`, `isprint` and `isgraph`. (Part 2 of 2.)

## Self Check

**1** *(Multiple Choice)* Which functions is described by "Returns a true value if the argument character is a digit or a letter; otherwise, returns 0 (false)"?

a) `isalnum`.
b) `isdigit`.
c) `isalpha`.
d) `isxdigit`.

**Answer:** a.

**2** *(Code)* What does the following `printf` print?

```
printf("%s\n%s%s\n%s%s%s\n%s%s\n%s%s\n\n", "According to isalpha:",
   isalpha('X') ? "X is a " : "X is not a ", "letter",
   isalpha('m') ? "m is a " : "m is not a ", "letter",
   isalpha('$') ? "$ is a " : "$ is not a ", "letter",
   isalpha('7') ? "7 is a " : "7 is not a ", "letter");
```

**Answer:**

```
According to isalpha:
X is a letter
m is a letter
$ is not a letter
7 is not a letter
```

## 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. The following table summarizes the string-conversion functions. The C standard also includes `strtoll` and `strtoull` for converting strings to `long long int` and `unsigned long long int`, respectively.

| Function prototype | Function description |
|---|---|
| `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`. |

### 8.4.1 Function `strtod`

Function `strtod` (Fig. 8.4) converts a sequence of characters representing a floating-point value to `double`. The function returns `0` if it's unable to convert part of its first argument to `double`. The function receives two arguments—a string (`char *`) and a pointer to a string (`char **`). The string argument contains the character sequence to be converted to `double`. Whitespace characters at the beginning of the string are ignored. The function uses the `char **` argument to aim a `char *` in the caller (`stringPtr`) at the first character after the converted portion of the string. If nothing can be converted, the function aims the pointer at the beginning of the string. Line 10 assigns `d` the `double` value converted from `string` and aims `stringPtr` at the `%` in `string`.

```
1  // fig08_04.c
2  // Using function strtod
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  int main(void) {
7     const char *string = "51.2% are admitted";
8     char *stringPtr = NULL;
```

**Fig. 8.4** | Using function `strtod`. (Part 1 of 2.)

```
9
10     double d = strtod(string, &stringPtr);
11
12     printf("The string \"%s\" is converted to the\n", string);
13     printf("double value %.2f and the string \"%s\"\n", d, stringPtr);
14  }
```

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

**Fig. 8.4** | Using function `strtod`. (Part 2 of 2.)

### 8.4.2 Function `strtol`

Function `strtol` (Fig. 8.5) converts to `long int` a sequence of characters representing an integer. The function returns `0` if it's unable to convert any portion of its first argument to `long int`. The function's three arguments are a string (`char *`), a pointer to a string and an integer. This function works identically to `strtod`, but the third argument specifies the *base* of the value being converted.

```
1  // fig08_05.c
2  // Using function strtol
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  int main(void) {
7     const char *string = "-1234567abc";
8     char *remainderPtr = NULL;
9
10    long x = strtol(string, &remainderPtr, 0);
11
12    printf("%s\"%s\"\n%s%ld\n%s\"%s\"\n%s%ld\n",
13       "The original string is ", string,
14       "The converted value is ", x,
15       "The remainder of the original string is ", remainderPtr,
16       "The converted value plus 567 is ", x + 567);
17 }
```

```
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. 8.5** | Using function `strtol`.

Line 10 assigns `x` the `long` value converted from `string` and aims `remainderPtr` at the `"a"` in `string`. Using `NULL` for the second argument causes the *remainder of the string to be ignored*. The third argument, `0`, indicates that the value to convert can be in octal (base 8), decimal (base 10) or hexadecimal (base 16) format. The base can be

specified as 0 or any value between 2 and 36.[1] Integer representations from base 11 to base 36 use the letters A–Z to represent the integer values 10–35. For example, hexadecimal values can consist of the digits 0–9 and the characters A–F.

### 8.4.3 Function `strtoul`

Function `strtoul` (Fig. 8.6) converts to `unsigned long int` a sequence of characters representing an `unsigned long int` value. The function works identically to function `strtol`. Line 10 assigns `x` the `unsigned long int` value converted from `string` and aims `remainderPtr` at the `"a"` in `string`. The third argument, `0`, indicates that the value to convert can be in octal, decimal or hexadecimal format.

```
1  // fig08_06.c
2  // Using function strtoul
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  int main(void) {
7     const char *string = "1234567abc";
8     char *remainderPtr = NULL;
9
10    unsigned long int x = strtoul(string, &remainderPtr, 0);
11
12    printf("%s\"%s\"\n%s%lu\n%s\"%s\"\n%s%lu\n",
13       "The original string is ", string,
14       "The converted value is ", x,
15       "The remainder of the original string is ", remainderPtr,
16       "The converted value minus 567 is ", x - 567);
17 }
```

```
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
```

**Fig. 8.6** | Using function `strtoul`.

## Self Check

**1** *(Discussion)* Why would a function's parameter list contain a `char **` parameter?
**Answer:** A `char **` typically is a pointer to a `char *` pointer in the caller. A called function uses such a pointer to receive a `char *` by reference to modify it in the caller—for example, to aim it at another string. This is an example of a pointer to a pointer.

**2** *(Multiple Choice)* Which of the following statements about function `strtol` is *false*?
a) It converts to `long int` a sequence of characters representing an integer, or it returns `0` if it's unable to convert any portion of its first argument to `long int`.

1. See online Appendix E for a detailed explanation of the octal, decimal and hexadecimal number systems.

b) `strtol`'s three arguments are a string (`char *`), a pointer to a string (`char **`) and an integer.
c) The string argument contains the character sequence to convert to `long`—any whitespace characters at the beginning of the string are ignored.
d) The function uses the `char **` argument to give the caller access to the numeric portion of the string being converted.

**Answer:** d) is *false*. Actually, the function uses the `char **` argument to modify a `char *` in the caller to point to the location of the first character *after* the string's converted portion. If nothing is converted, the function modifies the `char *` to point to the entire string.

## 8.5 Standard Input/Output Library Functions

This section presents the standard input/output (`<stdio.h>`) library's character- and string-manipulation functions, which we summarize in the following table.

| Function prototype | Function description |
|---|---|
| `int getchar(void);` | Returns the next character from the standard input as an integer. |
| `char *fgets(char *s, int n, FILE *stream);` | Reads 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. This chapter uses the stream `stdin`—the standard input stream—to read characters from the keyboard. A terminating null character is appended to the array. Returns the string that was read into `s`. If a newline is encountered, it's included in the stored string. |
| `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 nonzero integer if successful, or `EOF` if an error occurs. |
| `int sprintf(char *s, const char *format, ...);` | Equivalent to `printf`, but 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. |
| `int sscanf(char *s, const char *format, ...);` | Equivalent to `scanf`, but 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. |

### 8.5.1 Functions `fgets` and `putchar`

Figure 8.7 uses functions `fgets` and `putchar` to read a line of text from the standard input (keyboard) and recursively output the line's characters in reverse order. Line 12 uses `fgets` to read characters into its `char` array argument until it encounters a newline or the end-of-file indicator, or until the maximum number of characters is read.

The maximum number of characters is one fewer than `fgets`'s second argument. The third argument is the stream from which to read characters—in this case, the standard input stream (`stdin`). When reading terminates, `fgets` appends a null character (`'\0'`) to the array. Function `putchar` (line 27) prints its character argument.

```
1  // fig08_07.c
2  // Using functions fgets and putchar
3  #include <stdio.h>
4  #define SIZE 80
5
6  void reverse(const char * const sPtr);
7
8  int main(void) {
9     char sentence[SIZE] = "";
10
11    puts("Enter a line of text:");
12    fgets(sentence, SIZE, stdin); // read a line of text
13
14    printf("\n%s", "The line printed backward is:");
15    reverse(sentence);
16    puts("");
17 }
18
19 // recursively outputs characters in string in reverse order
20 void reverse(const char * const sPtr) {
21    // if end of the string
22    if ('\0' == sPtr[0]) { // base case
23       return;
24    }
25    else { // if not end of the string
26       reverse(&sPtr[1]); // recursion step
27       putchar(sPtr[0]); // use putchar to display character
28    }
29 }
```

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

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

**Fig. 8.7** | Using functions `fgets` and `putchar`.

### Function reverse

The program calls the recursive function `reverse`[2] to print the line of text backward. If the array's first character is the null character `'\0'`, `reverse` returns. Otherwise, `reverse` calls itself recursively with the subarray's address beginning at element `sPtr[1]`. Line 27 outputs the character at `sPtr[0]` when the recursive call completes.

2. We use recursion here for demonstration purposes. It's usually more efficient to use a loop to iterate from a string's last character (the one at the position one less than the string's length) to its first character (the one at position 0).

The order of the two statements in lines 26 and 27 causes `reverse` to walk to the string's terminating null character *before* displaying any characters. As the recursive calls complete, the characters are output in reverse order.

### 8.5.2 Function `getchar`

Figure 8.8 uses functions **`getchar`** to read one character at a time from the standard input into the character array `sentence`, then uses `puts` to display the characters as a string. Function `getchar` reads a character from the standard input and returns the character as an integer. Recall from Section 4.6 that an integer is returned to support the end-of-file indicator. As you know, `puts` takes a string as an argument and displays the string followed by a newline character. The program stops inputting characters when 79 characters have been read or when `getchar` reads a newline character. Line 18 appends a null character to `sentence` to terminate the string. Then line 21 uses `puts` to display `sentence`.

```
1  // fig08_08.c
2  // Using function getchar
3  #include <stdio.h>
4  #define SIZE 80
5
6  int main(void) {
7     int c = 0; // variable to hold character input by user
8     char sentence[SIZE] = "";
9     int i = 0;
10
11    puts("Enter a line of text:");
12
13    // use getchar to read each character
14    while ((i < SIZE - 1) && (c = getchar()) != '\n') {
15       sentence[i++] = c;
16    }
17
18    sentence[i] = '\0'; // terminate string
19
20    puts("\nThe line entered was:");
21    puts(sentence); // display sentence
22 }
```

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

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

**Fig. 8.8** | Using function `getchar`.

### 8.5.3 Function `sprintf`

Figure 8.9 uses function **`sprintf`** to print formatted data into `char` array `s`. The function uses the same conversion specifications as `printf` (see Chapter 9 for a detailed

discussion of formatting). The program inputs an `int` value and a `double` value to be formatted and printed to array `s`. Array `s` is the first argument of `sprintf`.

```
1  // fig08_09.c
2  // Using function sprintf
3  #include <stdio.h>
4  #define SIZE 80
5
6  int main(void) {
7     int x = 0;
8     double y = 0.0;
9
10    puts("Enter an integer and a double:");
11    scanf("%d%lf", &x, &y);
12
13    char s[SIZE] = {'\0'}; // create char array
14    sprintf(s, "integer:%6d\ndouble:%7.2f", x, y);
15
16    printf("The formatted output stored in array s is:\n%s\n", s);
17 }
```

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

**Fig. 8.9** | Using function `sprintf`.

### 8.5.4 Function sscanf

Figure 8.10 demonstrates function **`sscanf`**, which works like `scanf` but reads formatted data from a string. The program reads an `int` and a `double` from `char` array `s`, stores them in `x` and `y`, then displays them.

```
1  // fig08_10.c
2  // Using function sscanf
3  #include <stdio.h>
4
5  int main(void) {
6     char s[] = "31298 87.375";
7     int x = 0;
8     double y = 0;
9
10    sscanf(s, "%d%lf", &x, &y);
11    puts("The values stored in character array s are:");
12    printf("integer:%6d\ndouble:%8.3f\n", x, y);
13 }
```

**Fig. 8.10** | Using function `sscanf`. (Part 1 of 2.)

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

**Fig. 8.10** | Using function `sscanf`. (Part 2 of 2.)

## Self Check

**1** *(Multiple Choice)* Which function is described by "Prints the character stored in its parameter and returns it as an integer"?

a) `getchar`.
b) `sprintf`.
c) `puts`.
d) `putchar`.

**Answer:** d.

**2** *(True/False)* Function `getchar` reads a character from the standard input and returns it as a `char`.

**Answer:** *False*. Actually, `getchar` returns an `int` to support the end-of-file indicator, which is `-1`.

## 8.6 String-Manipulation Functions of the String-Handling Library

The string-handling library (`<string.h>`) provides 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.

This section presents the string-handling library's string-manipulation functions, which are summarized in the following table. Other than `strncpy`, each function appends the null character to its result.

| Function prototype | Function description |
|---|---|
| `char *strcpy(char *s1, const char *s2)` | Copies string `s2` into array `s1` and returns `s1`. |
| `char *strncpy(char *s1, const char *s2, size_t n)` | Copies at most `n` characters of string `s2` into array `s1` and returns `s1`. |
| `char *strcat(char *s1, const char *s2)` | Appends string `s2` to array `s1` and returns `s1`. String `s2`'s first character overwrites `s1`'s terminating null character. |

| Function prototype | Function description |
|---|---|
| `char *strncat(char *s1, const char *s2, size_t n)` | Appends at most `n` characters of string `s2` to array `s1` and returns `s1`. String `s2`'s first character overwrites `s1`'s terminating null character. |

Functions **`strncpy`** and **`strncat`** specify a `size_t` parameter. Function `strcpy` copies the string in the second argument into the `char` array in its first argument. You must ensure that the array is large enough to store the string and its terminating null character, which is also copied. Function `strncpy` is equivalent to `strcpy` but copies only the specified number of characters. *Function `strncpy` will not copy the terminating null character of its second argument unless the number of characters to be copied is more than the string's length.* 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`, some implementations append null characters to the array until the total number of characters specified by the third argument is written. Other implementations stop after writing the first null character. It's a logic error if you do not append a terminating null character to `strncpy`'s first argument when the third argument is less than or equal to the second argument's string length.

ERR

## 8.6.1 Functions `strcpy` and `strncpy`

Figure 8.11 uses `strcpy` to copy the entire string in array `x` into array `y`. It uses `strncpy` to copy the first `14` characters of array `x` into array `z`. Line 19 appends a null character (`'\0'`) to array `z` because the `strncpy` call *does not write a terminating null character*—the third argument is less than the second argument's string length.

```
1  // fig08_11.c
2  // Using functions strcpy and strncpy
3  #include <stdio.h>
4  #include <string.h>
5  #define SIZE1 25
6  #define SIZE2 15
7
8  int main(void) {
9     char x[] = "Happy Birthday to You"; // initialize char array x
10    char y[SIZE1] = ""; // create char array y
11    char z[SIZE2] = ""; // create char array z
12
13    // copy contents of x into y
14    printf("%s%s\n%s%s\n",
15       "The string in array x is: ", x,
16       "The string in array y is: ", strcpy(y, x));
17
```

**Fig. 8.11** | Using functions `strcpy` and `strncpy`. (Part 1 of 2.)

```
18     strncpy(z, x, SIZE2 - 1); // copy first 14 characters of x into z
19     z[SIZE2 - 1] = '\0'; // terminate string in z, because '\0' not copied
20     printf("The string in array z is: %s\n", z);
21  }
```

```
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
```

**Fig. 8.11** | Using functions `strcpy` and `strncpy`. (Part 2 of 2.)

### 8.6.2 Functions `strcat` and `strncat`

Function **`strcat`** appends its second argument string to the string in its `char` array first argument, replacing the first argument's null (`'\0'`) character. *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 and adds a terminating `'\0'`. Figure 8.12 demonstrates function `strcat` and function `strncat`.

```
1  // fig08_12.c
2  // Using functions strcat and strncat
3  #include <stdio.h>
4  #include <string.h>
5
6  int main(void) {
7     char s1[20] = "Happy "; // initialize char array s1
8     char s2[] = "New Year "; // initialize char array s2
9     char s3[40] = ""; // initialize char array s3 to empty
10
11    printf("s1 = %s\ns2 = %s\n", s1, s2);
12
13    // concatenate s2 to s1
14    printf("strcat(s1, s2) = %s\n", strcat(s1, s2));
15
16    // concatenate first 6 characters of s1 to s3
17    printf("strncat(s3, s1, 6) = %s\n", strncat(s3, s1, 6));
18
19    // concatenate s1 to s3
20    printf("strcat(s3, s1) = %s\n", strcat(s3, s1));
21 }
```

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

**Fig. 8.12** | Using functions `strcat` and `strncat`.

## Self Check

**1** *(Multiple Choice)* Which of the following statements about functions `strcat` and `strncat` is *false*?

a) Function `strcat` appends its second argument string to the string in its `char` array first argument.

b) The first character of `strcat`'s second argument is placed immediately after the null (`'\0'`) that terminates the string in the first argument.

c) You must ensure that the array containing the first string is large enough to store the first string, the second string and the terminating `'\0'` copied from the second string.

d) Function `strncat` appends a specified number of characters from the second string to the first string. A terminating `'\0'` is automatically appended to the result.

**Answer:** b) is *false*. Actually, the first character of `strcat`'s second argument *replaces* the null (`'\0'`) that terminates the string in the first argument.

**2** *(Fill-In)* Function `strcpy` copies its second argument (a string) into its first argument, which is a character array that must be _____.

**Answer:** large enough to store the string, including its terminating null character.

## 8.7 Comparison Functions of the String-Handling Library

This section presents the string-handling library's **string-comparison functions**, **`strcmp`** and **`strncmp`**, which are summarized below.

| 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. |

Figure 8.13 compares three strings using `strcmp` and `strncmp`. Function `strcmp` performs a character-by-character comparison of its two string arguments. The function returns:

- 0 if the strings are equal,
- a *negative value* if the first string is less than the second string, or
- a *positive value* if the first string is greater than the second string.

Function `strncmp` is equivalent to `strcmp` but compares up to a specified number of characters. Function `strncmp` does *not* compare characters following a null character in a string. The program prints the integer value returned by each function call.

```
1  // fig08_13.c
2  // Using functions strcmp and strncmp
3  #include <stdio.h>
4  #include <string.h>
5
6  int main(void) {
7     const char *s1 = "Happy New Year"; // initialize char pointer
8     const char *s2 = "Happy New Year"; // initialize char pointer
9     const char *s3 = "Happy Holidays"; // initialize char pointer
10
11    printf("s1 = %s\ns2 = %s\ns3 = %s\n\n%s%2d\n%s%2d\n%s%2d\n\n",
12       s1, s2, s3,
13       "strcmp(s1, s2) = ", strcmp(s1, s2),
14       "strcmp(s1, s3) = ", strcmp(s1, s3),
15       "strcmp(s3, s1) = ", strcmp(s3, s1));
16
17    printf("%s%2d\n%s%2d\n%s%2d\n",
18       "strncmp(s1, s3, 6) = ", strncmp(s1, s3, 6),
19       "strncmp(s1, s3, 7) = ", strncmp(s1, s3, 7),
20       "strncmp(s3, s1, 7) = ", strncmp(s3, s1, 7));
21 }
```

```
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) =  1
strncmp(s3, s1, 7) = -1
```

**Fig. 8.13** | Using functions `strcmp` and `strncmp`.

### How Strings Are Compared

To understand what it means for one string to be "greater than" or "less than" another, consider the process of alphabetizing last names. You'd, no doubt, place "Jones" before "Smith" because "J" comes before "S" in the alphabet. But the alphabet is more than just a list of 26 letters—it's an ordered list of characters. Each letter occurs in a specific position within the list. "Z" is more than merely a letter of the alphabet—specifically, "Z" is the alphabet's 26th letter. Also, recall that lowercase letters have higher numeric values than uppercase letters, so "a" is greater than "A."

How do the string-comparison functions know that one particular letter comes before another? All characters are represented inside the computer as **numeric codes**

in character sets such as ASCII and Unicode; when the computer compares two strings, it actually compares the characters' numeric codes in each string. This is called a lexicographical comparison. See Appendix B for the ASCII characters' numeric values. ASCII is a subset of the Unicode character set.

The negative and positive values returned by `strcmp` and `strncmp` are *implementation-dependent*. For some, these values are `-1` or `1`, as in Fig. 8.13. For others, the values returned are the difference between the numeric codes of the first different characters in each string. For this program's comparisons, that's the difference between the numeric codes of `"N"` in `"New"` and `"H"` in `"Holidays"`—`6` or `-6`, depending on which string is the first argument.

## Self Check

**1** ***(Multiple Choice)*** Which of the following statements about functions `strcmp` and `strncmp` is *false*?

a) Function `strcmp` compares its first string argument with its second string argument, character-by-character.

b) Function `strcmp` returns 0 if the strings are equal, a negative value if the first string is less than the second and a positive value if the first string is greater than the second.

c) Function `strncmp` is equivalent to `strcmp` but compares up to a specified number of characters.

d) Function `strncmp` compares characters following a null character in a string.

**Answer:** d) is *false*. Actually, function `strncmp` does not compare characters following a null character in a string.

**2** ***(Discussion)*** How do the string-comparison functions `strcmp` and `strncmp` know that one particular letter "comes before" another?

**Answer:** All characters are represented inside the computer as numeric codes in character sets such as ASCII and Unicode. When the computer compares two strings, it compares the characters' numeric codes. This is called a lexicographical comparison.

## 8.8 Search Functions of the String-Handling Library

This section presents the string-handling library functions that search strings for characters and other strings, summarized in the following table.

| Function prototypes and descriptions |
|---|
| `char *strchr(const char *s, int c);`<br>Locates the first occurrence of character `c` in string `s`. If `c` is found, `strchr` returns a pointer to `c` in `s`. Otherwise, a `NULL` pointer is returned. |
| `size_t strcspn(const char *s1, const char *s2);`<br>Determines and returns the length of the initial segment of string `s1` consisting of characters not contained in string `s2`. |

**Function prototypes and descriptions**

`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 `s2` is found, `strpbrk` returns a pointer to that character in `s1`. Otherwise, it returns `NULL`.

`char *strrchr(const char *s, int c);`

Locates the last occurrence of `c` in string `s`. If `c` is found, `strrchr` returns a pointer to `c` in string `s`. Otherwise, it returns `NULL`.

`char *strstr(const char *s1, const char *s2);`

Locates the first occurrence in string `s1` of string `s2`. If the string is found, `strstr` returns a pointer to the string in `s1`. Otherwise, it returns `NULL`.

`char *strtok(char *s1, const char *s2);`

A sequence of calls to `strtok` breaks string `s1` into tokens separated by characters contained in string `s2`. Tokens are logical pieces, such as words in a line of text. The first call uses `s1` as the first argument. Subsequent calls to continue tokenizing the same string require `NULL` as the first argument. Each call returns a pointer to the current token. If there are no more tokens, `strtok` returns `NULL`.

### 8.8.1 Function `strchr`

Function `strchr` searches for the first occurrence of a character in a string. If the character is found, `strchr` returns a pointer to the character in the string; otherwise, `strchr` returns `NULL`. Figure 8.14 searches for the first occurrences of `'a'` and `'z'` in `"This is a test"`.

```
1  // fig08_14.c
2  // Using function strchr
3  #include <stdio.h>
4  #include <string.h>
5
6  int main(void) {
7     const char *string = "This is a test"; // initialize char pointer
8     char character1 = 'a';
9     char character2 = 'z';
10
11    // if character1 was found in string
12    if (strchr(string, character1) != NULL) { // can remove "!= NULL"
13       printf("\'%c\' was found in \"%s\".\n", character1, string);
14    }
15    else { // if character1 was not found
16       printf("\'%c\' was not found in \"%s\".\n", character1, string);
17    }
```

**Fig. 8.14** | Using function `strchr`. (Part 1 of 2.)

```
18
19    // if character2 was found in string
20    if (strchr(string, character2) != NULL) { // can remove "!= NULL"
21       printf("\'%c\' was found in \"%s\".\n", character2, string);
22    }
23    else { // if character2 was not found
24       printf("\'%c\' was not found in \"%s\".\n", character2, string);
25    }
26 }
```

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

**Fig. 8.14** | Using function `strchr`. (Part 2 of 2.)

### 8.8.2 Function `strcspn`

Function **`strcspn`** (Fig. 8.15) determines the length of the initial part of its first string argument that does *not* contain any characters from its second string argument. The function returns the segment's length.

```
1  // fig08_15.c
2  // Using function strcspn
3  #include <stdio.h>
4  #include <string.h>
5
6  int main(void) {
7     // initialize two char pointers
8     const char *string1 = "The value is 3.14159";
9     const char *string2 = "1234567890";
10
11    printf("string1 = %s\nstring2 = %s\n\n%s\n%s%zu\n", string1, string2,
12       "The length of the initial segment of string1",
13       "containing no characters from string2 = ",
14       strcspn(string1, string2));
15 }
```

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

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

**Fig. 8.15** | Using function `strcspn`.

### 8.8.3 Function `strpbrk`

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, it returns `NULL`. Figure 8.16 locates the first occurrence in `string1` of any character from `string2`.

```c
// fig08_16.c
// Using function strpbrk
#include <stdio.h>
#include <string.h>

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

   printf("%s\"%s\"\n'%c'%s \"%s\"\n",
      "Of the characters in ", string2, *strpbrk(string1, string2),
      " appears earliest in ", string1);
}
```

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

**Fig. 8.16** | Using function `strpbrk`.

### 8.8.4 Function `strrchr`

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, it returns `NULL`. Figure 8.17 searches for the last occurrence of the character `'z'` in the string `"A zoo has many animals including zebras"`.

```c
// fig08_17.c
// Using function strrchr
#include <stdio.h>
#include <string.h>

int main(void) {
   const char *s1 = "A zoo has many animals including zebras";
   int c = 'z'; // character to search for

   printf("%s '%c' %s\n\"%s\"\n",
      "Remainder of s1 beginning with the last occurrence of character",
      c, "is:", strrchr(s1, c));
}
```

```
Remainder of s1 beginning with the last occurrence of character 'z' is:
"zebras"
```

**Fig. 8.17** | Using function `strrchr`.

### 8.8.5 Function `strspn`

Function **`strspn`** (Fig. 8.18) determines the length of the initial part of its first argument containing only characters from the string in its second argument. The function returns the length of the segment.

```c
// fig08_18.c
// Using function strspn
#include <stdio.h>
#include <string.h>

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

   printf("string1 = %s\nstring2 = %s\n\n%s\n%s%zu\n", string1, string2,
      "The length of the initial segment of string1",
      "containing only characters from string2 = ",
      strspn(string1, string2));
}
```

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

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

**Fig. 8.18** | Using function `strspn`.

### 8.8.6 Function `strstr`

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, `strstr` returns a pointer to the second string's location in the first. Figure 8.19 uses `strstr` to find the string `"def"` in the string `"abcdefabcdef"`.

```c
// fig08_19.c
// Using function strstr
#include <stdio.h>
#include <string.h>

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

   printf("string1 = %s\nstring2 = %s\n\n%s\n%s%s\n", string1, string2,
      "The remainder of string1 beginning with the",
      "first occurrence of string2 is: ", strstr(string1, string2));
}
```

```
string1 = abcdefabcdef
string2 = def

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

**Fig. 8.19** | Using function `strstr`.

### 8.8.7 Function `strtok`

Function **`strtok`** (Fig. 8.20) breaks a string into a series of **tokens**—also called tokenizing the string. A token is a sequence of characters separated by **delimiters**, such as *spaces* or *punctuation marks*. A delimiter can be any character. For example, in a line of text, each word is a token, and the spaces and punctuation separating the words are delimiters. You can change the delimiter string in each `strtok` call. Figure 8.20 tokenizes the string `"This is a sentence with 7 tokens"` and prints the tokens. Function `strtok` *modifies the input string* by placing `'\0'` at the end of each token, so copy the string if you intend to use it after the calls to `strtok`. See CERT recommendation STR06-C for the problems with assuming that `strtok` does not modify the string in its first argument

SEC

```
1  // fig08_20.c
2  // Using function strtok
3  #include <stdio.h>
4  #include <string.h>
5
6  int main(void) {
7     char string[] = "This is a sentence with 7 tokens";
8
9     printf("The string to be tokenized is:\n%s\n\n", string);
10    puts("The tokens are:");
11
12    char *tokenPtr = strtok(string, " "); // begin tokenizing sentence
13
14    // continue tokenizing sentence until tokenPtr becomes NULL
15    while (tokenPtr != NULL) {
16       printf("%s\n", tokenPtr);
17       tokenPtr = strtok(NULL, " "); // get next token
18    }
19 }
```

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

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

**Fig. 8.20** | Using function `strtok`.

#### First `strtok` Call

Multiple calls to `strtok` are required to tokenize a string, assuming it contains more than one token. The first call to `strtok` (line 12) receives as arguments a string to tokenize and a string containing characters that separate the tokens. The statement

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

assigns `tokenPtr` a pointer to the first token in `string`. The second argument, `" "`, indicates that tokens are separated by spaces. Function `strtok` searches for the first character in `string` that's not a delimiter (space). This begins the first token. The function then finds the next delimiter in the string and *replaces it with a null (`'\0'`) character* to terminate the current token. Function `strtok` saves a pointer to the character following that token in `string` and returns a pointer to the current token.

### Subsequent `strtok` Calls

Subsequent `strtok` calls in line 17 continue tokenizing `string`. These calls receive *`NULL` as their first argument* to indicate that they should continue tokenizing from the location in `string` saved by the last call. If no tokens remain, `strtok` returns `NULL`.

## Self Check

**1** ***(Multiple Choice)*** Which function is described by "Locates the first occurrence in string `s1` of string `s2`—if the string is found, the function returns a pointer to the string in `s1`; otherwise, it returns `NULL`"?

a) `strpbrk`.
b) `strstr`.
c) `strspn`.
d) `strcspn`.

**Answer:** b. `strstr`.

**2** ***(Fill-In)*** In the context of function `strtok`, a _____ is a sequence of characters separated by delimiters.

**Answer:** token.

## 8.9 Memory Functions of the String-Handling Library

The string-handling library functions in this section manipulate, compare and search blocks of memory. These functions treat memory as character arrays and can manipulate any block of data. The following table summarizes the string-handling library's memory functions. In the function discussions, "object" refers to a block of data.

| Function prototype | Function description |
|---|---|
| `void *memcpy(void *s1, const void *s2, size_t n);` | Copies `n` bytes from the object pointed to by `s2` into the object pointed to by `s1`, then returns a pointer to the resulting object. |
| `void *memmove(void *s1, const void *s2, size_t n);` | Copies `n` bytes from the object pointed to by `s2` into the object pointed to by `s1`. The copy is performed as if the bytes were first copied from the object pointed to by `s2` into a temporary array and then from the temporary array into the object pointed to by `s1`. A pointer to the resulting object is returned. |

| Function prototype | Function description |
|---|---|
| `int memcmp(const void *s1, const void *s2, size_t n);` | Compares the first `n` bytes of the objects pointed to by `s1` and `s2`. The function returns `0`, less than `0` or greater than `0` if `s1` is equal to, less than or greater than `s2`. |
| `void *memchr(const void *s, int c, size_t n);` | Locates the first occurrence of `c` (converted to `unsigned char`) in the first `n` bytes of the object pointed to by `s`. If `c` is found, `memchr` returns a pointer to `c` in the object; otherwise, it returns `NULL`. |
| `void *memset(void *s, int c, size_t n);` | Copies `c` (converted to `unsigned char`) into the first `n` bytes of the object pointed to by `s`, then returns a pointer to the result. |

The pointer parameters are declared `void *`, so they can be used to manipulate memory for any data type. Recall from Chapter 7 that any pointer can be assigned directly to a `void *` pointer, and a `void *` pointer can be assigned directly to a pointer of any other type. Because a `void *` pointer cannot be dereferenced, each function receives a size argument that specifies the number of bytes the function will process. For simplicity, the examples in this section manipulate character arrays (blocks of characters). The preceding table's functions *do not* check for terminating null characters because they manipulate blocks of memory that are not necessarily strings.

### 8.9.1 Function `memcpy`

Function `memcpy` copies a specified number of bytes from the object pointed to by its second argument into the one pointed to by its first argument. The function can receive a pointer to any type of object. Its result is *undefined* if the two objects overlap in memory—that is, they're parts of the same object. In such cases, use `memmove` instead. Figure 8.21 uses `memcpy` to copy the string in array `s2` to array `s1`. Function `memcpy` is more efficient than `strcpy` when you know the size of the string you're copying.

PERF

```
1  // fig08_21.c
2  // Using function memcpy
3  #include <stdio.h>
4  #include <string.h>
5
6  int main(void) {
7     char s1[17] = "";
8     char s2[] = "Copy this string";
9
10    memcpy(s1, s2, 17); // 17 so we copy s2's terminating \0
11    puts("After s2 is copied into s1 with memcpy, s1 contains:");
12    puts(s1);
13 }
```

**Fig. 8.21** | Using function `memcpy`. (Part 1 of 2.)

```
After s2 is copied into s1 with memcpy, s1 contains:
Copy this string
```

**Fig. 8.21** | Using function `memcpy`. (Part 2 of 2.)

### 8.9.2 Function `memmove`

Like `memcpy`, function **`memmove`** copies a specified number of bytes from the object pointed to by its second argument into the object pointed to by its first argument. Copying is performed as if the bytes were copied from the second argument into a temporary array, then copied from the temporary array into the first argument. This allows bytes from one part of a string (or block of memory) to be copied into another part of the *same* string (or block of memory), even if the two portions overlap. Other than `memmove`, string-manipulation functions that copy characters have undefined results when copying between parts of the same string. Figure 8.22 uses `memmove` to copy the last 10 bytes of array `x` into the first 10 bytes of array `x`.

ERR

```
1  // fig08_22.c
2  // Using function memmove
3  #include <stdio.h>
4  #include <string.h>
5
6  int main(void) {
7     char x[] = "Home Sweet Home"; // initialize char array x
8
9     printf("The string in array x before memmove is: %s\n", x);
10    printf("The string in array x after memmove is: %s\n",
11       (char *) memmove(x, &x[5], 10));
12 }
```

```
The string in array x before memmove is: Home Sweet Home
The string in array x after memmove is: Sweet Home Home
```

**Fig. 8.22** | Using function `memmove`.

### 8.9.3 Function `memcmp`

Function **`memcmp`** (Fig. 8.23) compares the specified number of bytes of its first argument with its second argument's corresponding bytes. The function returns a value greater than 0 if the first argument is greater than the second, 0 if the arguments are equal or a value less than 0 if the first argument is less than the second.

```
1  // fig08_23.c
2  // Using function memcmp
3  #include <stdio.h>
4  #include <string.h>
```

**Fig. 8.23** | Using function `memcmp`. (Part 1 of 2.)

```
5
6  int main(void) {
7     char s1[] = "ABCDEFG";
8     char s2[] = "ABCDXYZ";
9
10    printf("s1 = %s\ns2 = %s\n\n%s%2d\n%s%2d\n%s%2d\n", s1, s2,
11       "memcmp(s1, s2, 4) = ", memcmp(s1, s2, 4),
12       "memcmp(s1, s2, 7) = ", memcmp(s1, s2, 7),
13       "memcmp(s2, s1, 7) = ", memcmp(s2, s1, 7));
14 }
```

```
s1 = ABCDEFG
s2 = ABCDXYZ

memcmp(s1, s2, 4) =  0
memcmp(s1, s2, 7) = -1
memcmp(s2, s1, 7) =  1
```

**Fig. 8.23** | Using function `memcmp`. (Part 2 of 2.)

### 8.9.4 Function `memchr`

Function `memchr` searches for the first occurrence of a byte, represented as `unsigned char`, in the specified number of bytes of an object. If the byte is found, `memchr` returns a pointer to the byte in the object; otherwise, it returns `NULL`. Figure 8.24 searches for the byte containing `'r'` in the string `"This is a string"`.

```
1  // fig08_24.c
2  // Using function memchr
3  #include <stdio.h>
4  #include <string.h>
5
6  int main(void) {
7     const char *s = "This is a string";
8
9     printf("The remainder of s after character 'r' is found is \"%s\"\n",
10       (char *) memchr(s, 'r', 16));
11 }
```

```
The remainder of s after character 'r' is found is "ring"
```

**Fig. 8.24** | Using function `memchr`.

### 8.9.5 Function `memset`

Function `memset` copies the value of the byte in its second argument into the first *n* bytes of the object pointed to by its first argument, where *n* is specified by the third argument. You can use `memset` to set an array's elements to `0` rather than assigning `0` to each element. For example, a five-element `int` array `n` could be reset to `0`s with

PERF

```
memset(n, 0, 5);
```

Many hardware architectures have a block copy or clear instruction that the compiler can use to optimize `memset` for high-performance zeroing of memory. Figure 8.25 uses `memset` to copy `'b'` into the first `7` bytes of `string1`.

```
1  // fig08_25.c
2  // Using function memset
3  #include <stdio.h>
4  #include <string.h>
5
6  int main(void) {
7     char string1[15] = "BBBBBBBBBBBBBB";
8
9     printf("string1 = %s\n", string1);
10    printf("string1 after memset = %s\n", (char *) memset(string1, 'b', 7));
11 }
```

```
string1 = BBBBBBBBBBBBBB
string1 after memset = bbbbbbbBBBBBBB
```

**Fig. 8.25** | Using function `memset`.

## Self Check

**1** *(Multiple Choice)* Which of the following statements about function `memcpy` is *false*?

a) The function copies a specified number of bytes from the object pointed to by its second argument into the object pointed to by its first argument.
b) The function can receive a pointer to any type of object.
c) The result of this function is undefined if the two objects are completely separate in memory.
d) The function is more efficient than `strcpy` when you know the size of the string you're copying.

**Answer:** c) is *false*. Actually, the result is undefined if the two objects overlap in memory—that is, they're parts of the same object. In such cases, use `memmove`.

**2** *(Fill-In)* The memory-handling functions of the string-handling library manipulate, compare and search blocks of memory, which the functions treat as _____.
**Answer:** character arrays.

**3** *(True/False)* Function `memmove` copies a specified number of bytes from the object pointed to by its second argument into the object pointed to by its first argument. Copying is performed as if the bytes were copied from the second argument into a temporary array, then copied from the temporary array into the first argument. This allows bytes from one block of memory to be copied into another part of the *same* block of memory, even if the two portions overlap.
**Answer:** *True*.

## 8.10 Other Functions of the String-Handling Library

The two remaining string-handling library functions are `strerror` and `strlen`, which are summarized in the following table.

| Function prototype | Function description |
|---|---|
| `char *strerror(int errornum);` | Maps `errornum` to a full text string in a compiler- and locale-specific manner and returns the string. Error numbers are defined in `errno.h`. |
| `size_t strlen(const char *s);` | Returns the length of string `s`—that is, the number of characters preceding the string's terminating null character. |

### 8.10.1 Function `strerror`

Function `strerror` takes an error number and creates an error message string. A pointer to the string is returned. Figure 8.26 demonstrates `strerror`.

```
1  // fig08_26.c
2  // Using function strerror
3  #include <stdio.h>
4  #include <string.h>
5
6  int main(void) {
7     printf("%s\n", strerror(2));
8  }
```

```
No such file or directory
```

**Fig. 8.26** | Using function `strerror`.

### 8.10.2 Function `strlen`

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. Figure 8.27 demonstrates function `strlen`.

```
1  // fig08_27.c
2  // Using function strlen
3  #include <stdio.h>
4  #include <string.h>
5
6  int main(void) {
7     const char *string1 = "abcdefghijklmnopqrstuvwxyz";
8     const char *string2 = "four";
9     const char *string3 = "Boston";
```

**Fig. 8.27** | Using function `strlen`. (Part 1 of 2.)

```
10
11     printf("%s\"%s\"%s%zu\n%s\"%s\"%s%zu\n%s\"%s\"%s%zu\n",
12        "The length of ", string1, " is ", strlen(string1),
13        "The length of ", string2, " is ", strlen(string2),
14        "The length of ", string3, " is ", strlen(string3));
15  }
```

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

**Fig. 8.27** | Using function `strlen`. (Part 2 of 2.)

## Self Check

**1** *(True/False)* The error message strings returned by function `strerror` are uniform across platforms.
**Answer:** *False*. The messages vary by compiler and locale.

**2** *(True/False)* Function `strlen` takes a string as an argument and returns the number of characters in the string, including the terminating null character.
**Answer:** *False*. Actually, the terminating null character is not included in the length.

# 8.11 Secure C Programming

### Secure String-Processing Functions

Earlier Secure C Programming sections covered C11's functions `printf_s` and `scanf_s`. This chapter presented functions `sprintf`, `strcpy`, `strncpy`, `strcat`, `strncat`, `strtok`, `strlen`, `memcpy`, `memmove` and `memset`. The C11 standard's *optional* Annex K provides versions of these and many other string-processing and input/output functions. If your C compiler supports Annex K, consider using the secure versions of these functions. Among other things, the Annex K versions help prevent buffer overflows by requiring additional parameters that specify the number of elements in a target array and by ensuring that pointer arguments are non-`NULL`.

### Reading Numeric Inputs and Input Validation

It's important to validate the data that you input into a program. For example, when you ask the user to enter an `int` in the range 1–100, then attempt to read that `int` using `scanf`, there are several possible problems. The user could enter:

- an `int` that's outside the program's required range (such as 102).
- an `int` that's outside that computer's allowed range for `int`s (such as 8,000,000,000 on a machine with 32-bit `int`s).
- a non-integer numeric value (such as 27.43).
- a non-numeric value (such as FOVR).

You can use various functions that you learned in this chapter to fully validate such input. For example, you could

- use `fgets` to read the input as a line of text,
- convert the string to a number using `strtol` and ensure that the conversion was successful, then
- ensure that the value is in range.

For more information and techniques for converting input to numeric values, see CERT guideline INT05-C at `https://wiki.sei.cmu.edu/`.

## Self Check

**1** ***(Fill-In)*** Among other things, the secure string-processing functions of Annex K help prevent buffer _____ by requiring additional parameters that specify the number of elements in a target array and by ensuring that pointer arguments are non-`NULL`.
**Answer:** overflows.

**2** ***(True/False)*** It's important to validate the data you input into a program. You can use various string-and-character-processing functions to fully validate inputs. For example, you could use `fgets` to read the input as a line of text, convert the string to a number using `strtol`, ensure the conversion was successful, then ensure that the value is in range.
**Answer:** *True*.

## Summary

### Section 8.2 Fundamentals of Strings and Characters

- **Characters** are the fundamental building blocks of source programs. Every program is composed of a sequence of characters that—when grouped together meaningfully—is interpreted by the computer as instructions used to accomplish a task.
- A **character constant** (p. 442) is an `int` value represented as a character in single quotes. The value of a character constant is the character's integer value in the machine's **character set** (p. 443).
- A **string** (p. 443) is a series of characters treated as a single unit. A string may include letters, digits and various special characters (p. 443) such as +, -, *, / and $. String literals, or string constants, are written in double quotation marks.
- A string in C is an **array of characters** ending in the **null character** (p. 443; `'\0'`).
- A string is accessed via a **pointer** to its first character. The value of a string is the **address** of its first character.
- A **character array** or a **variable of type `char *`** can be initialized with a string in a definition.
- When defining a character array to contain a string, the array must be large enough to store the string and its terminating null character.
- A string can be stored in an array using `scanf`. Function `scanf` will read characters until a space, tab, newline or end-of-file indicator is encountered.
- For a character array to be printed as a string, the array must contain a terminating null character.

## Section 8.3 Character-Handling Library

- Function **isdigit** (p. 445) determines whether its argument is a **digit** (0–9).
- Function **isalpha** (p. 445) determines whether its argument is an **uppercase letter** (A–Z) or a **lowercase letter** (a–z).
- Function **isalnum** (p. 445) determines whether its argument is an **uppercase letter** (A–Z), a **lowercase letter** (a–z) or a **digit** (0–9).
- Function **isxdigit** (p. 445) determines whether its argument is a **hexadecimal digit** (p. 445; A–F, a–f, 0–9).
- Function **islower** (p. 447) determines whether its argument is a **lowercase letter** (a–z).
- Function **isupper** (p. 447) determines whether its argument is an **uppercase letter** (A–Z).
- Function **toupper** (p. 447) converts a lowercase letter to uppercase and returns it.
- Function **tolower** (p. 447) converts an uppercase letter to lowercase and returns it.
- Function **isspace** (p. 448) determines whether its argument is one of the following **whitespace characters**: ' ' (space), '\f', '\n', '\r', '\t' or '\v'.
- Function **iscntrl** (p. 448) determines whether its argument is one of the following **control characters**: '\t', '\v', '\f', '\a', '\b', '\r' or '\n'.
- Function **ispunct** (p. 448) determines whether its argument is a **printing character** other than a space, a digit or a letter.
- Function **isprint** (p. 448) determines whether its argument is any printing character, including the space character.
- Function **isgraph** (p. 448) determines whether its argument is a printing character other than the space character.

## Section 8.4 String-Conversion Functions

- Function **strtod** (p. 450) converts a sequence of characters representing a floating-point value to double. The location specified by its pointer to char * argument is assigned the remainder of the string after the conversion, or to the entire string if no portion of the string can be converted.
- Function **strtol** (p. 451) converts a sequence of characters representing an integer to long. This function works identically to strtod, but the third argument specifies the base of the value being converted.
- Function **strtoul** (p. 452) works identically to strtol but converts a sequence of characters representing an integer to unsigned long int.

## Section 8.5 Standard Input/Output Library Functions

- Function **fgets** (p. 453) reads characters until a newline character or the end-of-file indicator is encountered. The arguments to fgets are an array of type char, the maximum number of characters to read and the stream from which to read. A null character ('\0') is appended to the array after reading terminates. If a newline is encountered, it's included in the input string.
- Function **putchar** (p. 453) prints its character argument.
- Function **getchar** (p. 455) reads a single character from the standard input and returns it as an integer. If the end-of-file indicator is encountered, getchar returns EOF.
- Function **puts** (p. 455) takes a string (char *) as an argument and prints the string followed by a newline character.
- Function **sprintf** (p. 455) uses the same conversion specifications as function printf to print formatted data into an array of type char.

- Function **sscanf** (p. 456) uses the same conversion specifications as function scanf to read formatted data from a string.

## Section 8.6 String-Manipulation Functions of the String-Handling Library

- Function **strcpy** copies its second argument string into its first argument char array. You must ensure that the array is large enough to store the string and its terminating null character.
- Function **strncpy** (p. 458) is equivalent to strcpy, but specifies the maximum number of characters to copy from the string into the array. The terminating null character will be copied only if the number of characters to be copied is one more than the string's length.
- Function **strcat** (p. 459) appends its second argument string—including its terminating null character—to its first argument string. The first character of the second string replaces the null ('\0') character of the first string. You must ensure that the array used to store the first string is large enough to store both the first string and the second string.
- Function **strncat** (p. 458) appends a specified number of characters from the second string to the first string. A terminating null character is appended to the result.

## Section 8.7 Comparison Functions of the String-Handling Library

- Function **strcmp** (p. 460) compares its first string argument to its second string argument, character by character. It returns 0 if the strings are equal, a negative value if the first string is less than the second or a positive value if the first string is greater than the second.
- Function **strncmp** (p. 460) is equivalent to strcmp, except that strncmp compares a specified number of characters. If one of the strings is shorter than the number of characters specified, strncmp compares characters until the null character in the shorter string is encountered.

## Section 8.8 Search Functions of the String-Handling Library

- Function **strchr** (p. 463) searches for the first occurrence of a character in a string. If found, strchr returns a pointer to the character in the string; otherwise, strchr returns NULL.
- Function **strcspn** (p. 464) 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 segment's length.
- Function **strpbrk** (p. 464) searches for the first occurrence in its first argument of any character in its second argument. If a character from the second argument is found, strpbrk returns a pointer to the character; otherwise, strpbrk returns NULL.
- Function **strrchr** (p. 465) searches for the last occurrence of a character in a string. If found, strrchr returns a pointer to the character in the string; otherwise, strrchr returns NULL.
- Function **strspn** (p. 466) 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.
- Function **strstr** (p. 466) 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.
- A sequence of calls to **strtok** (p. 467) breaks its first string argument into tokens (p. 467) that are separated by characters contained in the second string argument. The first call contains the string to tokenize as the first argument. Subsequent calls to continue tokenizing that 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, it returns NULL.

### Section 8.9 Memory Functions of the String-Handling Library

- Function **memcpy** (p. 469) copies a specified number of bytes from the object to which its second argument points into the object to which its first argument points. The function can receive a pointer to any type of object.
- Function **memmove** (p. 470) copies a specified number of bytes from the object pointed to by its second argument to the object pointed to by its first argument. Copying is accomplished as if the bytes were copied from the second argument to a temporary array and then copied from the temporary array to the first argument.
- Function **memcmp** (p. 470) compares the specified number of bytes of its first and second arguments.
- Function **memchr** (p. 471) searches for the first occurrence of a byte, represented as `unsigned char`, in the specified number of bytes of an object. If the byte is found, a pointer to the byte is returned; otherwise, a `NULL` pointer is returned.
- Function **memset** (p. 471) copies its second argument, treated as an `unsigned char`, to a specified number of bytes of the object pointed to by the first argument.

### Section 8.10 Other Functions of the String-Handling Library

- Function **strerror** (p. 473) maps an integer error number into a full text string in a locale-specific manner. A pointer to the string is returned.
- Function **strlen** (p. 473) 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 of the string.

## Self-Review Exercises

**8.1** Write a single statement to accomplish each of the following. Assume variable `c` is a `char`, variables `x`, `y` and `z` are `int`s, variables `d`, `e` and `f` are `double`s, variable `ptr` is a `char *` and `s1` and `s2` are 100-element `char` arrays.

a) Convert the character stored in variable `c` to an uppercase letter. Assign the result to variable `c`.
b) Determine whether the value of variable `c` is a digit. Use the conditional operator as shown in Figs. 8.1–8.3 to print `" is a "` or `" is not a "` when the result is displayed.
c) Determine whether the value of variable `c` is a control character. Use the conditional operator to print `" is a "` or `" is not a "` when the result is displayed.
d) Read a line of text into array `s1` from the keyboard. Do not use `scanf`.
e) Print the line of text stored in array `s1`. Do not use `printf`.
f) Assign `ptr` the location of the last occurrence of `c` in `s1`.
g) Print the value of variable `c`. Do not use `printf`.
h) Determine whether the value of `c` is a letter. Use the conditional operator to print `" is a "` or `" is not a "` when the result is displayed.
i) Read a character from the keyboard and store the character in variable `c`.
j) Assign `ptr` the location of the first occurrence of `s2` in `s1`.
k) Determine whether the value of variable `c` is a printing character. Use the conditional operator to print `" is a "` or `" is not a "` when the result is displayed.

l) Read three `double` values into variables `d`, `e` and `f` from the string `"1.27 10.3 9.432"`.
m) Copy the string stored in array `s2` into array `s1`.
n) Assign `ptr` the location of the first occurrence in `s1` of any character from `s2`.
o) Compare the string in `s1` with the string in `s2`. Print the result.
p) Assign `ptr` the location of the first occurrence of `c` in `s1`.
q) Use `sprintf` to print the values of integer variables `x`, `y` and `z` into array `s1`. Each value should be printed with a field width of `7`.
r) Append 10 characters from the string in `s2` to the string in `s1`.
s) Determine the length of the string in `s1`. Print the result.
t) Assign `ptr` to the location of the first token in `s2`. Tokens in the string `s2` are separated by commas (`,`).

**8.2** Show two different ways to initialize `char` array `vowel` with the string `"AEIOU"`.

**8.3** What, if anything, prints when each of the following C statements is performed? If the statement contains an error, describe the error and indicate how to correct it. Assume the following variable definitions:

```
char s1[50] = "jack";
char s2[50] = "jill";
char s3[50] = "";
```

a) `printf("%c%s", toupper(s1[0]), &s1[1]);`
b) `printf("%s", strcpy(s3, s2));`
c) `printf("%s", strcat(strcat(strcpy(s3, s1), " and "), s2));`
d) `printf("%zu", strlen(s1) + strlen(s2));`
e) `printf("%zu", strlen(s3)); // using s3 after part (c) executes`

**8.4** Find the error in each of the following and explain how to correct it:

a)
```
char s[10] = "";
strncpy(s, "hello", 5);
printf("%s\n", s);
```
b) `printf("%s", 'a');`
c)
```
char s[12] = "";
strcpy(s, "Welcome Home");
```
d)
```
if (strcmp(string1, string2)) {
   puts("The strings are equal");
}
```

## Answers to Self-Review Exercises

**8.1** See the answers below:

a) `c = toupper(c);`
b) `printf("'%c'%sdigit\n", c, isdigit(c) ? " is a " : " is not a ");`
c)
```
printf("'%c'%scontrol character\n",
   c, iscntrl(c) ? " is a " : " is not a ");
```
d) `fgets(s1, 100, stdin);`

e) `puts(s1);`
f) `ptr = strrchr(s1, c);`
g) `putchar(c);`
h) `printf("'%c'%sletter\n", c, isalpha(c) ? " is a " : " is not a ");`
i) `c = getchar();`
j) `ptr = strstr(s1, s2);`
k)
```
printf("'%c'%sprinting character\n",
   c, isprint(c) ? " is a " : " is not a ");
```
l) `sscanf("1.27 10.3 9.432", "%f%f%f", &d, &e, &f);`
m) `strcpy(s1, s2);`
n) `ptr = strpbrk(s1, s2);`
o) `printf("strcmp(s1, s2) = %d\n", strcmp(s1, s2));`
p) `ptr = strchr(s1, c);`
q) `sprintf(s1, "%7d%7d%7d", x, y, z);`
r) `strncat(s1, s2, 10);`
s) `printf("strlen(s1) = %zu\n", strlen(s1));`
t) `ptr = strtok(s2, ",");`

**8.2**
```
char vowel[] = "AEIOU";
char vowel[] = {'A', 'E', 'I', 'O', 'U', '\0'};
```

**8.3** See the answers below:
a) `Jack`
b) `jill`
c) `jack and jill`
d) `8`
e) `13`

**8.4** See the answers below:
a) Error: Function `strncpy` does not write a terminating null character to array `s`, because its third argument is equal to the length of the string `"hello"`.
Correction: Make the third argument of `strncpy` `6`, or assign `'\0'` to `s[5]`.
b) Error: Attempting to print a character constant as a string.
Correction: Use `%c` to output the character, or replace `'a'` with `"a"`.
c) Error: Character array `s` is not large enough to store the terminating null character.
Correction: Declare the array with more elements.
d) Error: Function `strcmp` returns 0 if the strings are equal; therefore, the condition in the `if` statement is false, and the `printf` will not execute.
Correction: Compare the result of `strcmp` with `0` in the condition.

## Exercises

**8.5** ***(Character Testing)*** Write a program that inputs a character from the keyboard and tests it with each of the character-handling library functions. The program should print the value returned by each function.

**8.6** *(Displaying Strings in Alternating Uppercase and Lowercase)* Write a program that inputs a line of text into `char` array `s[100]`. Output the line in alternating uppercase letters and lowercase letters.

**8.7** *(Converting Strings to Integers for Calculations)* Write a program that inputs six strings representing integers, converts the strings to integers, and calculates the sum and average of the six values.

**8.8** *(Converting Strings to Floating Point for Calculations)* Write a program that inputs six strings that represent floating-point values, converts the strings to double values, stores the values into a double array, and calculates the sum and average of the values.

**8.9** *(Concatenating Strings)* Write a program that uses function `strcat` to concatenate two strings input by the user. The program should print the strings before and after concatenating as well as the length of the concatenated string.

**8.10** *(Appending Part of a String)* Write a program that uses function `strncat` to append part of a string to another string. The program should input the strings, and the number of characters to be appended, then display the first string and its length after the second string was appended.

**8.11** *(Random Sentences)* Use random-number generation to create sentences. Your program should use four arrays of pointers to `char` called `article`, `noun`, `verb` and `preposition`. Create a sentence by selecting a word at random from each array in the following order: `article`, `noun`, `verb`, `preposition`, `article` and `noun`. The arrays should be filled as follows: The `article` array should contain the articles `"the"`, `"a"`, `"one"`, `"some"` and `"any"`; the `noun` array should contain the nouns `"boy"`, `"girl"`, `"dog"`, `"town"` and `"car"`; the `verb` array should contain the verbs `"drove"`, `"jumped"`, `"ran"`, `"walked"` and `"skipped"`; the `preposition` array should contain the prepositions `"to"`, `"from"`, `"over"`, `"under"` and `"on"`.

As each word is picked, concatenate it to the previous words in an array large enough to hold the entire sentence. Separate the words by spaces. The final sentence should start with a capital letter and end with a period. Generate 20 such sentences. Modify your program to produce a short story consisting of several of these sentences. (How about the possibility of a random term-paper writer?)

**8.12** *(Limericks)* A limerick is a humorous five-line verse in which the first and second lines rhyme with the fifth, and the third line rhymes with the fourth. Using techniques similar to those developed in Exercise 8.11, write a program that produces random limericks. Polishing this program to produce good limericks is a challenging problem, but the result will be worth the effort!

**8.13** *(Pig Latin)* Write a program that encodes English-language phrases into pig Latin. Pig Latin is a form of coded language often used for amusement. Many variations exist in the methods used to form pig-Latin phrases. For simplicity, use the following algorithm:

To form a pig-Latin phrase from an English-language phrase, tokenize the phrase into words with function `strtok`. To translate each English word into a pig-Latin

word, place the first letter of the English word at the end of the English word and add the letters "ay". Thus the word "jump" becomes "umpjay", the word "the" becomes "hetay" and the word "computer" becomes "omputercay". Blanks between words remain as blanks. Assume the following: The English phrase consists of words separated by blanks, there are no punctuation marks, and all words have two or more letters. Function `printLatinWord` should display each word. [*Hint*: Each time `strtok` finds a token, pass the token pointer to function `printLatinWord`, and print the pig-Latin word. We've provided simplified pig-Latin conversion rules here. For more detailed rules and variations, visit `https://en.wikipedia.org/wiki/Pig_latin`.]

**8.14** ***(Tokenizing Telephone Numbers)*** Write a program that inputs a telephone number as a string in the form `(555) 555-5555`. Use function `strtok` to extract as tokens the area code, the first three digits of the phone number and the last four digits of the phone number. Concatenate the phone number's seven digits into one string. Convert the area-code string and phone-number string to integers, then display both.

**8.15** ***(Displaying a Sentence with Its Words Reversed)*** Write a program that inputs a line of text, tokenizes the line with function `strtok` and outputs the tokens in reverse order.

**8.16** ***(Searching for Substrings)*** Write a program that inputs a line of text and a search string from the keyboard. Using function `strstr`, locate the first occurrence of the search string in the line of text. Assign the location to variable `searchPtr` of type `char *`. If the search string is found, print the remainder of the line of text beginning with the search string. Then, use `strstr` again to locate the next occurrence of the search string in the line of text. If a second occurrence is found, print the remainder of the line of text beginning with the second occurrence. [*Hint*: The second call to `strstr` should contain `searchPtr + 1` as its first argument.]

**8.17** ***(Counting the Occurrences of a Substring)*** Write a program based on Exercise 8.16 that inputs several lines of text and a search string and uses function `strstr` to determine the total occurrences of the search string in the lines of text. Print the result.

**8.18** ***(Counting Occurrences of Various Characters in a String)*** Write a program that inputs a line of text and counts the total numbers of vowels, consonants, digits, and white spaces in the given line of text.

**8.19** ***(Removing a Particular Word from a Given Line of Text)*** Write a program that inputs a line of text and a given word. The program should use string library functions `strcmp` and `strcpy` to remove all occurrences of the given word from the input line of text. The program should also count the number of words in the given line of text before and after removing the given word using the `strtok` function.

**8.20** ***(Counting the Number of Words in a String)*** Write a program that inputs several lines of text and uses `strtok` to count the total number of words. Assume that the words are separated by either spaces or newline characters.

**8.21** *(Alphabetizing a List of Strings)* Use the string-comparison functions and the techniques for sorting arrays to write a program that alphabetizes a list of strings. Use the names of 10 or 15 towns in your area as data for your program.

**8.22** Appendix B shows the numeric code representations for the ASCII character set. Study Appendix B, then state whether each of the following is *true* or *false*.

a) The letter `"A"` comes before the letter `"B"`.
b) The digit `"9"` comes before the digit `"0"`.
c) The commonly used symbols for addition, subtraction, multiplication and division all come before any of the digits.
d) The digits come before the letters.
e) If a sort program sorts strings into ascending sequence, then the program will place the symbol for a right parenthesis before the symbol for a left parenthesis.

**8.23** *(Strings Starting with `"Th"`)* Write a program that reads a series of strings and prints only those beginning with `"Th"`.

**8.24** *(Strings Ending with `"tion"`)* Write a program that reads a series of strings and prints only those that end with the letters `"tion"`.

**8.25** *(Printing Letters for Various ASCII Codes)* Write a program that inputs an ASCII code and prints the corresponding character.

**8.26** *(Write Your Own Character-Handling Functions)* Using the ASCII character chart in Appendix B as a guide, write your own versions of the character-handling functions in Section 8.3.

**8.27** *(Write Your String-Conversion Functions)* Write your own versions of the functions in Section 8.4 for converting strings to numbers.

**8.28** *(Write Your Own String-Copy and String-Concatenation Functions)* Write two versions of each string-copy and string-concatenation function in Section 8.6. The first version should use array indexing, and the second should use pointers and pointer arithmetic.

**8.29** *(Write Your Own String-Comparison Functions)* Write two versions of each string-comparison function in Fig. 8.13. The first version should use array indexing, and the second should use pointers and pointer arithmetic.

**8.30** *(Write Your Own String-Length Function)* Write two versions of function `strlen` in Fig. 8.27. The first version should use array indexing, and the second should use pointers and pointer arithmetic.

## Special Section: Advanced String-Manipulation Exercises

The preceding exercises are keyed to the text and designed to test the reader's understanding of fundamental string-manipulation concepts. This section contains intermediate and advanced problems that you should find challenging yet enjoyable. They vary considerably in difficulty. Some require an hour or two of programming.

Others are useful for lab assignments that might require two or three weeks of study and implementation. Some are challenging term projects.

**8.31** ***(Text Analysis)*** String-manipulation capabilities enable some rather interesting approaches to analyzing the writings of great authors. Much attention has been focused on whether William Shakespeare ever lived. Some scholars find substantial evidence that Christopher Marlowe actually penned the masterpieces attributed to Shakespeare. Researchers have used computers to find similarities in the writings of these two authors. This exercise examines three methods for analyzing texts with a computer.

a) Write a program that reads several lines of text and prints a table indicating the number of occurrences of each letter of the alphabet in the text. For example, the following phrase contains one "a," two "b's," no "c's," and so on:

```
To be, or not to be: that is the question:
```

b) Write a program that reads several lines of text and prints a table indicating the number of one-letter words, two-letter words, three-letter words, and so on, appearing in the text. For example, the phrase

```
Whether 'tis nobler in the mind to suffer
```

contains

| Word length | Occurrences |
|---|---|
| 1 | 0 |
| 2 | 2 |
| 3 | 1 |
| 4 | 2 (including 'tis) |
| 5 | 0 |
| 6 | 2 |
| 7 | 1 |

c) Write a program that reads several lines of text and prints a table indicating the number of occurrences of each different word in the text. The program should include the words in the table in the same order in which they appear in the text. For example, the lines

```
To be, or not to be: that is the question:
Whether 'tis nobler in the mind to suffer
```

contain the words `"to"` three times, `"be"` two times, `"or"` once, and so on.

**8.32** ***(Printing Dates in Various Formats)*** Dates are commonly printed in several different formats in business correspondence. Two of the more common formats are

```
07/21/2003 and July 21, 2003
```

Write a program that reads a date in the first format and prints it in the second format.

**8.33** ***(Check Protection)*** Computers are frequently used in check-writing systems, such as payroll and accounts-payable applications. Many stories circulate regarding weekly paychecks being printed (by mistake) for amounts in excess of $1 million. Weird amounts are printed by computerized check-writing systems because of human error and/or machine failure. Systems designers, of course, make every effort to build controls into their systems to prevent erroneous checks from being issued.

Another serious problem is someone intentionally altering a check amount then cashing it fraudulently. To prevent a dollar amount from being altered, most computerized check-writing systems employ a technique called *check protection.*

Checks designed for imprinting by computer contain a fixed number of spaces in which the computer may print an amount. Suppose a paycheck contains nine blank spaces in which the computer is supposed to print the amount of a weekly paycheck. If the amount is large, then all nine of those spaces will be filled—for example:

```
11,230.60  (check amount)
---------
123456789  (position numbers)
```

On the other hand, if the amount is less than $1,000, then several of the spaces will ordinarily be left blank—for example,

```
    99.87
---------
123456789
```

contains four blank spaces. If a check is printed with blank spaces, it's easier for someone to alter the amount of the check. To prevent such alteration, many check-writing systems insert *leading asterisks* to protect the amount as follows:

```
****99.87
---------
123456789
```

Write a program that inputs a dollar amount to be printed on a check and then prints the amount in check-protected format with leading asterisks if necessary. Assume that nine spaces are available for printing an amount.

**8.34** ***(Word Equivalent of a Check Amount)*** Continuing the discussion of the previous exercise: One common check-writing security method requires that the check amount be both written in numbers and "spelled out" in words. Even if someone is able to alter the numerical amount of the check, it's extremely difficult to change the amount in words. Write a program that inputs a numeric check amount and writes the word equivalent of the amount. For example, the amount 52.43 should be written as

```
FIFTY TWO and 43/100
```

**8.35** ***(Project: A Metric Conversion Program)*** Write a program that assists the user with metric conversions. Allow the user to specify the names of the units as strings (i.e., centimeters, liters, grams, and so on for the metric system and inches, quarts, pounds, and so on for the English system) and should respond to simple questions such as

```
"How many inches are in 2 meters?"
"How many liters are in 10 quarts?"
```

Your program should recognize invalid conversions. For example, the following question is not meaningful—"feet" are length units while "kilograms" are mass units.

```
"How many feet are in 5 kilograms?"
```

**8.36** ***(Cooking with Healthier Ingredients)*** Obesity in the United States is increasing at an alarming rate. Check the Centers for Disease Control and Prevention (CDC) webpage at `www.cdc.gov/obesity/data/index.html`, which contains United States obesity data and facts. As obesity increases, so do occurrences of related problems (e.g., heart disease, high blood pressure, high cholesterol, type 2 diabetes). Write a program that helps users choose healthier ingredients when cooking, and helps those allergic to certain foods (e.g., nuts, gluten) find substitutes. The program should read a recipe from the user and suggest healthier replacements for some of the ingredients. For simplicity, your program should assume the recipe has no abbreviations for measures such as teaspoons, cups, and tablespoons, and uses numerical digits for quantities (e.g., 1 egg, 2 cups) rather than spelling them out (one egg, two cups). Some common substitutions are shown in the following table. Your program should display a warning such as, "Always consult your physician before making significant changes to your diet."

| Ingredient | Substitution |
|---|---|
| 1 cup sour cream | 1 cup yogurt |
| 1 cup milk | 1/2 cup evaporated milk and 1/2 cup water |
| 1 teaspoon lemon juice | 1/2 teaspoon vinegar |
| 1 cup sugar | 1/2 cup honey, 1 cup molasses or 1/4 cup agave nectar |
| 1 cup butter | 1 cup margarine or yogurt |
| 1 cup flour | 1 cup rye or rice flour |
| 1 cup mayonnaise | 1 cup cottage cheese or 1/8 cup mayonnaise and 7/8 cup yogurt |
| 1 egg | 2 tablespoons cornstarch, arrowroot flour or potato starch or 2 egg whites or 1/2 of a large banana (mashed) |
| 1 cup milk | 1 cup soy milk |
| 1/4 cup oil | 1/4 cup applesauce |
| white bread | whole-grain bread |

Your program should take into consideration that replacements are not always one-for-one. For example, if a cake recipe calls for three eggs, it might reasonably use six egg whites instead. Conversion data for measurements and substitutes can be obtained at various websites. Your program should consider the user's health concerns, such as high cholesterol, high blood pressure, weight loss, gluten allergy, and so on. For high cholesterol, the program should suggest substitutes for eggs and dairy products; if the user wishes to lose weight, low-calorie substitutes for ingredients such as sugar should be suggested.

**8.37** ***(Spam Scanner)*** Spam (or junk e-mail) costs U.S. organizations billions of dollars a year in spam-prevention software, equipment, network resources, bandwidth,

and lost productivity. Research online some of the most common spam e-mail messages and words, and check your own junk e-mail folder. Create a list of 30 words and phrases commonly found in spam messages. Write a program in which the user enters an e-mail message. Read the message into a large character array and ensure that the program does not attempt to insert characters past the end of the array. Then scan the message for each of the 30 keywords or phrases. For each occurrence of one of these within the message, add a point to the message's "spam score." Next, rate the likelihood that the message is spam, based on the number of points it received.

**8.38** ***(SMS Language)*** Short Message Service (SMS) is a communications service that allows sending text messages of 160 or fewer characters between mobile phones. With the proliferation of mobile phone use worldwide, SMS is being used in many developing nations for political purposes (e.g., voicing opinions and opposition), reporting news about natural disasters, and so on. Because the length of SMS messages is limited, SMS Language—abbreviations of common words and phrases in mobile text messages, e-mails, instant messages, etc.—is often used. For example, "in my opinion" is "IMO" in SMS Language. Research SMS Language online. Write a program that lets the user enter a message using SMS Language, then translates it into English (or your own language). Also provide a mechanism to translate text written in English (or your own language) into SMS Language. One potential problem is that one SMS abbreviation could expand into a variety of phrases. For example, IMO (as used above) could also stand for "International Maritime Organization," "in memory of," etc.

**8.39** ***(Gender Neutrality)*** In Exercise 1.6, you researched eliminating sexism in all forms of communication. You then described the algorithm you'd use to read through a paragraph of text and replace gender-specific words with gender-neutral equivalents. Create a program that reads a paragraph of text, then replaces gender-specific words with gender-neutral ones. Display the resulting gender-neutral text.

## A Challenging String-Manipulation Project

**8.40** ***(Project: A Crossword-Puzzle Generator)*** Most people have worked a crossword puzzle at one time or another, but few have ever attempted to generate one. Generating a crossword puzzle is a difficult problem. It's suggested here as a string-manipulation project requiring substantial sophistication and effort. There are many issues you must resolve to get even the simplest crossword-puzzle generator program working. For example, how does one represent the grid of a crossword puzzle inside the computer? Should one use a series of strings, or perhaps two-dimensional arrays? You need a source of words (i.e., a computerized dictionary) that can be directly referenced by the program. In what form should these words be stored to facilitate the complex manipulations required by the program? The really ambitious reader will want to generate the "clues" portion of the puzzle in which the brief hints for each "across" word and each "down" word are printed for the puzzle worker. Merely printing a version of the blank puzzle itself is not a simple problem.

## Pqyoaf X Nylfomigrob Qwbbfmh Mndogvk: Rboqlrut yua Boklnxhmywex

**8.41** ***(Pqyoaf X Nylfomigrob: Cuzqvbpcxo vlk Adzdujcjjl)*** No doubt, you noticed the section title above and this exercise's title both look like gibberish. This is not a mistake! In this exercise, we continue our focus on security by introducing cryptography. You'll create functions that implement a Vigenère secret-key cipher.[3,4] After encrypting and decrypting your own text, you can use your `decrypt` function with our secret key to decrypt the encrypted titles above.

### Cryptography

Cryptography has been used for thousands of years[5,6] and is critically important in today's connected world. Every day, cryptography is used behind the scenes to ensure that your Internet-based communications are private and secure. For example, most websites (including `deitel.com`) now use the HTTPS protocol to encrypt and decrypt your web interactions.

SEC

### Caesar Cipher

Julius Caesar used a simple substitution cipher to encrypt military communications.[7] Known as the Caesar cipher, his technique replaces every letter in a message with the letter three ahead in the alphabet. So, A is replaced with D, B with E, C with F, … X with A, Y with B and Z with C. Thus, the unencrypted text

```
Caesar Cipher
```

would be encrypted as

```
Fdhvdu Flskhu
```

The encrypted text is known as ciphertext. The unencrypted text is known as plaintext or cleartext.

### Experimenting with Ciphers

For a fun way to play with the Caesar cipher and many other cipher algorithms, visit:

```
https://cryptii.com/pipes/caesar-cipher
```

which is an online implementation of the open-source `cryptii` project:

```
https://github.com/cryptii/cryptii
```

On `cryptii.com`, you can enter plaintext, choose a cipher to use, specify that cipher's settings and view the resulting ciphertext.

3. "Crypto Corner—Vigenère Cipher." Accessed December 23, 2020. `https://crypto.interactive-maths.com/vigenegravere-cipher.html`.
4. "Vigenère cipher." Accessed December 23, 2020. `https://en.wikipedia.org/wiki/Vigenère_cipher`.
5. "Cryptography." Accessed December 23, 2020. `https://en.wikipedia.org/wiki/Cryptography#History_of_cryptography_and_cryptanalysis`.
6. Binance Academy, "History of Cryptography." Accessed December 23, 2020. `https://www.binance.vision/security/history-of-cryptography`.
7. "Caesar Cipher." Accessed December 23, 2020. `https://en.wikipedia.org/wiki/Caesar_cipher`.

### Vigenère Cipher

Simple substitution ciphers like the Caesar cipher are relatively easy to decrypt. For example, "e" is the most frequently used English letter. So, you could study English ciphertext and assume that the most frequently appearing character probably is an "e."

The Vigenère secret-key cipher uses letters from the plaintext and a secret key to locate replacement characters in 26 Caesar ciphers—one for each letter of the alphabet. These 26 ciphers form a 26-by-26 two-dimensional array called the **Vigenère square**:

| | **A** | **B** | **C** | **D** | **E** | **F** | **G** | **H** | **I** | **J** | **K** | **L** | **M** | **N** | **O** | **P** | **Q** | **R** | **S** | **T** | **U** | **V** | **W** | **X** | **Y** | **Z** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **A** | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
| **B** | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A |
| **C** | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B |
| **D** | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C |
| **E** | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D |
| **F** | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E |
| **G** | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F |
| **H** | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G |
| **I** | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H |
| **J** | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I |
| **K** | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J |
| **L** | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K |
| **M** | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L |
| **N** | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M |
| **O** | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N |
| **P** | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| **Q** | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P |
| **R** | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q |
| **S** | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R |
| **T** | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S |
| **U** | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T |
| **V** | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U |
| **W** | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V |
| **X** | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W |
| **Y** | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X |
| **Z** | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y |

You look up substitutions using the bold blue letters that label the rows and columns.

### Secret-Key Requirements

For the Vigenère cipher described here, the secret key must contain only letters. Like passwords, the secret key should not be easy to guess. To create the ciphertext in the titles at the beginning of this exercise, we used as our secret key the following 11 randomly selected characters:

```
XMWUJBVYHXZ
```

Your key can have as many characters as you like. The person decrypting the ciphertext **must know the secret key used to create the ciphertext**.[8] Presumably, you'd provide that in advance—possibly in a face-to-face meeting. The secret key must, of course, be carefully guarded.

8. There are many websites offering Vigenère cipher decoders that attempt to decrypt ciphertext without the original secret key. We tried several, but none restored our original text.

### The Vigenère Cipher Encryption Algorithm

To see how the Vigenère cipher works, let's use the key `"XMWUJBVYHXZ"` and encrypt the plaintext string:

```
Welcome to encryption
```

Our encryption and decryption implementations preserve the plaintext's original case. Uppercase letters in the plaintext remain as uppercase in the ciphertext and vice versa, and lowercase letters in the plaintext remain as lowercase in the ciphertext and vice versa. We chose to pass non-letters in the plaintext—like spaces, digits and punctuation—through to the ciphertext and vice versa.

First, we repeat the secret key until the length matches the plaintext:

| | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *Plaintext:* | W | e | l | c | o | m | e | | t | o | | e | n | c | r | y | p | t | i | o | n |
| *Repeating key text:* | X | M | W | U | J | B | V | | Y | H | | X | Z | X | M | W | U | J | B | V | Y |

In the diagram above, we highlighted in light blue the secret key, then in darker blue the secret key's eight repeated letters.

We begin the encryption by using the first letter in the repeating key text (`'X'`) to select a row in the Vigenère square and using the first letter in the plaintext (`'W'`) to select a column. The intersection of that row and column (highlighted below) contains the letter to substitute in the ciphertext for `'W'`—in this case, `'T'`:

| | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
| B | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A |
| C | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B |
| D | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C |
| E | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D |
| F | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E |
| G | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F |
| H | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G |
| I | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H |
| J | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I |
| K | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J |
| L | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K |
| M | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L |
| N | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M |
| O | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N |
| P | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| Q | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P |
| R | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q |
| S | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R |
| T | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S |
| U | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T |
| V | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U |
| W | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V |
| X | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W |
| Y | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X |
| Z | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y |

This process continues for each pair of letters from the secret key and the plaintext:

| | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *Plaintext:* | W | e | l | c | o | m | e | | t | o | | e | n | c | r | y | p | t | i | o | n |
| *Repeating key text:* | X | M | W | U | J | B | V | | Y | H | | X | Z | X | M | W | U | J | B | V | Y |
| *Ciphertext:* | T | q | h | w | x | n | z | | r | v | | b | m | z | d | u | j | c | j | j | l |

## Decrypting with the Vigenère Cipher

The decryption process returns the ciphertext to the original plaintext. It's similar to what we described above and **requires the same secret key used to encrypt the text.** Like the encryption algorithm, the decryption algorithm cycles through the secret key's letters. So, again, we repeat the secret key until the length matches the ciphertext:

| | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *Ciphertext:* | T | q | h | w | x | n | z | | r | v | | b | m | z | d | u | j | c | j | j | l |
| *Repeating key text:* | X | M | W | U | J | B | V | | Y | H | | X | Z | X | M | W | U | J | B | V | Y |

We begin the decryption by using the first letter in the repeating key text (`'X'`) to select a row in the Vigenère square. Next, we locate within that row the first letter in the ciphertext (`'T'`). Finally, we replace the ciphertext letter with the plaintext letter at the top of that column (`'W'`), as highlighted in the Vigenère square below:

| | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
| B | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A |
| C | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B |
| D | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C |
| E | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D |
| F | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E |
| G | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F |
| H | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G |
| I | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H |
| J | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I |
| K | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J |
| L | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K |
| M | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L |
| N | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M |
| O | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N |
| P | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| Q | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P |
| R | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q |
| S | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R |
| T | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S |
| U | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T |
| V | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U |
| W | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V |
| X | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W |
| Y | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X |
| Z | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y |

This process continues for each pair of letters from the secret key and the ciphertext:

| | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *Ciphertext:* | T | q | h | w | x | n | z | | r | v | | b | m | z | d | u | j | c | j | j | l |
| *Repeating key text:* | X | M | W | U | J | B | V | | Y | H | | X | Z | X | M | W | U | J | B | V | Y |
| *Plaintext:* | W | e | l | c | o | m | e | | t | o | | e | n | c | r | y | p | t | i | o | n |

### Implementing the Vigenère Cipher

For this exercise, you should implement your Vigenère cipher code in the file `cipher.c`. This source-code file should contain the following items:

- **Function `checkKey`** receives a secret-key string and returns `true` if that string consists only of letters. Otherwise, this function returns `false`, in which case the key cannot be used with the Vigenère cipher algorithm. This function is called by the **`encrypt`** and **`decrypt`** functions described below.
- **Function `getSubstitution`** receives a secret-key character, a character from a plaintext or ciphertext string and a `bool` indicating whether to encrypt (`true`) or decrypt (`false`) the character in the second argument. This function is called by the **`encrypt`** and **`decrypt`** functions (described below) to perform the Vigenère cipher encryption or decryption algorithm for one character. The function contains the Vigenère square as a 26-by-26 two-dimensional `static const char` array.
- **Function `encrypt`** receives a string containing the **plaintext to encrypt**, a character array in which to write the **encrypted text**, and the **secret key**. The function iterates through the plaintext characters. For each letter, **`encrypt`** calls **`getSubstitution`**, passing the current secret-key character, the letter to encrypt and `true`. Function **`getSubstitution`** then performs the Vigenère cipher encryption algorithm for that letter and returns its ciphertext equivalent.
- **Function `decrypt`** receives a string containing the **ciphertext to decrypt**, a character array in which to write the resulting **plaintext**, and the **secret key** used to create the ciphertext. The function iterates through the ciphertext characters. For each letter, **`decrypt`** calls **`getSubstitution`**, passing the current secret key character, the letter to decrypt and `false`. Function **`getSubstitution`** then performs the Vigenère cipher decryption algorithm for that letter and returns its plaintext equivalent.

### Other Files You Should Create

In addition to `cipher.c`, you should create the following code files:

- **`cipher.h`** should contain the **`encrypt` and `decrypt` function prototypes**.
- **`cipher_test.c`**, which **`#includes "cipher.h"`** and uses your `encrypt` and `decrypt` functions to encrypt and decrypt text.

### `cipher_test.c`

In your application, perform the following tasks:

1. Prompt for and input a **plaintext sentence to encrypt** and a **secret key** consisting only of letters, call **`encrypt`** to create the **ciphertext**, then display it. Use our secret key `XMWUJBVYHXZ`—this will enable you to decrypt the gibberish at the beginning of this exercise.

2. Use your **`decrypt` function** and the **secret key** you entered in *Step 1* to **decrypt** the **ciphertext** you just created. Display the resulting **plaintext** to ensure your **`decrypt` function** worked correctly.
3. Prompt for and input either the ciphertext section title that precedes this exercise or the exercise ciphertext title. Then, use your **`decrypt` function** and the **secret-key text** you entered in *Step 1* to **decrypt the ciphertext**.

As always, you should ensure that the character arrays into which you write encrypted or decrypted text are large enough to store the text and its terminating null character.

Once your Vigenère cipher encryption and decryption algorithms work, have some fun sending and receiving encrypted messages with your friends. When you pass your secret key to the person who'll use it to decrypt your ciphertext messages, focus on keeping your key secure.

### Compiling Your Code

In Visual C++ and Xcode, simply add all three files to your project, then compile and run the code. For GNU `gcc`, execute the following command from the folder containing `cipher.c`, `cipher.h` and `cipher_test.c` files:

```
gcc -std=c18 -Wall cipher.c cipher_test.c -o cipher_test
```

This will create the command `cipher_test`, which you can run with `./cipher_test`.

### Weakness in Secret-Key Cryptography: A Look to Public-Key Cryptography

Secret-key encryption and decryption have a weakness—the ciphertext is only as secure as the secret key. The ciphertext can be decrypted by anyone who discovers or steals the secret key. In the next exercise, we introduce **public-key cryptography**. This technique performs encryption with a public key known to every sender who may want to send a secret message to a particular receiver. The public key can be used to encrypt messages but not decrypt them. The messages can be decrypted only with a paired private key known only to the receiver, so it's much more secure than the secret key in secret-key cryptography. In the next case study exercise, you'll explore public-key cryptography.

### A Note about Cryptography and Computing Power

Ideally, Ciphertext should be impossible to "break"—that is, it should not be possible to determine the plaintext from the ciphertext *without* the decryption key. For various reasons, that goal is impractical. So, designers of cryptography schemes settle for making them extraordinarily difficult to break. One problem with today's increasingly powerful computers is that they're making it possible to break most encryption schemes in use over the last few decades.

Cryptography is at the root of cryptocurrencies such as Bitcoin.[9] The phenomenally powerful computers that quantum computing will make possible are putting

9. "Cryptocurrency." Accessed December 25, 2020. `https://www.investopedia.com/terms/c/cryptocurrency.asp`.

cryptography schemes and cryptocurrencies at risk.[10,11] The cryptocurrency community is working on these challenges.[12,13,14]

**8.42** ***(Vigenère Cipher Modification—Supporting All ASCII Characters)*** Your Vigenère cipher implementation from **Exercise 8.41** encrypts and decrypts only the letters A–Z. All other characters simply pass through as is. Modify your implementation to support the complete ASCII character set shown in **Appendix B**.

## SEC Secure C Programming Case Study: Public-Key Cryptography

**8.43** ***(RSA[15,16,17] Public-Key Cryptography)*** In the last case study, you began learning about secret-key cryptography. The sender's plaintext is encrypted with a secret key to form ciphertext. The receiver uses the *same* secret key to decode the ciphertext, forming the original plaintext—this is called **symmetric encryption**. A problem with secret-key cryptography is that the security of the ciphertext is only as good as the security of the secret key, and several copies of that key are "floating around." In an attempt to correct this problem, public-key cryptography was proposed by Diffie–Hellman.[18]

In this case-study exercise, we walk step-by-step through the **RSA Public-Key Cryptography algorithm**. In particular, we focus on how to generate:

- the **public key** that any sender can use to encrypt plaintext into ciphertext for a particular receiver, and
- the **private key** that only the particular receiver can use to decrypt the ciphertext.

RSA is based on sophisticated mathematics, but the steps you need to perform to generate the public and private keys, encrypt messages with the public key and decrypt

10. "The Impact of Quantum Computing on Present Cryptography." Accessed December 25, 2020. https://arxiv.org/pdf/1804.00200.pdf.
11. "Quantum Computing and its Impact on Cryptography." Accessed December 25, 2020. https://www.cryptomathic.com/news-events/blog/quantum-computing-and-its-impact-on-cryptography.
12. "How Should Crypto Prepare for Google's 'Quantum Supremacy'?" Accessed December 25, 2020. https://www.coindesk.com/how-should-crypto-prepare-for-googles-quantum-supremacy.
13. "Here's Why Quantum Computing Will Not Break Cryptocurrencies." Accessed December 25, 2020. https://www.forbes.com/sites/rogerhuang/2020/12/21/heres-why-quantum-computing-will-not-break-cryptocurrencies/.
14. "How the Crypto World Is Preparing for Quantum Computing, Explained." Accessed December 25, 2020. https://cointelegraph.com/explained/how-the-crypto-world-is-preparing-for-quantum-computing-explained.
15. "RSA (cryptosystem)." Accessed January 6, 2021. https://en.wikipedia.org/wiki/RSA_(cryptosystem).
16. "RSA Algorithm." Accessed January 6, 2021. https://simple.wikipedia.org/wiki/RSA_algorithm.
17. "PKCS #1: RSA Cryptography Specifications Version 2.2." Accessed January 8, 2021. https://tools.ietf.org/html/rfc8017.
18. "New Directions in Cryptography." Accessed January 8, 2021. https://ee.stanford.edu/~hellman/publications/24.pdf.

messages with the private key are straightforward, as we'll show momentarily. Industrial-quality RSA works with enormous prime numbers consisting of hundreds of digits. To keep our explanations simple and to enable you to quickly build a small-scale working version of RSA, we're going to use only small prime numbers in our explanations. Such small-prime-number RSA versions are not very secure, but they'll help you understand how RSA works.

### Public-Key Cryptography

Whitfield Diffie and Martin Hellman, in their paper "New Directions in Cryptography,"[19] introduced **public-key cryptography** to address the weakness of secret-key cryptography—which is the vulnerability of the secret key having to be known by both the sender and the receiver. They came up with the idea but not an implementation of the scheme.

### RSA Public-Key Cryptography

Rivest, Shamir and Adelman were the first to publish a working implementation of public-key cryptography. The scheme, called RSA[20], bears the initials of their last names. RSA is one of the most widely implemented public-key cryptography schemes in the world.[21] Because RSA can be slow,[22] many organizations prefer to stick to faster private-key encryption, using RSA to securely send the secret key.

PERF

### Historical Notes

Clifford Cocks in the U.K. created a workable public-key scheme several years before the RSA paper was published,[23] but his work was classified, so it was not revealed until about 20 years after RSA appeared.

The company RSA Security held a patent on the RSA algorithm. In 2000, that patent was coming due for renewal—instead of renewing, they placed the algorithm into the public domain.[24]

---

19. "New Directions in Cryptography." Accessed January 8, 2021. `https://ee.stanford.edu/~hellman/publications/24.pdf`.
20. R. Rivest; A. Shamir; L. Adleman (February 1978). "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems" (PDF). Communications of the ACM. 21 (2): 120–126. `https://people.csail.mit.edu/rivest/Rsapaper.pdf`.
21. "RSA algorithm (Rivest-Shamir-Adleman)." Accessed January 8, 2021. `https://searchsecurity.techtarget.com/definition/RSA`.
22. "RSA (cryptosystem)." Accessed January 6, 2021. `https://en.wikipedia.org/wiki/RSA_(cryptosystem)`.
23. "Clifford Cocks." Accessed January 8, 2021. `https://en.wikipedia.org/wiki/Clifford_Cocks`.
24. "RSA Security Releases RSA Encryption Algorithm into Public Domain." Accessed January 8, 2021. `https://web.archive.org/web/20071120112201/http://www.rsa.com/press_release.aspx?id=261`.

### RSA Algorithm Steps

*Steps 1–5* below use small integer values to explain how the RSA algorithm generates a public-key/private-key pair. Then, *Step 6* uses the public key to encrypt plaintext into ciphertext, and *Step 7* uses the private key to decrypt the ciphertext back to the original plaintext. The steps we show are based on the original RSA paper[25] and the RSA Algorithm Wikipedia page.[26]

### RSA Algorithm Step 1—Choose Two Prime Numbers

Choose two different prime numbers `p` and `q`. For this case study, we'll use small prime numbers—`p = 13` and `q = 17`. This will keep the calculations manageable in our discussions and on your computer using C with its limited-range, built-in integer data types. In commercial-grade RSA cryptography systems, these prime numbers typically are hundreds of digits each and chosen at random. For a sense of how large the integers in RSA can be, visit the RSA Numbers webpage

```
https://en.wikipedia.org/wiki/RSA_numbers
```

which shows various integers from 100 to 617 digits in length. The C integer data types `int`, `long int` and `long long int` cannot hold integers this large, so special processing is required to accommodate such large numbers.

### RSA Algorithm Step 2—Calculate the Modulus (n), Which Is Part of Both the Public and Private Keys

Calculate the **modulus `n`**, which is simply the product of `p` and `q`:

```
n = p * q
```

Based on `p = 13` and `q = 17`, `n` is `221`. As you'll see, `n` is part of both the public and private keys. The `p` and `q` values are kept private.

### RSA Algorithm Step 3—Calculate the Totient Function

Calculate $\Phi(n)$—pronounced "phi of `n`"—which is Euler's totient function.[27] This is calculated simply as:

```
Φ(n) = (p - 1) * (q - 1)
```

Given `p = 13` and `q = 17`, $\Phi(n)$ is

```
Φ(n) = 12 * 16 = 192
```

This number is used in the calculations that determine the **encryption exponent** (**`e`**) and **decryption exponent** (**`d`**), which will help us encrypt plaintext and decrypt ciphertext, respectively, as you'll see below.

---

25. Rivest, R.L., A. Shamir, and L. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems" (1977). Accessed January 8, 2021. `https://people.csail.mit.edu/rivest/Rsapaper.pdf`.
26. "RSA Algorithm." Accessed January 6, 2021. `https://simple.wikipedia.org/wiki/RSA_algorithm`.
27. "Euler's totient function." Accessed January 7, 2021. `https://en.wikipedia.org/wiki/Euler%27s_totient_function`.

### RSA Algorithm Step 4—Select the Public-Key Exponent (e) for Encryption Calculations

Next, we **choose an exponent, e, for encryption**, which is subject to the following rules:

- `1 < e < Φ(n)`
- `e` must be **coprime** with `Φ(n)`.

Two integers are coprime if they have no common factors other than 1.

In our example, the integers that satisfy the first rule for `Φ(n) = 192` are the values `2`–`191`. The prime factorization of 192 is

```
192 = 2 * 2 * 2 * 2 * 2 * 2 * 3
```

The value for `e` must be coprime with `Φ(n)`, so we must eliminate from consideration for `e` any prime factors and all their multiples. Thus, the value `2` and all the other even integers from `2`–`190` are eliminated, as are the value `3` and all its multiples. This leaves the following odd values as possible values for `e`:

```
  5   7  11  13  17  19  23  25  29  31  35  37  41  43  47  49
 53  55  59  61  65  67  71  73  77  79  83  85  89  91  95  97
101 103 107 109 113 115 119 121 125 127 131 133 137 139 143 145
149 151 155 157 161 163 167 169 173 175 179 181 185 187 191
```

Any of these values can be used as the public encryption key's exponent (`e`). For our continuing discussion we'll choose `37`, so **our public key is `(37, 221)`**.

### RSA Algorithm Step 5—Select the Private-Key Exponent (d) for Encryption Calculations

The final step is to **determine the private key's exponent, `d`, for decryption**. We must choose a value for `d` such that

```
(d * e) mod Φ(n) = 1
```

In our example, the first value of `d` for which this is true is `109`. We can check whether the preceding calculation produces `1` by plugging in the values of `d`, `e` and `Φ(n)`:

```
(109 * 37) mod 192
```

The value of `109 * 37` is `4033`. If you multiply `192` by `21`, the result is `4032`, leaving a remainder of `1`. So, `109` is a valid value for `d`. There are many potential values of `d`—each is `109` plus a multiple of the totient (`192`). For instance, `301` (`109 + 1 * 192`):

```
(301 * 37) mod 192
```

`301 * 37` is `11137`, which has the remainder `1` when divided by `192`—`192 * 58` is `11136`, leaving a remainder of `1`. So values for `d` such as the following will work:

```
109  301  493  685  877  ...
```

We chose `109`, so our private key is `(109, 221)`.

### Encrypting a Message with RSA

Once you have the public key, it's easy to encrypt a message using RSA. **Given a plaintext integer message (`M`) to encrypt into ciphertext (`C`) and a public key consisting**

**of two positive integers `e` (for encrypt) and `n`—commonly represented as `(e, n)`—a message sender can encrypt `M` with the calculation:**

$$C = M^e \bmod n$$

The value of `M` must be in the range `0 <= M < n`. Otherwise, you must break the message into values within that range and encrypt each separately.

Let's encrypt the `M` value `122` using our public key `(37, 221)`:

$$C = 122^{37} \bmod 221$$

The value $122^{37}$ is an enormous number, but you can perform this calculation using Wolfram Alpha at

```
https://www.wolframalpha.com/input/
```

Enter the calculation as follows (the `^` represents exponentiation in Wolfram Alpha):

```
122^37 mod 221
```

You'll see that the result is `5`, which is our ciphertext.

### Decrypting a Message with RSA

It's also easy to decrypt a message if you have the private key. **Given a ciphertext integer message (`C`) to decrypt into the original plaintext message (`M`) and a private key consisting of two positive integers `d` and `n`—commonly represented as `(d, n)`—a message receiver can decrypt `C` with the following calculation:**

$$M = C^d \bmod n$$

Let's decrypt the `C` value `5` using our public key `(109, 221)`:

$$C = 5^{109} \bmod 221$$

Once again, the value $5^{109}$ is an enormous number, but you can perform this calculation using Wolfram Alpha by entering the calculation as follows:

```
5^109 mod 221
```

You'll see that the result is `122`, which is our plaintext.

**Note that `n` is part of *both* the public key and the private key. You'll also see that the exponent `d`'s value is based on the exponent `e` and the modulus value `n`.**

### Encrypting and Decrypting Strings

Suppose you wish to use RSA to encrypt a plaintext message, such as

```
Damn the torpedoes, full speed ahead!
```[28]

As you know, the RSA algorithm encrypts *only* integer messages in the range `0 <= M < n`. To encrypt the preceding message, you must map the characters to integer values.

One way to convert characters to integers is to use each character's numeric value in the underlying character set. For this exercise, assume ASCII characters, which have integer values in the range 0–127 (see Appendix B). Provided that a character's

28. David Glasgow Farragut—an American Civil War Union officer and the first full admiral in the U.S. Navy. Accessed January 8, 2021. `https://en.wikipedia.org/wiki/David_Farragut`.

integer value is less than n, you can encrypt that value as shown previously. You can store each resulting ciphertext integer in an integer array. If you try to display those ciphertext integers as characters, you may see some strange symbols. For instance, the ciphertext integers may represent special characters, such as newlines or tabs, or may be outside the ASCII range. When you decrypt the ciphertext, you can take each resulting integer, cast it to a `char`, then place it into a `char` array that will represent the deciphered plaintext. Be sure to null-terminate your string before displaying it.

### Programming the RSA Algorithm

Now, implement the RSA algorithm in C. Enable the user to encrypt and decrypt a simple integer, then encrypt and decrypt a line of text. Your program should produce an output dialog similar to the following:

```
Enter a prime number for p: 13
Enter a prime number for q: 17
n is 221
totient is 192

Candidates for e: 5 7 11 13 17 19 23 25 29 31 35 37 41 43 47 49 53 55 59 61 65
67 71 73 77 79 83 85 89 91 95 97 101 103 107 109 113 115 119 121 125 127 131 133
137 139 143 145 149 151 155 157 161 163 167 169 173 175 179 181 185 187 191

Select a value for e from the preceding candidates: 37

Candidate for d: 109

Select a value for d--either the d candidate above
or d plus a multiple of the totient: 109

Enter a non-negative integer less than n to encrypt: 122

The ciphertext is: 5

The decrypted plaintext is: 122

Enter a sentence to encrypt:
Damn the torpedoes, full speed ahead!

The ciphertext is:
DG`
ue
;X}eW;es9 fh s}eeW GueGW!

The decrypted plaintext is:
Damn the torpedoes, full speed ahead!
```

As you implement the RSA algorithm, keep the following hints in mind:

- **Modular exponentiation**: Raising a plaintext message to a large exponent (e.g., $122^{37}$) results in enormous values that C's limited-range, built-in integer types cannot represent. As you know, RSA encryption and decryption calculations perform both exponentiation and modulus operations. These can be combined using modular exponentiation to keep the RSA encryption and decryption

calculations within manageable ranges. Define a function named `modularPow` that performs modular exponentiation. For the modular exponentiation algorithm, see the psuedocode at

```
https://en.wikipedia.org/wiki/Modular_exponentiation#Memory-efficient_method
```

- **Calculating the greatest common divisor**: The candidate values for `e` (*Step 4*) must be coprime with the totient—again, their only common factor is 1. To determine if two numbers are coprime, you'll need a **function `gcd`** that calculates the **greatest common divisor of two integers**. Your program should display all possible candidate values for `e`. You were asked to write a `gcd` function in Exercise 5.29.
- **Checking for prime numbers**—The RSA algorithm requires two prime numbers, `p` and `q`. You should define a **function `isPrime`** that **determines if an integer is indeed a prime number**—use it to confirm that the `p` and `q` values the user entered are prime. Exercise 6.30 asked you to implement the Sieve of Eratosthenes to find prime values.

Your program also should define the following functions:

- A **function to encrypt a plaintext message `M` using the public key `(e, n)`**:

```
int encrypt(int M, int e, int n);
```

- A **function to decrypt a ciphertext message `C` using the private key `(d, n)`**:

```
int decrypt(int C, int d, int n);
```

- A **function to encrypt a string by calling the `encrypt` function for each character of the string and placing the results into an integer array**:

```
void encryptString(
   char* plaintext, int ciphertext[], int e, int n);
```

- A **function to decrypt ciphertext from an integer array by calling the `decrypt` function for each integer and placing the results into a `char` array**. The size parameter represents the number of characters that were encrypted. Be sure to terminate the string in `decryptedPlaintext` with a null character (`'\0'`):

```
void decryptString(int ciphertext[],
   char decryptedPlaintext[], size_t size, int d, int n);
```

## References

For a nice video explanation of the RSA algorithm, see the following two-part video presentation:

- The RSA Encryption Algorithm (1 of 2: Computing an Example):[29]

```
https://www.youtube.com/watch?v=4zahvcJ9glg
```

29. Eddie Woo (misterwootube). "The RSA Encryption Algorithm (1 of 2: Computing an Example)," November 4, 2014. `https://www.youtube.com/watch?v=4zahvcJ9glg`.

- The RSA Encryption Algorithm (2 of 2: Generating the Keys):[30]

  https://www.youtube.com/watch?v=oOcTVTpUsPQ

**8.44** ***(An Improvement to the RSA Algorithm)*** In 1998, an improvement was made to the RSA algorithm replacing Φ(n) with λ(n) (pronounced "lambda of n"):[31,32]

```
λ(n) = lcm((p - 1), (q - 1))
```

where `lcm` represents the **least common multiple**.[33] We used Φ(n) in the previous RSA exercise with `p` = 13 and `q` = 17. The corresponding new λ(n) calculation would be

```
λ(n) = lcm(12, 16)
```

where the least common multiple of 12 and 16 is 48, as you can see in the lists of multiples below:

```
12  24  36  48  ...
16  32  48  60  ...
```

Make a copy of your code solution for the previous exercise and replace each use of Φ(n) with λ(n), then test your updated code with the same prime-number values for `p` and `q`. When you encrypt the plaintext using the λ(n) approach, your ciphertext will likely be different, but the decrypted plaintext should be the same.

**8.45** ***(Stress Testing Your RSA Algorithm's Limits)*** Try your program with gradually increasing values for `p` and `q`. How large do they get before the program no longer works? Also, test your program with increasingly larger candidates for `e` and `d`.

**8.46** ***(Enhancing Your RSA Code)*** Modify your RSA program as follows:

a) Your program displayed all the possible candidates for the encryption exponent `e`. Modify your program to show the first five potential values for the decryption exponent `d` (i.e., the first value of `d` plus `1 * totient`, the first value of `d` plus `2 * totient`, etc.). Follow your list of possibilities with an ellipsis (...).

b) As your prime numbers `p` and `q` get larger, you'll eventually surpass the `int` type's maximum value limit, which you can find in `<limits.h>`. Modify your code to do all RSA integer calculations using type `long long int`. Note that even that type will be inadequate for holding the enormous integers you'd use in industrial quality RSA. It would require special programming to use such larger integer values. Remember to change any `printf` and `scanf` statements' `%d` conversion specifiers to `%lld`.

---

30. Eddie Woo (misterwootube). "The RSA Encryption Algorithm (2 of 2: Generating the Keys)," November 4, 2014. https://www.youtube.com/watch?v=oOcTVTpUsPQ.
31. "RSA Algorithm." Accessed January 7, 2021. https://simple.wikipedia.org/wiki/RSA_algorithm.
32. "PKCS #1: RSA Cryptography Specifications, Version 2.0." Accessed January 7, 2021. https://tools.ietf.org/html/rfc2437.
33. "Least common multiple." Accessed January 7, 2021. https://en.wikipedia.org/wiki/Least_common_multiple.

**8.47** ***(Challenge Project: The RSA Problem[34])*** In this exercise, you'll research attacks that have been perpetrated on industrial-strength RSA implementations. You'll then try your own hand at cracking RSA ciphertext created by the small-scale RSA implementation you built in Exercise 8.43. Again, such small-scale implementations are not secure.

a) Research the kinds of attacks that have been perpetrated against industrial-strength RSA systems. Note which kinds have succeeded and which have failed.

b) RSA's strength comes from the enormous prime numbers `p` and `q` (each typically hundreds of digits) used to calculate the far more enormous value of `n` (which is `p * q`) and the computational expense of factoring `n` to find `p` and `q`. The "RSA Problem" is the task of decrypting ciphertext given only the public key `(e, n)`. This requires you to find `n`'s prime factors `p` and `q` from which you would then derive `d` and decrypt the ciphertext.

Assume you have a public key `(e, n)` and ciphertext that was encrypted using that key with your small-scale RSA implementation, but you do not know the private key required to decrypt the ciphertext. Use brute-force computing techniques to find `n`'s prime factors `p` and `q`. Then, do the calculations necessary to recover `d` and decrypt the message.

34. "RSA Problem." Accessed January 8, 2021. `https://en.wikipedia.org/wiki/RSA_problem`.