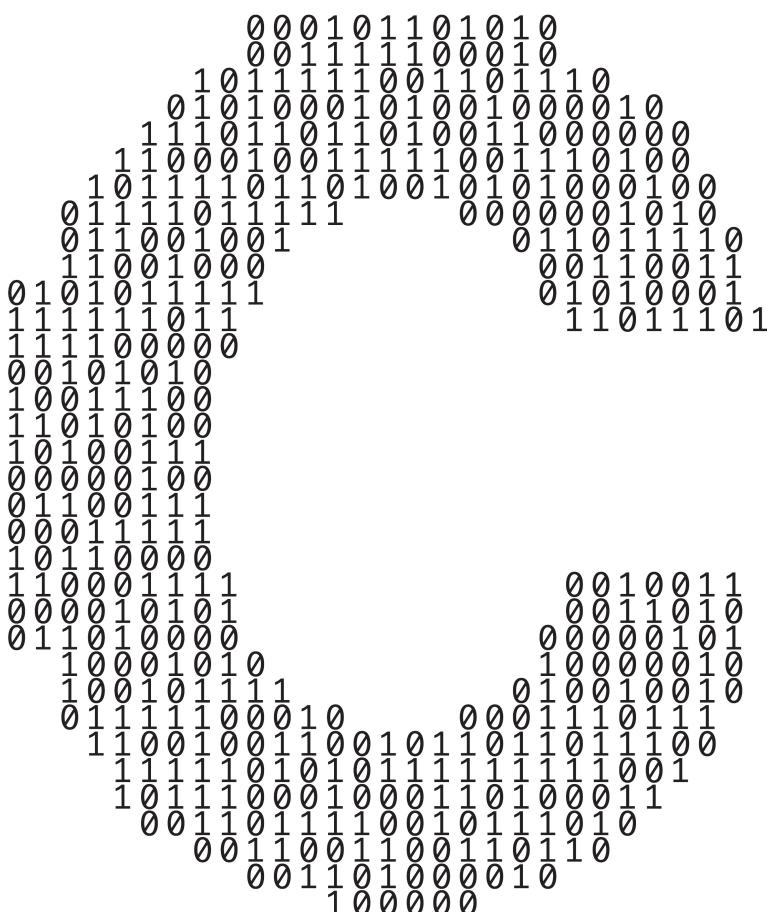


# 9

## Formatted Input/Output

### Objectives

In this chapter, you'll:



A large binary number pattern is arranged to form the shape of the digit '9'. The pattern starts at the top right and curves down and to the left, ending at the bottom left. The binary digits are 0s and 1s.

- Use input and output streams.
- Use print formatting capabilities.
- Use input formatting capabilities.
- Print integers, floating-point numbers, strings and characters.
- Print with field widths and precisions.
- Use formatting flags in the `printf` format control string.
- Output literals and escape sequences.
- Read formatted input using `scanf`.

<b>9.1</b> Introduction	<b>9.9</b> <code>printf</code> Format Flags
<b>9.2</b> Streams	9.9.1 Right- and Left-Alignment
<b>9.3</b> Formatting Output with <code>printf</code>	9.9.2 Printing Positive and Negative Numbers with and without the + Flag
<b>9.4</b> Printing Integers	9.9.3 Using the Space Flag
<b>9.5</b> Printing Floating-Point Numbers	9.9.4 Using the # Flag
9.5.1 Conversion Specifiers e, E and f	9.9.5 Using the 0 Flag
9.5.2 Conversion Specifiers g and G	
9.5.3 Demonstrating Floating-Point Conversion Specifiers	
<b>9.6</b> Printing Strings and Characters	<b>9.10</b> Printing Literals and Escape Sequences
<b>9.7</b> Other Conversion Specifiers	
<b>9.8</b> Printing with Field Widths and Precision	<b>9.11</b> Formatted Input with <code>scanf</code>
9.8.1 Field Widths for Integers	9.11.1 <code>scanf</code> Syntax
9.8.2 Precisions for Integers, Floating-Point Numbers and Strings	9.11.2 <code>scanf</code> Conversion Specifiers
9.8.3 Combining Field Widths and Precisions	9.11.3 Reading Integers
	9.11.4 Reading Floating-Point Numbers
	9.11.5 Reading Characters and Strings
	9.11.6 Using Scan Sets
	9.11.7 Using Field Widths
	9.11.8 Skipping Characters in an Input Stream
	<b>9.12</b> Secure C Programming

[Summary](#) | [Self-Review Exercises](#) | [Answers to Self-Review Exercises](#) | [Exercises](#)

## 9.1 Introduction

Presenting results is an important part of the solution to any problem. This chapter discusses in-depth `printf` and `scanf` formatting features, which output data to the standard output stream and input data from the standard input stream. Include the header `<stdio.h>` in programs that call these functions. Chapter 11 discusses several additional functions included in the standard input/output (`<stdio.h>`) library.

## 9.2 Streams

Input and output are performed with sequences of bytes called **streams**:

- In input operations, the bytes flow into main memory from a device, such as a keyboard, a solid-state drive, a network connection, and so on.
- In output operations, bytes flow from main memory to a device, such as a computer's screen, a printer, a solid-state drive, a network connection, and so on.

When program execution begins, the program has access to three streams:

- the standard input stream, which is connected to the keyboard,
- the standard output stream, which is connected to the screen, and
- the **standard error stream**, which also is connected to the screen.

Operating systems allow these streams to be redirected to other devices. Chapter 11 discusses stream-processing in detail.

 **Self Check**

- 1 (Fill-In) You can \_\_\_\_\_ the standard streams to other devices.

Answer: redirect.

- 2 (Multiple Choice) Which of the following statements is *false*?

- a) Input and output are performed with arrays, which are sequences of bytes.
- b) In input operations, the bytes flow from a device to main memory.
- c) In output operations, bytes flow from main memory to a device.
- d) When execution begins, the standard streams are connected to the program.

Answer: a) is *false*. Actually, input and output are performed with streams, which are sequences of bytes.

## 9.3 Formatting Output with printf

Throughout the book, you've seen various `printf` output formatting features. Every `printf` call contains a **format control string** that describes the output format. The format control string consists of **conversion specifiers**, **flags**, **field widths**, **precisions** and **literal characters**. Together with the percent sign (%), these form **conversion specifications**. Function `printf` can perform the following formatting capabilities:

1. **Rounding** floating-point values to an indicated number of decimal places.
2. Aligning columns of numbers at their decimal points.
3. **right-aligning** and **left-aligning** outputs.
4. Inserting literal characters at precise locations in a line of output.
5. Representing floating-point numbers in exponential format.
6. Representing unsigned integers in octal and hexadecimal format. Online Appendix E discusses octal and hexadecimal values.
7. Displaying data with fixed-size field widths and precisions.

The `printf` function has the form

```
printf(format-control-string, other-arguments);
```

The *format-control-string* describes the output format, and the optional *other-arguments* correspond to the *format-control-string*'s conversion specifications. Every conversion specification begins with a percent sign (%) and ends with a conversion specifier. There can be many conversion specifications in one format control string.

 **Self Check**

- 1 (Fill-In) Every `printf` call contains a \_\_\_\_\_ that describes the output format.

Answer: format control string.

- 2 (Multiple Choice) Which of the following is a formatting capability function `printf` can perform?

- a) Rounding floating-point values to an indicated number of decimal places, and aligning a column of numbers at their decimal points.

- b) Representing floating-point numbers in exponential format. Representing unsigned integers in octal and hexadecimal format.
- c) Displaying all types of data with fixed-size field widths and precisions.
- d) All of the above are `printf` formatting capabilities.

**Answer:** d.

## 9.4 Printing Integers

An integer is a whole number, such as 776, 0 or -52. Integer values are displayed in one of several formats described by the following **integer conversion specifiers**.

Conversion specifier	Description
<code>d</code>	Display as a signed decimal integer.
<code>i</code>	Display as a signed decimal integer.
<code>o</code>	Display as an unsigned octal integer.
<code>u</code>	Display as an unsigned decimal integer.
<code>x</code> or <code>X</code>	Display as an unsigned hexadecimal integer. <code>X</code> uses the digits 0-9 and the uppercase letters A-F, and <code>x</code> uses the digits 0-9 and the lowercase letters a-f.
<code>h</code> , <code>l</code> or <code>ll</code> (letter “ell”)	These <b>length modifiers</b> are placed before any integer conversion specifier to indicate that the value to display is a <code>short</code> , <code>long</code> or <code>long long</code> integer.

Figure 9.1 prints an integer using each integer conversion specifier. Note that plus signs do not display by default, but we'll show later how to force them to display. Lines 10–11 use the `hd` and `ld` conversion specifiers to display `short` and `long` integer values. The `L` suffix on the literal `2000000000L` indicates that its type is `long`—C treats whole-number literals as `int`. Printing a negative value with a conversion specifier that expects an `unsigned` value is a logic error. When line 14 displays `-455` with `%u`, the result is the `unsigned` value `4294966841`. A small negative value displays as a large positive integer due to the value's “sign bit” in the underlying binary representation. See online Appendix E for a discussion of the binary number system and the sign bit.

---

```

1 // fig09_01.c
2 // Using the integer conversion specifiers
3 #include <stdio.h>
4
5 int main(void) {
6     printf("%d\n", 455);
7     printf("%i\n", 455); // i same as d in printf
8     printf("%d\n", +455); // plus sign does not print
9     printf("%d\n", -455); // minus sign prints

```

---

**Fig. 9.1** | Using the integer conversion specifiers. (Part I of 2.)



```
10    printf("%hd\n", 32000); // print as type short
11    printf("%ld\n", 2000000000L); // print as type long
12    printf("%o\n", 455); // octal
13    printf("%u\n", 455);
14    printf("%U\n", -455);
15    printf("%x\n", 455); // hexadecimal with lowercase letters
16    printf("%X\n", 455); // hexadecimal with uppercase letters
17 }
```

```
455
455
455
-455
32000
2000000000
707
455
4294966841
1c7
1C7
```

**Fig. 9.1** | Using the integer conversion specifiers. (Part 2 of 2.)

### ✓ Self Check

- 1 (*Multiple Choice*) Which integer conversion specifier is described by “Display as an unsigned decimal integer”?
- a) ud.
  - b) ui.
  - c) u.
  - d) None of the above.

**Answer:** c.

- 2 (*What Does This Code Do?*) Show precisely what the following code prints:

```
printf("%d\n", 235);
printf("%i\n", 235);
printf("%d\n", +235);
printf("%d\n", -235);
```

**Answer:**

```
235
235
235
-235
```

## 9.5 Printing Floating-Point Numbers

Floating-point values contain a decimal point, as in 33.5, 0.0 or -657.983. Floating-point values are displayed using the conversion specifiers summarized below.

Conversion specifier	Description
e or E	Display a floating-point value in exponential notation.
f or F	Display floating-point values in fixed-point notation.
g or G	Display a floating-point value in either the fixed-point form f or the exponential form e (or E), based on the value's magnitude.
L	Place this length modifier before any floating-point conversion specifier to indicate that a long double floating-point value should be displayed.

## Exponential Notation

The **conversion specifiers e and E** display floating-point values in **exponential notation**—the computer equivalent of **scientific notation** used in mathematics. For example, the value 150.4582 is represented in scientific notation as

1.504582 × 10<sup>2</sup>

and in exponential notation as

1.504582E+02

In this notation, the E stands for “exponent” and indicates that 1.504582 is multiplied by 10 raised to the second power (E+02).

### 9.5.1 Conversion Specifiers e, E and f

Values displayed with the conversion specifiers e, E and f show six digits of precision to the decimal point’s right by default (e.g., 1.045927). You can specify other precisions explicitly. **Conversion specifier f** always prints at least one digit to the left of the decimal point, so fractional values will be preceded by "0.". Conversion specifiers e and E precede the exponent with lowercase e or uppercase E. Each prints exactly one digit to the decimal point’s left.

### 9.5.2 Conversion Specifiers g and G

**Conversion specifier g (or G)** prints in either e (E) or f format with no trailing zeros, so 1.234000 displays as 1.234. The conversion specifier g uses the e (E) format if, after conversion to exponential notation, the value’s exponent is less than -4, or the exponent is greater than or equal to the specified precision. Otherwise, g uses the conversion specifier f to print the value. The default precision is six significant digits for g and G—a maximum of six digits will display.

At least one decimal digit is required for the decimal point to be output. For example, the values 0.0000875, 8750000.0, 8.75 and 87.50 are printed as 8.75e-05, 8.75e+06, 8.75 and 87.5 with the conversion specifier g. The value 0.0000875 uses e notation because, when it’s converted to exponential notation, its exponent (-5) is less than -4. The value 8750000.0 uses e notation because its exponent (6) is equal to the default precision.

## Precision

For conversion specifiers `g` and `G`, the precision indicates the maximum number of significant digits to display, including the digit to the left of the decimal point. So, the value `1234567.0` displays as `1.23457e+06`, using conversion specification `%g`. Remember that all floating-point conversion specifiers have a default precision of 6. There are six significant digits in the result—1 to the left of the decimal point and 23457 to the right. For exponential notation, `g` and `G` precede the exponent with a lowercase `e` or uppercase `E`. When displaying data, make it clear to users whether the data may be imprecise due to formatting, such as rounding errors from specifying precisions.

### 9.5.3 Demonstrating Floating-Point Conversion Specifiers

Figure 9.2 demonstrates each of the floating-point conversion specifiers. The `%E`, `%e` and `%g` conversion specifications perform rounding, but `%f` does not.

```
1 // fig09_02.c
2 // Using the floating-point conversion specifiers
3 #include <stdio.h>
4
5 int main(void) {
6     printf("%e\n", 1234567.89);
7     printf("%e\n", +1234567.89); // plus does not print
8     printf("%e\n", -1234567.89); // minus prints
9     printf("%E\n", 1234567.89);
10    printf("%f\n", 1234567.89); // six digits to right of decimal point
11    printf("%g\n", 1234567.89); // prints with lowercase e
12    printf("%G\n", 1234567.89); // prints with uppercase E
13 }
```

```
1.234568e+06
1.234568e+06
-1.234568e+06
1.234568E+06
1234567.890000
1.23457e+06
1.23457E+06
```

Fig. 9.2 | Using the floating-point conversion specifiers.

## ✓ Self Check

- 1 (*Fill-In*) The conversion specifiers `e` and `E` display floating-point values in exponential notation—the computer equivalent of \_\_\_\_\_ used in mathematics.

**Answer:** scientific notation.

- 2 (*Multiple Choice*) Which statement about conversion specifiers `e`, `E` and `f` is *false*?
- Values displayed with the conversion specifiers `e`, `E` and `f` show six digits of precision to the decimal point's right by default.

- b) Conversion specifier `f` always prints exactly one digit to the left of the decimal point.
- c) Conversion specifiers `e` and `E` print lowercase `e` and uppercase `E`, respectively, preceding the exponent, and exactly one digit to the left of the decimal point.
- d) All of the above statements are *true*.

**Answer:** b) is *false*. Actually, conversion specifier `f` prints at least one digit to the left of the decimal point.

## 9.6 Printing Strings and Characters

The `c` and `s` conversion specifiers are used to print individual characters and strings, respectively. **Conversion specifier `c`** requires a `char` argument. **Conversion specifier `s`** requires a pointer to `char` as an argument. Conversion specifier `s` prints characters until a terminating null ('`\0`') character is encountered. If the string does not have a null terminator, the result is undefined—`printf` will either continue printing until it encounters a zero byte or the program will terminate prematurely (i.e., “crash”) and



indicate a “segmentation fault” or “access violation” error. The program in Fig. 9.3 displays characters and strings with conversion specifiers `c` and `s`.

---

```

1 // fig09_03.c
2 // Using the character and string conversion specifiers
3 #include <stdio.h>
4
5 int main(void) {
6     char character = 'A'; // initialize char
7     printf("%c\n", character);
8
9     printf("%s\n", "This is a string");
10
11    char string[] = "This is a string"; // initialize char array
12    printf("%s\n", string);
13
14    const char *stringPtr = "This is also a string"; // char pointer
15    printf("%s\n", stringPtr);
16 }
```

```

A
This is a string
This is a string
This is also a string

```

**Fig. 9.3** | Using the character and string conversion specifiers.

### Errors in Format Control Strings

Most compilers do not catch format-control-string errors. You’ll typically become aware of such errors when a program fails or produces incorrect results at runtime.



- Using `%c` to print a string is a logic error—`%c` expects a `char` argument. A string is a pointer to `char` (i.e., a `char *`).



- Using %s to print a char argument usually causes a fatal execution-time logic error called an access violation. The conversion specification %s expects an argument of type pointer to char, so it treats the char's numeric value as a pointer. Such small numeric values often represent memory addresses that are restricted by the operating system.

### ✓ Self Check

1 (*Fill-In*) Conversion specifier s causes characters to be printed until a \_\_\_\_\_ is encountered.

**Answer:** terminating null ('\0') character.

2 (*True/False*) Compilers catch errors in the format-control string, so you will not experience incorrect results at runtime.

**Answer:** *False*. Actually, most compilers do not catch errors in the format-control string. You typically will not become aware of such errors until a program fails or produces incorrect results at runtime.

## 9.7 Other Conversion Specifiers

Consider the p and % conversion specifiers:

- p—Displays a pointer value in an implementation-defined manner.
- %—Displays the percent character.

Figure 9.4's %p prints ptr's value and x's address in an implementation-defined manner, typically using hexadecimal notation. Variables ptr and x have identical values because line 7 assigns x's address to ptr. The addresses displayed on your system will vary. The last printf statement uses %% to display the % character—%% is required because printf normally treats % as the beginning of a conversion specification. Trying to display a literal percent character using % rather than %% in the format control string is an error. When % appears in a format control string, it must be followed by a conversion specifier.



```
1 // fig09_04.c
2 // Using the p and % conversion specifiers
3 #include <stdio.h>
4
5 int main(void) {
6     int x = 12345;
7     int *ptr = &x;
8
9     printf("The value of ptr is %p\n", ptr);
10    printf("The address of x is %p\n\n", &x);
11
12    printf("Printing a %% in a format control string\n");
13 }
```

Fig. 9.4 | Using the p and % conversion specifiers. (Part I of 2.)

```
The value of ptr is 0x7ffff6eb911c
The address of x is 0x7ffff6eb911c
```

```
Printing a % in a format control string
```

**Fig. 9.4** | Using the p and % conversion specifiers. (Part 2 of 2.)

### ✓ Self Check

- 1 (*Fill-In*) A printf statement uses \_\_\_\_\_ to print the % character.

Answer: %.

- 2 (*True/False*) The conversion specifier p displays an address in decimal notation.

Answer: *False*. Actually, the conversion specifier p displays an address in an implementation-defined manner—typically, using hexadecimal notation.

## 9.8 Printing with Field Widths and Precision

The exact size of a field in which data is printed is specified by a **field width**. If the field width is larger than the data being printed, the data will normally be right-aligned within that field. An integer representing the field width is inserted between the percent sign (%) and the conversion specifier (e.g., %4d).

### 9.8.1 Field Widths for Integers

Figure 9.5 prints two groups of five numbers each, right-aligning those numbers containing fewer digits than the field width. Values wider than the field still display in full. Note that the minus sign for a negative value uses one character position in the field width. Field widths can be used with all conversion specifiers. Not providing a sufficiently large field width to handle a printed value can offset other data being printed, producing confusing outputs. Know your data!



```

1 // fig09_05.c
2 // Right-aligning integers in a field
3 #include <stdio.h>
4
5 int main(void) {
6     printf("%4d\n", 1);
7     printf("%4d\n", 12);
8     printf("%4d\n", 123);
9     printf("%4d\n", 1234);
10    printf("%4d\n\n", 12345);
11
12    printf("%4d\n", -1);
13    printf("%4d\n", -12);
14    printf("%4d\n", -123);
15    printf("%4d\n", -1234);
16    printf("%4d\n", -12345);
17 }
```

**Fig. 9.5** | Right-aligning integers in a field. (Part I of 2.)

```
1  
12  
123  
1234  
12345  
  
-1  
-12  
-123  
-1234  
-12345
```

Fig. 9.5 | Right-aligning integers in a field. (Part 2 of 2.)

### 9.8.2 Precisions for Integers, Floating-Point Numbers and Strings

Function `printf` also enables you to specify the precision with which data is printed. Precision has different meanings for different types:

- When used with integer conversion specifiers, precision indicates the minimum number of digits to be printed. If the printed value contains fewer digits than the specified precision and the precision value has a leading zero or decimal point, zeros are prefixed to the printed value until the total number of digits is equivalent to the precision. If neither a zero nor a decimal point is present in the precision value, spaces are inserted instead. The default precision for integers is 1.
- When used with floating-point conversion specifiers `e`, `E` and `f`, the precision is the number of digits to appear after the decimal point.
- When used with conversion specifiers `g` and `G`, the precision is the maximum number of significant digits to be printed.
- When used with conversion specifier `s`, the precision is the maximum number of characters written from the beginning of the string.

To use precision, place a decimal point (.), followed by an integer representing the precision between the percent sign and the conversion specifier. Figure 9.6 demonstrates the use of precision in format control strings. When you print a floating-point value with a precision smaller than the value's original number of decimal places, it's rounded.

```
1 // fig09_06.c  
2 // Printing integers, floating-point numbers and strings with precisions  
3 #include <stdio.h>  
4  
5 int main(void) {  
6     puts("Using precision for integers");  
7     int i = 873; // initialize int i  
8     printf("\t%.4d\n\t%.9d\n", i, i);
```

Fig. 9.6 | Printing integers, floating-point numbers and strings with precisions. (Part I of 2.)

```

9
10 puts("Using precision for floating-point numbers");
11 double f = 123.94536; // initialize double f
12 printf("\t%.3f\n\t%.3e\n\t%.3g\n\n", f, f, f);
13
14 puts("Using precision for strings");
15 char s[] = "Happy Birthday"; // initialize char array s
16 printf("\t%.11s\n", s);
17 }

```

Using precision for integers

```

0873
000000873

```

Using precision for floating-point numbers

```

123.945
1.239e+02
124

```

Using precision for strings

```

Happy Birth

```

**Fig. 9.6** | Printing integers, floating-point numbers and strings with precisions. (Part 2 of 2.)

### 9.8.3 Combining Field Widths and Precisions

The field width and the precision can be combined by placing the field width, followed by a decimal point, followed by a precision between the percent sign and the conversion specifier, as in the statement

```
printf("%9.3f", 123.456789);
```

which displays 123.457 with three digits to the right of the decimal point right-aligned in a nine-digit field.

### Specifying Field Widths and Precisions As Arguments

It's possible to specify the field width and the precision using integer expressions in the argument list following the format control string. To use this feature, insert an asterisk (\*) in place of the field width or precision (or both). The matching int argument in the argument list is evaluated and used in place of the asterisk. A field width's value may be either positive or negative (which causes the output to be left-aligned in the field, as described in the next section). The statement

```
printf("%*.*.f", 7, 2, 98.736);
```

uses 7 for the field width, 2 for the precision and outputs the value 98.74 right-aligned.



### Self Check

- I (*Multiple Choice*) Which of the following statements is *false*?
- The default precision for integers is 1.

- b) When used with floating-point conversion specifiers e, E and f, the precision is the number of digits to appear before the decimal point.
- c) When used with conversion specifiers g and G, the precision is the maximum number of significant digits to be printed.
- d) When used with conversion specifier s, the precision is the maximum number of characters written from the beginning of the string.

**Answer:** b) is *false*. When used with floating-point conversion specifiers e, E and f, the precision is the number of digits to appear after the decimal point.

**2 (What Does This Code Do?)** Describe precisely what the following code prints:

```
printf("%9.3f", 123.456789);
```

**Answer:** The code right-aligns in a nine-digit field the rounded value 123.457.

**3 (What Does This Code Do?)** Describe precisely what the following code prints:

```
printf("%*.2f", 7, 2, 98.736);
```

**Answer:** The statement uses 7 for the field width, 2 for the precision and outputs the value 98.74 right-aligned in a field of 7.

## 9.9 printf Format Flags

Function `printf` also provides *flags* to supplement its output formatting capabilities. The following table summarizes the five flags you can use in format control strings.

Flag	Description
- (minus sign)	Left-align the output within the specified field.
+	Display a plus sign preceding positive values and a minus sign preceding negative values.
space	Print a space before a positive value not printed with the + flag.
#	Prefix 0 to the output value when used with the octal conversion specifier o. Prefix 0x or 0X to the output value when used with the hexadecimal conversion specifiers x or X.
	Force a decimal point for a floating-point number printed with e, E, f, g or G that does not contain a fractional part. Normally, the decimal point is printed only if a digit follows it. For g and G specifiers, trailing zeros are not eliminated.
0 (zero)	Pad a field with leading zeros.

### 9.9.1 Right- and Left-Alignment

Flags in a conversion specification are placed immediately to the right of the % and before the format specifier. Several flags may be combined in one conversion specifier. Figure 9.7 demonstrates right-alignment and left-alignment of a string, an integer, a character and a floating-point number. Lines 6 and 8 output lines of numbers representing the column positions, so you can confirm that the right- and left-alignment worked correctly.

```

1 // fig09_07.c
2 // Right- and left-aligning values
3 #include <stdio.h>
4
5 int main(void) {
6     puts("1234567890123456789012345678901234567890");
7     printf("%10s%10d%10c%10f\n\n", "hello", 7, 'a', 1.23);
8     puts("1234567890123456789012345678901234567890");
9     printf("%-10s%-10d%-10c%-10f\n", "hello", 7, 'a', 1.23);
10 }
```

```
123456789012345678901234567890
hello      7      a  1.230000
```

```
123456789012345678901234567890
hello      7      a  1.230000
```

**Fig. 9.7** | Right- and left-aligning values.

### 9.9.2 Printing Positive and Negative Numbers with and without the + Flag

Figure 9.8 prints a positive number and a negative number, each with and without the `+ flag`. The minus sign is displayed in both cases, but the plus sign is displayed only when the `+ flag` is used.

```

1 // fig09_08.c
2 // Printing positive and negative numbers with and without the + flag
3 #include <stdio.h>
4
5 int main(void) {
6     printf("%d\n%d\n", 786, -786);
7     printf("%+d\n%+d\n", 786, -786);
8 }
```

```
786
-786
+786
-786
```

**Fig. 9.8** | Printing positive and negative numbers with and without the + flag.

### 9.9.3 Using the Space Flag

Figure 9.9 prefixes a space to the positive number with the `space flag`. This is useful for aligning positive and negative numbers with the same number of digits. The value `-547` is not preceded by a space in the output because of its minus sign.

```

1 // fig09_09.c
2 // Using the space flag
3 // not preceded by + or -
4 #include <stdio.h>
5
6 int main(void) {
7     printf("% d\n% d\n", 547, -547);
8 }
```

```

547
-547
```

**Fig. 9.9** | Using the space flag.

#### 9.9.4 Using the # Flag

Figure 9.10 uses the **# flag** to prefix 0 to the octal value and 0x and 0X to the hexadecimal values. For g, it forces the decimal point to print.

```

1 // fig09_10.c
2 // Using the # flag with conversion specifiers
3 // o, x, X and any floating-point specifier
4 #include <stdio.h>
5
6 int main(void) {
7     int c = 1427; // initialize c
8     printf("%#o\n", c);
9     printf("%#x\n", c);
10    printf("%#X\n", c);
11
12    double p = 1427.0; // initialize p
13    printf("\n%g\n", p);
14    printf("%#g\n", p);
15 }
```

```

02623
0x593
0X593

1427
1427.00
```

**Fig. 9.10** | Using the # flag with conversion specifiers.

#### 9.9.5 Using the 0 Flag

Figure 9.11 combines the **+** flag and the **0 (zero) flag** to print 452 in a nine-space field with a + sign and leading zeros, then prints 452 again using only the 0 flag and a nine-space field.

```

1 // fig09_11.c
2 // Using the 0 (zero) flag
3 #include <stdio.h>
4
5 int main(void) {
6     printf("%+09d\n", 452);
7     printf("%09d\n", 452);
8 }
```

```
+00000452
00000452
```

**Fig. 9.11** | Using the 0 (zero) flag.

### ✓ Self Check

- 1 (*Multiple Choice*) Which `printf` format flag is described by, “display a plus sign preceding positive values and a minus sign preceding negative values”?
- .
  - +.
  - 0).
  - None of the above.

**Answer:** b.

- 2 (*What Does This Code Do?*) Show precisely what the following code prints:

```

puts("1234567890123456789012345678901234567890");
printf("%10s%10d%10c%10f\n\n", "C18", 9, 'g', 6.41);
puts("1234567890123456789012345678901234567890");
printf("%-10s%-10d%-10c%-10f\n", "C18", 9, 'g', 6.41);
```

**Answer:**

```

1234567890123456789012345678901234567890
          C18      9      g   6.410000
1234567890123456789012345678901234567890
          C18      9      g   6.410000
```

- 3 (*What Does This Code Do?*) Show precisely what the following code prints:

```

printf("%d\n%d\n", 437, -437);
printf("%+d\n%+d\n", 437, -437);
```

**Answer:**

```

437
-437
+437
-437
```

## 9.10 Printing Literals and Escape Sequences

As you’ve seen throughout the book, literal characters included in the format control string are simply output by `printf`. However, there are several “problem” characters,

such as the *quotation mark* (") that delimits the format control string itself. Various control characters, such as *newline* and *tab*, must be represented by escape sequences. An escape sequence is represented by a backslash (\), followed by a particular escape character. The following table lists the escape sequences and the actions they cause.

Escape sequence	Description
\' (single quote)	Output the single quote (' ) character.
\\" (double quote)	Output the double quote (") character.
\? (question mark)	Output the question mark (?) character.
\\\ (backslash)	Output the backslash (\) character.
\a (alert or bell)	Cause an audible (bell) or visual alert (typically, flashing the window in which the program is running).
\b (backspace)	Move the cursor back one position on the current line.
\f (new page or form feed)	Move the cursor to the next logical page's start.
\n (newline)	Move the cursor to the beginning of the <i>next</i> line.
\r (carriage return)	Move the cursor to the beginning of the <i>current</i> line.
\t (horizontal tab)	Move the cursor to the next horizontal tab position.
\v (vertical tab)	Move the cursor to the next vertical tab position.



## Self Check

- 1 (*Multiple Choice*) Which of the following statements a), b) or c) is *false*?
- Literal characters included in the format control string are ignored by `printf`.
  - Various control characters, such as newline and tab, must be represented by escape sequences.
  - An escape sequence is represented by a backslash (\), followed by a particular escape character.
  - All of the above statements are *true*.

**Answer:** a) is *false*. Actually, `printf` displays any literal characters included in the format control string.

- 2 (*Multiple Choice*) Which escape sequence is described by “Cause an audible (bell) or visual alert (typically, flashing the window in which the program is running)”?

- \b.
- \r.
- \a.
- \v.

**Answer:** \a.

## 9.11 Formatted Input with `scanf`

Precise *input formatting* can be accomplished with `scanf`. Every `scanf` statement contains a format control string that describes the format of the data to be input. The

format control string consists of conversion specifiers and literal characters. Function `scanf` has the following input formatting capabilities:

1. Inputting all types of data.
2. Inputting specific characters from an input stream.
3. Skipping specific characters in the input stream.

### 9.11.1 `scanf` Syntax

Function `scanf` is written in the following form:

```
scanf(format-control-string, other-arguments);
```

The *format-control-string* describes the input formats, and *other-arguments* are pointers to variables in which the inputs will be stored.

When inputting data, prompt the user for one data item or a few data items at a time. Avoid asking the user to enter many data items in response to a single prompt. Always consider what the user and your program will do when incorrect data is entered—for example, a value for an integer that's nonsensical in a program's context, or a string with missing punctuation or spaces.

### 9.11.2 `scanf` Conversion Specifiers

The following table summarizes the conversion specifiers used to input all types of data. Note that the `d` and `i` conversion specifiers have different meanings for input with `scanf`, but are interchangeable for output with `printf`.

Conversion specifier	Description
<b>Integers</b>	
<code>d</code>	Read an optionally signed decimal integer. The corresponding argument is a pointer to an <code>int</code> .
<code>i</code>	Read an optionally signed decimal, octal or hexadecimal integer. The corresponding argument is a pointer to an <code>int</code> .
<code>o</code>	Read an octal integer. The corresponding argument is a pointer to an <code>unsigned int</code> .
<code>u</code>	Read an unsigned decimal integer. The corresponding argument is a pointer to an <code>unsigned int</code> .
<code>x</code> or <code>X</code>	Read a hexadecimal integer. The corresponding argument is a pointer to an <code>unsigned int</code> .
<code>h</code> , <code>l</code> and <code>ll</code>	Place before any integer conversion specifier to indicate that a <code>short</code> , <code>long</code> or <code>long long</code> integer is to be input.
<b>Floating-point numbers</b>	
<code>e</code> , <code>E</code> , <code>f</code> , <code>g</code> or <code>G</code>	Read a floating-point value. The corresponding argument is a pointer to a floating-point variable.

Conversion specifier	Description
<code>l</code> or <code>L</code>	Place before any floating-point conversion specifier to indicate that a <code>double</code> or <code>long double</code> value is to be input. The corresponding argument is a pointer to a <code>double</code> or <code>long double</code> variable.
<b>Characters and strings</b>	
<code>c</code>	Read a character. The corresponding argument is a pointer to a <code>char</code> ; no null (' <code>\0</code> ') is added.
<code>s</code>	Read a string. The corresponding argument is a pointer to an array of type <code>char</code> that's large enough to hold the string and a terminating null (' <code>\0</code> ') character—which is automatically added.
<b>Scan set</b>	
<code>[scan characters]</code>	Scan a string for a set of characters that are stored in an array.
<b>Miscellaneous</b>	
<code>p</code>	Read an address of the same form produced when an address is output with <code>%p</code> in a <code>printf</code> statement.
<code>n</code>	Store the number of characters input so far in this call to <code>scanf</code> . The corresponding argument must be a pointer to an <code>int</code> .
<code>%</code>	Skip a percent sign (%) in the input.

### 9.11.3 Reading Integers

Figure 9.12 reads integers with the various integer conversion specifiers and displays the integers as decimal numbers. Conversion specification `%i` can input decimal, octal and hexadecimal integers.

---

```

1 // fig09_12.c
2 // Reading input with integer conversion specifiers
3 #include <stdio.h>
4
5 int main(void) {
6     int a = 0;
7     int b = 0;
8     int c = 0;
9     int d = 0;
10    int e = 0;
11    int f = 0;
12    int g = 0;
13
14    puts("Enter seven integers: ");
15    scanf("%d%i%i%i%o%u%x", &a, &b, &c, &d, &e, &f, &g);
16
17    puts("\nThe input displayed as decimal integers is:");
18    printf("%d %d %d %d %d %d\n", a, b, c, d, e, f, g);
19 }
```

---

**Fig. 9.12** | Reading input with integer conversion specifiers. (Part I of 2.)

```
Enter seven integers:  
-70 -70 070 0x70 70 70 70
```

```
The input displayed as decimal integers is:  
-70 -70 56 112 56 70 112
```

**Fig. 9.12** | Reading input with integer conversion specifiers. (Part 2 of 2.)

#### 9.11.4 Reading Floating-Point Numbers

When inputting floating-point numbers, any of the floating-point conversion specifiers e, E, f, g or G can be used. Figure 9.13 reads three floating-point numbers, one with each of the three types of floating conversion specifiers, and displays all three numbers with conversion specifier f.

```
1 // fig09_13.c  
2 // Reading input with floating-point conversion specifiers  
3 #include <stdio.h>  
4  
5 int main(void) {  
6     double a = 0.0;  
7     double b = 0.0;  
8     double c = 0.0;  
9  
10    puts("Enter three floating-point numbers:");  
11    scanf("%le%lf%lg", &a, &b, &c);  
12  
13    puts("\nUser input displayed in plain floating-point notation:");  
14    printf("%f\n%f\n%f\n", a, b, c);  
15 }
```

```
Enter three floating-point numbers:  
1.27987 1.27987e+03 3.38476e-06
```

```
User input displayed in plain floating-point notation:  
1.279870  
1279.870000  
0.000003
```

**Fig. 9.13** | Reading input with floating-point conversion specifiers.

#### 9.11.5 Reading Characters and Strings

Characters and strings are input using the conversion specifiers c and s, respectively. Figure 9.14 prompts the user to enter a string. The program inputs the first character of the string with %c and stores it in the character variable x, then inputs the remainder of the string with %s and stores it in character array y.

```
1 // fig09_14.c
2 // Reading characters and strings
3 #include <stdio.h>
4
5 int main(void) {
6     char x = '\0';
7     char y[9] = "";
8
9     printf("%s", "Enter a string: ");
10    scanf("%c%8s", &x, y);
11
12    printf("The input was '%c' and \"%s\"\n", x, y);
13 }
```

```
Enter a string: Sunday
The input was 'S' and "unday"
```

Fig. 9.14 | Reading characters and strings.

### 9.11.6 Using Scan Sets

A sequence of characters can be input using a **scan set**—a set of characters enclosed in square brackets, [], and preceded by a percent sign in the format control string. A scan set scans the characters in the input stream, looking only for those characters that match characters contained in the scan set. Each time a character is matched, it's stored in the scan set's corresponding character array argument. The scan set stops inputting characters when `scanf` encounters a character not contained in the scan set. If the first character in the input stream does not match a character in the scan set, `scanf` does not modify its corresponding array argument. Figure 9.15 uses the scan set [aeiou] to scan the input stream for vowels. For our input "ooeeooahah", the first seven letters are input. The eighth letter (h) is not in the scan set, so `scanf` stops scanning for characters.

```
1 // fig09_15.c
2 // Using a scan set
3 #include <stdio.h>
4
5 int main(void) {
6     char z[9] = "";
7
8     printf("%s", "Enter string: ");
9     scanf("%8[aeiou]", z); // search for set of characters
10
11    printf("The input was \"%s\"\n", z);
12 }
```

Fig. 9.15 | Using a scan set. (Part I of 2.)

```
Enter string: ooeeooahah
The input was "ooeeooa"
```

**Fig. 9.15** | Using a scan set. (Part 2 of 2.)

### Inverting the Scan Set

An **inverted scan set** can scan for characters *not* contained in the scan set. To create an inverted scan set, place a **caret (^)** in the square brackets before the scan characters. When a character contained in the inverted scan set is encountered, input terminates. Figure 9.16 uses the inverted scan set `[^aeiou]` to search for “non-vowels.”

---

```

1 // fig09_16.c
2 // Using an inverted scan set
3 #include <stdio.h>
4
5 int main(void) {
6     char z[9] = "";
7
8     printf("%s", "Enter a string: ");
9     scanf("%8[^aeiou]", z); // inverted scan set
10
11    printf("The input was \"%s\"\n", z);
12 }
```

```
Enter a string: String
The input was "Str"
```

**Fig. 9.16** | Using an inverted scan set.

### 9.11.7 Using Field Widths

A field width can be used in a `scanf` conversion specifier to read a specific number of characters from the input stream. Figure 9.17 inputs a series of consecutive digits as a two-digit integer and an integer consisting of the remaining digits in the input stream.

---

```

1 // fig09_17.c
2 // Inputting data with a field width
3 #include <stdio.h>
4
5 int main(void) {
6     int x = 0;
7     int y = 0;
8
9     printf("%s", "Enter a six digit integer: ");
10    scanf("%2d%d", &x, &y);
```

**Fig. 9.17** | Inputting data with a field width. (Part 1 of 2.)

```
11
12     printf("The integers input were %d and %d\n", x, y);
13 }
```

```
Enter a six digit integer: 123456
The integers input were 12 and 3456
```

Fig. 9.17 | Inputting data with a field width. (Part 2 of 2.)

### 9.11.8 Skipping Characters in an Input Stream

You may want to skip certain characters in the input stream. Whitespace characters, such as space, newline and tab, at the beginning of a format control string skip all leading whitespace. Other literal characters ignore those characters at specific positions in the input. For example, your program might input a date as

```
11-10-1999
```

Each number in the date needs to be stored, but the dashes that separate the numbers can be discarded. To eliminate unnecessary characters, include them in `scanf`'s format control string. For example, to discard the dashes in the input, use the statement

```
scanf("%d-%d-%d", &month, &day, &year);
```

### Assignment Suppression Character

Although the preceding `scanf` does eliminate the dashes in the input, it's possible that the user might enter the date as

```
10/11/1999
```

In this case, the preceding `scanf` would not eliminate the unnecessary characters. For this reason, `scanf` provides the **assignment suppression character** `*`. This character enables `scanf` to read and discard data from the input without assigning it to a variable. Figure 9.18 uses the assignment suppression character in the `%c` conversion specification to indicate that a character appearing in the input stream should be read and discarded. Only the month, day and year are stored. We print the variable's values to demonstrate that they're input correctly. The argument lists for each `scanf` call do not contain variables for the conversion "`%*c`" specifiers containing the assignment suppression character. The corresponding characters are simply discarded.

```
1 // fig09_18.c
2 // Reading and discarding characters from the input stream
3 #include <stdio.h>
4
5 int main(void) {
6     int month = 0;
7     int day = 0;
8     int year = 0;
```

Fig. 9.18 | Reading and discarding characters from the input stream. (Part 1 of 2.)

```

9   printf("%s", "Enter a date in the form mm-dd-yyyy: ");
10  scanf("%d%c%d%c%d", &month, &day, &year);
11  printf("month = %d day = %d year = %d\n\n", month, day, year);
12
13  printf("%s", "Enter a date in the form mm/dd/yyyy: ");
14  scanf("%d%c%d%c%d", &month, &day, &year);
15  printf("month = %d day = %d year = %d\n", month, day, year);
16 }

```

Enter a date in the form mm-dd-yyyy: 07-04-2021  
 month = 7 day = 4 year = 2021

Enter a date in the form mm/dd/yyyy: 01/01/2021  
 month = 1 day = 1 year = 2021

**Fig. 9.18** | Reading and discarding characters from the input stream. (Part 2 of 2.)

### ✓ Self Check

- 1 (*Multiple Choice*) Which of the following statements is *false*?
- A scan set scans the characters in the input stream, looking only for those characters that match characters in the scan set.
  - Each time a character is matched, it's stored in the scan set's corresponding character array argument.
  - The scan set stops inputting characters when a character that's not contained in the scan set is encountered.
  - If the stream's first character matches a character in the scan set, `scanf` does not modify the corresponding array argument.

**Answer:** d) is *false*. Actually, if the stream's first character *does not match* a character in the scan set, `scanf` does not modify the corresponding array argument.

- 2 (*Fill-In*) A(n) \_\_\_\_\_ can be used in a `scanf` conversion specifier to read a specific number of characters from the input stream.

**Answer:** field width.

- 3 (*Fill-In*) The `scanf` \_\_\_\_\_ character \_\_\_\_\_ enables `scanf` to read and discard data from the input without assigning it to a variable.

**Answer:** assignment suppression, \*.

## SEC 9.12 Secure C Programming

The C standard lists many cases in which using incorrect library-function arguments can result in undefined behaviors. These can cause security vulnerabilities, so they should be avoided. Such problems can occur when using `printf` (or any of its variants, such as `sprintf`, `fprintf`, `printf_s`, etc.) with improperly formed conversion specifications. CERT rule FIO00-C (<https://wiki.sei.cmu.edu/>) discusses these issues. It presents a table showing the valid combinations of formatting flags, length modifiers and conversion-specifier characters that can be used to form conversion

specifications. The table also shows the proper argument type for each valid conversion specification. As you study *any* programming language, if the language specification says that doing something can lead to undefined behavior, avoid doing it to prevent security vulnerabilities.

## ✓ Self Check

- 1** (*True/False*) Undefined behaviors can cause security vulnerabilities, so they should be avoided.

**Answer:** *True*.

## Summary

### Section 9.2 Streams

- Input and output are performed with **streams** (p. 504), which are sequences of bytes.
- The **standard input stream** is connected to the keyboard. The **standard output and error streams** are connected to the computer screen (p. 504).
- Operating systems allow the standard streams to be **redirected** to other devices.

### Section 9.3 Formatting Output with **printf**

- A **format control string** (p. 505) describes the formats for values to output. It consists of **conversion specifiers, flags, field widths, precisions and literal characters**.
- A **conversion specification** (p. 505) consists of a % (p. 505) and a **conversion specifier**.

### Section 9.4 Printing Integers

- Integers are printed with the following conversion specifiers (p. 506): **d** or **i** for optionally signed integers, **o** for unsigned integers in octal form, **u** for unsigned integers in decimal form and **x** or **X** for unsigned integers in hexadecimal form. The modifiers **h**, **l** or **ll** are pre-fixed to the preceding conversion specifiers to indicate a **short**, **long** or **long long** integer.

### Section 9.5 Printing Floating-Point Numbers

- Floating-point values are printed with the following conversion specifiers: **e** or **E** (p. 508) for exponential notation, **f** (p. 508) for regular floating-point notation, and **g** or **G** for either **e** (or **E**) notation or **f** notation. When the **g** (or **G**, p. 508) conversion specifier is indicated, the **e** (or **E**) conversion specifier is used if the value's exponent is less than -4 or greater than or equal to the precision with which the value is printed.
- The precision for the **g** and **G** conversion specifiers indicates the maximum number of significant digits printed.

### Section 9.6 Printing Strings and Characters

- The conversion specifier **c** (p. 510) prints a character.
- The conversion specifier **s** (p. 510) prints a **string of characters** ending in the null character.

### Section 9.7 Other Conversion Specifiers

- The conversion specifier **p** (p. 511) displays an **address** in an implementation-defined manner (on many systems, hexadecimal notation is used).
- The conversion specifier **%%** (p. 511) causes a literal % to be output.

## Section 9.8 Printing with Field Widths and Precision

- If the field width (p. 505) is larger than the object being printed, the object is **right-aligned** by default.
- **Field widths** can be used with all conversion specifiers.
- **Precision** for integer conversion specifiers **i**, **I** and **u** indicates the minimum number of digits printed.
- **Precision** for floating-point conversion specifiers **e**, **E** and **f** indicates the number of digits after the decimal point. Precision for floating-point conversion specifiers **g** and **G** indicates the number of significant digits to appear.
- **Precision** for conversion specifier **s** indicates the number of characters to print.
- The **field width** and the **precision** can be combined by placing the field width, followed by a decimal point, followed by the precision between the percent sign and the conversion specifier.
- It's possible to specify the **field width** and the **precision** through **integer expressions** in the argument list following the format control string. To do so, use an asterisk (\*) for the field width or precision. The matching argument in the argument list is used in place of the asterisk.

## Section 9.9 printf Format Flags

- The **- flag** left-aligns its argument in a field.
- The **+ flag** (p. 516) prints a plus sign for positive values and a minus sign for negative values.
- The **space flag** (p. 516) prints a space preceding a positive value that's not displayed with the **+ flag**.
- The **# flag** (p. 517) prefixes **0** to octal values and **0x** or **0X** to hexadecimal values and forces the decimal point to be printed for floating-point values printed with **e**, **E**, **f**, **g** or **G**.
- The **0 flag** (p. 517) prints leading zeros for a value that does not occupy its entire field width.

## Section 9.10 Printing Literals and Escape Sequences

- Most **literal characters** to be printed in a **printf** statement can simply be included in the format control string. However, there are several "problem" characters, such as the quotation mark ("), p. 519) that delimits the format control string itself. Various **control characters**, such as newline and tab, must be represented by **escape sequences**. An escape sequence is represented by a backslash (\) followed by a particular escape character.

## Section 9.11 Formatted Input with scanf

- Input **formatting** is accomplished with the **scanf** library function.
- **scanf** inputs integers with the conversion specifiers **d** and **i** (p. 521) for optionally signed integers and **o**, **u**, **x** or **X** for unsigned integers in octal, decimal and hexadecimal formats. The modifiers **h**, **l** or **ll** are placed before an integer conversion specifier to input a **short**, **long** or **long long** integer.
- **scanf** inputs floating-point values with the conversion specifiers **e**, **E**, **f**, **g** or **G**. The modifiers **l** and **L** are placed before any of the floating-point conversion specifiers to indicate that the input value is a **double** or **long double** value.
- **scanf** inputs characters with the conversion specifier **c** (p. 522).
- **scanf** inputs strings with the conversion specifier **s** (p. 522).
- A **scanf** with a **scan set** (p. 523) scans the characters in the input, looking only for those characters that match characters contained in the scan set. Each matching character is stored in a character array. Input stops when a character not contained in the scan set is encountered.

- To create an **inverted scan set** (p. 524), place a caret (^) in the square brackets before the scan characters. `scanf` stores characters not appearing in the inverted scan set and stops when a character contained in the inverted scan set is encountered.
- `scanf` inputs **address values** with the conversion specifier `p`.
- Conversion specifier `n` stores the **number of characters input** so far in the current `scanf`. The corresponding argument is a pointer to `int`.
- The **assignment suppression character** (\*, p. 525) reads and discards data from the input stream.
- A **field width** is used in `scanf` to read a specific number of characters from the input stream.

## Self-Review Exercises

### 9.1 Fill-In the blanks in each of the following:

- Input and output are dealt with in the form of \_\_\_\_.
- The \_\_\_\_\_ stream is normally connected to the keyboard.
- The \_\_\_\_\_ stream is normally connected to the computer screen.
- Precise output formatting is accomplished with the \_\_\_\_\_ function.
- The format control string may contain \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_ and \_\_\_\_\_.
- The conversion specifier \_\_\_\_\_ or \_\_\_\_\_ may be used to output a signed decimal integer.
- The conversion specifiers \_\_\_\_\_, \_\_\_\_\_ and \_\_\_\_\_ display unsigned integers in octal, decimal and hexadecimal form.
- The modifiers \_\_\_\_\_ and \_\_\_\_\_ are placed before the integer conversion specifiers to display `short` or `long` integer values.
- The conversion specifier \_\_\_\_\_ displays a floating-point value in exponential notation.
- The modifier \_\_\_\_\_ is placed before any floating-point conversion specifier to display a `long double` value.
- The conversion specifiers `e`, `E` and `f` are displayed with \_\_\_\_\_ digits of precision to the decimal point's right if no precision is specified.
- The conversion specifiers \_\_\_\_\_ and \_\_\_\_\_ print strings and characters.
- All strings end in the \_\_\_\_\_ character.
- The field width and precision in a `printf` conversion specifier can be controlled with integer expressions by substituting a(n) \_\_\_\_\_ for the field width or for the precision and placing an integer expression in the corresponding argument.
- The \_\_\_\_\_ flag left-aligns output in a field.
- The \_\_\_\_\_ flag displays values with either a plus sign or a minus sign.
- Precise input formatting is accomplished with the \_\_\_\_\_ function.
- A(n) \_\_\_\_\_ scans a string for specific characters and stores the characters in an array.
- The conversion specifier \_\_\_\_\_ inputs optionally signed octal, decimal and hexadecimal integers.

- t) The conversion specifiers \_\_\_\_\_ can be used to input a `double` value.
- u) The \_\_\_\_\_ reads and discards data from the input stream without assigning it to a variable.
- v) A(n) \_\_\_\_\_ can be used in a `scanf` conversion specifier to indicate that a specific number of characters or digits should be read from the input stream.

**9.2** Find the error in each of the following and explain how it can be corrected.

- a) The following statement should print the character 'c'.

```
printf("%s\n", 'c');
```

- b) The following statement should print 9.375%.

```
printf("%.3f%", 9.375);
```

- c) The following statement should print the first character of "Monday".

```
printf("%c\n", "Monday");
```

- d) `puts("A string in quotes")`;

- e) `printf(%d%d, 12, 20)`;

- f) `printf("%c", "x")`;

- g) `printf("%s\n", 'Richard')`;

**9.3** Write a statement for each of the following:

- a) Print 1234 right-aligned in a 10-digit field.

- b) Print 123.456789 in exponential notation with a sign (+ or -) and 3 digits of precision.

- c) Read a `double` value into variable `number`.

- d) Print 100 in octal form preceded by 0.

- e) Read a string into character array `string`.

- f) Read characters into array `n` until a nondigit character is encountered.

- g) Use integer variables `x` and `y` to specify the field width and precision used to display the `double` value 87.4573.

- h) Read a value of the form 3.5%. Store the percentage in `float` variable `percent` and eliminate the % from the input stream. Do not use the assignment suppression character.

- i) Print 3.33333 as a `long double` value with a sign (+ or -) in a field of 20 characters with a precision of 3.

## Answers to Self-Review Exercises

**9.1** a) streams. b) standard input. c) standard output. d) `printf`. e) conversion specifiers, flags, field widths, precisions, literal characters. f) d, i, g) o, u, x (or X). h) h, l, i) e (or E). j) L. k) 6. l) s, c. m) NULL ('\0'). n) asterisk (\*). o) - (minus). p) + (plus). q) `scanf`. r) scan set. s) i. t) 1e, 1E, 1f, 1g or 1G. u) assignment suppression character (\*). v) field width.

**9.2** See the answers below:

- a) Error: Conversion specifier `s` expects an argument of type pointer to `char`.  
Correction: To print the character 'c', use the conversion specifier `%c` or change 'c' to "c".

- b) Error: Trying to print the literal character % without using the conversion specifier %%.  
Correction: Use %% to print a literal % character.
- c) Error: Conversion specifier c expects an argument of type char.  
Correction: To print the first character of "Monday" use the conversion specifier %.1s.
- d) Error: Trying to print the literal character " without using the \" escape sequence.  
Correction: Replace each quote in the inner set of quotes with \".
- e) Error: The format control string is not enclosed in double quotes.  
Correction: Enclose %d%d in double quotes.
- f) Error: The character x is enclosed in double quotes.  
Correction: Character constants to be printed with %c must be enclosed in single quotes.
- g) Error: The string to be printed is enclosed in single quotes.  
Correction: Use double quotes instead of single quotes to represent a string.

**9.3**

- a) `printf("%10d\n", 1234);`
- b) `printf("%+.3e\n", 123.456789);`
- c) `scanf("%1f", &number);`
- d) `printf("%#o\n", 100);`
- e) `scanf("%s", string);`
- f) `scanf("%[0123456789]", n);`
- g) `printf("%.*f\n", x, y, 87.4573);`
- h) `scanf("%f%", &percent);`
- i) `printf("%+20.3Lf\n", 3.333333);`

**Exercises****9.4**

- Write a `printf` or `scanf` statement for each of the following:
- a) Print unsigned integer 40000 left-aligned in a 15-digit field with 8 digits.
  - b) Read a hexadecimal value into variable hex.
  - c) Print 200 with and without a sign.
  - d) Print 100 in hexadecimal form preceded by 0x.
  - e) Read characters into array s until the letter p is encountered.
  - f) Print 1.234 in a 9-digit field with preceding zeros.
  - g) Read a time of the form hh:mm:ss, storing the parts of the time in the integer variables hour, minute and second. Skip the colons (:) in the input stream. Use the assignment suppression character.
  - h) Read a string of the form "characters" from the standard input. Store the string in character array s. Eliminate the quotation marks from the input.
  - i) Read a time of the form hh:mm:ss, storing the parts of the time in the integer variables hour, minute and second. Skip the colons (:) in the input stream. Do not use the assignment suppression character.

**9.5** Show what each of the following statements prints. If a statement is incorrect, indicate why.

- a) `printf("%-10d\n", 10000);`
- b) `printf("%c\n", "This is a string");`
- c) `printf("%.1f\n", 8, 3, 1024.987654);`
- d) `printf("%#o\n%#X\n%#e\n", 17, 17, 1008.83689);`
- e) `printf("% 1d\n%+1d\n", 1000000, 1000000);`
- f) `printf("%10.2E\n", 444.93738);`
- g) `printf("%10.2g\n", 444.93738);`
- h) `printf("%d\n", 10.987);`

**9.6** Find the error(s) in each of the following program segments. Explain how each error can be corrected.

- a) `printf("%s\n", 'Happy Birthday');`
- b) `printf("%c\n", 'Hello');`
- c) `printf("%c\n", "This is a string");`
- d) The following statement should print "Bon Voyage":  
`printf(""%s"" , "Bon Voyage");`
- e) `char day[] = "Sunday";`  
`printf("%s\n", day[3]);`
- f) `puts('Enter your name: ');`
- g) `printf(%f, 123.456);`
- h) The following statement should print the characters '0' and 'K':  
`printf("%s%s\n", '0', 'K');`
- i) `char s[10];`  
`scanf("%c", s[7]);`

**9.7** (*Differences Between %d and %i*) Write a program to test the difference between the %d and %i conversion specifiers when used in `scanf` statements. Ask the user to enter two integers separated by a space. Use the statements

```
scanf("%j%d", &x, &y);
printf("%d %d\n", x, y);
```

to input and print the values. Test the program with the following sets of input data:

10	10
-10	-10
010	010
0x10	0x10

**9.8** (*Printing Numbers in Various Field Widths*) Write a program that prints the integer value 12345 and the floating-point value 1.2345 in fields of various sizes. What happens when the values are printed in fields containing fewer digits than the values?

**9.9** (*Rounding Floating-Point Numbers*) Write a program that prints 100.453627 rounded to the nearest digit, tenth, hundredth, thousandth and ten-thousandth.

**9.10 (Temperature Conversions)** Write a program that converts integer Fahrenheit temperatures from 0 to 212 degrees to floating-point Celsius temperatures with 3 digits of precision. Perform the calculation using the formula

```
celsius = 5.0 / 9.0 * (fahrenheit - 32);
```

Display the output in two right-aligned columns of 10 characters each. Precede the Celsius temperatures by a sign for both positive and negative values.

**9.11 (Escape Sequences)** Write a program to test the escape sequences \', \", \?, \\, \a, \b, \n, \r and \t. For the escape sequences that move the cursor, print a character before and after printing the escape sequence so it's clear where the cursor has moved.

**9.12 (Printing a Question Mark)** Write a program that determines whether ? can be printed as part of a `printf` format control string as a literal character rather than using the \? escape sequence.

**9.13 (Reading an Integer with Each `scanf` Conversion Specifier)** Write a program that inputs the value 437 using each of the `scanf` integer conversion specifiers. Print each input value using all the integer conversion specifiers.

**9.14 (Outputting a Number with the Floating-Point Conversion Specifiers)** Write a program that uses each of the conversion specifiers e, f and g to input the value 1.2345. Print the values of each variable to prove that each conversion specifier can be used to input this same value.

**9.15 (Reading Strings in Quotes)** In some programming languages, strings are entered surrounded by either single or double quotation marks. Write a program that reads the three strings suzy, "suzy" and 'suzy'. Are the single and double quotes ignored by C or read as part of the string?

**9.16 (Printing a Question Mark as a Character Constant)** Write a program that determines whether ? can be printed as the character constant '?' rather than the character constant escape sequence '\?'. Use the conversion specifier %c in the format control string of a `printf` statement.

**9.17 (Using %g with Various Precisions)** Write a program that uses the conversion specifier g to output the value 9876.12345. Print the value with precisions ranging from 1 to 9.

*This page is intentionally left blank*