

# ENG1008 C Programming

## Topic 4

### ❖ Formatted Input / Output

# Learning Objectives

- 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`.

# Formatting Output with `printf` (1 of 3)

- Precise output formatting is accomplished with `printf`.
- 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**.

# Formatting Output with `printf` (2 of 3)

- Function `printf` can perform the following formatting capabilities, each of which is discussed in this chapter:
  - **Rounding** floating-point values to an indicated number of decimal places.
  - Aligning a column of numbers with decimal points appearing one above the other.
  - **Right justification** and **left justification** of outputs.
  - Inserting literal characters at precise locations in a line of output.
  - Representing floating-point numbers in exponential format.
  - Representing unsigned integers in octal and hexadecimal format. See Appendix C (textbook) for more information on octal and hexadecimal values.
  - Displaying all types of data with fixed-size field widths and precisions.

# Formatting Output with `printf` (3 of 3)

- The `printf` function has the form
  - `printf(format-control-string, other-arguments);`

**format-control-string** describes the output format, and **other-arguments** (which are optional) correspond to each conversion specification in format-control-string.
- Each conversion specification begins with a percent sign and ends with a conversion specifier.
- There can be many conversion specifications in one format control string.

# Printing Integers (1 of 2)

- An integer is a whole number, such as 776, 0 or -52, that contains no decimal point.
- Integer values are displayed in one of several formats.
- Figure 9.1 describes the **integer conversion specifiers**.

## Figure 9.1 Integer Conversion Specifiers

Conversion specifier	Description
d	Display as a <b>signed decimal integer</b> .
i	Display as a <b>signed decimal integer</b> . [Note: The i and d specifiers are <b>different</b> when used with scanf.]
o	Display as an <b>unsigned octal integer</b> .
u	Display as an <b>unsigned decimal integer</b> .
x or X	Display as an <b>unsigned hexadecimal integer</b> . X causes the digits 0-9 and the <b>uppercase</b> letters A-F to be used in the display and x causes the digits 0-9 and the <b>lowercase</b> letters a-f to be used in the display.
h, l or ll (letter “ell”)	Place <b>before</b> any integer conversion specifier to indicate that a short, long or long long integer is displayed, respectively. These are called <b>length modifiers</b> .

# Printing Integers (2 of 2)

- Figure 9.2 prints an integer using each of the integer conversion specifiers.
- Only the minus sign prints; plus signs are normally suppressed.
- Also, the value `-455`, when read by `%u` (line 15), is interpreted as an unsigned value `4294966841`.



# Figure 9.2 Using the Integer Conversion Specifiers (1 of 2)

```

1 // Fig. 9.2: fig09_02.c
2 // Using the integer conversion specifiers
3 #include <stdio.h>
4
5 int main(void)
6 {
7     printf("%d\n", 455);
8     printf("%i\n", 455); // i same as d in printf
9     printf("%d\n", +455); // plus sign does not print
10    printf("%d\n", -455); // minus sign prints
11    printf("%hd\n", 32000);
12    printf("%ld\n", 2000000000L); // L suffix makes literal a long int
13    printf("%o\n", 455); // octal
14    printf("%u\n", 455);
15    printf("%u\n", -455);
16    printf("%x\n", 455); // hexadecimal with lowercase letters
17    printf("%X\n", 455); // hexadecimal with uppercase letters
18 }

```

```

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

```

# Figure 9.2 Using the Integer Conversion Specifiers (2 of 2)

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

# Printing Floating-Point Numbers (1 of 7)

- A floating-point value contains a decimal point as in 33.5, 0.0 or -657.983.
- Floating-point values are displayed in one of several formats.
- Figure 9.3 describes the floating-point conversion specifiers.
- The **conversion specifiers e and E** display floating-point values in **exponential notation**—the computer equivalent of **scientific notation** used in mathematics.

# Printing Floating-Point Numbers (2 of 7)

- For example, the value 150.4582 is represented in scientific notation as
  - $1.504582 \times 10^2$
- and in exponential notation by the computer as
  - 1.504582E+02
- This notation indicates that 1.504582 is multiplied by 10 raised to the second power (E+02).
- The E stands for “exponent.”

# Figure 9.3 Floating-Point Conversion Specifiers

Conversion specifier	Description
e or E	Display a floating-point value in <b>exponential notation</b> .
f or F	Display floating-point values in <b>fixed-point notation</b> (F is supported in the Microsoft Visual C++ compiler in Visual Studio 2015 and higher).
g or G	Display a floating-point value in either the <b>floating-point form</b> f or the exponential form e (or E), based on the magnitude of the value.
L	Place before any floating-point conversion specifier to indicate that a <b>long double</b> floating-point value should be displayed.

# Printing Floating-Point Numbers (3 of 7)

- Values displayed with the conversion specifiers `e`, `E` and `f` show **six digits of precision** to the right of the decimal point by default (e.g., 1.04592); other precisions can be specified explicitly.
- **Conversion specifier `f`** always prints at least one digit to the **left** of the decimal point.
- Conversion specifiers `e` and `E` print **lowercase** `e` and **uppercase** `E`, respectively, preceding the exponent, and print **exactly one** digit to the left of the decimal point.
- **Conversion specifier `g` (or `G`)** prints in either `e`(`E`) or `f` format with no trailing zeros (1.234000 is printed as 1.234).

# Printing Floating-Point Numbers (4 of 7)

- Values are printed with e (E) 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 (**six significant digits** by default for g and G).
- Otherwise, conversion specifier f is used to print the value.
- Trailing zeros are **not** printed in the fractional part of a value output with g or G.

# Printing Floating-Point Numbers (5 of 7)

- 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.



# Printing Floating-Point Numbers (6 of 7)

- The precision for conversion specifiers `g` and `G` indicates the maximum number of significant digits printed, **including** the digit to the **left** of the decimal point.
- The value `1234567.0` is printed as `1.23457e+06`, using conversion specifier `%g` (remember that all floating-point conversion specifiers have a **default precision of 6**).
- There are six significant digits in the result.
- The difference between `g` and `G` is identical to the difference between `e` and `E` when the value is printed in exponential notation—lowercase `g` causes a lowercase `e` to be output, and uppercase `G` causes an uppercase `E` to be output.

# Printing Floating-Point Numbers (7 of 7)

- Figure 9.4 demonstrates each of the floating-point conversion specifiers.
- The %E, %e and %g conversion specifiers cause the value to be **rounded** in the output and the conversion specifier %f does **not**.
- With some compilers, the exponent in the outputs will be shown with two digits to the right of the + sign.

# Figure 9.4 Using the Floating-Point Conversion Specifiers

```
1 // Fig. 9.4: fig09_04.c
2 // Using the floating-point conversion specifiers
3 #include <stdio.h>
4
5 int main(void)
6 {
7     printf("%e\n", 1234567.89);
8     printf("%e\n", +1234567.89); // plus does not print
9     printf("%e\n", -1234567.89); // minus prints
10    printf("%E\n", 1234567.89);
11    printf("%f\n", 1234567.89); // six digits to right of decimal point
12    printf("%g\n", 1234567.89); // prints with lowercase e
13    printf("%G\n", 1234567.89); // prints with uppercase E
14 }
```

```
1.234568e+006
1.234568e+006
-1.234568e+006
1.234568E+006
1234567.890000
1.23457e+006
1.23457E+006
```

# 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` causes characters to be printed until a terminating null ( `'\0'` ) character is encountered.
- The program shown in Figure 9.5 displays characters and strings with conversion specifiers `c` and `s`.

# Figure 9.5 Using the Character and String Conversion Specifiers

```
1 // Fig. 9.5: fig09_05c
2 // Using the character and string conversion specifiers
3 #include <stdio.h>
4
5 int main(void)
6 {
7     char character = 'A'; // initialize char
8     printf("%c\n", character);
9
10    printf("%s\n", "This is a string");
11
12    char string[] = "This is a string"; // initialize char array
13    printf("%s\n", string);
14
15    const char *stringPtr = "This is also a string"; // char pointer
16    printf("%s\n", stringPtr);
17 }
```

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

## Figure 9.6 Other Conversion Specifiers

Conversion specifier	Description
p	Display a pointer value in an implementation-defined manner.
%	Display the percent character.

# Figure 9.7 Using the p and % Conversion Specifiers

```
1 // Fig. 9.7: fig09_07.c
2 // Using the p and % conversion specifiers
3 #include <stdio.h>
4
5 int main(void)
6 {
7     int x = 12345; // initialize int x
8     int *ptr = &x; // assign address of x to ptr
9
10    printf("The value of ptr is %p\n", ptr);
11    printf("The address of x is %p\n\n", &x);
12
13    printf("Printing a %% in a format control string\n");
14 }
```

The value of ptr is 002EF778

The address of x is 002EF778

Printing a % in a format control string

# Printing with Field Widths and Precision (1 of 6)

- 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 justified** within that field.
- An integer representing the field width is inserted between the percent sign (%) and the conversion specifier (e.g., %4d).
- Figure 9.8 prints two groups of five numbers each, right justifying those containing fewer digits than the field width.
- The field width is increased to print values wider than the field.
- 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.

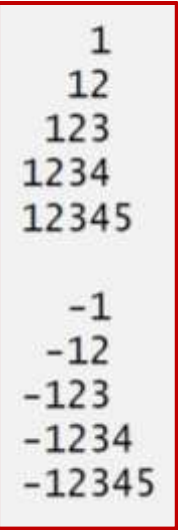


# Figure 9.8 Right Justifying Integers in a Field

(1 of 2)

```
1 // Fig. 9.8: fig09_08.c
2 // Right justifying integers in a field
3 #include <stdio.h>
4
5 int main(void)
6 {
7     printf("%4d\n", 1);
8     printf("%4d\n", 12);
9     printf("%4d\n", 123);
10    printf("%4d\n", 1234);
11    printf("%4d\n\n", 12345);
12
13    printf("%4d\n", -1);
14    printf("%4d\n", -12);
15    printf("%4d\n", -123);
16    printf("%4d\n", -1234);
17    printf("%4d\n", -12345);
18 }
```

4 spaces



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

# Figure 9.8 Right Justifying Integers in a Field (2 of 2)

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

# Printing with Field Widths and Precision (2 of 6)

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

# Printing with Field Widths and Precision (3 of 6)

- 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 to be written from the string**.

# Printing with Field Widths and Precision (4 of 6)

- To use precision, place a decimal point ( . ), followed by an integer representing the precision between the percent sign and the conversion specifier.
- Figure 9.9 demonstrates the use of precision in format control strings.
- When a floating-point value is printed with a precision smaller than the original number of decimal places in the value, the value is **rounded**.

# Figure 9.9 Printing Integers, Floating-Point Numbers and Strings with Precisions (1 of 2)

```

1  // Fig. 9.9: fig09_09.c
2  // Printing integers, floating-point numbers and strings with precisions
3  #include <stdio.h>
4
5  int main(void)
6  {
7      puts("Using precision for integers");
8      int i = 873; // initialize int i
9      printf("\t%.4d\n\t%.9d\n\n", i, i);
10
11     puts("Using precision for floating-point numbers");
12     double f = 123.94536; // initialize double f
13     printf("\t%.3f\n\t%.3e\n\t%.3g\n\n", f, f, f);
14
15     puts("Using precision for strings");
16     char s[] = "Happy Birthday"; // initialize char array s
17     printf("\t%.11s\n", s);
18 }

```

Annotations for the integer printf statement (line 9):

- `%.4d`: *blank space* (indicated by a red arrow pointing to the space before the first `8` in the output)
- `%.9d`: *0873* (indicated by a red arrow pointing to the output)

# Figure 9.9 Printing Integers, Floating-Point Numbers and Strings with Precisions (2 of 2)

Using precision for integers

```
0873  
000000873
```

Using precision for floating-point numbers

```
123.945    ← %0.3f  
1.239e+002 ← %0.3e  
124        ← %0.3g
```

Using precision for strings

```
Happy Birth ← stop at 11th character
```

# Printing with Field Widths and Precision (5 of 6)

- 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 justified in a nine-digit field.

- It's possible to specify the field width and the precision using integer expressions in the argument list following the format control string.



# Printing with Field Widths and Precision (6 of 6)

- 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 justified 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 justified.

# Using Flags in the `printf` Format Control String (1 of 6)

- Function `printf` also provides flags to supplement its output formatting capabilities.
- Five flags are available for use in format control strings (Figure 9.10).
- To use a flag in a format control string, place the flag immediately to the right of the percent sign.
- Several flags may be combined in one conversion specifier.

## Figure 9.10 Format-Control-String Flags

Flag	Description
- (minus sign)	<b>Left justify</b> the output within the specified field.
+	Display a <b>plus sign</b> preceding positive values and a <b>minus sign</b> preceding negative values.
<b>space</b>	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. <b>Force a decimal point</b> for a floating-point number printed with e, E, f, g or G that does <b>not</b> contain a fractional part. (Normally the decimal point is printed <b>only</b> if a digit follows it.) For g and G specifiers, trailing zeros are not eliminated.
0 (zero)	Pad a field with <b>leading zeros</b> .

# Using Flags in the `printf` Format Control String (2 of 6)



- Figure 9.11 demonstrates right justification and left justification of a string, an integer, a character and a floating-point number.

# Figure 9.11 Right Justifying and Left Justifying Values

```

1  // Fig. 9.11: fig09_11.c
2  // Right justifying and left justifying values
3  #include <stdio.h>
4
5  int main(void)
6  {
7      puts("1234567890123456789012345678901234567890\n");
8      printf("%10s%10d%10c%10f\n\n", "hello", 7, 'a', 1.23);
9      printf("%-10s%-10d%-10c%-10f\n", "hello", 7, 'a', 1.23);
10 }
```

↑  
left justify

1234567890	1234567890	1234567890	1234567890
hello	7	a	1.230000
hello	7	a	1.230000

# Using Flags in the `printf` Format Control String (3 of 6)

- Figure 9.12 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.

# Figure 9.12 Printing Positive and Negative Numbers with and Without the + Flag

```
1 // Fig. 9.12: fig09_12.c
2 // Printing positive and negative numbers with and without the + flag
3 #include <stdio.h>
4
5 int main(void)
6 {
7     printf("%d\n%d\n", 786, -786);
8     printf("%+d\n%+d\n", 786, -786);
9 }
```

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

# Using Flags in the `printf`

## Format Control String (4 of 6)

- Figure 9.13 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.



## Figure 9.13 Using the Space Flag

```
1 // Fig. 9.13: fig09_13.c
2 // Using the space flag
3 // not preceded by + or -
4 #include <stdio.h>
5
6 int main(void)
7 {
8     printf("%0d\n% d\n", 547, -547);
9 }
```

```
0547
-547
```

# Using Flags in the `printf` Format Control String (5 of 6)

- Figure 9.14 uses the **# flag** to prefix `0` to the octal value and `0x` and `0X` to the hexadecimal values, and to force the decimal point on a value printed with `g`.

# Figure 9.14 Using the # Flag with Conversion Specifiers

```
1 // Fig. 9.14: fig09_14.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 {
8     int c = 1427; // initialize c
9     printf("%#o\n", c);
10    printf("%#x\n", c);
11    printf("%#X\n", c);
12
13    double p = 1427.0; // initialize p
14    printf("\n%g\n", p);
15    printf("%#g\n", p);
16 }
```

02623  
0x593  
0X593

1427  
1427.00

# Using Flags in the `printf` Format Control String (6 of 6)

- Figure 9.15 combines the `+` flag and the **0 (zero) flag** to print 452 in a 9-space field with a `+` sign and leading zeros, then prints 452 again using only the `0` flag and a 9-space field.

## Figure 9.15 Using the 0 (Zero) Flag

```
1 // Fig. 9.15: fig09_15.c
2 // Using the 0 (zero) flag
3 #include <stdio.h>
4
5 int main(void)
6 {
7     printf("%+09d\n", 452);
8     printf("%09d\n", 452);
9 }
```

```
+00000452
000000452
```

# 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** (") 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.
- Figure 9.16 lists the escape sequences and the actions they cause.

# Escape Sequences

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 start of the next logical page.
\n (newline)	Move the cursor to the beginning of the <b>next</b> line.
\r (carriage return)	Move the cursor to the beginning of the <b>current</b> 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.

# Reading Formatted Input with `scanf` (1 of 3)

- 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:
  - Inputting all types of data.
  - Inputting specific characters from an input stream.
  - Skipping specific characters in the input stream.



# Reading Formatted Input with `scanf` (2 of 3)

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

- Function `scanf` is written in the following form:

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

# Reading Formatted Input with `scanf` (3 of 3)

- Figure 9.17 summarizes the conversion specifiers used to input all types of data, this is the same as the output conversion specifier

# Conversion Specifiers for scanf (1 of 3)

Conversion specifier	Description
<b>Integers</b>	
d	Read an <b>optionally signed decimal integer</b> . The corresponding argument is a pointer to an int.
i	Read an <b>optionally signed decimal, octal or hexadecimal integer</b> . The corresponding argument is a pointer to an int.
o	Read an <b>octal integer</b> . The corresponding argument is a pointer to an unsigned int.
u	Read an <b>unsigned decimal integer</b> . The corresponding argument is a pointer to an unsigned int.
x or X	Read a <b>hexadecimal integer</b> . The corresponding argument is a pointer to an unsigned int.
h, l and ll	Place <b>before</b> any of the integer conversion specifiers to indicate that a short, long or long long integer is to be input, respectively.

# Conversion Specifiers for scanf (2 of 3)

Conversion specifier	Description
<b>Floating-point numbers</b>	
e, E, f, g or G	Read a <b>floating-point value</b> . The corresponding argument is a pointer to a floating-point variable.
l or L	Place before any of the floating-point conversion specifiers to indicate that a double or long double value is to be input. The corresponding argument is a pointer to a <u>double</u> or <u>long double</u> variable. <div style="text-align: center;"> <span style="color: red; margin-right: 100px;"><u>%lf</u></span> <span style="color: red;"><u>%llf</u></span> </div>
<b>Characters and strings</b>	
c	Read a <b>character</b> . The corresponding argument is a pointer to a char; no null ('\0') is added.
s	Read a <b>string</b> . The corresponding argument is a pointer to an array of type char that's large enough to hold the string and a terminating null ('\0') character—which is automatically added.
<b>Scan set</b>	
[scan characters]	Scan a string for a set of characters that are stored in an array.

# Conversion Specifiers for `scanf` (3 of 3)

Conversion specifier	Description
<b>Miscellaneous</b>	
<code>p</code>	Read an <b>address</b> 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.

# Figure 9.18 Reading Input with Integer Conversion Specifiers (1 of 2)

```
1  // Fig. 9.18: fig09_18.c
2  // Reading input with integer conversion specifiers
3  #include <stdio.h>
4
5  int main(void)
6  {
7      int a;
8      int b;
9      int c;
10     int d;
11     int e;
12     int f;
13     int g;
14
15     puts("Enter seven integers: ");
16     scanf("%d%i%i%i%o%u%x", &a, &b, &c, &d, &e, &f, &g);
17
18     puts("\nThe input displayed as decimal integers is:");
19     printf("%d %d %d %d %d %d %d\n", a, b, c, d, e, f, g);
20 }
```

# Figure 9.18 Reading Input with Integer Conversion Specifiers (2 of 2)

*%d*      *%i*

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

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

*decimal*   *octal*   *hex*

*070 octal = 56 decimal*  
*0x70 hex = 112 decimal*

# Figure 9.19 Reading Input with Floating-Point Conversion Specifiers (1 of 2)

```
1 // Fig. 9.19: fig09_19.c
2 // Reading input with floating-point conversion specifiers
3 #include <stdio.h>
4
5 // function main begins program execution
6 int main(void)
7 {
8     double a;
9     double b;
10    double c;
11
12    puts("Enter three floating-point numbers:");
13    scanf("%le%lf%lg", &a, &b, &c);
14
15    printf("\nHere are the numbers entered in plain:");
16    puts("floating-point notation:\n");
17    printf("%f\n%f\n%f\n", a, b, c);
18 }
```



# Figure 9.19 Reading Input with Floating-Point Conversion Specifiers (2 of 2)

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

Here are the numbers entered in plain floating-point notation:  
1.279870  
1279.870000  
0.000003