



# C Formatted Input/Output

C How to Program, 6/e



## OBJECTIVES

In this chapter, you'll learn:

- To use input and output streams.
- To use all print formatting capabilities.
- To use all input formatting capabilities.
- To print with field widths and precisions.
- To use formatting flags in the `printf` format control string.
- To output literals and escape sequences.
- To format input using `scanf`.



- 9.1 Introduction
- 9.2 Streams
- 9.3 Formatting Output with `printf`
- 9.4 Printing Integers
- 9.5 Printing Floating-Point Numbers
- 9.6 Printing Strings and Characters
- 9.7 Other Conversion Specifiers
- 9.8 Printing with Field Widths and Precision
- 9.9 Using Flags in the `printf` Format Control String
- 9.10 Printing Literals and Escape Sequences
- 9.11 Reading Formatted Input with `scanf`



## 9.1 Introduction

- ▶ An important part of the solution to any problem is the presentation of the results.
- ▶ In this chapter, we discuss in depth the formatting features of `scanf` and `printf`.
- ▶ These functions input data from the `standard input stream` and output data to the standard output stream.
- ▶ Four other functions that use the standard input and standard output—`gets`, `puts`, `getchar` and `putchar`—were discussed in Chapter 8.
- ▶ Include the header `<stdio.h>` in programs that call these functions.



## 9.2 Streams

- ▶ All input and output is performed with **streams**, which are sequences of bytes.
- ▶ In input operations, the bytes flow from a device (e.g., a keyboard, a disk drive, a network connection) to main memory.
- ▶ In output operations, bytes flow from main memory to a device (e.g., a display screen, a printer, a disk drive, a network connection, and so on).
- ▶ When program execution begins, three streams are connected to the program automatically.



## 9.2 Streams (Cont.)

- ▶ Normally, the standard input stream is connected to the keyboard and the standard output stream is connected to the screen.
- ▶ Operating systems often allow these streams to be redirected to other devices.
- ▶ A third stream, the **standard error stream**, is connected to the screen.
- ▶ Error messages are output to the standard error stream.



## 9.3 Formatting Output with `printf`

- ▶ 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**.



## 9.3 Formatting Output with `printf` (Cont.)

- ▶ 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 for more information on octal and hexadecimal values.
  - Displaying all types of data with fixed-size field widths and precisions.



## 9.3 Formatting Output with `printf` (Cont.)

- ▶ 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.



## Common Programming Error 9.1

*Forgetting to enclose a format-control-string in quotation marks is a syntax error.*



## Good Programming Practice 9.1

*Format outputs neatly for presentation to make program outputs more readable and reduce user errors.*



## 9.4 Printing Integers

- ▶ 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**.



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

**Fig. 9.1** | Integer conversion specifiers.



## 9.4 Printing Integers (Cont.)

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



```
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 );
10     printf( "%d\n", -455 );
11     printf( "%hd\n", 32000 );
12     printf( "%ld\n", 2000000000L ); /* L suffix makes literal a long */
13     printf( "%o\n", 455 );
14     printf( "%u\n", 455 );
15     printf( "%u\n", -455 );
16     printf( "%x\n", 455 );
17     printf( "%X\n", 455 );
18     return 0; /* indicates successful termination */
19 } /* end main */
```

**Fig. 9.2** | Using integer conversion specifiers.



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

**Fig. 9.2** | Using integer conversion specifiers.



## Common Programming Error 9.2

*Printing a negative value with a conversion specifier that expects an unsigned value.*



## 9.5 Printing Floating-Point Numbers

- ▶ 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.



## 9.5 Printing Floating-Point Numbers (Cont.)

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



## 9.5 Printing Floating-Point Numbers (Cont.)

- ▶ 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).



## 9.5 Printing Floating-Point Numbers (Cont.)

- ▶ 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**.
- ▶ At least one decimal digit is required for the decimal point to be output.



## 9.5 Printing Floating-Point Numbers (Cont.)

- ▶ The values `0.0000875`, `8750000.0`, `8.75`, `87.50` and `875` are printed as `8.75e-05`, `8.75e+06`, `8.75`, `87.5` and `875` 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.



| Conversion specifier | Description                                                                                                                               |
|----------------------|-------------------------------------------------------------------------------------------------------------------------------------------|
| e or E               | Display a floating-point value in exponential notation.                                                                                   |
| f                    | Display floating-point values in fixed-point notation. [ <i>Note:</i> In C99, you can also use F.]                                        |
| g or G               | Display a floating-point value in either the floating-point form 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 long double floating-point value is displayed.                    |

**Fig. 9.3** | Floating-point conversion specifiers.



## 9.5 Printing Floating-Point Numbers (Cont.)

- ▶ 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 6 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.



### **Error-Prevention Tip 9.1**

*When outputting data, be sure that the user is aware of situations in which data may be imprecise due to formatting (e.g., rounding errors from specifying precisions).*



## 9.5 Printing Floating-Point Numbers (Cont.)

- ▶ 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.



---

```
1  /* Fig 9.4: fig09_04.c */
2  /* Printing floating-point numbers with
3     floating-point conversion specifiers */
4
5  #include <stdio.h>
6
7  int main( void )
8  {
9     printf( "%e\n", 1234567.89 );
10    printf( "%e\n", +1234567.89 );
11    printf( "%e\n", -1234567.89 );
12    printf( "%E\n", 1234567.89 );
13    printf( "%f\n", 1234567.89 );
14    printf( "%g\n", 1234567.89 );
15    printf( "%G\n", 1234567.89 );
16    return 0; /* indicates successful termination */
17 } /* end main */
```

---

**Fig. 9.4** | Using floating-point conversion specifiers.



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

**Fig. 9.4** | Using floating-point conversion specifiers.



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



### Common Programming Error 9.3

*Using %c to print a string is an error. The conversion specifier %c expects a char argument. A string is a pointer to char (i.e., a char \*).*



## Common Programming Error 9.4

*Using %s to print a char argument often causes a fatal execution-time error called an access violation. The conversion specifier %s expects an argument of type pointer to char.*



## Common Programming Error 9.5

*Using single quotes around character strings is a syntax error. Character strings must be enclosed in double quotes.*



## Common Programming Error 9.6

*Using double quotes around a character constant creates a pointer to a string consisting of two characters, the second of which is the terminating null.*



```
1  /* Fig 9.5: fig09_05c */
2  /* Printing strings and characters */
3  #include <stdio.h>
4
5  int main( void )
6  {
7      char character = 'A'; /* initialize char */
8      char string[] = "This is a string"; /* initialize char array */
9      const char *stringPtr = "This is also a string"; /* char pointer */
10
11     printf( "%c\n", character );
12     printf( "%s\n", "This is a string" );
13     printf( "%s\n", string );
14     printf( "%s\n", stringPtr );
15     return 0; /* indicates successful termination */
16 }
```

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

**Fig. 9.5** | Using the character and string conversion specifiers.



## 9.7 Other Conversion Specifiers

- ▶ The three remaining conversion specifiers are `p`, `n` and `%` (Fig. 9.6).
- ▶ The **conversion specifier** `%n` stores the number of characters output so far in the current `printf`—the corresponding argument is a pointer to an integer variable in which the value is stored—nothing is printed by a `%n`.
- ▶ The conversion specifier `%` causes a percent sign to be output.



| Conversion specifier | Description                                                                                                                                                                          |
|----------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| p                    | Display a pointer value in an implementation-defined manner.                                                                                                                         |
| n                    | Store the number of characters already output in the current <code>printf</code> statement. A pointer to an integer is supplied as the corresponding argument. Nothing is displayed. |
| %                    | Display the percent character.                                                                                                                                                       |

**Fig. 9.6** | Other conversion specifiers.



## 9.7 Other Conversion Specifiers (Cont.)

- ▶ Figure 9.7's `%p` prints the value of `ptr` and the address of `x`; these values are identical because `ptr` is assigned the address of `x`.
- ▶ Next, `%n` stores the number of characters output by the third `printf` statement (line 15) in integer variable `y`, and the value of `y` is printed.
- ▶ The last `printf` statement (line 21) uses `%%` to print the `%` character in a character string.



## 9.7 Other Conversion Specifiers (Cont.)

- ▶ Every `printf` call returns a value—either the number of characters output, or a negative value if an output error occurs.
- ▶ *[Note: This example will not execute in Microsoft Visual C++ because `%n` has been disabled by Microsoft “for security reasons.” To execute the rest of the program, remove lines 15–16.]*



### Portability Tip 9.1

*The conversion specifier `p` displays an address in an implementation-defined manner (on many systems, hexadecimal notation is used rather than decimal notation).*



## Common Programming Error 9.7

*Trying to print a literal percent character using % rather than %% in the format control string. When % appears in a format control string, it must be followed by a conversion specifier.*



```
1  /* Fig 9.7: fig09_07.c */
2  /* Using the p, n, and % conversion specifiers */
3  #include <stdio.h>
4
5  int main( void )
6  {
7      int *ptr; /* define pointer to int */
8      int x = 12345; /* initialize int x */
9      int y; /* define int y */
10
11     ptr = &x; /* assign address of x to ptr */
12     printf( "The value of ptr is %p\n", ptr );
13     printf( "The address of x is %p\n\n", &x );
14
15     printf( "Total characters printed on this line:%n", &y );
16     printf( " %d\n\n", y );
17
18     y = printf( "This line has 28 characters\n" );
19     printf( "%d characters were printed\n\n", y );
20
21     printf( "Printing a %% in a format control string\n" );
22     return 0; /* indicates successful termination */
23 } /* end main */
```

**Fig. 9.7** | Using the p, n and % conversion specifiers. (Part 1 of 2.)



```
The value of ptr is 0012FF78
The address of x is 0012FF78

Total characters printed on this line: 38

This line has 28 characters
28 characters were printed

Printing a % in a format control string
```

**Fig. 9.7** | Using the p, n and % conversion specifiers. (Part 2 of 2.)



## 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 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 and 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.



### Common Programming Error 9.8

*Not providing a sufficiently large field width to handle a value to be printed can offset other data being printed and can produce confusing outputs. Know your data!*



---

```
1  /* Fig 9.8: fig09_08.c */
2  /* Printing integers right-justified */
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     return 0; /* indicates successful termination */
19 }
```

---

**Fig. 9.8** | Right justifying integers in a field. (Part I of 2.)



```
1
12
123
1234
12345

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

**Fig. 9.8** | Right justifying integers in a field. (Part 2 of 2.)



## 9.8 Printing with Field Widths and Precision (Cont.)

- ▶ 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.



## 9.8 Printing with Field Widths and Precision (Cont.)

- ▶ 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.
- ▶ To use precision, place a decimal point (.), followed by an integer representing the precision between the percent sign and the conversion specifier.



## 9.8 Printing with Field Widths and Precision (Cont.)

- ▶ 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.



```
1  /* Fig 9.9: fig09_09.c */
2  /* Using precision while printing integers,
3     floating-point numbers, and strings */
4  #include <stdio.h>
5
6  int main( void )
7  {
8     int i = 873; /* initialize int i */
9     double f = 123.94536; /* initialize double f */
10    char s[] = "Happy Birthday"; /* initialize char array s */
11
12    printf( "Using precision for integers\n" );
13    printf( "\t%.4d\n\t%.9d\n\n", i, i );
14
15    printf( "Using precision for floating-point numbers\n" );
16    printf( "\t%.3f\n\t%.3e\n\t%.3g\n\n", f, f, f );
17
18    printf( "Using precision for strings\n" );
19    printf( "\t%.11s\n", s );
20    return 0; /* indicates successful termination */
21 } /* end main */
```

**Fig. 9.9** | Using precisions to display information of several types. (Part 1 of 2.)



Using precision for integers

0873

000000873

Using precision for floating-point numbers

123.945

1.239e+002

124

Using precision for strings

Happy Birth

**Fig. 9.9** | Using precisions to display information of several types. (Part 2 of 2.)



## 9.8 Printing with Field Widths and Precision (Cont.)

- ▶ 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.



## 9.8 Printing with Field Widths and Precision (Cont.)

- ▶ 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.



## 9.9 Using Flags in the `printf` Format Control String

- ▶ Function `printf` also provides flags to supplement its output formatting capabilities.
- ▶ Five flags are available for use in format control strings (Fig. 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.



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

**Fig. 9.10** | Format control string flags.



## 9.9 Using Flags in the printf Format Control String (Cont.)

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



```
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      printf( "%10s%10d%10c%10f\n\n", "hello", 7, 'a', 1.23 );
8      printf( "%-10s%-10d%-10c%-10f\n", "hello", 7, 'a', 1.23 );
9      return 0; /* indicates successful termination */
10 }
```

|       |   |   |          |
|-------|---|---|----------|
| hello | 7 | a | 1.230000 |
| hello | 7 | a | 1.230000 |

**Fig. 9.11** | Left justifying strings in a field.



## 9.9 Using Flags in the printf Format Control String (Cont.)

- ▶ 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.



```
1  /* Fig 9.12: fig09_12.c */
2  /* Printing 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      return 0; /* indicates successful termination */
10 }
```

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

**Fig. 9.12** | Printing positive and negative numbers with and without the + flag.



## 9.9 Using Flags in the printf Format Control String (Cont.)

- ▶ 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.



```
1  /* Fig 9.13: fig09_13.c */
2  /* Printing a space before signed values
3     not preceded by + or - */
4  #include <stdio.h>
5
6  int main( void )
7  {
8      printf( "% d\n% d\n", 547, -547 );
9      return 0; /* indicates successful termination */
10 }
```

```
547
-547
```

**Fig. 9.13** | Using the space flag.



## 9.9 Using Flags in the printf Format Control String (Cont.)

- ▶ 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.



---

```
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     double p = 1427.0; /* initialize p */
10
11     printf( "%#o\n", c );
12     printf( "%#x\n", c );
13     printf( "%#X\n", c );
14     printf( "\n%g\n", p );
15     printf( "%#g\n", p );
16     return 0; /* indicates successful termination */
17 } /* end main */
```

---

**Fig. 9.14** | Using the # flag. (Part I of 2.)



```
02623
0x593
0X593

1427
1427.00
```

**Fig. 9.14** | Using the # flag. (Part 2 of 2.)



## 9.9 Using Flags in the printf Format Control String (Cont.)

- ▶ 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.



```
1  /* Fig 9.15: fig09_15.c */
2  /* Printing with the 0( zero ) flag fills in leading zeros */
3  #include <stdio.h>
4
5  int main( void )
6  {
7      printf( "%+09d\n", 452 );
8      printf( "%09d\n", 452 );
9      return 0; /* indicates successful termination */
10 }
```

```
+00000452
000000452
```

**Fig. 9.15** | Using the 0 (zero) flag.



## 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 (") 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.



## Common Programming Error 9.9

*Attempting to print as literal data in a `printf` statement a single quote, double quote or backslash character without preceding that character with a backslash to form a proper escape sequence is an error.*



| 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.               |
| \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 next line.     |
| \r (carriage return)       | Move the cursor to the beginning of the current 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.     |

**Fig. 9.16** | Escape sequences.



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



## 9.11 Reading Formatted Input with `scanf` (Cont.)

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

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

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



## Good Programming Practice 9.2

*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.*



### Good Programming Practice 9.3

*Always consider what the user and your program will do when (not if) incorrect data is entered—for example, a value for an integer that is nonsensical in a program's context, or a string with missing punctuation or spaces.*



## 9.11 Reading Formatted Input with `scanf` (Cont.)

- ▶ Figure 9.17 summarizes the conversion specifiers used to input all types of data.
- ▶ The remainder of this section provides programs that demonstrate reading data with the various `scanf` conversion specifiers.

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

**Fig. 9.17** | Conversion specifiers for scanf. (Part I of 3.)



| Conversion specifier          | Description                                                                                                                                                                                                                                                  |
|-------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>Floating-point numbers</i> |                                                                                                                                                                                                                                                              |
| e, E, f, g or G               | Read a floating-point value. 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 <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. |
| <i>Characters and strings</i> |                                                                                                                                                                                                                                                              |
| c                             | Read a character. The corresponding argument is a pointer to a <code>char</code> ; no null ( <code>'\0'</code> ) is added.                                                                                                                                   |
| s                             | Read a string. The corresponding argument is a pointer to an array of type <code>char</code> that is large enough to hold the string and a terminating null ( <code>'\0'</code> ) character—which is automatically added.                                    |
| <i>Scan set</i>               |                                                                                                                                                                                                                                                              |
| [ <i>scan characters</i> ]    | Scan a string for a set of characters that are stored in an array.                                                                                                                                                                                           |

**Fig. 9.17** | Conversion specifiers for `scanf`. (Part 2 of 3.)

| Conversion specifier | Description                                                                                                           |
|----------------------|-----------------------------------------------------------------------------------------------------------------------|
| <i>Miscellaneous</i> |                                                                                                                       |
| p                    | Read an address of the same form produced when an address is output with %p in a printf statement.                    |
| n                    | Store the number of characters input so far in this call to scanf. The corresponding argument is a pointer to an int. |
| %                    | Skip a percent sign (%) in the input.                                                                                 |

**Fig. 9.17** | Conversion specifiers for scanf. (Part 3 of 3.)



## 9.11 Reading Formatted Input with `scanf` (Cont.)

- ▶ Figure 9.18 reads integers with the various integer conversion specifiers and displays the integers as decimal numbers.
- ▶ Conversion specifier `%i` is capable of inputting decimal, octal and hexadecimal integers.



---

```
1  /* Fig 9.18: fig09_18.c */
2  /* Reading integers */
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     printf( "Enter seven integers: " );
16     scanf( "%d%i%i%i%o%u%x", &a, &b, &c, &d, &e, &f, &g );
17
18     printf( "The input displayed as decimal integers is:\n" );
19     printf( "%d %d %d %d %d %d %d\n", a, b, c, d, e, f, g );
20     return 0; /* indicates successful termination */
21 }
```

---

**Fig. 9.18** | 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. 9.18** | Reading input with integer conversion specifiers. (Part 2 of 2.)



## 9.11 Reading Formatted Input with `scanf` (Cont.)

- ▶ When inputting floating-point numbers, any of the floating-point conversion specifiers `e`, `E`, `f`, `g` or `G` can be used.
- ▶ Figure 9.19 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`.
- ▶ The program output confirms the fact that floating-point values are imprecise—this is highlighted by the third value printed.



```
1  /* Fig 9.19: fig09_19.c */
2  /* Reading floating-point numbers */
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     printf( "Enter three floating-point numbers: \n" );
13     scanf( "%le%lf%lg", &a, &b, &c );
14
15     printf( "Here are the numbers entered in plain\n" );
16     printf( "floating-point notation:\n" );
17     printf( "%f\n%f\n%f\n", a, b, c );
18     return 0; /* indicates successful termination */
19 } /* end main */
```

**Fig. 9.19** | Reading input with floating-point conversion specifiers. (Part I 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
```

**Fig. 9.19** | Reading input with floating-point conversion specifiers. (Part 2 of 2.)



## 9.11 Reading Formatted Input with `scanf` (Cont.)

- ▶ Characters and strings are input using the conversion specifiers `C` and `S`, respectively.
- ▶ Figure 9.20 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  /* Fig 9.20: fig09_20.c */
2  /* Reading characters and strings */
3  #include <stdio.h>
4
5  int main( void )
6  {
7      char x;
8      char y[ 9 ];
9
10     printf( "Enter a string: " );
11     scanf( "%c%s", &x, y );
12
13     printf( "The input was:\n" );
14     printf( "the character \"%c\" ", x );
15     printf( "and the string \"%s\"\n", y );
16     return 0; /* indicates successful termination */
17 } /* end main */
```

Enter a string: Sunday  
The input was:  
the character "S" and the string "unday"

**Fig. 9.20** | Inputting characters and strings.



## 9.11 Reading Formatted Input with `scanf` (Cont.)

- ▶ A sequence of characters can be input using a **scan set**.
- ▶ A scan set is 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 argument—a pointer to a character array.
- ▶ The scan set stops inputting characters when a character that is not contained in the scan set is encountered.



## 9.11 Reading Formatted Input with `scanf` (Cont.)

- ▶ If the first character in the input stream does not match a character in the scan set, only the null character is stored in the array.
- ▶ Figure 9.21 uses the scan set `[aeiou]` to scan the input stream for vowels.
- ▶ Notice that the first seven letters of the input are read.
- ▶ The eighth letter (**h**) is not in the scan set and therefore the scanning is terminated.



```
1  /* Fig 9.21: fig09_21.c */
2  /* Using a scan set */
3  #include <stdio.h>
4
5  /* function main begins program execution */
6  int main( void )
7  {
8      char z[ 9 ]; /* define array z */
9
10     printf( "Enter string: " );
11     scanf( "%[aeiou]", z ); /* search for set of characters */
12
13     printf( "The input was \"%s\\n\"", z );
14     return 0; /* indicates successful termination */
15 }
```

Enter string: ooeoooahah  
The input was "ooeooa"

**Fig. 9.21** | Using a scan set.



## 9.11 Reading Formatted Input with `scanf` (Cont.)

- ▶ The scan set can also be used to scan for characters not contained in the scan set by using an **inverted scan set**.
- ▶ To create an inverted scan set, place a **caret (^)** in the square brackets before the scan characters.
- ▶ This causes characters not appearing in the scan set to be stored.
- ▶ When a character contained in the inverted scan set is encountered, input terminates.
- ▶ Figure 9.22 uses the inverted scan set `[^aeiou]` to search for consonants—more properly to search for “nonvowels.”



```
1  /* Fig 9.22: fig09_22.c */
2  /* Using an inverted scan set */
3  #include <stdio.h>
4
5  int main( void )
6  {
7      char z[ 9 ];
8
9      printf( "Enter a string: " );
10     scanf( "%[^aeiou]", z ); /* inverted scan set */
11
12     printf( "The input was \"%s\"\n", z );
13     return 0; /* indicates successful termination */
14 } /* end main */
```

Enter a string: String  
The input was "Str"

**Fig. 9.22** | Using an inverted scan set.



## 9.11 Reading Formatted Input with `scanf` (Cont.)

- ▶ A field width can be used in a `scanf` conversion specifier to read a specific number of characters from the input stream.
- ▶ Figure 9.23 inputs a series of consecutive digits as a two-digit integer and an integer consisting of the remaining digits in the input stream.



```
1  /* Fig 9.23: fig09_23.c */
2  /* inputting data with a field width */
3  #include <stdio.h>
4
5  int main( void )
6  {
7      int x;
8      int y;
9
10     printf( "Enter a six digit integer: " );
11     scanf( "%2d%d", &x, &y );
12
13     printf( "The integers input were %d and %d\n", x, y );
14     return 0; /* indicates successful termination */
15 }
```

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

**Fig. 9.23** | Inputting data with a field width.



## 9.11 Reading Formatted Input with `scanf` (Cont.)

- ▶ Often it's necessary to skip certain characters in the input stream.
- ▶ For example, a date could be entered 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 the format control string of `scanf` (**white-space** characters—such as space, newline and tab—skip all leading white-space).



## 9.11 Reading Formatted Input with `scanf` (Cont.)

- ▶ For example, to skip the dashes in the input, use the statement
  - `scanf( "%d-%d-%d", &month, &day, &year );`
- ▶ Although, this `scanf` does eliminate the dashes in the preceding input, it's possible that the date could be entered 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** `*`.



## 9.11 Reading Formatted Input with `scanf` (Cont.)

- ▶ The assignment suppression character enables `scanf` to read any type of data from the input and discard it without assigning it to a variable.
- ▶ Figure 9.24 uses the assignment suppression character in the `%C` conversion specifier to indicate that a character appearing in the input stream should be read and discarded.
- ▶ Only the month, day and year are stored.
- ▶ The values of the variables are printed to demonstrate that they are in fact input correctly.
- ▶ The argument lists for each `scanf` call do not contain variables for the conversion specifiers that use the assignment suppression character.
- ▶ The corresponding characters are simply discarded.



```
1  /* Fig 9.24: fig09_24.c */
2  /* Reading and discarding characters from the input stream */
3  #include <stdio.h>
4
5  int main( void )
6  {
7      int month1;
8      int day1;
9      int year1;
10     int month2;
11     int day2;
12     int year2;
13
14     printf( "Enter a date in the form mm-dd-yyyy: " );
15     scanf( "%d%c%d%c%d", &month1, &day1, &year1 );
16
17     printf( "month = %d  day = %d  year = %d\n\n", month1, day1, year1 );
18
19     printf( "Enter a date in the form mm/dd/yyyy: " );
20     scanf( "%d%c%d%c%d", &month2, &day2, &year2 );
21
22     printf( "month = %d  day = %d  year = %d\n", month2, day2, year2 );
23     return 0; /* indicates successful termination */
24 } /* end main */
```

**Fig. 9.24** | Reading and discarding characters from the input stream. (Part I of 2.)



```
Enter a date in the form mm-dd-yyyy: 11-18-2003  
month = 11  day = 18  year = 2003
```

```
Enter a date in the form mm/dd/yyyy: 11/18/2003  
month = 11  day = 18  year = 2003
```

**Fig. 9.24** | Reading and discarding characters from the input stream. (Part 2 of 2.)