Chapter 9
C Formatted Input/Output

C How to Program, 7/e

OBJECTIVES

In this chapter, you'll:

- Use input and output streams.
- Use all print formatting capabilities.
- Use all input formatting capabilities.
- Print with field widths and precisions.
- Use formatting flags in the `printf` format control string.
- Output literals and escape sequences.
- Format input using `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.
- 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.
- We'll show how to output error messages to the *standard error stream* in Chapter 11.

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.

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 uppercase letters `A-F` to be displayed and `x` causes the digits `0-9` and the lowercase letters `a-f` to be displayed.
`h`, `l` or `ll` (letter "ell")	Place *before* any integer conversion specifier to indicate that a `short`, `long` or `long long` integer is displayed, respectively. These are called length modifiers.

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 normally suppressed.
- Also, the value `-455`, when read by `%u` (line 15), is interpreted as an unsigned value `4294966841`.

Common Programming Error 9.2

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

```c
// Fig. 9.2: fig09_02.c
// Using the integer conversion specifiers
#include <stdio.h>

int main( void )
{
   printf( "%d\n", 455 );
   printf( "%i\n", 455 ); // i same as d in printf
   printf( "%d\n", +455 ); // plus sign does not print
   printf( "%d\n", -455 ); // minus sign prints
   printf( "%hd\n", 32000 );
   printf( "%ld\n", 2000000000L ); // L suffix makes literal a long int
   printf( "%o\n", 455 ); // octal
   printf( "%u\n", 455 );
   printf( "%u\n", -455 );
   printf( "%x\n", 455 ); // hexadecimal with lowercase letters
   printf( "%X\n", 455 ); // hexidecimal with uppercase letters
} // end main
```

Fig. 9.2 | Using the integer conversion specifiers. (Part 1 of 2.)

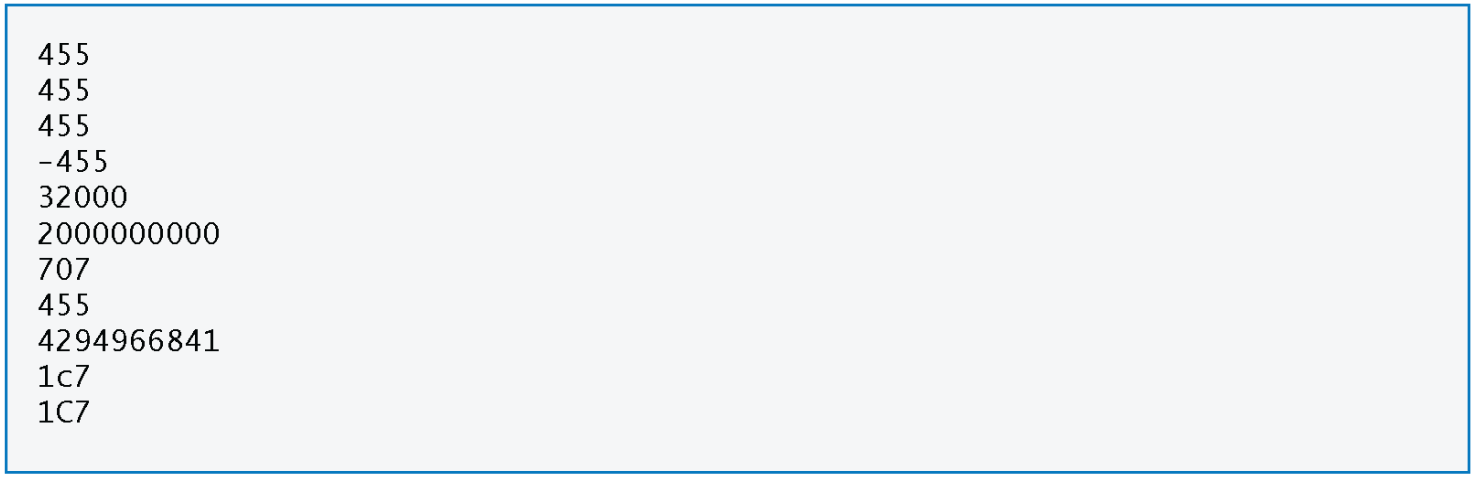

Fig. 9.2 | Using the integer conversion specifiers. (Part 2 of 2.)

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
  - 1.504582×10^2
- and 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."

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* (F is not supported in the Visual C++ compiler).
`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.)

- 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` 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.

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 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.

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.

```c
// Fig. 9.4: fig09_04.c
// Using the floating-point conversion specifiers
#include <stdio.h>

int main( void )
{
   printf( "%e\n", 1234567.89 );
   printf( "%e\n", +1234567.89 ); // plus does not print
   printf( "%e\n", -1234567.89 ); // minus prints
   printf( "%E\n", 1234567.89 );
   printf( "%f\n", 1234567.89 );
   printf( "%g\n", 1234567.89 );
   printf( "%G\n", 1234567.89 );
} // end main
```

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

Fig. 9.4 | Using the 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`.

```c
// Fig. 9.5: fig09_05c
// Using the character and string conversion specifiers
#include <stdio.h>

int main( void )
{
   char character = 'A'; // initialize char
   char string[] = "This is a string"; // initialize char array
   const char *stringPtr = "This is also a string"; // char pointer

   printf( "%c\n", character );
   printf( "%s\n", "This is a string" );
   printf( "%s\n", string );
   printf( "%s\n", stringPtr );
} // end main
```

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

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

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.

9.7 Other Conversion Specifiers

- Figure 9.6 shows the `p` and `%` conversion specifiers.
- 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`.
- The last `printf` statement uses `%%` to print the `%` character in a character string.

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.

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

Fig. 9.6 | Other conversion specifiers.

```c
// Fig. 9.7: fig09_07.c
// Using the p and % conversion specifiers
#include <stdio.h>

int main( void )
{
   int *ptr; // define pointer to int
   int x = 12345; // initialize int x

   ptr = &x; // assign address of x to ptr
   printf( "The value of ptr is %p\n", ptr );
   printf( "The address of x is %p\n\n", &x );

   puts( "Printing a %% in a format control string" );
} // end main
```

```
The value of ptr is 002EF778
The address of x is 002EF778

Printing a % in a format control string
```

Fig. 9.7 | Using the p and % conversion specifiers.

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.
- 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.

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!

```c
// Fig. 9.8: fig09_08.c
// Right justifying integers in a field
#include <stdio.h>

int main( void )
{
   printf( "%4d\n", 1 );
   printf( "%4d\n", 12 );
   printf( "%4d\n", 123 );
   printf( "%4d\n", 1234 );
   printf( "%4d\n\n", 12345 );

   printf( "%4d\n", -1 );
   printf( "%4d\n", -12 );
   printf( "%4d\n", -123 );
   printf( "%4d\n", -1234 );
   printf( "%4d\n", -12345 );
} // end main
```

Fig. 9.8 | Right justifying integers in a field. (Part 1 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 ma*ximum 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*.

9.8 Printing with Field Widths and Precision (Cont.)

- 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*.

```
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     int i = 873; // initialize int i
8     double f = 123.94536; // initialize double f
9     char s[] = "Happy Birthday"; // initialize char array s
10
11    puts( "Using precision for integers" );
12    printf( "\t%.4d\n\t%.9d\n\n", i, i );
13
14    puts( "Using precision for floating-point numbers" );
15    printf( "\t%.3f\n\t%.3e\n\t%.3g\n\n", f, f, f );
16
17    puts( "Using precision for strings" );
18    printf( "\t%.11s\n", s );
19 } // end main
```

Fig. 9.9 | Printing integers, floating-point numbers and strings with precisions. (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 | Printing integers, floating-point numbers and strings with precisions. (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.
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*.

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  } // end main
```

```
     hello         7         a  1.230000

hello     7         a         1.230000
```

Fig. 9.11 | Right justifying and left justifying values.

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.

```c
// Fig. 9.12: fig09_12.c
// Printing positive and negative numbers with and without the + flag
#include <stdio.h>

int main( void )
{
   printf( "%d\n%d\n", 786, -786 );
   printf( "%+d\n%+d\n", 786, -786 );
} // end main
```

```
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.

```c
// Fig. 9.13: fig09_13.c
// Using the space flag
// not preceded by + or -
#include <stdio.h>

int main( void )
{
   printf( "% d\n% d\n", 547, -547 );
} // end main
```

```
 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`.

```c
// Fig. 9.14: fig09_14.c
// Using the # flag with conversion specifiers
// o, x, X and any floating-point specifier
#include <stdio.h>

int main( void )
{
   int c = 1427; // initialize c
   double p = 1427.0; // initialize p

   printf( "%#o\n", c );
   printf( "%#x\n", c );
   printf( "%#X\n", c );
   printf( "\n%g\n", p );
   printf( "%#g\n", p );
} // end main
```

```
02623
0x593
0X593

1427
1427.00
```

Fig. 9.14 | Using the # flag with conversion specifiers.

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.

```c
// Fig. 9.15: fig09_15.c
// Using the 0( zero ) flag
#include <stdio.h>

int main( void )
{
   printf( "%+09d\n", 452 );
   printf( "%09d\n", 452 );
} // end main
```

```
+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.

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 9.1

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 9.2

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's 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
Integers	
`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`, `l` and `ll`	Place before any of the integer conversion specifiers to indicate that a `short`, `long` or `long long` integer is to be input, respectively.

Fig. 9.17 | Conversion specifiers for `scanf`. (Part 1 of 3.)

Conversion specifier	Description
Floating-point numbers	
`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 `double` or `long double` value is to be input. The corresponding argument is a pointer to a `double` or `long double` variable.
Characters and strings	
`c`	Read a *character*. The corresponding argument is a pointer to a `char`; no null (`'\0'`) is added.
`s`	Read a *string*. 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.
Scan set	
[*scan characters*]	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
Miscellaneous	
`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` can input decimal, octal and hexadecimal integers.

```c
// Fig. 9.18: fig09_18.c
// Reading input with integer conversion specifiers
#include <stdio.h>

int main( void )
{
   int a;
   int b;
   int c;
   int d;
   int e;
   int f;
   int g;

   puts( "Enter seven integers: " );
   scanf( "%d%i%i%i%o%u%x", &a, &b, &c, &d, &e, &f, &g );

   puts( "\nThe input displayed as decimal integers is:" );
   printf( "%d %d %d %d %d %d %d\n", a, b, c, d, e, f, g );
} // end main
```

Fig. 9.18 | Reading input with integer conversion specifiers. (Part 1 of 2.)

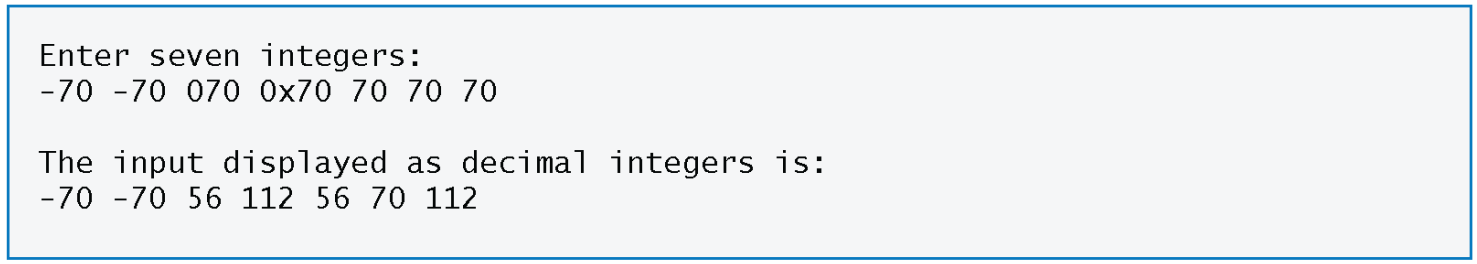

Fig. 9.18 | Reading input with integer conversion specifiers. (Part 2 of 2.)

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 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    puts( "\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 } // end main
```

Fig. 9.19 | Reading input with floating-point conversion specifiers. (Part 1 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`.

```c
// Fig. 9.20: fig09_20.c
// Reading characters and strings
#include <stdio.h>

int main( void )
{
   char x;
   char y[ 9 ];

   printf( "%s", "Enter a string: " );
   scanf( "%c%8s", &x, y );

   puts( "The input was:\n" );
   printf( "the character \"%c\" and the string \"%s\"\n", x, y );
} // end main
```

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

Fig. 9.20 | Reading 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's 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, the array is not modified.
- 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( "%s", "Enter string: " );
11    scanf( "%8[aeiou]", z ); // search for set of characters
12
13    printf( "The input was \"%s\"\n", z );
14 } // end main
```

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

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."

```c
// Fig. 9.22: fig09_22.c
// Using an inverted scan set
#include <stdio.h>

int main( void )
{
   char z[ 9 ];

   printf( "%s", "Enter a string: " );
   scanf( "%8[^aeiou]", z ); // inverted scan set

   printf( "The input was \"%s\"\n", z );
} // 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.

```c
// Fig. 9.23: fig09_23.c
// inputting data with a field width
#include <stdio.h>

int main( void )
{
   int x;
   int y;

   printf( "%s", "Enter a six digit integer: " );
   scanf( "%2d%d", &x, &y );

   printf( "The integers input were %d and %d\n", x, y );
} // end main
```

```
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` (whitespace characters—such as space, newline and tab—skip all leading whitespace).

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.)

- This 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're 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( "%s", "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( "%s", "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 } // end main
```

Fig. 9.24 | Reading and discarding characters from the input stream. (Part 1 of 2.)

```
Enter a date in the form mm-dd-yyyy: 11-18-2012
month = 11  day = 18  year = 2012

Enter a date in the form mm/dd/yyyy: 11/18/2012
month = 11  day = 18  year = 2012
```

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

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 (`www.securecoding.cert.org`) discusses these issues and presents a table showing the valid combinations of formatting flags, length modifiers and conversion-specifier characters that can be used to form conversion specifications.

9.12 Secure C Programming (Cont.)

- The table also shows the proper argument type for each valid conversion specification.
- In general, 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.