VIET NAM NATIONAL UNIVERSITY HO CHI MINH CITY

# Programming for Engineers

## Lecture 3: Iteration & I/O Operation

Course ID: EE057IU

# Lecture Outline

- Iteration (Chapter 4)
- I/O Operation (Chapter 9)

# Simple arithmetic operation in C

| Operation | Symbol | Example Syntax | Example Code | Explanation |
|---|---|---|---|---|
| Addition | + | result =  a + b | sum = a + b | Adds two numbers |
| Subtraction | - | result =  a - b | difference = a – b | Subtracts second number from first number |
| Multiplication | * | result =  a * b | product = a * b | Multiplies two numbers |
| Division | / | result =  a / b | quotient = a / b | Divides first number by second number |
| Modulus | % | result =  a % b | remainder = a % b | Gives remainder of division |

# Assignment operators

- C provides several assignment operators for abbreviating assignment expressions
- For example, the statement
    - $c = c + 3$
- can be abbreviated with the addition assignment operator += as
    - $c += 3$
- The += operator
    - adds the value of the expression on the right of the operator to the value of the variable on the left of the operator
    - and store the result in the variable on the left of the operator

# Assignment operators

- Any statement of the form
  - $variable = variable\ operator\ expression;$
  - $c = c + 3;$
- where the operator is one of the binary operators +,-,*,/ or %, can be written in the form
  - $variable\ operator =\ expression;$
  - $c += 3;$
- Thus, the assignment $c += 3;$ add 3 to c

# Comparison of Prefix and Postfix Increments

- Given: $i = 2$
- Calculate: $j = ++i$; and $j = i++$;

$$j = ++i;$$

Prefix:
Increment i and then use it

**After**

| i | j |
|---|---|
| 3 | 3 |

$$j = i++;$$

Postfix:
Use i and then increment it

| i | j |
|---|---|
| 3 | 2 |

# Assignment operators - examples

| Assignment | Sample expression | Explanation | Assigns |
|---|---|---|---|
| Assume: int c = 3, d = 5, e = 4, f = 6, g = 12; | | | |
| += | c+=7 | | |
| -= | d-=4 | | |
| *= | e*=5 | | |
| /= | f/=3 | | |
| %= | g%=9 | | |

# Unary increment & decrement operators

| Operator | Sample Expression | Explanation |
|---|---|---|
| ++ | ++a | Increment ***a*** by 1, then use the new value of ***a*** in the expression in which ***a*** resides. |
| ++ | a++ | Use the current value of ***a*** in the expression in which ***a*** resides, then increment ***a*** by 1. |
| -- | --b | Decrement ***b*** by 1, then use the new value of ***b*** in the expression in which ***b*** resides. |
| -- | b-- | Use the current value of ***b*** in the expression in which ***b*** resides, then decrement ***b*** by 1. |

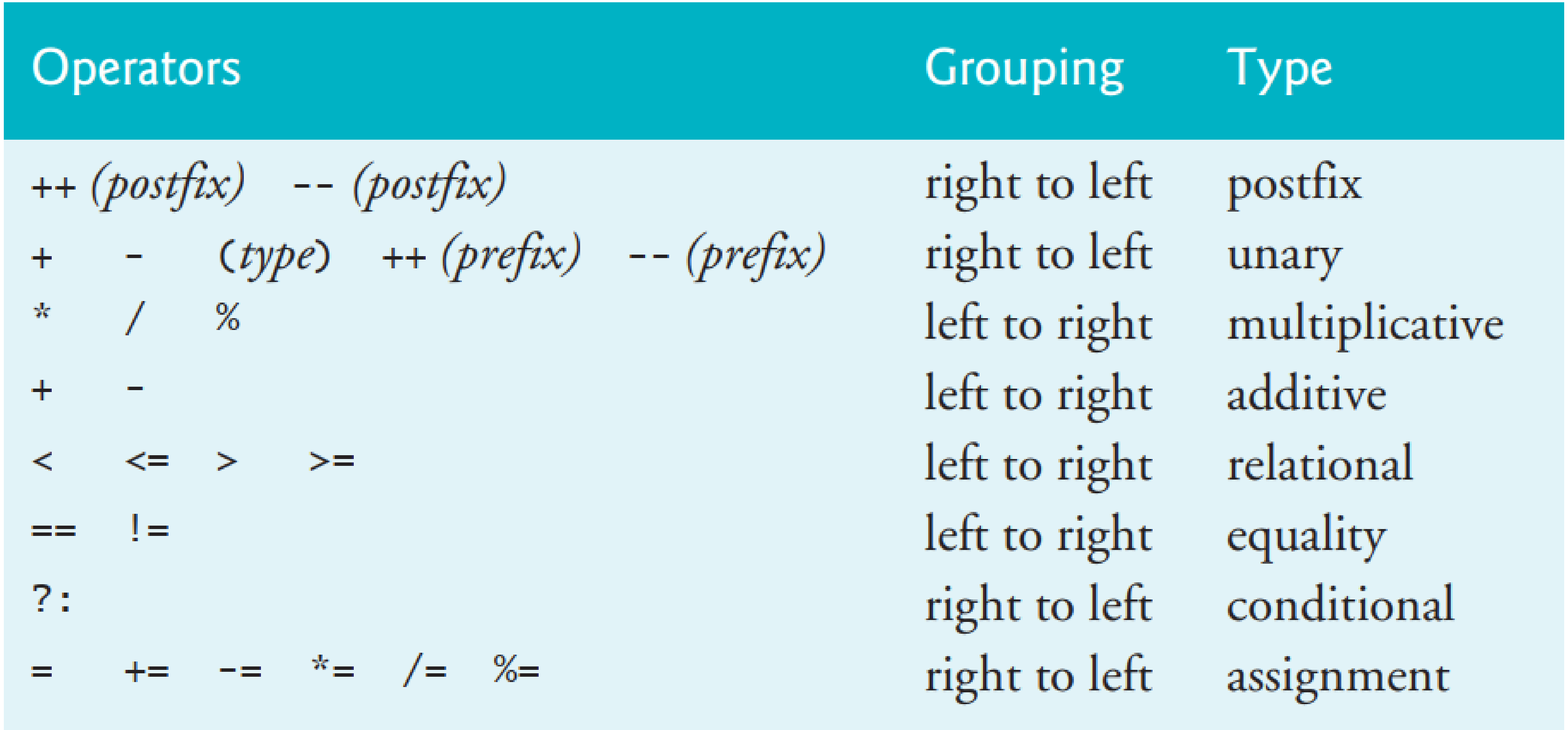
A **standalone statement**, like *a*++ or ++*a*

They both have the same effect: they simply increment the value of *a* by one

# Increment example

```
#include <stdio.h>

int main(void) {
    int a = 5;
    int b = 5;
    int result1, result2;

    result1 = ++a;                      // a is incremented to 6, then result1 is assigned the value of a (6)
    result2 = b++;                      // result2 is assigned the current value of b (5), then b is incremented to 6
    printf("a: %d\n", a);               // a is 6
    printf("result1: %d\n", result1);    // result1 is 6
    printf("b: %d\n", b);                // b is 6
    printf("result2: %d\n", result2);   // result2 is 5
 }
```

# Precedence



| Operators | Grouping | Type |
|---|---|---|
| ++ *(postfix)* -- *(postfix)* | right to left | postfix |
| + - *(type)* ++ *(prefix)* -- *(prefix)* | right to left | unary |
| * / % | left to right | multiplicative |
| + - | left to right | additive |
| < <= > >= | left to right | relational |
| == != | left to right | equality |
| ?: | right to left | conditional |
| = += -= *= /= %= | right to left | assignment |

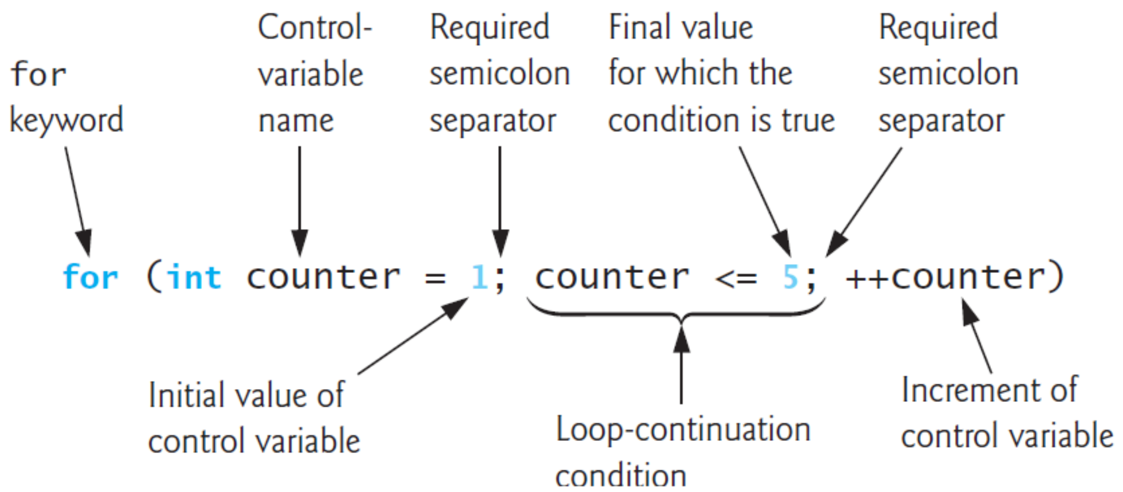
# *for* Iteration Statement

**Pseudocode**

```
for (initialization expression; loop  repetition condition; update expression)
{
            statement;
}
```

**Example**

```
for (count_star = 0; count_star<N; count_star++)
            printf("*")
```



```
for (int counter = 1; counter <= 5; ++counter)
```

for keyword
Control-variable name
Required semicolon separator
Final value for which the condition is true
Required semicolon separator
Initial value of control variable
Loop-continuation condition
Increment of control variable

# $for$ Iteration Statement

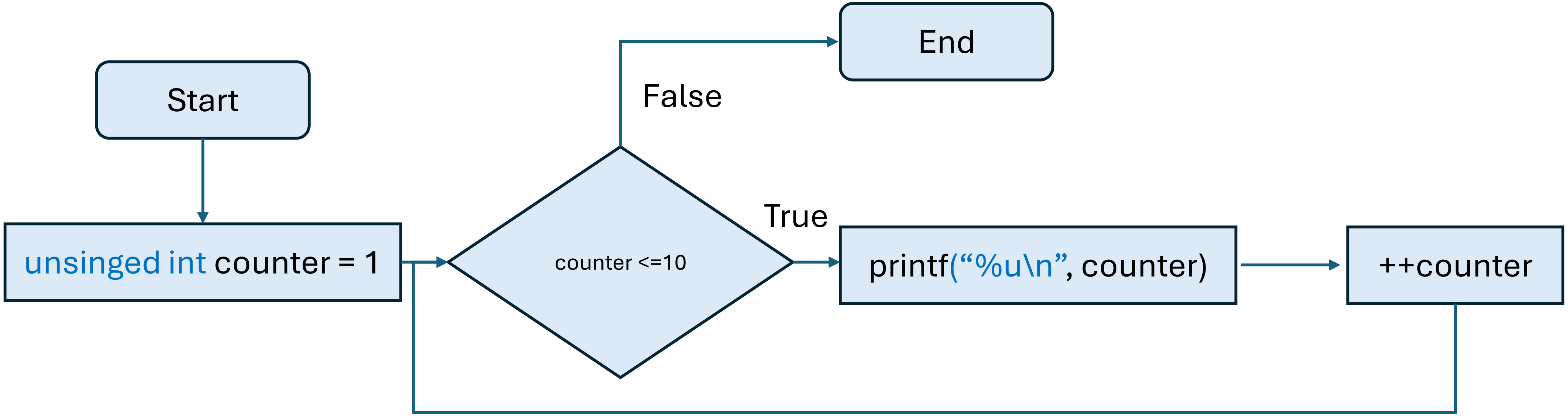
- The general format of the for statement

```
for (initialization; condition; update expression){
        statement
}
```

- The initialization expression initializes the loop-control variable (and might define it)
- The condition expression is the loop-continuation condition and
- The update expression increments the control variable

# $for$ Iteration Statement - flowchart

```
#include <stdio.h>

int main(void) {
        for (unsigned int counter = 1; counter <=10; ++counter){
                printf(“%u\n”, counter)
        }
}
```



# $for$ Iteration Statement - Common Error

- ***Off-By-One Errors***
- Notice the program condition when using <= or < only
  - if initialize counter = 1 and counter <= 10, loop is executed 10 times
  - if initialize counter = 1 and counter < 10, loop is executed 9 times

- These codes are the same, but in C, it is recommended to start the loop from 0

```
for (int i=0; i<10; i++)
    printf("It will be printed 10 times. \n");
for (int i=1; i<=10; i++)
    printf("It will be printed 10 times. \n");
```

# Examples using the $for$ statement

1.Vary the control variable from 1 to 100 in increments of 1.

$$for\ (int\ i\ =\ 1;\ i\ <=\ 100;++i)$$

2. Vary the control variable from 100 to 1 in increments of -1 (i.e., *decrements* of 1).

$$for\ (int\ i\ =\ 100;\ i\ >=\ 1;--i)$$

3. Vary the control variable from 7 to 77 in increments of 7.

$$for\ (int\ i\ =\ 7;\ i\ <=\ 77;\ i\ +=\ 7)$$

4. Vary the control variable from 20 to 2 in increments of -2.

$$for\ (int\ i\ =\ 20;\ i\ >=\ 2;\ i\ -=\ 2)$$

5. Vary the control variable over the values 2, 5, 8, 11, 14 and 17.

$$for\ (int\ j\ =\ 2;\ j\ <=\ 17;\ j\ +=\ 3)$$

6. Vary the control variable over the following sequence of values: 44, 33, 22, 11, 0.

$$for\ (int\ j\ =\ 44;\ j\ >=\ 0;\ j\ -=\ 11)$$

# Nested $for$ Loop

```
int row, col;
for (row = 0; row<2; row++){
        for (col = 0; col<3; col++){
                printf(“Pixel coordinate (%d, %d)\n”, row, col);
        }
}
```

# Nested $for$ Loop

```
int row, col;
for (row = 0; row<2; row++){
            for (col = 0; col<3; col++){
                        printf("Pixel coordinate (%d, %d)\n", row, col);
            }
}
```

**Output:**

```
Pixel coordinate (0,0)
Pixel coordinate (0,1)
Pixel coordinate (0,2)
Pixel coordinate (1,0)
Pixel coordinate (1,1)
Pixel coordinate (1,2)
```

# $for$ Iteration Statement – Book example

- Consider the following problem statement:
  - *A person invests $1000.00 in a savings account yielding 5% interest. Assuming all interest is left on deposit in the account, calculate and print the amount of money in the account at the end of each year for 10 years. Use the following formula for determining these amounts:*

$$a = p(1+r)^n$$

*where*

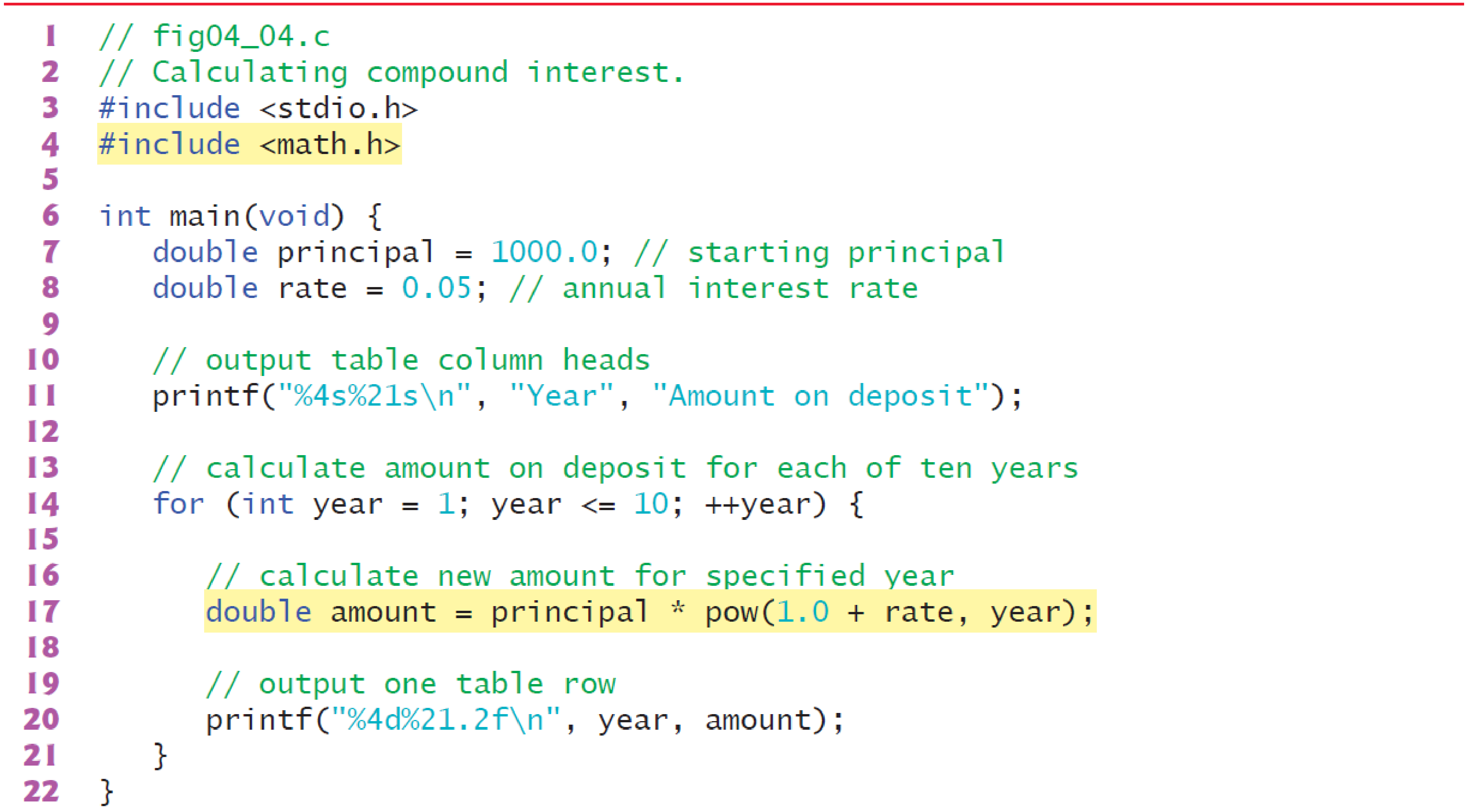
$\boldsymbol{p}$ *is the original amount invested (i.e., the principal, $1000.00 here),*

$\boldsymbol{r}$ *is the annual interest rate (for example, .05 for 5%),*

$\boldsymbol{n}$ *is the number of years, which is 10 here, and*

$\boldsymbol{a}$ *is the amount on deposit at the end of the nth year.*

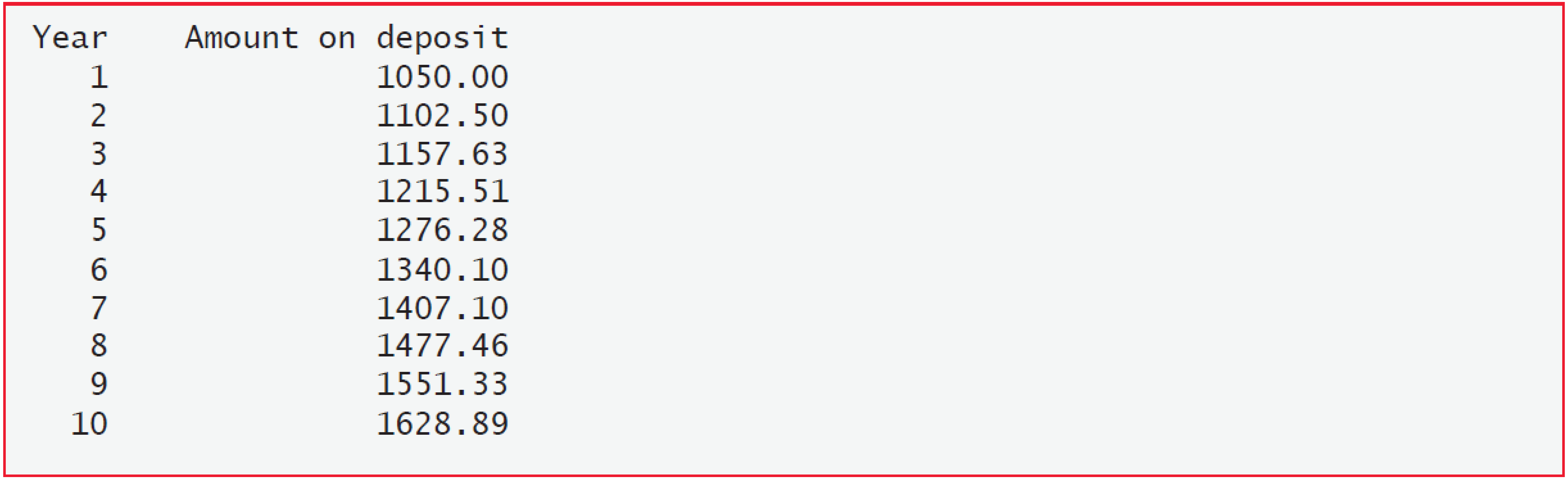
# $for$ Iteration Statement – Book example



```c
// fig04_04.c
// Calculating compound interest.
#include <stdio.h>
#include <math.h>

int main(void) {
   double principal = 1000.0; // starting principal
   double rate = 0.05; // annual interest rate

   // output table column heads
   printf("%4s%21s\n", "Year", "Amount on deposit");

   // calculate amount on deposit for each of ten years
   for (int year = 1; year <= 10; ++year) {

      // calculate new amount for specified year
      double amount = principal * pow(1.0 + rate, year);

      // output one table row
      printf("%4d%21.2f\n", year, amount);
   }
}
```

# $for$ Iteration Statement – Book example

## Output:



```
Year     Amount on deposit
   1               1050.00
   2               1102.50
   3               1157.63
   4               1215.51
   5               1276.28
   6               1340.10
   7               1407.10
   8               1477.46
   9               1551.33
  10               1628.89
```

# $for$ Iteration Statement – In-class Practice

- Write a C program to add the even number only from the range 0-50

  → Write a Pseudocode (3 minutes)

  → Draw the flowchart (3 minutes)

  → Write C code (4 minutes)

# In-class practice

Write a C program to sum the even integer from 2 to 50

# $while, do\ \ldots while$ Iteration Statement

- $while$ is an iteration statement repeats an action while some condition remains *true*. If condition is *false*, break the loop.

  *While there are more items on my shopping list*

  *Purchase next item and cross it off my list*

- $do\ \ldots\ while$ is similar to the $while$ statement

# $while, do$ …$while$ Iteration Statement

## Syntax

```
while (condition){
        statement;
}
```

→ The loop-continuation condition is tested at the beginning of the loop

```
do {
        statement;
} while (condition)
```

→ The loop-continuation condition is tested after the loop is performed
→ The loop body will be executed at least one.

# $while, do\ ...while$ Iteration Statement

## Example

```
#include <stdio.h>

int main(void){

    int product = 3;
    while (product<=100)
    {
        product = 3*product;
    }
}
```

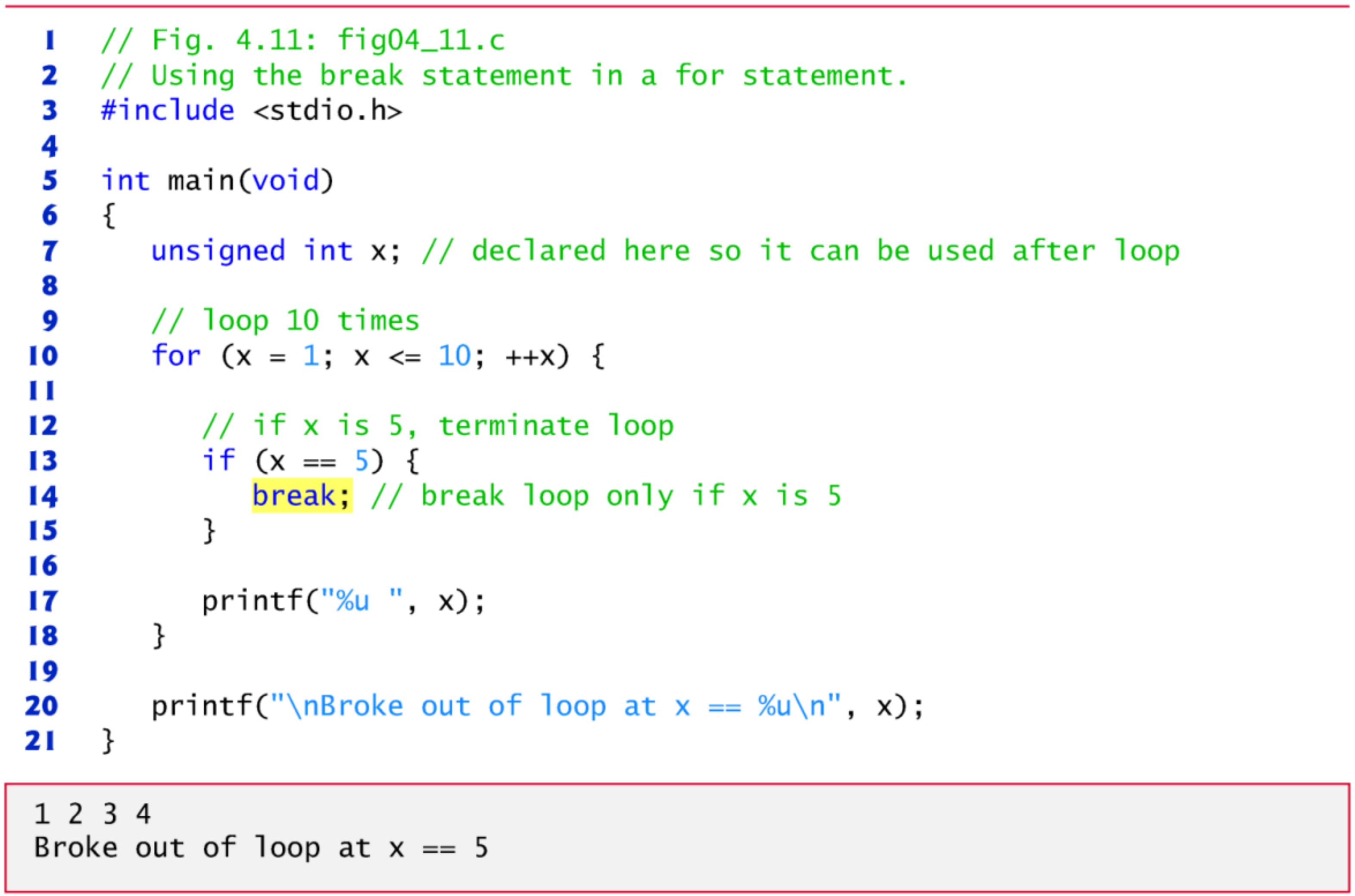
```
#include <stdio.h>

int main(void){

    unsigned int counter = 1;
    do {
        printf(“%u \n”, counter);
    } while (++counter<=10);
}
```

# $break$ and $continue$ statements

- $break$
  - Used inside $while, for\ , do\ \ \ldots\ \ while, switch$ statements
  - When executed, program exits the statements

- $continue$
  - Used inside $while, for\ , do\ \ \ldots\ \ while$ statements
  - When executed, the loop-continuation test is evaluated immediately after the $continue$ statement is executed.
  - In the $for$ statement, the increment expression is executed, then the loop-continuation test is evaluated.

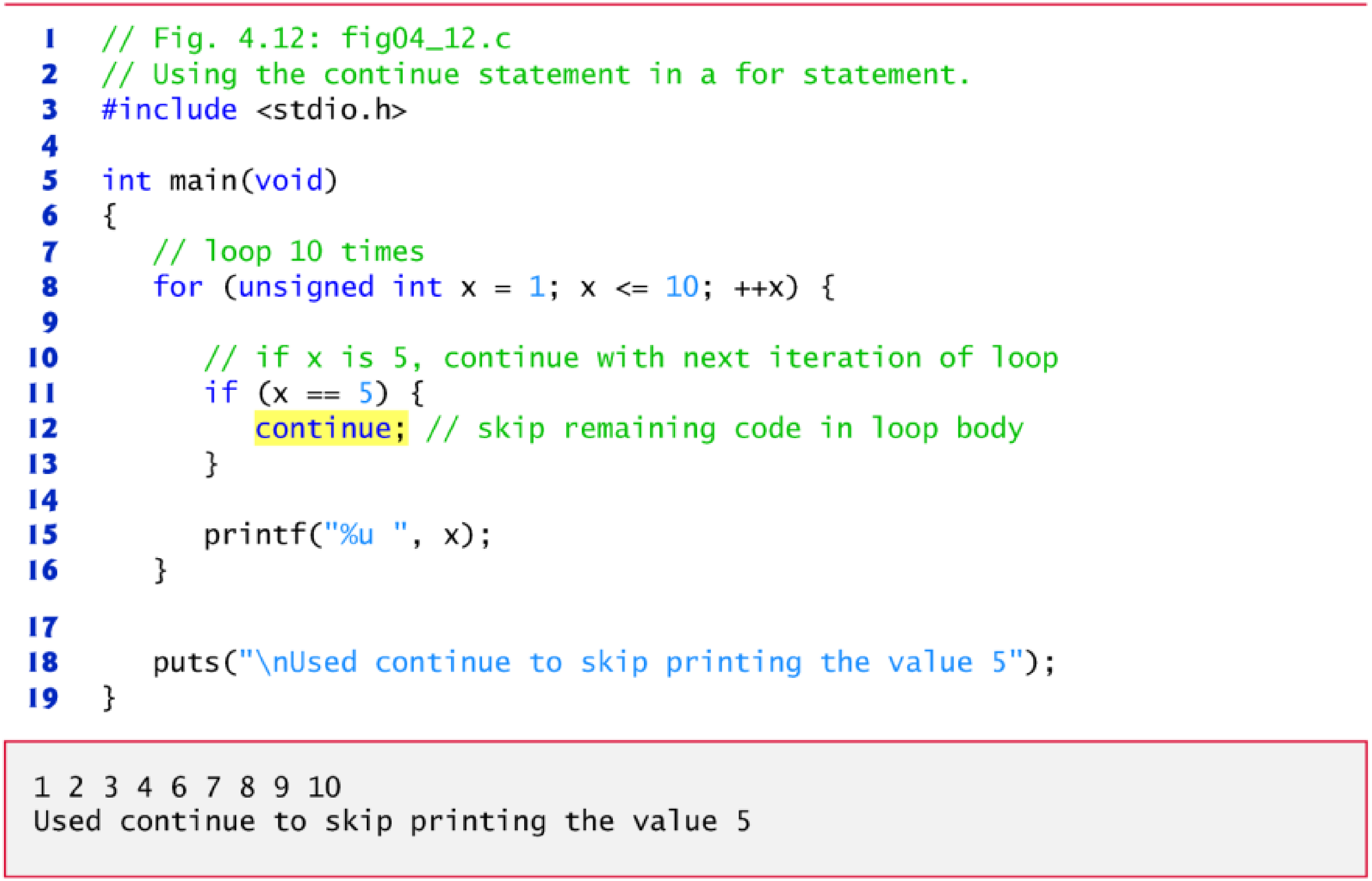
# *break* statement - Example



```c
// Fig. 4.11: fig04_11.c
// Using the break statement in a for statement.
#include <stdio.h>

int main(void)
{
   unsigned int x; // declared here so it can be used after loop

   // loop 10 times
   for (x = 1; x <= 10; ++x) {

      // if x is 5, terminate loop
      if (x == 5) {
         break; // break loop only if x is 5
      }

      printf("%u ", x);
   }

   printf("\nBroke out of loop at x == %u\n", x);
}
```

```
1 2 3 4
Broke out of loop at x == 5
```

# *continue* statement - Example



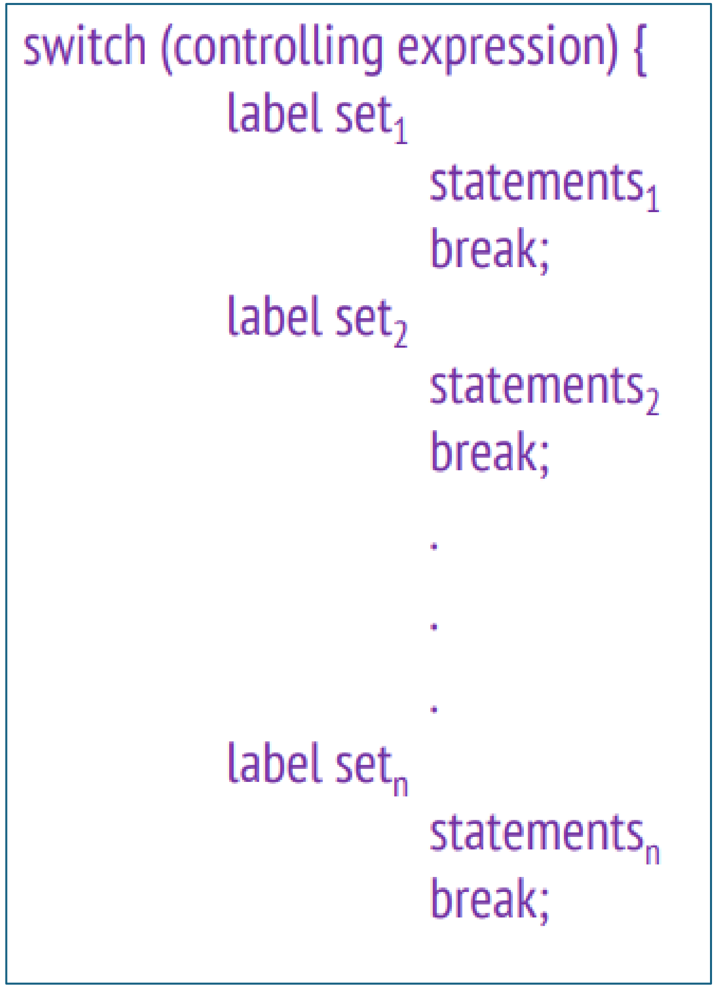
```c
// Fig. 4.12: fig04_12.c
// Using the continue statement in a for statement.
#include <stdio.h>

int main(void)
{
   // loop 10 times
   for (unsigned int x = 1; x <= 10; ++x) {

      // if x is 5, continue with next iteration of loop
      if (x == 5) {
         continue; // skip remaining code in loop body
      }

      printf("%u ", x);
   }

   puts("\nUsed continue to skip printing the value 5");
}
```

```
1 2 3 4 6 7 8 9 10
Used continue to skip printing the value 5
```

# *switch* statement

- In **Lecture 2**, we study *if* (single-selection statement) and *if … else* (double-selection statement)
- *switch*
  - used to select one of several alternatives
  - Useful when the selection is based on the value of a single variable or simple expression
  - Values may be of type int or char, but not double

**Syntax**



```
switch (controlling expression) {
        label set$_1$
                statements$_1$
                break;
        label set$_2$
                statements$_2$
                break;
                .
                .
                .
        label set$_n$
                statements$_n$
                break;
```

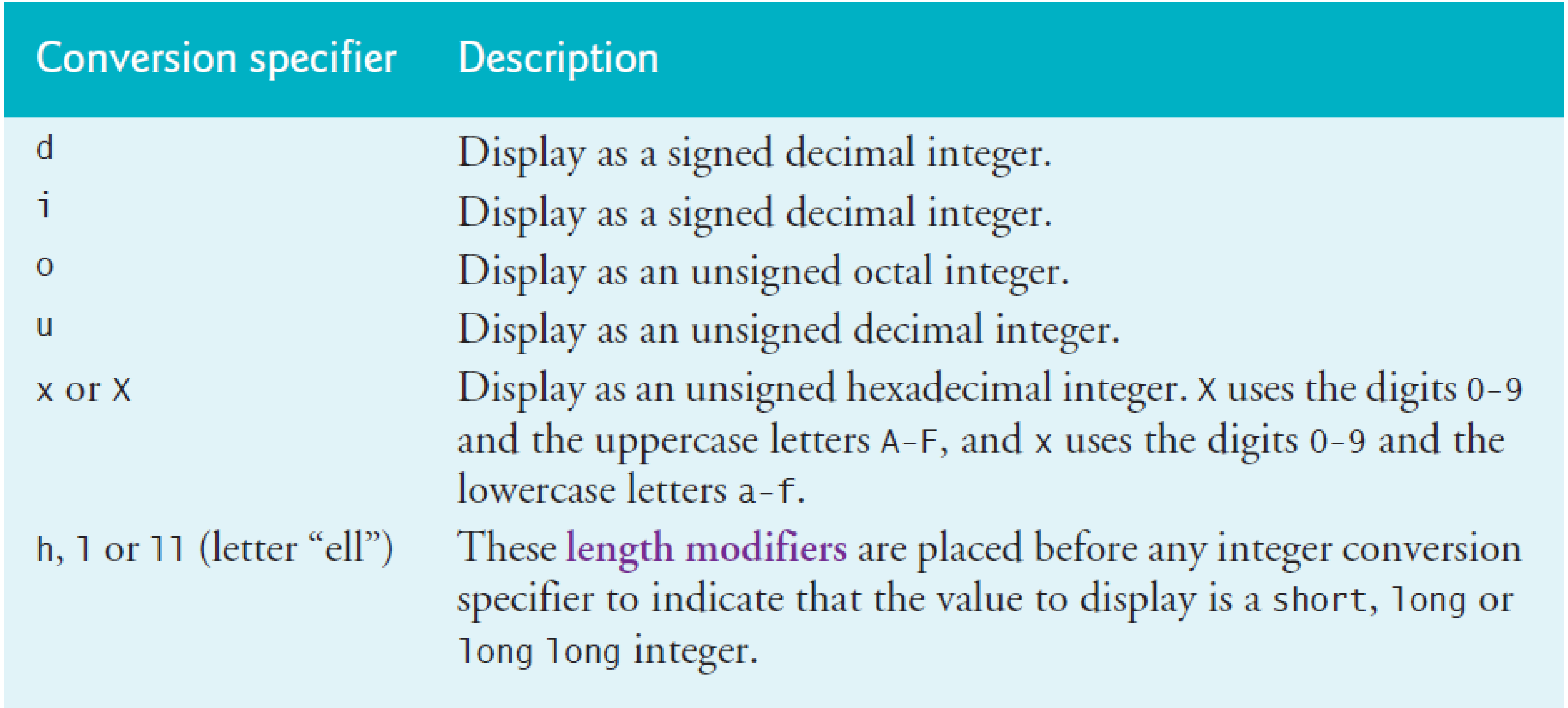
# $switch$ statement - Example

```
#include <stdio.h>

int main(void){
        int grade = 80;
        switch (grade)
        {
                case 90:
                    printf(“The grade is 90 points.\n”);
                    break;
                case 80:
                    printf(“The grade is 80 points.\n”);
                    break;
                default:
                    printf(“The grade is unknown.\n”);
                    break;
        }
}
```

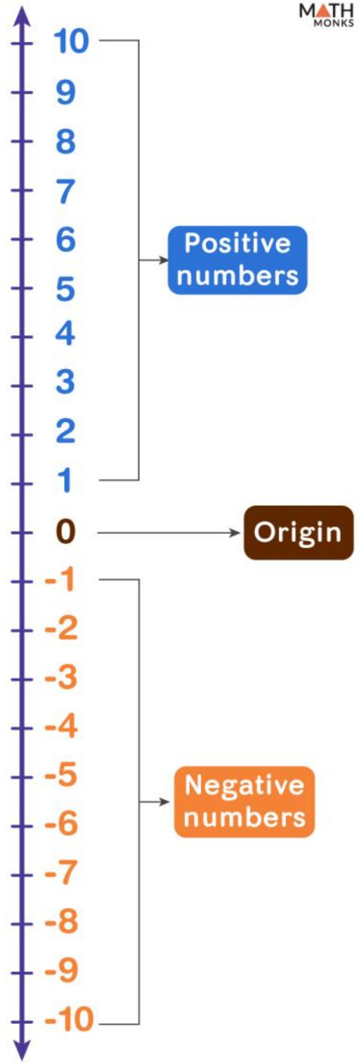
# Formated I/O

- This section discusses in-depth $printf$ and $scanf$ formatting features
  - Outputs data to standard output stream
  - Inputs data from standard input stream
  - Prints the field with different widths and precisions
  - Uses formatting flags
  - Includes header <stdio.h> and several additional function in <stdio.h>

# $printf$ formatting features – Integer (Chap 9.4)



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



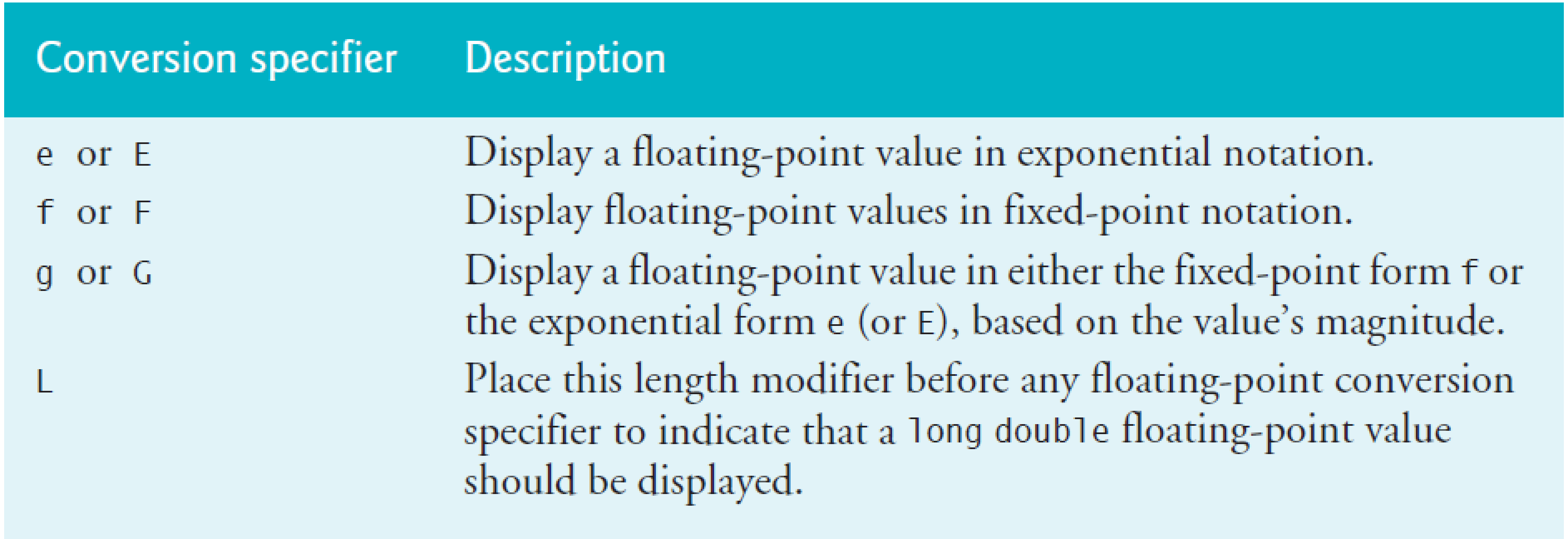
# $printf$ formatting features – Integer (Chap 9.4)

```c
// fig09_01.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);         // print as type short
    printf("%ld\n", 2000000000L);   // print as type long
    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);            // hexadecimal with uppercase letters
}
```

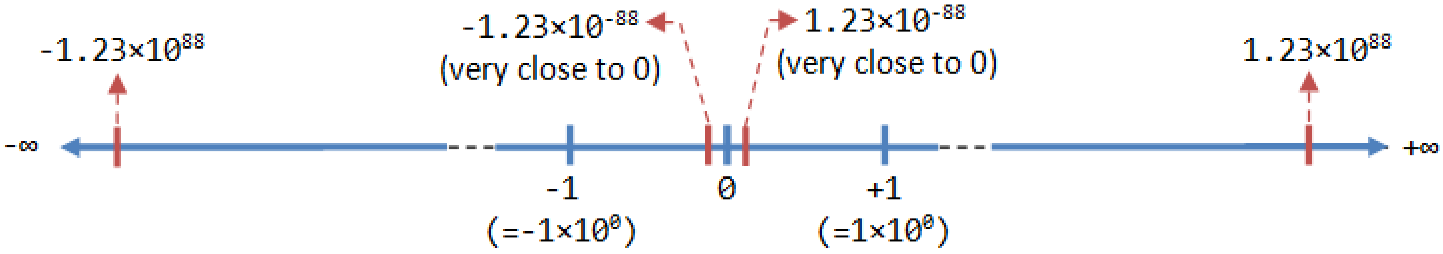
**OUTPUT**

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

# $printf$ formatting features – Floating (Chap 9.5)



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



Floating-point Numbers (Decimal)

# $printf$ formatting features – Floating (Chap 9.5)

- Remember that all floating-point conversion specifiers have a default precision of 6

```c
// fig09_02.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); // six digits to right of decimal point
printf("%g\n", 1234.56); // prints with lowercase e
printf("%G\n", 123456789); // prints with uppercase E
}
```

**OUTPUT**

```
1.234568e+06
1.234568e+06
-1.234568e+06
1.234568E+06
1234567.890000
1234.56
1.23457E+06
```

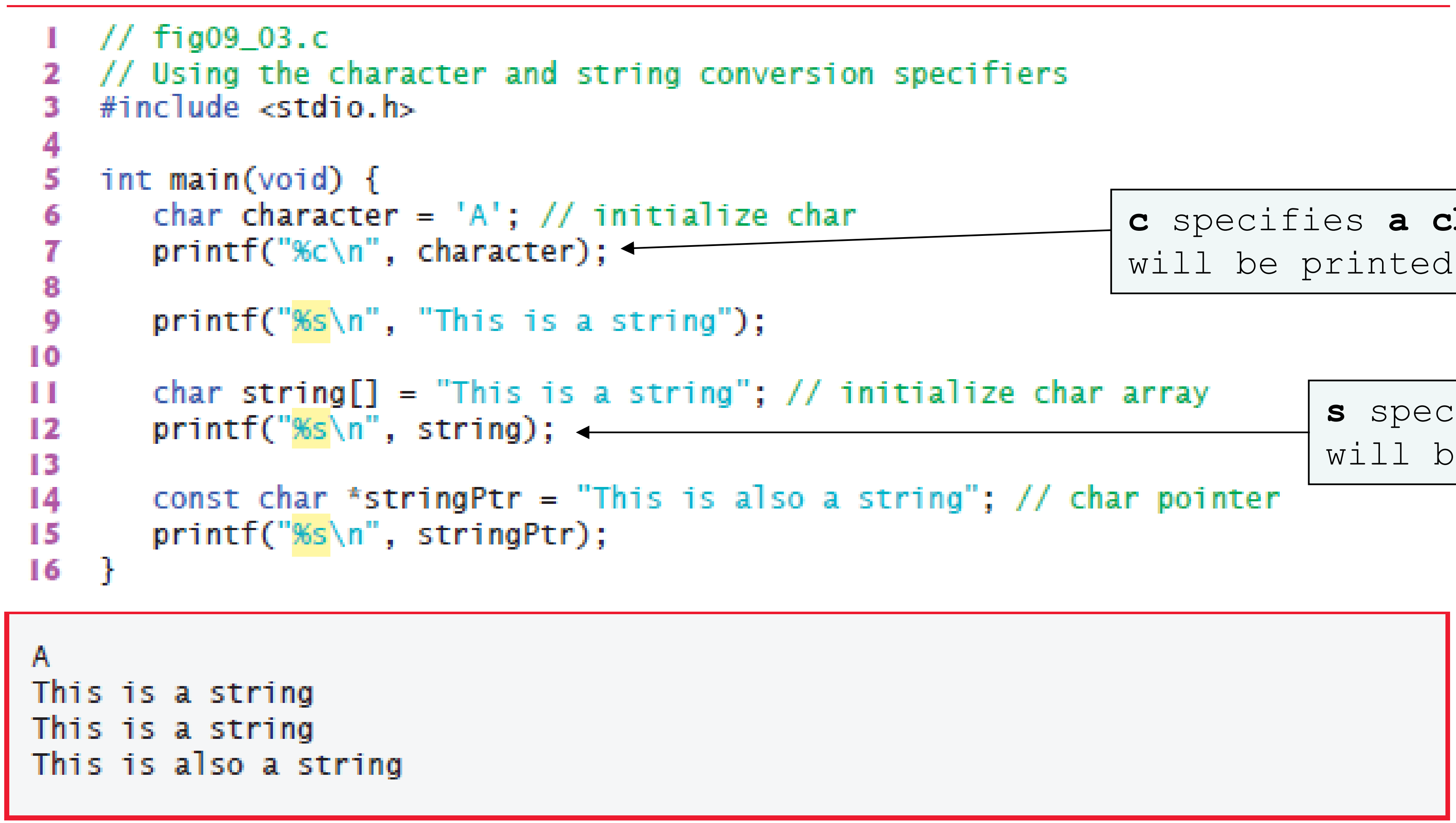
# $printf$ – Strings and Characters (Chap 9.6)

- **Conversion specifier** c
  - Prints char argument
  - Cannot be used to print the first character of a string
- **Conversion specifier** s
  - Requires a pointer to char as an argument
  - Prints characters until NULL ('\0') encountered
  - Cannot print a char argument
- Remember
  - Single quotes for character constants ('z')
  - Double quotes for strings "z" (which actually contains two characters, 'z' and '\0')

# $printf$ – Strings and Characters (Chap 9.6)

- Common program error
  - 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 *).
  - Using %s to print a char argument, on some systems, causes a fatal execution-time error called an access violation. The conversion specifier %s expects an argument of type pointer to char.

# *printf* – Strings and Characters (Chap 9.6)



```c
// fig09_03.c
// Using the character and string conversion specifiers
#include <stdio.h>

int main(void) {
   char character = 'A'; // initialize char
   printf("%c\n", character);

   printf("%s\n", "This is a string");

   char string[] = "This is a string"; // initialize char array
   printf("%s\n", string);

   const char *stringPtr = "This is also a string"; // char pointer
   printf("%s\n", stringPtr);
}
```

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

**c** specifies **a character** will be printed

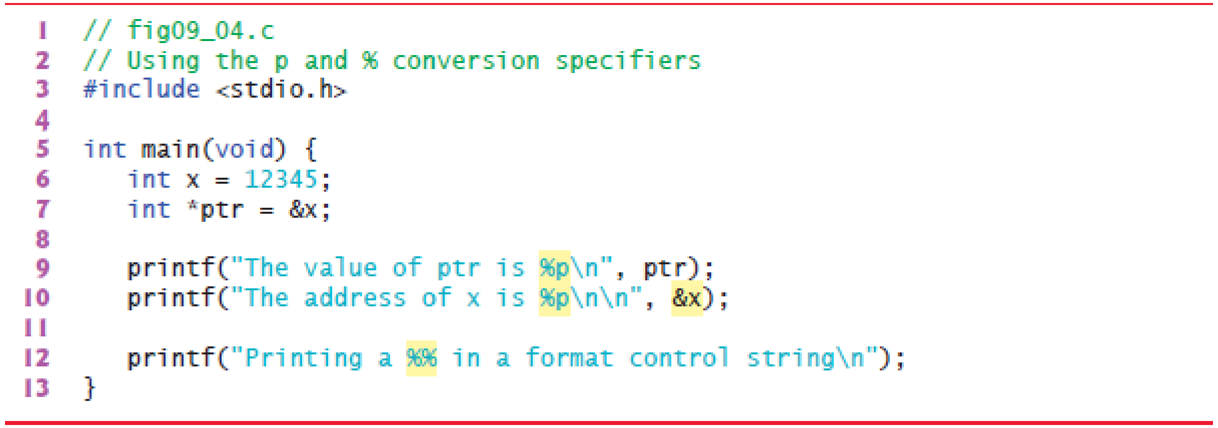
**s** specifies **a string** will be printed

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

# $printf$ – Other Conversion Specifiers (Chap 9.7)

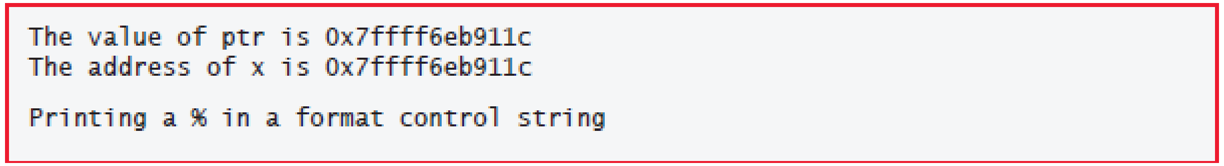
## Consider the p and % conversion specifiers:

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



```c
// fig09_04.c
// Using the p and % conversion specifiers
#include <stdio.h>

int main(void) {
   int x = 12345;
   int *ptr = &x;

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

   printf("Printing a %% in a format control string\n");
}
```

**Fig. 9.4** | Using the p and % conversion specifiers. (Part 1 of 2.)



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

Printing a % in a format control string
```

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

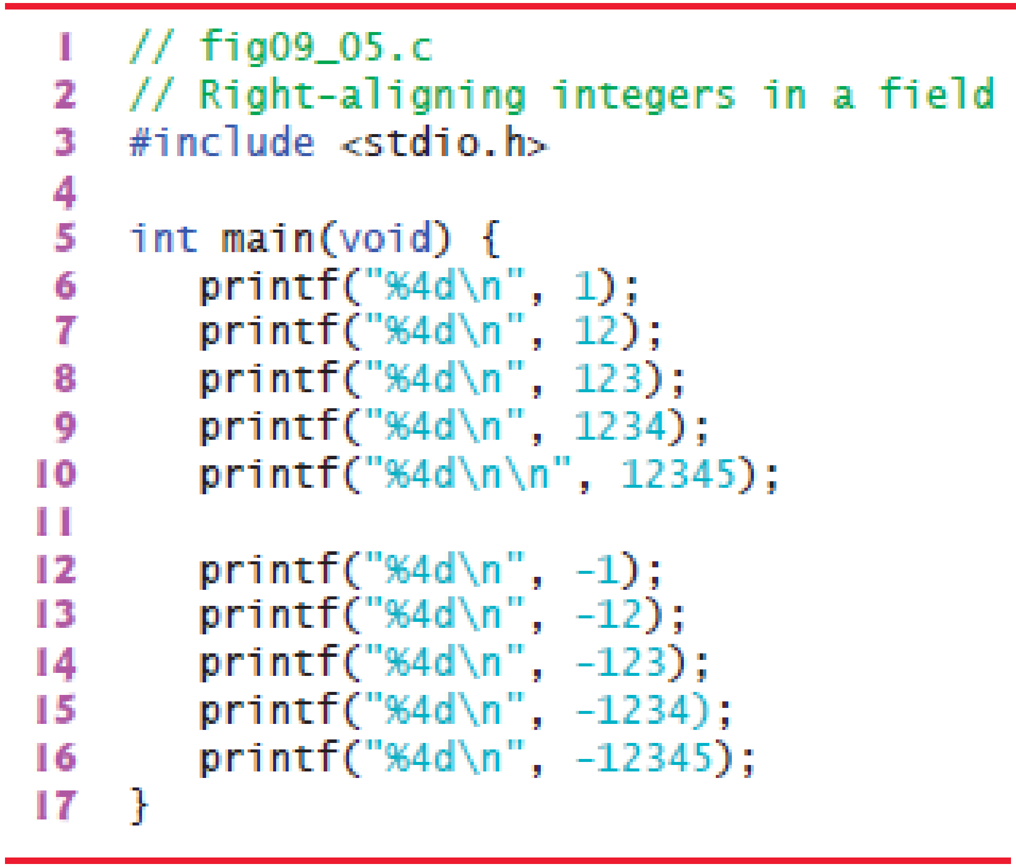
# $printf$ – Printing with Field Widths and Precision (Chap 9.8)

- Field width
  - exact size of field in which data is printed
  - If field width is larger than data being printed

  → data will be right-aligned with that field
  - Value wider than the field, → display in full
  - Minus sign for negative value take 1 position in field width
  - Integer width inserted between % and conversion specifier

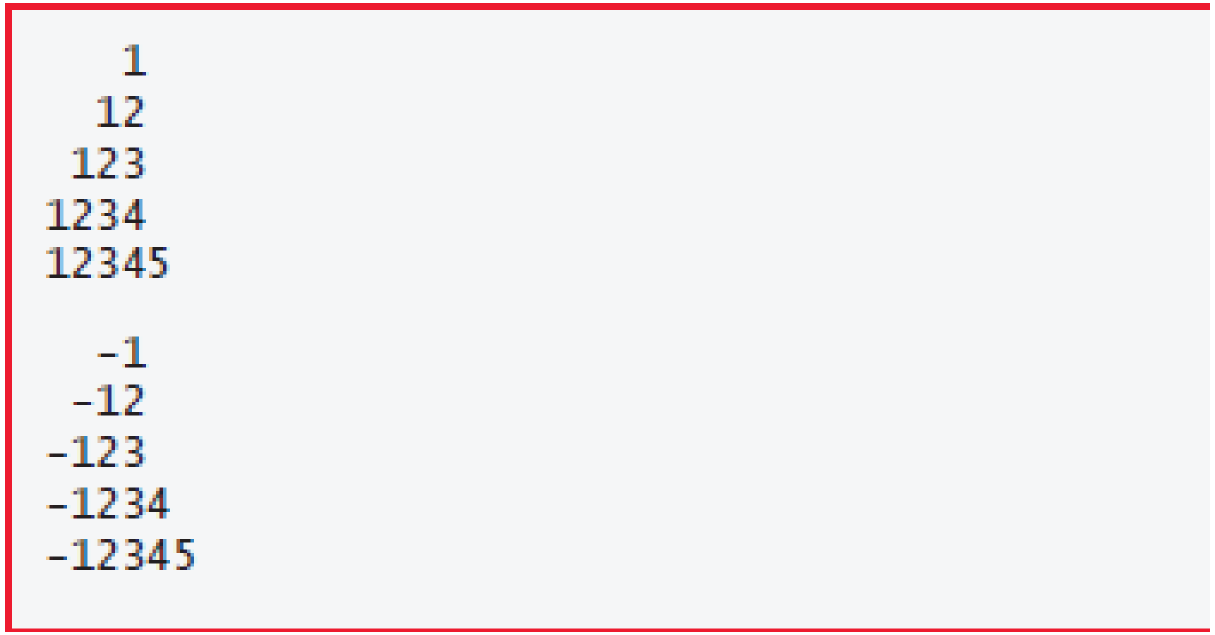
Format: %field_widthconversion_specifier

# $printf$ – Printing with Field Widths and Precision (Chap 9.8)



```c
// fig09_05.c
// Right-aligning 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);
}
```

**Fig. 9.5** | Right-aligning integers in a field. (Part 1 o



```
   1
  12
 123
1234
12345

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

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

# $printf$ – Printing with Field Widths and Precision (Chap 9.8)

- Precisions
  - Specify the precision with which data is printed
  - **Integer conversion specifiers**

    → minimum number of digits to be printed

    → if printed int is small, prefixed with zeros

    → default precision for integers is 1
  - **Floating-point conversion specifier**

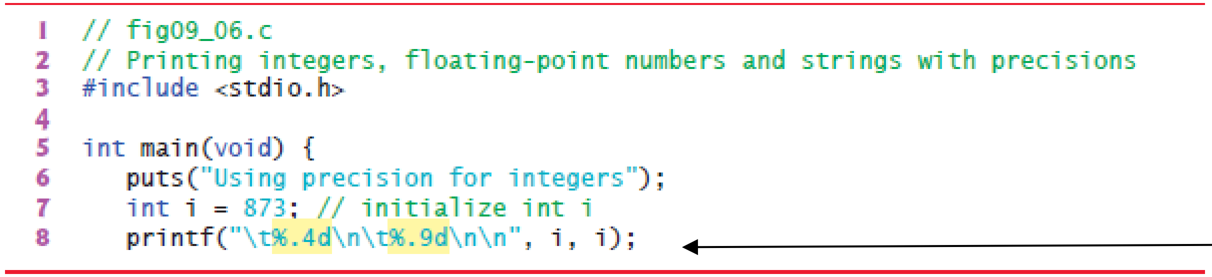
    → e, E, and f: number of digits after decimal point

    → g, G: maximum number of significant digits to be printed
  - **String conversion specifier**

    → s: maximum number of characters written from the the beginning of the strings
- Format

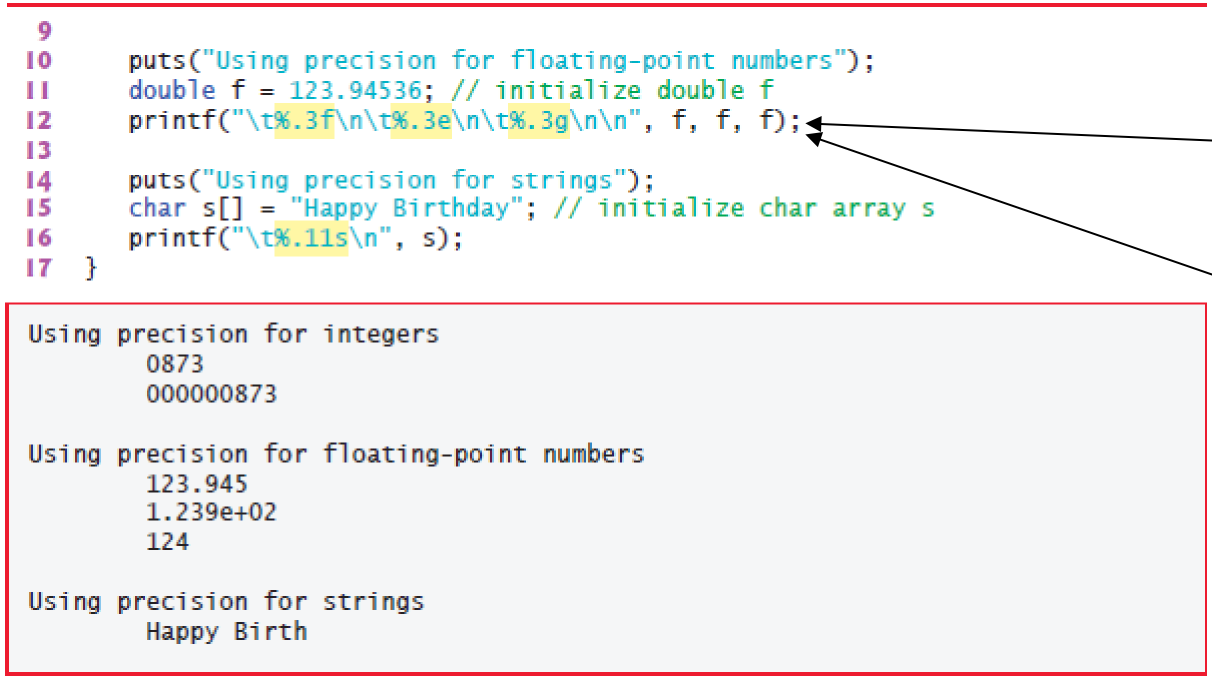
→ %.precisionconversion_specifier

# $printf$ – Printing with Field Widths and Precision (Chap 9.8)



```c
// fig09_06.c
// Printing integers, floating-point numbers and strings with precisions
#include <stdio.h>

int main(void) {
   puts("Using precision for integers");
   int i = 873; // initialize int i
   printf("\t%.4d\n\t%.9d\n\n", i, i);
```

Fig. 9.6 | Printing integers, floating-point numbers and strings with precisions. (Part 1 of 2.)



```c

   puts("Using precision for floating-point numbers");
   double f = 123.94536; // initialize double f
   printf("\t%.3f\n\t%.3e\n\t%.3g\n\n", f, f, f);

   puts("Using precision for strings");
   char s[] = "Happy Birthday"; // initialize char array s
   printf("\t%.11s\n", s);
}
```

```
Using precision for integers
        0873
        000000873

Using precision for floating-point numbers
        123.945
        1.239e+02
        124

Using precision for strings
        Happy Birth
```

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

Precision for integers specifies **the minimum number of characters to be printed**

Precision for **f** and **e** specifiers **controls the number of digits after the decimal point**

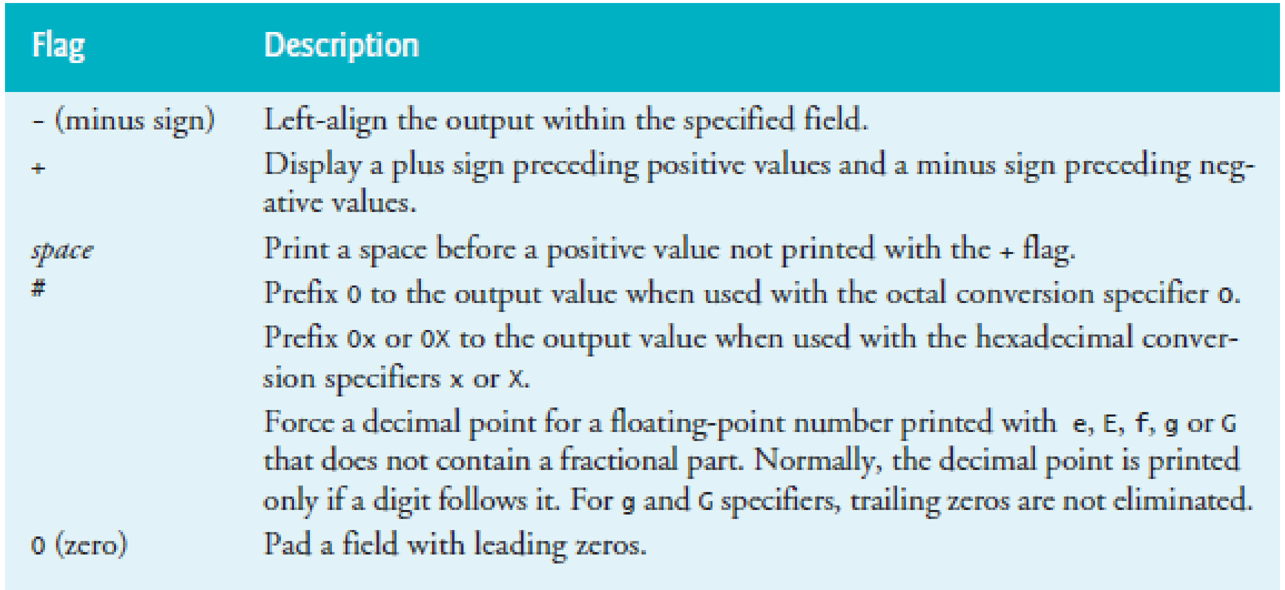
Precision for the **g** specifier **controls the maximum number of significant digits printed**

# $printf$ – Printing with Field Widths and Precision (Chap 9.8)

- Combining field widths and precision

→ printf("%9.3f", 123.456789);

→ 123.457



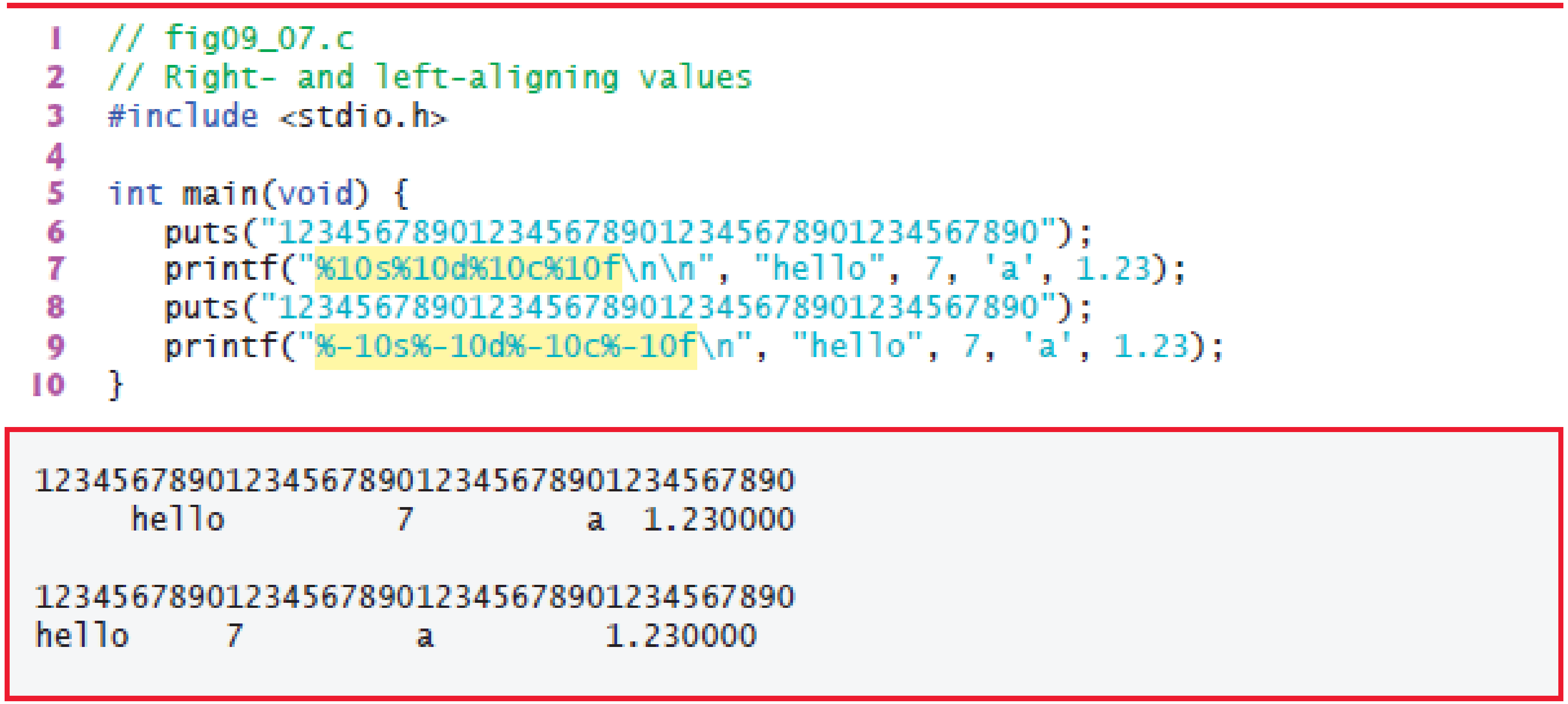
| Flag | Description |
|---|---|
| - (minus sign) | Left-align the output within the specified field. |
| + | Display a plus sign preceding positive values and a minus sign preceding negative values. |
| *space* | Print a space before a positive value not printed with the + flag. |
| # | Prefix 0 to the output value when used with the octal conversion specifier o.<br>Prefix 0x or 0X to the output value when used with the hexadecimal conversion specifiers x or X.<br>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. |

# $printf$ – Format Flags

Supplement its output formatting capabilities

# $printf$ – Format Flags

## Chap 9.9.1 Right- and Left-Alignment



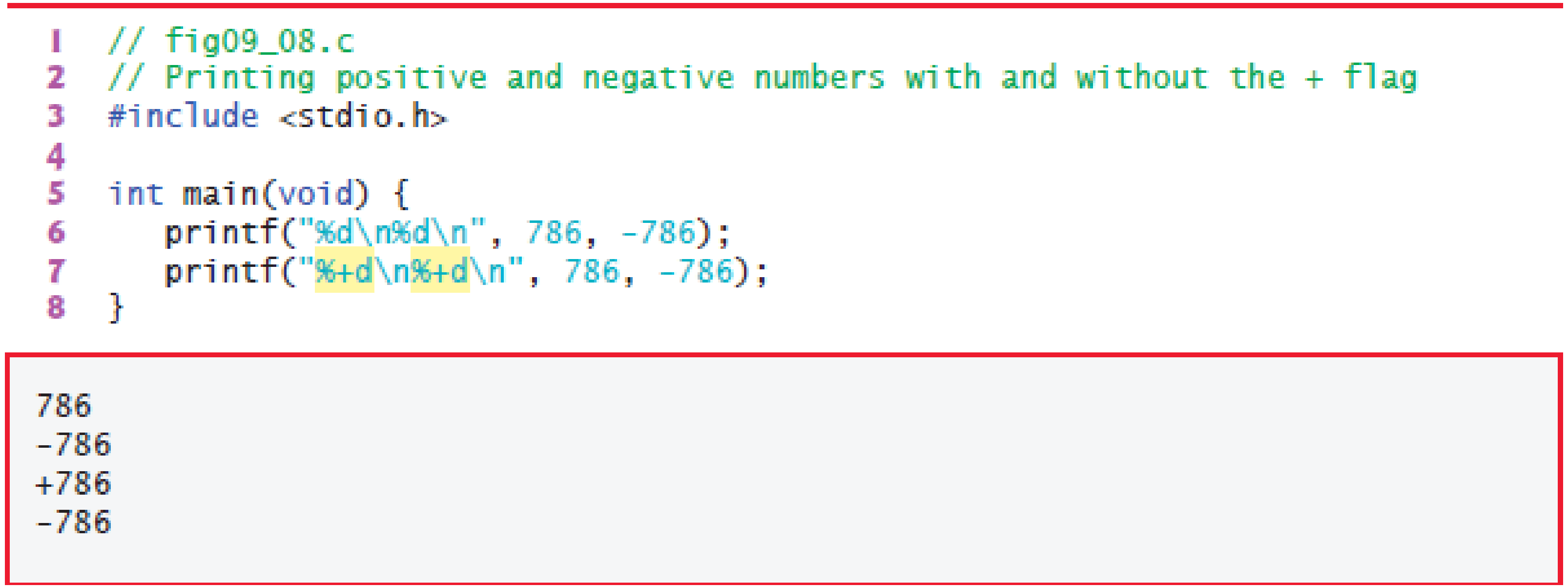
```
// fig09_07.c
// Right- and left-aligning values
#include <stdio.h>

int main(void) {
   puts("1234567890123456789012345678901234567890");
   printf("%10s%10d%10c%10f\n\n", "hello", 7, 'a', 1.23);
   puts("1234567890123456789012345678901234567890");
   printf("%-10s%-10d%-10c%-10f\n", "hello", 7, 'a', 1.23);
}
```

```
1234567890123456789012345678901234567890
     hello         7         a  1.230000

1234567890123456789012345678901234567890
hello     7         a         1.230000
```

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

# *printf* – Format Flags

## Chap 9.9.1 Print Positive and Negative Numbers with and without the + Flag



```c
// fig09_08.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);
}
```

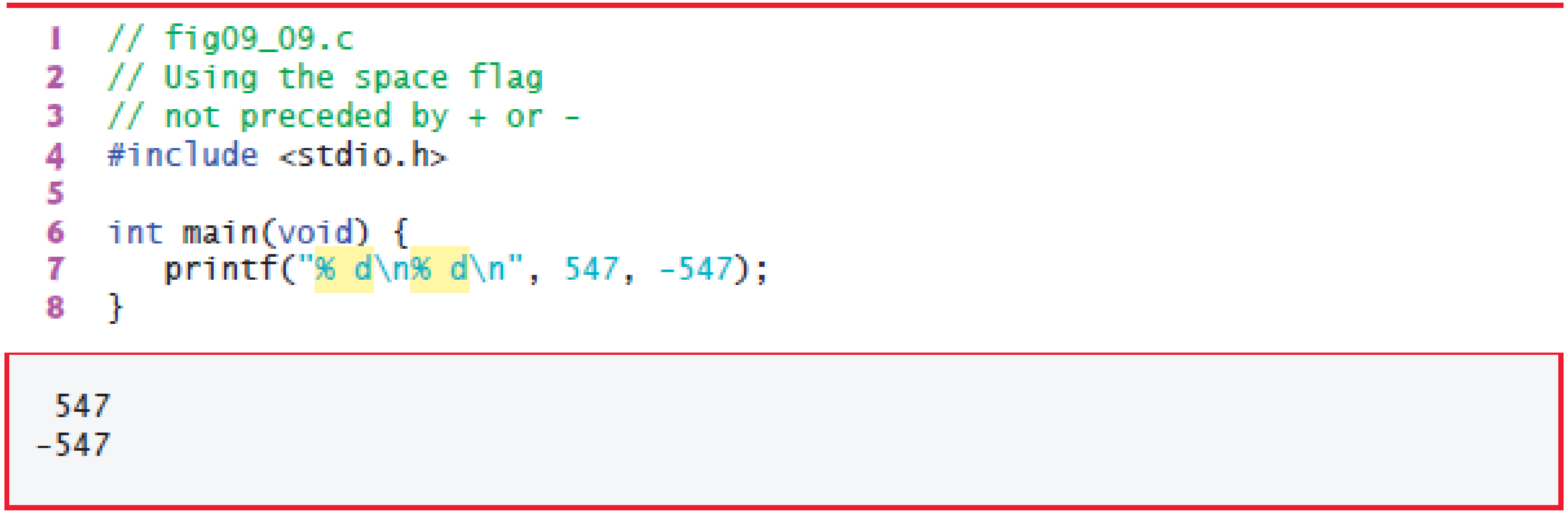
```
786
-786
+786
-786
```

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

# $printf$ – Format Flags

## Chap 9.9.3 Using the Space Flag

→ Useful for aligning positive and negative number



```c
// fig09_09.c
// Using the space flag
// not preceded by + or -
#include <stdio.h>

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

```
 547
-547
```

Fig. 9.9 | Using the space flag.

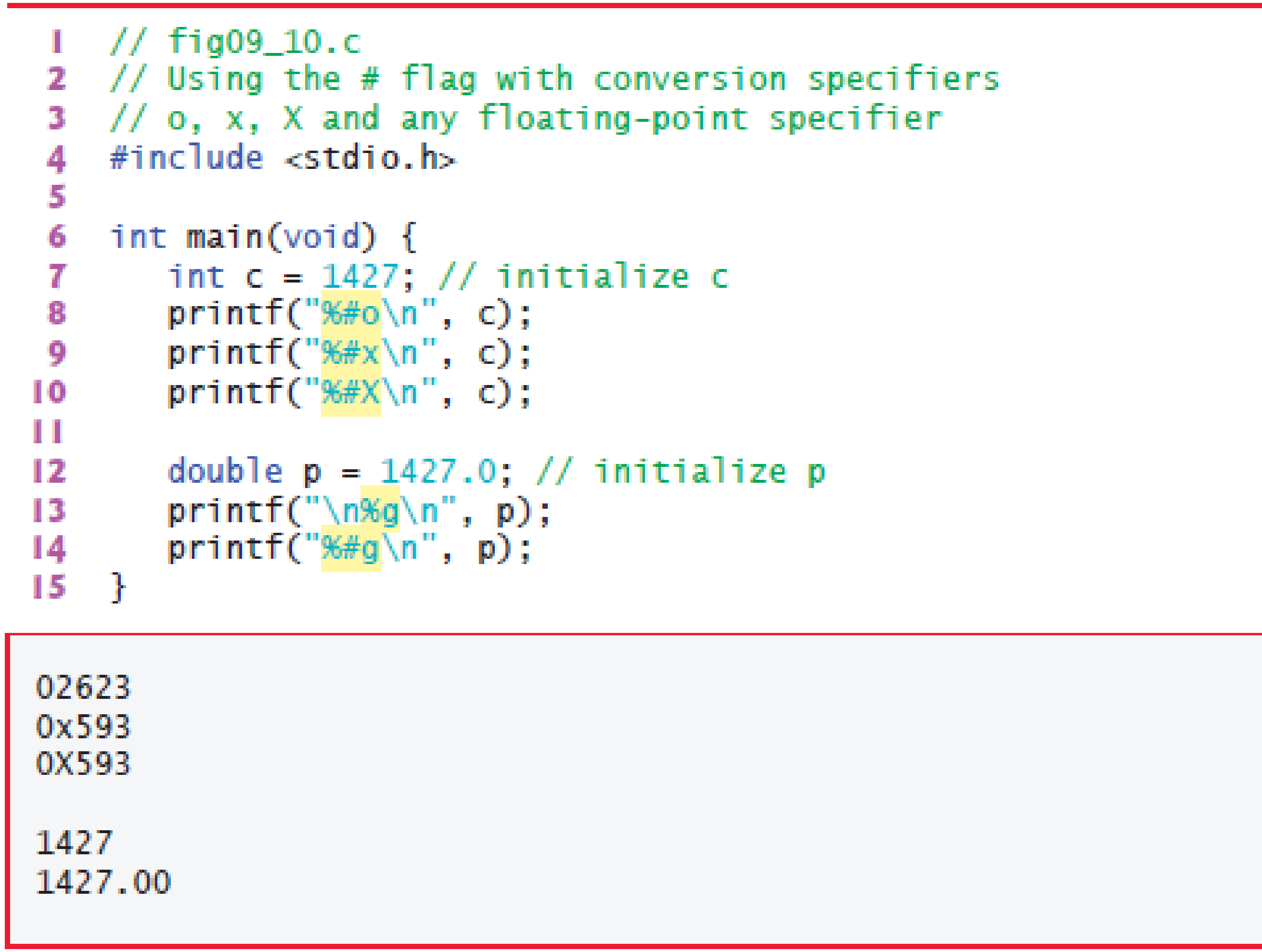
# $printf$ – Format Flags

- Chap 9.9.4 Using the # Flag

  → prefix 0 to octal value

  → prefix 0x or 0X to hexadecimal values

  → with g, force the decimal point to print



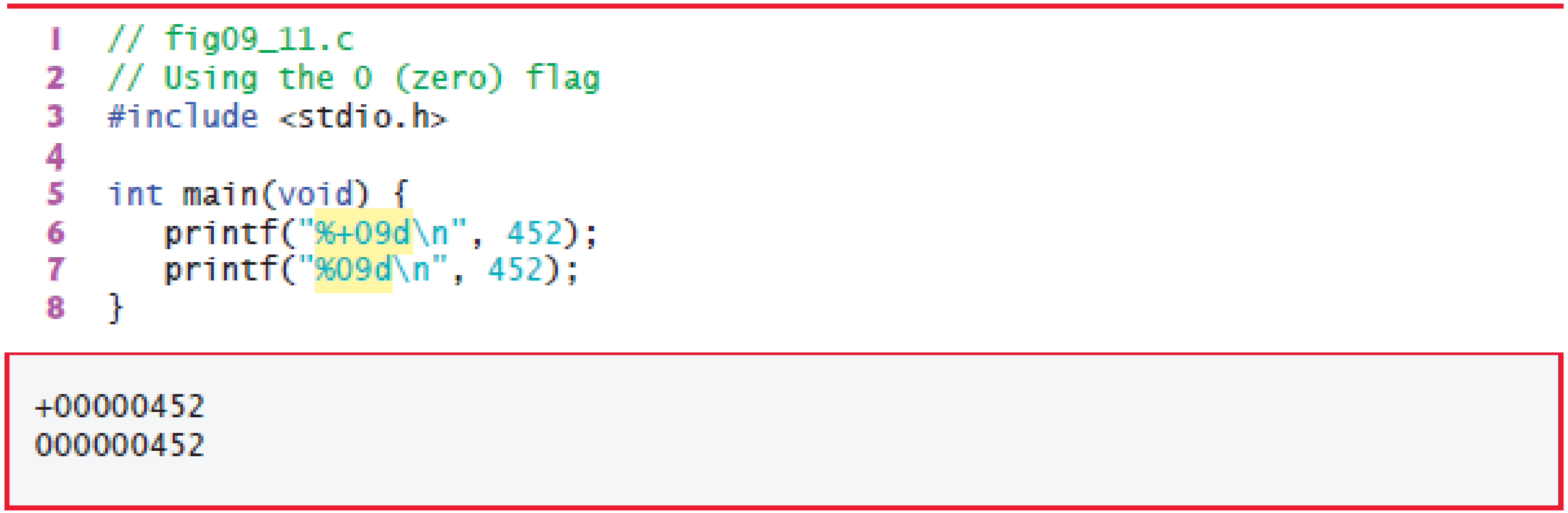
```c
// fig09_10.c
// Using the # flag with conversion specifiers
// o, x, X and any floating-point specifier
#include <stdio.h>

int main(void) {
   int c = 1427; // initialize c
   printf("%#o\n", c);
   printf("%#x\n", c);
   printf("%#X\n", c);

   double p = 1427.0; // initialize p
   printf("\n%g\n", p);
   printf("%#g\n", p);
}
```

```
02623
0x593
0X593

1427
1427.00
```

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

# *printf* – Format Flags

## Chap 9.9.5 Using the 0 Flag

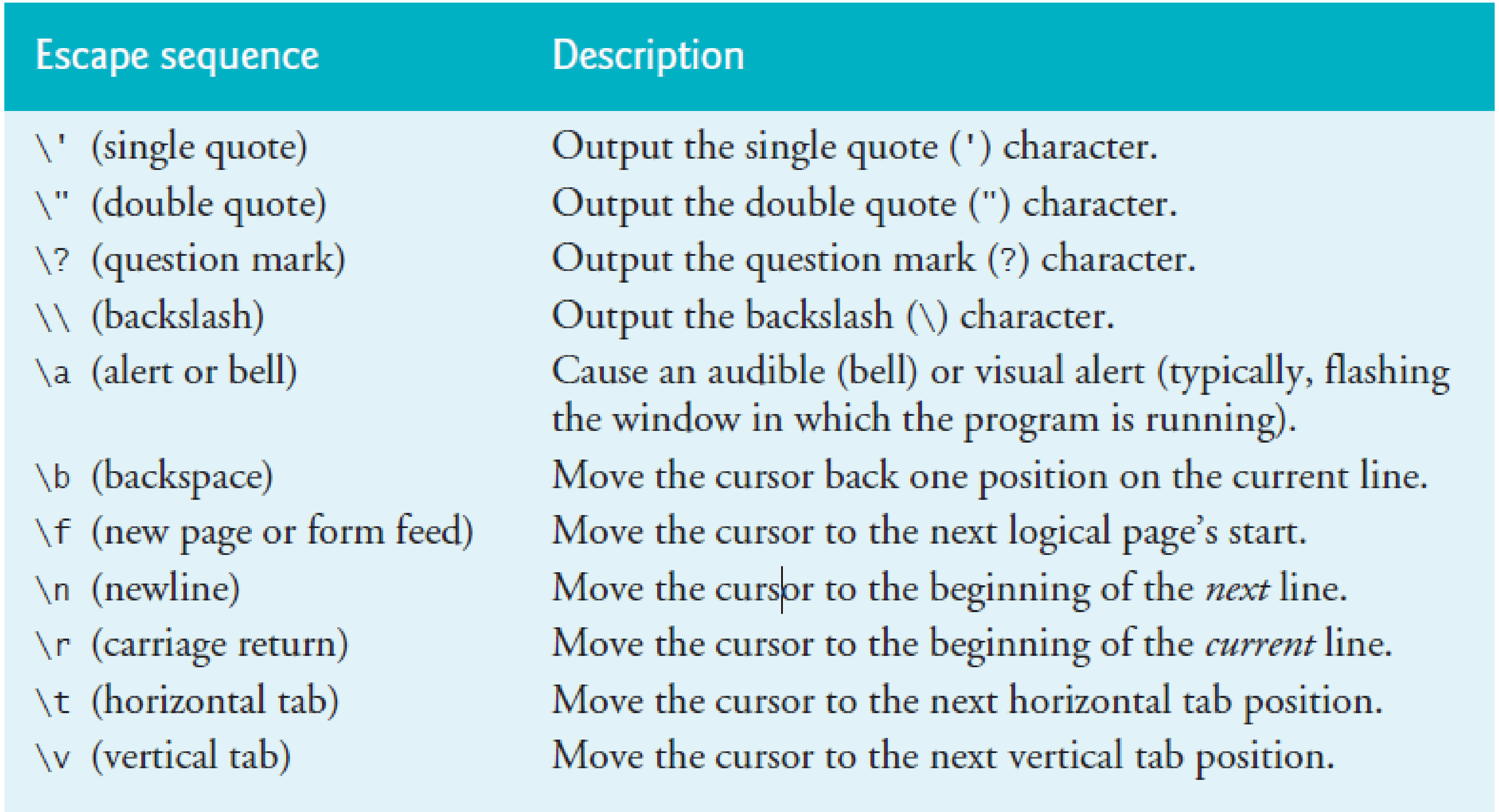


```c
// fig09_11.c
// Using the 0 (zero) flag
#include <stdio.h>

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

```
+00000452
000000452
```

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

# $printf$ – Literals and Escape Sequence

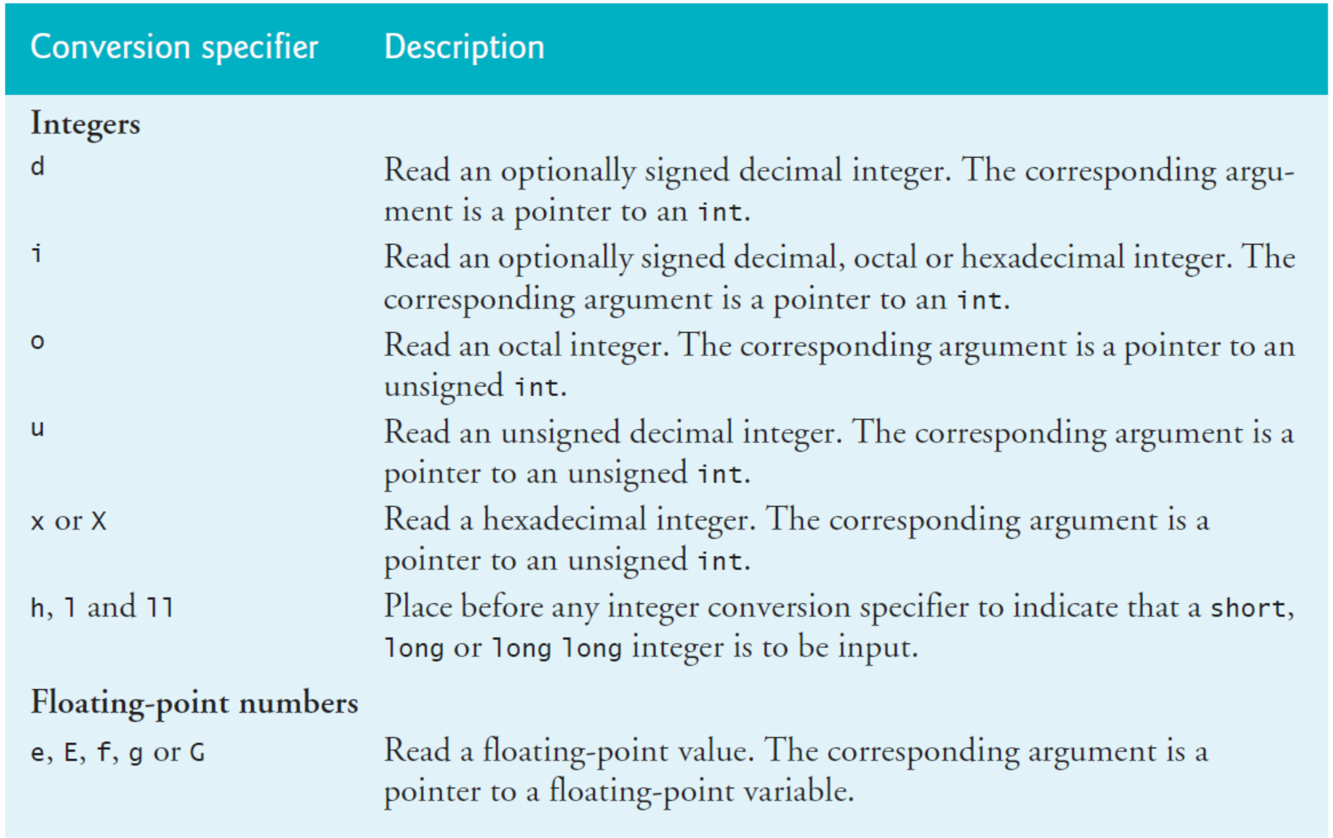


| Escape sequence | Description |
|---|---|
| \' (single quote) | Output the single quote (') character. |
| \" (double quote) | Output the double quote (") character. |
| \? (question mark) | Output the question mark (?) character. |
| \\ (backslash) | Output the backslash (\) character. |
| \a (alert or bell) | Cause an audible (bell) or visual alert (typically, flashing the window in which the program is running). |
| \b (backspace) | Move the cursor back one position on the current line. |
| \f (new page or form feed) | Move the cursor to the next logical page's start. |
| \n (newline) | Move the cursor to the beginning of the *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. |

# $scanf$ conversion specifiers

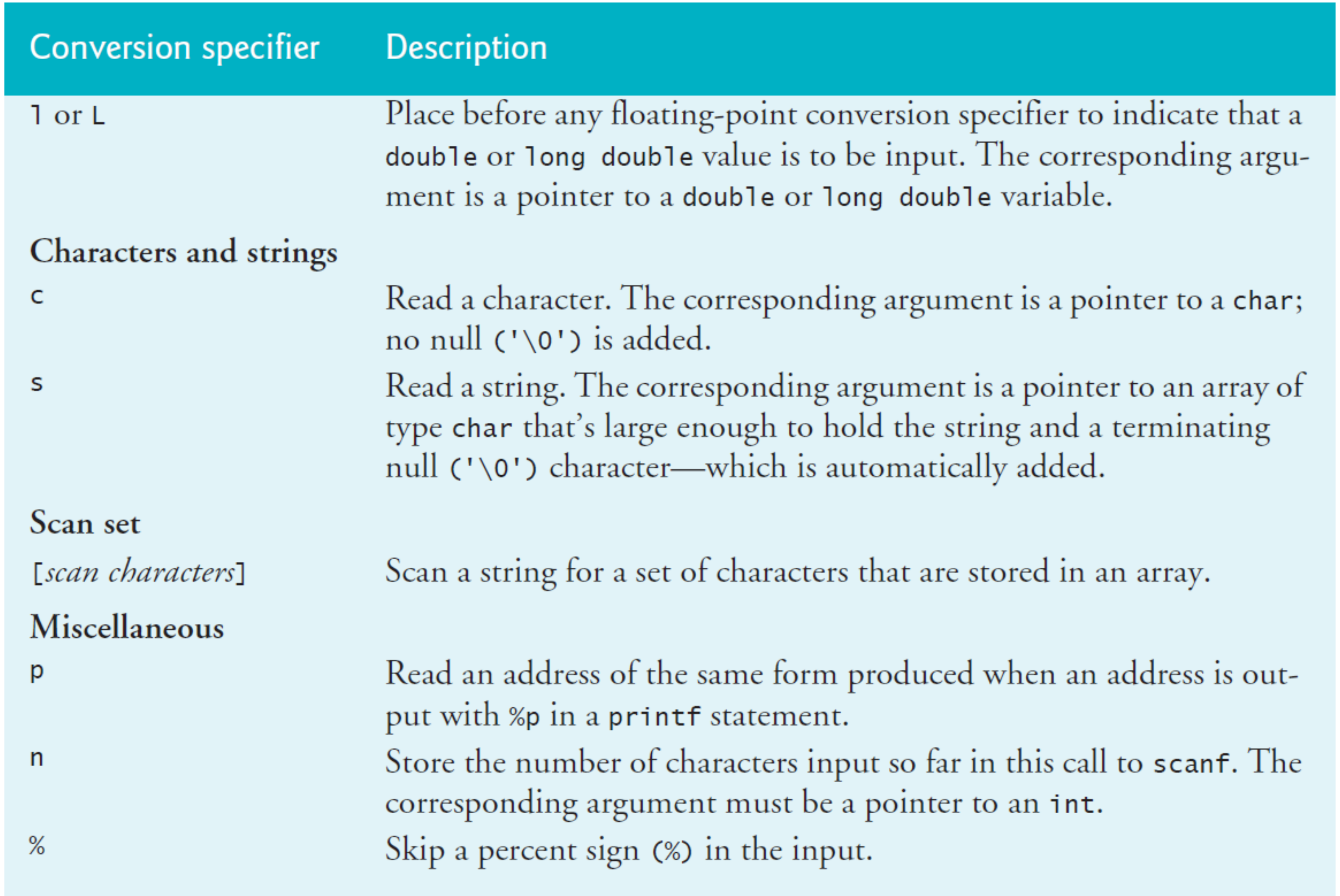
- Prompt the user to input one data item for few data items at a time
- Function scanf is written in the follwwing form:

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



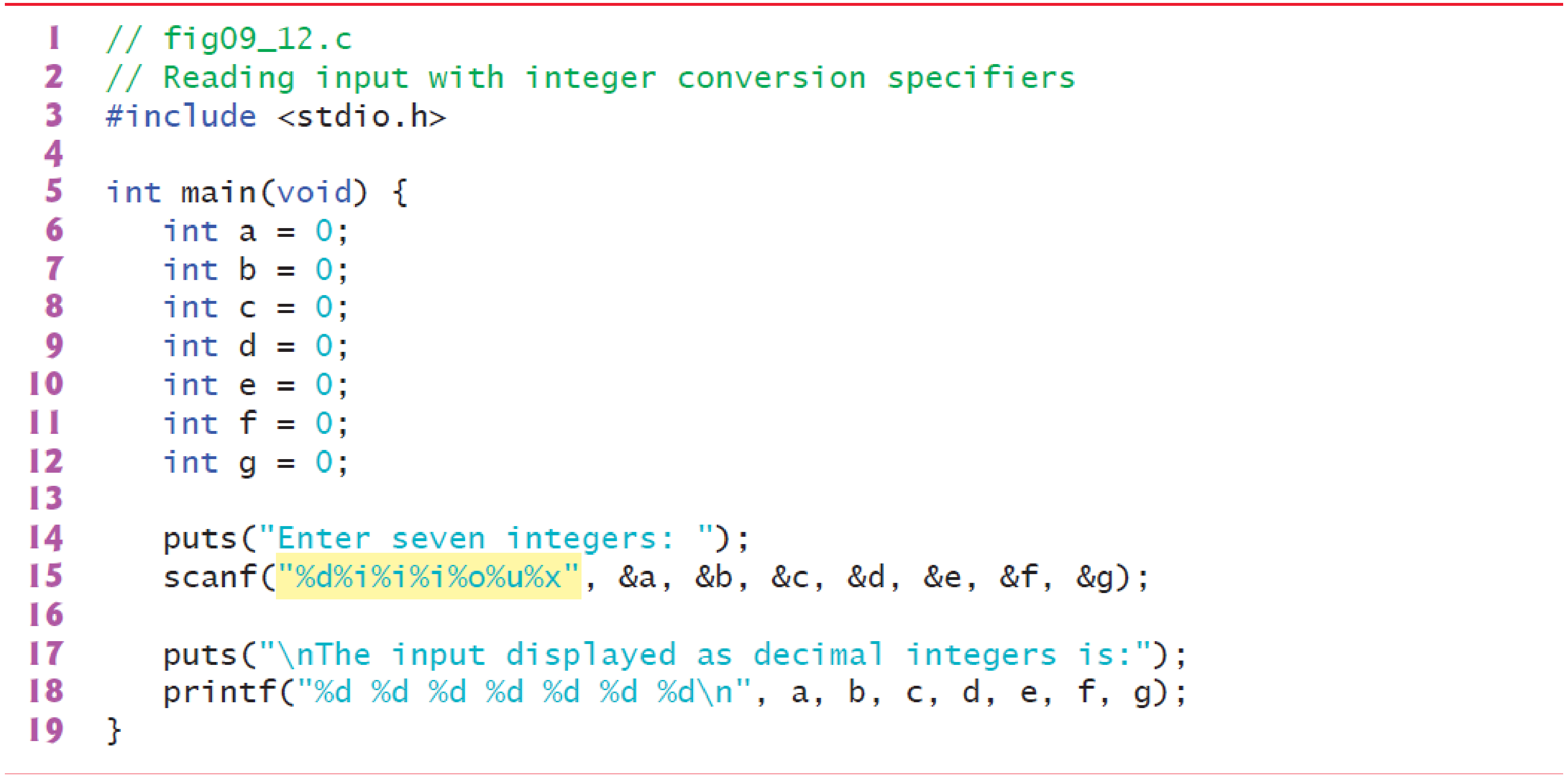
| Conversion specifier | Description |
|---|---|
| **Integers** | |
| `d` | Read an optionally signed decimal integer. The corresponding argument is a pointer to an `int`. |
| `i` | Read an optionally signed decimal, octal or hexadecimal integer. The corresponding argument is a pointer to an `int`. |
| `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 integer conversion specifier to indicate that a `short`, `long` or `long long` integer is to be input. |
| **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. |

# $scanf$ conversion specifiers

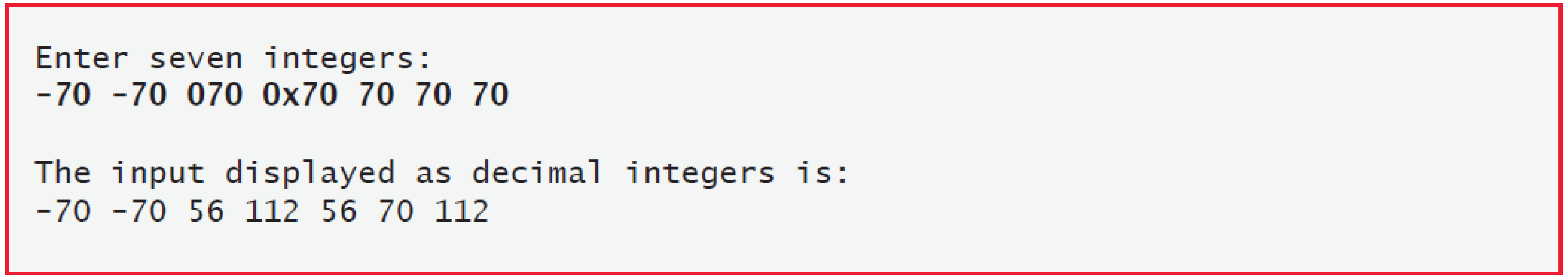


| Conversion specifier | Description |
|---|---|
| `l` or `L` | Place before any floating-point conversion specifier to indicate that a `double` or `long double` value is to be input. The corresponding argument is a pointer to a `double` or `long double` variable. |
| **Characters and strings** | |
| `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. |
| **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 must be a pointer to an `int`. |
| `%` | Skip a percent sign (`%`) in the input. |

# *scanf* – Reading Integer Input

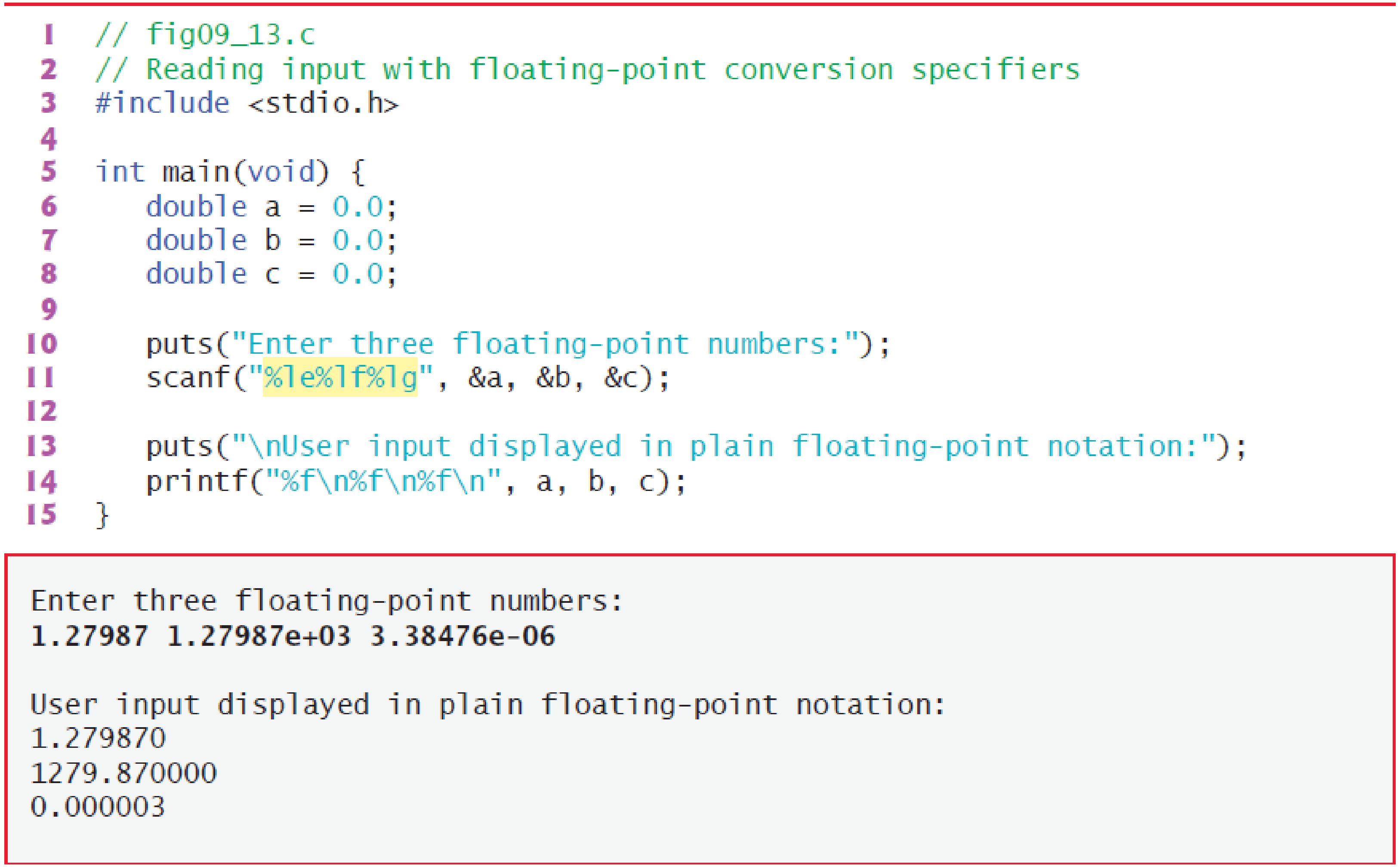


```c
// fig09_12.c
// Reading input with integer conversion specifiers
#include <stdio.h>

int main(void) {
   int a = 0;
   int b = 0;
   int c = 0;
   int d = 0;
   int e = 0;
   int f = 0;
   int g = 0;

   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);
}
```



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

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

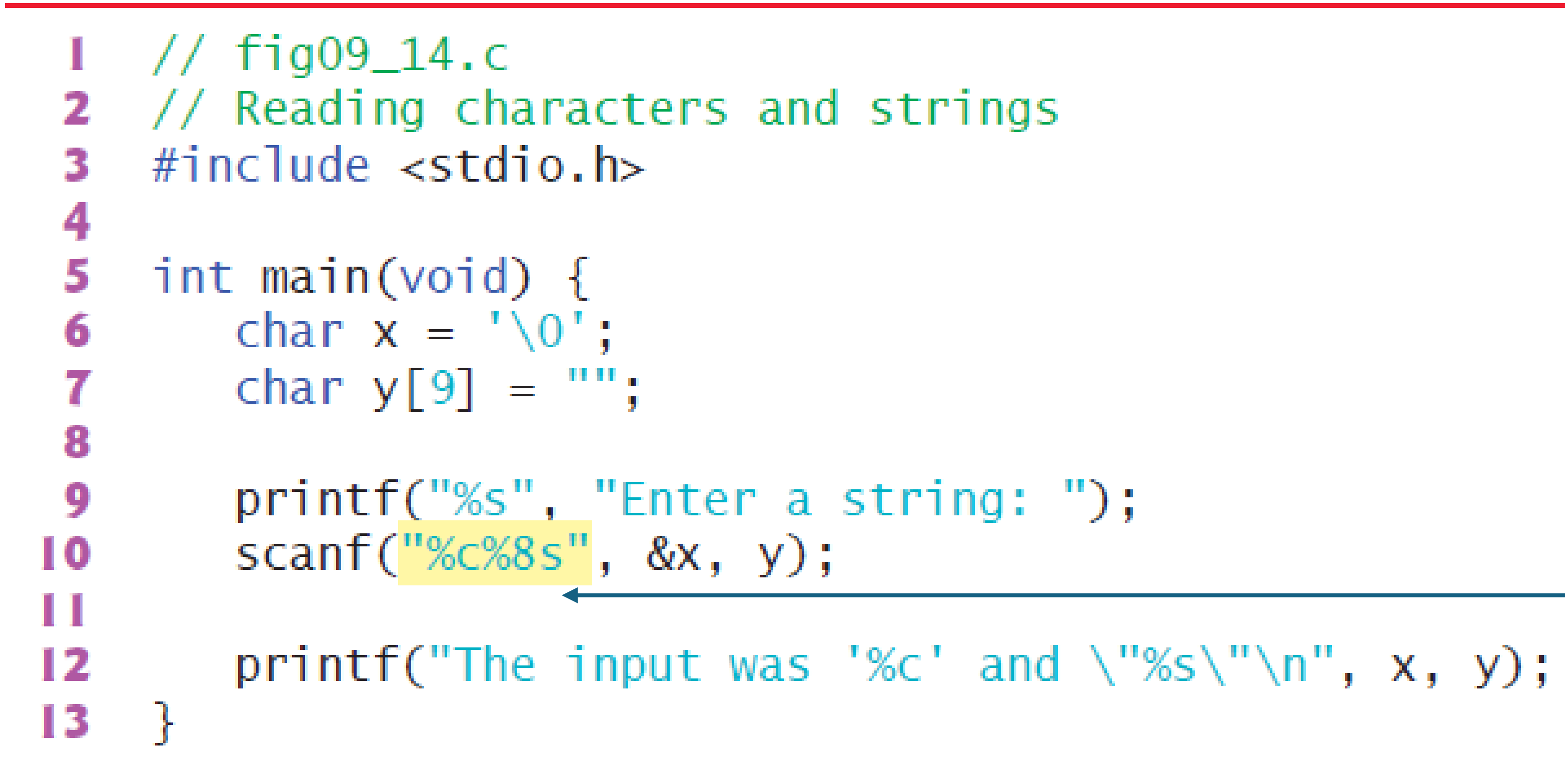
# $scanf$ – Floating Point Input



```c
// fig09_13.c
// Reading input with floating-point conversion specifiers
#include <stdio.h>

int main(void) {
   double a = 0.0;
   double b = 0.0;
   double c = 0.0;

   puts("Enter three floating-point numbers:");
   scanf("%le%lf%lg", &a, &b, &c);

   puts("\nUser input displayed in plain floating-point notation:");
   printf("%f\n%f\n%f\n", a, b, c);
}
```

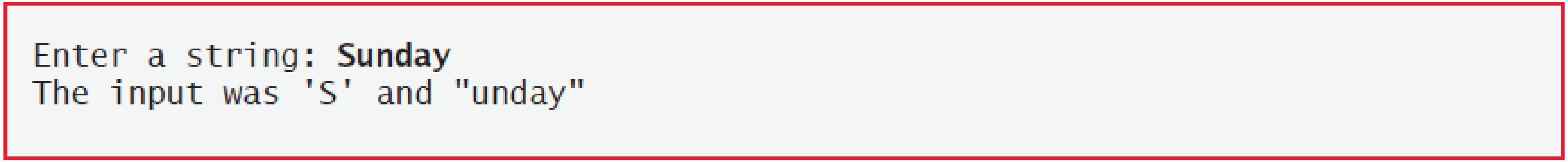
```
Enter three floating-point numbers:
1.27987 1.27987e+03 3.38476e-06

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

# $scanf$ – Characters and Strings



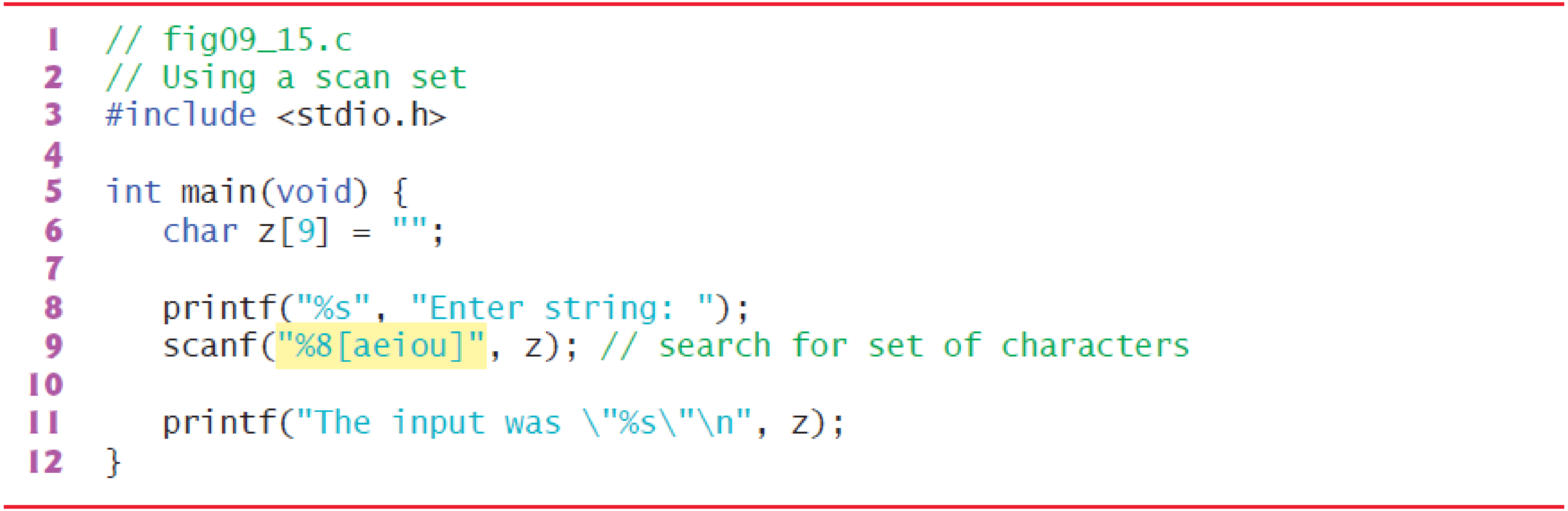
```c
// fig09_14.c
// Reading characters and strings
#include <stdio.h>

int main(void) {
   char x = '\0';
   char y[9] = "";

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

   printf("The input was '%c' and \"%s\"\n", x, y);
}
```

Stops reading when it has consumed 8 characters **OR** when it encounters the first whitespace character (space, tab, or newline). Whichever comes first

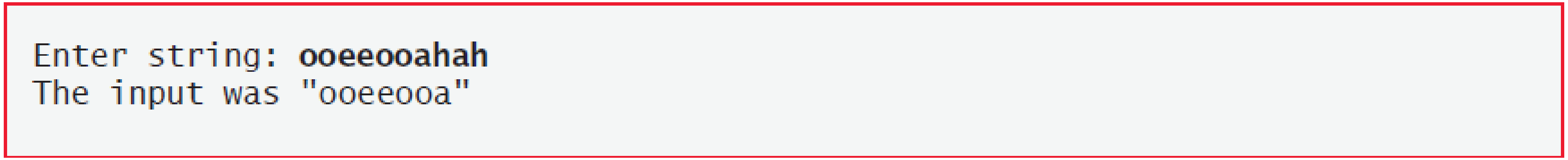


```
Enter a string: Sunday
The input was 'S' and "unday"
```

# $scanf$ – Using Scan Sets

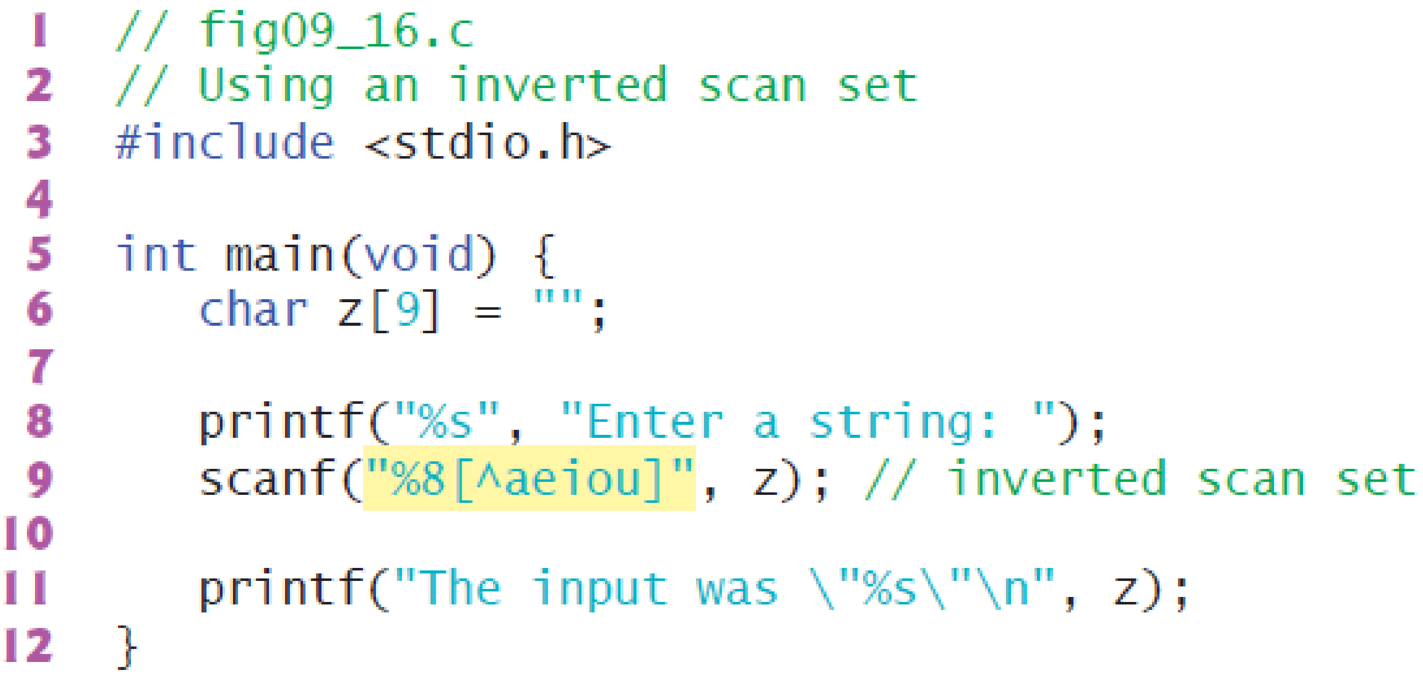


```c
// fig09_15.c
// Using a scan set
#include <stdio.h>

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

   printf("%s", "Enter string: ");
   scanf("%8[aeiou]", z); // search for set of characters

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

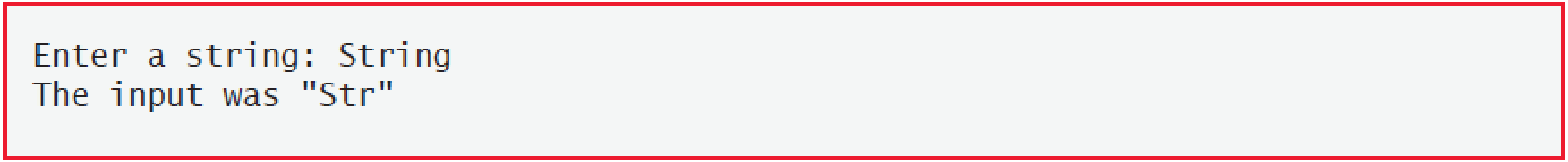


```
Enter string: ooeeooahah
The input was "ooeeooa"
```

# $scanf$ – Using Inverted Scan Sets

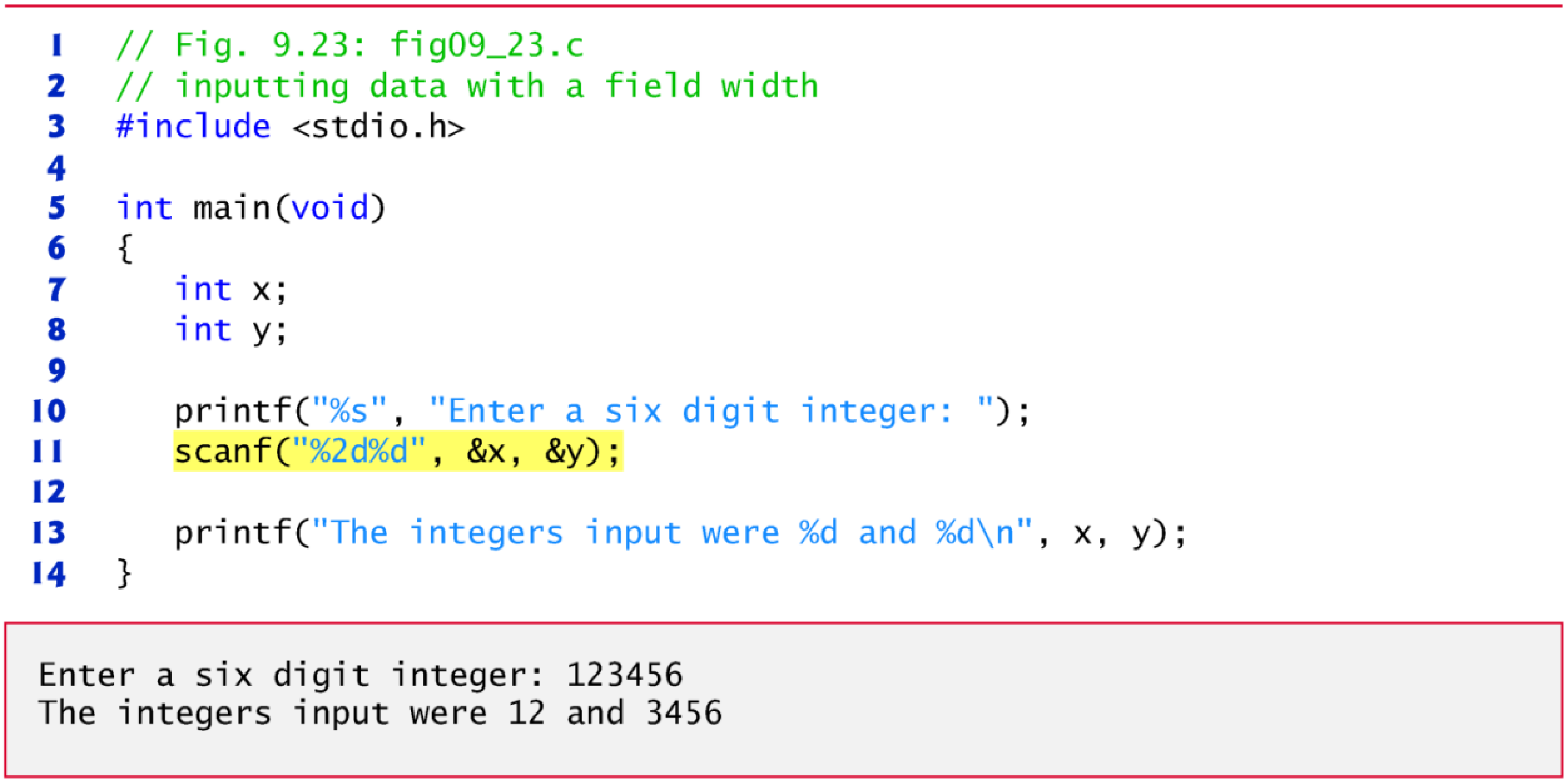


```c
// fig09_16.c
// Using an inverted scan set
#include <stdio.h>

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);
}
```



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

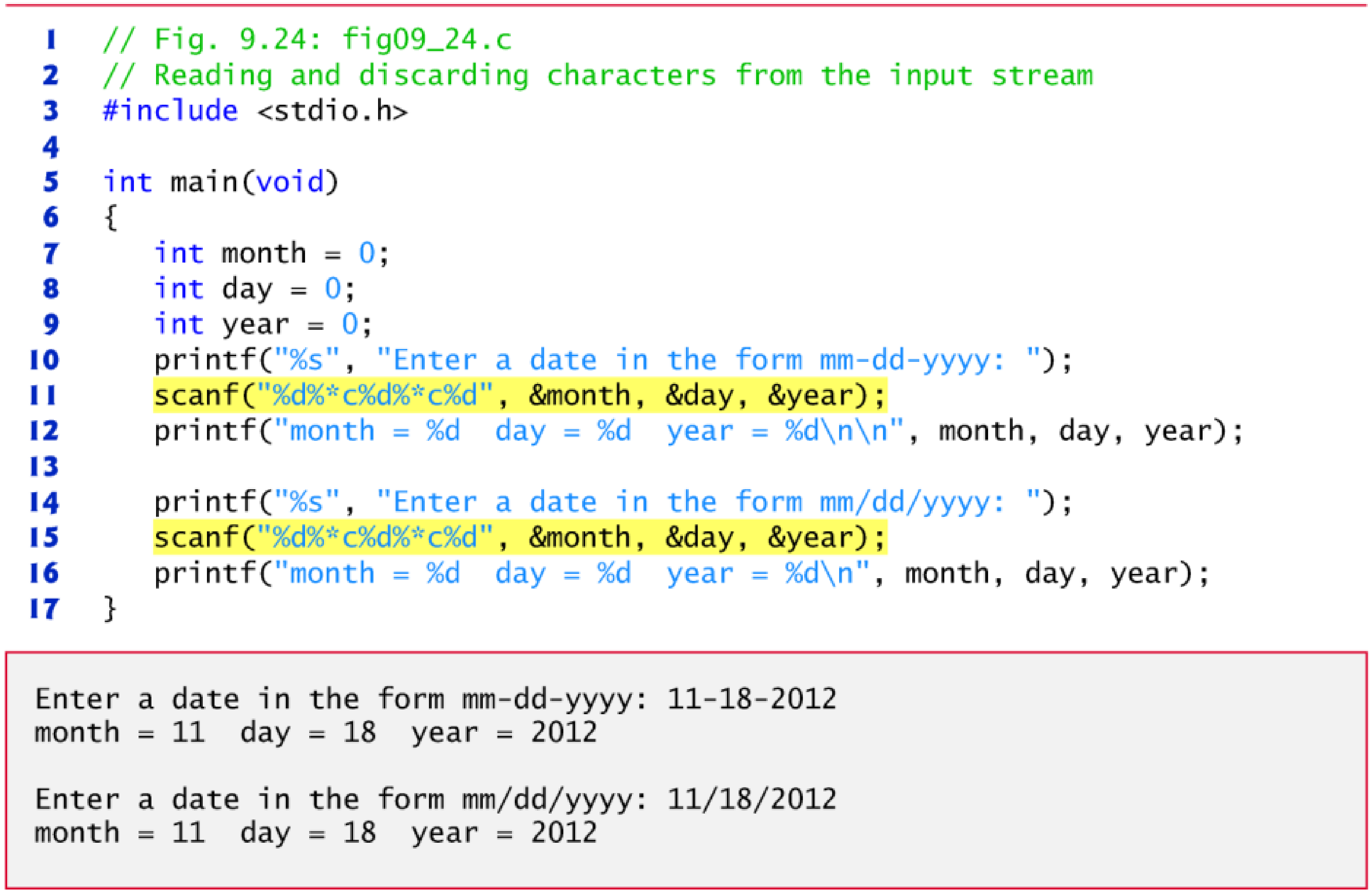
# $scanf$ – Using Field Widths



```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);
}
```

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

# $scanf$ – Reading and Discarding Characters



```c
// Fig. 9.24: fig09_24.c
// Reading and discarding characters from the input stream
#include <stdio.h>

int main(void)
{
   int month = 0;
   int day = 0;
   int year = 0;
   printf("%s", "Enter a date in the form mm-dd-yyyy: ");
   scanf("%d%*c%d%*c%d", &month, &day, &year);
   printf("month = %d  day = %d  year = %d\n\n", month, day, year);

   printf("%s", "Enter a date in the form mm/dd/yyyy: ");
   scanf("%d%*c%d%*c%d", &month, &day, &year);
   printf("month = %d  day = %d  year = %d\n", month, day, year);
}
```

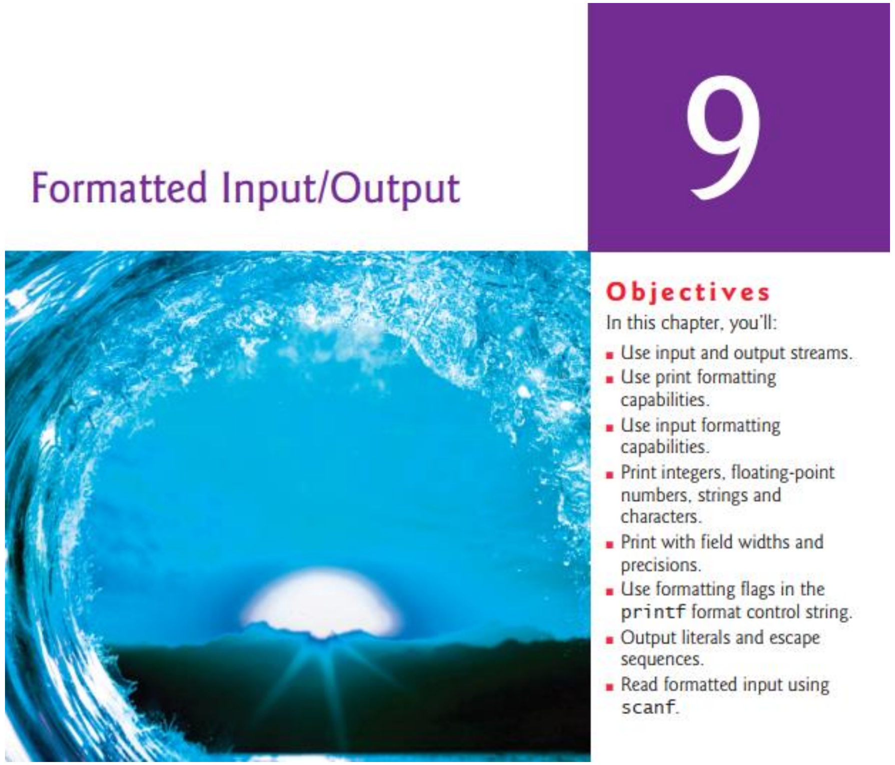
```
Enter a date in the form mm-dd-yyyy: 11-18-2012
month = 11  day = 18  year = 2012

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

# In-class exercise

-Rebuild all the  above example codes and test  for different inputs

-Observe the outputs and explain the codes.

-READ CAREFULLY CHAPTER 9 IN THE TEXT-BOOK.



9

Formatted Input/Output

**Objectives**

In this chapter, you'll:

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