CMPE 252
C PROGRAMMING

SPRING 2022

WEEK 2 & 3

TOP-DOWN DESIGN WITH FUNCTIONS
CHAPTER 3

Problem Solving & Program Design in C

Eighth Edition
Global Edition

Jeri R. Hanly & Elliot B. Koffman

Library Functions

- code reuse
  - reusing program fragments that have already been written and tested
- C standard libraries
  - many predefined functions can be found here

C Library Functions

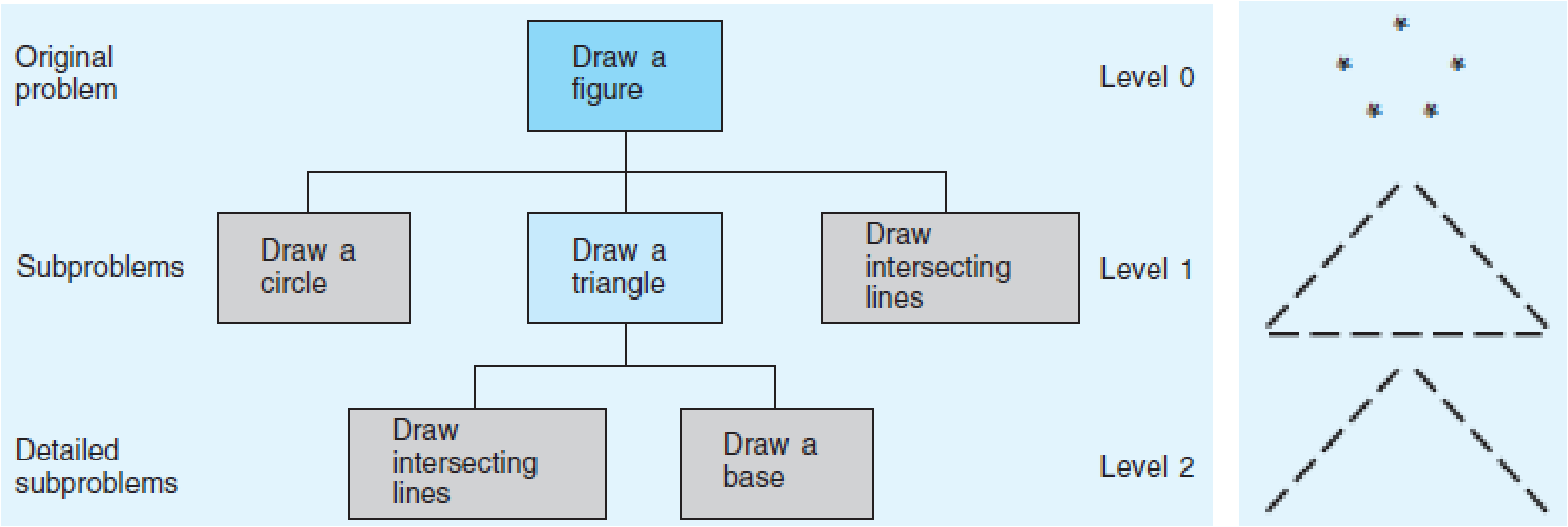
- You need to prepend library name with #include
- Examples:

Function	Standard header file	Purpose	Arguments	Result
abs(x)	<stdlib.h>	Absolute value computation, e.g. abs(-5) is 5	int	int
ceil(x)	<math.h>	Smallest integral value that is not less than x, e.g. ceil(45.23) is 46.0	double	double
log(x)	<math.h>	Natural logarithm of x.	double	double
pow(x,y)	<math.h>	x^y	double, double	double
sin(x)	<math.h>	Sine of angle x, e.g. sin(1.5708) is 1.0	double (radians)	double

Top-Down Design and Structure Charts

- top-down design
  - a problem solving method
  - first, break a problem up into its major subproblems
  - solve the subproblems to derive the solution to the original problem
- structure chart
  - a documentation tool that shows the relationships among the subproblems of a problem

Figure 3.10
Structure Chart for Drawing a Stick Figure



Original problem

Draw a figure

Level 0

Subproblems

Draw a circle

Draw a triangle

Draw intersecting lines

Level 1

Detailed subproblems

Draw intersecting lines

Draw a base

Level 2

Functions Call Statement (Function Without Arguments)

- Syntax

```
fname();
```

- Example:

```
draw_circle();
```

- Interpretation
  - the function fname is called
  - after fname has finished execution, the program statement that follows the function call will be executed

Function Prototype
(Function Without Arguments)

- Syntax

```
ftype fname(void);
```

- Example:

```
void draw_circle(void);
```

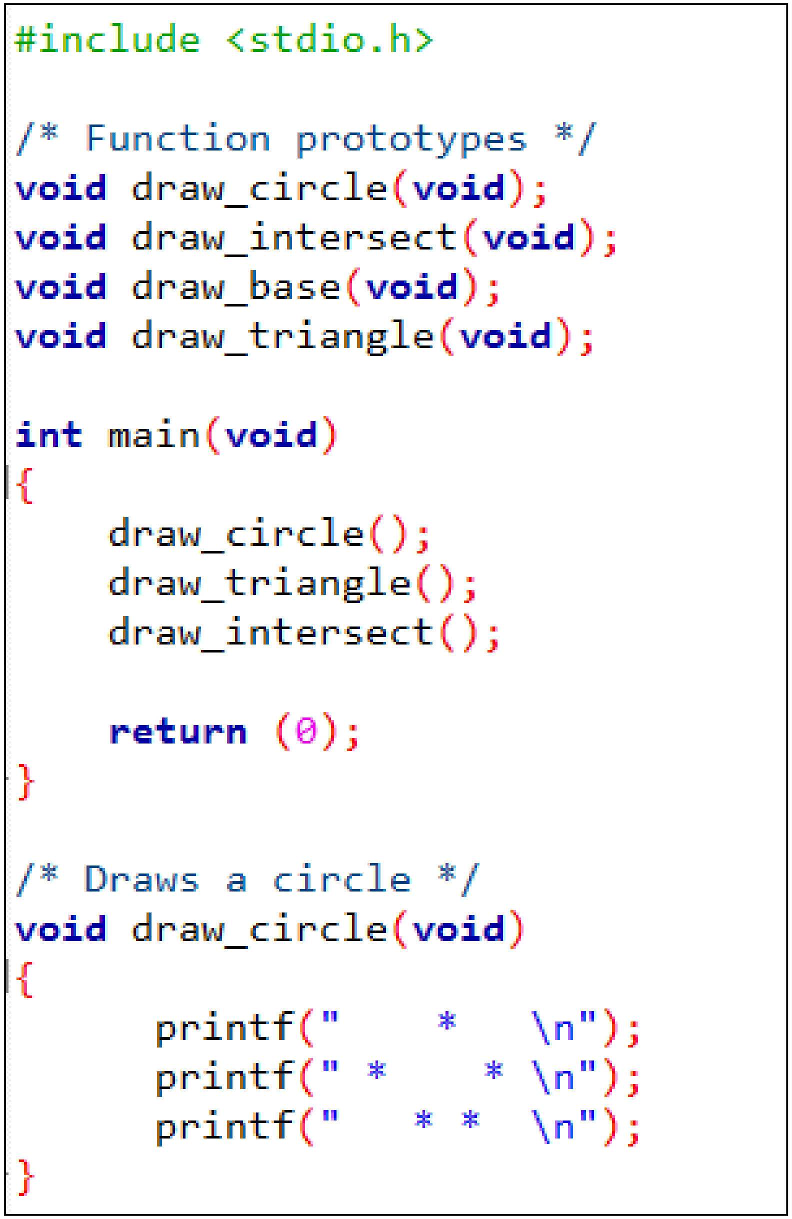
- Interpretation
  - the identifier fname is declared to be the name of a function
  - the identifier ftype specifies the data type of the function result

Function Definitions
(Function Without Arguments)

- Syntax

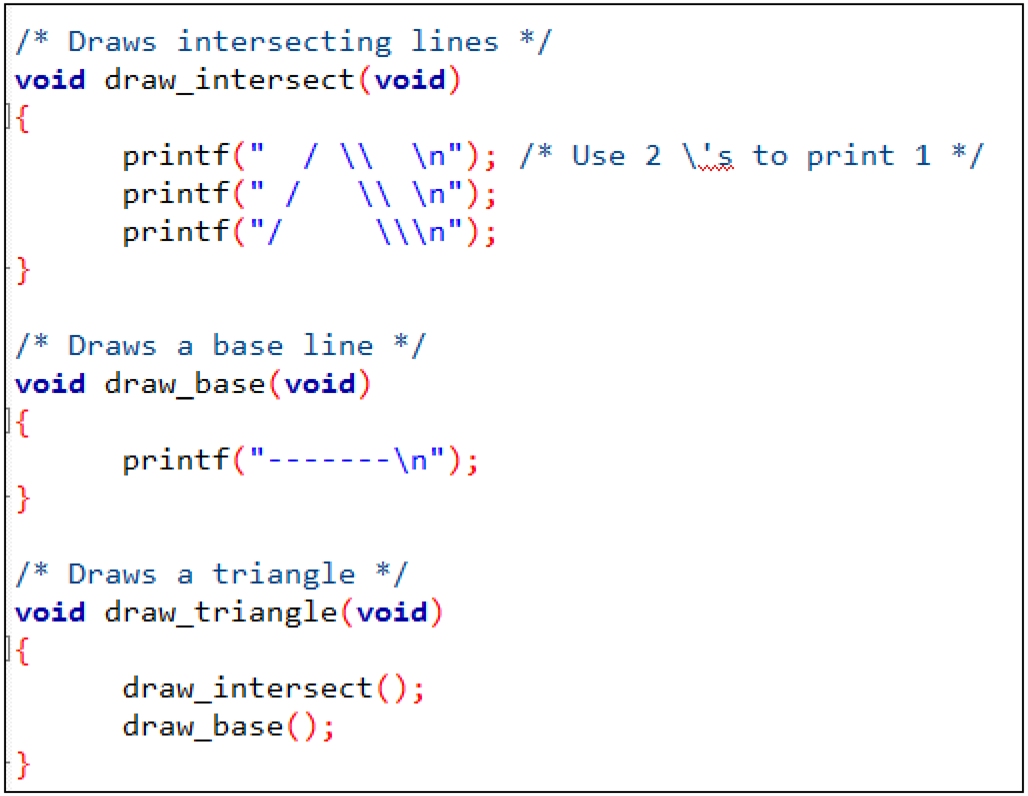
```
ftype fname(void)
{
        local declarations
        executable statements
}
```

Figure 3.14



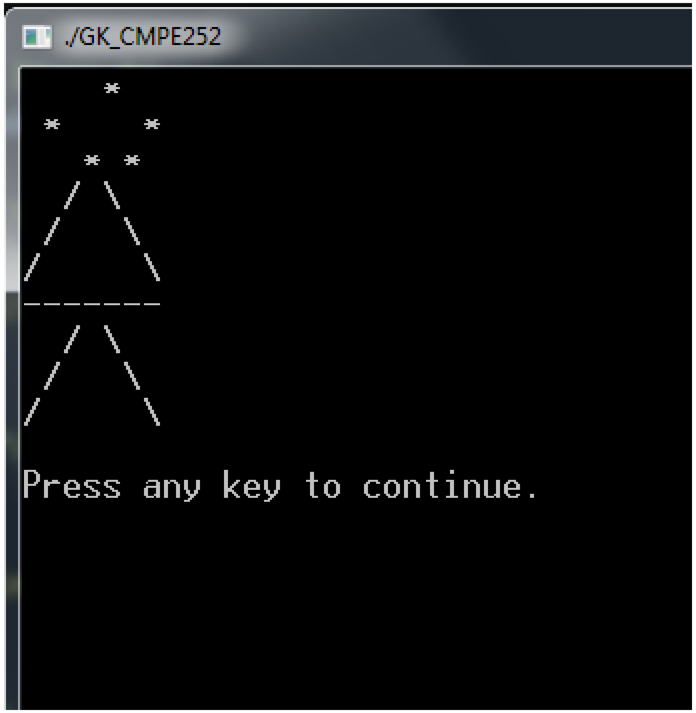
```c
#include <stdio.h>

/* Function prototypes */
void draw_circle(void);
void draw_intersect(void);
void draw_base(void);
void draw_triangle(void);

int main(void)
{
    draw_circle();
    draw_triangle();
    draw_intersect();

    return (0);
}

/* Draws a circle */
void draw_circle(void)
{
      printf("    *   \n");
      printf(" *    * \n");
      printf("   * *  \n");
}
```

Figure 3.14 (cont.)



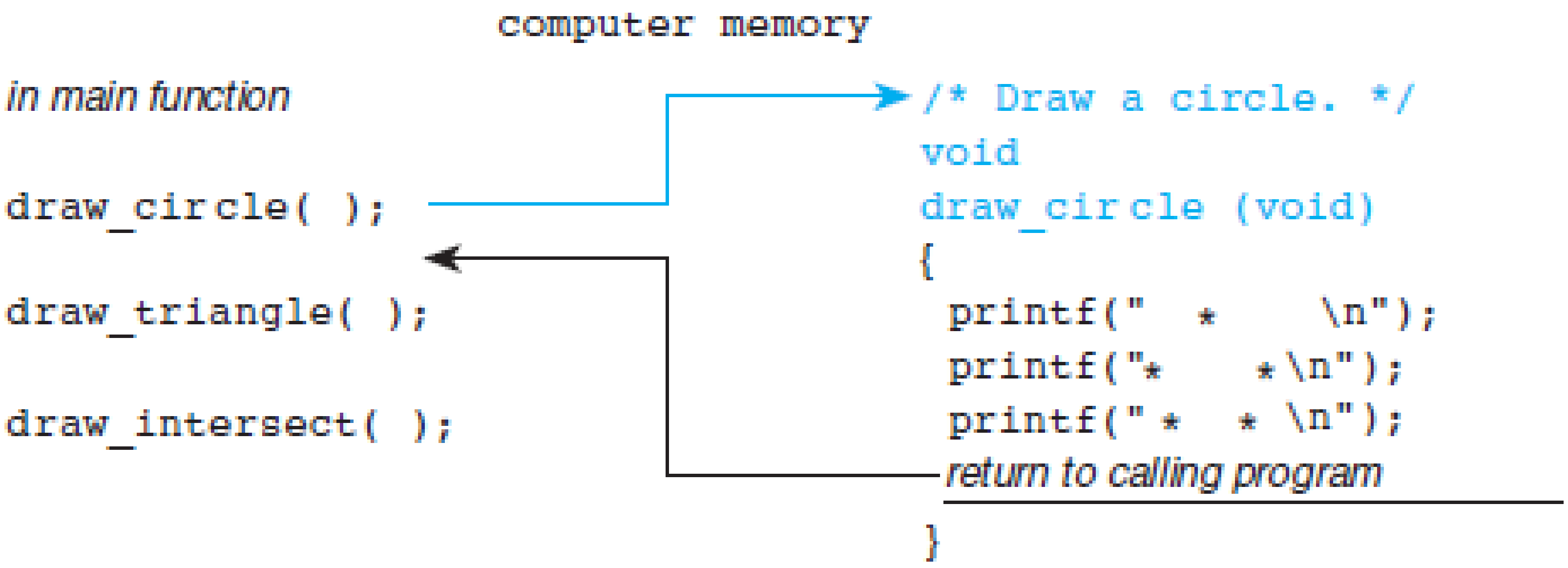
```
/* Draws intersecting lines */
void draw_intersect(void)
{
      printf("  / \\  \n"); /* Use 2 \'s to print 1 */
      printf(" /   \\ \n");
      printf("/     \\\n");
}

/* Draws a base line */
void draw_base(void)
{
      printf("-------\n");
}

/* Draws a triangle */
void draw_triangle(void)
{
      draw_intersect();
      draw_base();
}
```

Output



```
    *
 *    *
   * *
  /\
 /  \
/    \
-------
  /\
 /  \
/    \

Press any key to continue.
```

Figure 3.15
Flow of Control Between the main Function and a Function Subprogram



```
draw_circle( );

draw_triangle( );

draw_intersect( );
```

```
/* Draw a circle. */
void
draw_circle (void)
{
 printf("   *    \n");
 printf(" *    *\n");
 printf("  *   * \n");
}
```

Functions with Input Arguments

- input argument
  - arguments used to pass information into a function subprogram

- output argument
  - arguments used to return results to the calling function

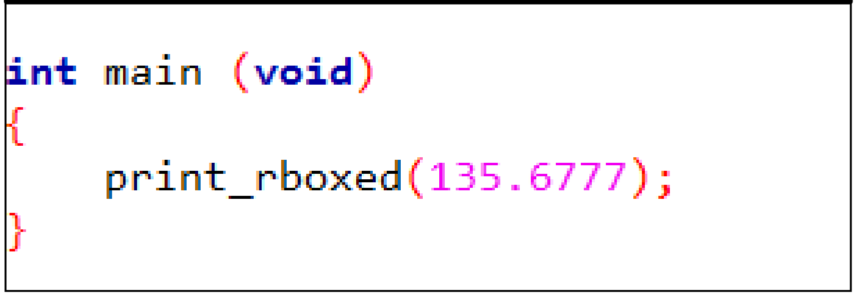
void Functions with Input Arguments

- actual argument
  - an expression used inside the parentheses of a function call

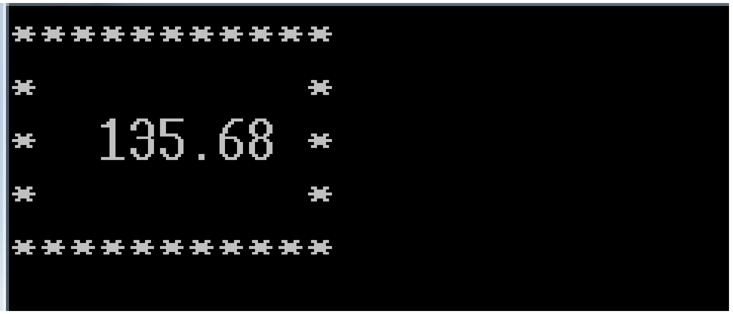
- formal parameter
  - an identifier that represents a corresponding actual argument in a function definition

Figure 3.18
Function print_rboxed and Sample Run

Quick check: Write a function called print_rboxed which *gets a double number as parameter* and *returns nothing*. See the main method and the sample run below.



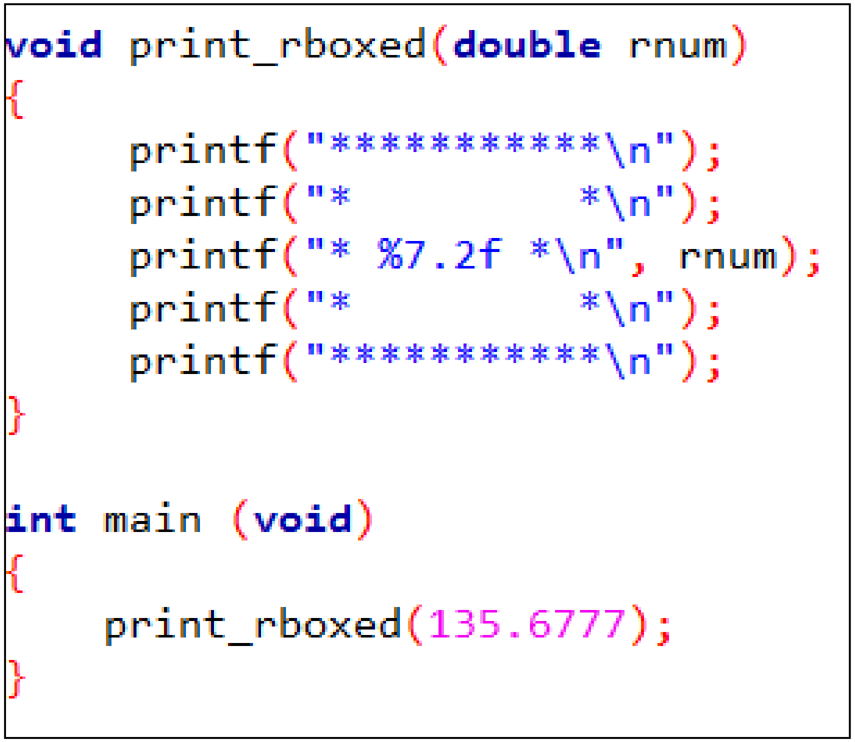
```
int main (void)
{
    print_rboxed(135.6777);
}
```



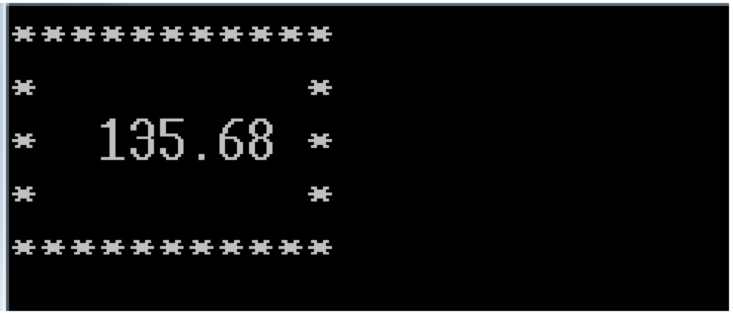
```
***********
*         *
*  135.68 *
*         *
***********
```

Figure 3.18
Function print_rboxed and Sample Run

Quick check: Write a function called print_rboxed which *gets a double number as parameter* and *returns nothing*. See the main method and the sample run below.



```c
void print_rboxed(double rnum)
{
     printf("***********\n");
     printf("*         *\n");
     printf("* %7.2f *\n", rnum);
     printf("*         *\n");
     printf("***********\n");
}

int main (void)
{
    print_rboxed(135.6777);
}
```



```
***********
*         *
*  135.68 *
*         *
***********
```

Function Definition
(Input Arguments and a Single Result)

- Syntax

What can it be?

```
function interface comment
ftype  fname(formal parameter declaration list)
{
        local variable declarations
        executable statements
}
```

Function Interface Comment

- precondition
  - a condition assumed to be true before a function call

- postcondition
  - a condition assumed to be true after a function executes

Quick Check

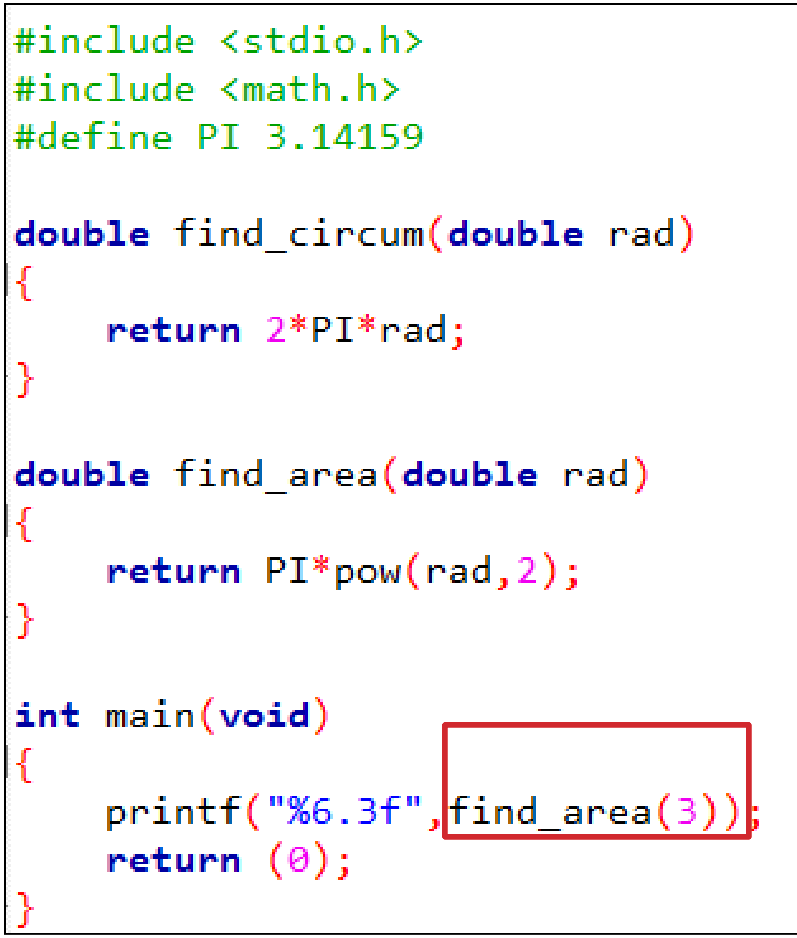
Write 2 functions find_circum and find_area where each one has one formal double parameter: radius and returns circumference or area. Use pow function of Math library.

What are their pre- and post conditions?

Quick Check

Write 2 functions find_circum and find_area where each one has one formal double parameter: radius and returns circumference or area. Use pow function of Math library.

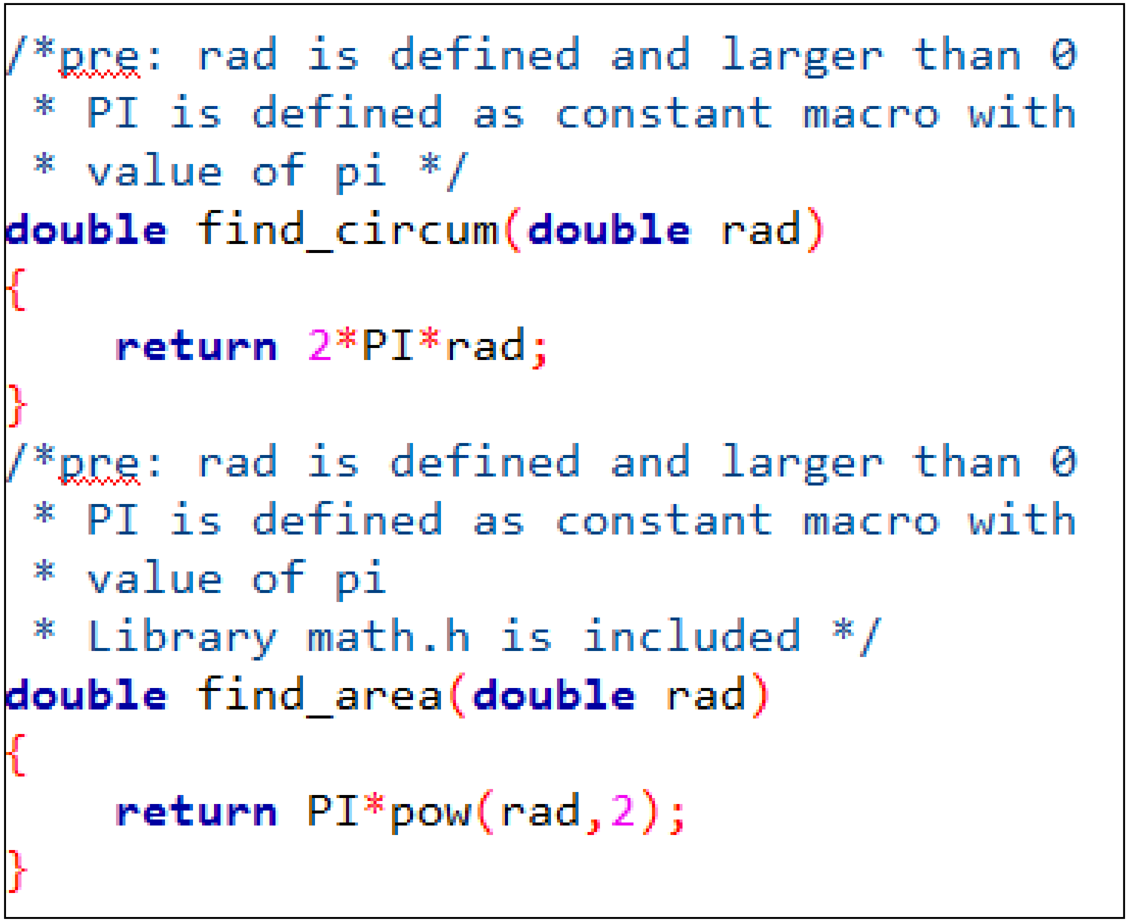
What are their pre- and post conditions?



```
#include <stdio.h>
#include <math.h>
#define PI 3.14159

double find_circum(double rad)
{
    return 2*PI*rad;
}

double find_area(double rad)
{
    return PI*pow(rad,2);
}

int main(void)
{
    printf("%6.3f",find_area(3));
    return (0);
}
```

Output: 28.274

Quick Check



```
/*pre: rad is defined and larger than 0
 * PI is defined as constant macro with
 * value of pi */
double find_circum(double rad)
{
    return 2*PI*rad;
}
/*pre: rad is defined and larger than 0
 * PI is defined as constant macro with
 * value of pi
 * Library math.h is included */
double find_area(double rad)
{
    return PI*pow(rad,2);
}
```

Functions with Multiple Arguments

Argument List Correspondence

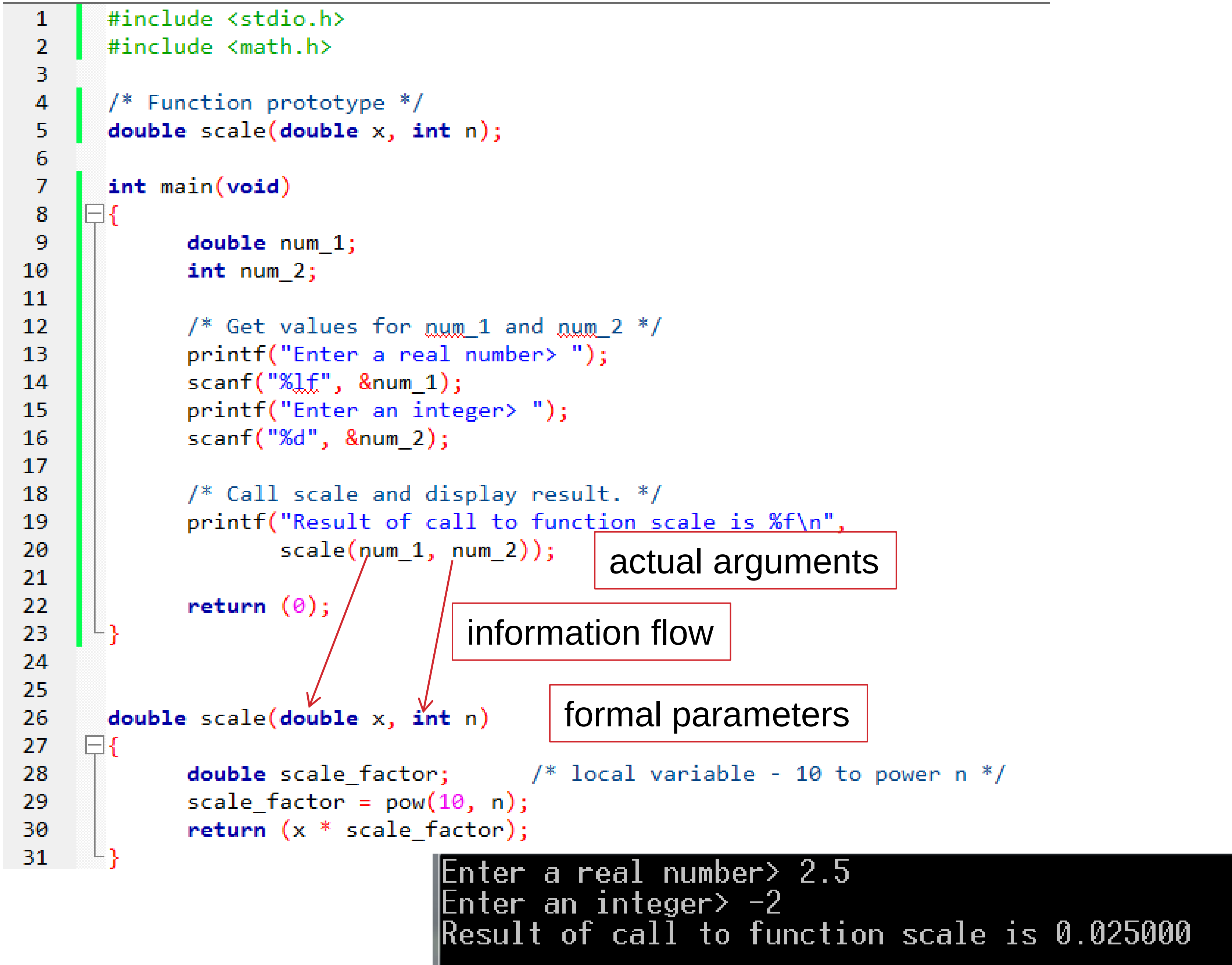
- The number of actual arguments used in a call to a function must be the same as the number of formal parameters listed in the function prototype.

- Each actual argument must be of a data type that can be assigned to the corresponding format parameter with no unexpected loss of information.

Functions with Multiple Arguments

Argument List Correspondence

- The order of arguments in the lists determines correspondence.
  - The first actual argument corresponds to the first formal parameter.
  - The second actual argument corresponds to the second form parameter.
  - And so on…



```c
#include <stdio.h>
#include <math.h>

/* Function prototype */
double scale(double x, int n);

int main(void)
{
        double num_1;
        int num_2;

        /* Get values for num_1 and num_2 */
        printf("Enter a real number> ");
        scanf("%lf", &num_1);
        printf("Enter an integer> ");
        scanf("%d", &num_2);

        /* Call scale and display result. */
        printf("Result of call to function scale is %f\n",
               scale(num_1, num_2));

        return (0);
}


double scale(double x, int n)
{
        double scale_factor;     /* local variable - 10 to power n */
        scale_factor = pow(10, n);
        return (x * scale_factor);
}
```

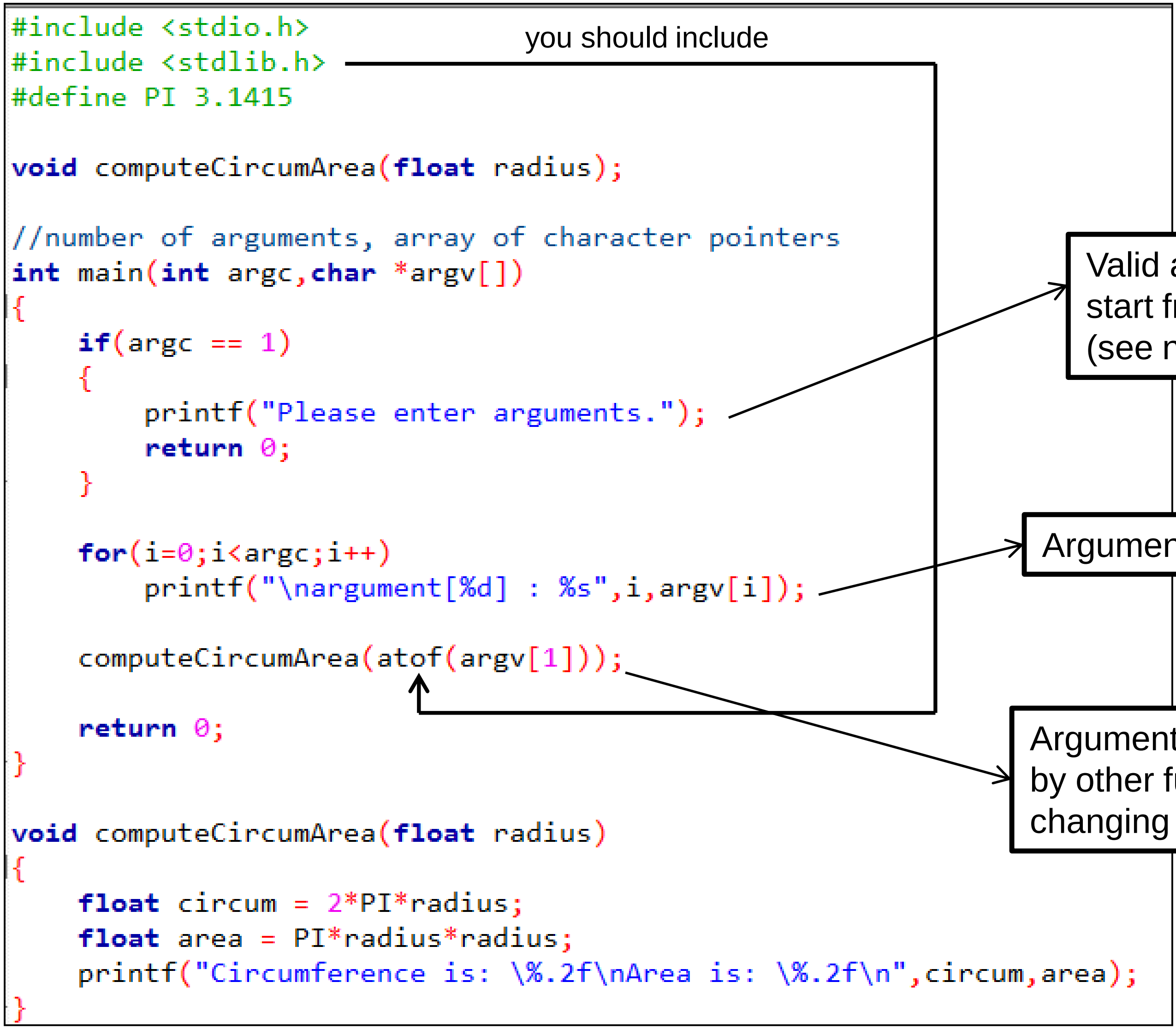
actual arguments

information flow

formal parameters

```
Enter a real number> 2.5
Enter an integer> -2
Result of call to function scale is 0.025000
```

Command Line Arguments



```c
#include <stdio.h>
#include <stdlib.h>
#define PI 3.1415

void computeCircumArea(float radius);

//number of arguments, array of character pointers
int main(int argc,char *argv[])
{
    if(argc == 1)
    {
        printf("Please enter arguments.");
        return 0;
    }

    for(i=0;i<argc;i++)
        printf("\nargument[%d] : %s",i,argv[i]);

    computeCircumArea(atof(argv[1]));

    return 0;
}

void computeCircumArea(float radius)
{
    float circum = 2*PI*radius;
    float area = PI*radius*radius;
    printf("Circumference is: \%.2f\nArea is: \%.2f\n",circum,area);
}
```

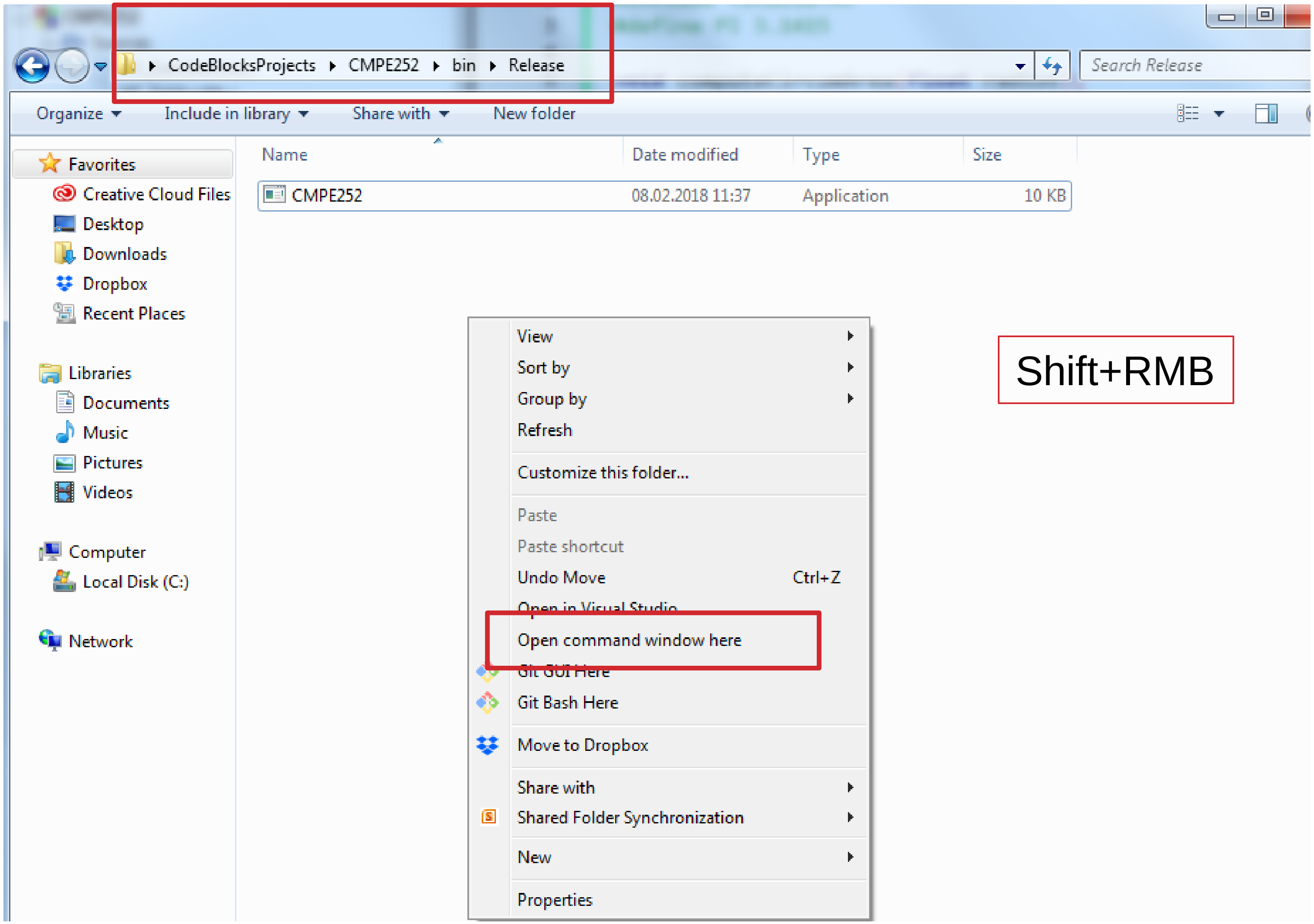
you should include

Valid arguments start from 1, not 0 (see next page)

Arguments are strings

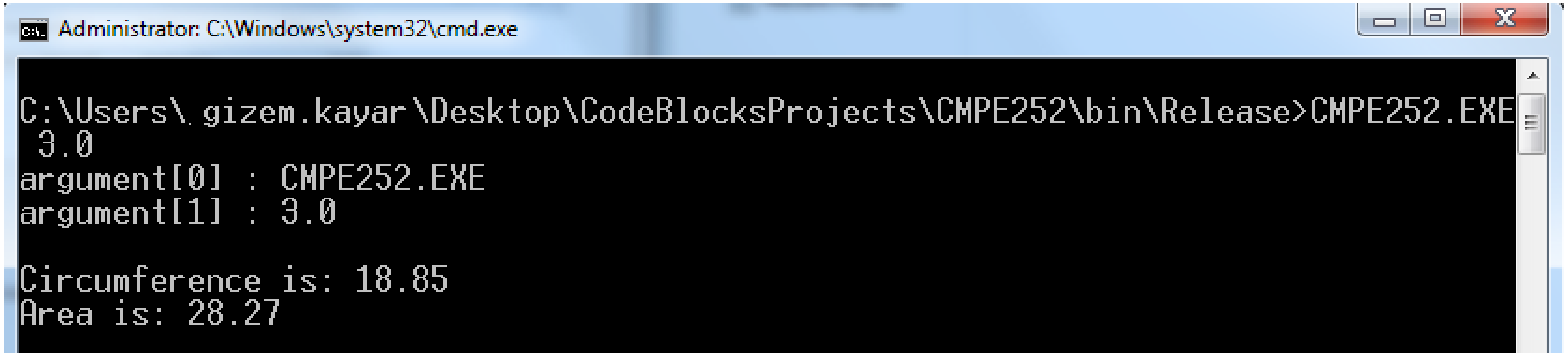
Arguments can be called by other functions after changing their types

From Command Window

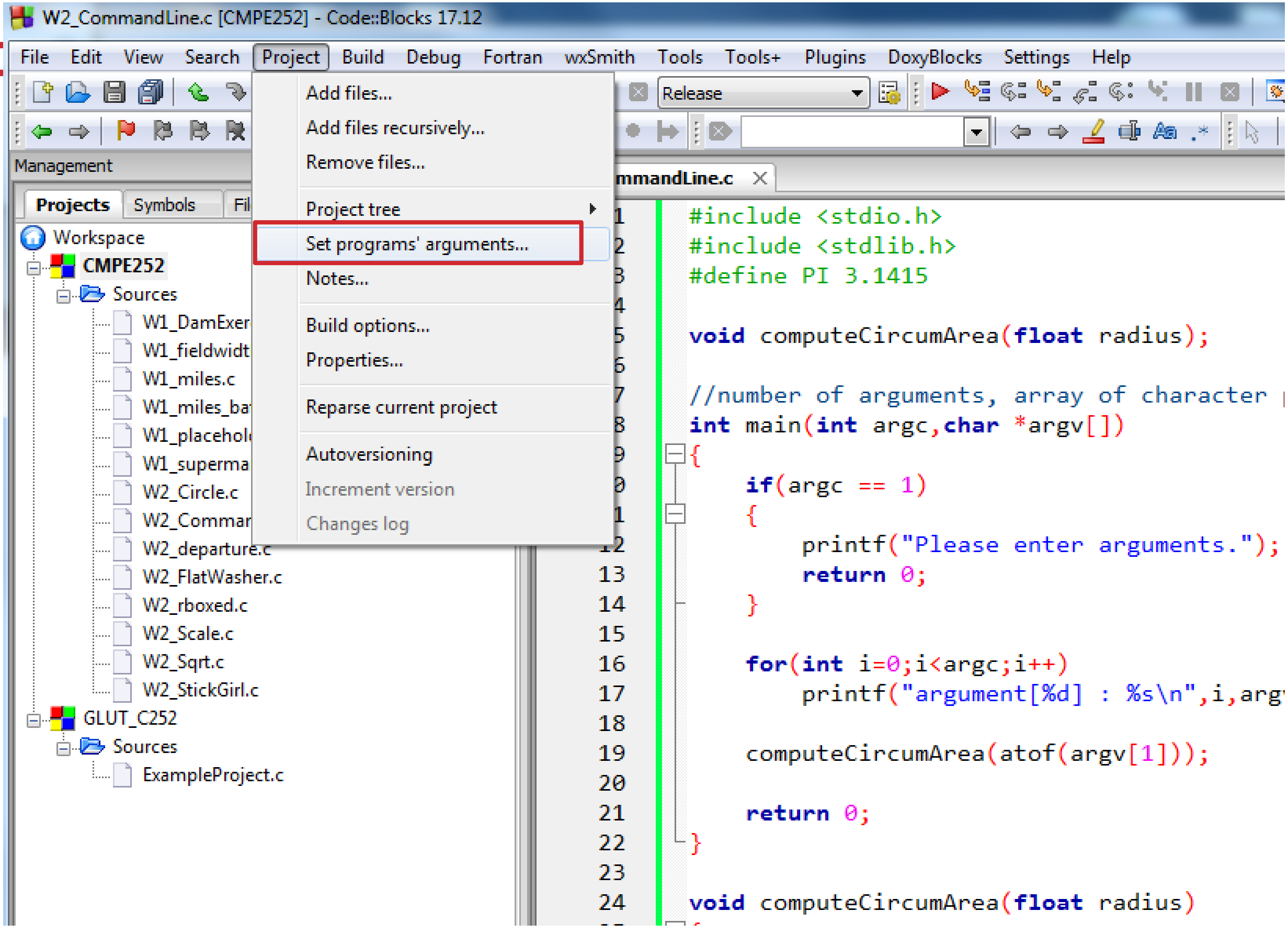


Name	Date modified	Type	Size
CMPE252	08.02.2018 11:37	Application	10 KB

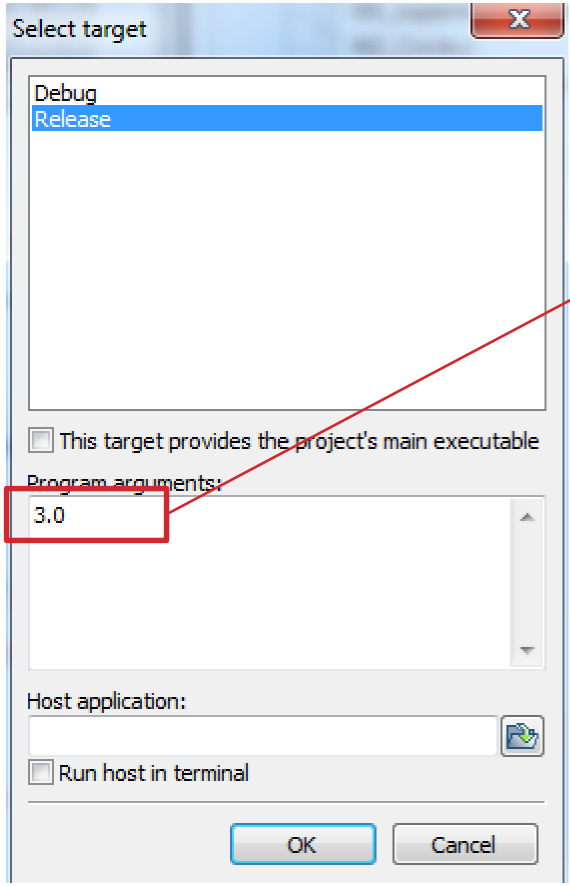
Shift+RMB



```
C:\Users\ gizem.kayar\Desktop\CodeBlocksProjects\CMPE252\bin\Release>CMPE252.EXE
 3.0
argument[0] : CMPE252.EXE
argument[1] : 3.0

Circumference is: 18.85
Area is: 28.27
```



```c
#include <stdio.h>
#include <stdlib.h>
#define PI 3.1415

void computeCircumArea(float radius);

//number of arguments, array of character 
int main(int argc,char *argv[])
{
    if(argc == 1)
    {
        printf("Please enter arguments.");
        return 0;
    }

    for(int i=0;i<argc;i++)
        printf("argument[%d] : %s\n",i,arg

    computeCircumArea(atof(argv[1]));

    return 0;
}

void computeCircumArea(float radius)
```



Select target

Debug

Release

□ This target provides the project's main executable

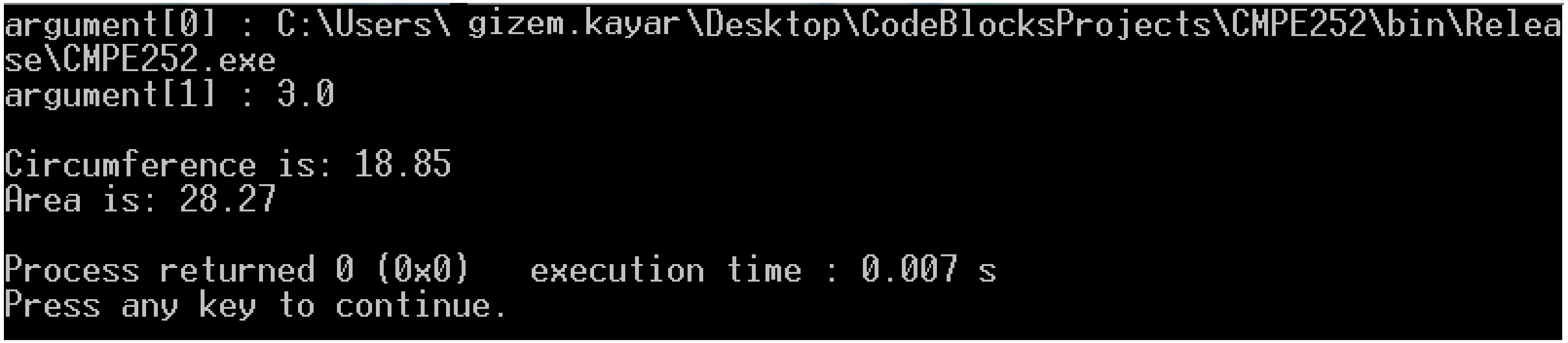
Program arguments:

3.0

Host application:

□ Run host in terminal

Press OK, now build and run your code



```
argument[0] : C:\Users\ gizem.kayar\Desktop\CodeBlocksProjects\CMPE252\bin\Relea
se\CMPE252.exe
argument[1] : 3.0

Circumference is: 18.85
Area is: 28.27

Process returned 0 (0x0)   execution time : 0.007 s
Press any key to continue.
```

SELECTION STRUCTURES: IF AND SWITCH STATEMENTS

CHAPTER 4

Problem Solving & Program Design in C

Eighth Edition
Global Edition

Jeri R. Hanly & Elliot B. Koffman

Control Structures

- control structure
  - a combination of individual instructions into a single logical unit with one entry point and one exit point

Instructions are organized into three kinds of control structures to control the execution flow :

- sequence (compound statements are used to specify sequential flow)
- selection structures (e.g. if then else)
- repetition structures (loops)

Compound Statement

- compound statement
  - a group of statements bracketed by { and } that are executed sequentially

```
{
        statement;
        statement;
                 .
                 .
                 .
        statement;
}
```

Used to specify sequential flow

Conditions

- an expression that is either FALSE
  - represented by 0
- or TRUE
  - usually represented by 1

my_age > 40

In ANSI C , there is NO «bool» type,

anything that is **NOT«0»** is **TRUE**

Relational and Equality Operators

Operator	Meaning	Type
<	less than	relational
>	greater than	relational
<=	less than or equal to	relational
>=	greater than or equal to	relational
==	equal to	equality
!=	not equal to	equality

Logical Operators

- logical expressions
  - an expression that uses one or more of the logical operators
    - && (AND)
    - || (OR)
    - ! (NOT)

Operator Precedence

Operator	Precedence
function calls	highest (evaluated first)
! + - & (unary operator)	↓
* / %	
+ -	
< <= >= >	
== !=	
&&	
||	
=	lowest (evaluated last)

Short-Circuit Evaluation

- stopping evaluation of a logical expression as soon as its value can be determined

```
!flag || (y + z >= x - z)
```

An expression of the form a || b must be true if a is true. Consequently, C stops evaluating the expression when it determines that the value of !flag is 1 (true).

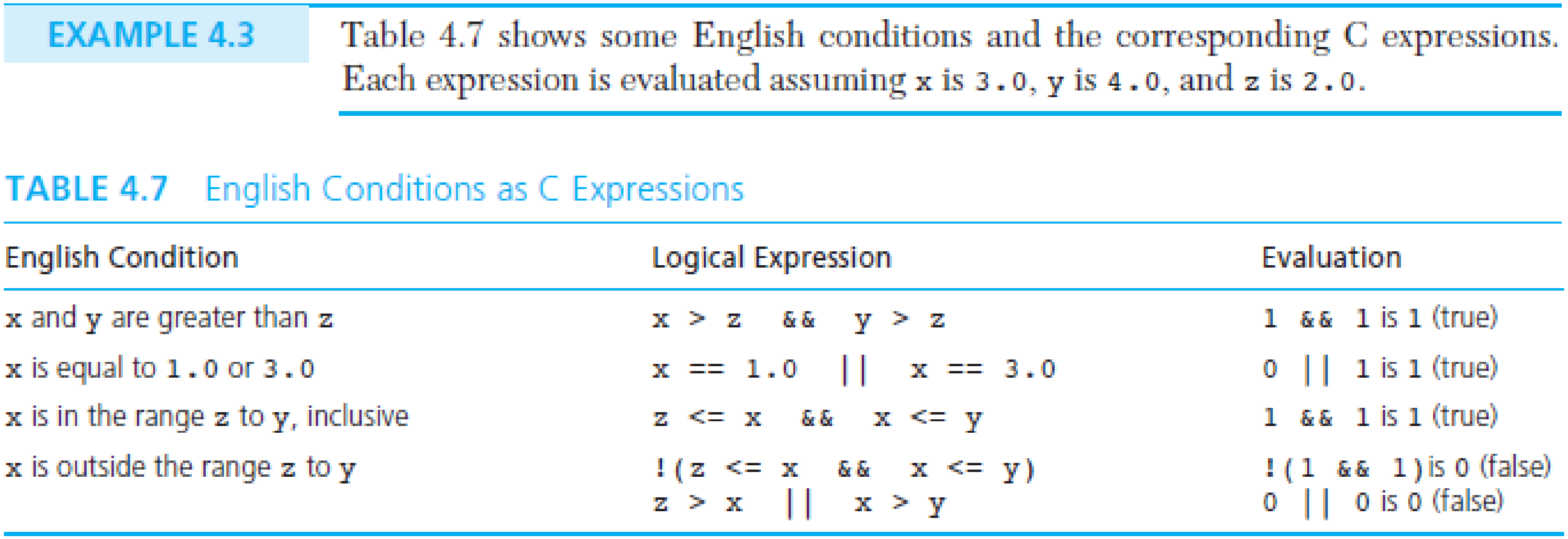
Similarly, an expression of the form a && b must be false if a is false, so C would stop evaluating such an expression if its first operand evaluates to 0.

Short-Circuit Evaluation

- We can use short-circuit evaluation to prevent potential run-time errors.

  e.g. (num % div == 0) – what if div ==0 ?
- In this case, the remainder calculation would cause a *division by zero* run-time error.
- However, we can prevent this error by using the revised condition
- (div != 0 && (num % div == 0))

Writing English Conditions in C



EXAMPLE 4.3 Table 4.7 shows some English conditions and the corresponding C expressions. Each expression is evaluated assuming `x` is `3.0`, `y` is `4.0`, and `z` is `2.0`.

TABLE 4.7 English Conditions as C Expressions

English Condition	Logical Expression	Evaluation
`x` and `y` are greater than `z`	`x > z && y > z`	`1 && 1` is `1` (true)
`x` is equal to `1.0` or `3.0`	`x == 1.0 || x == 3.0`	`0 || 1` is `1` (true)
`x` is in the range `z` to `y`, inclusive	`z <= x && x <= y`	`1 && 1` is `1` (true)

`x` is outside the range `z` to `y`	`!(z <= x && x <= y)`
	`z > x || x > y`
		`!(1 && 1)` is `0` (false)
		`0 || 0` is `0` (false)

Comparing Characters

Expression	Value
‘9’ >= ‘0’	1 (true)
‘a’ < ‘e’	1 (true)
‘B’ <= ‘A’	0 (false)
‘Z’ == ‘z’	0 (false)
‘a’ <= ‘A’	System dependent
‘a’ <= ch && ch <= ‘z’	1 (true) if ch is a lowercase letter

THE IF-STATEMENT

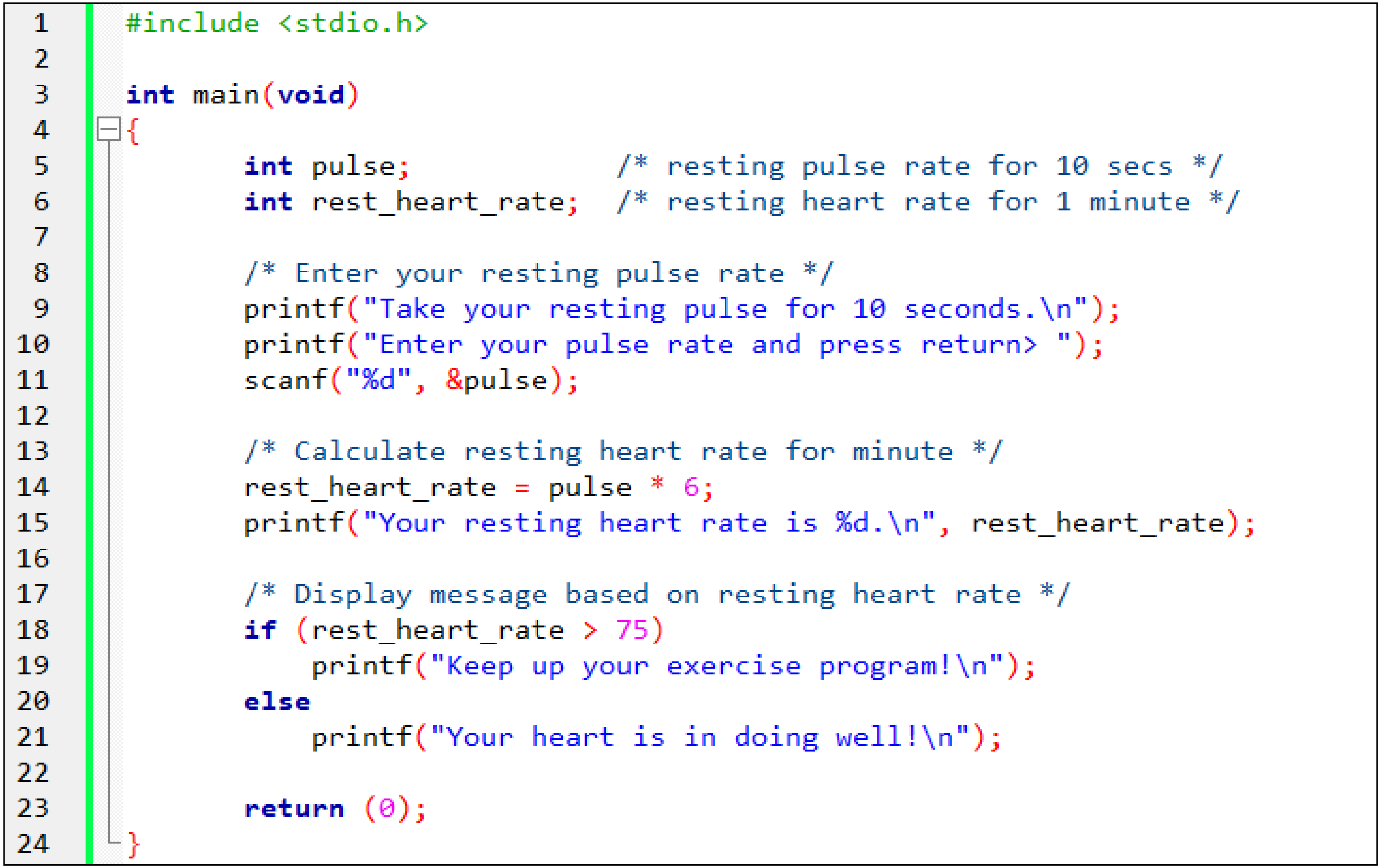
making decisions

if-statement with one alternative

```
if  (x != 0)
          product = product * x;
```

if-statement with two alternatives

```
if  (rest_heart_rate > 75)
        printf(“Keep up your exercise program!\n”);
else
        printf(“Your hear is doing well!\n”);
```



```c
#include <stdio.h>

int main(void)
{
       int pulse;            /* resting pulse rate for 10 secs */
       int rest_heart_rate;  /* resting heart rate for 1 minute */

       /* Enter your resting pulse rate */
       printf("Take your resting pulse for 10 seconds.\n");
       printf("Enter your pulse rate and press return> ");
       scanf("%d", &pulse);

       /* Calculate resting heart rate for minute */
       rest_heart_rate = pulse * 6;
       printf("Your resting heart rate is %d.\n", rest_heart_rate);

       /* Display message based on resting heart rate */
       if (rest_heart_rate > 75)
           printf("Keep up your exercise program!\n");
       else
           printf("Your heart is in doing well!\n");

       return (0);
}
```

Nested if-statements with more than one variable

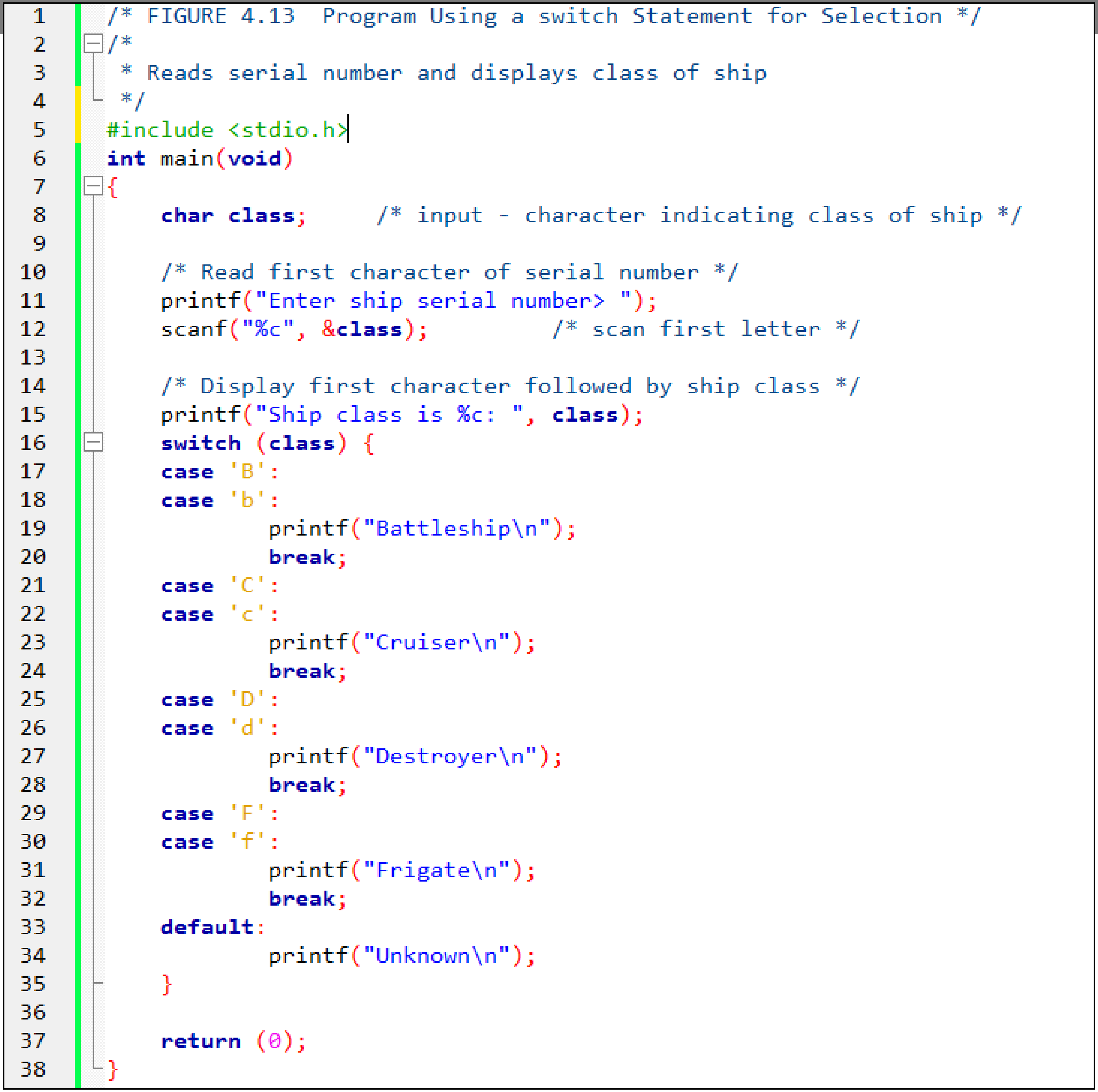
```
if (road_status == ‘S’)
        if (temp > 0) {
                printf(“Wet roads ahead\n”);
                printf(“Stopping time doubled\n”);
        }  else  {
                printf(“Icy roads ahead\n”);
                printf(“Stopping time quadrupled\n”);
        }
else
        printf(“Drive carefully!\n”)
```

The switch statement

- also used to select one of several alternatives
- useful when the selection is based on the value of
  - a single variable
  - or a simple expression
- values may of type int or char
  - not double

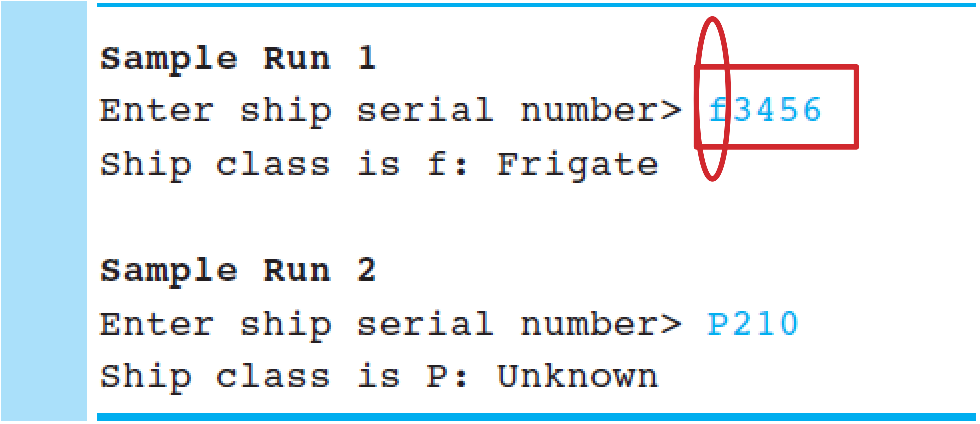
Syntax

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



```c
/* FIGURE 4.13  Program Using a switch Statement for Selection */
/*
 * Reads serial number and displays class of ship
 */
#include <stdio.h>
int main(void)
{
    char class;     /* input - character indicating class of ship */

    /* Read first character of serial number */
    printf("Enter ship serial number> ");
    scanf("%c", &class);         /* scan first letter */

    /* Display first character followed by ship class */
    printf("Ship class is %c: ", class);
    switch (class) {
    case 'B':
    case 'b':
            printf("Battleship\n");
            break;
    case 'C':
    case 'c':
            printf("Cruiser\n");
            break;
    case 'D':
    case 'd':
            printf("Destroyer\n");
            break;
    case 'F':
    case 'f':
            printf("Frigate\n");
            break;
    default:
            printf("Unknown\n");
    }

    return (0);
}
```

Figure 4.13
Program Using a *switch* Statement for Selection (cont.)



```
Sample Run 1
Enter ship serial number> f3456
Ship class is f: Frigate

Sample Run 2
Enter ship serial number> P210
Ship class is P: Unknown
```

REPETITION AND LOOP STATEMENTS
CHAPTER 5

Problem Solving & Program Design in C

Eighth Edition

Global Edition

Jeri R. Hanly & Elliot B. Koffman

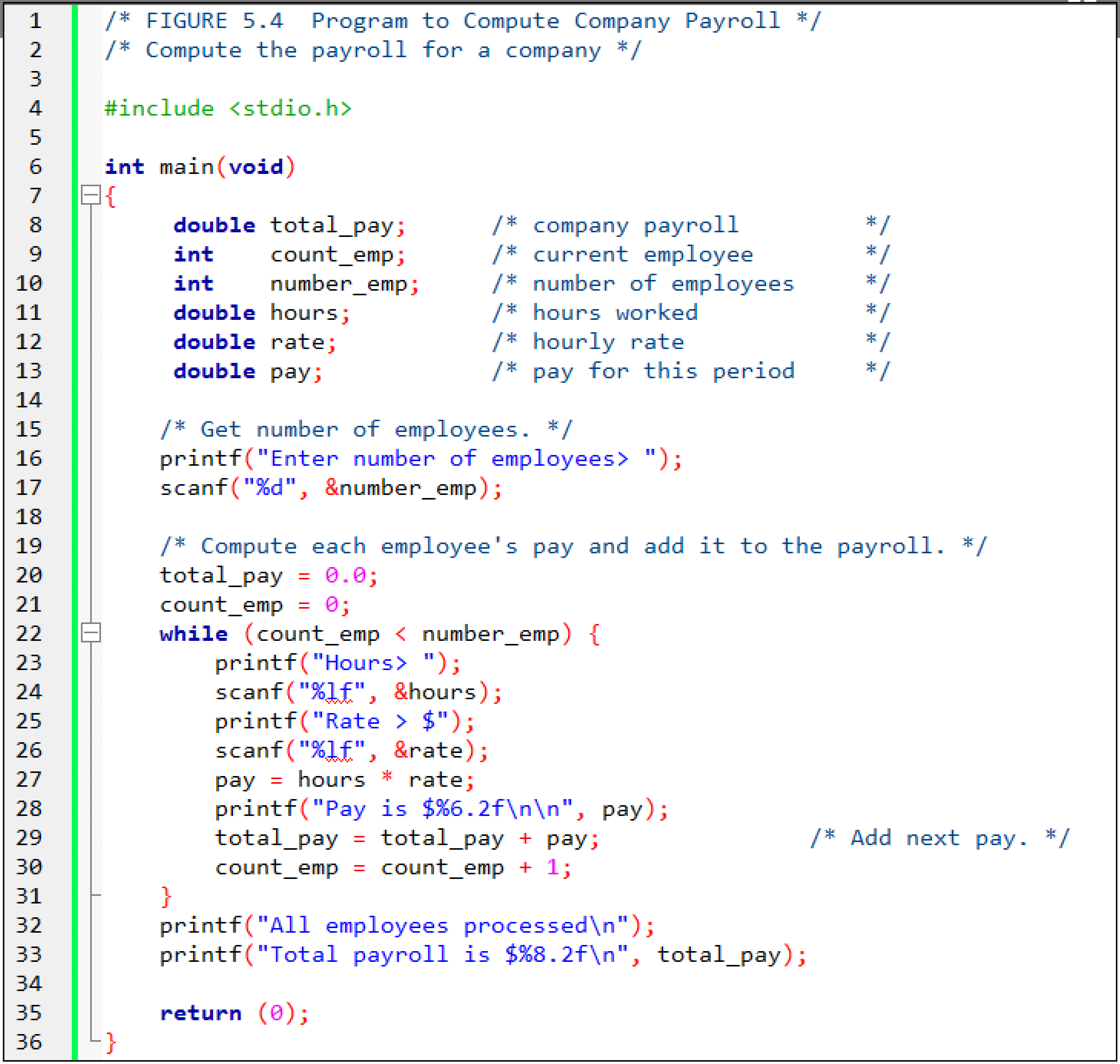
Repetition in Programs

- loop
  - a control structure that repeats a group of steps in a program

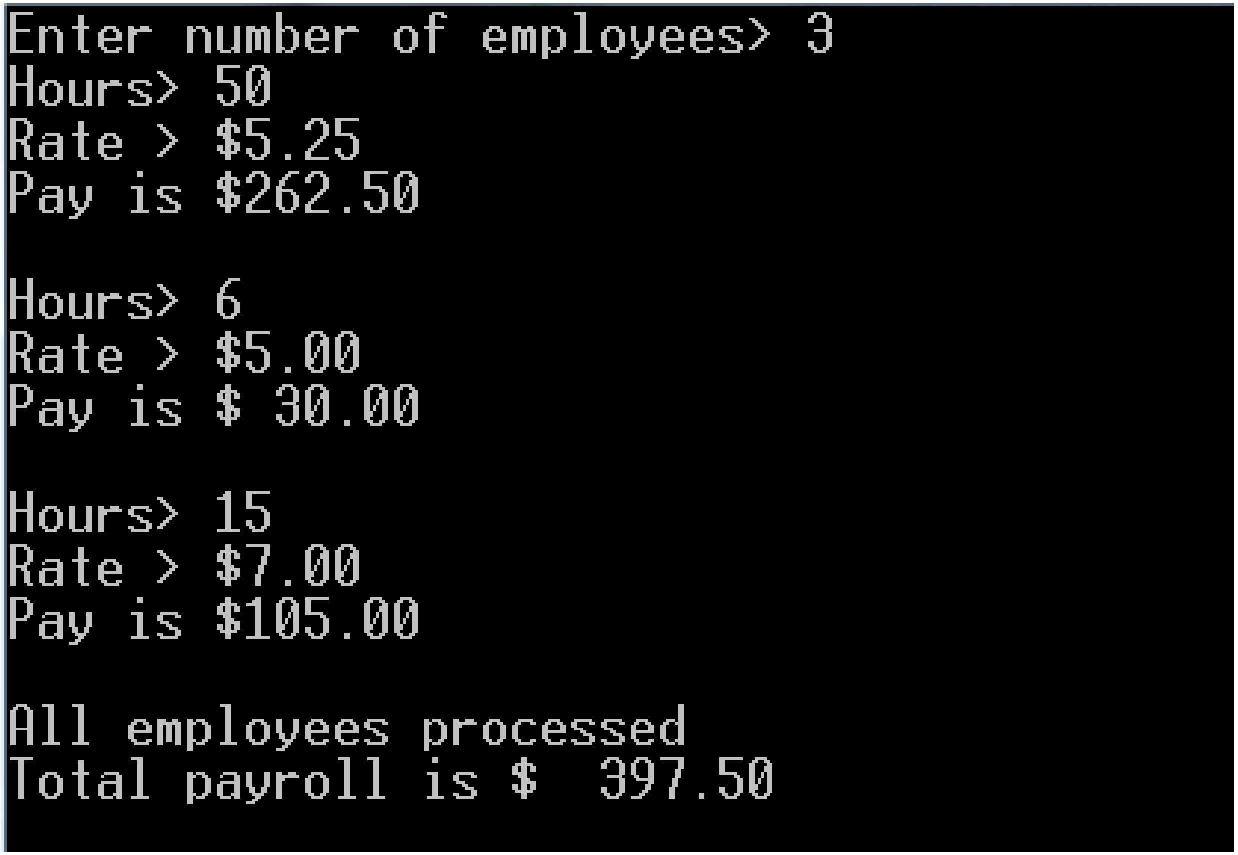
- loop body
  - the statements that are repeated in the loop

while Statement Syntax

```
while (loop repetition condition)
        statement;

/* display N asterisks. */
count_star = 0
while (count_star  <  N) {
        printf(“*”);
        count_star = count_star + 1;
}
```



```c
/* FIGURE 5.4  Program to Compute Company Payroll */
/* Compute the payroll for a company */

#include <stdio.h>

int main(void)
{
     double total_pay;      /* company payroll         */
     int    count_emp;      /* current employee        */
     int    number_emp;     /* number of employees     */
     double hours;          /* hours worked            */
     double rate;           /* hourly rate             */
     double pay;            /* pay for this period     */

    /* Get number of employees. */
    printf("Enter number of employees> ");
    scanf("%d", &number_emp);

    /* Compute each employee's pay and add it to the payroll. */
    total_pay = 0.0;
    count_emp = 0;
    while (count_emp < number_emp) {
        printf("Hours> ");
        scanf("%lf", &hours);
        printf("Rate > $");
        scanf("%lf", &rate);
        pay = hours * rate;
        printf("Pay is $%6.2f\n\n", pay);
        total_pay = total_pay + pay;               /* Add next pay. */
        count_emp = count_emp + 1;
    }
    printf("All employees processed\n");
    printf("Total payroll is $%8.2f\n", total_pay);

    return (0);
}
```

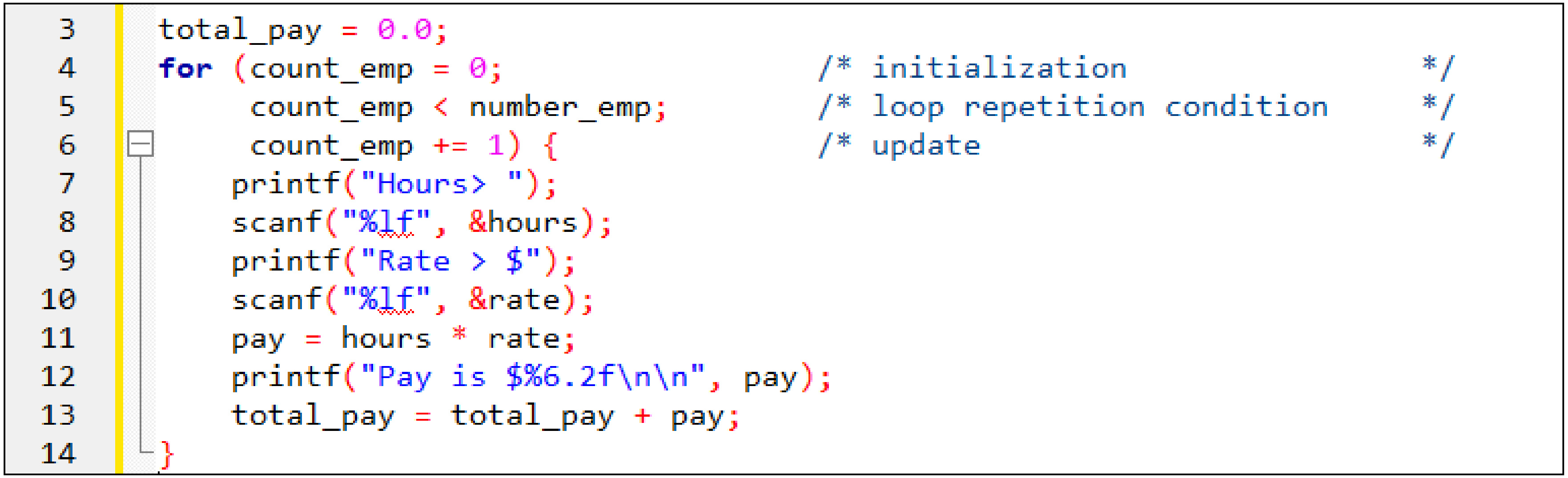


```
Enter number of employees> 3
Hours> 50
Rate > $5.25
Pay is $262.50

Hours> 6
Rate > $5.00
Pay is $ 30.00

Hours> 15
Rate > $7.00
Pay is $105.00

All employees processed
Total payroll is $  397.50
```

The for Statement Syntax

for (*initialization expression*;
    loop repetition condition;
    update expression)
  statement;

```
/* Display N asterisks. */
for  (count_star = 0; count_star < N; count_star += 1)
     printf(“*”);
```



```
total_pay = 0.0;
for (count_emp = 0;                 /* initialization                */
     count_emp < number_emp;        /* loop repetition condition     */
     count_emp += 1) {              /* update                        */
    printf("Hours> ");
    scanf("%lf", &hours);
    printf("Rate > $");
    scanf("%lf", &rate);
    pay = hours * rate;
    printf("Pay is $%6.2f\n\n", pay);
    total_pay = total_pay + pay;
}
```

Increment and Decrement Operators

- counter = counter + 1
  count += 1
  counter++ → Postfix increment
  ++counter → Prefix increment

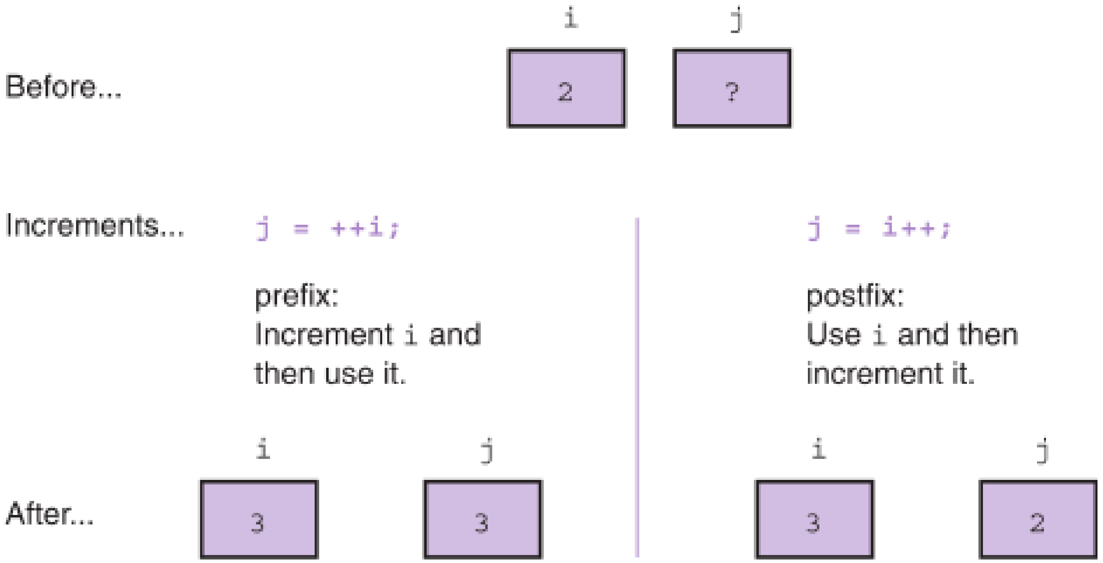
- counter = counter - 1
  count -= 1
  counter-- → Postfix decrement
  --counter → Prefix decrement

Increment and Decrement Operators

- side effect
  - a change in the value of a variable as a result of carrying out an operation

FIGURE 5.6

Comparison of Prefix and Postfix Increments



Before...

i j

2 ?

Increments...

`j = ++i;`

prefix:
Increment i and
then use it.

`j = i++;`

postfix:
Use i and then
increment it.

After...

i j

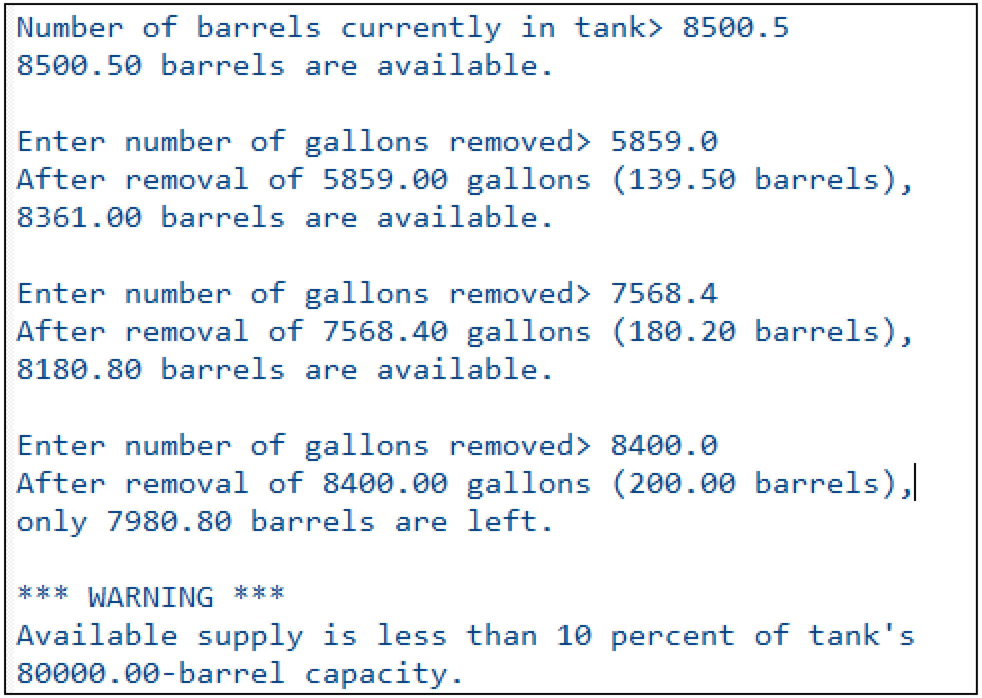
3 3

i j

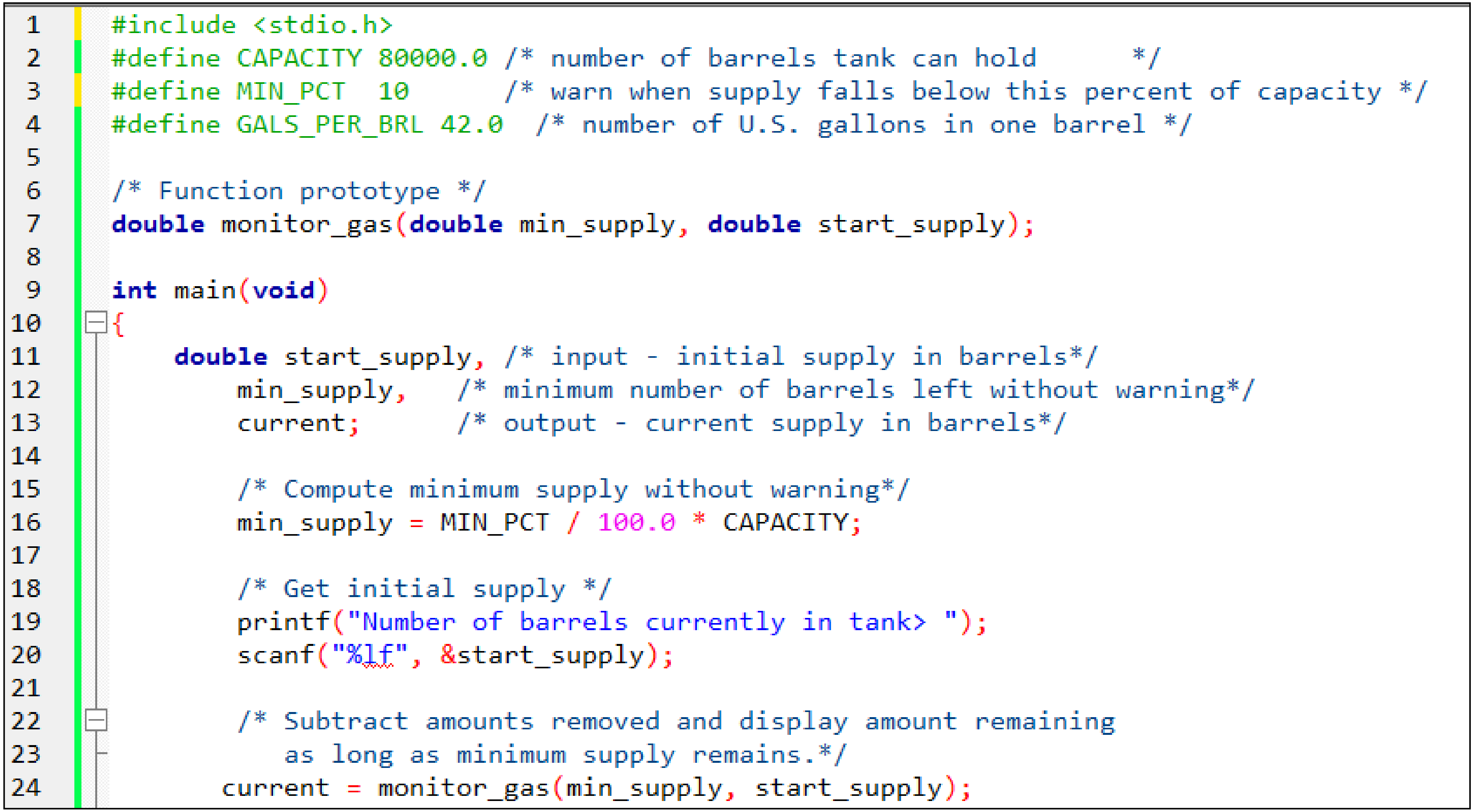
3 2

Conditional Loops

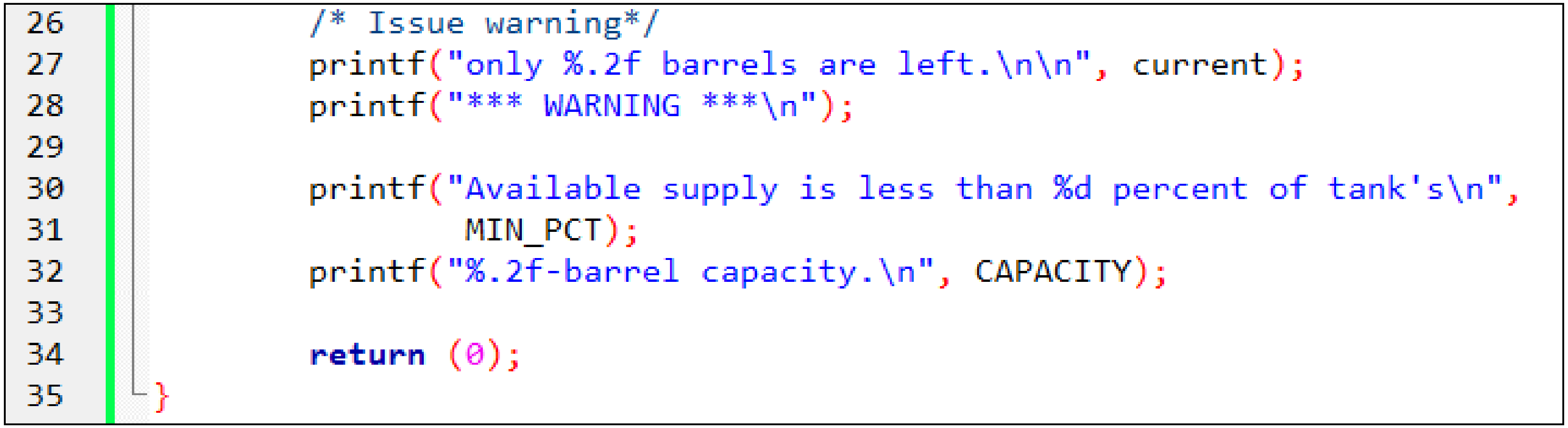
- used when there are programming conditions when you will not be able to determine the exact number of loop repetitions before loop execution begins
- Quick check with for loop:



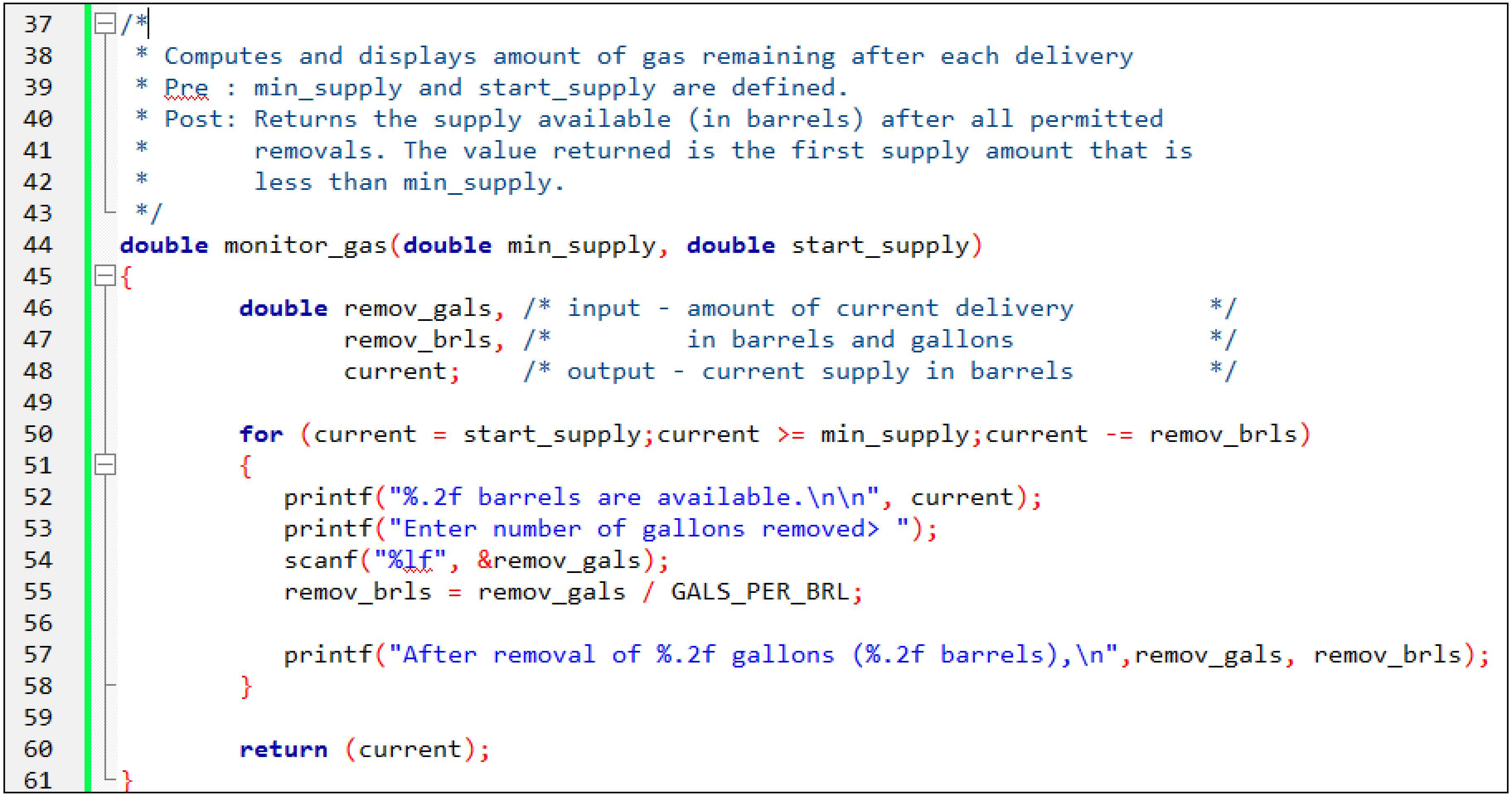
```
Number of barrels currently in tank> 8500.5
8500.50 barrels are available.

Enter number of gallons removed> 5859.0
After removal of 5859.00 gallons (139.50 barrels),
8361.00 barrels are available.

Enter number of gallons removed> 7568.4
After removal of 7568.40 gallons (180.20 barrels),
8180.80 barrels are available.

Enter number of gallons removed> 8400.0
After removal of 8400.00 gallons (200.00 barrels),
only 7980.80 barrels are left.

*** WARNING ***
Available supply is less than 10 percent of tank's
80000.00-barrel capacity.
```



```c
#include <stdio.h>
#define CAPACITY 80000.0 /* number of barrels tank can hold      */
#define MIN_PCT  10      /* warn when supply falls below this percent of capacity */
#define GALS_PER_BRL 42.0  /* number of U.S. gallons in one barrel */

/* Function prototype */
double monitor_gas(double min_supply, double start_supply);

int main(void)
{
    double start_supply, /* input - initial supply in barrels*/
        min_supply,   /* minimum number of barrels left without warning*/
        current;      /* output - current supply in barrels*/

        /* Compute minimum supply without warning*/
        min_supply = MIN_PCT / 100.0 * CAPACITY;

        /* Get initial supply */
        printf("Number of barrels currently in tank> ");
        scanf("%lf", &start_supply);

        /* Subtract amounts removed and display amount remaining
           as long as minimum supply remains.*/
       current = monitor_gas(min_supply, start_supply);
```



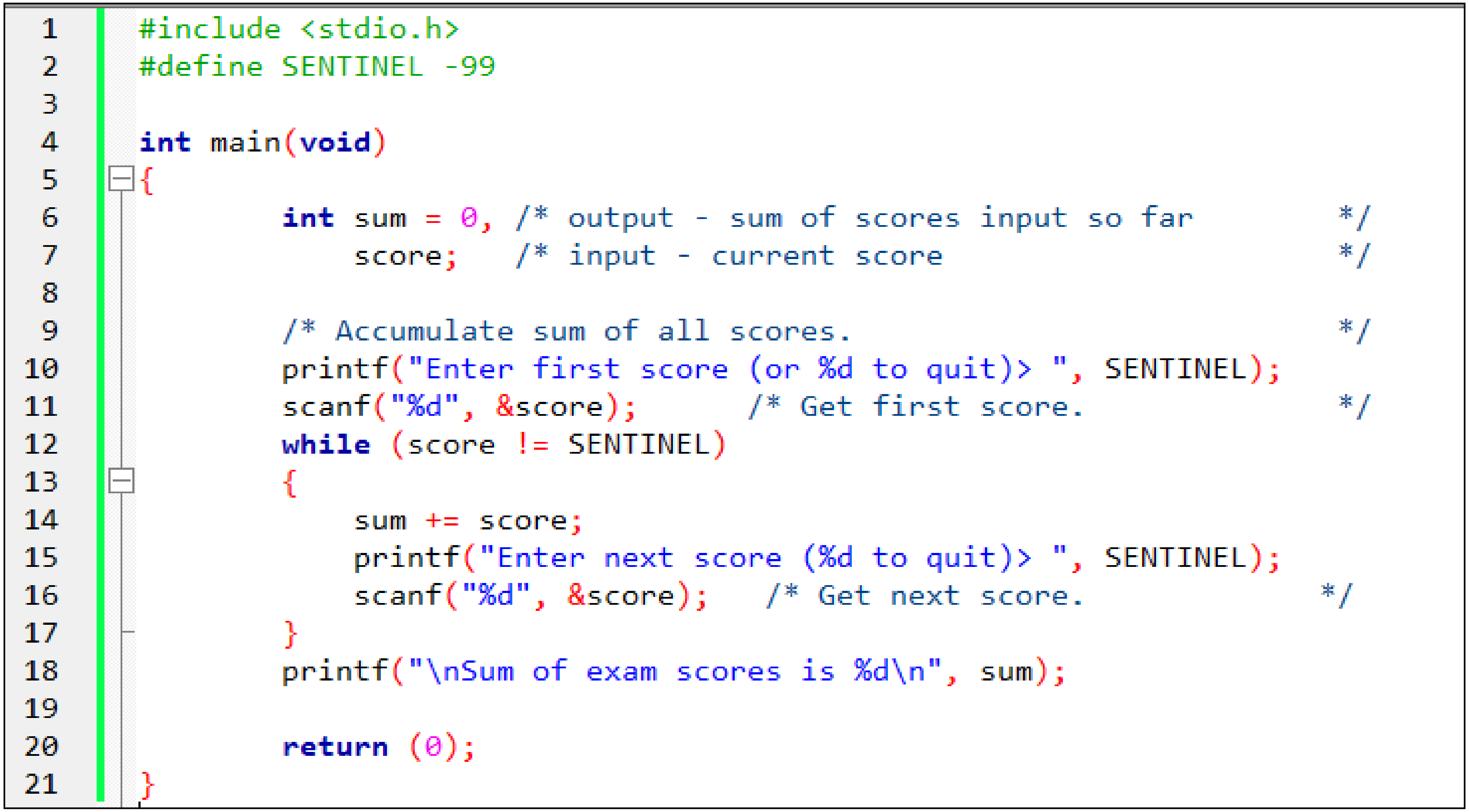
```
        /* Issue warning*/
        printf("only %.2f barrels are left.\n\n", current);
        printf("*** WARNING ***\n");

        printf("Available supply is less than %d percent of tank's\n",
                MIN_PCT);
        printf("%.2f-barrel capacity.\n", CAPACITY);

        return (0);
}
```



```c
/*
 * Computes and displays amount of gas remaining after each delivery
 * Pre : min_supply and start_supply are defined.
 * Post: Returns the supply available (in barrels) after all permitted
 *       removals. The value returned is the first supply amount that is
 *       less than min_supply.
 */
double monitor_gas(double min_supply, double start_supply)
{
        double remov_gals, /* input - amount of current delivery         */
               remov_brls, /*         in barrels and gallons             */
               current;    /* output - current supply in barrels         */

        for (current = start_supply;current >= min_supply;current -= remov_brls)
        {
           printf("%.2f barrels are available.\n\n", current);
           printf("Enter number of gallons removed> ");
           scanf("%lf", &remov_gals);
           remov_brls = remov_gals / GALS_PER_BRL;

           printf("After removal of %.2f gallons (%.2f barrels),\n",remov_gals, remov_brls);
        }

        return (current);
}
```

Loop Design

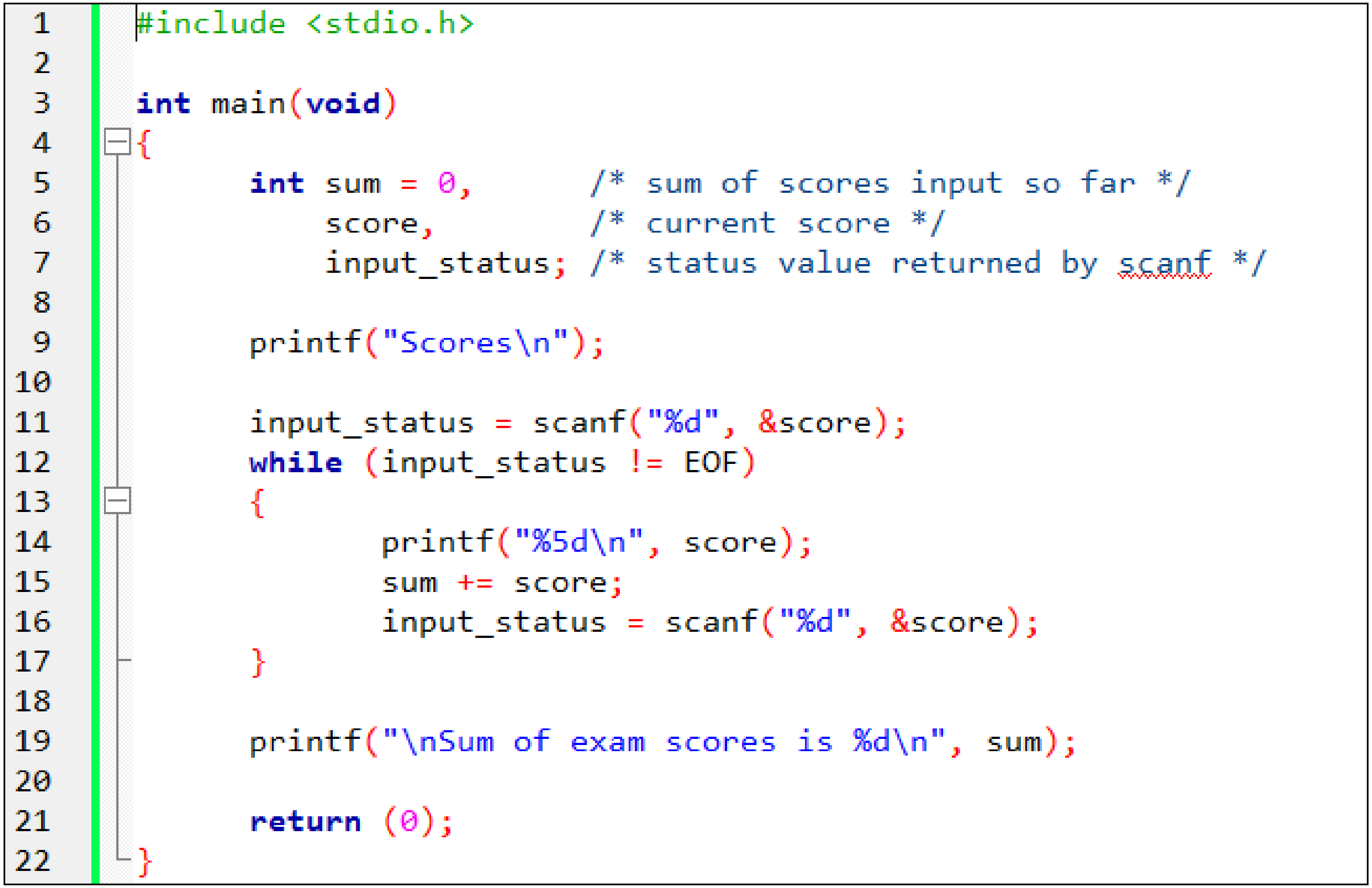
- Sentinel-Controlled Loops
  - sentinel value: an end marker that follows the last item in a list of data
- Endfile-Controlled Loops
- Infinite Loops on Faulty Data



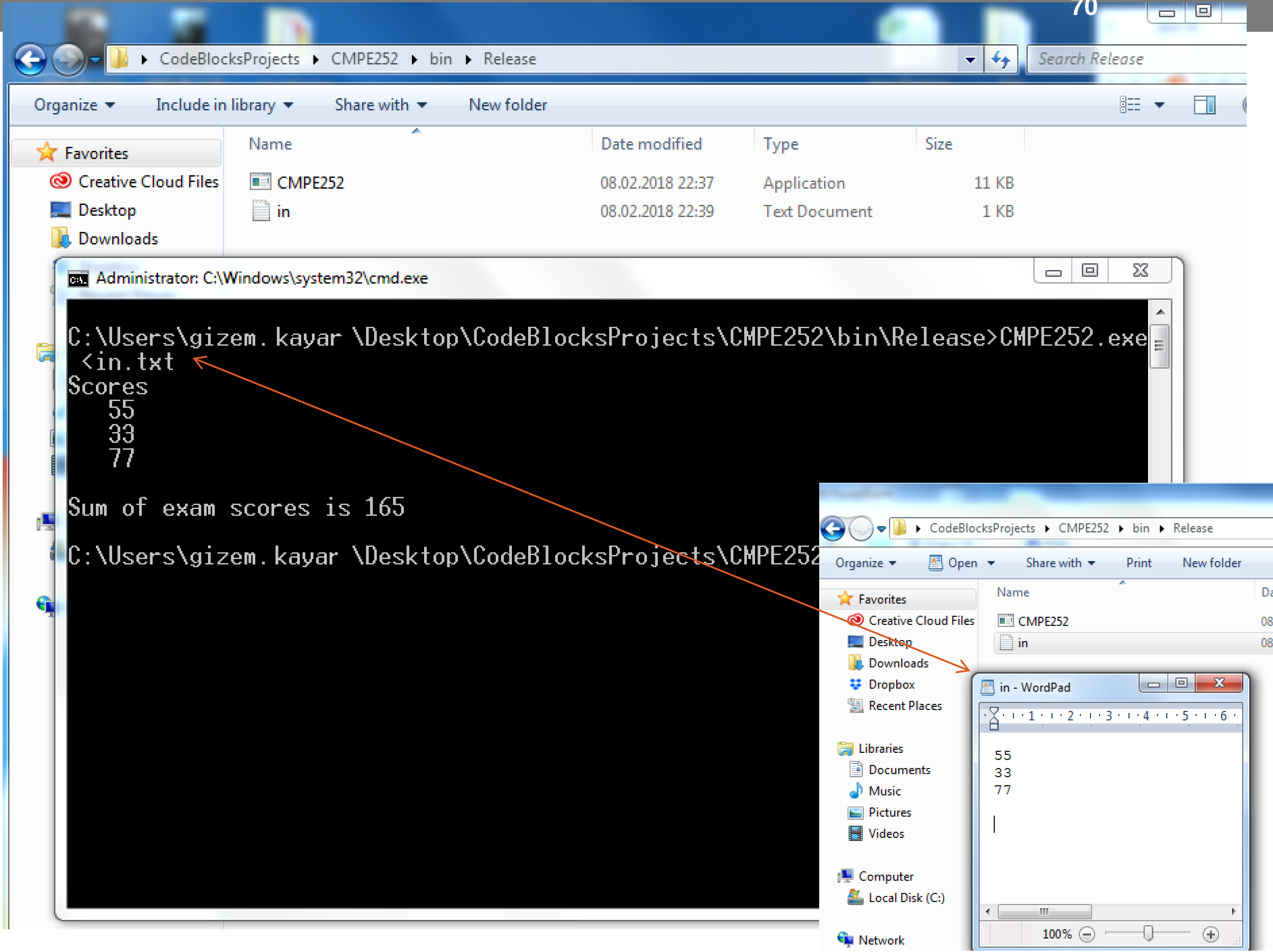
```c
#include <stdio.h>
#define SENTINEL -99

int main(void)
{
        int sum = 0, /* output - sum of scores input so far         */
            score;   /* input - current score                       */

        /* Accumulate sum of all scores.                           */
        printf("Enter first score (or %d to quit)> ", SENTINEL);
        scanf("%d", &score);      /* Get first score.              */
        while (score != SENTINEL)
        {
            sum += score;
            printf("Enter next score (%d to quit)> ", SENTINEL);
            scanf("%d", &score);   /* Get next score.             */
        }
        printf("\nSum of exam scores is %d\n", sum);

        return (0);
}
```

Endfile-Controlled Loop Design

1. Get the first *data value* and save *input status*
2. while *input status* does not indicate that end of file has been reached
    3. Process *data value*
    4. Get next *data value* and save *input status*



```c
#include <stdio.h>

int main(void)
{
      int sum = 0,      /* sum of scores input so far */
          score,        /* current score */
          input_status; /* status value returned by scanf */

      printf("Scores\n");

      input_status = scanf("%d", &score);
      while (input_status != EOF)
      {
             printf("%5d\n", score);
             sum += score;
             input_status = scanf("%d", &score);
      }

      printf("\nSum of exam scores is %d\n", sum);

      return (0);
}
```



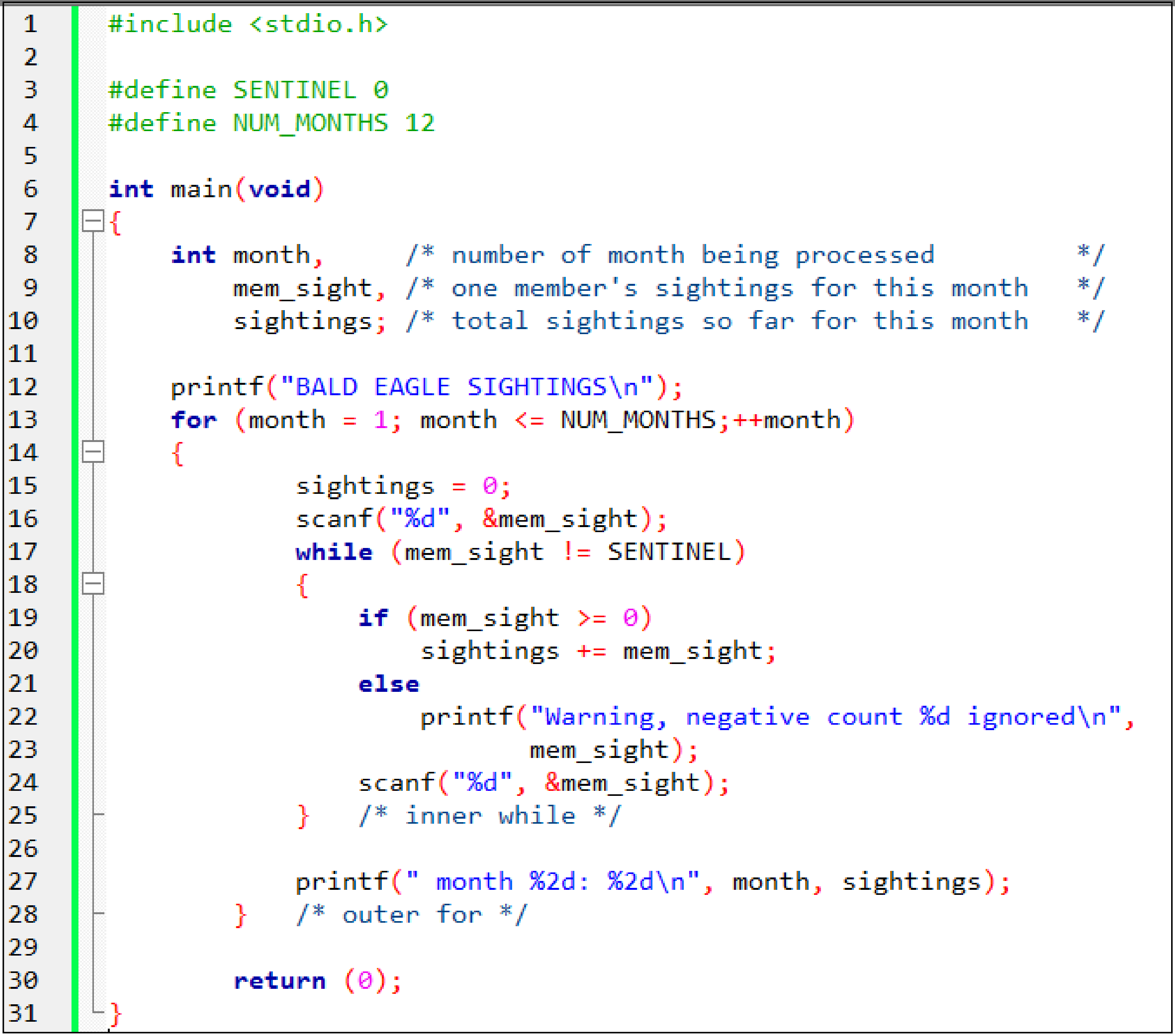
Name	Date modified	Type	Size
CMPE252	08.02.2018 22:37	Application	11 KB
in	08.02.2018 22:39	Text Document	1 KB

```
C:\Users\gizem.kayar \Desktop\CodeBlocksProjects\CMPE252\bin\Release>CMPE252.exe
 <in.txt
Scores
   55
   33
   77

Sum of exam scores is 165

C:\Users\gizem.kayar \Desktop\CodeBlocksProjects\CMPE252
```

```
55
33
77
```

Nested Loops

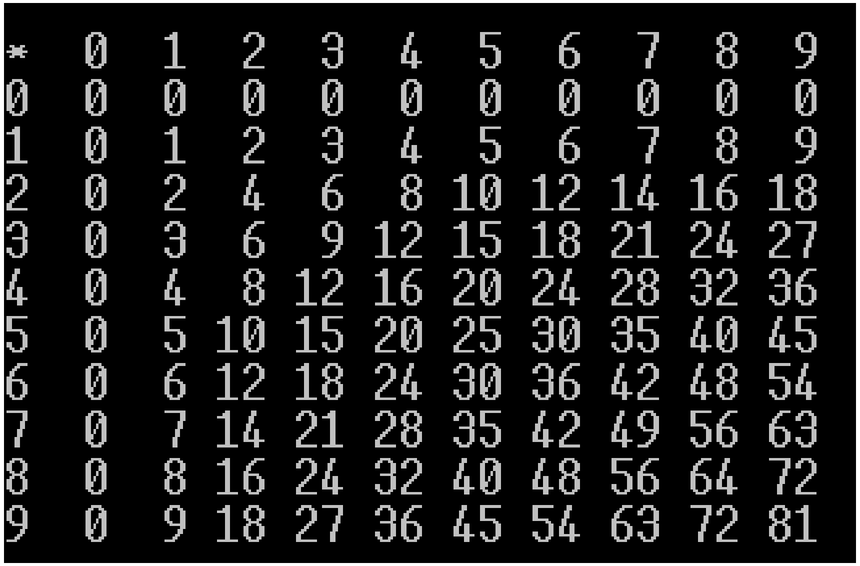
- Loops may be nested just like other control structures
- Nested loops consist of an outer loop with one or more inner loops
- Each time the outer loop is repeated, the inner loops are reentered, their loop control expressions are reevaluated, and all required iterations are performed



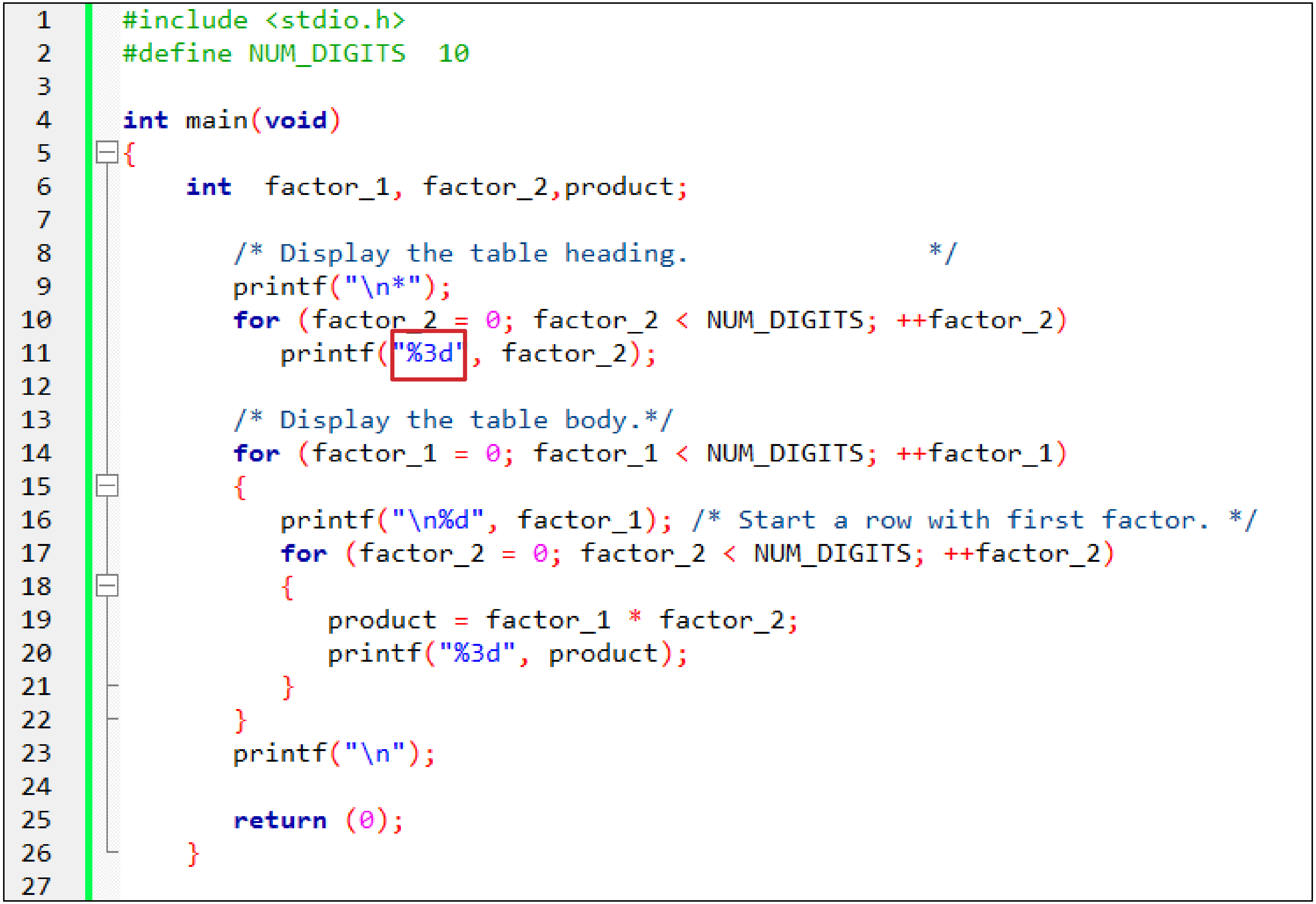
```c
#include <stdio.h>

#define SENTINEL 0
#define NUM_MONTHS 12

int main(void)
{
    int month,     /* number of month being processed          */
        mem_sight, /* one member's sightings for this month    */
        sightings; /* total sightings so far for this month    */

    printf("BALD EAGLE SIGHTINGS\n");
    for (month = 1; month <= NUM_MONTHS;++month)
    {
            sightings = 0;
            scanf("%d", &mem_sight);
            while (mem_sight != SENTINEL)
            {
                if (mem_sight >= 0)
                    sightings += mem_sight;
                else
                    printf("Warning, negative count %d ignored\n",
                           mem_sight);
                scanf("%d", &mem_sight);
            }   /* inner while */

            printf(" month %2d: %2d\n", month, sightings);
        }   /* outer for */

        return (0);
}
```

Quick Check

- Write a program that gives the following output (multiplication table from 0 to 9):



```
*   0   1   2   3   4   5   6   7   8   9
0   0   0   0   0   0   0   0   0   0   0
1   0   1   2   3   4   5   6   7   8   9
2   0   2   4   6   8  10  12  14  16  18
3   0   3   6   9  12  15  18  21  24  27
4   0   4   8  12  16  20  24  28  32  36
5   0   5  10  15  20  25  30  35  40  45
6   0   6  12  18  24  30  36  42  48  54
7   0   7  14  21  28  35  42  49  56  63
8   0   8  16  24  32  40  48  56  64  72
9   0   9  18  27  36  45  54  63  72  81
```



```
#include <stdio.h>
#define NUM_DIGITS  10

int main(void)
{
    int  factor_1, factor_2,product;

       /* Display the table heading.               */
       printf("\n*");
       for (factor_2 = 0; factor_2 < NUM_DIGITS; ++factor_2)
          printf("%3d", factor_2);

       /* Display the table body.*/
       for (factor_1 = 0; factor_1 < NUM_DIGITS; ++factor_1)
       {
          printf("\n%d", factor_1); /* Start a row with first factor. */
          for (factor_2 = 0; factor_2 < NUM_DIGITS; ++factor_2)
          {
             product = factor_1 * factor_2;
             printf("%3d", product);
          }
       }
       printf("\n");

       return (0);
    }

```

do-while Statement

- For conditions where we know that a loop must execute at least one time

1. Get a *data value*
2. If *data value* isn’t in the acceptable range, go back to step 1.

do-while Syntax

```
do
        statement;
while (loop repetition condition);

/* Find first even number input */
do
        status = scanf(“%d”, &num);
while (status > 0  &&  (num % 2)  !=  0);
```

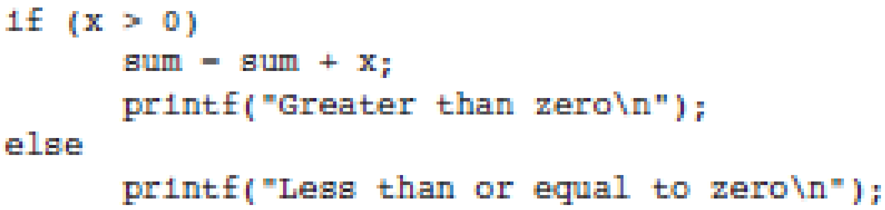
Common Errors

- if (0 <= x <= 4)
  - printf("Condition is true\n");

- For example, let’s consider the case when x is 5 .
- The value of 0 <= 5 is 1 , and 1 is certainly less than or equal to 4!
- In order to check if x is in the range 0 to 4 , you should use the condition
  - (0 <= x && x <= 4)

- if (x = 10)
  - printf("x is 10");

What about this one ?



```
if (x > 0)
        sum = sum + x;
        printf("Greater than zero\n");
else
        printf("Less than or equal to zero\n");
```

Null Statement

- The null statement is merely a semicolon alone.
- ;,
- A null statement does not do anything. It does not store a value anywhere. It does not cause time to pass during the execution of your program.
- Most often, a null statement is used as the body of a loop statement, or as one or more of the expressions in a for statement. Here is an example of a for statement that uses the null statement as the body of the loop (and also calculates the integer square root of n, just for fun):
- for (i = 1; i*i < n; i++)
- ;
- Here is another example that uses the null statement as the body of a for loop and also produces output:
- for (x = 1; x <= 5; printf ("x is now %d\n", x), x++)
- ;
- A null statement is also sometimes used to follow a label that would otherwise be the last thing in a block.

«break» in loops

- You can use the **break** statement to terminate a *while*, *do*, *for*, statement. Here is an example:

```
int x;
for (x = 1; x <= 10; x++)
 {
   if (x == 8)
     break;
   else
     printf ("%d ", x);
 }
```

«continue» in loops

- You can use the **continue** statement in loops **to terminate an iteration of the loop and begin the next iteration**. Here is an example:

```
for (x = 0; x < 100; x++)
 {
   if (x % 2 == 0)
     continue;

     sum_of_odd_numbers + = x;
    }
```

- If you put a continue statement inside a loop which itself is inside a loop, then it affects only the innermost loop.

RECURSION
CHAPTER 9

Problem Solving & Program Design in C

Eighth Edition
Global Edition

Jeri R. Hanly & Elliot B. Koffman

Recursion

- A recursive function is one that calls itself or that is part of a cycle in the sequence of function calls.

- The ability to invoke itself enables a recursive function to be repeated with different parameter values.

Recursion

- It can be used an an alternative to iteration - looping.
- Recursion is typically used to specify a natural, simple solution that would otherwise be very difficult to solve.

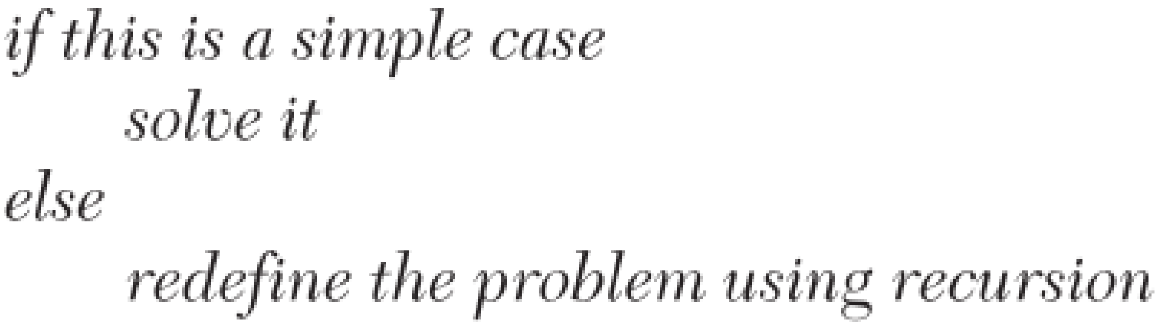
The Nature of Recursion

- One or more simple cases of the problem have a straightforward, nonrecursive solution.

- The other cases can be redefined in terms of problems that are closer to the simple cases.

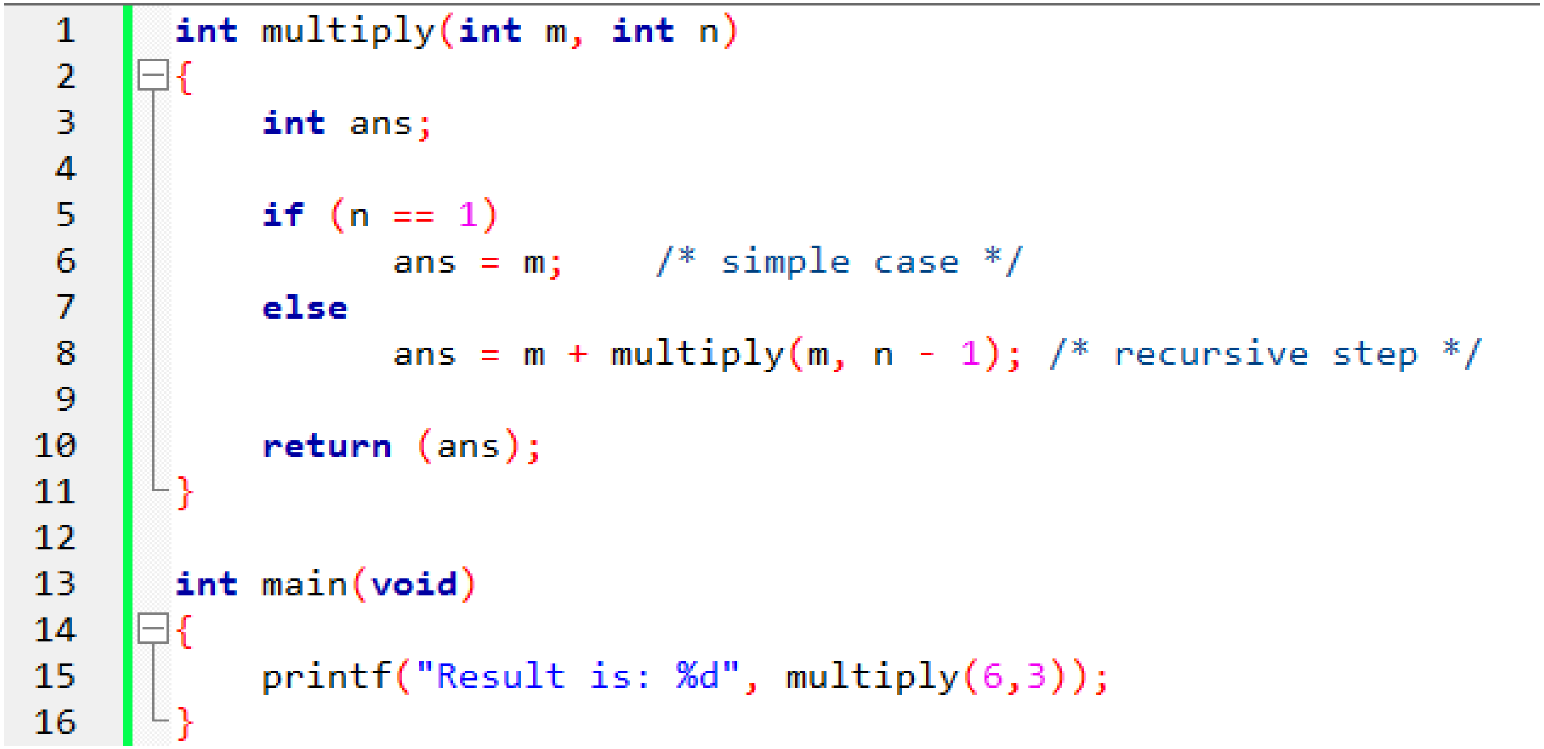
The Nature of Recursion

- By applying this redefinition process every time the recursive function is called, eventually the problem is reduced entirely to simple cases, which are relatively easy to solve.



```
if this is a simple case
      solve it
else
      redefine the problem using recursion
```

Quick Check – Multiply: x*y



```c
int multiply(int m, int n)
{
    int ans;

    if (n == 1)
          ans = m;    /* simple case */
    else
          ans = m + multiply(m, n - 1); /* recursive step */

    return (ans);
}

int main(void)
{
    printf("Result is: %d", multiply(6,3));
}
```

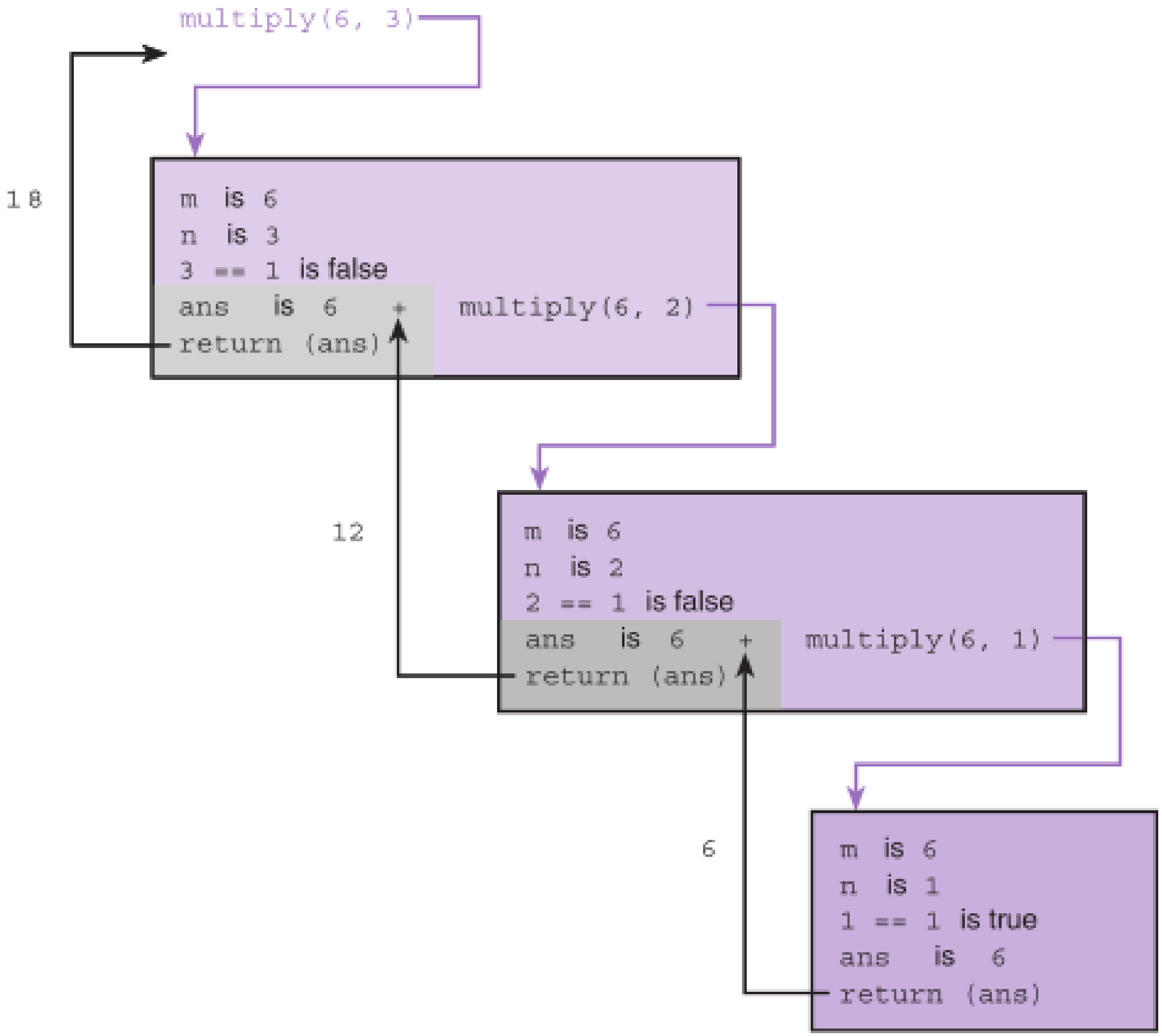
SPLIT:

- Multiply 6 by 2
- Add 6 to the result

- Multiply 6 by 1
- Add 6 to the result

Tracing Recursive Functions

- activation frame
  - representation of one call to a function
- terminating condition
  - a condition that is true when a recursive algorithm is processing a simple case
- system stack
  - area of memory where parameters and local variables are allocated when a function is called and deallocated when the function returns


multiply(6, 3)
18
m is 6
n is 3
3 == 1 is false
ans is 6 +
return (ans)
multiply(6, 2)
12
m is 6
n is 2
2 == 1 is false
ans is 6 +
return (ans)
multiply(6, 1)
6
m is 6
n is 1
1 == 1 is true
ans is 6
return (ans)

Quick Check

- Write a recursive function which computes the sum of its two integer parameters.

```
int add(int m, int n)
{
        int ans;
        if (n == 0)
                ans = m;
        else
                ans = 1 + add(m, n-1);

        return (ans);
}
```

Parameter and Local Variable Stacks

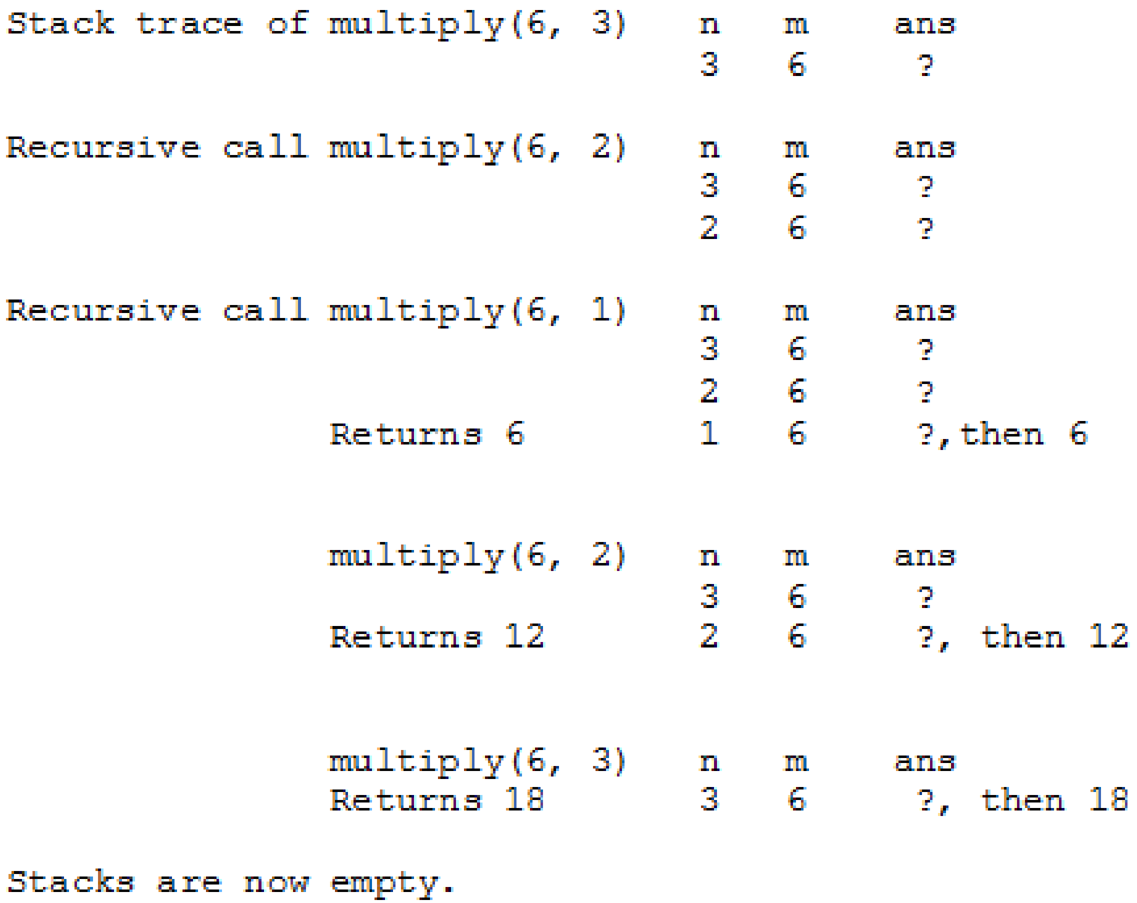
- stack
  - a data structure in which the last data item added is the first data item processed

- C keeps track of the values of variables from different recursive function calls by using a stack data structure.

Implementation of Parameter Stacks in C

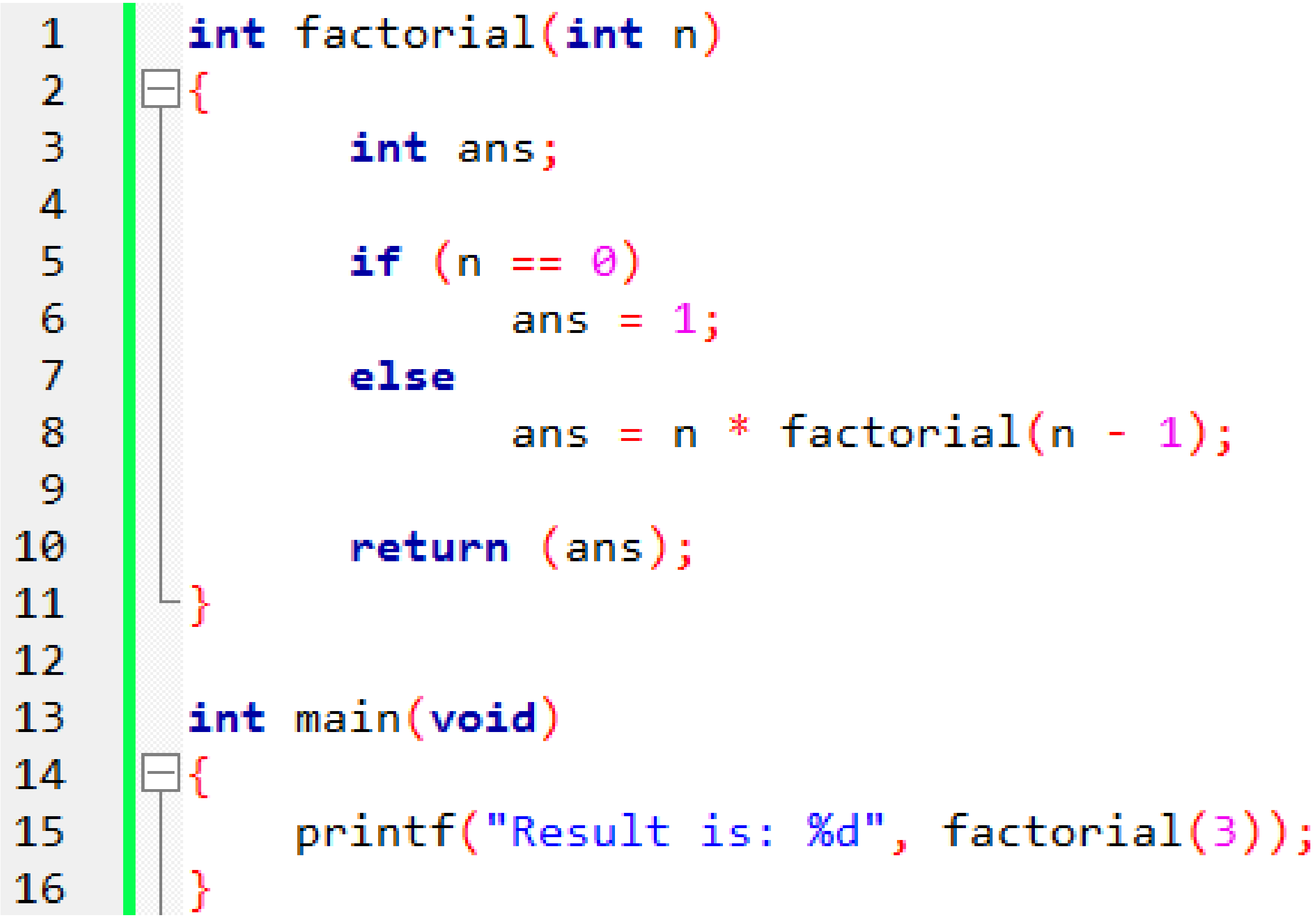
- The compiler actually maintains a single system stack for the tasks.

- system stack
  - area or memory where parameters and local variables are allocated when a function is called and deallocated when the function returns



```
Stack trace of multiply(6, 3)    n    m     ans
                                 3    6      ?

Recursive call multiply(6, 2)    n    m     ans
                                 3    6      ?
                                 2    6      ?

Recursive call multiply(6, 1)    n    m     ans
                                 3    6      ?
                                 2    6      ?
               Returns 6         1    6      ?,then 6


               multiply(6, 2)    n    m     ans
                                 3    6      ?
               Returns 12        2    6      ?, then 12


               multiply(6, 3)    n    m     ans
               Returns 18        3    6      ?, then 18

Stacks are now empty.
```

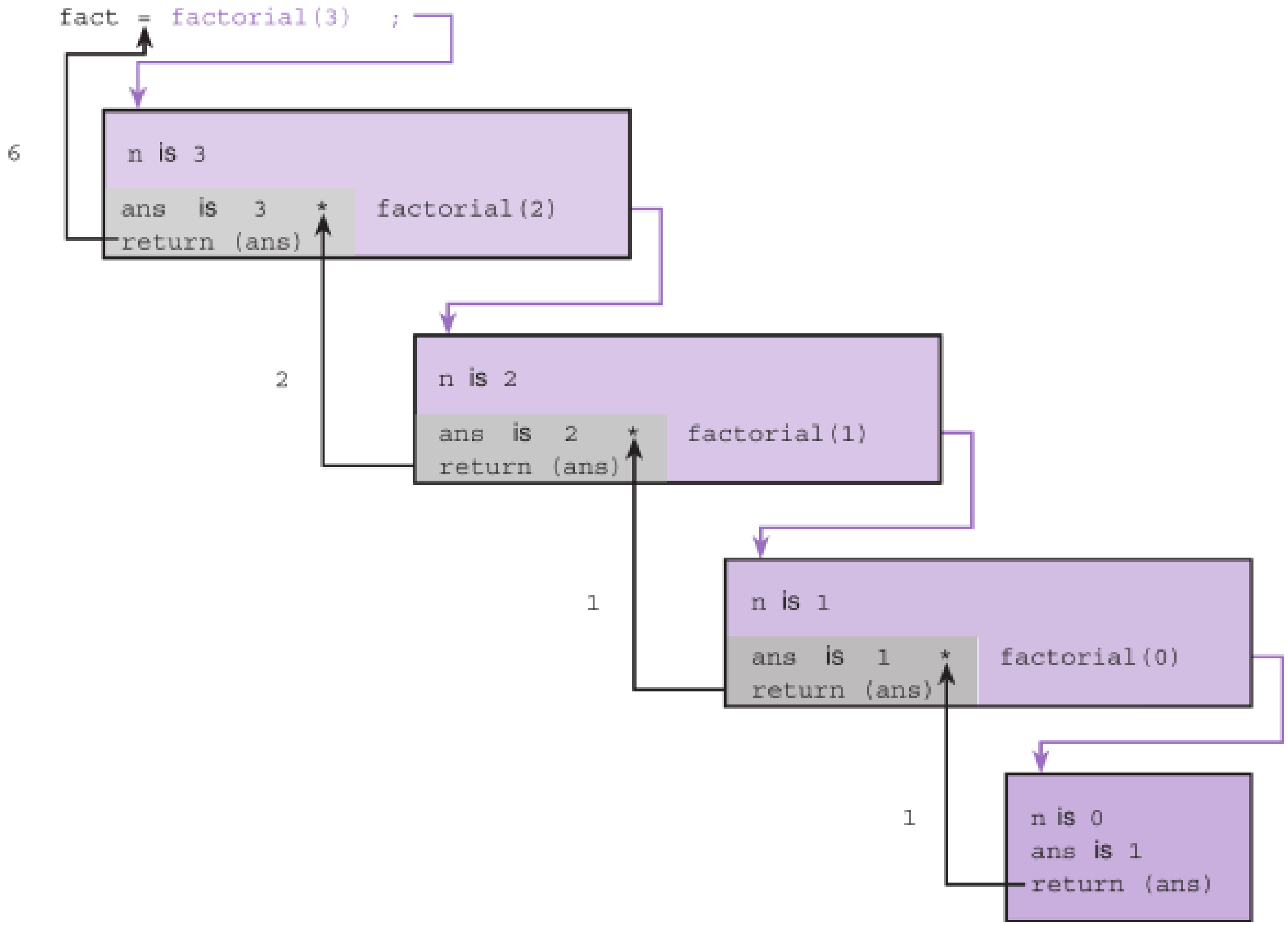
Quick Check – Recursive Factorial



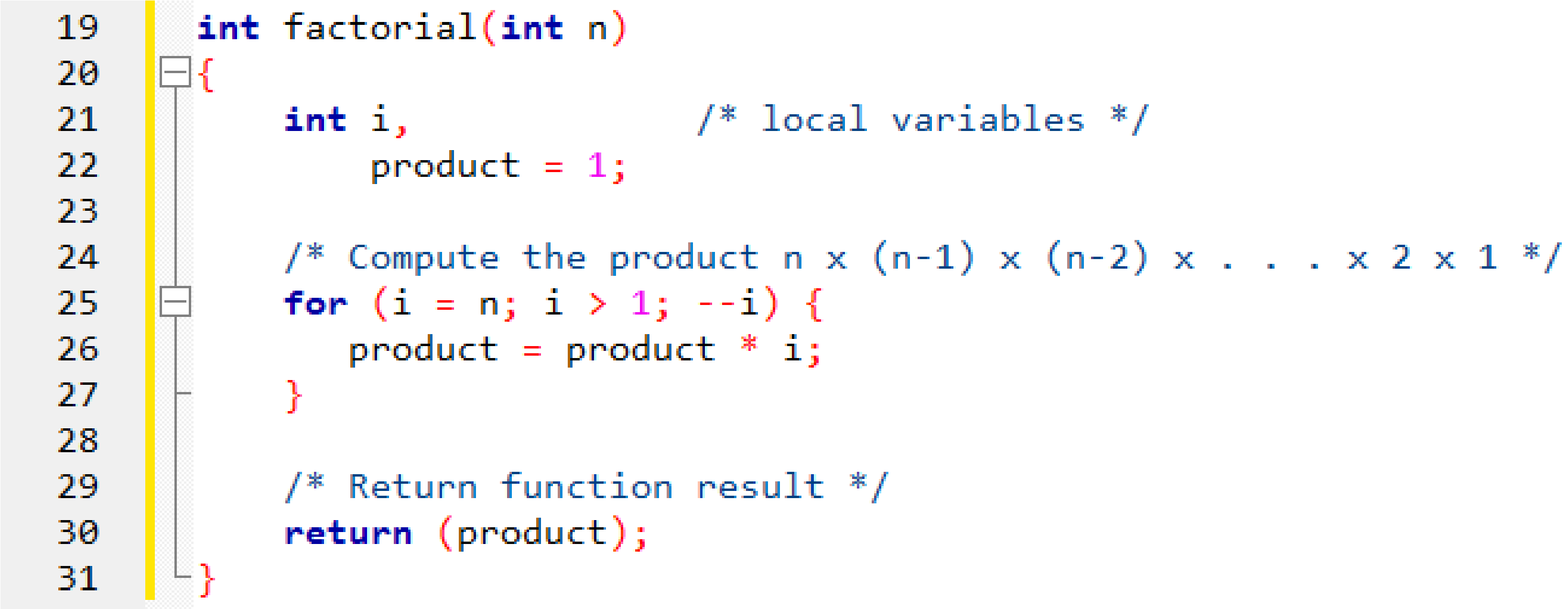
```
int factorial(int n)
{
      int ans;

      if (n == 0)
            ans = 1;
      else
            ans = n * factorial(n - 1);

      return (ans);
}

int main(void)
{
    printf("Result is: %d", factorial(3));
}
```

FIGURE 9.11

Trace of fact = factorial(3);



Iterative Factorial



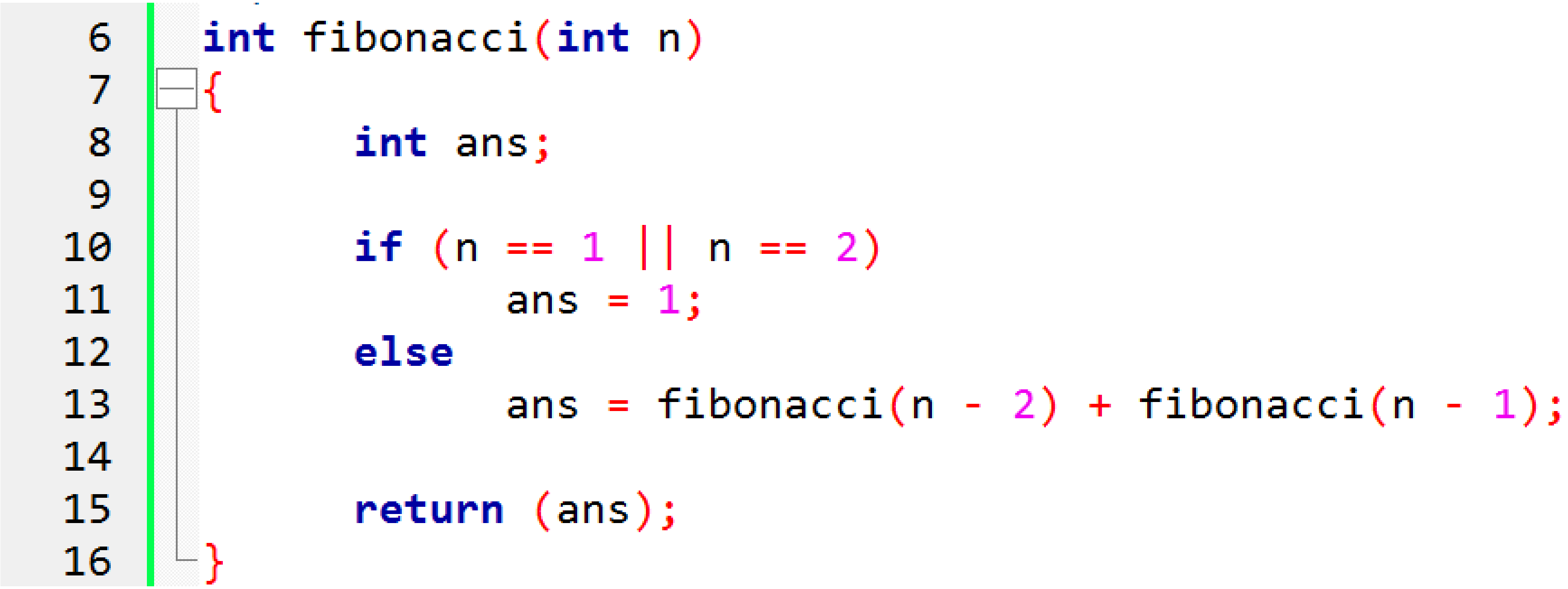
```
int factorial(int n)
{
    int i,                 /* local variables */
        product = 1;

    /* Compute the product n x (n-1) x (n-2) x . . . x 2 x 1 */
    for (i = n; i > 1; --i) {
       product = product * i;
    }

    /* Return function result */
    return (product);
}
```

Fibonacci

- Sequence is 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

- Fibonacci_1 is 1
- Fibonacci_2 is 1
- Fibonacci_n is Fibonacci_{n-2} + Fibonacci_{n-1} for $n > 2$

Quick Check: Write Fibonacci function in a recursive way



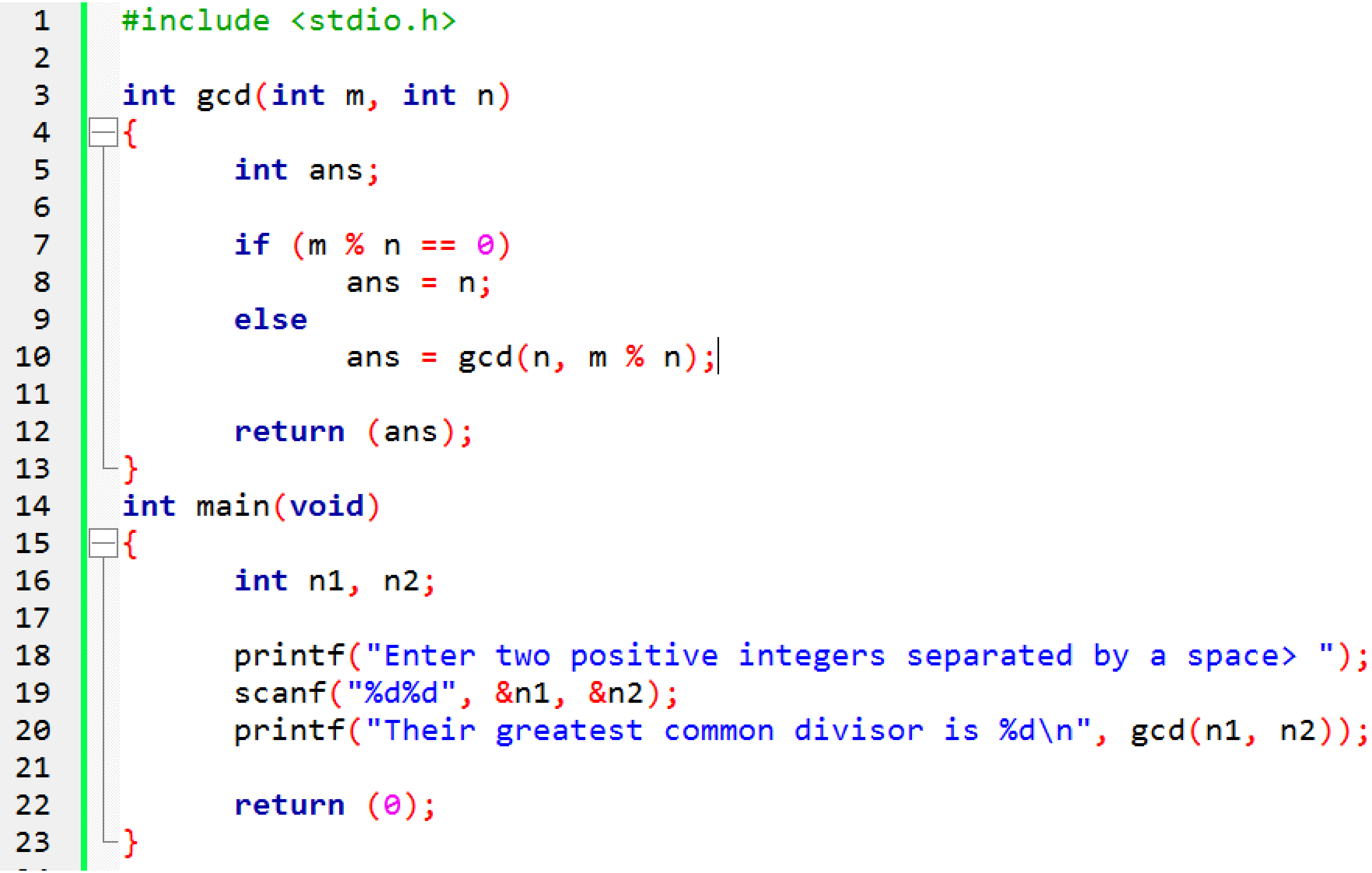
```
int fibonacci(int n)
{
      int ans;

      if (n == 1 || n == 2)
            ans = 1;
      else
            ans = fibonacci(n - 2) + fibonacci(n - 1);

      return (ans);
}
```

GCD

- greatest common divisor of two integers is the largest integer that divides them both evenly

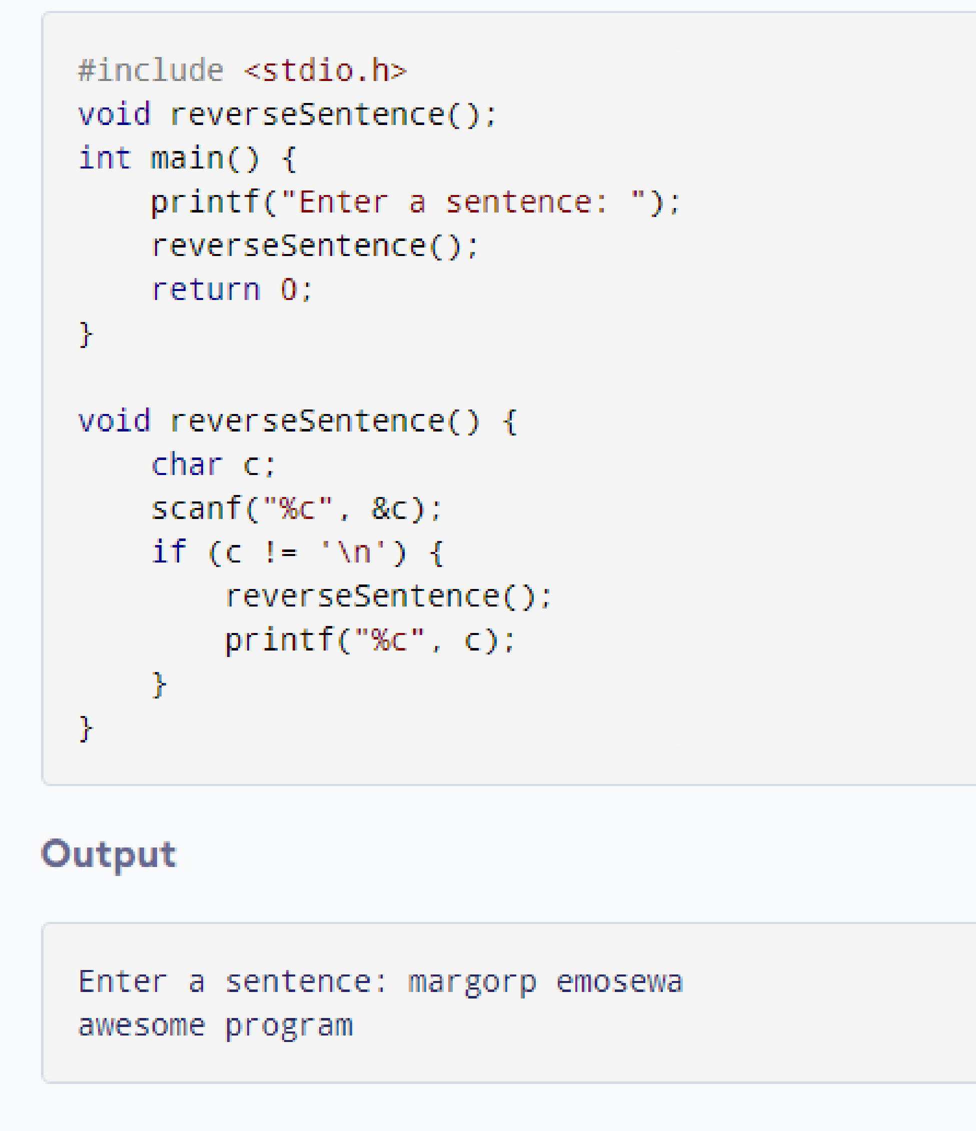
Quick Check: Write GCD function in a recursive way

- gcd(*m*,*n*) is *n* if *n* divides m evenly
- gcd(*m*,*n*) is gcd(*n*, remainder of *m* divided by *n*) otherwise



```c
#include <stdio.h>

int gcd(int m, int n)
{
      int ans;

      if (m % n == 0)
            ans = n;
      else
            ans = gcd(n, m % n);

      return (ans);
}
int main(void)
{
      int n1, n2;

      printf("Enter two positive integers separated by a space> ");
      scanf("%d%d", &n1, &n2);
      printf("Their greatest common divisor is %d\n", gcd(n1, n2));

      return (0);
}
```

Example: Reverse a sentence from the user command prompt using recursion



```
#include <stdio.h>
void reverseSentence();
int main() {
    printf("Enter a sentence: ");
    reverseSentence();
    return 0;
}

void reverseSentence() {
    char c;
    scanf("%c", &c);
    if (c != '\n') {
        reverseSentence();
        printf("%c", c);
    }
}
```

Output

```
Enter a sentence: margorp emosewa
awesome program
```

Pro’s and Con’s

- CONS:
  - **Recursion uses more memory** : function is added to the stack at each recursive call and keep the local variables until the call is finished.
  - **Recursion can be slow :** since it contains function calls . (iteration is more efficient)

- PROS:
  - **For certain problems, they are easier to solve using Recursion and Recursion enables less complex, more clear and understandable code** *(for instance, binary search problems, tree traversals etc.)*

Everything, written using Recursion can be implemented using Iteration and vice versa

References

1. Problem Solving & Program Design in C, Jeri R. Hanly & Elliot B. Koffman, Pearson 8. Edition, Global Edition