C Programming Language

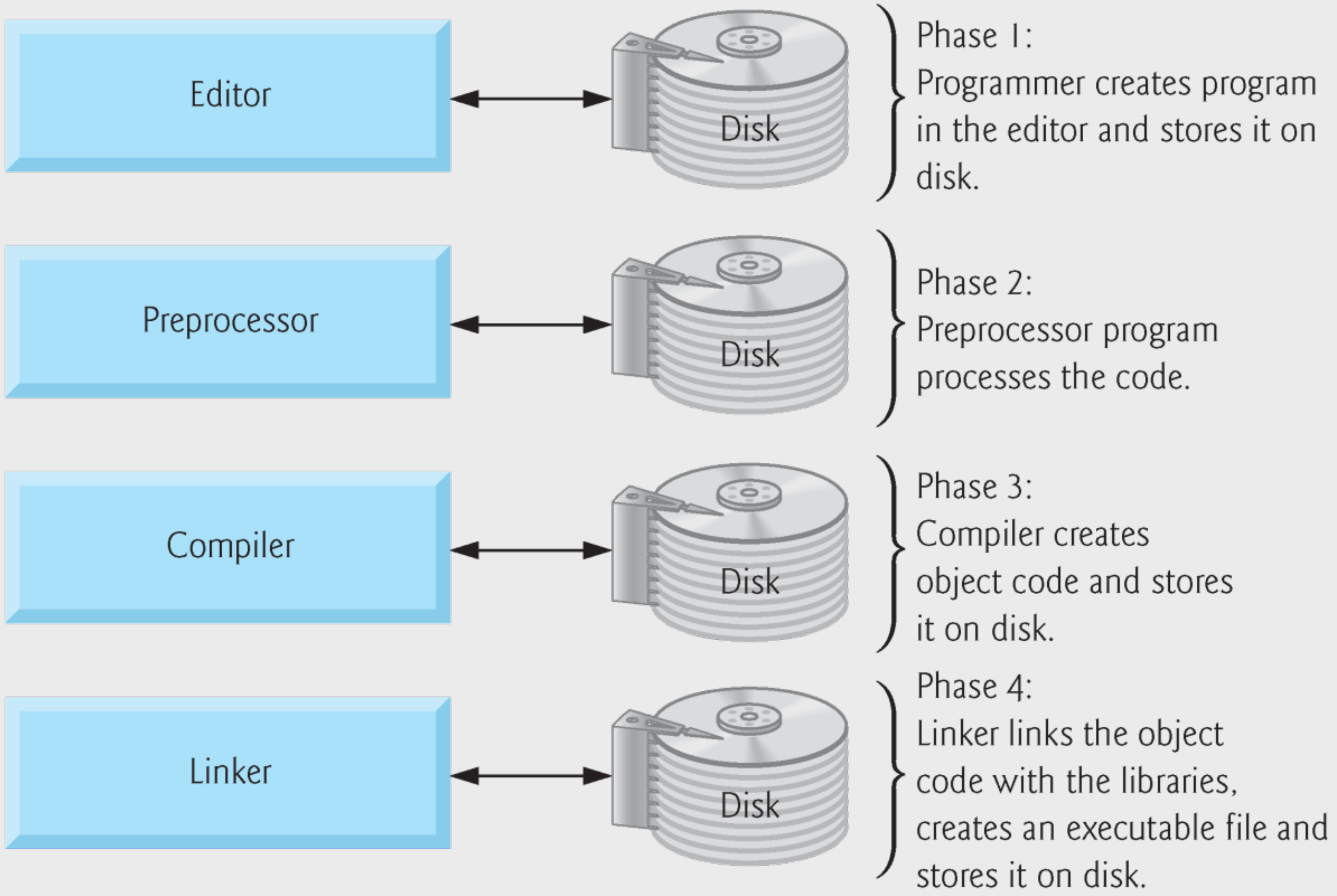
C Revisited

Typical C Program Development Environment

- C systems generally consist of several parts: a program development environment, the language and the C Standard Library.
- C programs typically go through six phases
  - edit, preprocess, compile, link, load and execute.
- Phase 1 consists of editing a file.
  - This is accomplished with an editor program.
  - C program file names should end with the .c extension.

Typical C Program Development Environment (Cont.)

- Phase 2, you give the command to compile the program.
  - In a C system, a preprocessor program executes automatically before the compiler's translation phase begins.
  - The C preprocessor obeys special commands called preprocessor directives.
- Phase 3, the compiler translates the C program into machine-language code.



Editor

Disk

Phase 1:
Programmer creates program in the editor and stores it on disk.

Preprocessor

Disk

Phase 2:
Preprocessor program processes the code.

Compiler

Disk

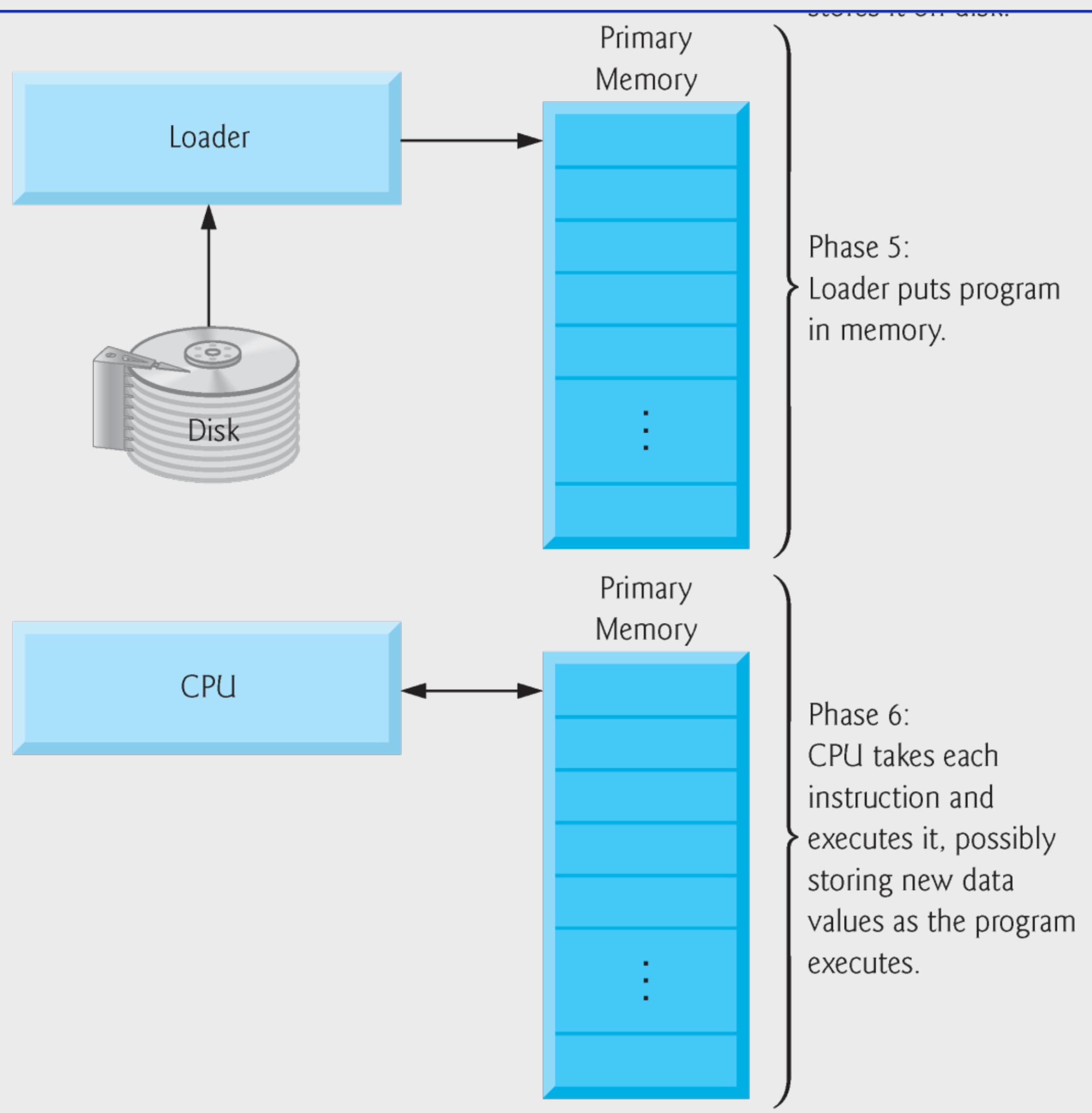
Phase 3:
Compiler creates object code and stores it on disk.

Linker

Disk

Phase 4:
Linker links the object code with the libraries, creates an executable file and stores it on disk.

Fig. 1.1 | Typical C development environment. (Part 1 of 2.)



Phase 5:
Loader puts program in memory.

Phase 6:
CPU takes each instruction and executes it, possibly storing new data values as the program executes.

Fig. 1.1 | Typical C development environment. (Part 2 of 2.)

Typical C Program Development Environment (Cont.)

- Phase 4 is called linking.
  - A linker links the object code with the code for the missing functions to produce an executable file (with no missing pieces).
- On a typical Linux system, the command to compile and link a program is called `cc` (or `gcc`).
- To compile and link a program named welcome.c type
  - `gcc welcome.c`
- If the program compiles and links correctly, a file called a.out is produced.

Typical C Program Development Environment (Cont.)

- Phase 5 is called loading.
  - Before a program can be executed, the program must first be placed in memory.
  - This is done by the loader, which takes the executable image from disk and transfers it to memory.
  - Additional components from shared libraries that support the program are also loaded.
  - Finally, the computer, under the control of its CPU, executes the program one instruction at a time.
  - To load and execute the program on a Linux system, type `./a.out` at the Linux prompt and press Enter.

Typical C Program Development Environment (Cont.)

- Most C programs input and/or output data.
- Certain C functions take their input from `stdin` (the standard input stream), which is normally the keyboard, but `stdin` can be connected to another stream.
- Data is often output to `stdout` (the standard output stream), which is normally the computer screen, but `stdout` can be connected to another stream.
- There is also a standard error stream referred to as `stderr`.
- The `stderr` stream (normally connected to the screen) is used for displaying error messages.

Elements of Program

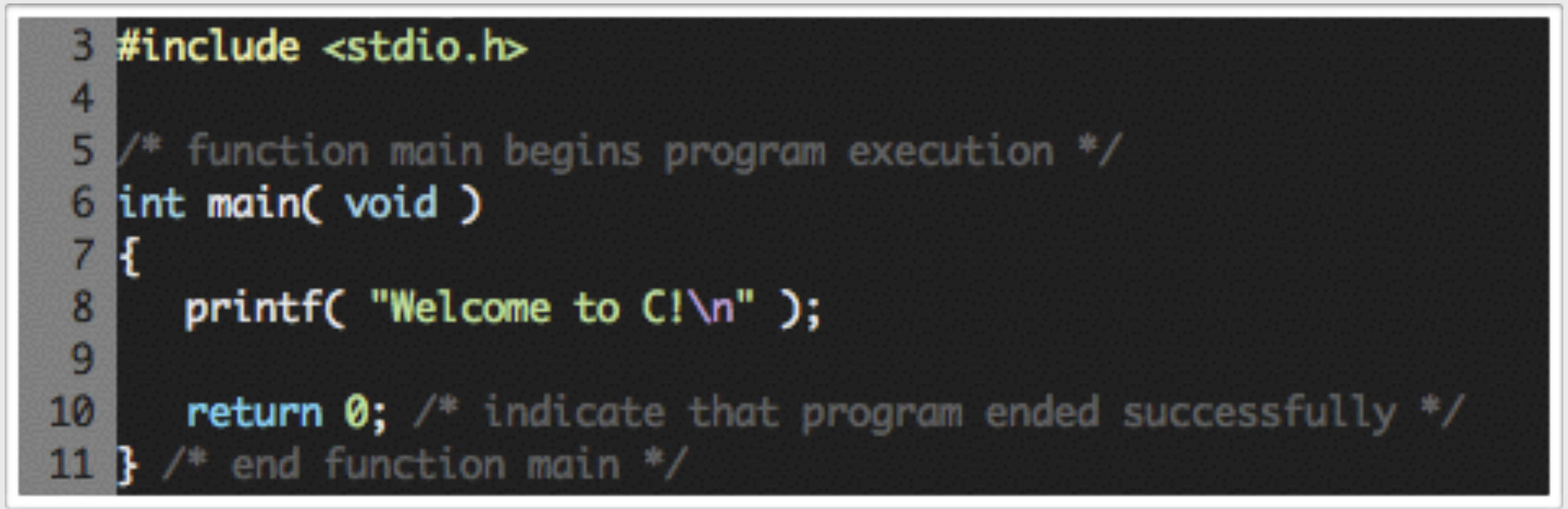
- Basic elements of a program
  - data declarations/definitions (variables)
  - instructions (functions)
  - comments

Basic Program Structure

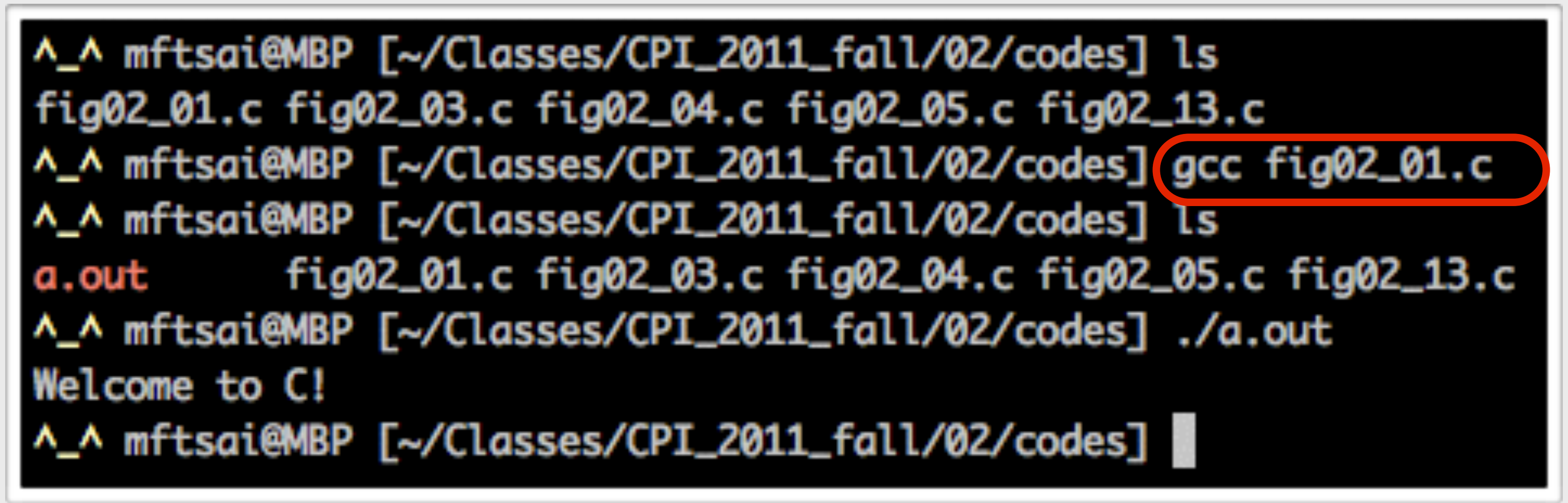
```
/************************************
**  Comments
*************************************/
... Data declarations ...

int main(){
  ... Executable statements ...
  return (0);
}
```

A Simple C Program: Printing a Line of Text

- Example: fig02_01.c



```
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   printf( "Welcome to C!\n" );

   return 0; /* indicate that program ended successfully */
} /* end function main */
```



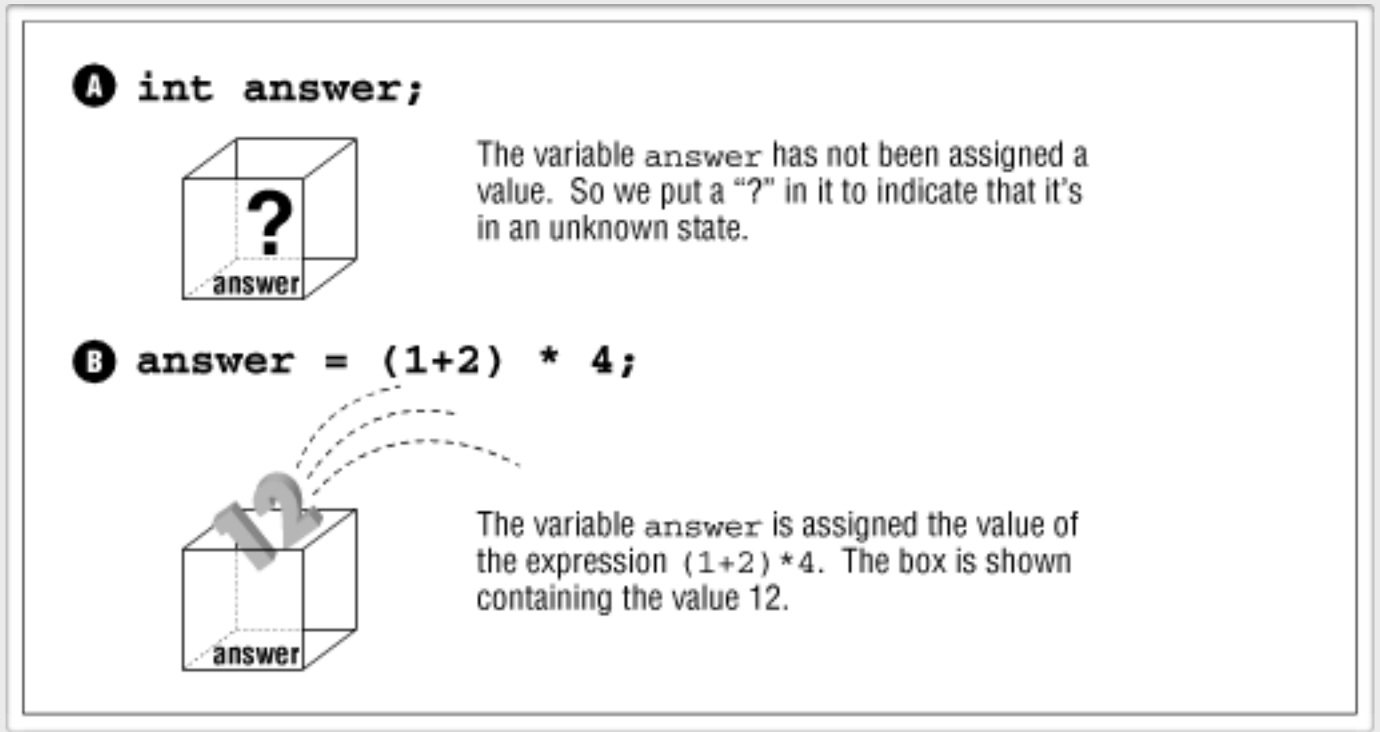
```
^_^ mftsai@MBP [~/Classes/CPI_2011_fall/02/codes] ls
fig02_01.c fig02_03.c fig02_04.c fig02_05.c fig02_13.c
^_^ mftsai@MBP [~/Classes/CPI_2011_fall/02/codes] gcc fig02_01.c
^_^ mftsai@MBP [~/Classes/CPI_2011_fall/02/codes] ls
a.out      fig02_01.c fig02_03.c fig02_04.c fig02_05.c fig02_13.c
^_^ mftsai@MBP [~/Classes/CPI_2011_fall/02/codes] ./a.out
Welcome to C!
^_^ mftsai@MBP [~/Classes/CPI_2011_fall/02/codes] 
```

Variables

- Invalid variables
  - 3rd_entity /* Begins with a number */
  - all$done /* Contains a “$” */
  - the end /* Contains a space */
  - int /* Reserved word */

Assignment Statements

- **`answer = (1+2) * 4;`**
  - "=" is not the meaning of equal
  - "=" is an assignment operator
  - The variable "answer" on the left side of the equal sign (=) is assigned the value on the right side.



A
```
int answer;
```

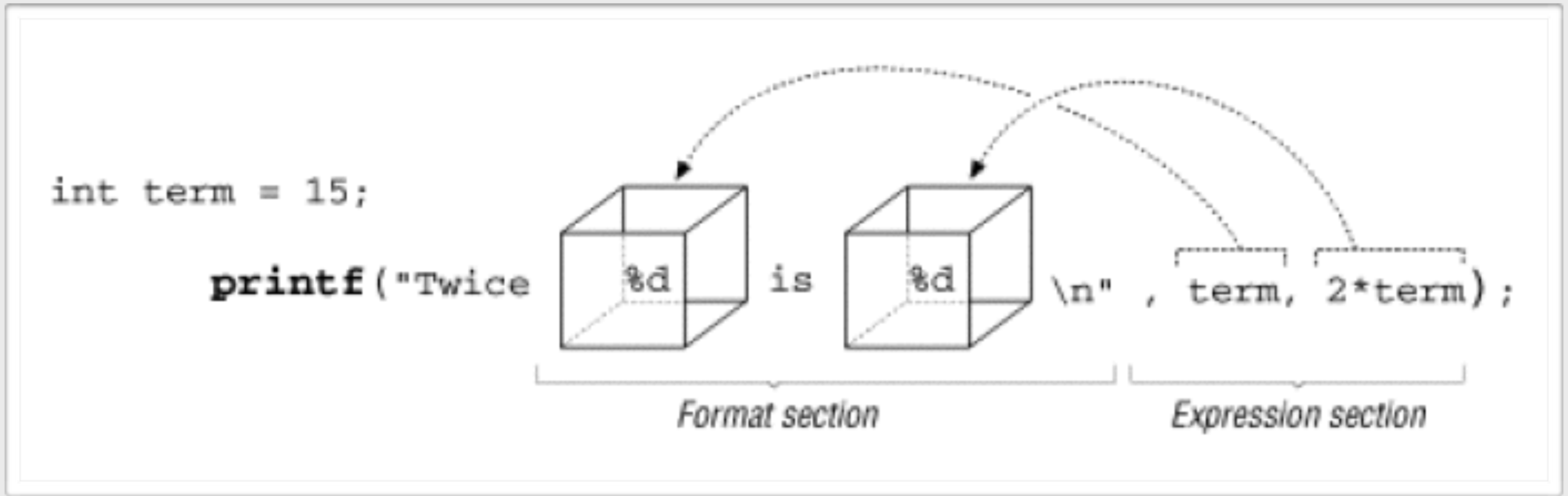
The variable answer has not been assigned a value. So we put a "?" in it to indicate that it's in an unknown state.

B
```
answer = (1+2) * 4;
```

The variable answer is assigned the value of the expression (1+2)*4. The box is shown containing the value 12.

`printf` Function

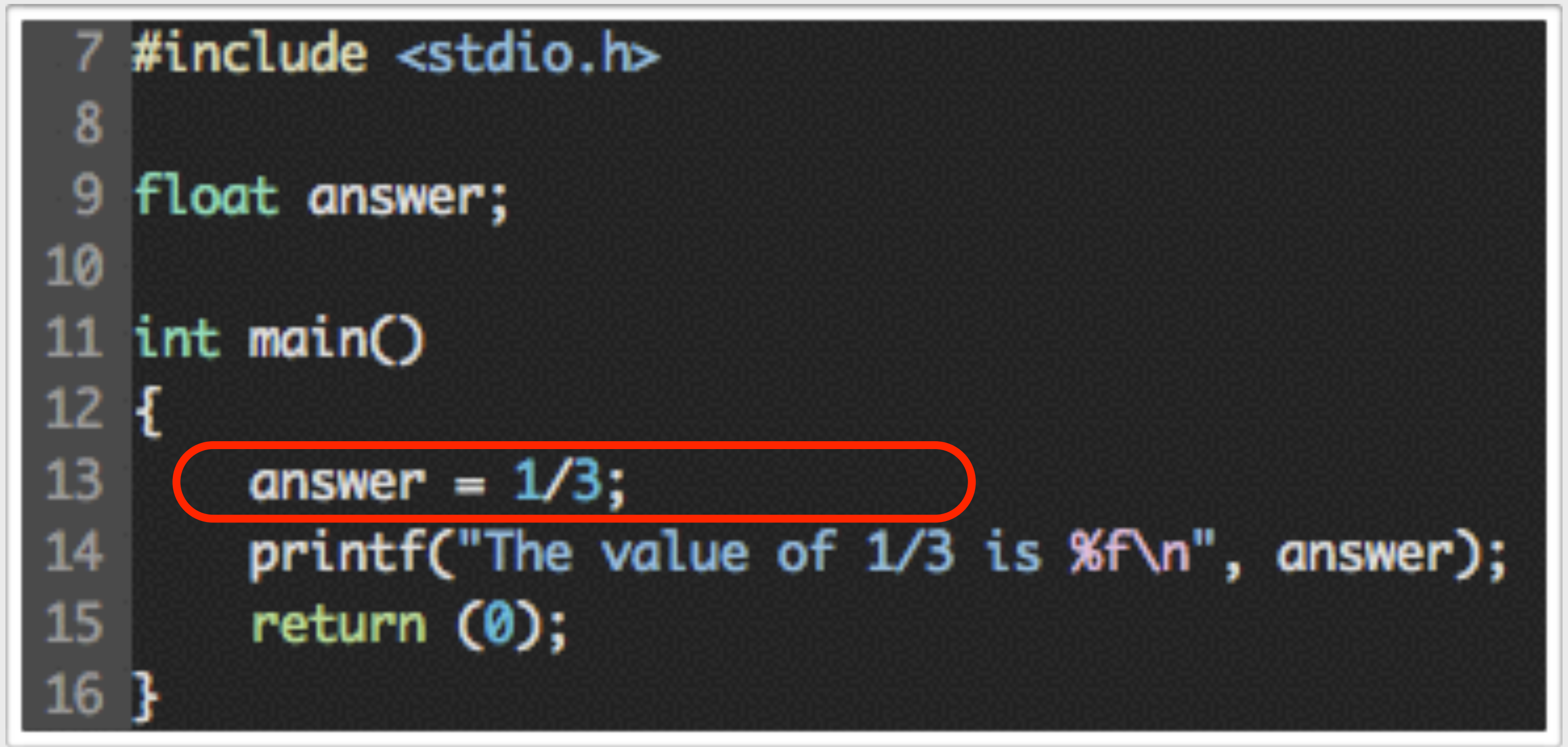
- **`printf(format, expression-1, expression-2, ...)`**
  - **`format`**: the string describing what to print
  - the value of **`expression-1`** is printed in place of the first "**`%d`**" in the format string
  - **`expression-2`** is printed in place of the second, and so on



```
int term = 15;
        printf("Twice %d is %d\n", term, 2*term);
```

Floating Point vs. Integer Divide

- Why is the result of the code 0.0?

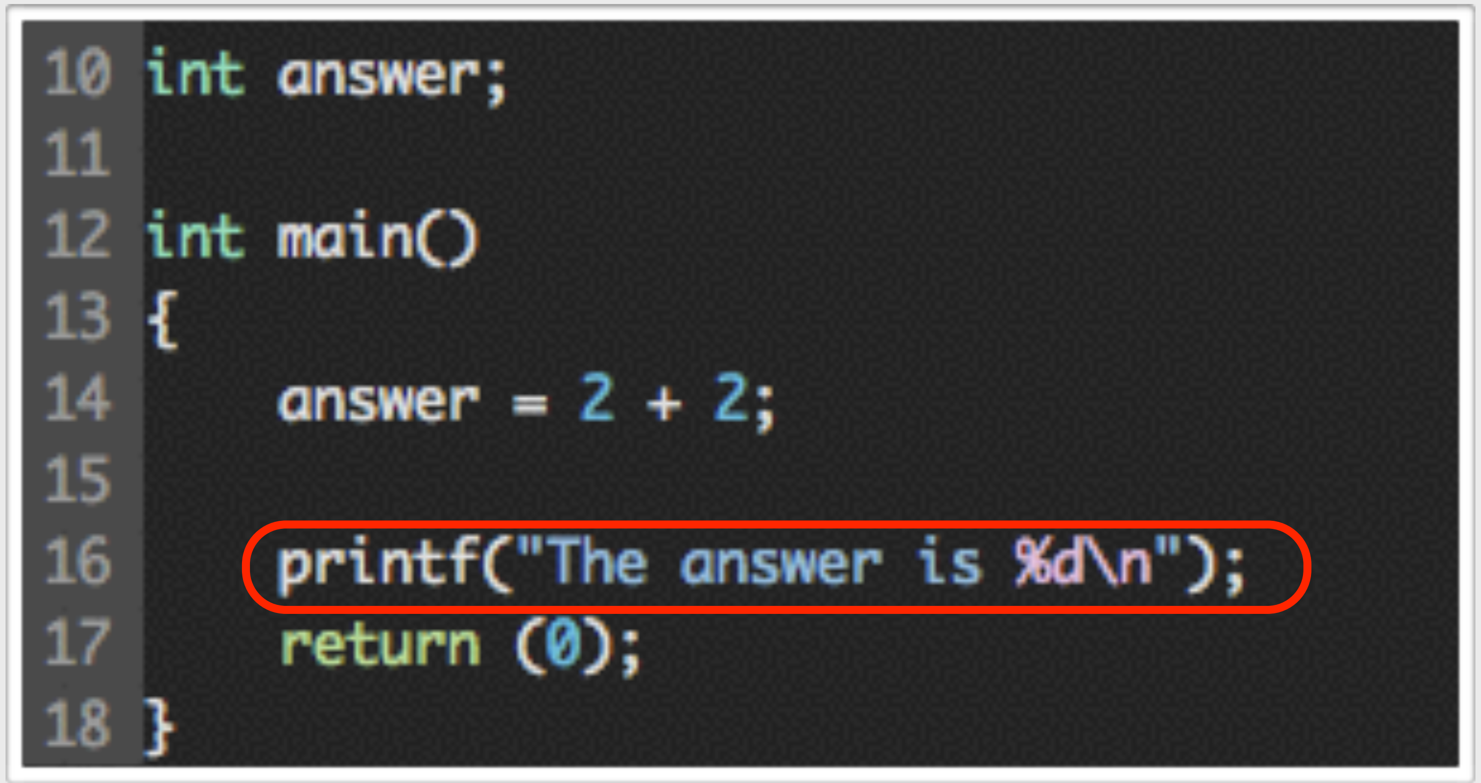


```
#include <stdio.h>

float answer;

int main()
{
    answer = 1/3;
    printf("The value of 1/3 is %f\n", answer);
    return (0);
}
```

```
answer = 1.0 / 3.0;
```

Floating Point vs. Integer Divide

- Why does 2+2 = 5928?

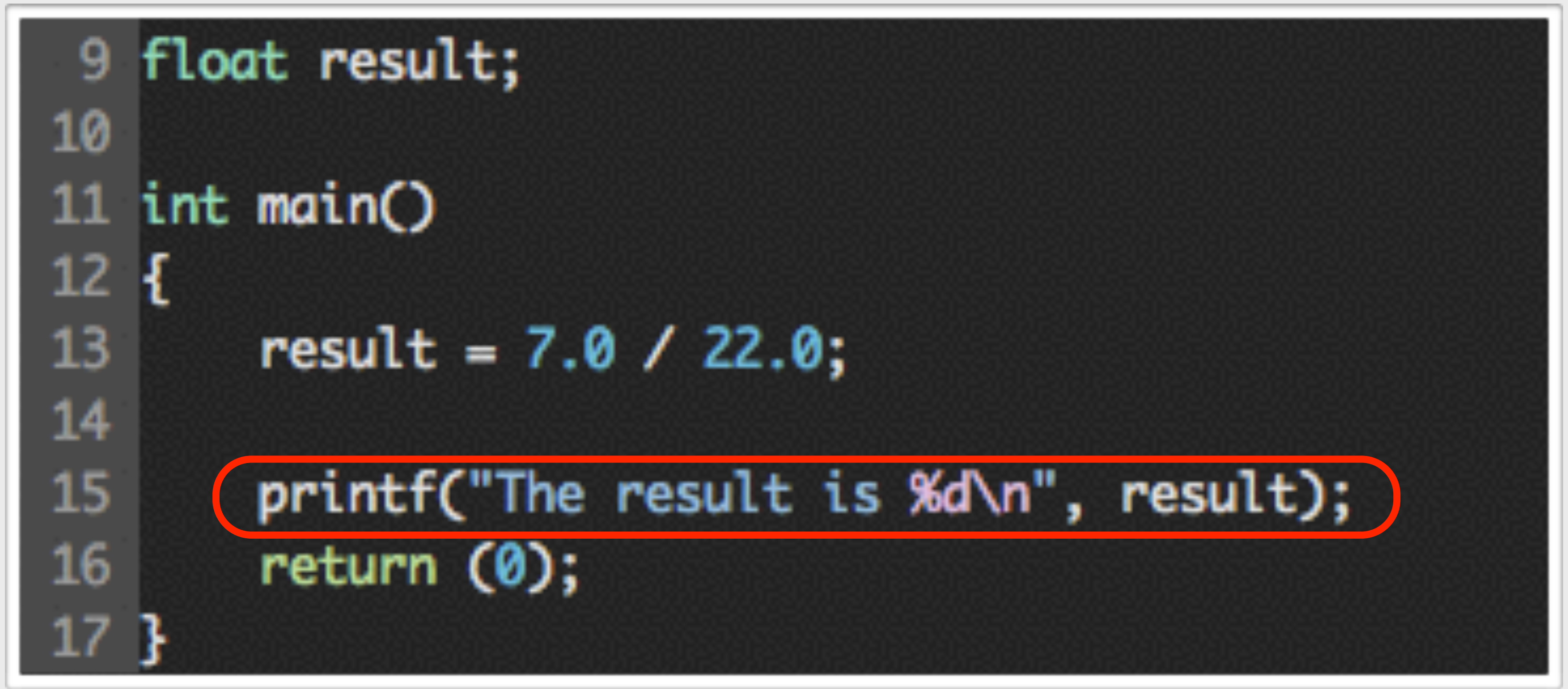


```
int answer;

int main()
{
	answer = 2 + 2;

	printf("The answer is %d\n");
	return (0);
}
```

```
printf("The answer is %d\n", answer);
```

Floating Point vs. Integer Divide

- Why does 7.0/22.0 = 1606412144?

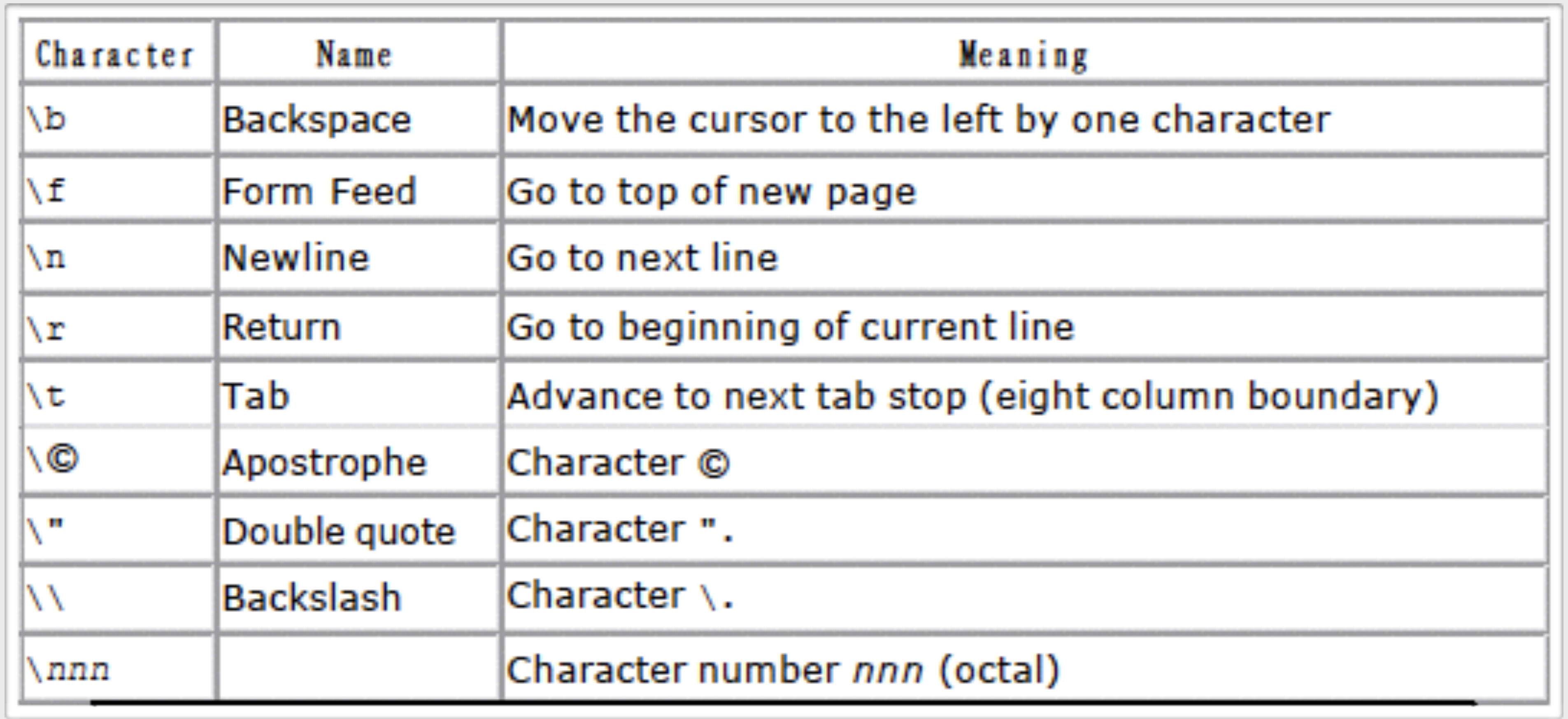


```
float result;

int main()
{
    result = 7.0 / 22.0;

    printf("The result is %d\n", result);
    return (0);
}
```

```
printf(“The result is %f\n”, result);
```

Characters

- Declaration
  `char variable; /* comment */`



Character	Name	Meaning
\b	Backspace	Move the cursor to the left by one character
\f	Form Feed	Go to top of new page
\n	Newline	Go to next line
\r	Return	Go to beginning of current line
\t	Tab	Advance to next tab stop (eight column boundary)
\©	Apostrophe	Character ©
\"	Double quote	Character ".
\\	Backslash	Character \.
nnn		Character number *nnn* (octal)

Arrays

- `int data_list[10];`
  - An array is a set of consecutive memory locations used to store data.
  - Each item in the array is called an element.

Strings

- Strings are sequences of characters.
- In C, strings are created out of character arrays.
- **‘\0’** is used to indicate the end of a string.

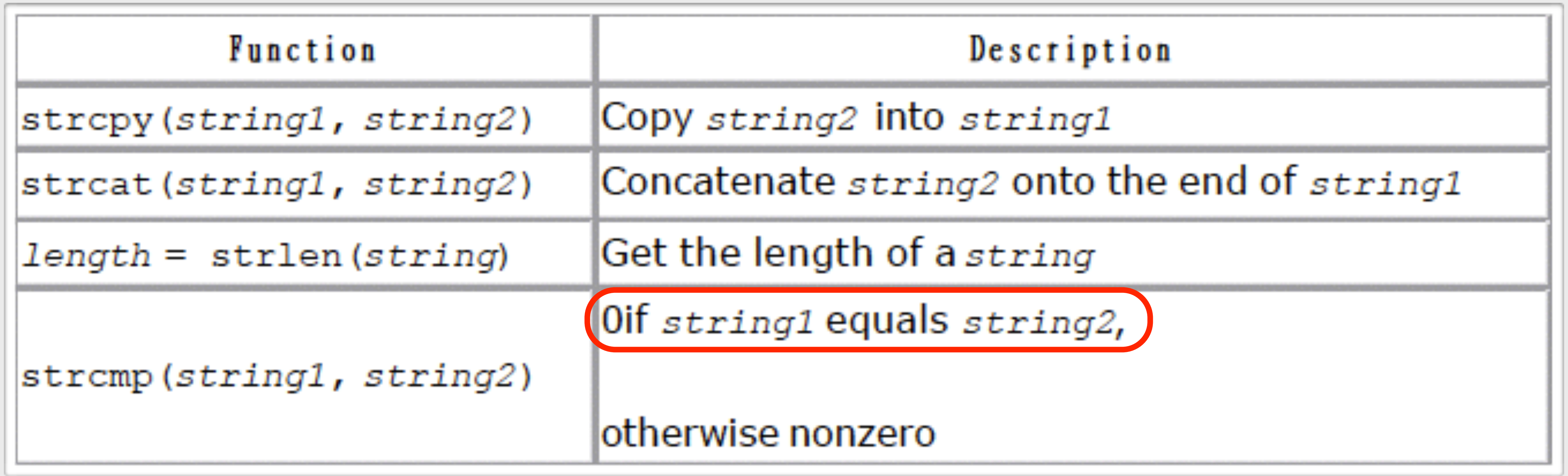
```
char name[4];

name[0] = ‘S’;
name[1] = ‘a’;
name[2] = ‘m’;
name[3] = ‘\0’;
```

Copy a String

- `name = “Sam”;    // illegal`
  - C does not allow one array to be assigned to another
- Use `strcpy()` to copy a string

```
char name[4];

strcpy(name, “Sam”);
```

Common String Functions



Function	Description
strcpy(*string1*, *string2*)	Copy *string2* into *string1*
strcat(*string1*, *string2*)	Concatenate *string2* onto the end of *string1*
length = strlen(*string*)	Get the length of a *string*
strcmp(*string1*, *string2*)	0if *string1* equals *string2*, otherwise nonzero

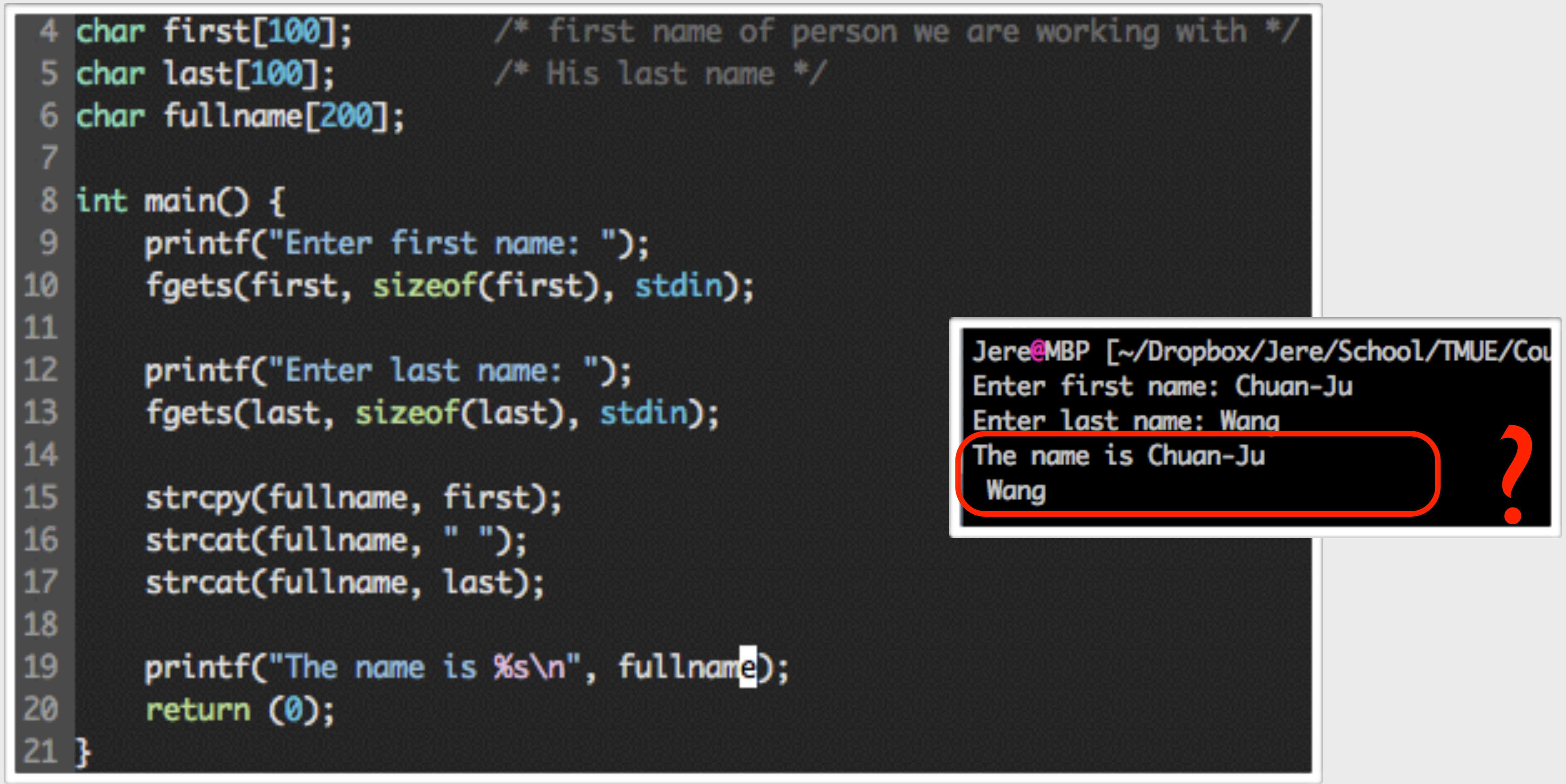
Reading Strings

- The standard functions **`fgets`** can be used to read a string from the keyboard

```
fgets(name, sizeof(name), stdin);
```

Reading Strings (cont.)

- Example: fullname.c



```c
char first[100];          /* first name of person we are working with */
char last[100];           /* His last name */
char fullname[200];

int main() {
    printf("Enter first name: ");
    fgets(first, sizeof(first), stdin);

    printf("Enter last name: ");
    fgets(last, sizeof(last), stdin);

    strcpy(fullname, first);
    strcat(fullname, " ");
    strcat(fullname, last);

    printf("The name is %s\n", fullname);
    return (0);
}
```

```
Jere@MBP [~/Dropbox/Jere/School/TMUE/Cou
Enter first name: Chuan-Ju
Enter last name: Wang
The name is Chuan-Ju
 Wang
```

Reading Strings (cont.)

- The **`fgets`** function gets the entire line, including the end-of-line.
- We have to get rid of the character before printing.

```
first[ strlen(first) - 1 ] = ‘\0’;
last[ strlen(last) - 1 ] = ‘\0’;
```

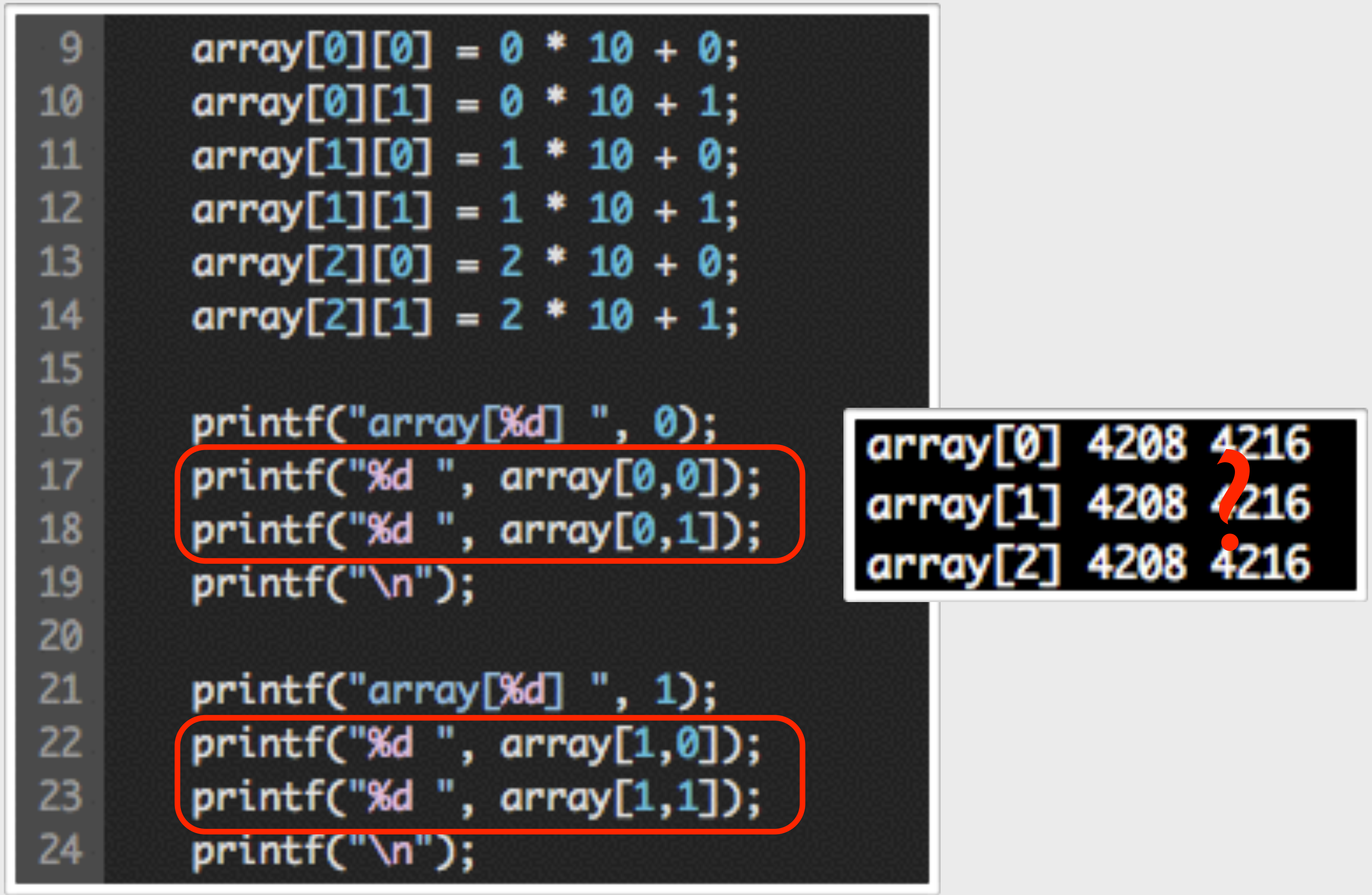
Multidimensional Arrays

- `type variable[size1][size2]`

```
int matrix[2][4];

/* declare a 2*4 int array */

matrix[1][2] = 10; /* assign 10 */
```

Multidimensional Arrays (cont.)

- Example: multiarray.c



```
    array[0][0] = 0 * 10 + 0;
    array[0][1] = 0 * 10 + 1;
    array[1][0] = 1 * 10 + 0;
    array[1][1] = 1 * 10 + 1;
    array[2][0] = 2 * 10 + 0;
    array[2][1] = 2 * 10 + 1;

    printf("array[%d] ", 0);
    printf("%d ", array[0,0]);
    printf("%d ", array[0,1]);
    printf("\n");

    printf("array[%d] ", 1);
    printf("%d ", array[1,0]);
    printf("%d ", array[1,1]);
    printf("\n");
```

```
array[0] 4208 4216
array[1] 4208 4216
array[2] 4208 4216
```

Multidimensional Arrays (cont.)

- C does not allow the notation used in other language of **`matrix[10,12]`**.

```
print(“%d”, array[0][0]);
print(“%d”, array[0][1]);
...
```

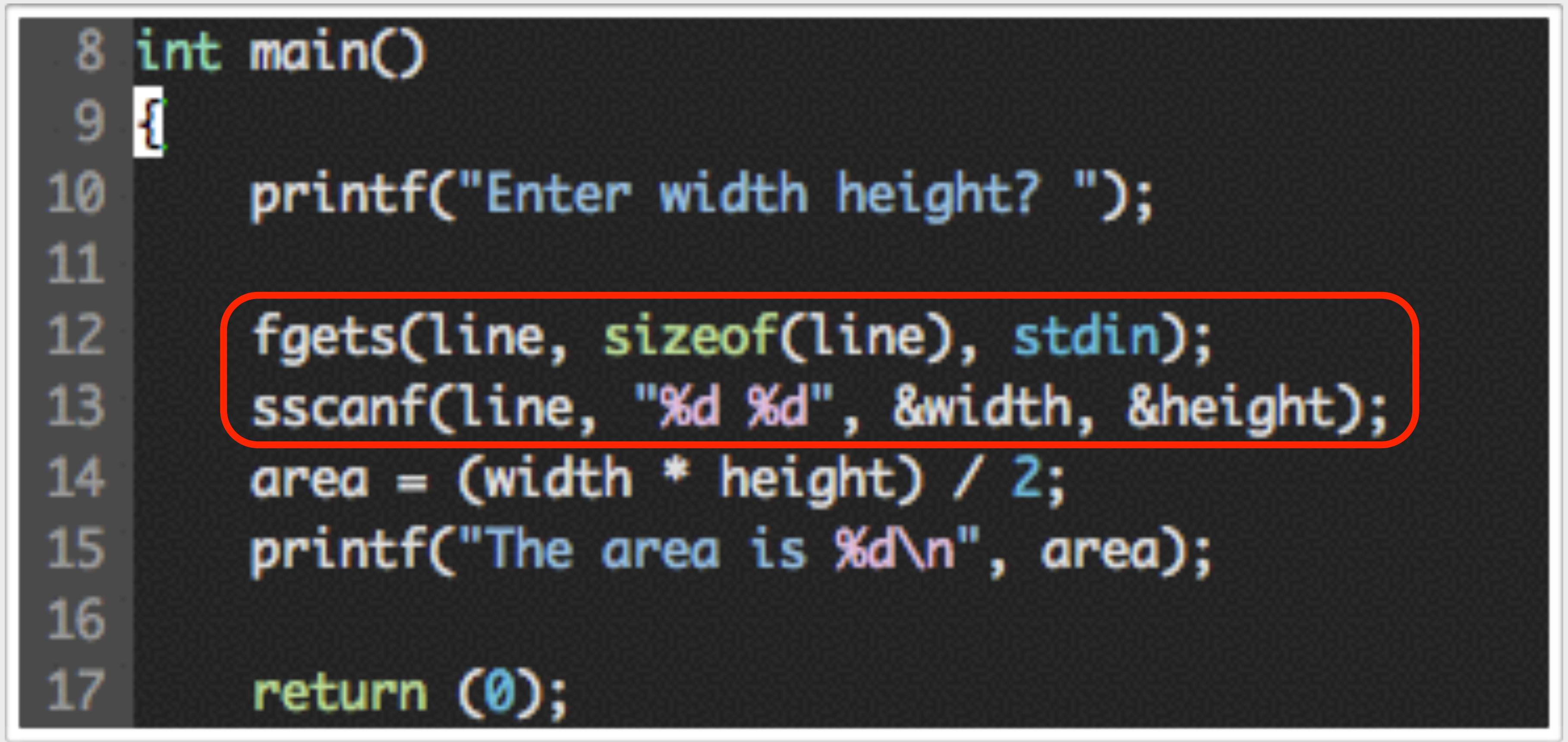
Reading Numbers

- The function **`scanf`** is notorious.
  - because of its poor end-of-line handling
- In stead, use **`fgets`** to read a line of input and **`sscanf`** to convert the text into numbers

```
char line[100];
fgets(line, sizeof(line), stdin);
sscanf(line, format, &variable1,
&variable2);
```

Reading Numbers (cont.)

- Example: triangle.c



```
int main()
{
    printf("Enter width height? ");

    fgets(line, sizeof(line), stdin);
    sscanf(line, "%d %d", &width, &height);
    area = (width * height) / 2;
    printf("The area is %d\n", area);

    return (0);
```

Initialize Variables into Array

```
int product_codes[3]

= {10,972,45};
```

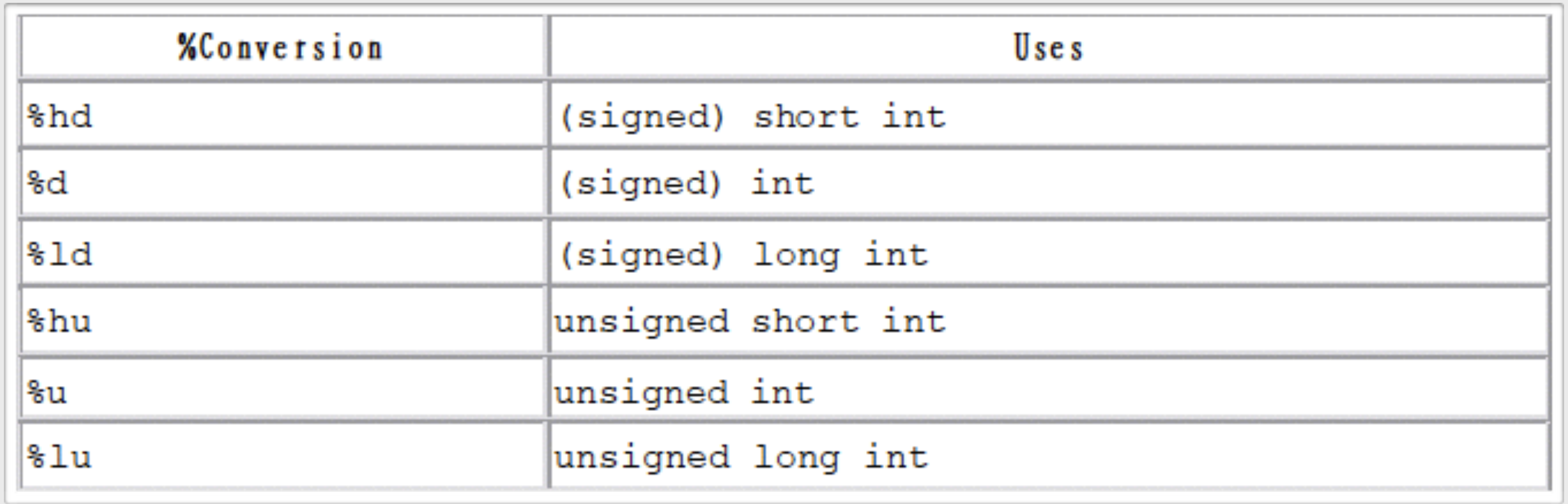
```
int matrix[2][4] = {

{1,2,3,4},

{10,20,30,40}

};
```

```
char name[50] = “Sam”;

is equivalent to

char name[50];

strcpy(name, “Sam”);
```

```
char name[50];

name=”Sam”; //Wrong!!
```

Types of Integers

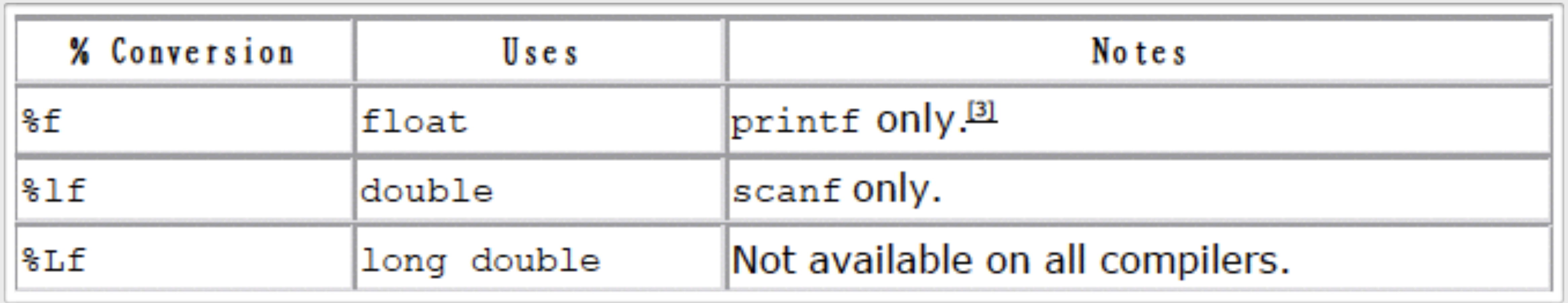
- Integer **`printf`**/**`sscanf`** Conversions



%Conversion	Uses
%hd	(signed) short int
%d	(signed) int
%ld	(signed) long int
%hu	unsigned short int
%u	unsigned int
%lu	unsigned long int

Type of Floats

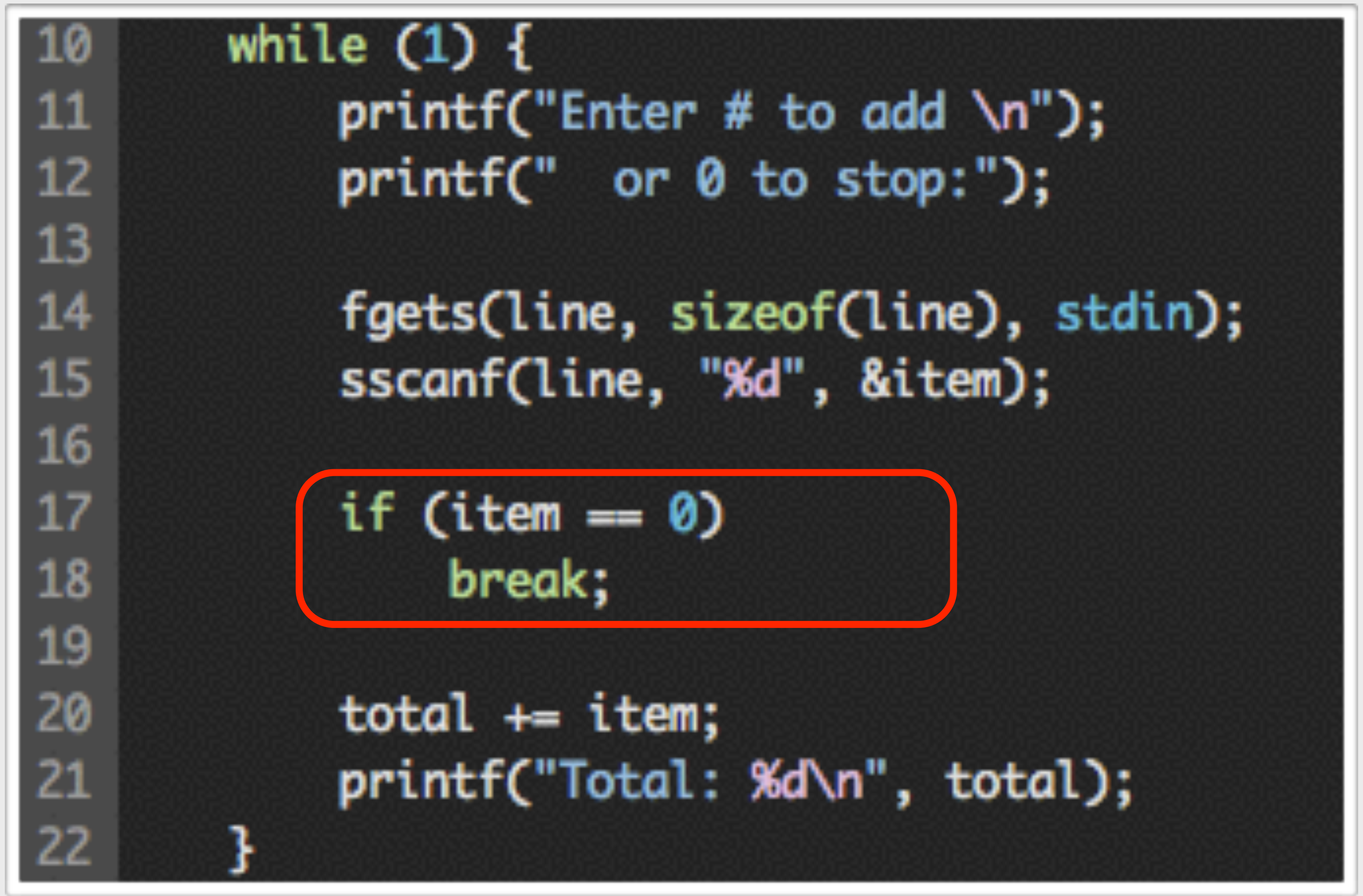
- Float **`printf`**/**`sscanf`** Conversions



% Conversion	Uses	Notes
`%f`	`float`	`printf` only.[3]
`%lf`	`double`	`scanf` only.
`%Lf`	`long double`	Not available on all compilers.

break Statement

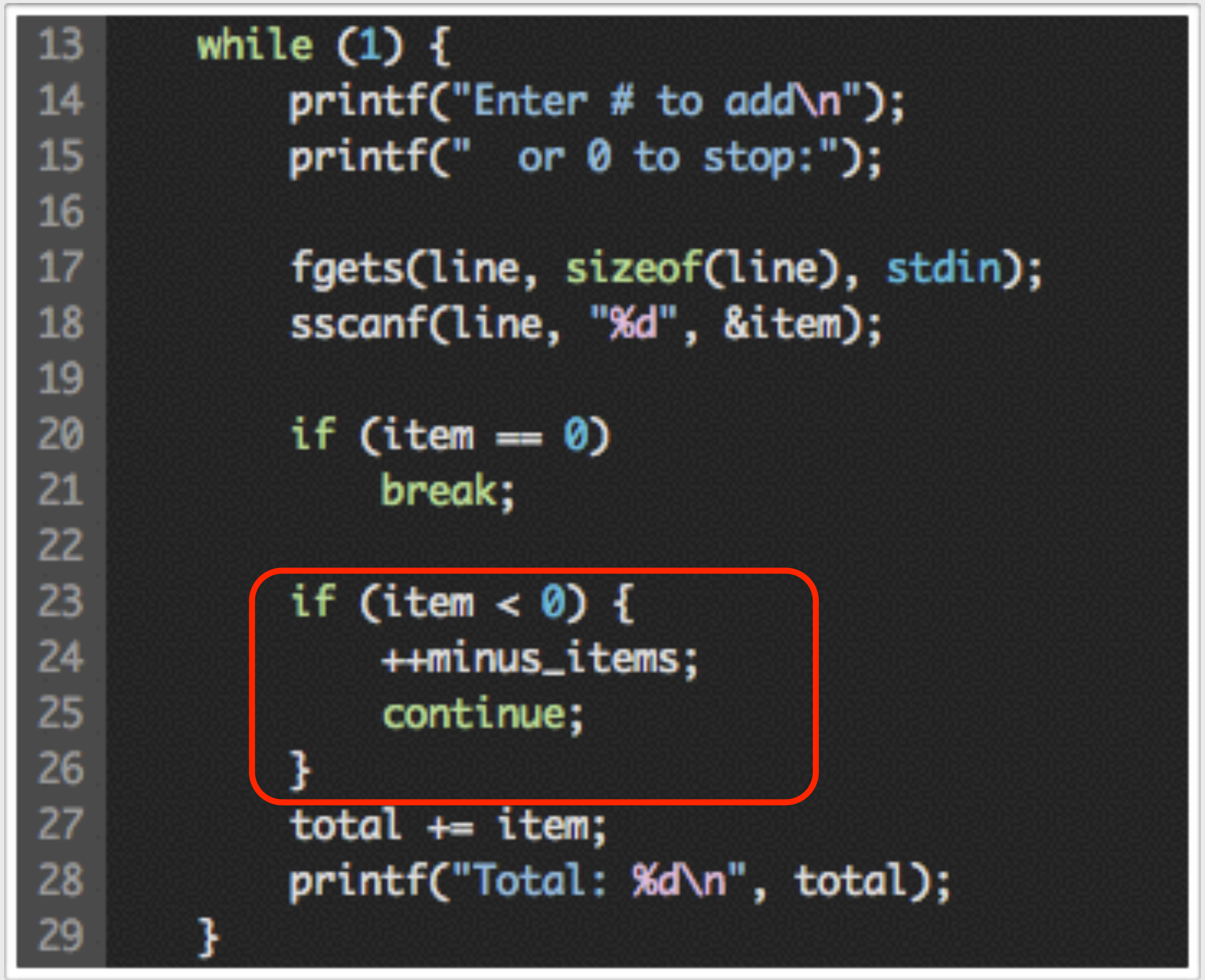
- Example: total_break.c



```
    while (1) {
        printf("Enter # to add \n");
        printf("  or 0 to stop:");

        fgets(line, sizeof(line), stdin);
        sscanf(line, "%d", &item);

        if (item == 0)
            break;

        total += item;
        printf("Total: %d\n", total);
    }
```

`continue` Statement

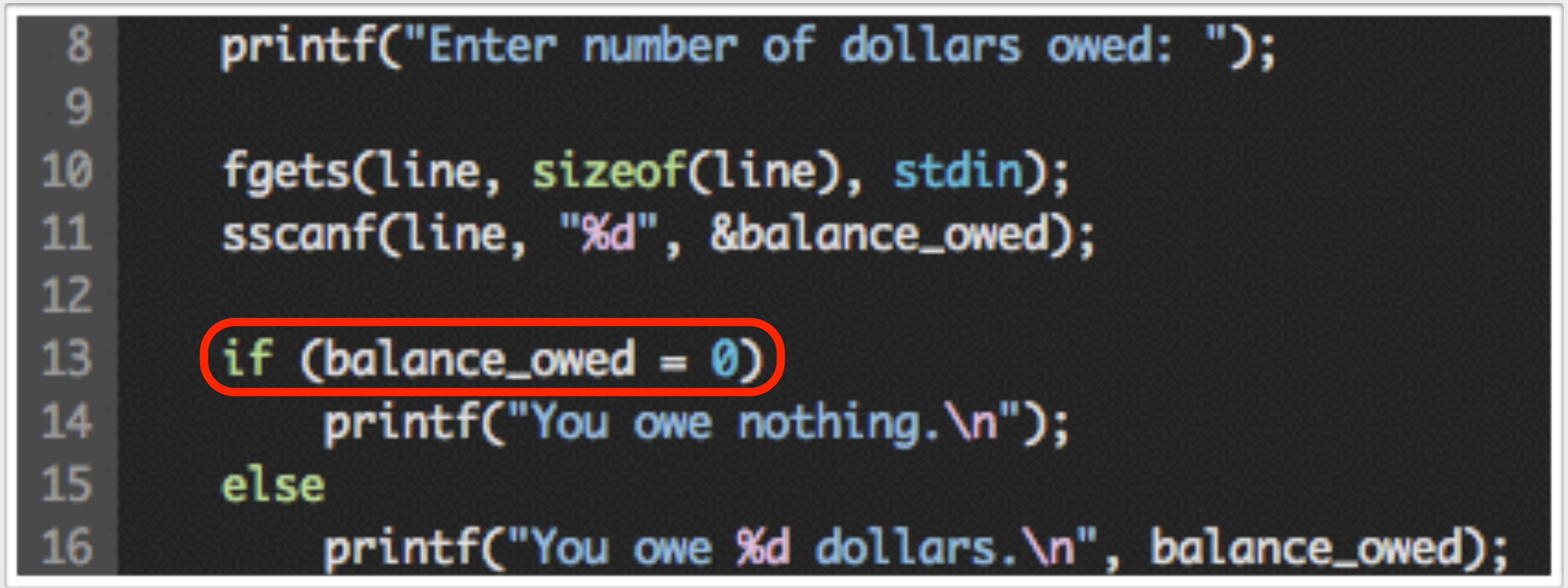
- Example: total_continue.c



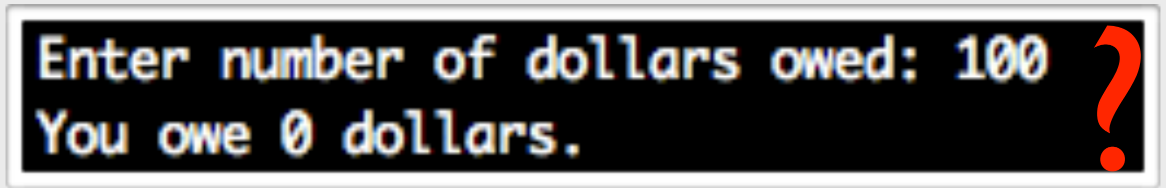
```c
    while (1) {
        printf("Enter # to add\n");
        printf("  or 0 to stop:");

        fgets(line, sizeof(line), stdin);
        sscanf(line, "%d", &item);

        if (item == 0)
            break;

        if (item < 0) {
            ++minus_items;
            continue;
        }
        total += item;
        printf("Total: %d\n", total);
    }
```

Assignment Anywhere Side Effect

- Example: owe0.c



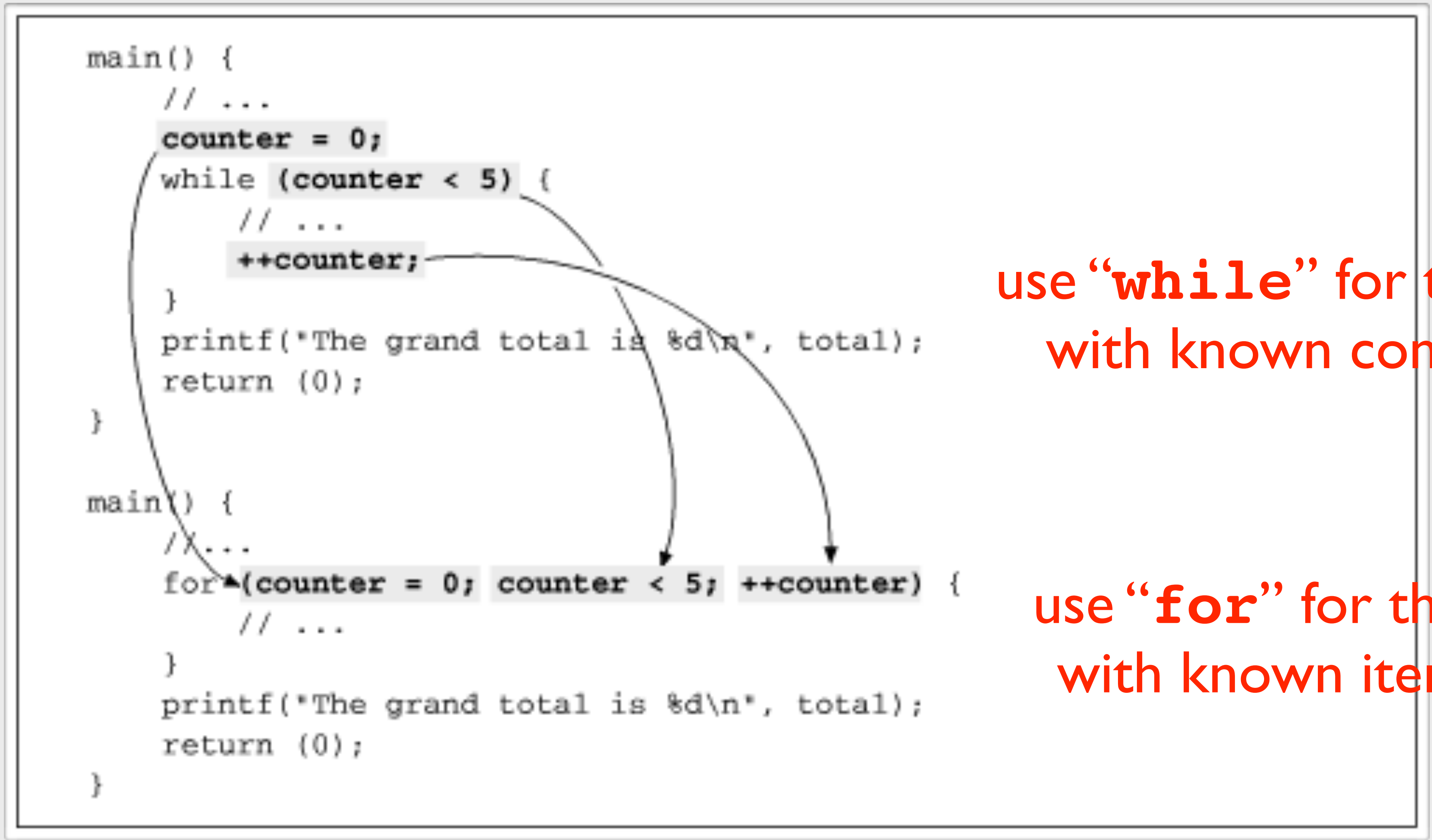
```c
    printf("Enter number of dollars owed: ");

    fgets(line, sizeof(line), stdin);
    sscanf(line, "%d", &balance_owed);

    if (balance_owed = 0)
        printf("You owe nothing.\n");
    else
        printf("You owe %d dollars.\n", balance_owed);
```



```
Enter number of dollars owed: 100
You owe 0 dollars.
```

?

Similarities between “**`while`**” and “**`for`**”



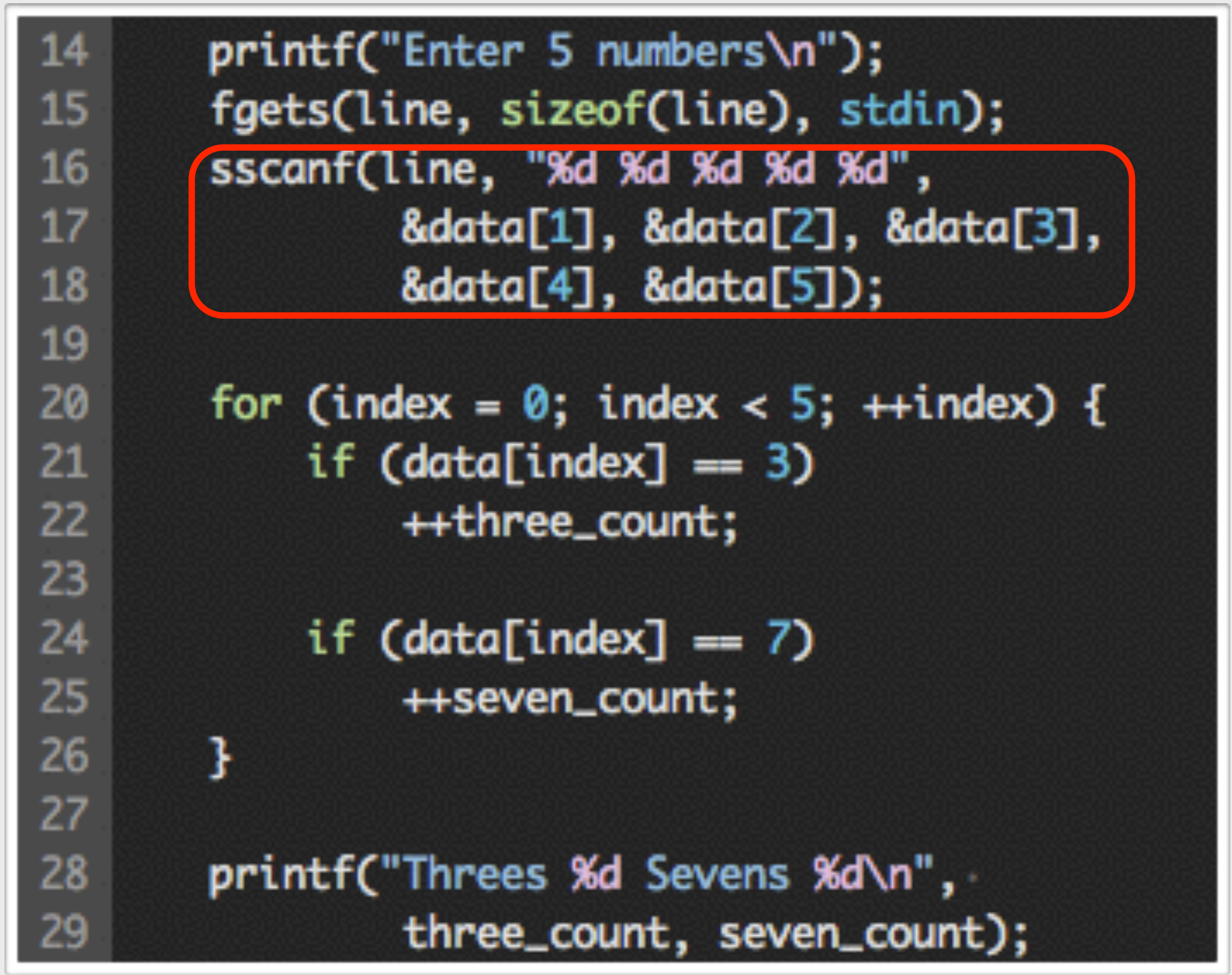
```
main() {
    // ...
    counter = 0;
    while (counter < 5) {
        // ...
        ++counter;
    }
    printf("The grand total is %d\n", total);
    return (0);
}

main() {
    // ...
    for (counter = 0; counter < 5; ++counter) {
        // ...
    }
    printf("The grand total is %d\n", total);
    return (0);
}
```

use “**`while`**” for the loops with known conditions

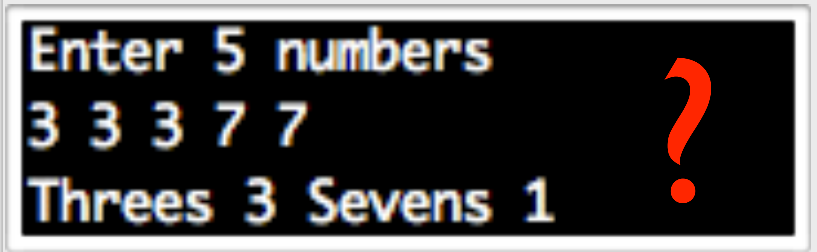
use “**`for`**” for the loops with known iterations

for Statement

- Example: count_number.c



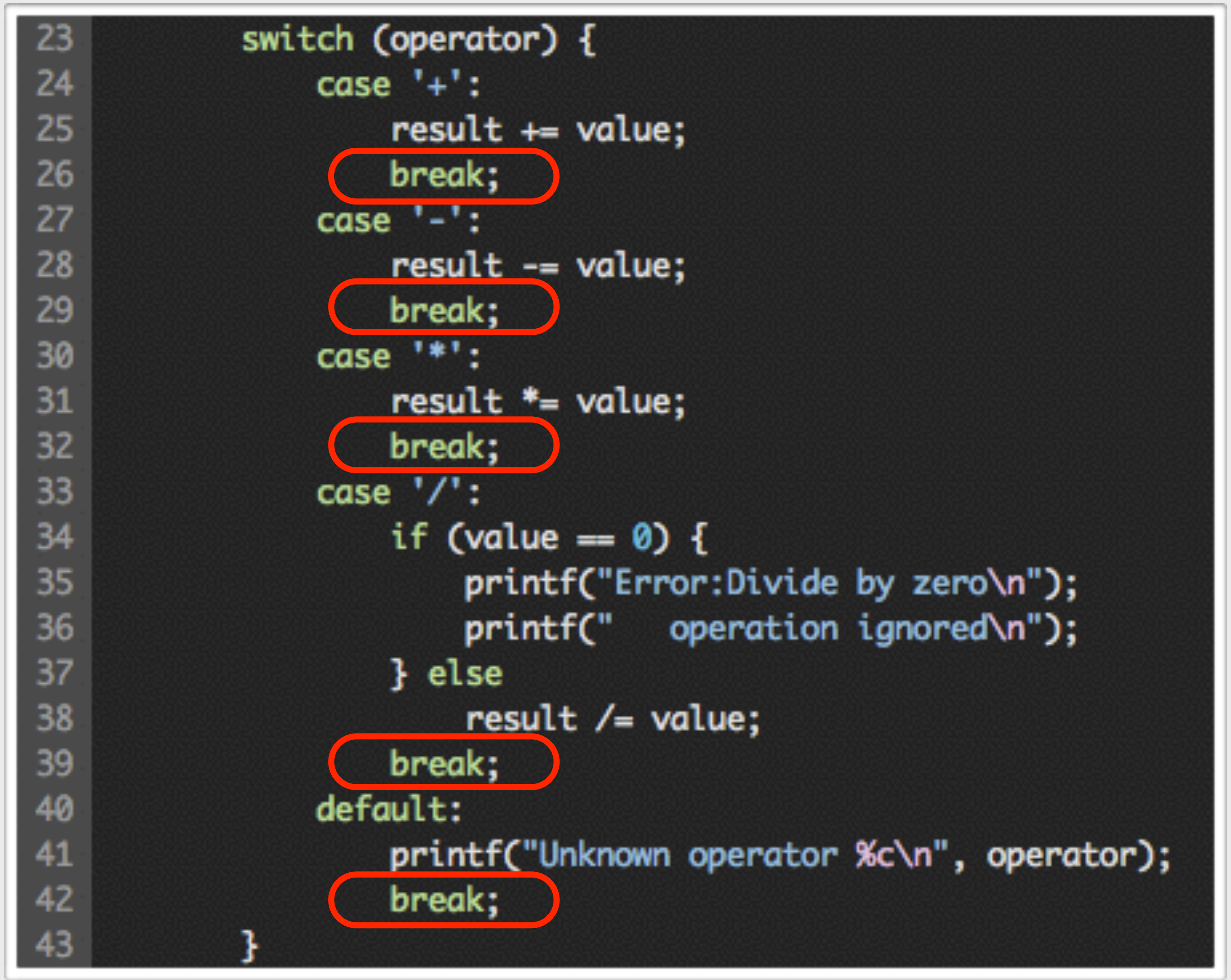
```c
    printf("Enter 5 numbers\n");
    fgets(line, sizeof(line), stdin);
    sscanf(line, "%d %d %d %d %d",
            &data[1], &data[2], &data[3],
            &data[4], &data[5]);

    for (index = 0; index < 5; ++index) {
        if (data[index] == 3)
            ++three_count;

        if (data[index] == 7)
            ++seven_count;
    }

    printf("Threes %d Sevens %d\n",
            three_count, seven_count);
```



```
Enter 5 numbers
3 3 3 7 7
Threes 3 Sevens 1
```

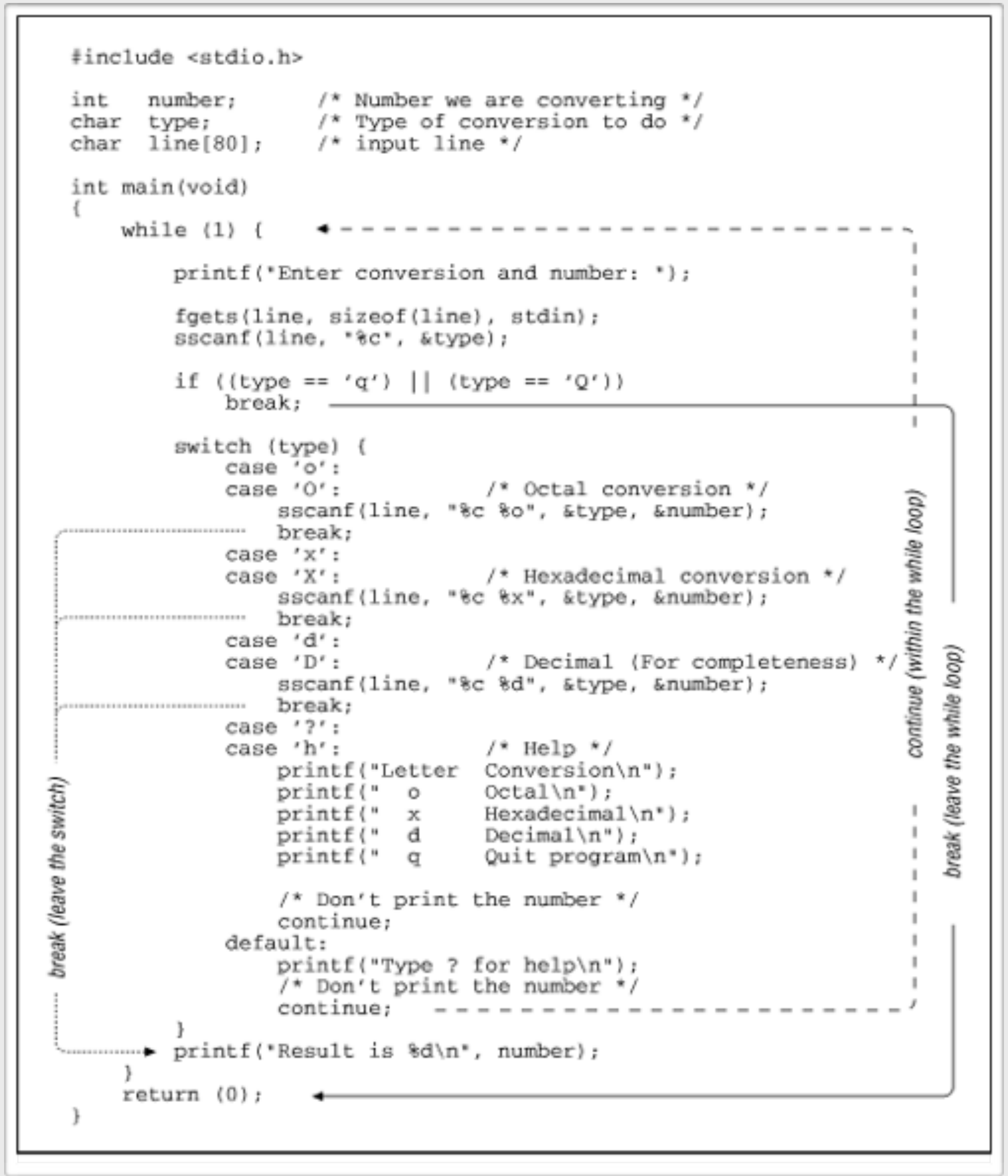
switch and break

- Example: calculator.c



```
switch (operator) {
    case '+':
        result += value;
        break;
    case '-':
        result -= value;
        break;
    case '*':
        result *= value;
        break;
    case '/':
        if (value == 0) {
            printf("Error:Divide by zero\n");
            printf("    operation ignored\n");
        } else
            result /= value;
        break;
    default:
        printf("Unknown operator %c\n", operator);
        break;
}
```

switch, **break**, and **continue**



```
#include <stdio.h>

int   number;        /* Number we are converting */
char  type;          /* Type of conversion to do */
char  line[80];      /* input line */

int main(void)
{
    while (1) {

        printf("Enter conversion and number: ");

        fgets(line, sizeof(line), stdin);
        sscanf(line, "%c", &type);

        if ((type == ‘q’) || (type == ‘Q’))
            break;

        switch (type) {
            case ‘o’:
            case ‘O’:             /* Octal conversion */
                sscanf(line, "%c %o", &type, &number);
                break;
            case ‘x’:
            case ‘X’:             /* Hexadecimal conversion */
                sscanf(line, "%c %x", &type, &number);
                break;
            case ‘d’:
            case ‘D’:             /* Decimal (For completeness) */
                sscanf(line, "%c %d", &type, &number);
                break;
            case ‘?’:
            case ‘h’:             /* Help */
                printf("Letter  Conversion\n");
                printf("  o     Octal\n");
                printf("  x     Hexadecimal\n");
                printf("  d     Decimal\n");
                printf("  q     Quit program\n");

                /* Don’t print the number */
                continue;
            default:
                printf("Type ? for help\n");
                /* Don’t print the number */
                continue;
        }
        printf("Result is %d\n", number);
    }
    return (0);
}
```

continue (within the while loop)
break (leave the while loop)
break (leave the switch)

Function

Introduction

- Experience has shown that the best way to develop and maintain a large program is to construct it from smaller pieces or modules, each of which is more manageable than the original program.
- This technique is called divide and conquer.

Program Modules in C

- Modules in C are called functions.
- C programs are typically written by combining new functions you write with “prepackaged” functions available in the C Standard Library.
- The C Standard Library provides a rich collection of functions for performing common mathematical calculations, string manipulations, character manipulations, input/output, and many other useful operations.

Program Modules in C (Cont.)

- The functions **`printf`**, **`scanf`** and **`pow`** are Standard Library functions.
- These are sometimes referred to as programmer-defined functions.
- Functions are invoked by a function call, which specifies the function name and provides information (as arguments) that the called function needs to perform its designated task.

Program Modules in C (Cont.)

- Figure 5.1 shows the **`main`** function communicating with several worker functions in a hierarchical manner.
- Note that **`worker1`** acts as a boss function to **`worker4`** and **`worker5`**.

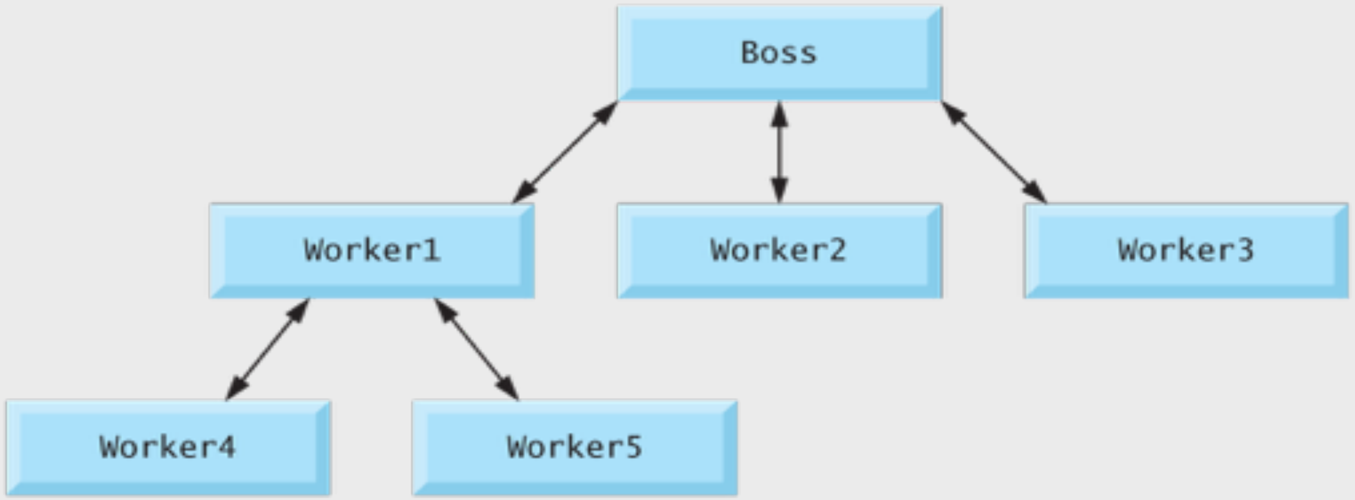


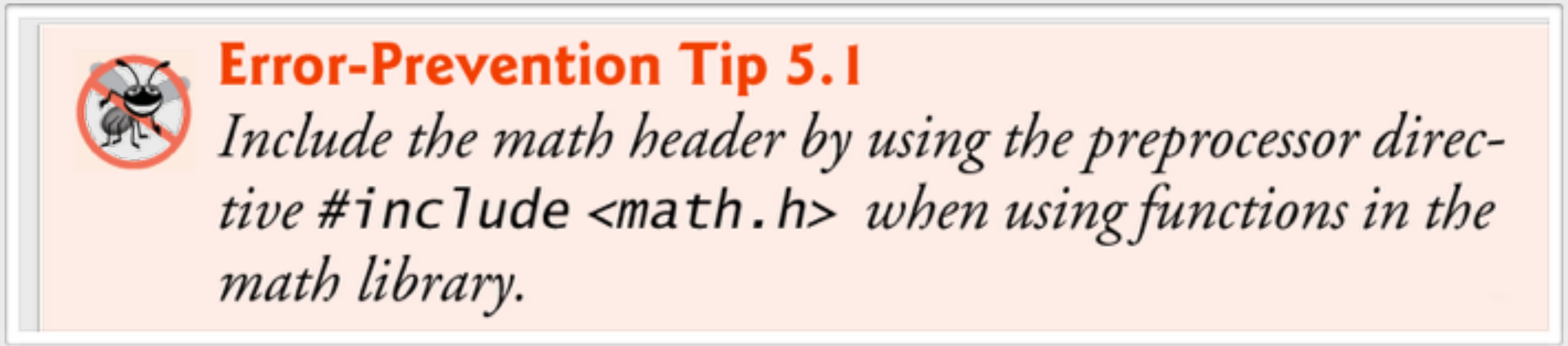
Fig. 5.1 | Hierarchical boss function/worker function relationship.

Math Library Functions

- Math library functions allow you to perform certain common mathematical calculations.
- Functions are normally used in a program by writing the name of the function followed by a left parenthesis followed by the argument (or a comma-separated list of arguments) of the function followed by a right parenthesis.
- For example

```
printf( "%.2f", sqrt( 900.0 ) );
```

Math Library Functions (Cont.)



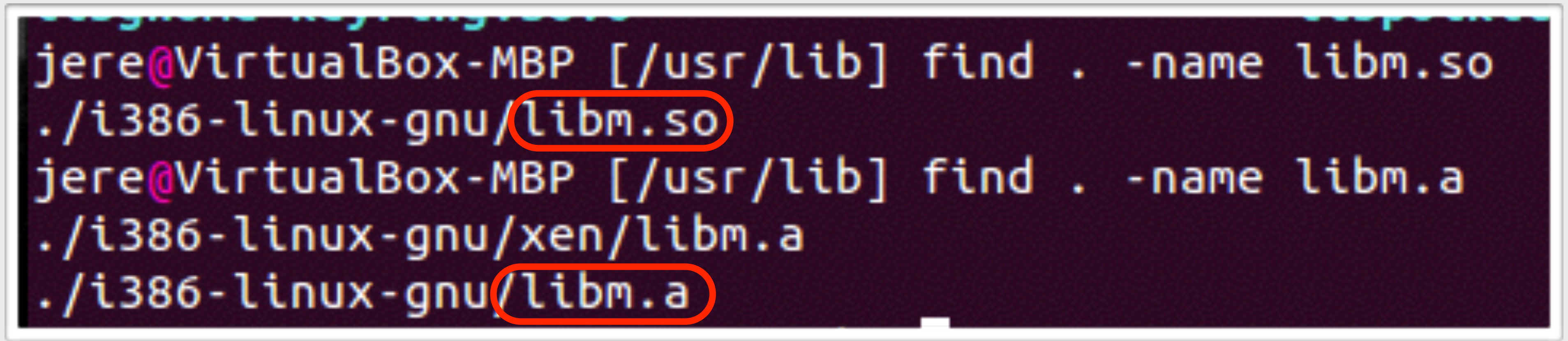
Error-Prevention Tip 5.1

Include the math header by using the preprocessor directive `#include <math.h>` *when using functions in the math library.*

Where are the library files?

Static (.a) vs Shared (.so.N)*

`gcc abc.c -lm`



```
jere@VirtualBox-MBP [/usr/lib] find . -name libm.so
./i386-linux-gnu/libm.so
jere@VirtualBox-MBP [/usr/lib] find . -name libm.a
./i386-linux-gnu/xen/libm.a
./i386-linux-gnu/libm.a
```

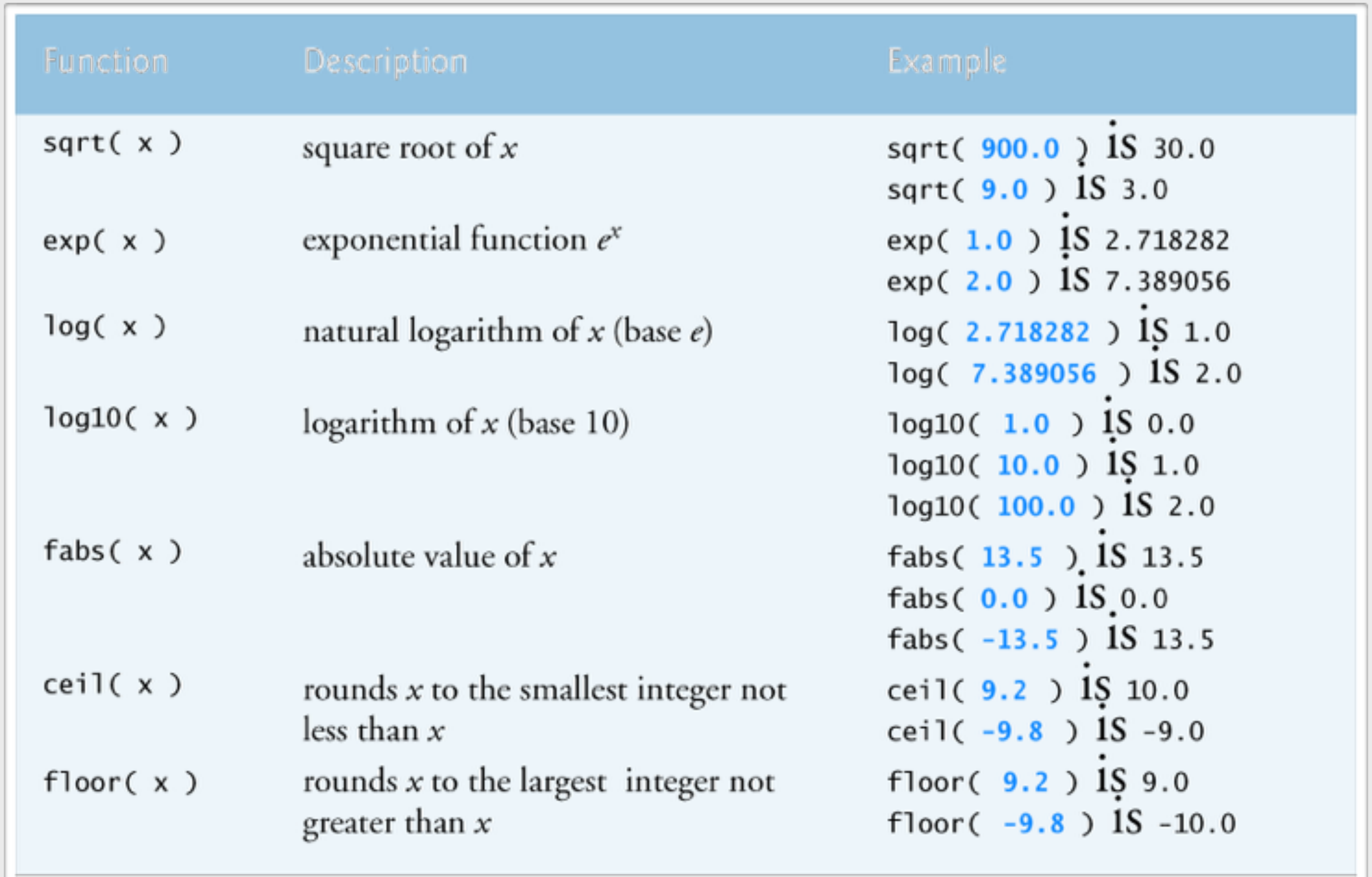
* Library usage: http://godleon.blogspot.com/2008/02/c-header-file-library-library-static.html http://blog.xuite.net/csiewap/cc/23626229-Using+GCC+to+create+static+and+shared+library+.so

Math Library Functions (Cont.)

- Function arguments may be constants, variables, or expressions.
- If **`c1=13.0`**, **`d=3.0`** and **`f=4.0`**, then the statement

```
printf( "%.2f", sqrt( c1 + d * f ) );
```

Math Library Functions (Cont.)



Function	Description	Example

`sqrt( x )`	square root of *x*	`sqrt( 900.0 ) is 30.0`
		`sqrt( 9.0 ) is 3.0`

`exp( x )`	exponential function e^x	`exp( 1.0 ) is 2.718282`
		`exp( 2.0 ) is 7.389056`

`log( x )`	natural logarithm of *x* (base *e*)	`log( 2.718282 ) is 1.0`
		`log( 7.389056 ) is 2.0`

`log10( x )`	logarithm of *x* (base 10)	`log10( 1.0 ) is 0.0`
		`log10( 10.0 ) is 1.0`
		`log10( 100.0 ) is 2.0`

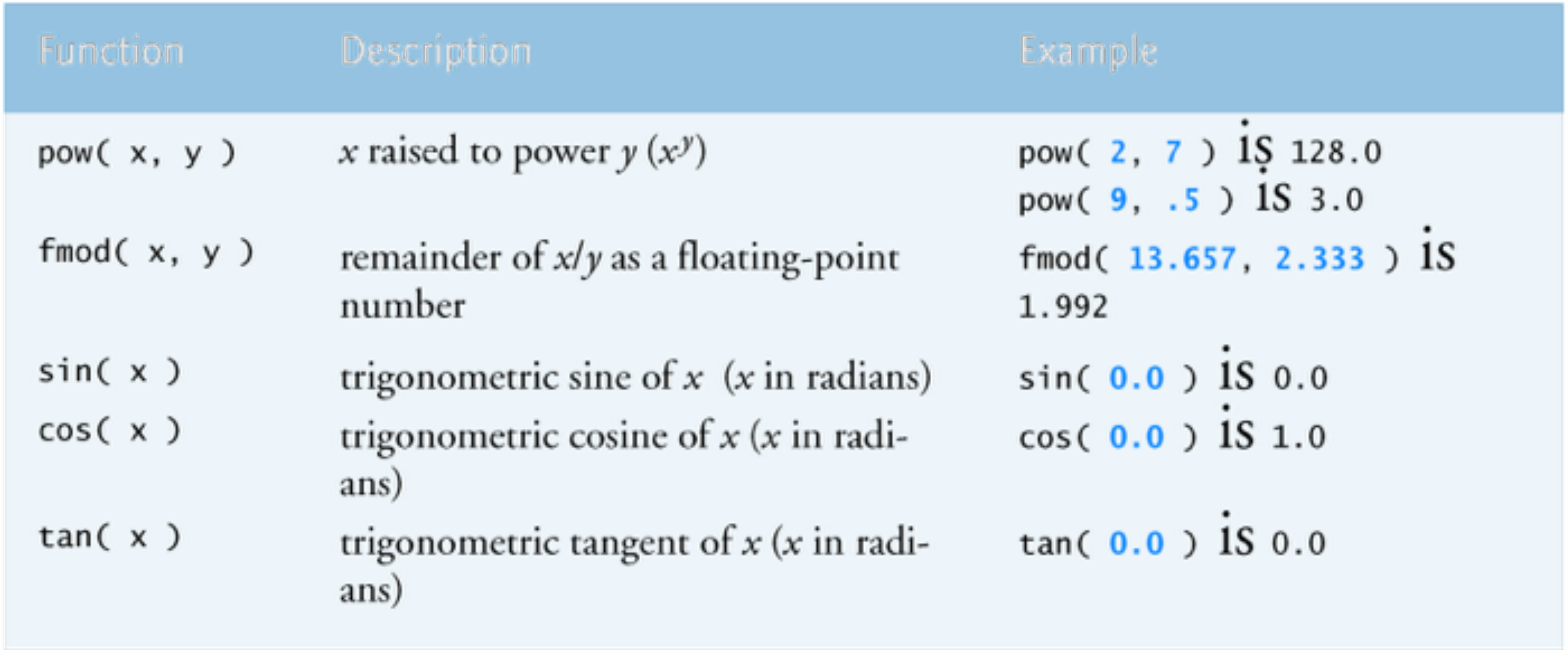
`fabs( x )`	absolute value of *x*	`fabs( 13.5 ) is 13.5`
		`fabs( 0.0 ) is 0.0`
		`fabs( -13.5 ) is 13.5`

`ceil( x )`	rounds *x* to the smallest integer not less than *x*	`ceil( 9.2 ) is 10.0`
		`ceil( -9.8 ) is -9.0`

`floor( x )`	rounds *x* to the largest integer not greater than *x*	`floor( 9.2 ) is 9.0`
		`floor( -9.8 ) is -10.0`

Fig. 5.2 | Commonly used math library functions. (Part 1 of 2.)

Math Library Functions (Cont.)



Function	Description	Example

`pow( x, y )`	*x* raised to power *y* (x^y)	`pow( 2, 7 )` is `128.0`
		`pow( 9, .5 )` is `3.0`

`fmod( x, y )`	remainder of *x*/*y* as a floating-point number	`fmod( 13.657, 2.333 )` is `1.992`
`sin( x )`	trigonometric sine of *x* (*x* in radians)	`sin( 0.0 )` is `0.0`
`cos( x )`	trigonometric cosine of *x* (*x* in radians)	`cos( 0.0 )` is `1.0`
`tan( x )`	trigonometric tangent of *x* (*x* in radians)	`tan( 0.0 )` is `0.0`

Fig. 5.2 | Commonly used math library functions. (Part 2 of 2.)

Functions

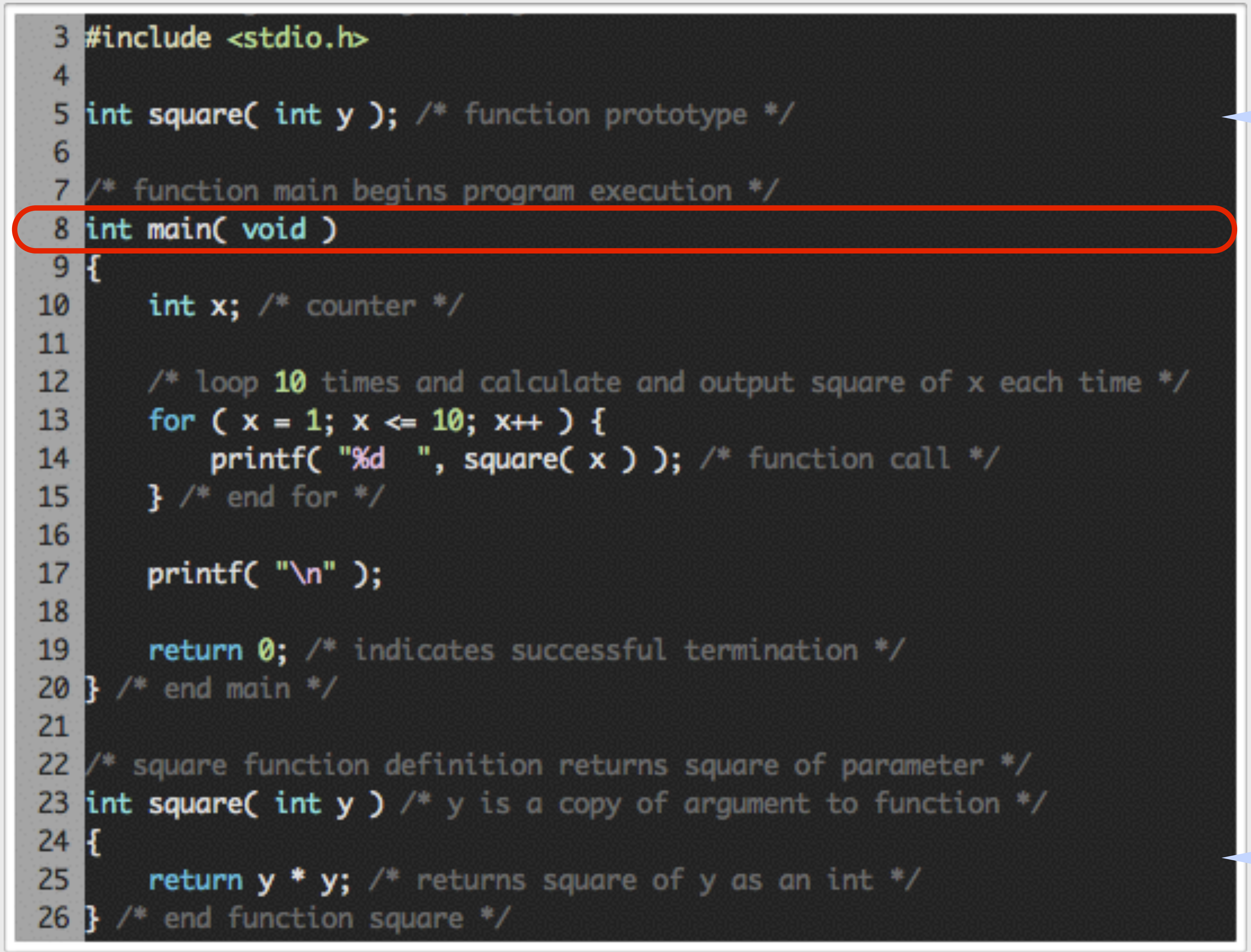
- Functions allow you to modularize a program.
- All variables defined in function definitions are local variables—they're known only in the function in which they're defined.
- Most functions have a list of parameters that provide the means for communicating information between functions.
- A function's parameters are also local variables of that function.

Function Definitions

- We now consider how to write custom functions.
- Consider the following example that uses a function `square` to calculate and print the squares of the integers from 1 to 10.

Function Definitions (Cont.)

- Example: fig05_03.c



```c
#include <stdio.h>

int square( int y ); /* function prototype */

/* function main begins program execution */
int main( void )
{
    int x; /* counter */

    /* loop 10 times and calculate and output square of x each time */
    for ( x = 1; x <= 10; x++ ) {
        printf( "%d  ", square( x ) ); /* function call */
    } /* end for */

    printf( "\n" );

    return 0; /* indicates successful termination */
} /* end main */

/* square function definition returns square of parameter */
int square( int y ) /* y is a copy of argument to function */
{
    return y * y; /* returns square of y as an int */
} /* end function square */
```

function prototype

function definition

Function Definitions (Cont.)

- Function prototype
  - informs the compiler
- The format of a function definition is

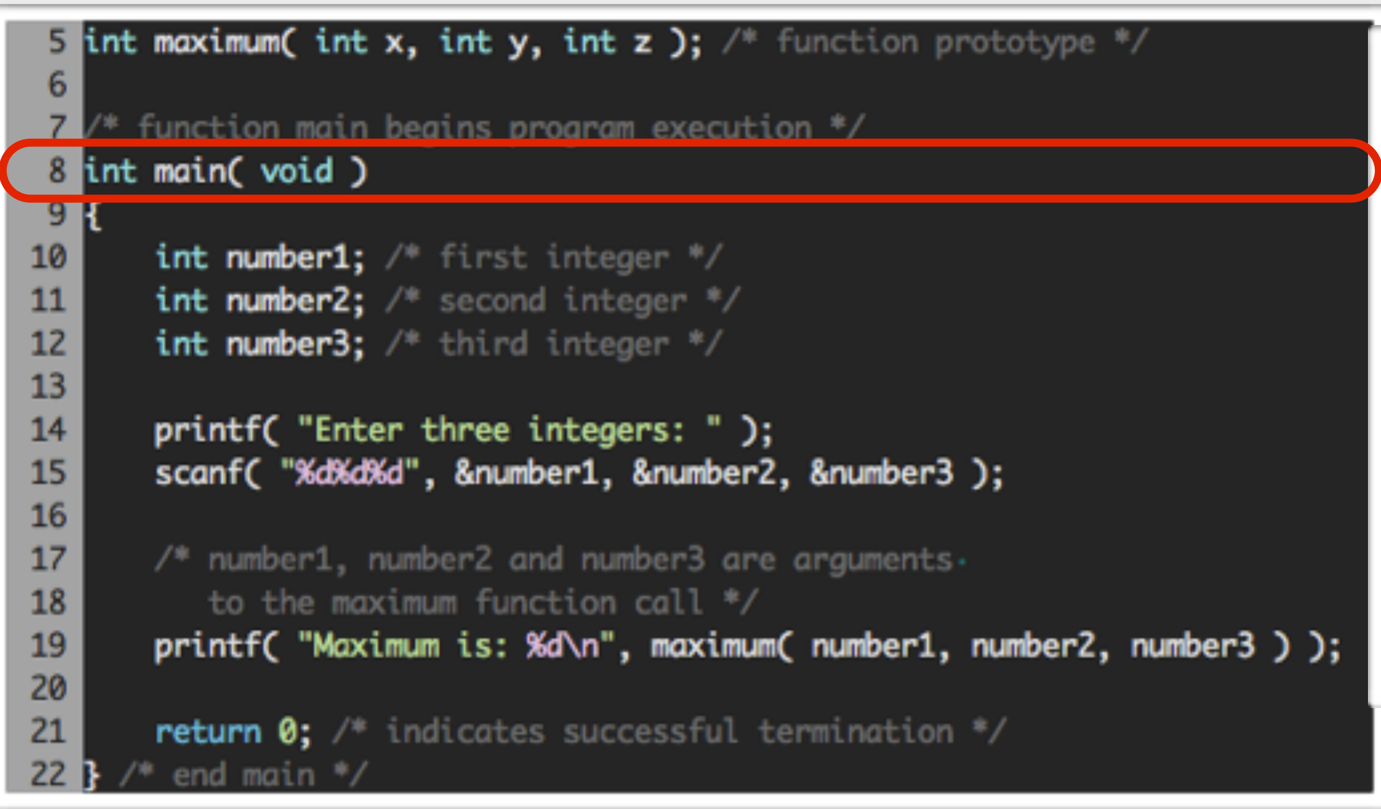
```
return-value-type function-name( parameter-
list )
{
   definitions
   statements
}
```

Function Definitions (Cont.)

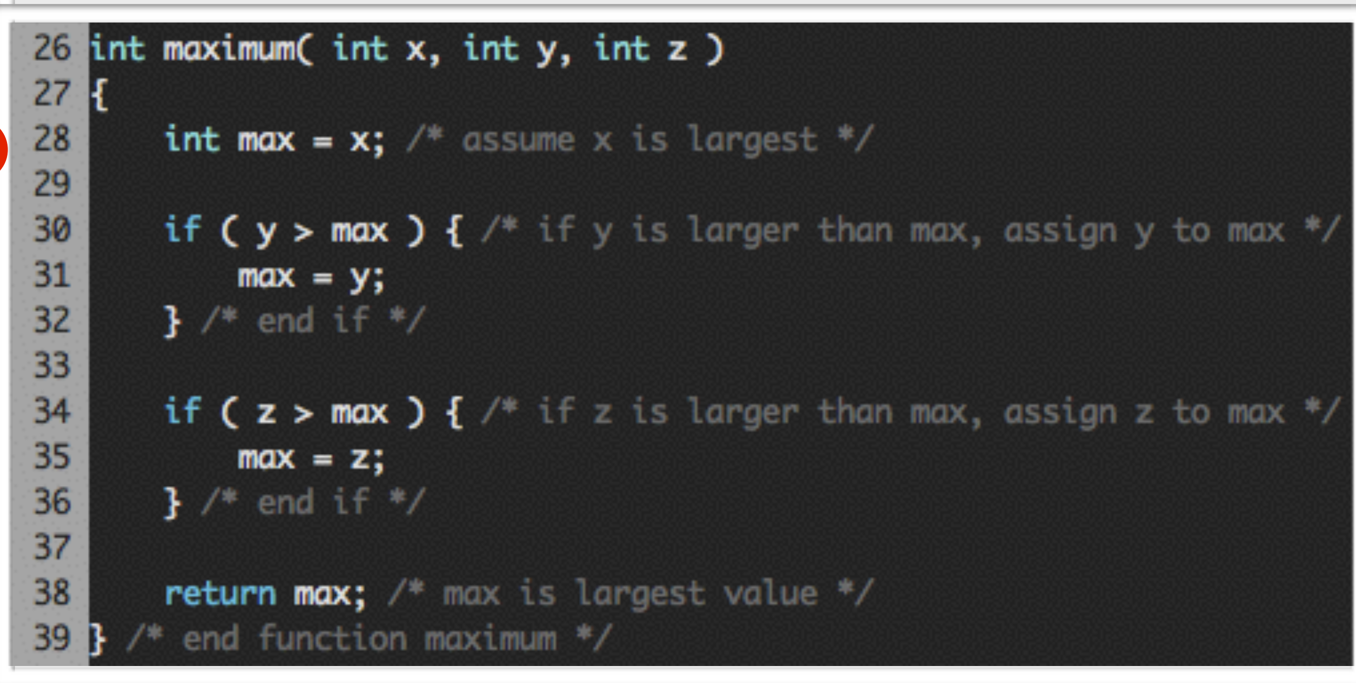
- The function-name is any valid identifier.
- The return-value-type is the data type of the result returned to the caller.
  - The return-value-type void indicates that a function does not return a value.
- Together, the return-value-type, function-name and parameter-list are sometimes referred to as the function header.

Function Definitions (Cont.)

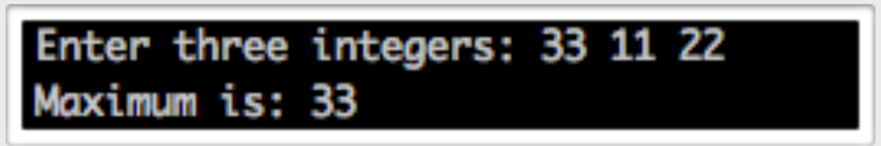
- Example: fig05_04.c



```c
int maximum( int x, int y, int z ); /* function prototype */

/* function main begins program execution */
int main( void )
{
    int number1; /* first integer */
    int number2; /* second integer */
    int number3; /* third integer */

    printf( "Enter three integers: " );
    scanf( "%d%d%d", &number1, &number2, &number3 );

    /* number1, number2 and number3 are arguments
       to the maximum function call */
    printf( "Maximum is: %d\n", maximum( number1, number2, number3 ) );

    return 0; /* indicates successful termination */
} /* end main */
```



```c
int maximum( int x, int y, int z )
{
    int max = x; /* assume x is largest */

    if ( y > max ) { /* if y is larger than max, assign y to max */
        max = y;
    } /* end if */

    if ( z > max ) { /* if z is larger than max, assign z to max */
        max = z;
    } /* end if */

    return max; /* max is largest value */
} /* end function maximum */
```



```
Enter three integers: 33 11 22
Maximum is: 33
```

Function Prototypes

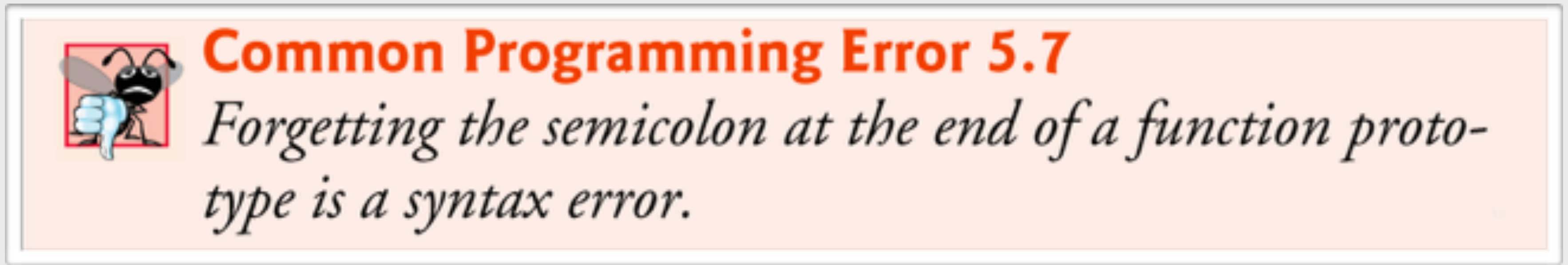
- One of the most important features of C is the function prototype.
- A function prototype tells the compiler the type of data returned by the function, the number of parameters the function expects to receive, the types of the parameters, and the order in which these parameters are expected.
- The compiler uses function prototypes to validate function calls.

Function Prototypes (Cont.)

- The function prototype for maximum in fig05_04.c

```
/* function prototype */
int maximum( int x, int y, int z );
```

- Notice that the function prototype is the same as the first line of the function definition of **`maximum`**.



Common Programming Error 5.7

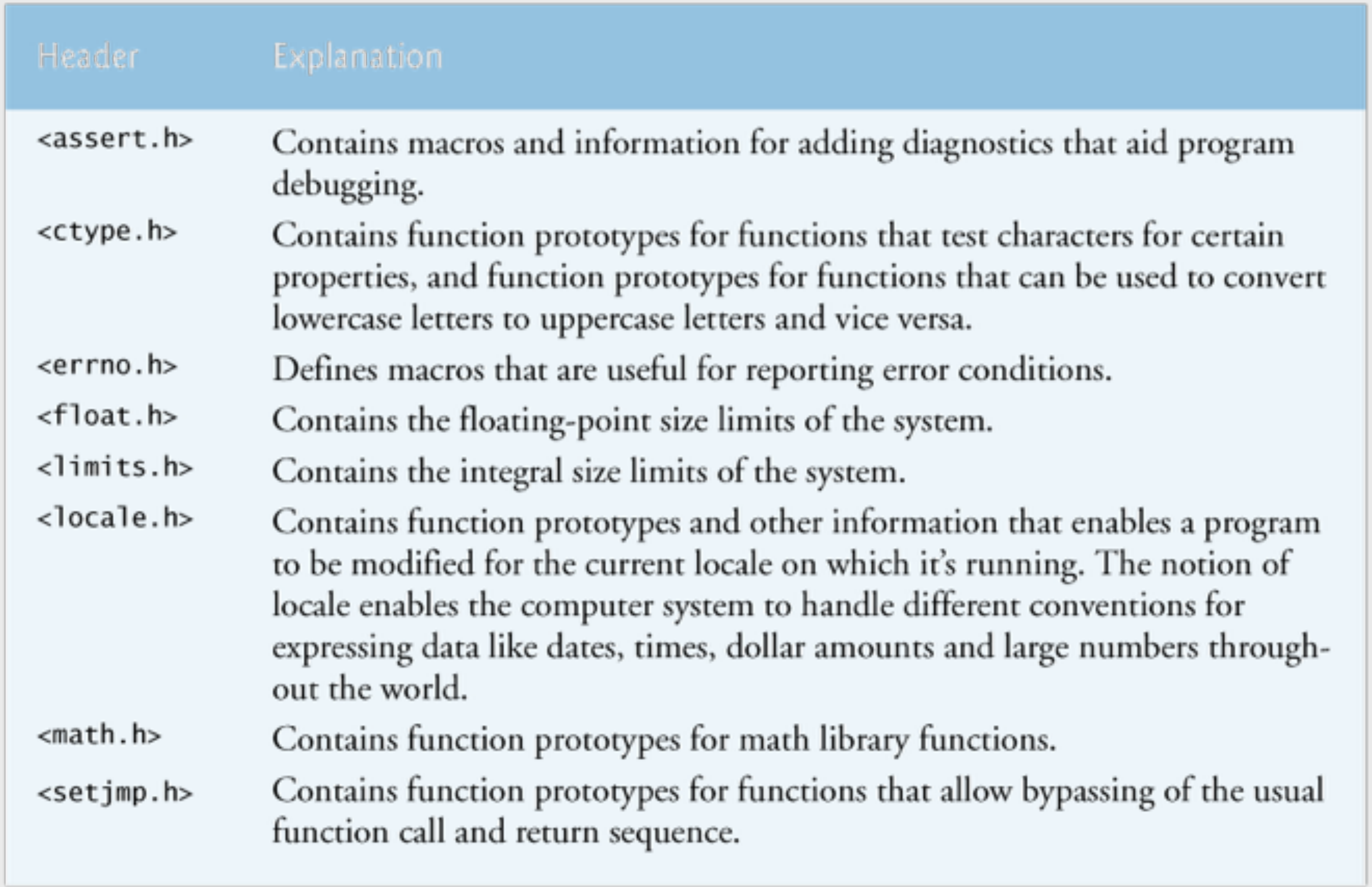
Forgetting the semicolon at the end of a function prototype is a syntax error.

Headers

- Each standard library has a corresponding header containing the function prototypes for all the functions in that library and definitions of various data types and constants needed by those functions.
- You can create custom headers.
- Programmer-defined headers should also use the .h filename extension.
- A programmer-defined header can be included by using the **#include** preprocessor directive. For example

```
#include "square.h"
```

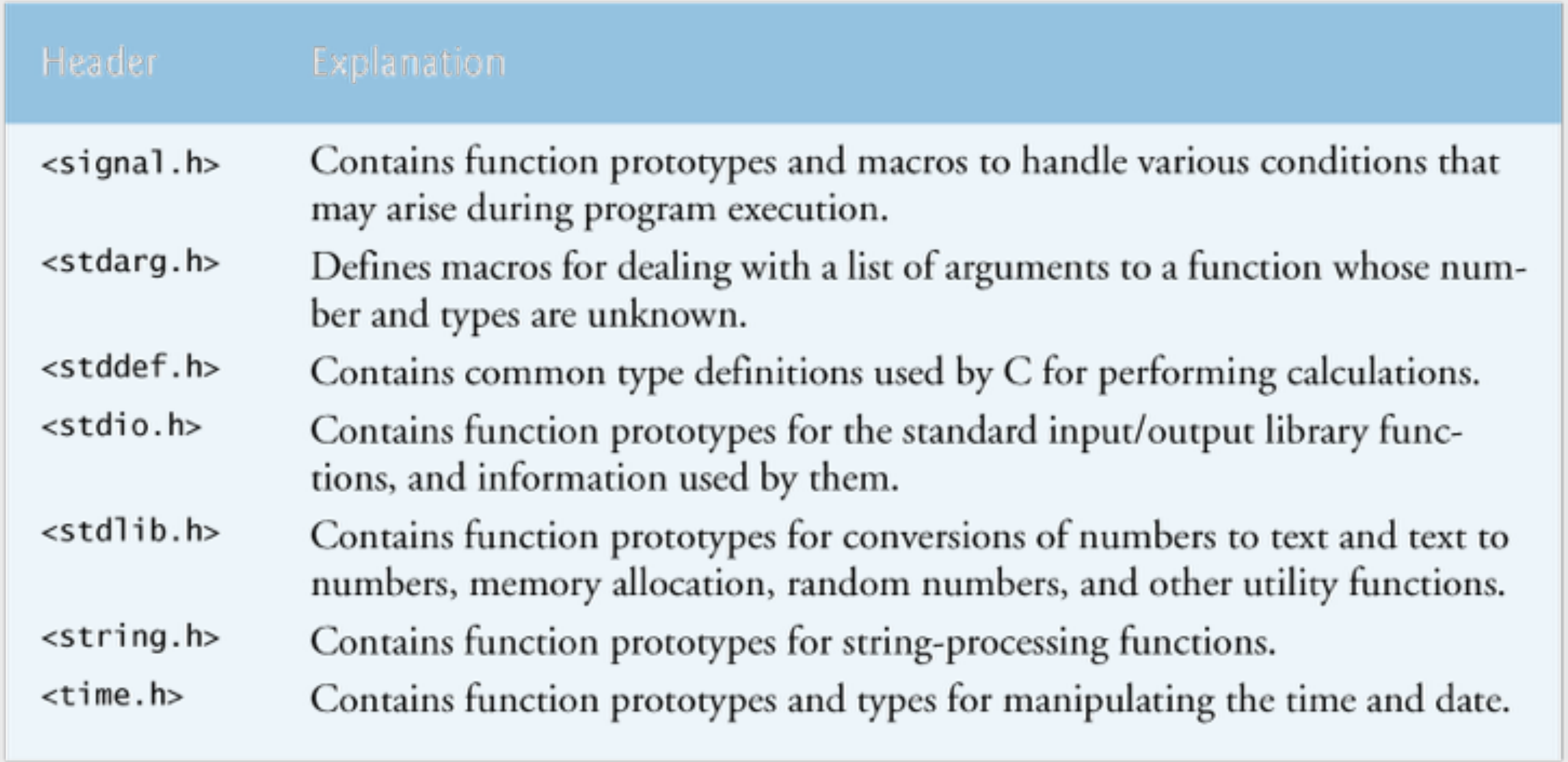
Headers (Cont.)



Header	Explanation
`<assert.h>`	Contains macros and information for adding diagnostics that aid program debugging.
`<ctype.h>`	Contains function prototypes for functions that test characters for certain properties, and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa.
`<errno.h>`	Defines macros that are useful for reporting error conditions.
`<float.h>`	Contains the floating-point size limits of the system.
`<limits.h>`	Contains the integral size limits of the system.
`<locale.h>`	Contains function prototypes and other information that enables a program to be modified for the current locale on which it's running. The notion of locale enables the computer system to handle different conventions for expressing data like dates, times, dollar amounts and large numbers throughout the world.
`<math.h>`	Contains function prototypes for math library functions.
`<setjmp.h>`	Contains function prototypes for functions that allow bypassing of the usual function call and return sequence.

Fig. 5.6 | Some of the standard library headers. (Part 1 of 2.)

Headers (Cont.)



Header	Explanation
`<signal.h>`	Contains function prototypes and macros to handle various conditions that may arise during program execution.
`<stdarg.h>`	Defines macros for dealing with a list of arguments to a function whose number and types are unknown.
`<stddef.h>`	Contains common type definitions used by C for performing calculations.
`<stdio.h>`	Contains function prototypes for the standard input/output library functions, and information used by them.
`<stdlib.h>`	Contains function prototypes for conversions of numbers to text and text to numbers, memory allocation, random numbers, and other utility functions.
`<string.h>`	Contains function prototypes for string-processing functions.
`<time.h>`	Contains function prototypes and types for manipulating the time and date.

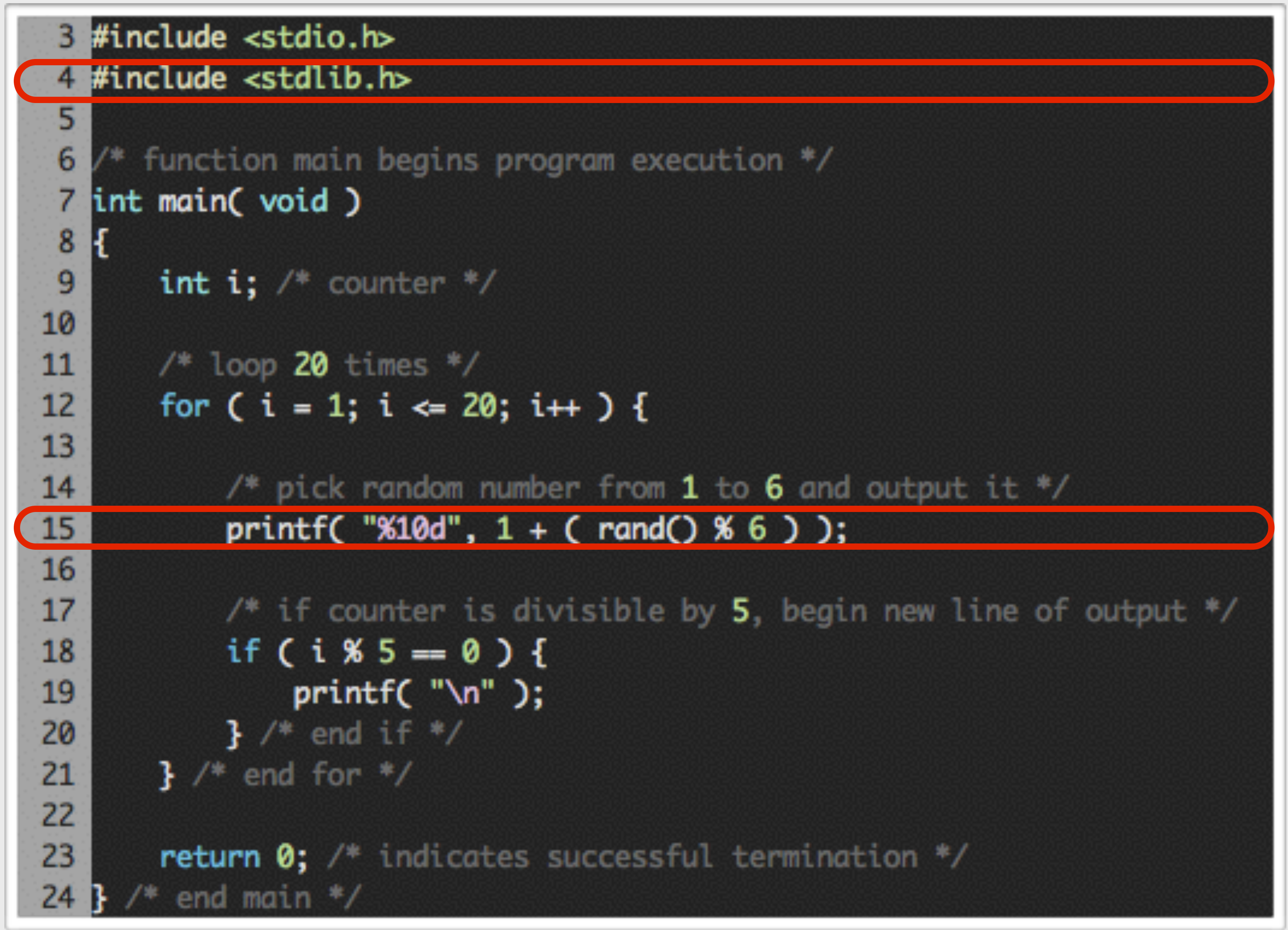
Fig. 5.6 | Some of the standard library headers. (Part 2 of 2.)

Call Functions By Value and By Reference

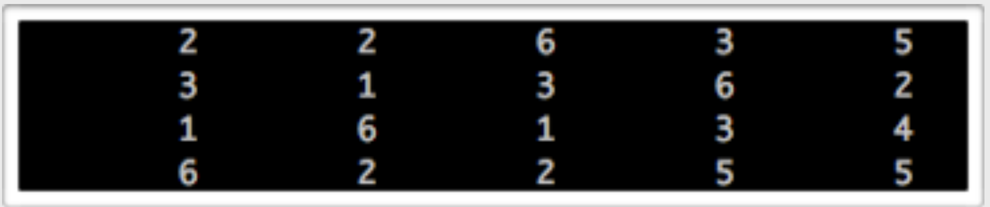
- There are two ways to invoke functions in many programming languages:
- Call-by-Value
  - A copy of the argument's value is made and passed to the called function
  - Changes to the copy do not affect an original variable's value in the caller.
- Call-by-Reference
  - The caller allows the called function to modify the original variable's value.

Random Number Generation

- Example: fig05_07.c



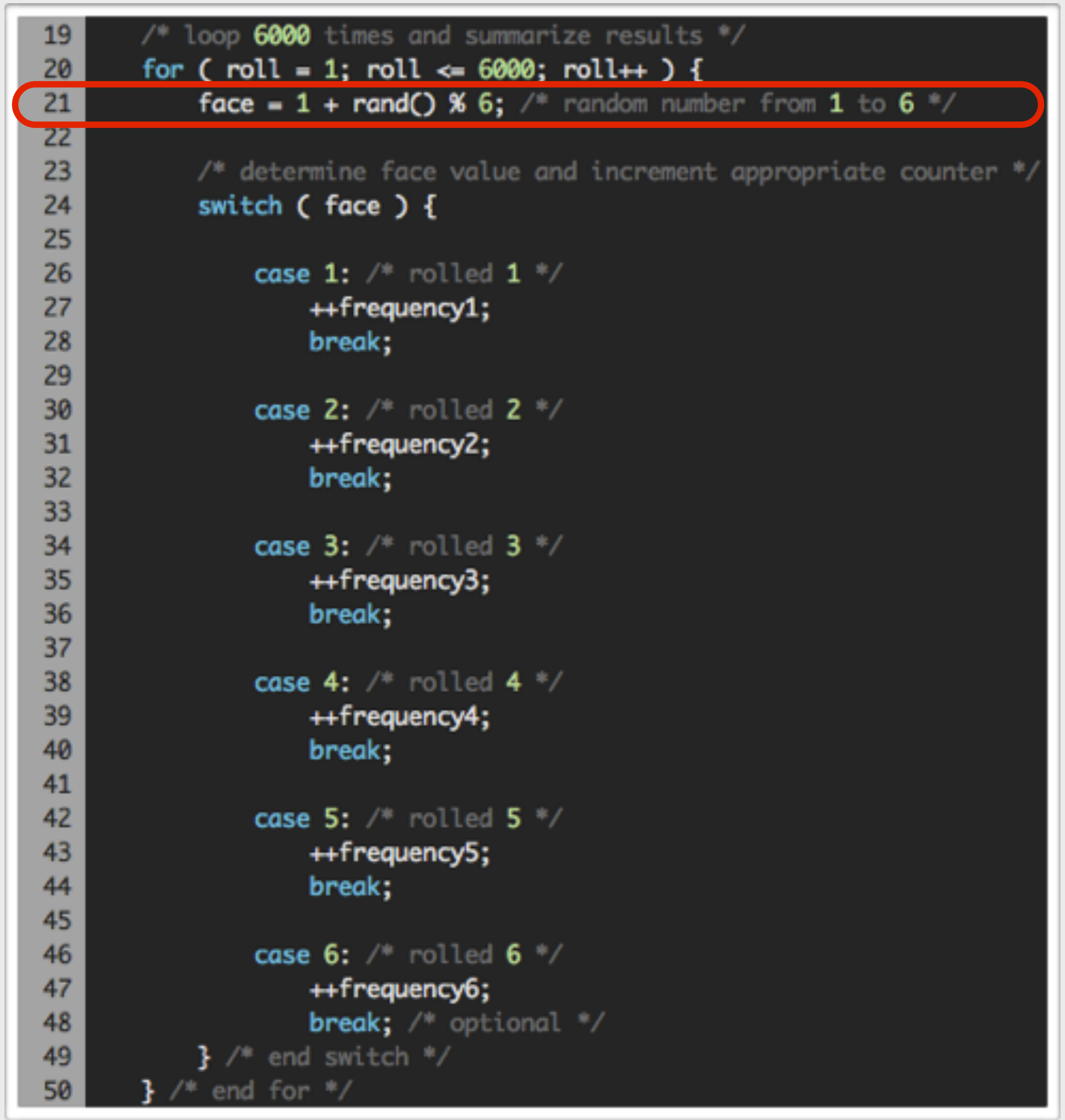
```c
#include <stdio.h>
#include <stdlib.h>

/* function main begins program execution */
int main( void )
{
    int i; /* counter */

    /* loop 20 times */
    for ( i = 1; i <= 20; i++ ) {

        /* pick random number from 1 to 6 and output it */
        printf( "%10d", 1 + ( rand() % 6 ) );

        /* if counter is divisible by 5, begin new line of output */
        if ( i % 5 == 0 ) {
            printf( "\n" );
        } /* end if */
    } /* end for */

    return 0; /* indicates successful termination */
} /* end main */
```



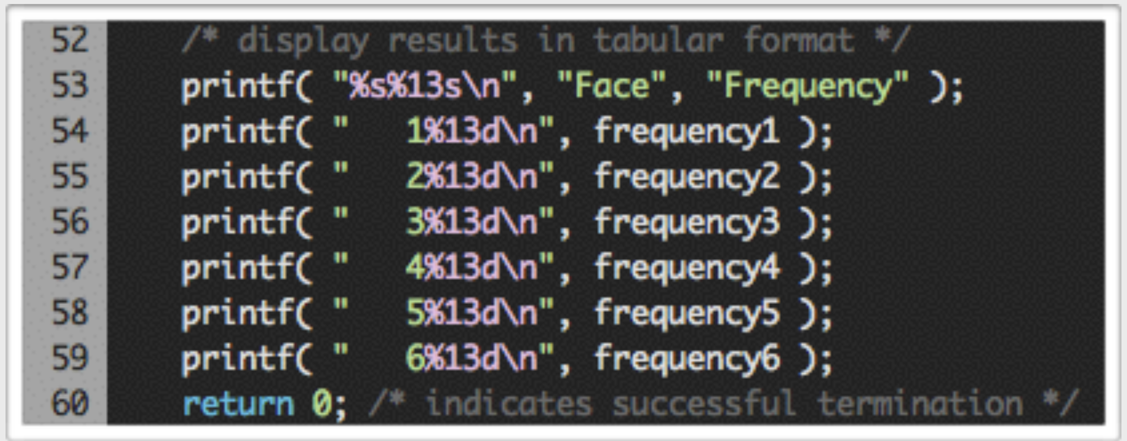
```
         2         2         6         3         5
         3         1         3         6         2
         1         6         1         3         4
         6         2         2         5         5
```

Random Number Generation (Cont.)

- Example: fig05_08.c



```c
    /* loop 6000 times and summarize results */
    for ( roll = 1; roll <= 6000; roll++ ) {
        face = 1 + rand() % 6; /* random number from 1 to 6 */

        /* determine face value and increment appropriate counter */
        switch ( face ) {

            case 1: /* rolled 1 */
                ++frequency1;
                break;

            case 2: /* rolled 2 */
                ++frequency2;
                break;

            case 3: /* rolled 3 */
                ++frequency3;
                break;

            case 4: /* rolled 4 */
                ++frequency4;
                break;

            case 5: /* rolled 5 */
                ++frequency5;
                break;

            case 6: /* rolled 6 */
                ++frequency6;
                break; /* optional */
        } /* end switch */
    } /* end for */
```



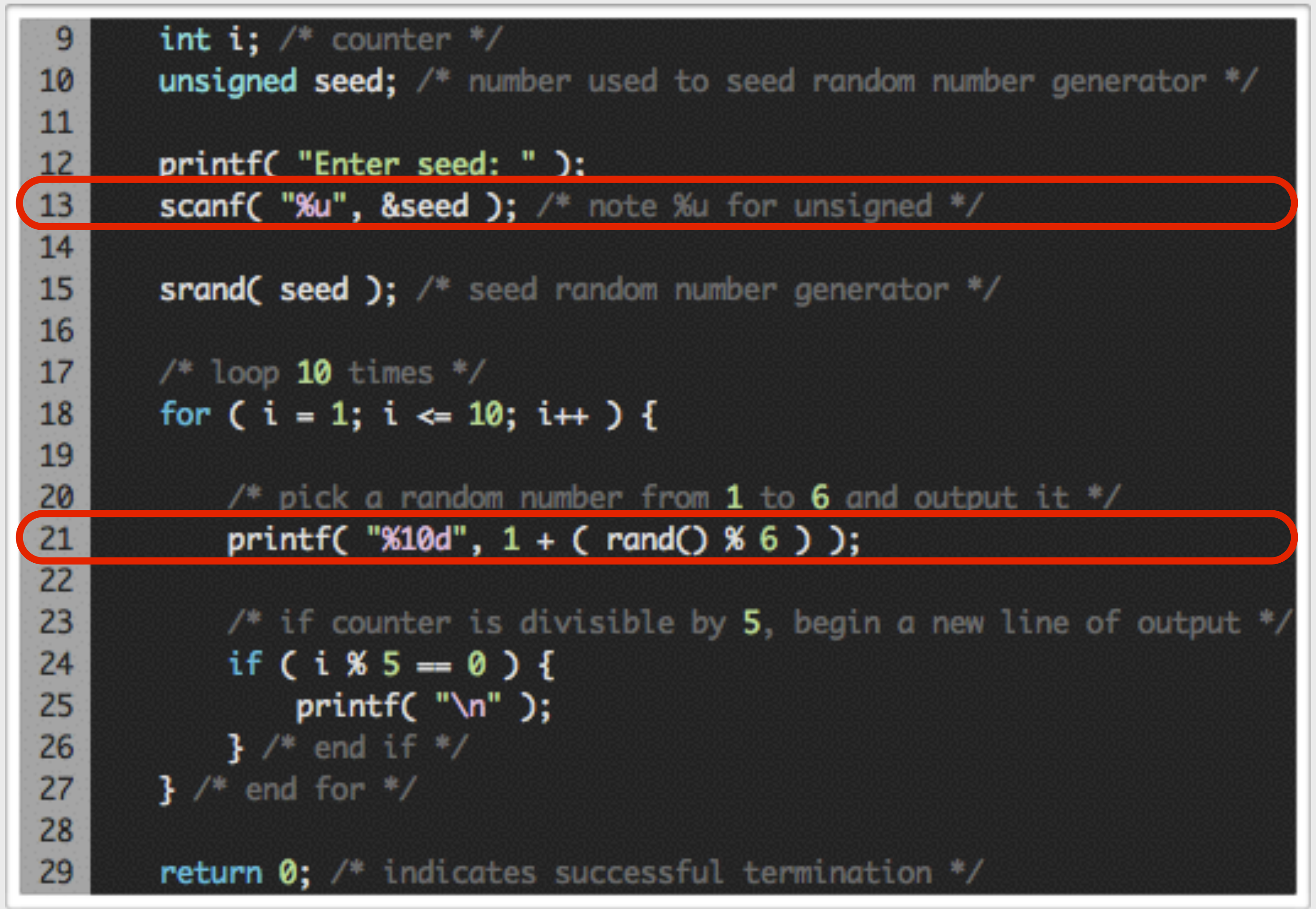
```c
    /* display results in tabular format */
    printf( "%s%13s\n", "Face", "Frequency" );
    printf( "   1%13d\n", frequency1 );
    printf( "   2%13d\n", frequency2 );
    printf( "   3%13d\n", frequency3 );
    printf( "   4%13d\n", frequency4 );
    printf( "   5%13d\n", frequency5 );
    printf( "   6%13d\n", frequency6 );
    return 0; /* indicates successful termination */
```

Random Number Generation (Cont.)

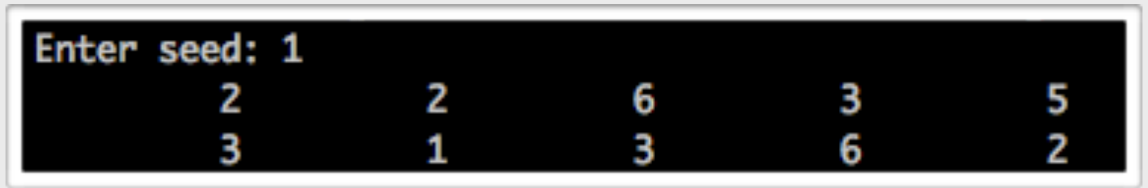
- Executing the program of fig05_07.c again produces exactly the same sequence of values.
- How can these be random numbers? Ironically, this repeatability is an important characteristic of function **`rand`**.
  - Calling **`rand`** repeatedly produces a sequence of numbers that appears to be random
- Another randomization is accomplished by **`srand`**

Random Number Generation (Cont.)

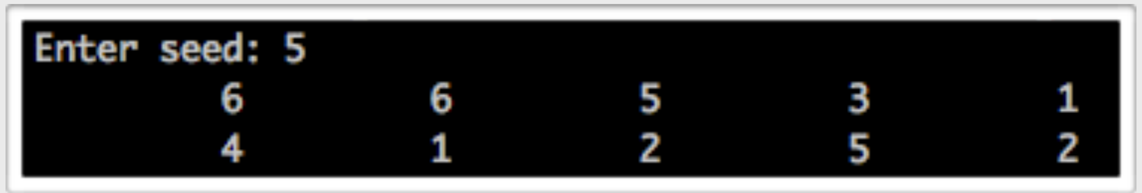
- Example: fig05_09.c



```c
    int i; /* counter */
    unsigned seed; /* number used to seed random number generator */

    printf( "Enter seed: " );
    scanf( "%u", &seed ); /* note %u for unsigned */

    srand( seed ); /* seed random number generator */

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

        /* pick a random number from 1 to 6 and output it */
        printf( "%10d", 1 + ( rand() % 6 ) );

        /* if counter is divisible by 5, begin a new line of output */
        if ( i % 5 == 0 ) {
            printf( "\n" );
        } /* end if */
    } /* end for */

    return 0; /* indicates successful termination */
```



```
Enter seed: 1
         2         2         6         3         5
         3         1         3         6         2
```



```
Enter seed: 5
         6         6         5         3         1
         4         1         2         5         2
```

Random Number Generation (Cont.)

- To randomize without entering a seed each time, use a statement like

```
srand( time( NULL ) );
```

  - This causes the computer to read its clock to obtain the value for the seed automatically.
  - Function time takes **`NULL`** as an argument (time is capable of providing you with a string representing the value it returns; `NULL` disables this capability for a specific call to time).
  - The function prototype for time is in **`<time.h>`**.

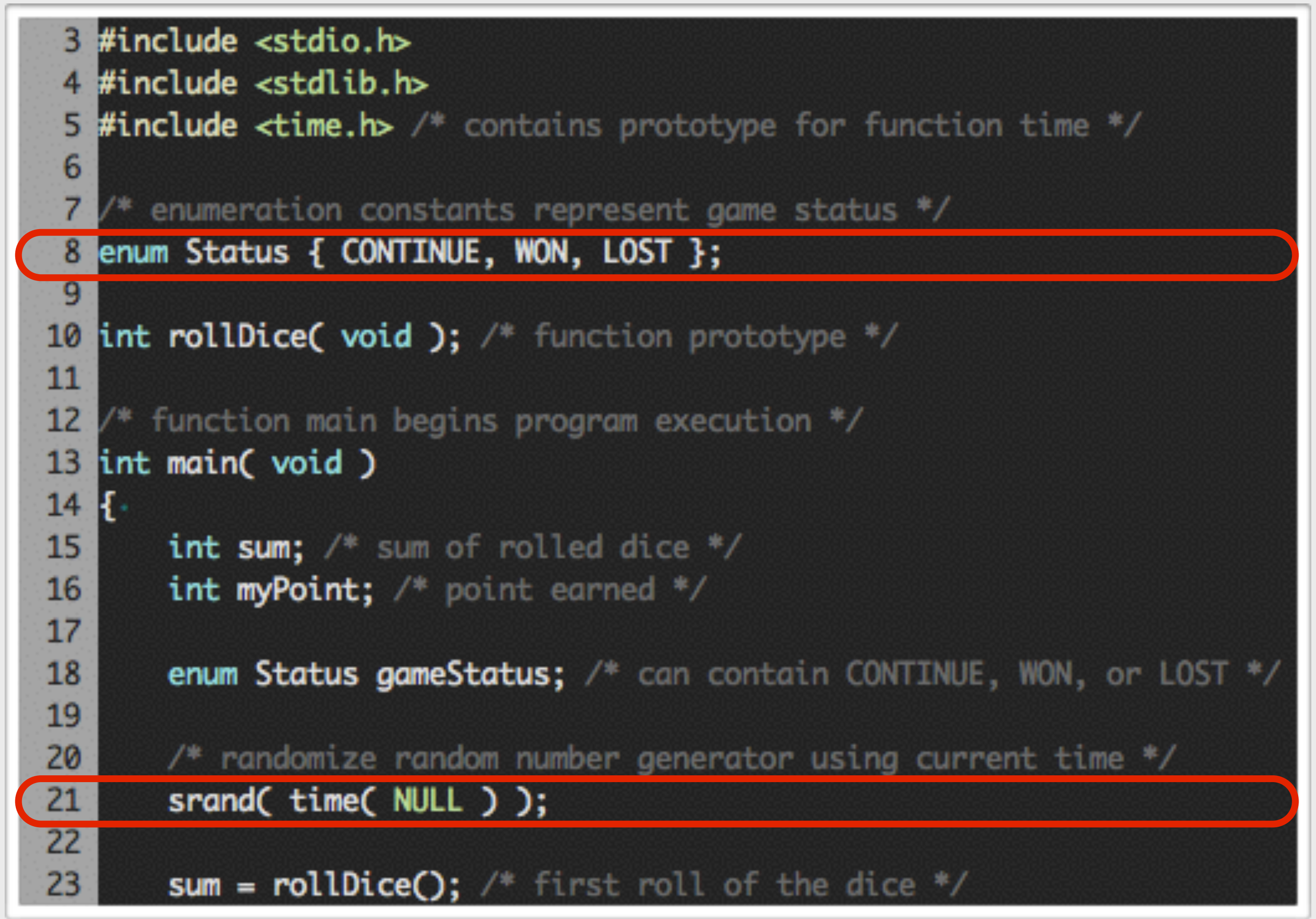
Example: A Game of Chance

- One of the most popular games of chance is a dice game known as "craps*." The rules of the game are simple.
  - A player rolls two dice. Each die has six faces. These faces contain 1, 2, 3, 4, 5, and 6 spots.
  - If the sum is 7 or 11 on the first throw, the player wins.
  - If the sum is 2, 3, or 12 on the first throw, the player loses.
  - If the sum is 4, 5, 6, 8, 9, or 10 on the first throw, then that sum becomes the player's "point."
    - To win, you must continue rolling the dice until you "make your point." The player loses by rolling a 7 before making the point.

* http://en.wikipedia.org/wiki/Craps

Example: A Game of Chance (Cont.)

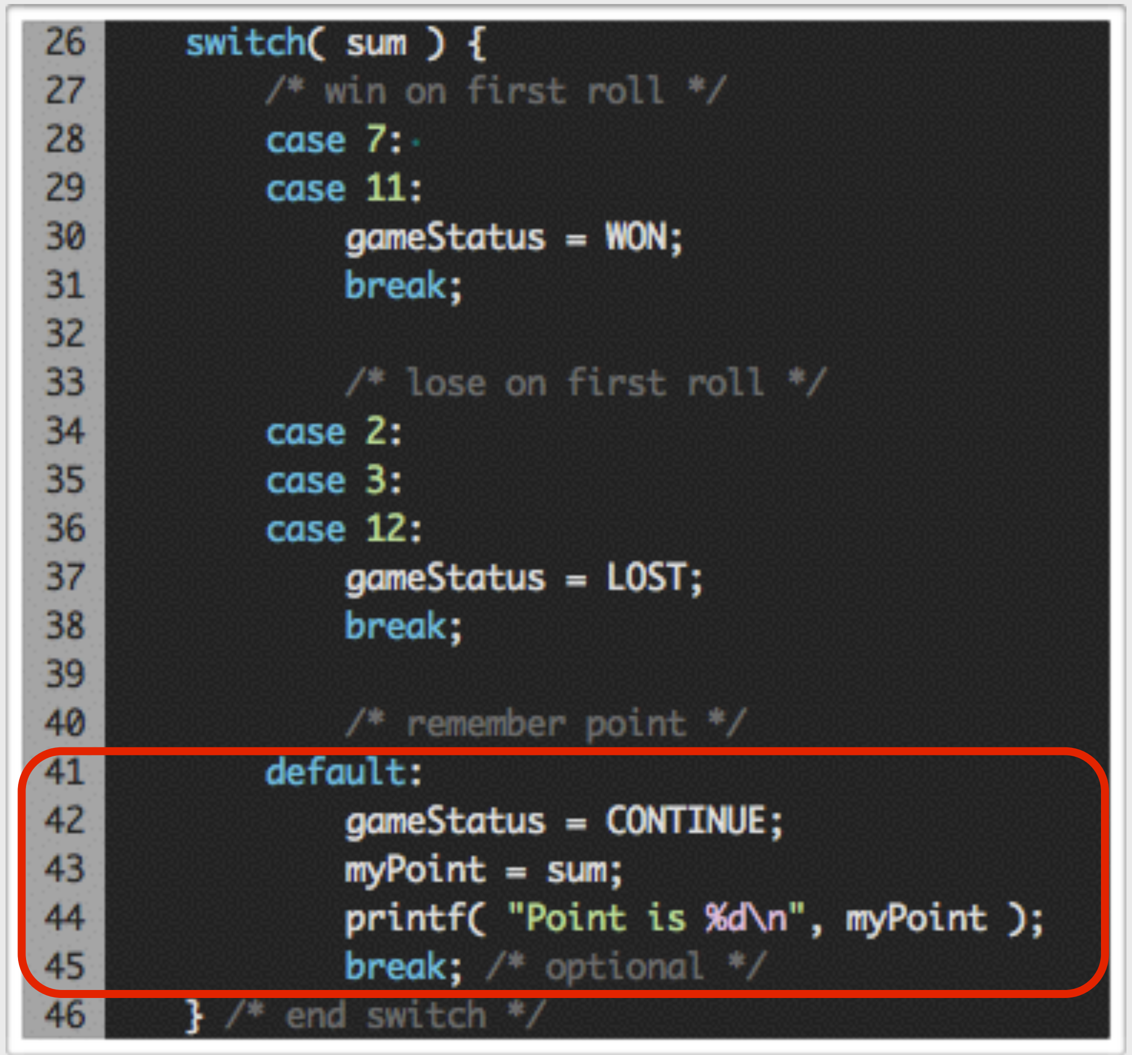
- Example: fig05_10.c



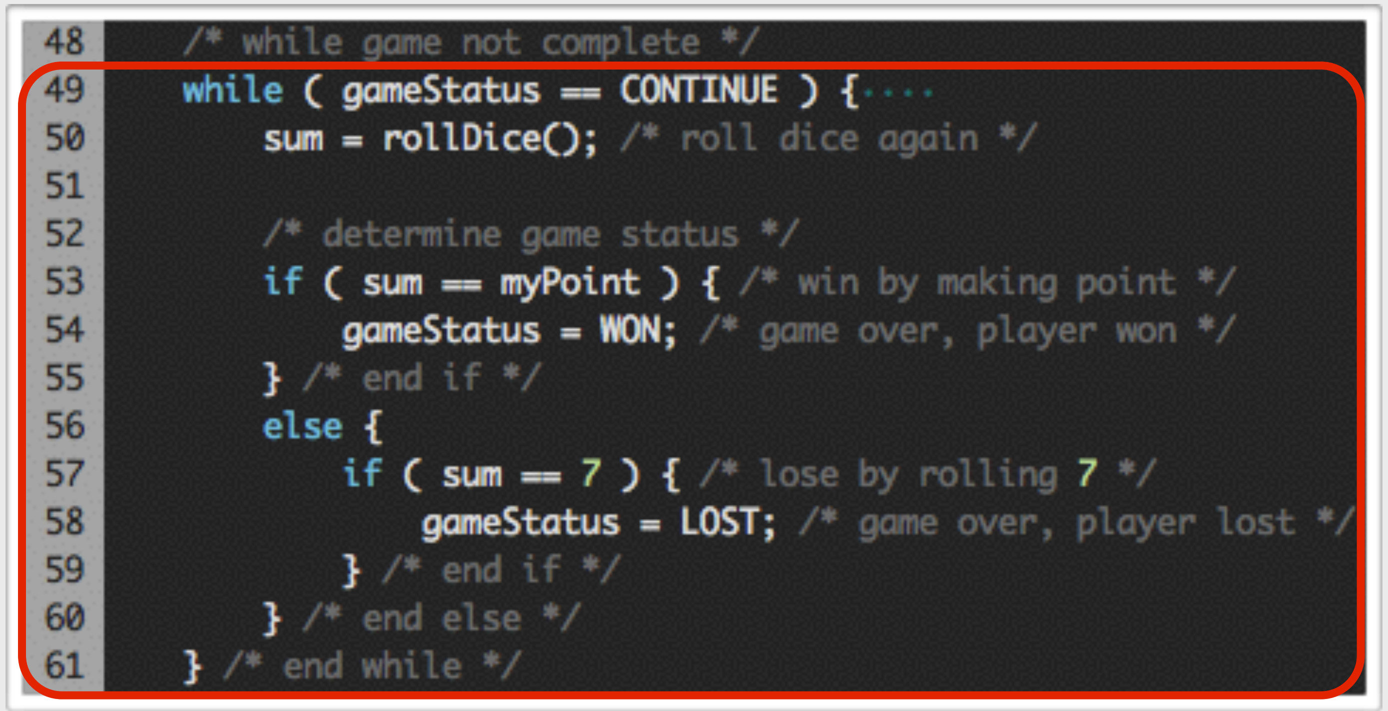
```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h> /* contains prototype for function time */

/* enumeration constants represent game status */
enum Status { CONTINUE, WON, LOST };

int rollDice( void ); /* function prototype */

/* function main begins program execution */
int main( void )
{
    int sum; /* sum of rolled dice */
    int myPoint; /* point earned */

    enum Status gameStatus; /* can contain CONTINUE, WON, or LOST */

    /* randomize random number generator using current time */
    srand( time( NULL ) );

    sum = rollDice(); /* first roll of the dice */
```

Example: A Game of Chance (Cont.)

- Example: fig05_10.c



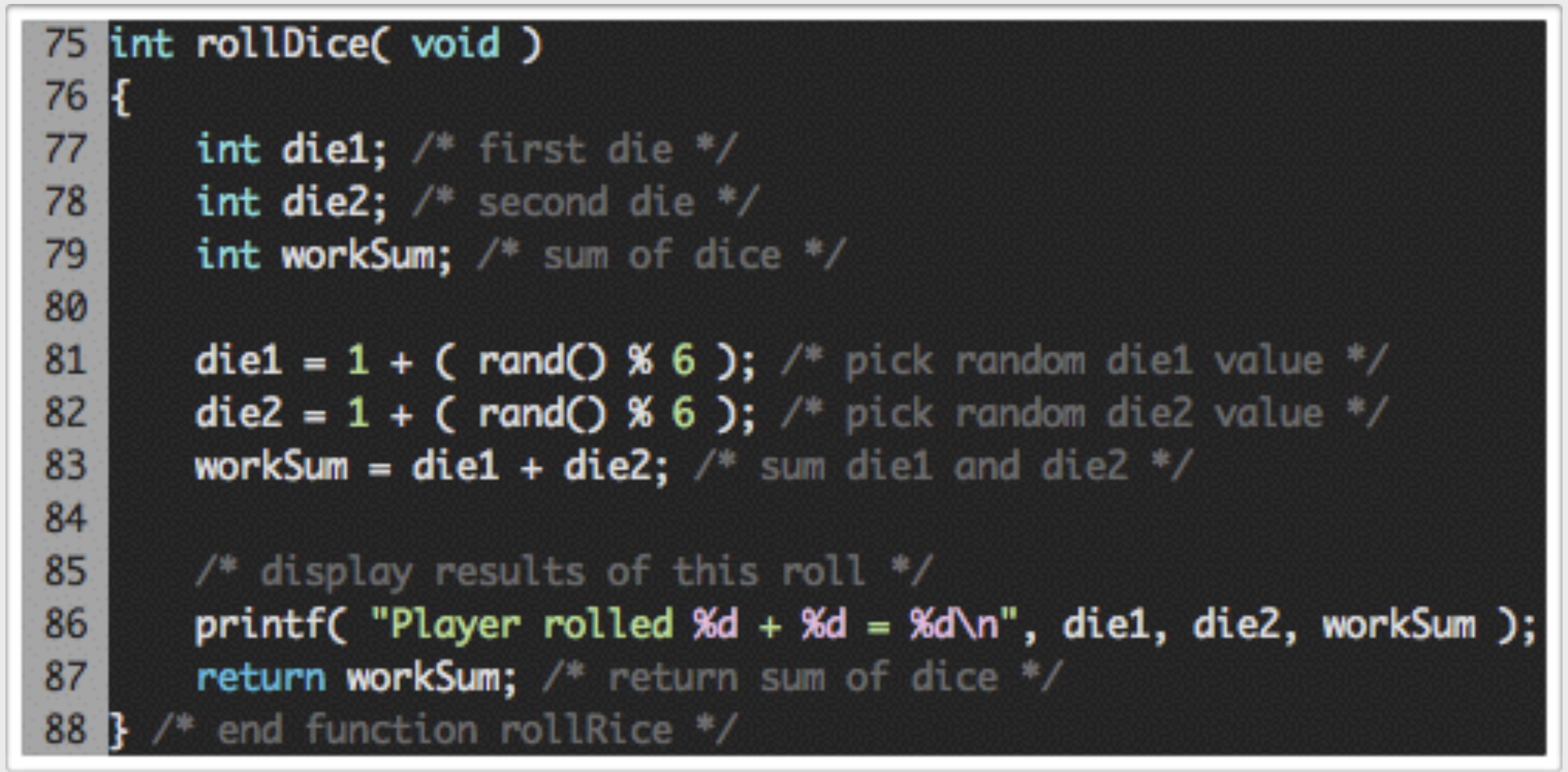
```c
   switch( sum ) {
      /* win on first roll */
      case 7:
      case 11:
         gameStatus = WON;
         break;

         /* lose on first roll */
      case 2:
      case 3:
      case 12:
         gameStatus = LOST;
         break;

         /* remember point */
      default:
         gameStatus = CONTINUE;
         myPoint = sum;
         printf( "Point is %d\n", myPoint );
         break; /* optional */
   } /* end switch */
```



```c
   /* while game not complete */
   while ( gameStatus == CONTINUE ) {
      sum = rollDice(); /* roll dice again */

      /* determine game status */
      if ( sum == myPoint ) { /* win by making point */
         gameStatus = WON; /* game over, player won */
      } /* end if */
      else {
         if ( sum == 7 ) { /* lose by rolling 7 */
            gameStatus = LOST; /* game over, player lost */
         } /* end if */
      } /* end else */
   } /* end while */
```

Example: A Game of Chance (Cont.)

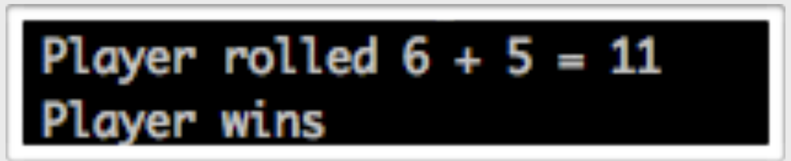
- Example: fig05_10.c



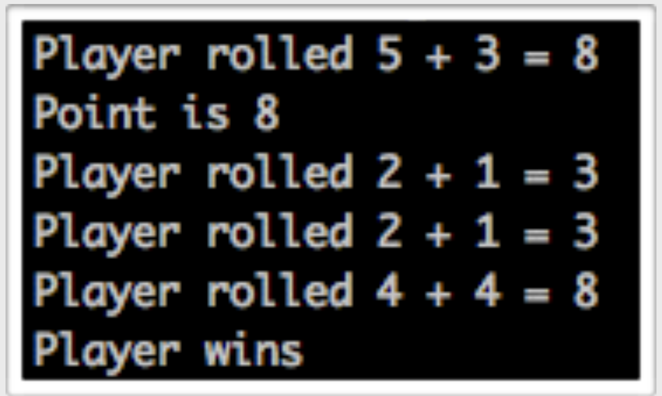
```
int rollDice( void )
{
    int die1; /* first die */
    int die2; /* second die */
    int workSum; /* sum of dice */

    die1 = 1 + ( rand() % 6 ); /* pick random die1 value */
    die2 = 1 + ( rand() % 6 ); /* pick random die2 value */
    workSum = die1 + die2; /* sum die1 and die2 */

    /* display results of this roll */
    printf( "Player rolled %d + %d = %d\n", die1, die2, workSum );
    return workSum; /* return sum of dice */
} /* end function rollRice */
```

Example: A Game of Chance (Cont.)

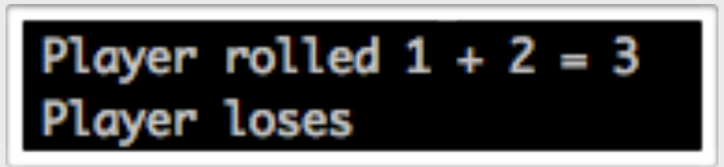
- Example: fig05_10.c



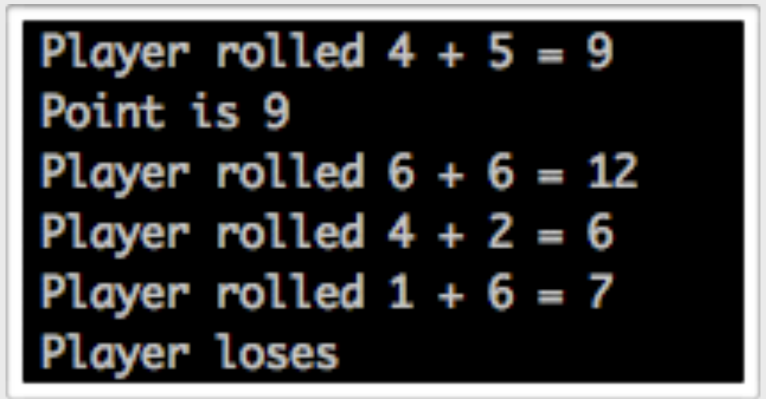
```
Player rolled 6 + 5 = 11
Player wins
```



```
Player rolled 5 + 3 = 8
Point is 8
Player rolled 2 + 1 = 3
Player rolled 2 + 1 = 3
Player rolled 4 + 4 = 8
Player wins
```



```
Player rolled 1 + 2 = 3
Player loses
```



```
Player rolled 4 + 5 = 9
Point is 9
Player rolled 6 + 6 = 12
Player rolled 4 + 2 = 6
Player rolled 1 + 6 = 7
Player loses
```

Example: A Game of Chance (Cont.)

- An enumeration, introduced by the keyword **`enum`**, is a set of integer constants represented by identifiers.
  - Enumeration constants are sometimes called symbolic constants.
  - The constant **`CONTINUE`** has the value 0, **`WON`** has the value 1 and **`LOST`** has the value 2.

Scope Rules

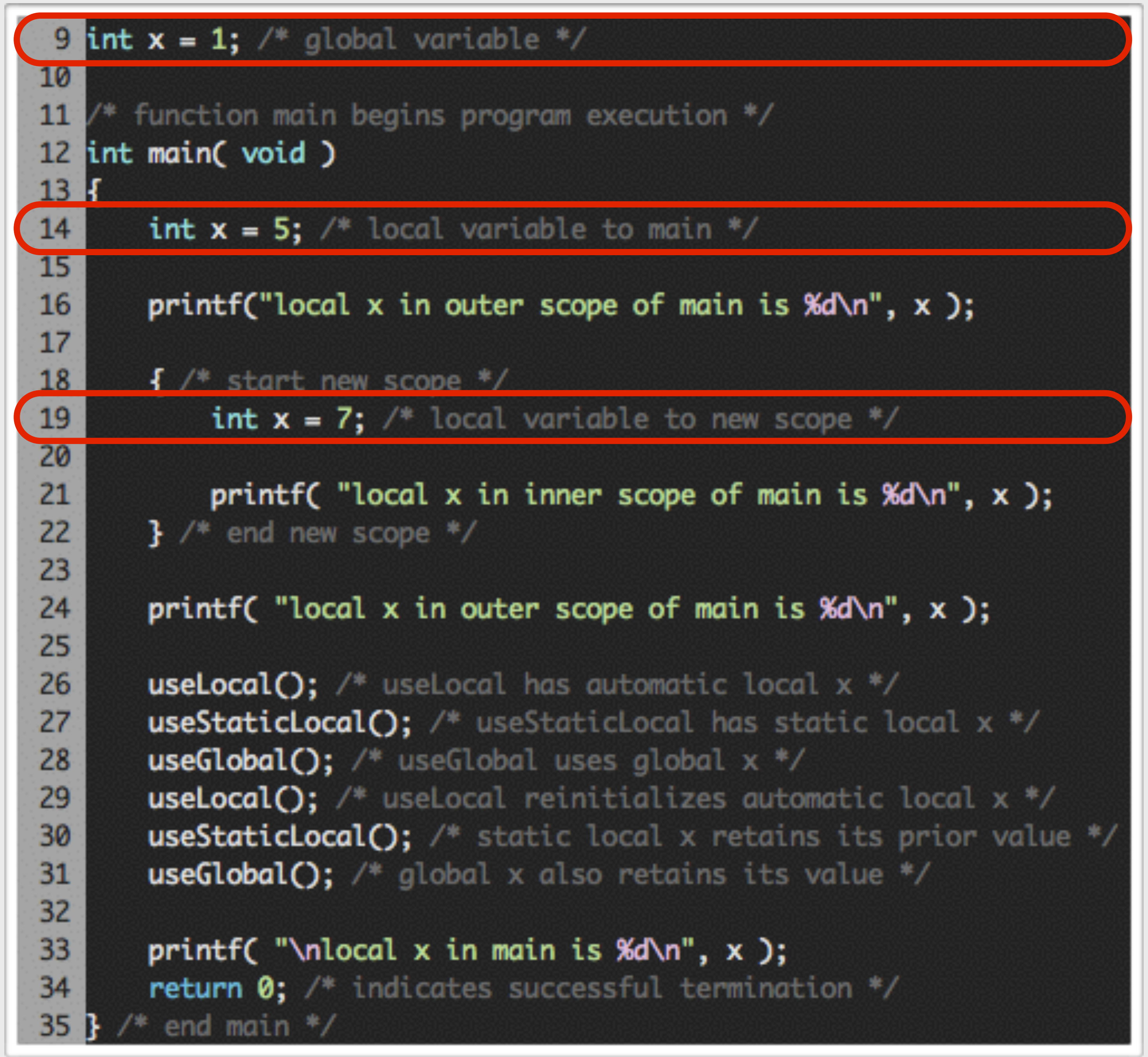
- The four identifier scopes are
    - function scope
    - file scope
    - block scope
    - function-prototype scope

Scope Rules (Cont.)

- Labels (an identifier followed by a colon such as **`start:`**) are the only identifiers with function scope.
- An identifier declared outside any function has file scope.
- Identifiers defined inside a block ({}) have block/local scope.
- The only identifiers with function-prototype scope are those used in the parameter list of a function prototype.

Scope Rules (Cont.)

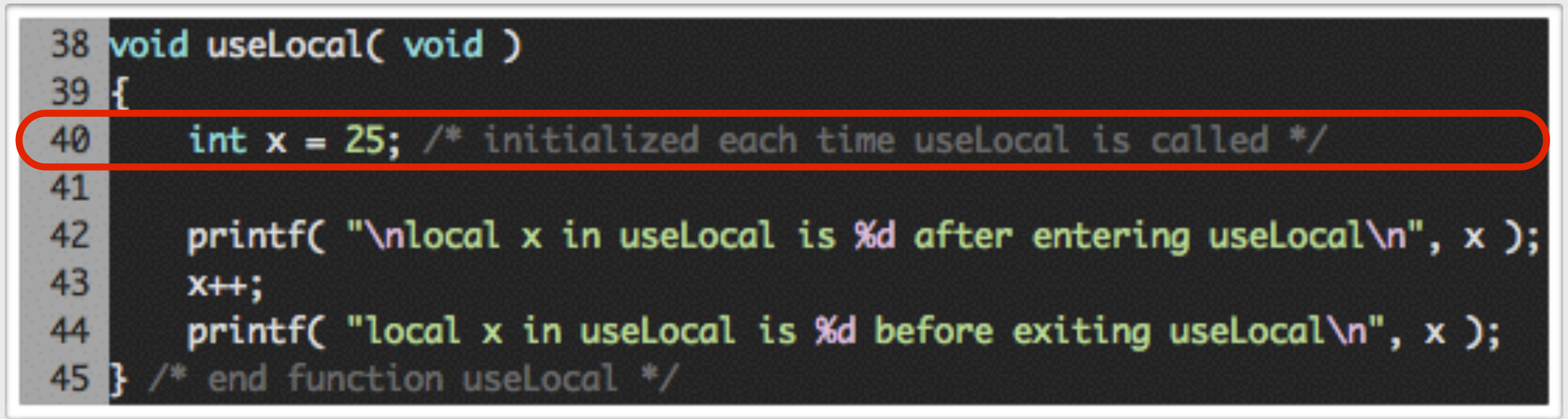
- Example: fig05_12.c



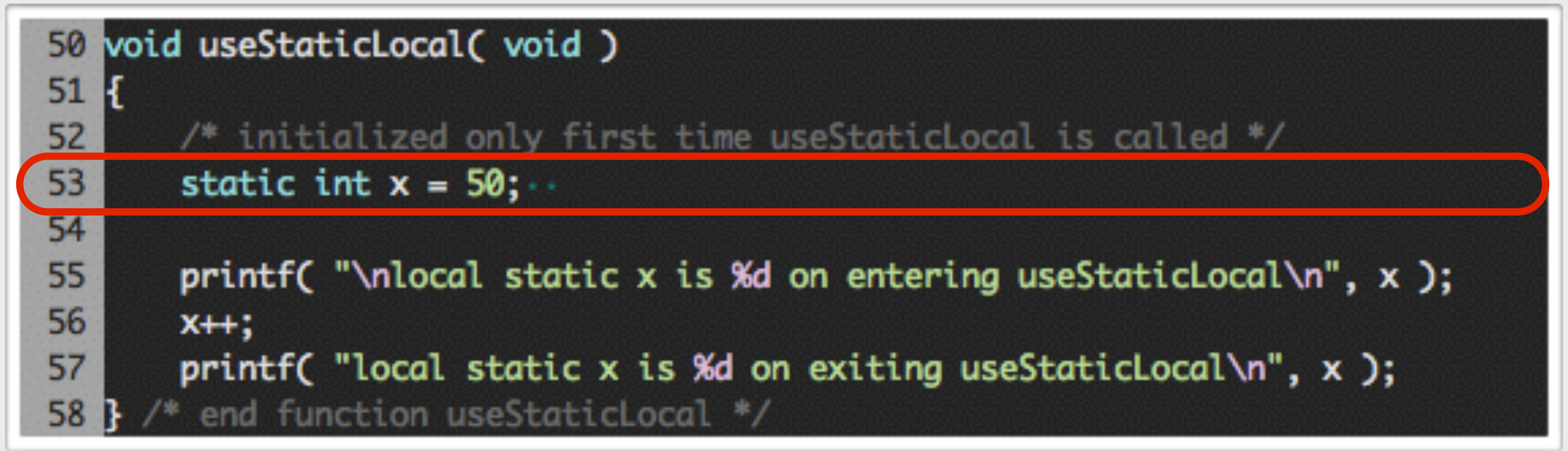
```
int x = 1; /* global variable */

/* function main begins program execution */
int main( void )
{
    int x = 5; /* local variable to main */

    printf("local x in outer scope of main is %d\n", x );

    { /* start new scope */
        int x = 7; /* local variable to new scope */

        printf( "local x in inner scope of main is %d\n", x );
    } /* end new scope */

    printf( "local x in outer scope of main is %d\n", x );

    useLocal(); /* useLocal has automatic local x */
    useStaticLocal(); /* useStaticLocal has static local x */
    useGlobal(); /* useGlobal uses global x */
    useLocal(); /* useLocal reinitializes automatic local x */
    useStaticLocal(); /* static local x retains its prior value */
    useGlobal(); /* global x also retains its value */

    printf( "\nlocal x in main is %d\n", x );
    return 0; /* indicates successful termination */
} /* end main */
```

Scope Rules (Cont.)

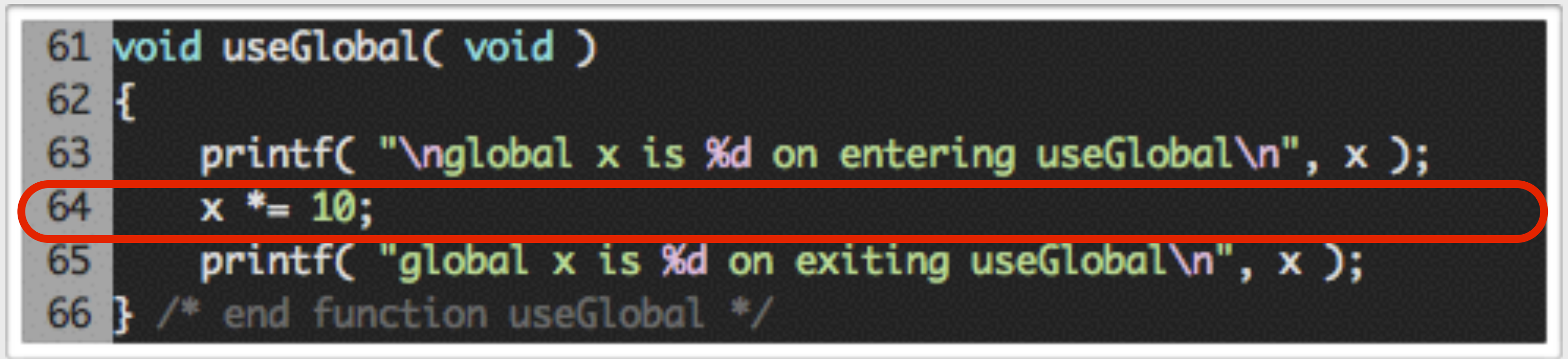
- Example: fig05_12.c



```c
void useLocal( void )
{
    int x = 25; /* initialized each time useLocal is called */

    printf( "\nlocal x in useLocal is %d after entering useLocal\n", x );
    x++;
    printf( "local x in useLocal is %d before exiting useLocal\n", x );
} /* end function useLocal */
```



```c
void useStaticLocal( void )
{
    /* initialized only first time useStaticLocal is called */
    static int x = 50;

    printf( "\nlocal static x is %d on entering useStaticLocal\n", x );
    x++;
    printf( "local static x is %d on exiting useStaticLocal\n", x );
} /* end function useStaticLocal */
```



```c
void useGlobal( void )
{
    printf( "\nglobal x is %d on entering useGlobal\n", x );
    x *= 10;
    printf( "global x is %d on exiting useGlobal\n", x );
} /* end function useGlobal */
```

Storage Classes

- The storage-class specifiers can be split into two storage classes: automatic storage class and static storage class.
    - Automatic: created, and initialized, each time the block is entered; destroyed when the block is exited.
    - Static: stored at a fixed memory location, created and initialized once when the program is first started.

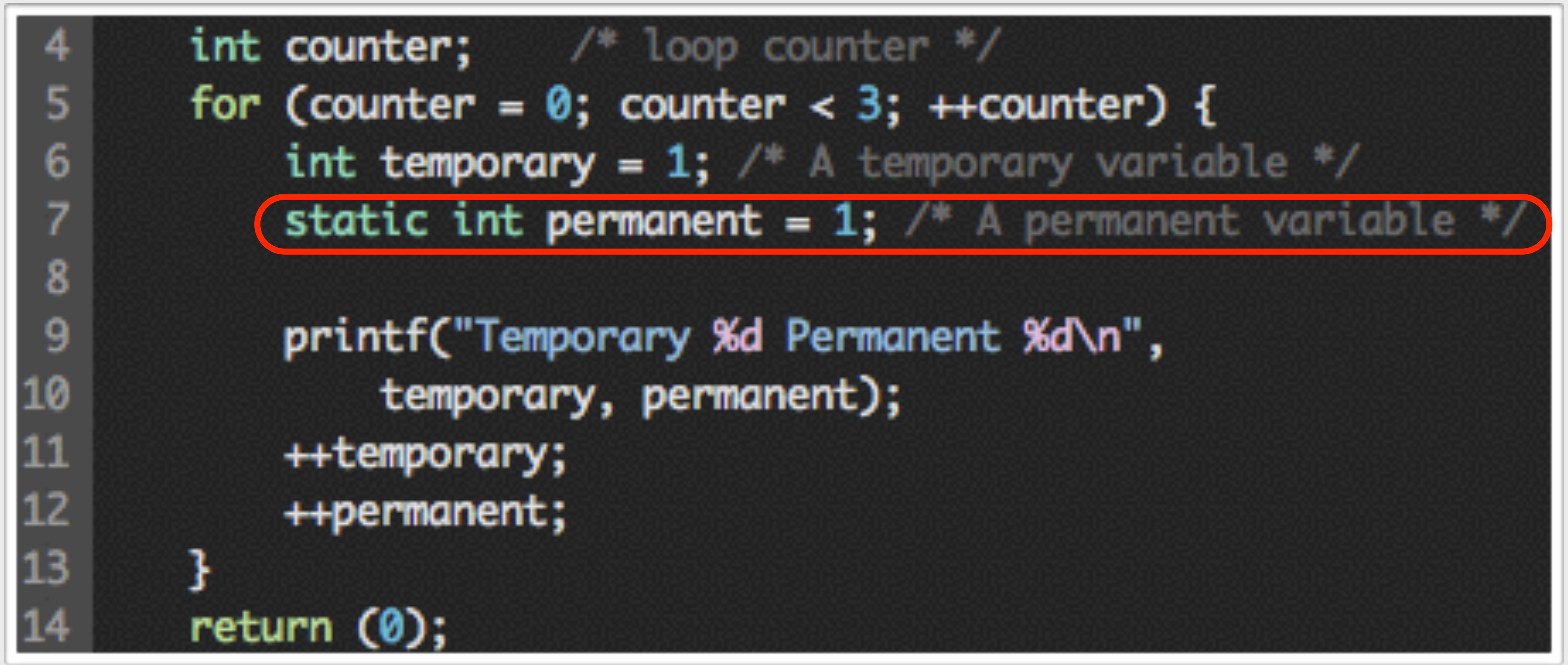
Storage Classes (cont.)

- Automatic storage class
  - Keywords **`auto`** and **`register`** are used to declare variables of the automatic storage class.
  - Such variables are created when program execution enters the block in which they are defined.
  - They exist while the block is active and they are destroyed when the program exits the block.
  - Only local variables of a function can be of automatic storage class.
    - Local variables are of automatic storage class by default.

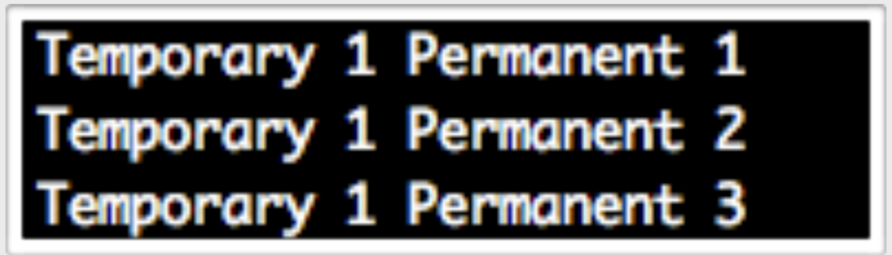
Storage Classes (cont.)

- Static Storage Class
  - Keywords **`extern`** and **`static`** declare identifiers for variables of the static storage class and for functions.
  - static storage class external identifiers (such as global variables)
  - local variables declared with the storage class specifier **`static`**

Storage Classes (cont.)



```c
    int counter;    /* loop counter */
    for (counter = 0; counter < 3; ++counter) {
        int temporary = 1; /* A temporary variable */
        static int permanent = 1; /* A permanent variable */

        printf("Temporary %d Permanent %d\n",
            temporary, permanent);
        ++temporary;
        ++permanent;
    }
    return (0);
```



```
Temporary 1 Permanent 1
Temporary 1 Permanent 2
Temporary 1 Permanent 3
```

Storage Classes (cont.)

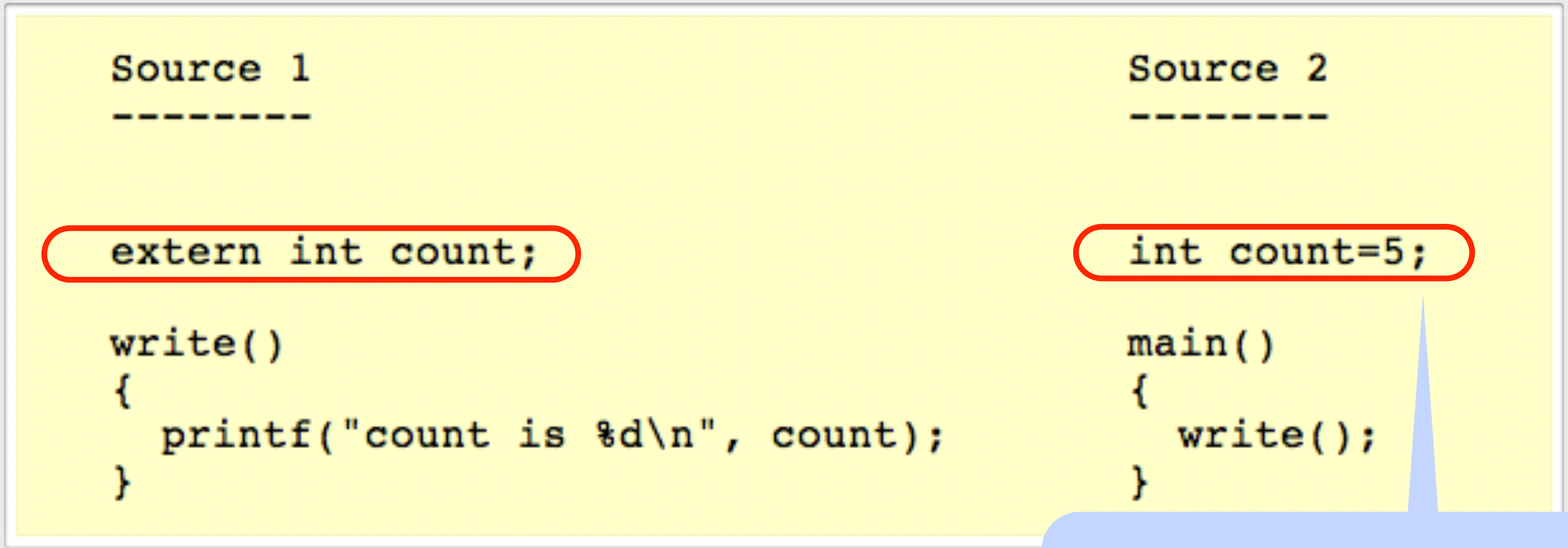
- A **`static`** declaration made outside blocks indicates the variable is local to the file in which it is declared.
- The global variable declaration

```
static double pi = 3.14159;
```

- The above indicates that **`pi`** is known only to functions in the file in which it is defined.

Storage Classes (cont.)

- Example: s1.c, s2.c



```
Source 1                                      Source 2
---------                                     ---------


extern int count;                             int count=5;

write()                                       main()
{                                             {
  printf("count is %d\n", count);               write();
}                                             }
```

What will happen if this becomes
`static int count =5;`

Storage Classes (cont.)

in/out block	keyword	natural scope	lifetime
local	`(auto) int a;`	block scope	to block end
	`static int a;`	block scope	whole program
global	`int a;`	file scope	whole program
	`static int a;`	file scope (strictly)	whole program