Computer Programming I

Ming-Feng Tsai (Victor Tsai)

Dept. of Computer Science
National Chengchi University

C Functions

Objectives

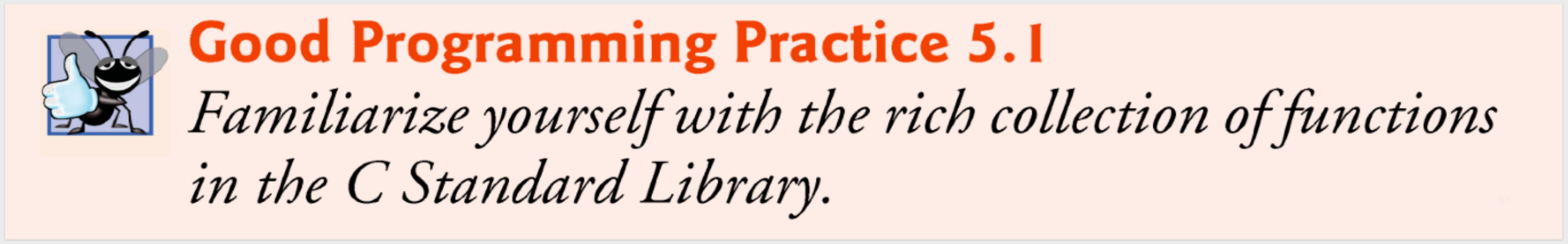
- In this chapter, you'll learn
  - To construct programs modularly from small pieces called functions
  - Common math functions in the C Standard Library
  - To create new functions
  - The mechanisms used to pass information between functions
  - How the function call/return mechanism is supported by the function call stack and activation records
  - Simulation techniques using random number generation
  - How to write and use functions that call themselves

Introduction

- Most computer programs that solve real-world problems are much larger than the programs presented in the first few chapters.
- 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.**
- This chapter describes the features of the C language that facilitate the design, implementation, operation and maintenance of large programs.

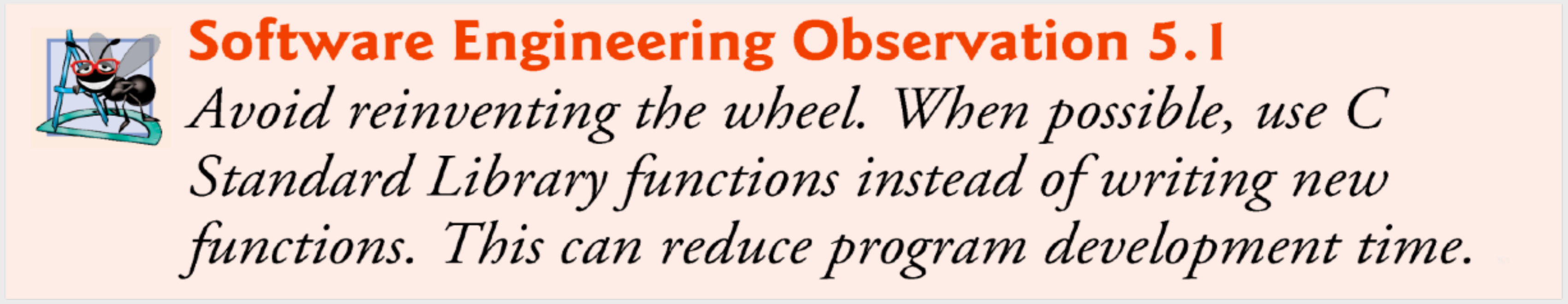
Program Modules in C

- 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.



Good Programming Practice 5.1

Familiarize yourself with the rich collection of functions in the C Standard Library.



Software Engineering Observation 5.1

Avoid reinventing the wheel. When possible, use C Standard Library functions instead of writing new functions. This can reduce program development time.

Program Modules in C (Cont.)

- The functions **`printf`**, **`scanf`** and **`pow`** that we’ve used in previous chapters 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**.

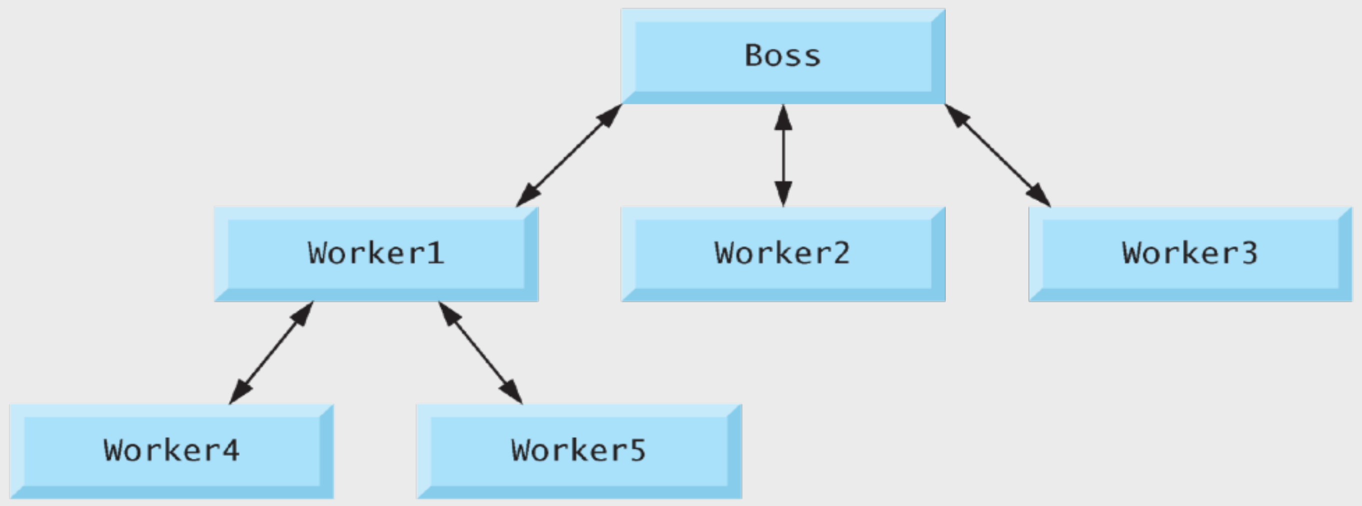


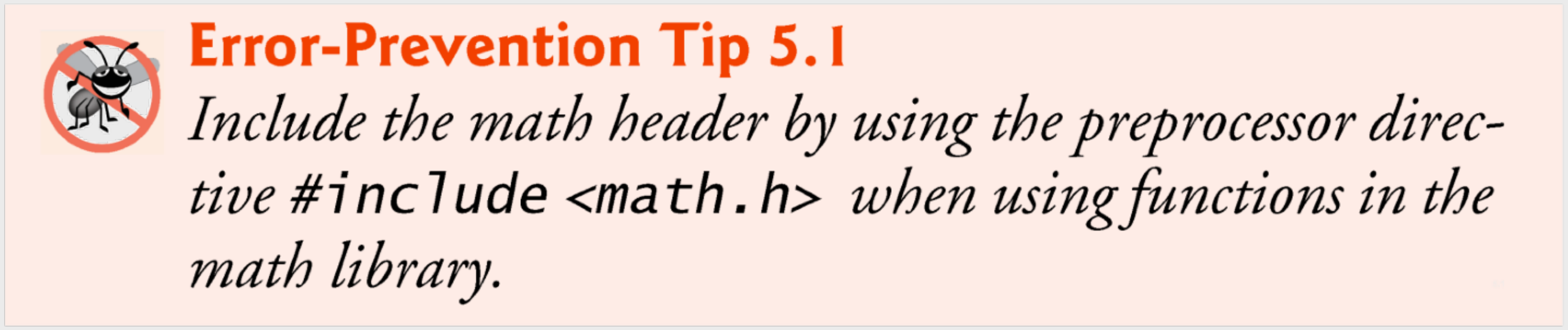
Fig. 5.1 | Hierarchical boss function/worker function relationship.

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

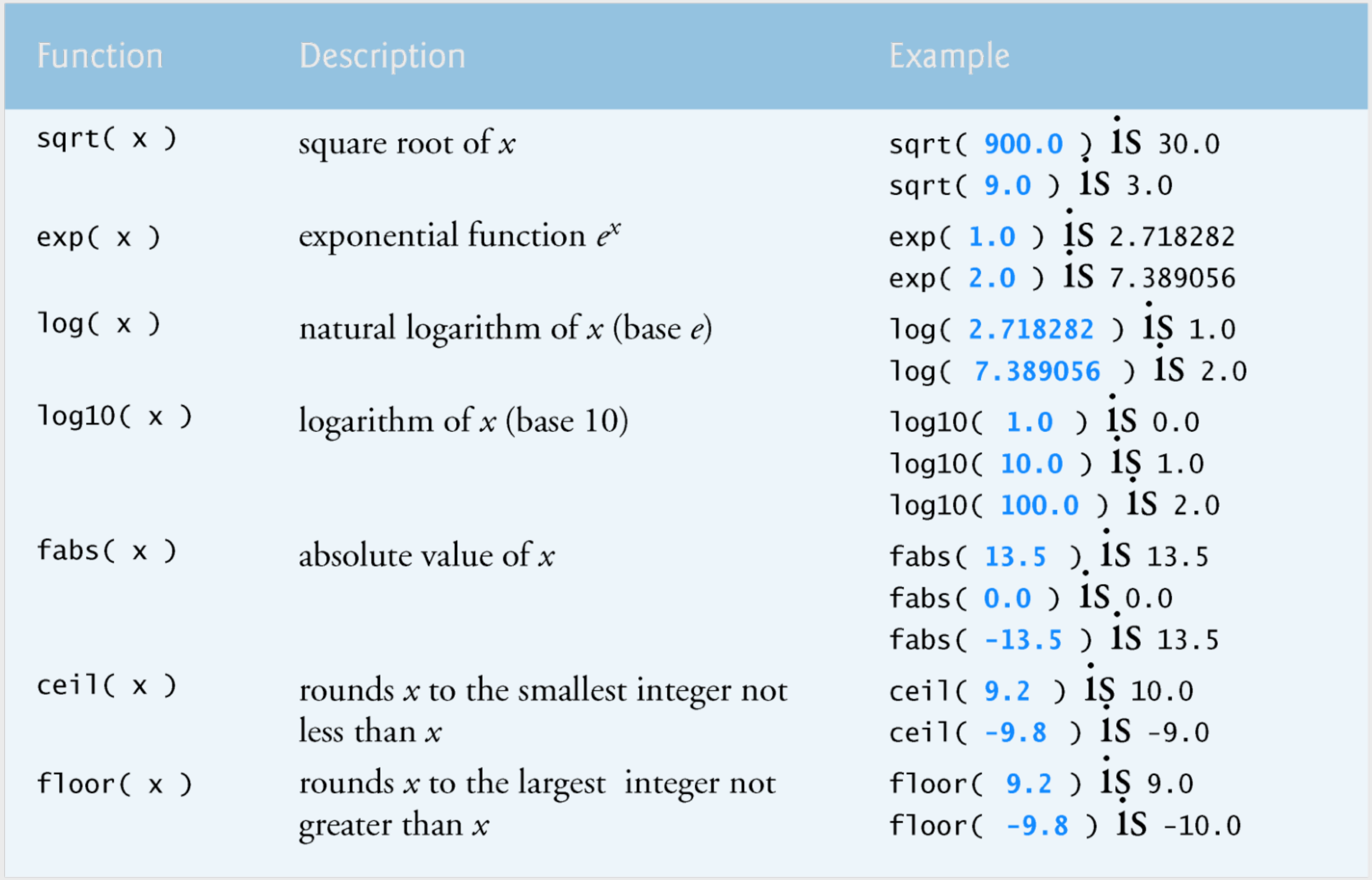
Include the math header by using the preprocessor directive `#include <math.h>` *when using functions in the math library.*

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`

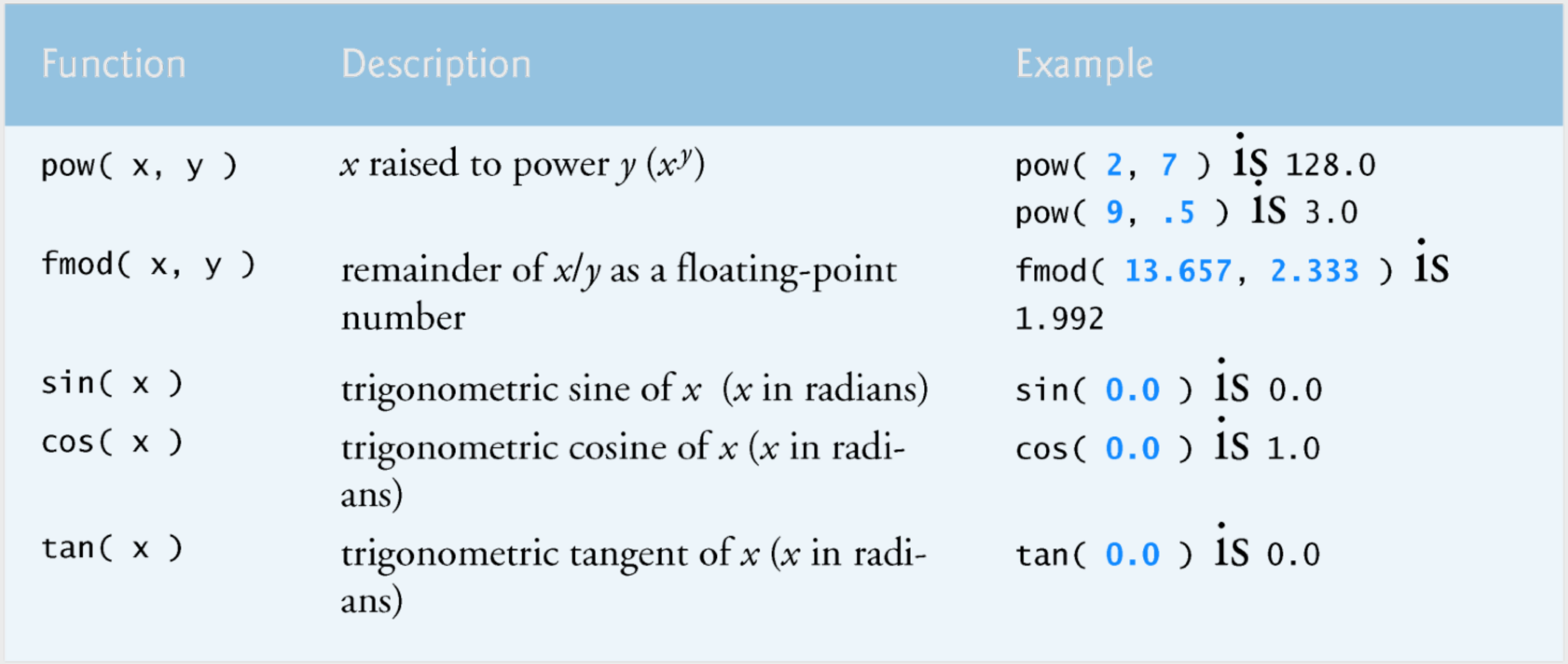
`fabs( x )`	absolute value of *x*	`fabs( 13.5 )` is `13.5`
		`fabs( 0.0 )` is `0.0`
		`fabs( -13.5 )` is `13.5`

`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.

Functions (Cont.)

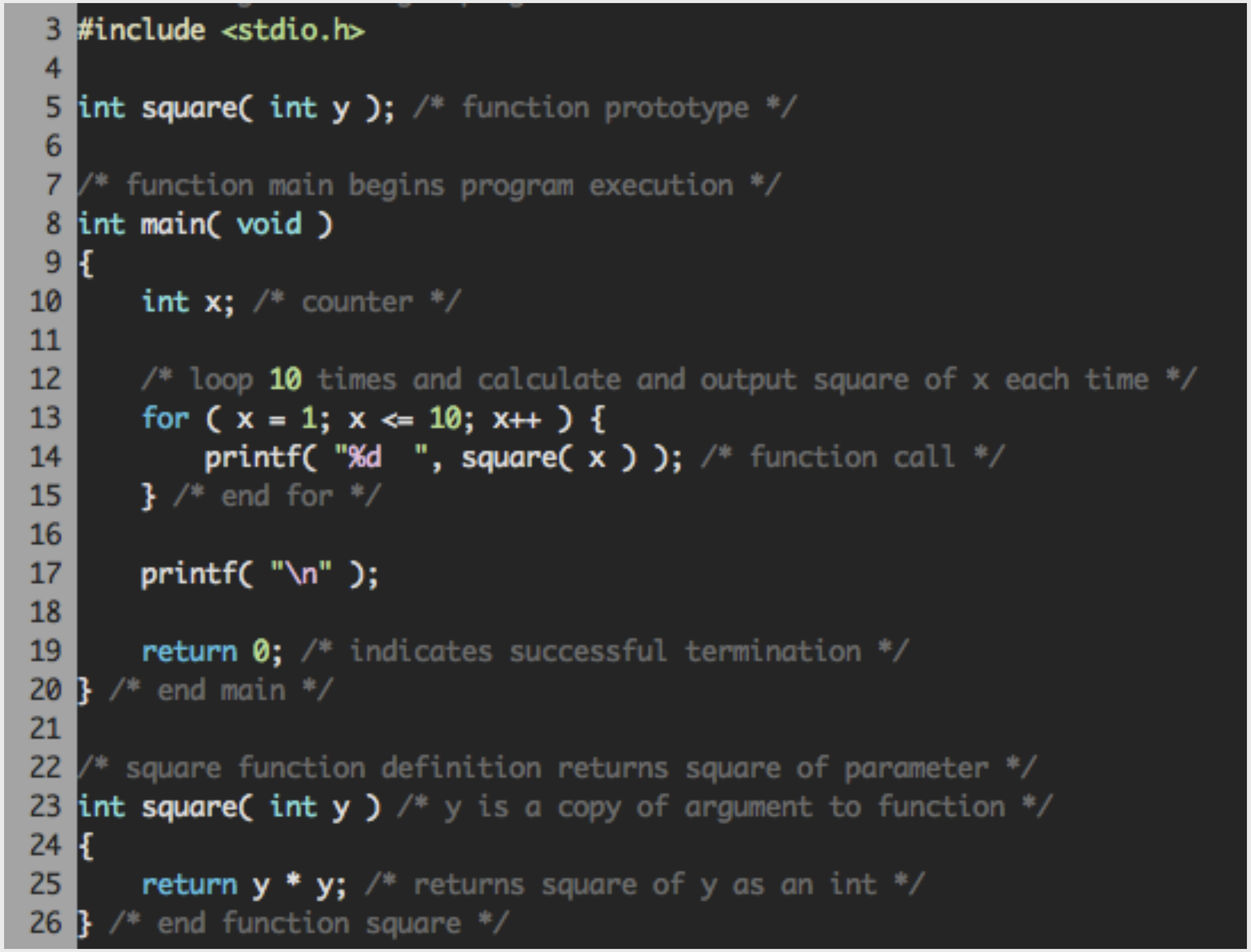
- There are several motivations for “functionalizing” a program.
  - The divide-and-conquer approach makes program development more manageable.
  - Another motivation is software reusability—using existing functions as building-blocks to create new programs.
  - We use abstraction each time we use standard library functions like **`printf`**, **`scanf`** and **`pow`**.
  - Avoid repeating codes in a program.

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.)

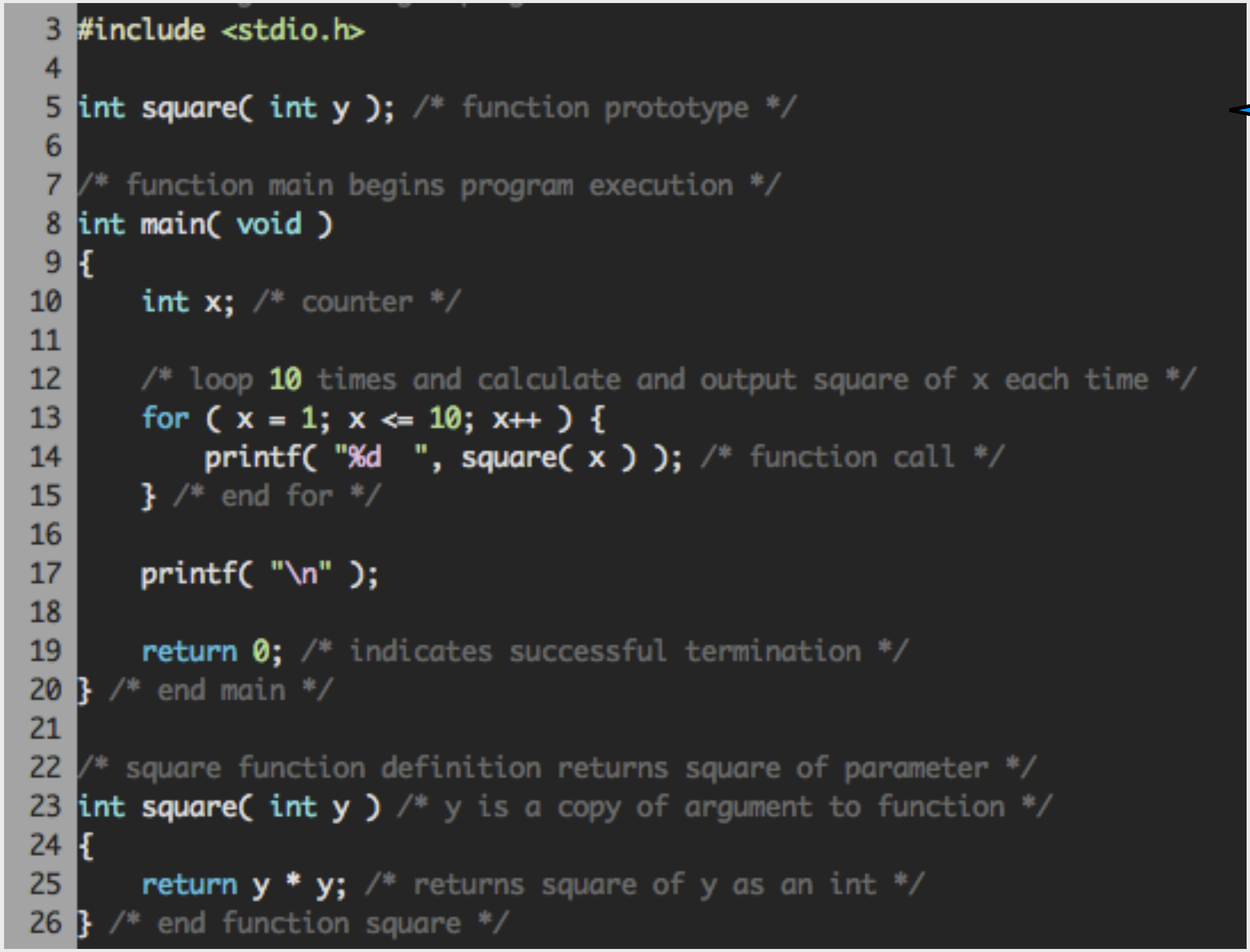
- Example: fig05_03.c



```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 Definitions (Cont.)

- Example: fig05_03.c

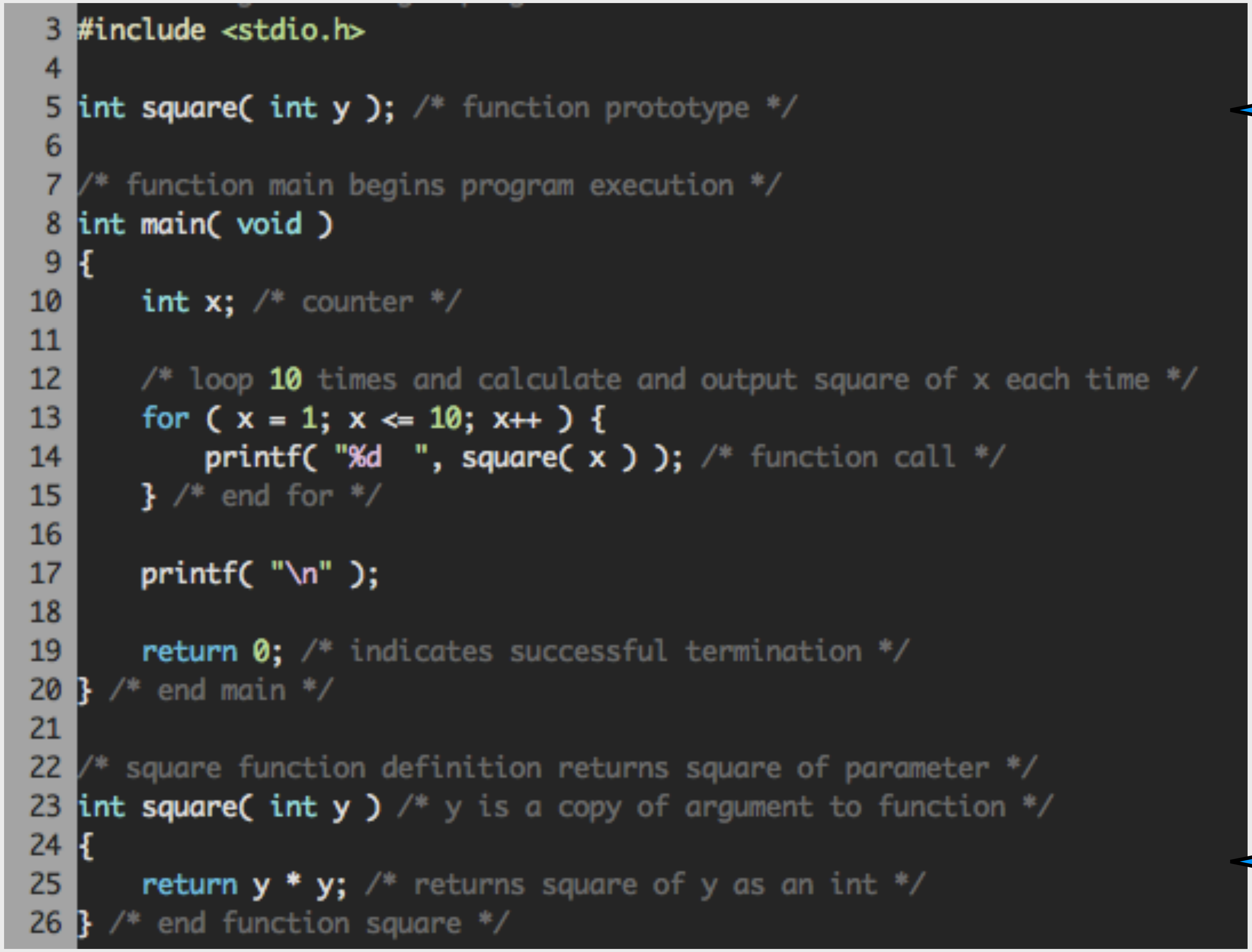


```
#include <stdio.h>

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 Definitions (Cont.)

- Example: fig05_03.c



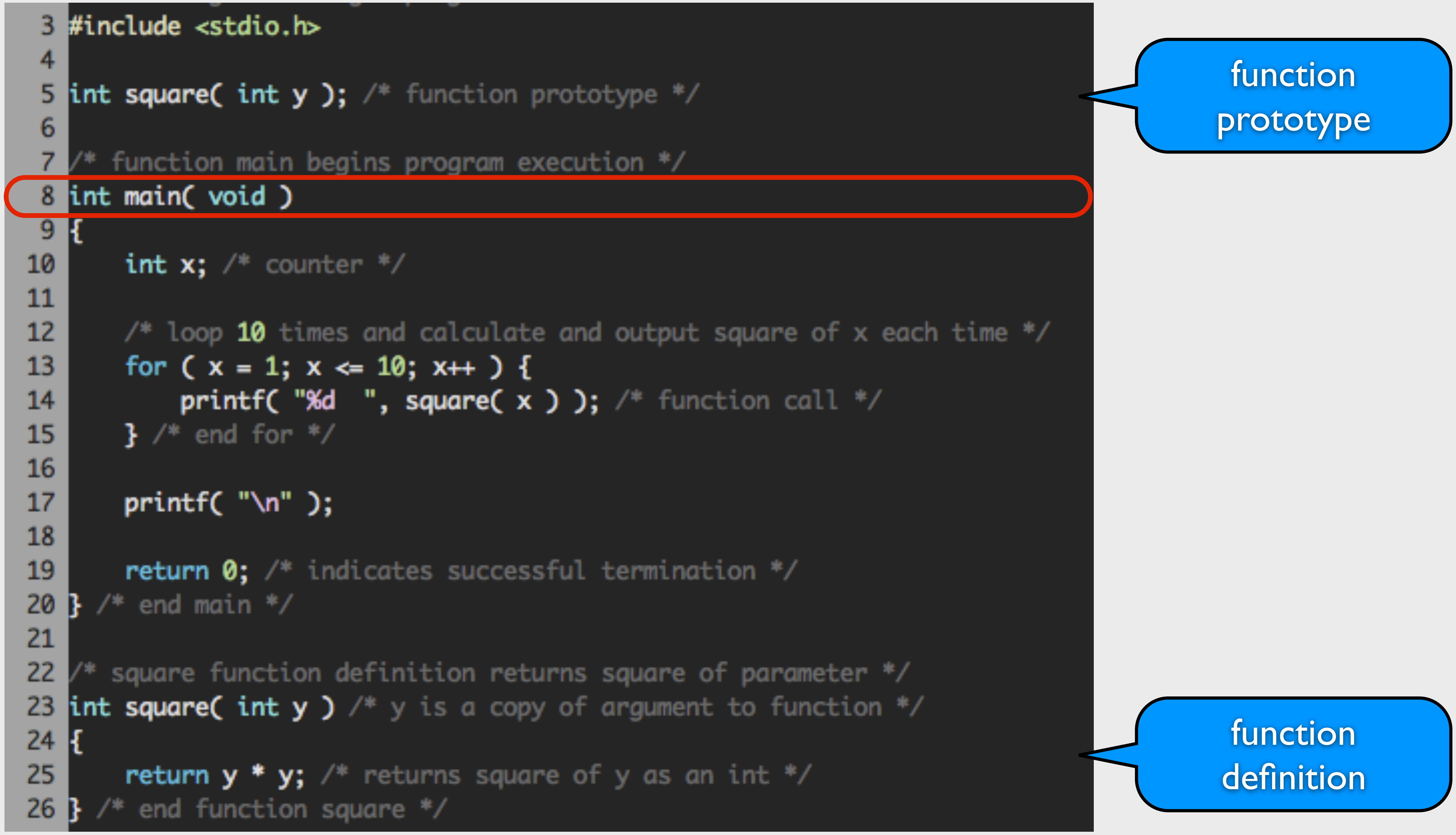
```c
#include <stdio.h>

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.)

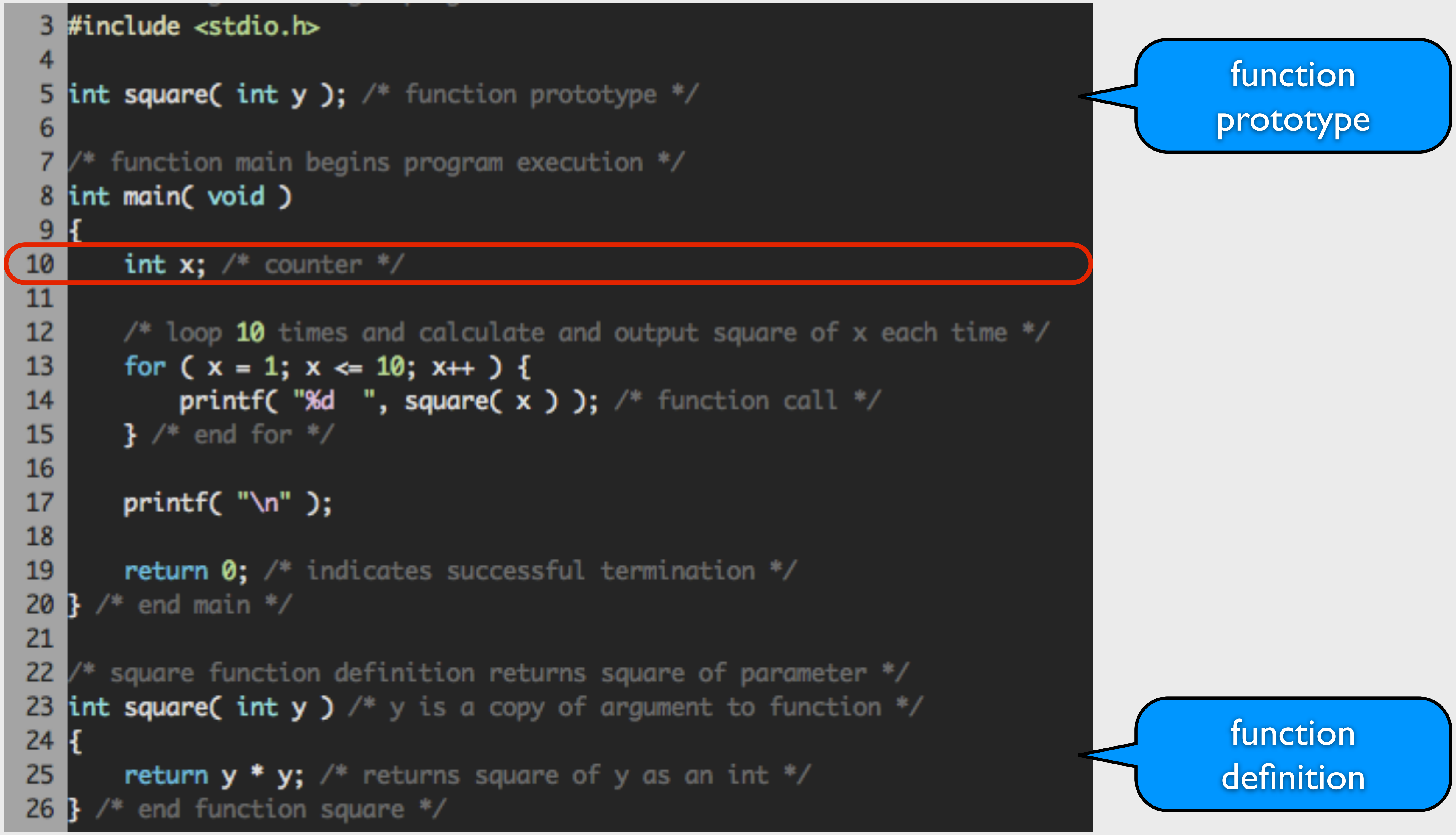
- Example: fig05_03.c



```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 Definitions (Cont.)

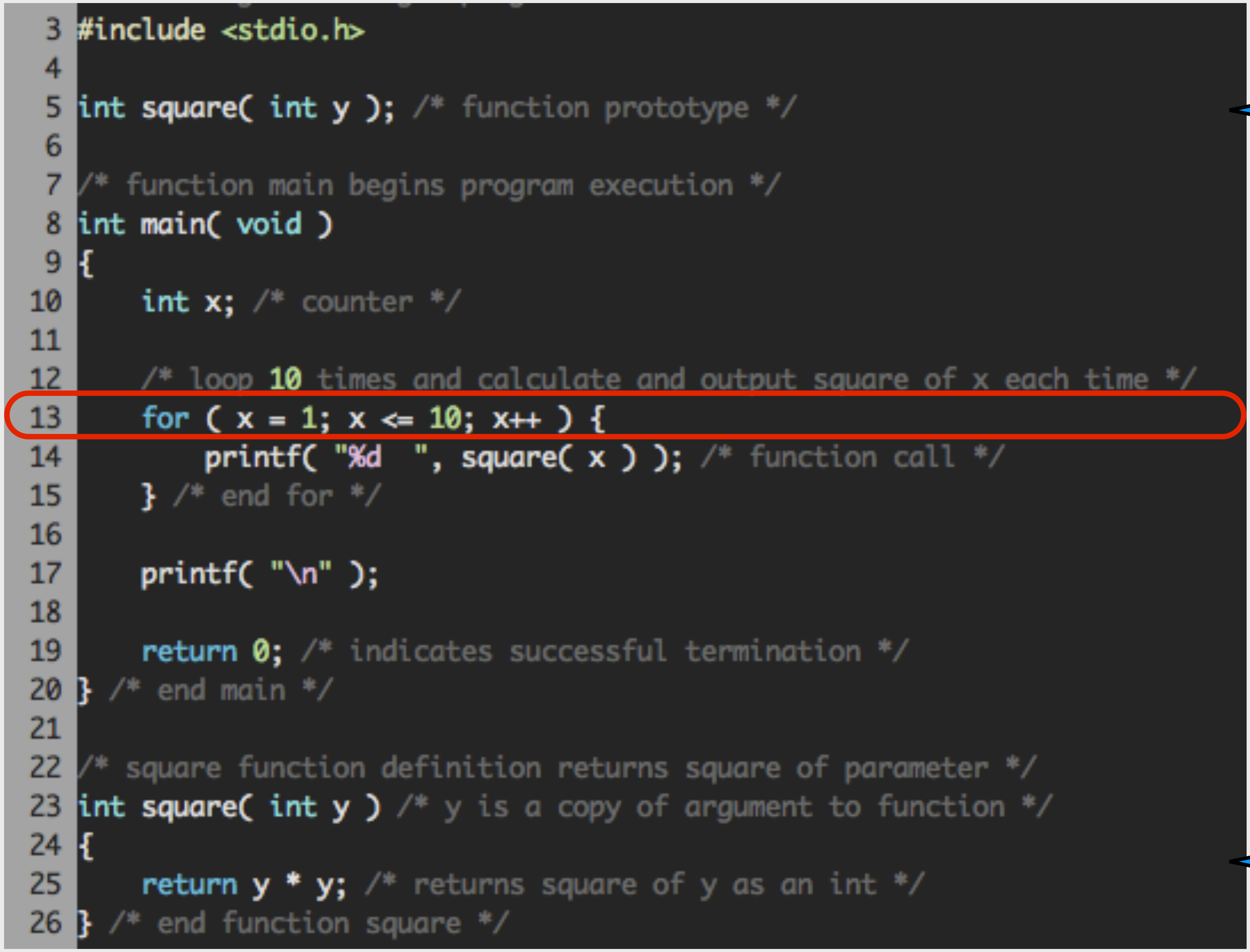
- Example: fig05_03.c



```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 Definitions (Cont.)

- Example: fig05_03.c



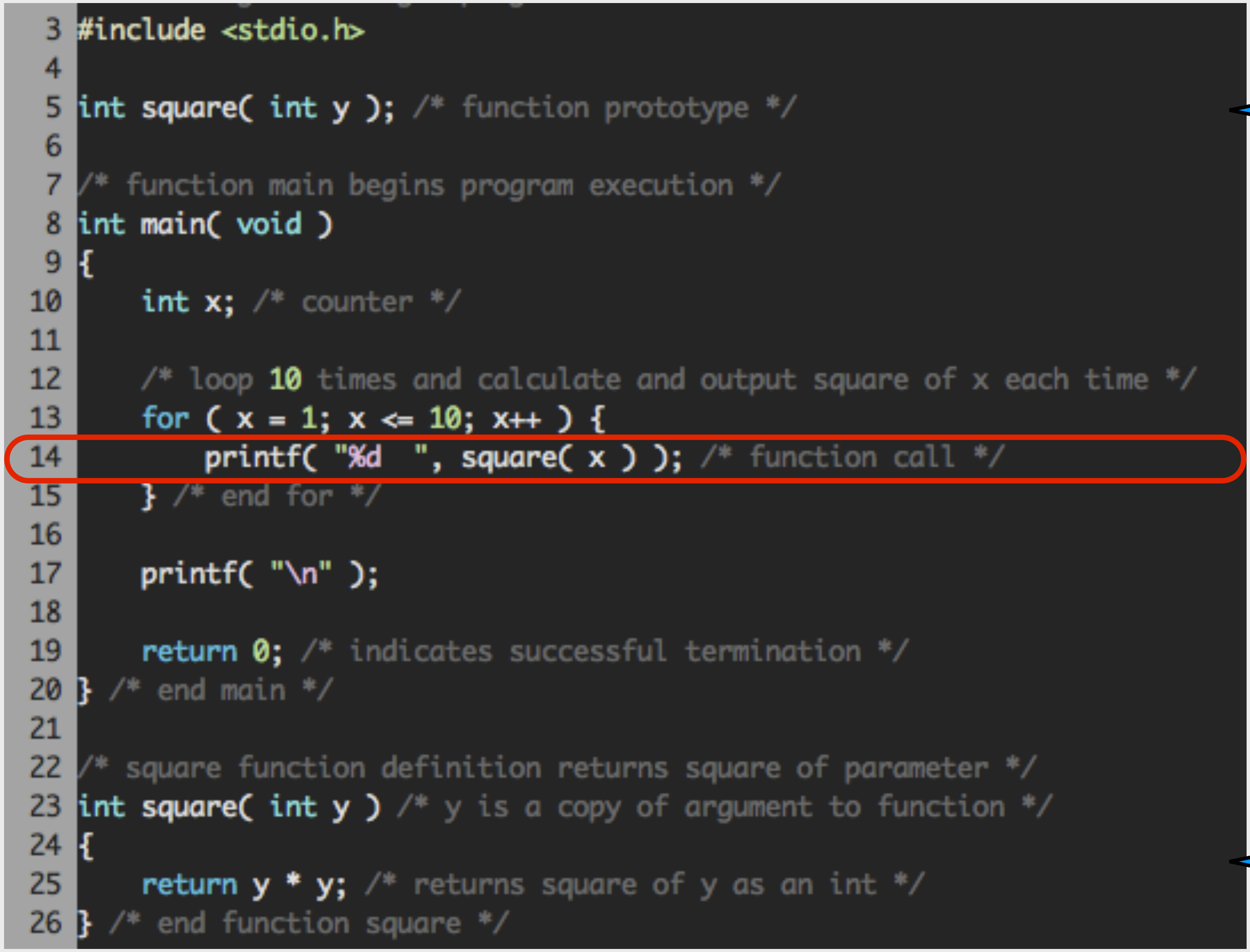
```c
#include <stdio.h>

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.)

- Example: fig05_03.c



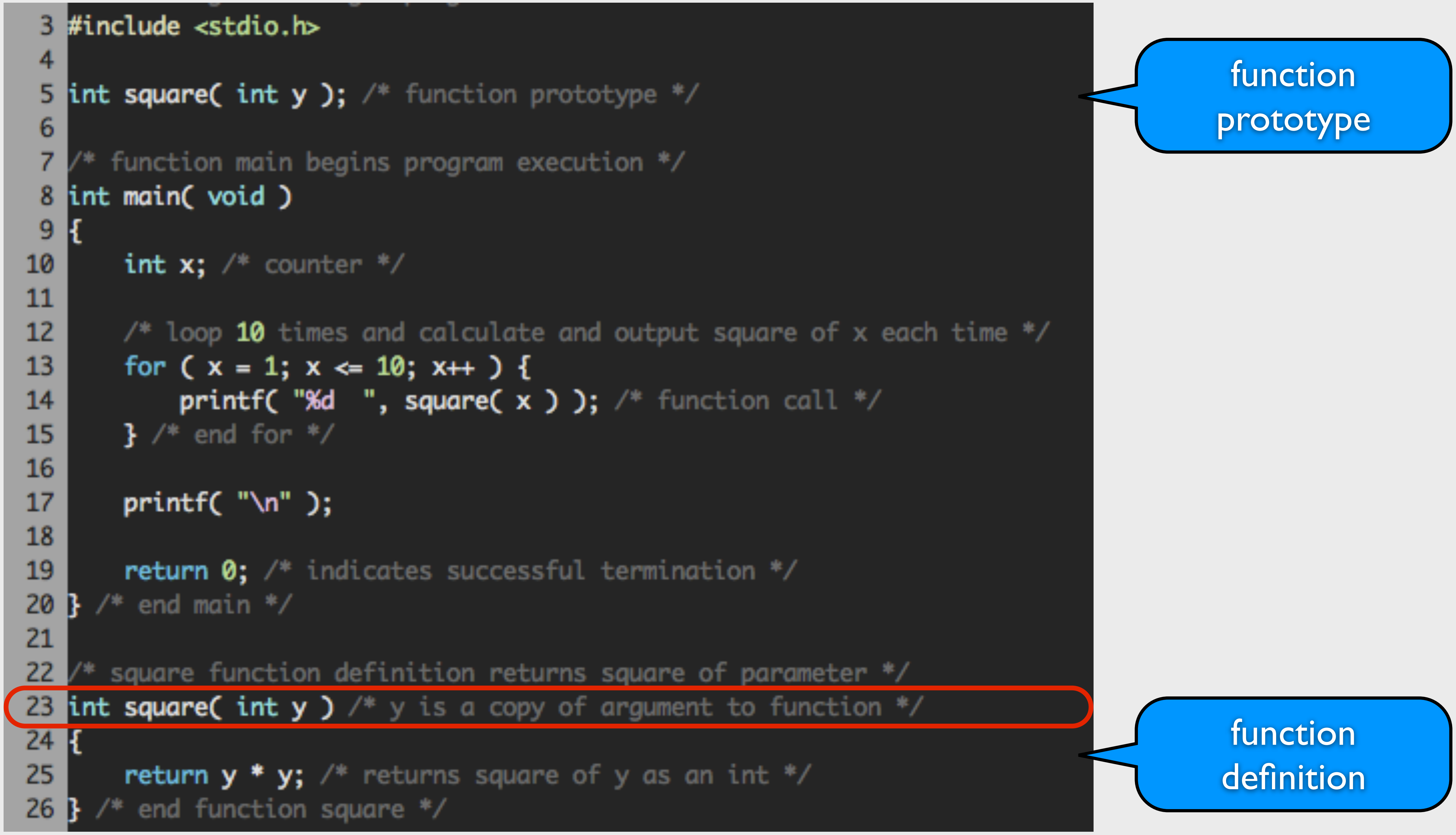
```c
#include <stdio.h>

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.)

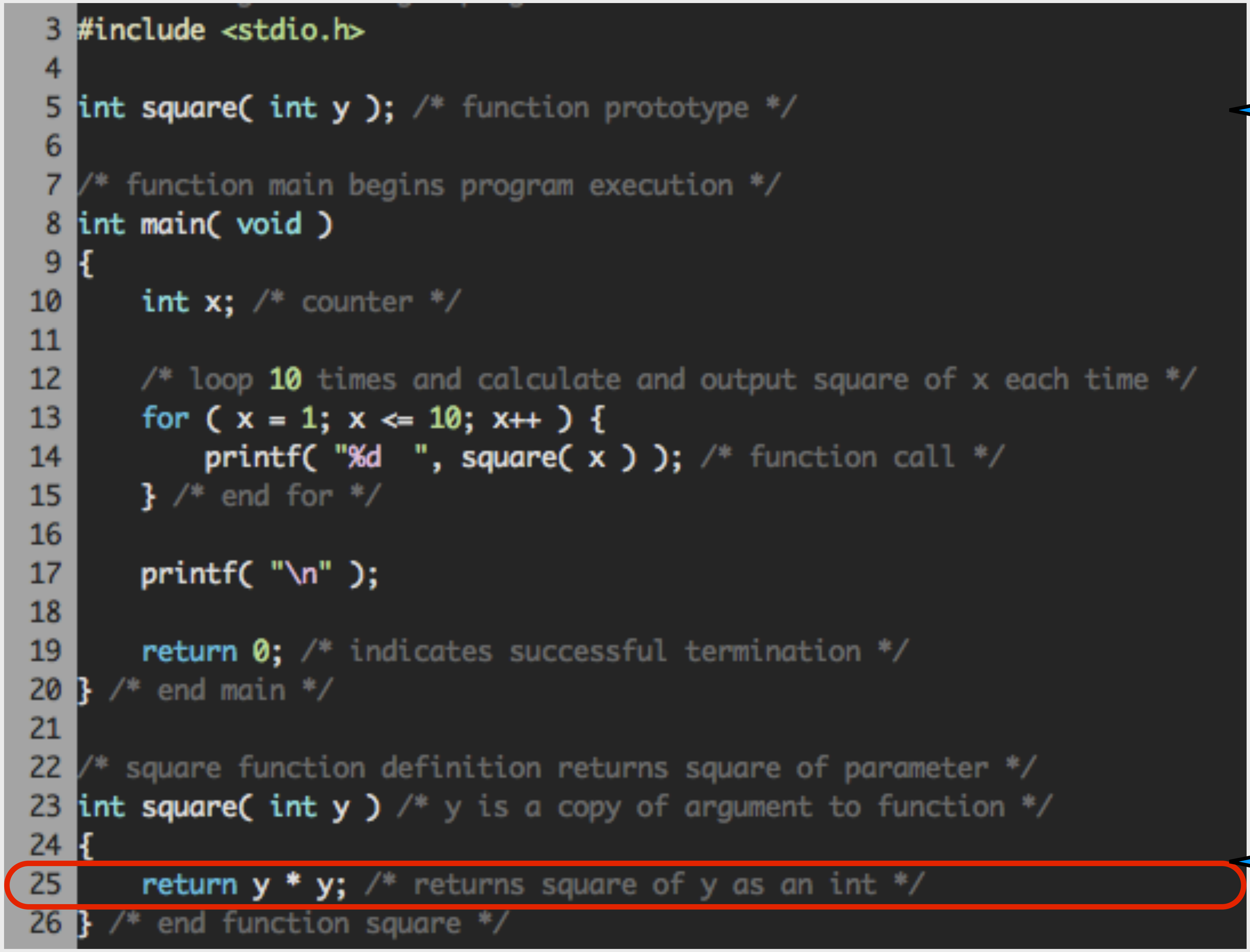
- Example: fig05_03.c



```
#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 Definitions (Cont.)

- Example: fig05_03.c



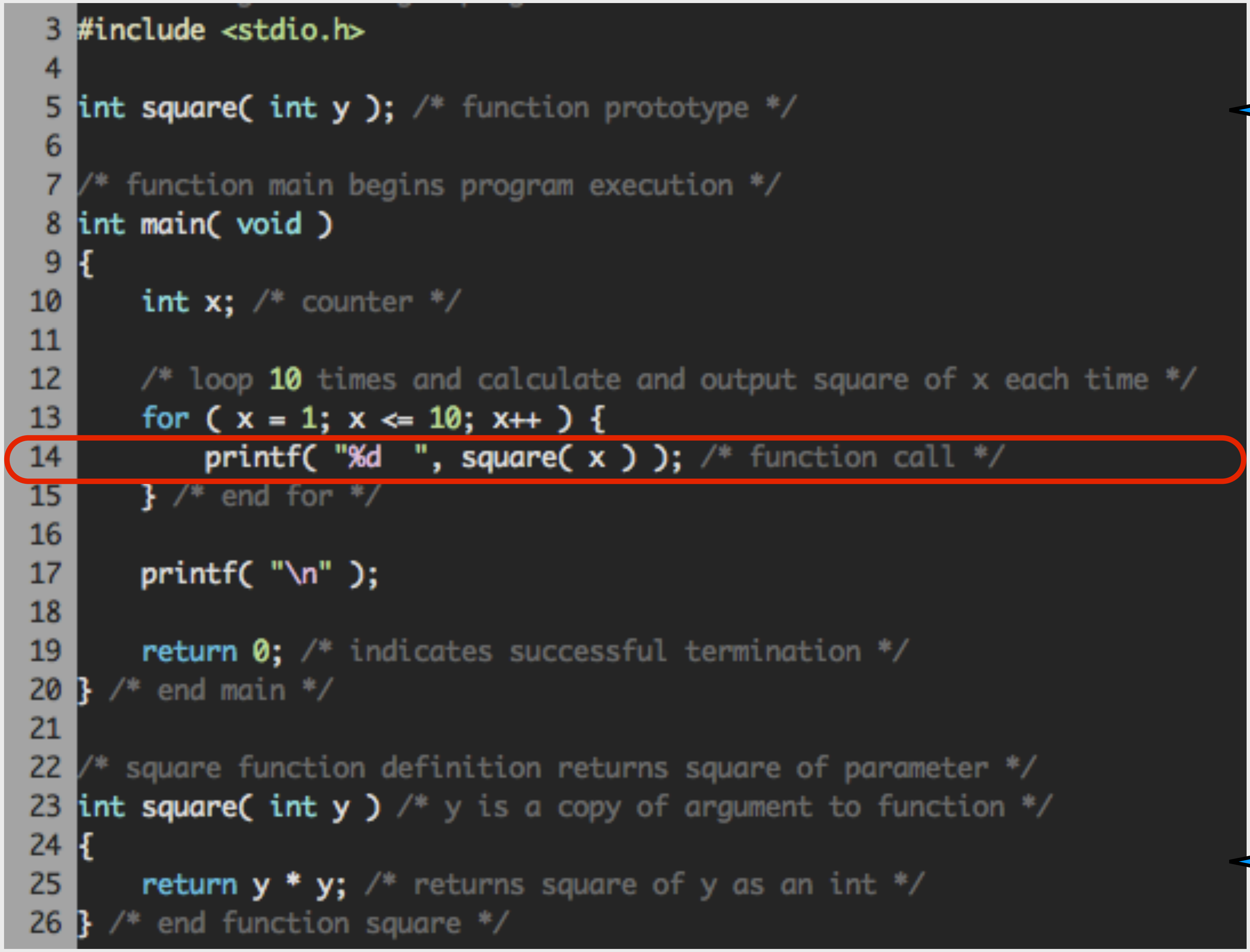
```c
#include <stdio.h>

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.)

- Example: fig05_03.c



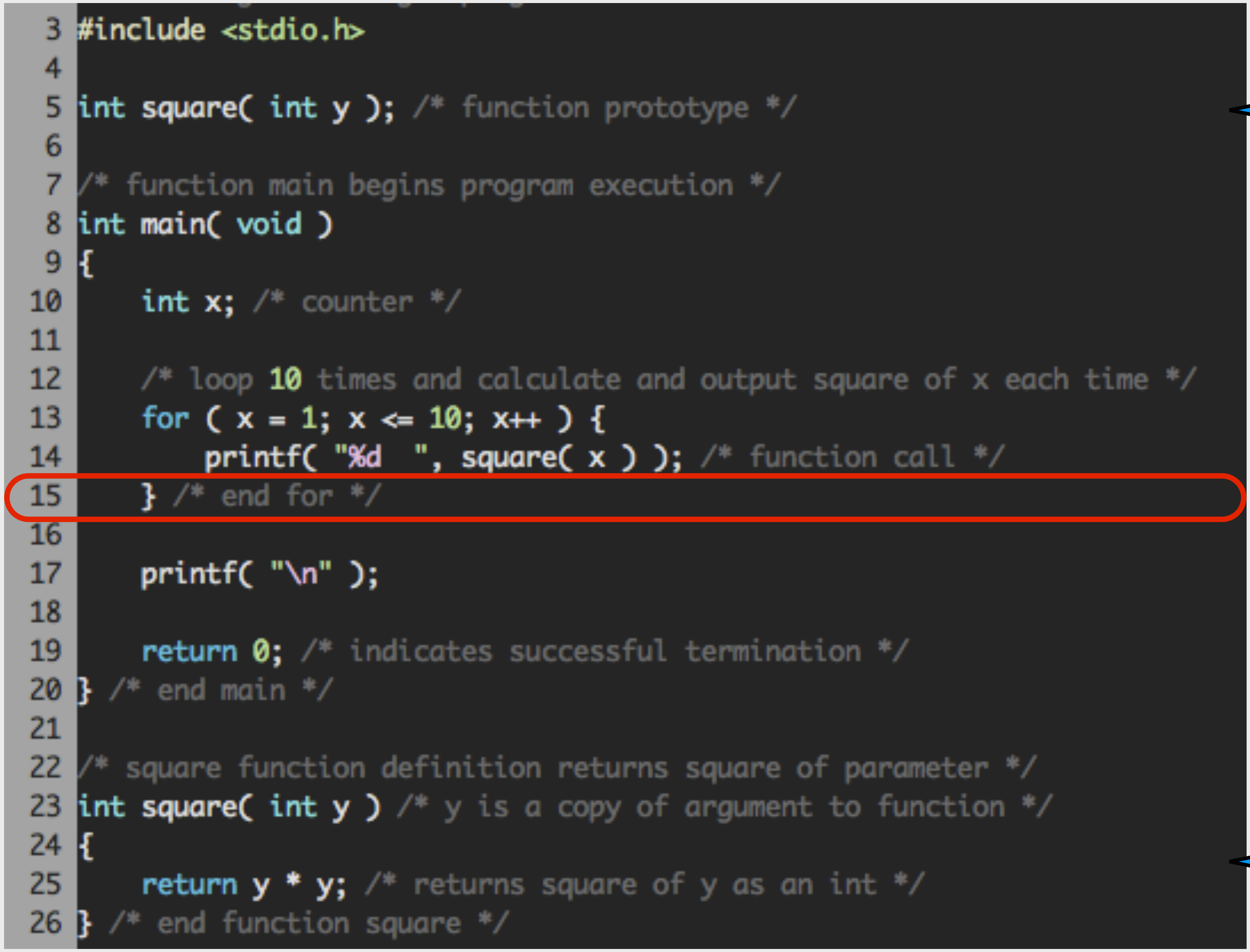
```c
#include <stdio.h>

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.)

- Example: fig05_03.c



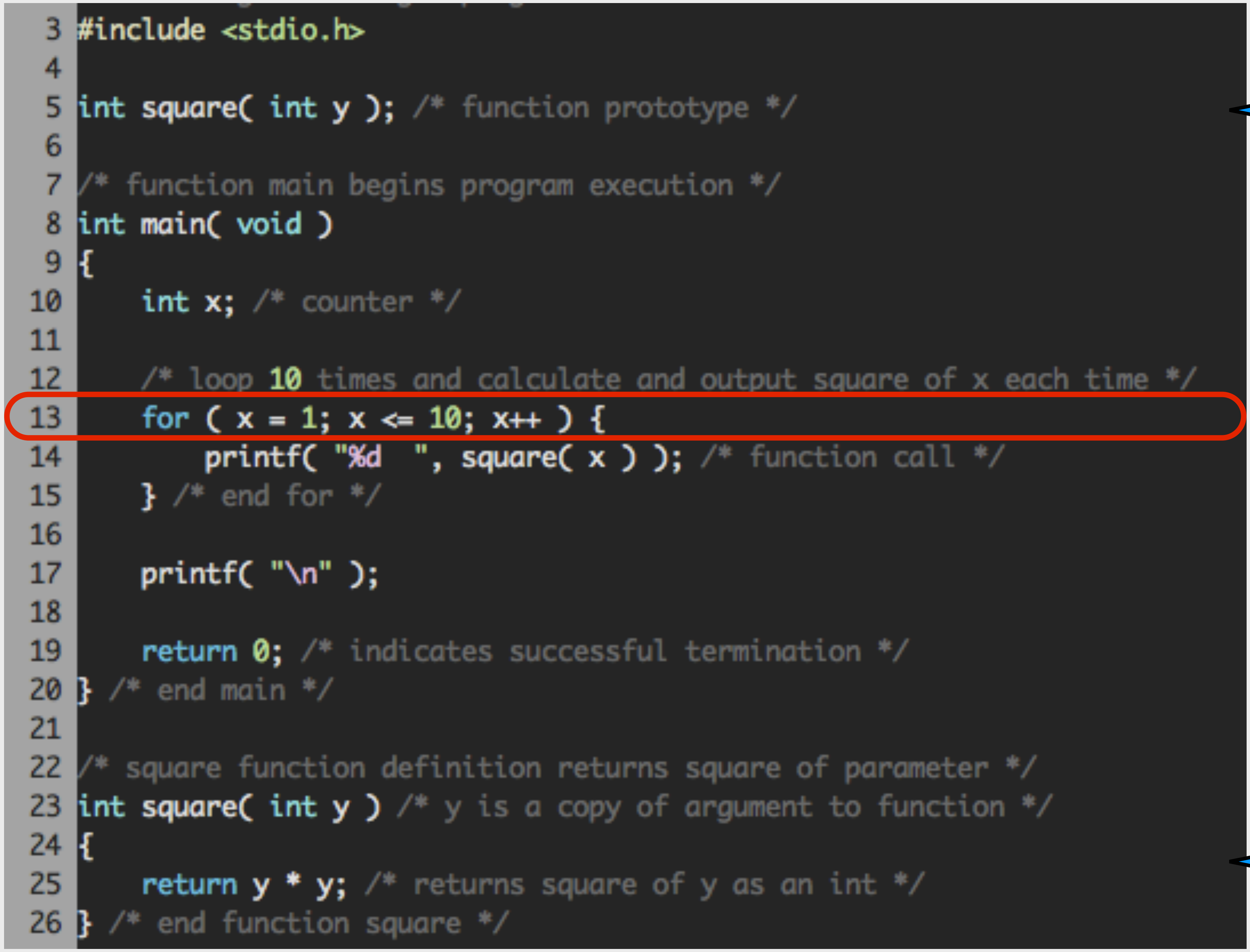
```c
#include <stdio.h>

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.)

- Example: fig05_03.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.)

- The **parameter-list** is a comma-separated list that specifies the parameters received by the function when it's called.
  - If a function does not receive any values, parameter-list is **void**.
  - A type must be listed explicitly for each parameter.

Function Definitions (Cont.)

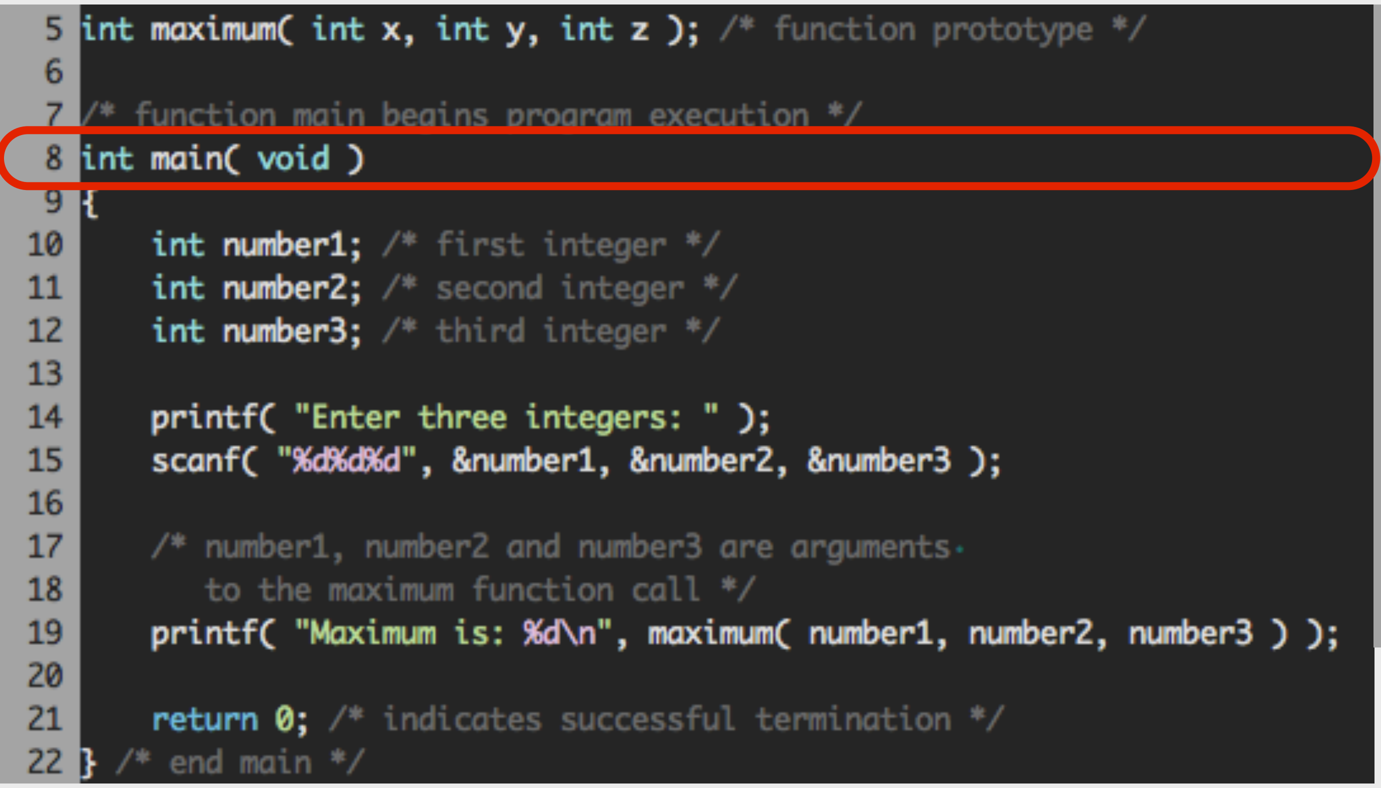
- `return` statement
  - If the function does return a result, the statement

    return expression;

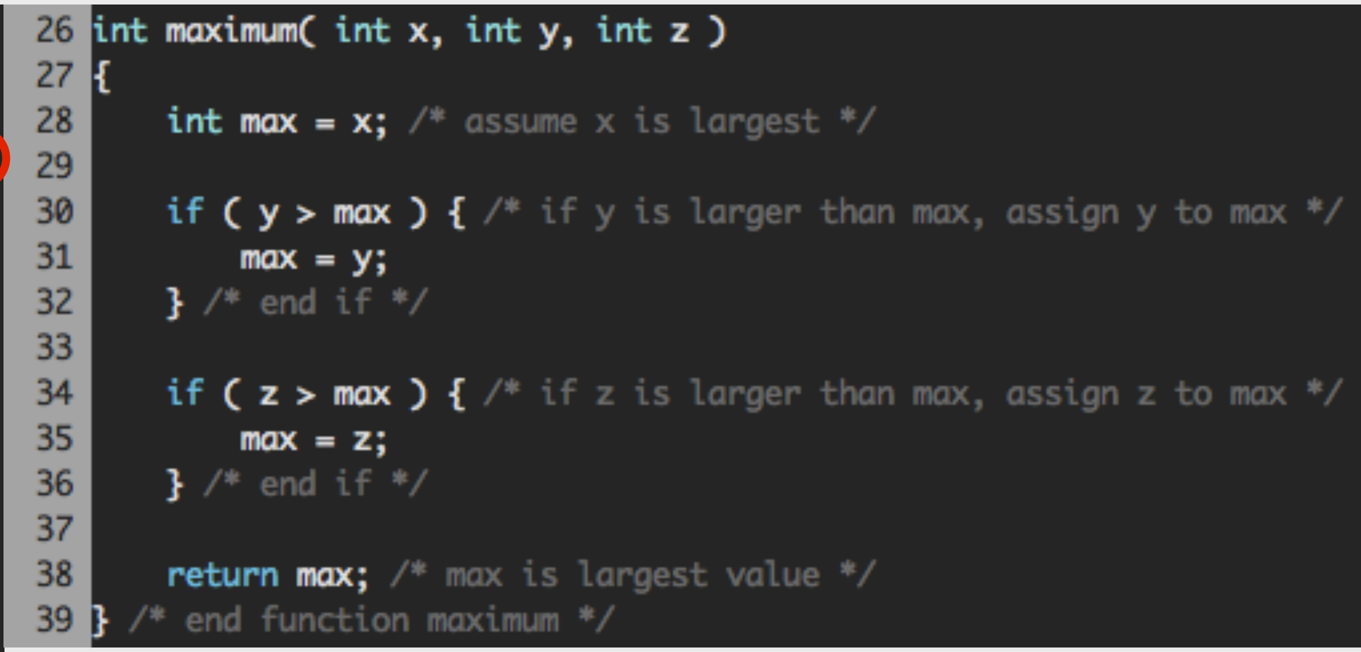
  - returns the value of expression to the caller.

Function Definitions (Cont.)

- Example: fig05_04.c



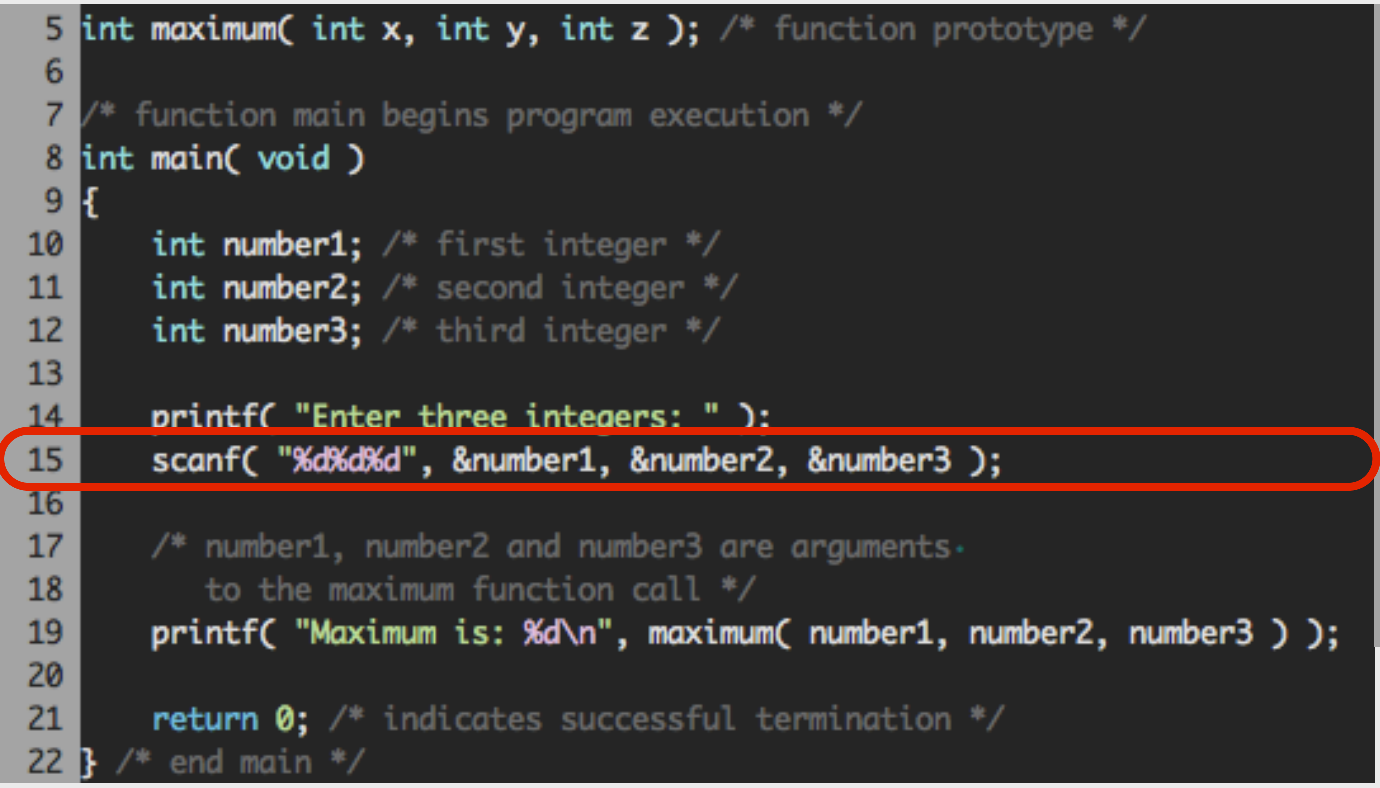
```c
int maximum( int x, int y, int z ); /* function prototype */

/* 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 */
```



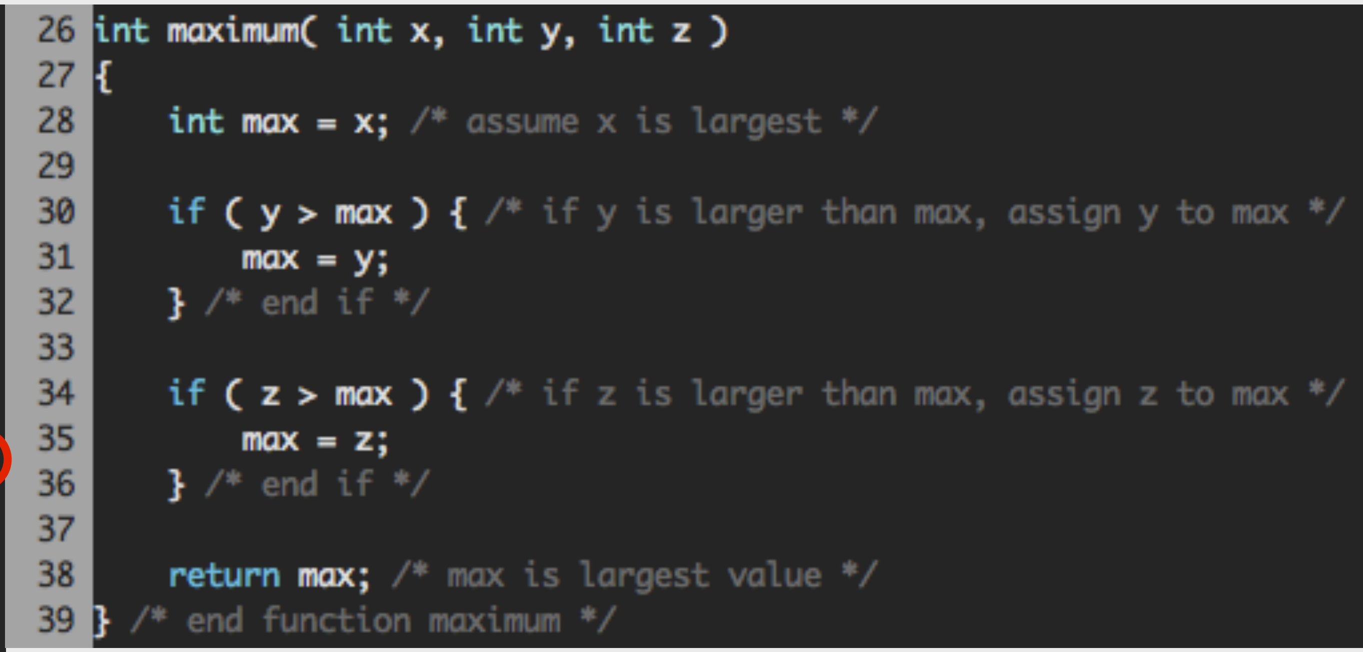
```c
int maximum( int x, int y, int z )
{
    int max = x; /* assume x is largest */

    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 */
```

Function Definitions (Cont.)

- Example: fig05_04.c



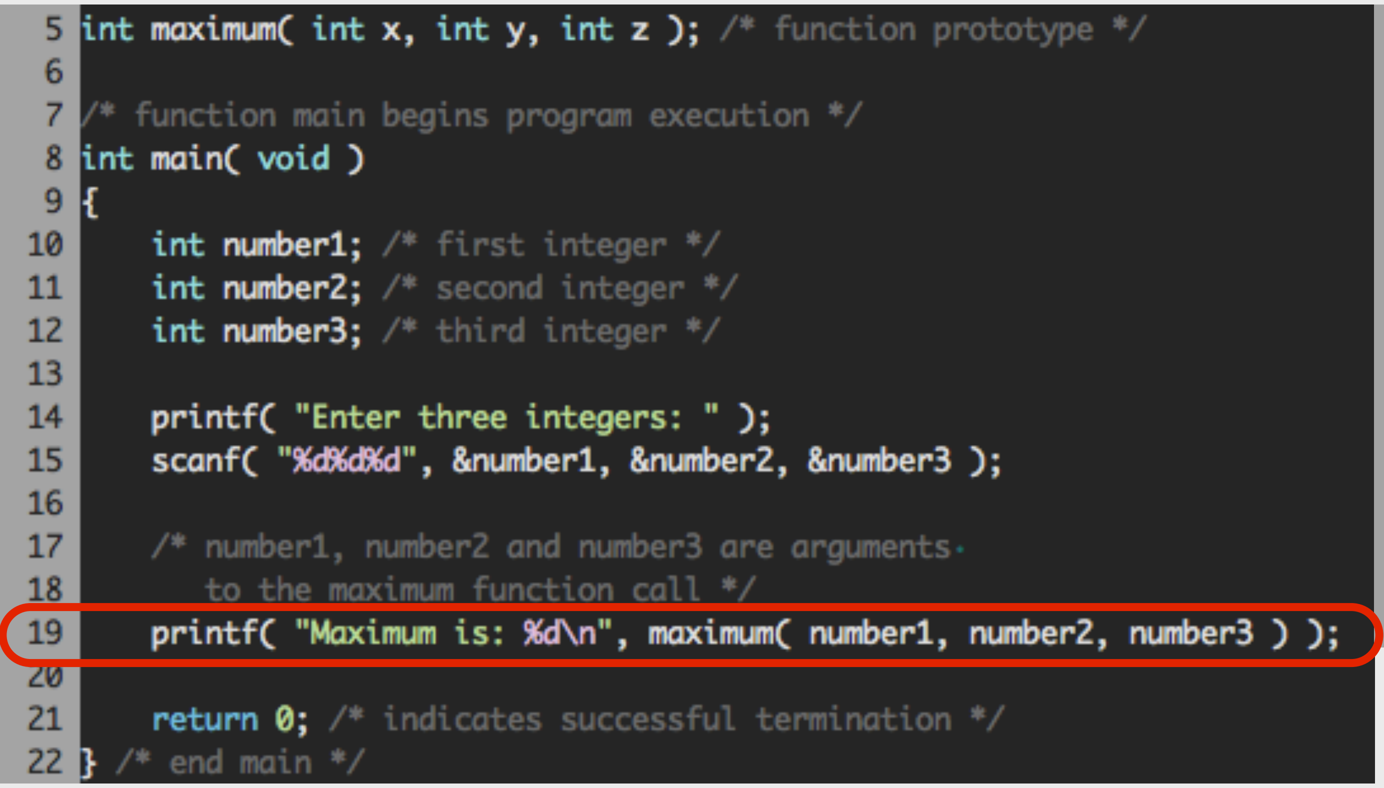
```c
int maximum( int x, int y, int z ); /* function prototype */

/* 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 */
```



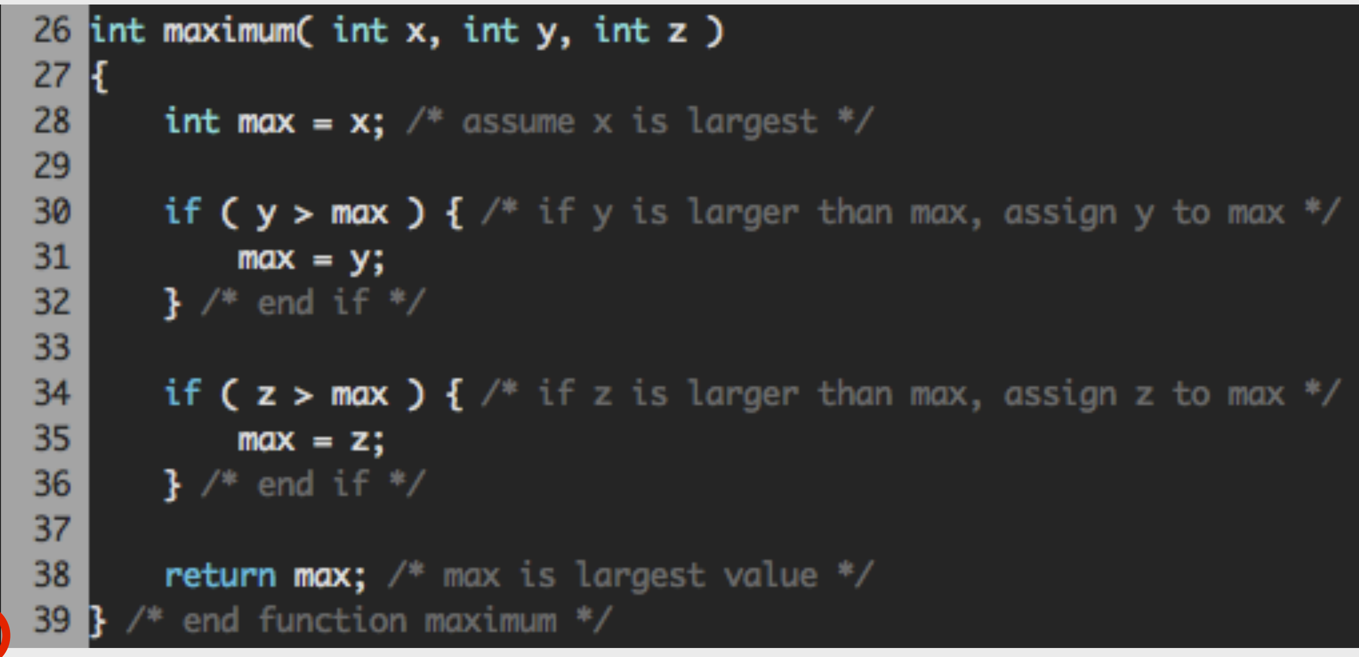
```c
int maximum( int x, int y, int z )
{
    int max = x; /* assume x is largest */

    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 */
```

Function Definitions (Cont.)

- Example: fig05_04.c



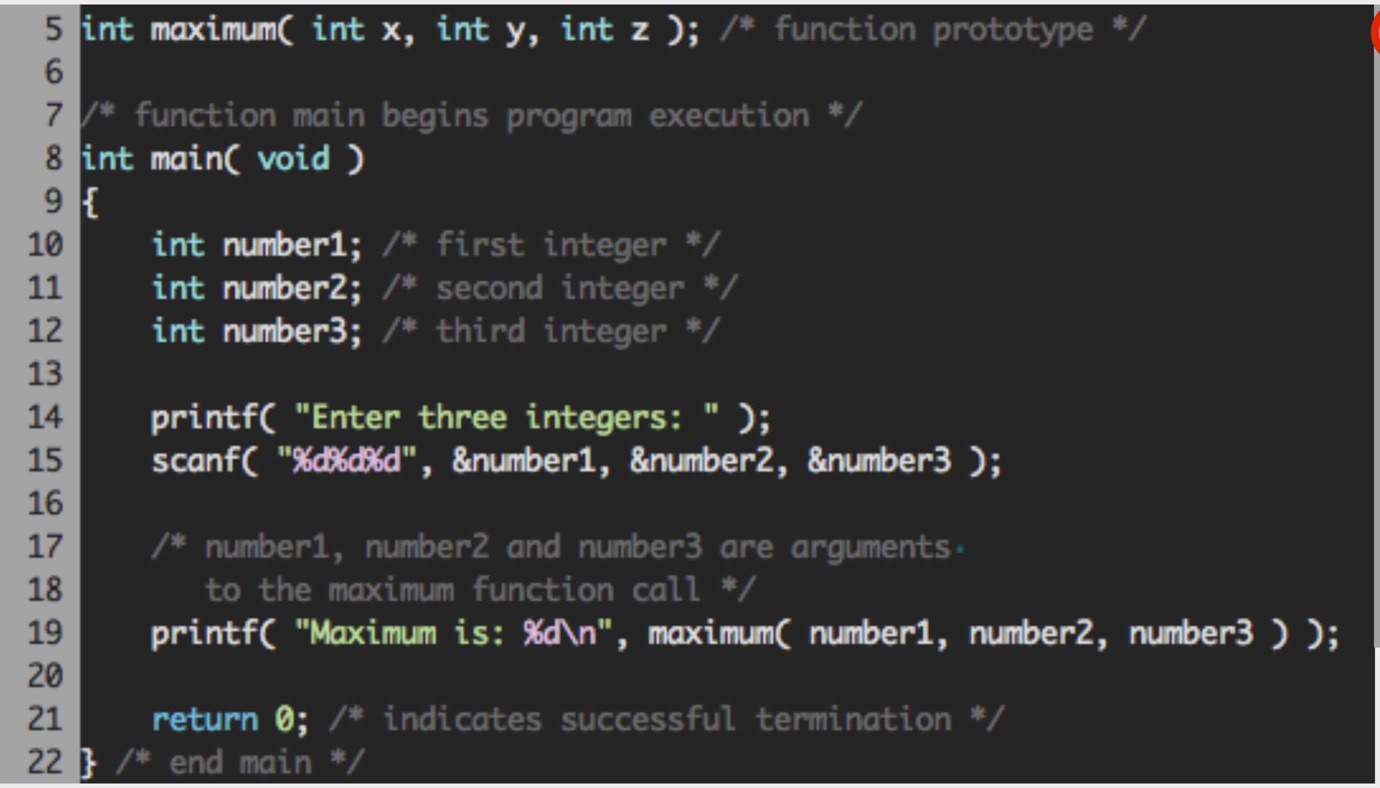
```c
int maximum( int x, int y, int z ); /* function prototype */

/* 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 */
```



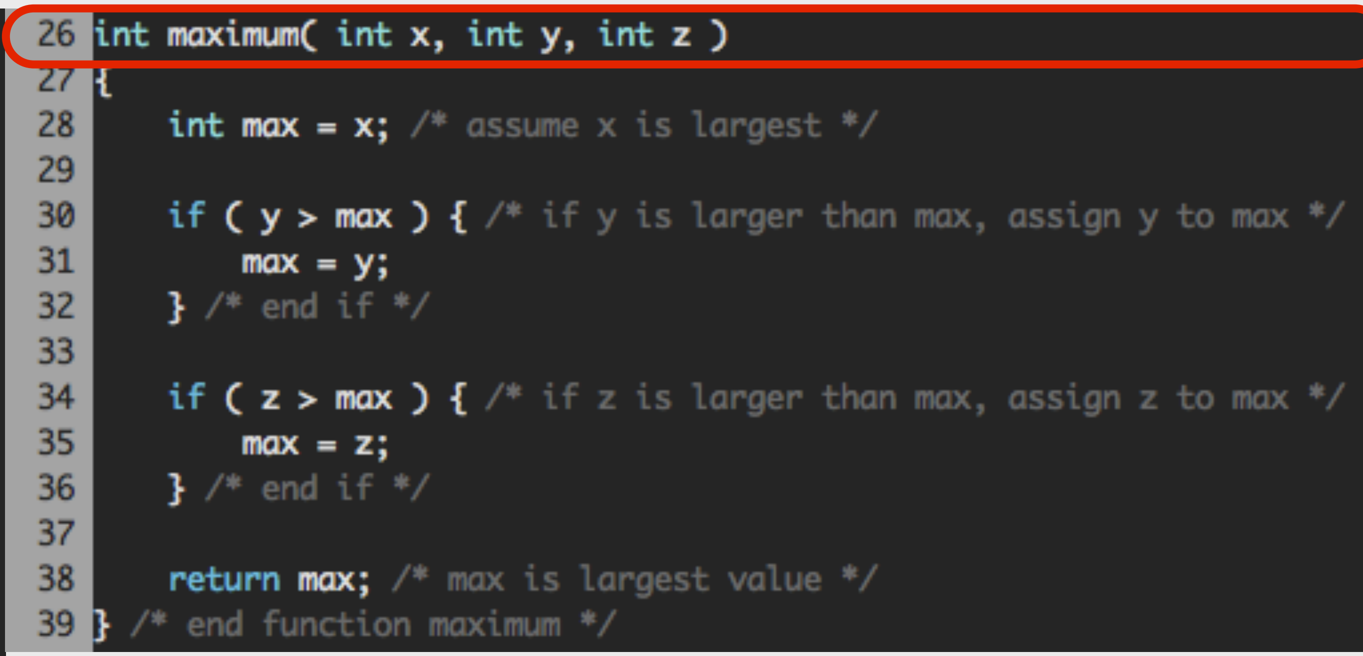
```c
int maximum( int x, int y, int z )
{
    int max = x; /* assume x is largest */

    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 */
```

Function Definitions (Cont.)

- Example: fig05_04.c



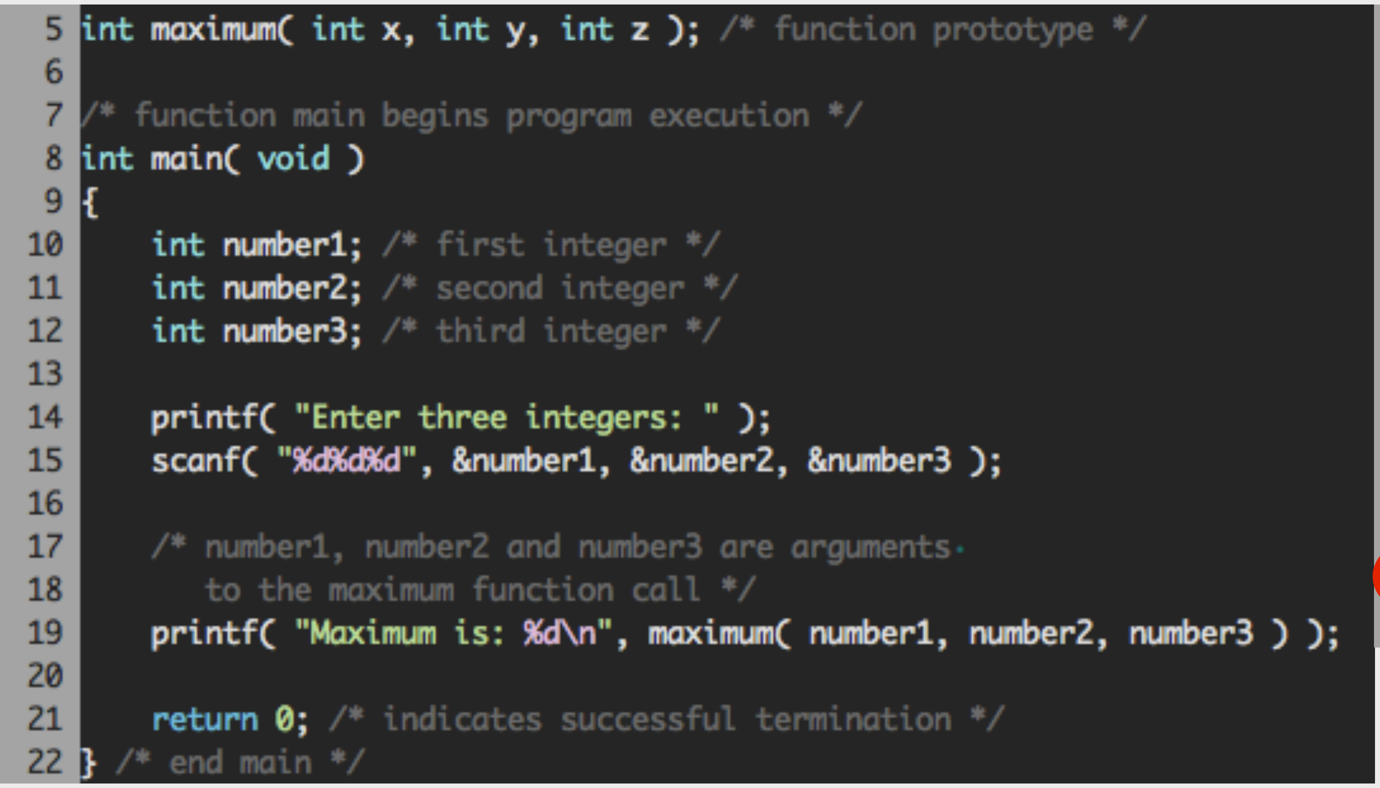
```c
int maximum( int x, int y, int z ); /* function prototype */

/* 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 */
```



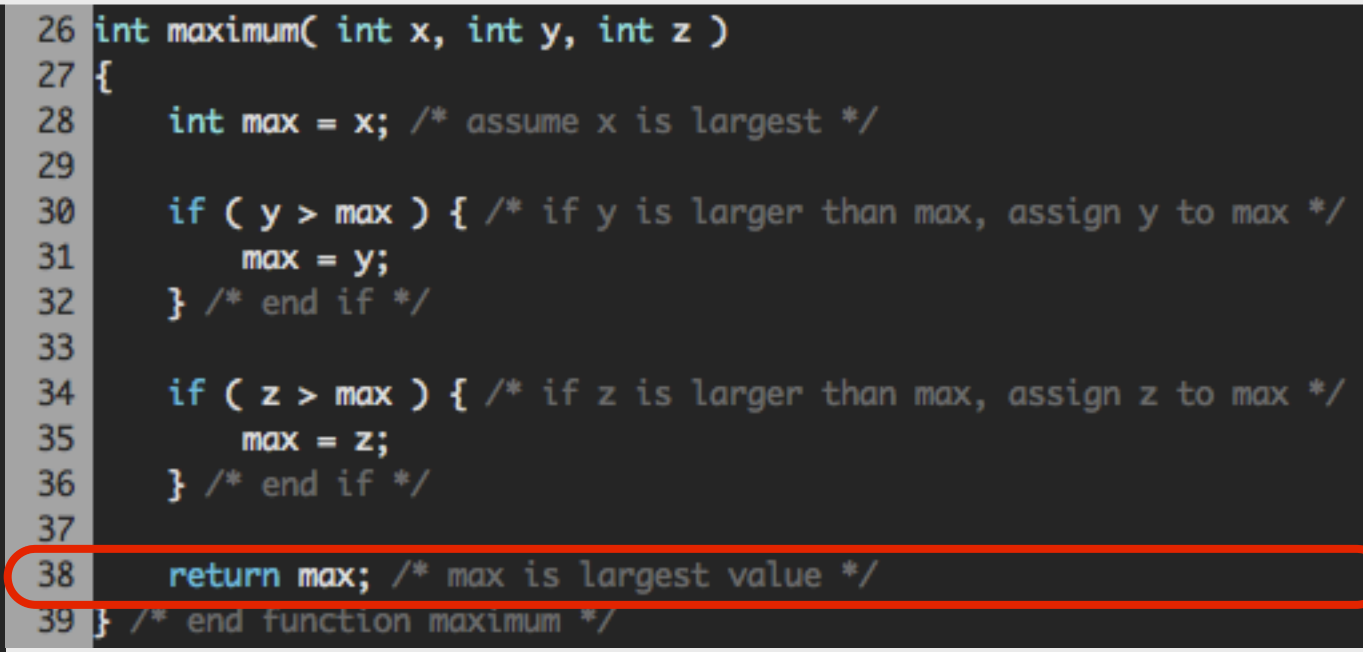
```c
int maximum( int x, int y, int z )
{
    int max = x; /* assume x is largest */

    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 */
```

Function Definitions (Cont.)

- Example: fig05_04.c



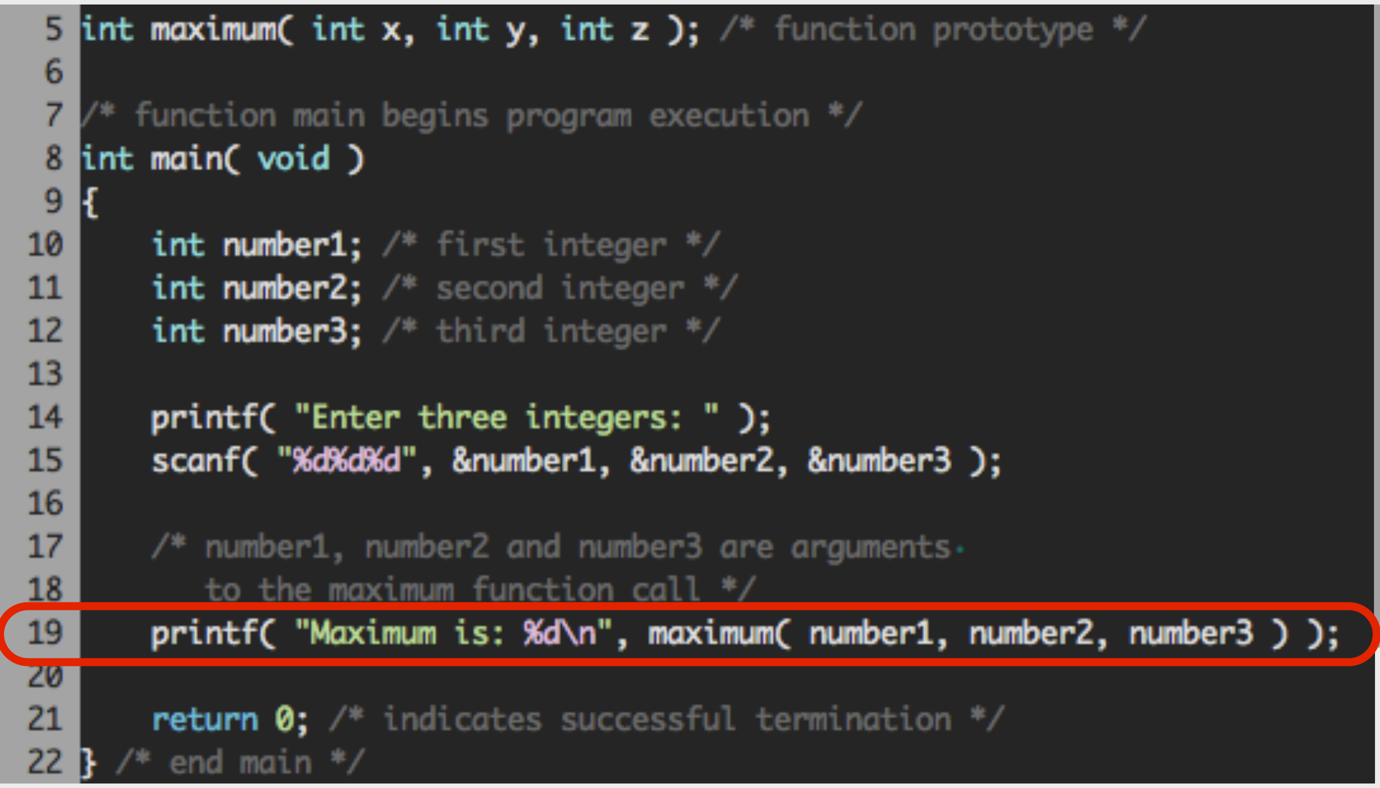
```c
int maximum( int x, int y, int z ); /* function prototype */

/* 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 */
```



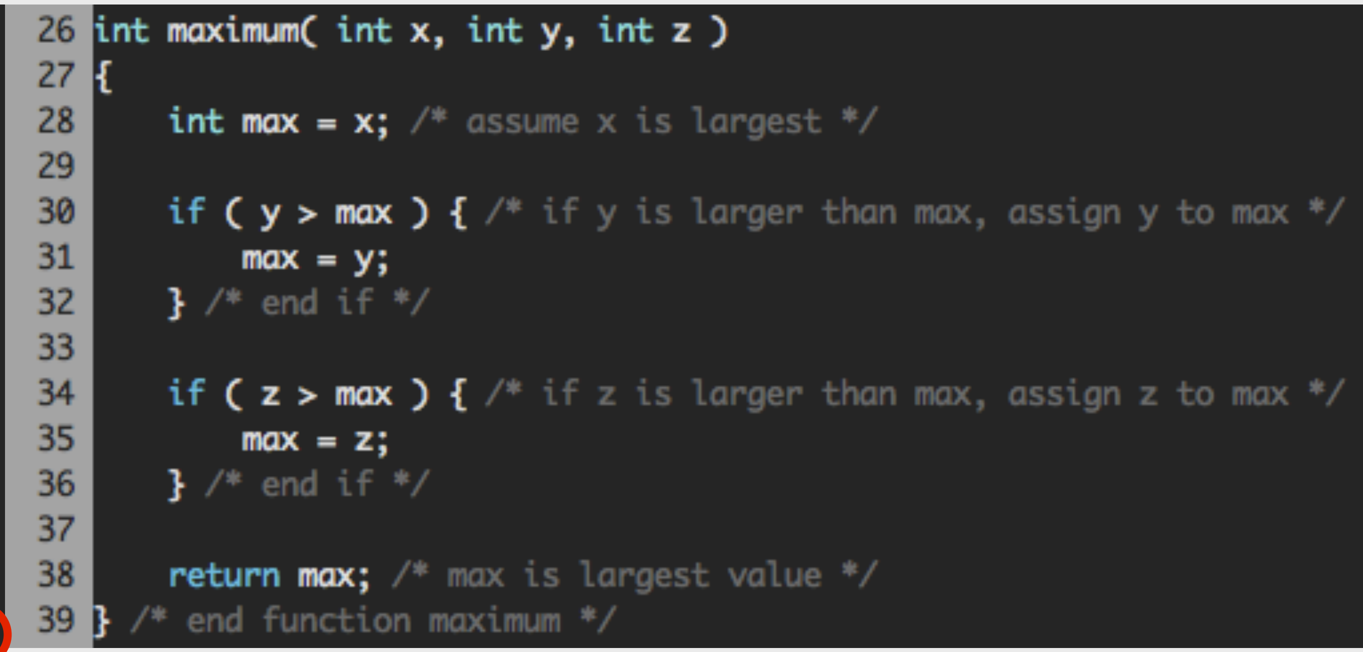
```c
int maximum( int x, int y, int z )
{
    int max = x; /* assume x is largest */

    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 */
```

Function Definitions (Cont.)

- Example: fig05_04.c



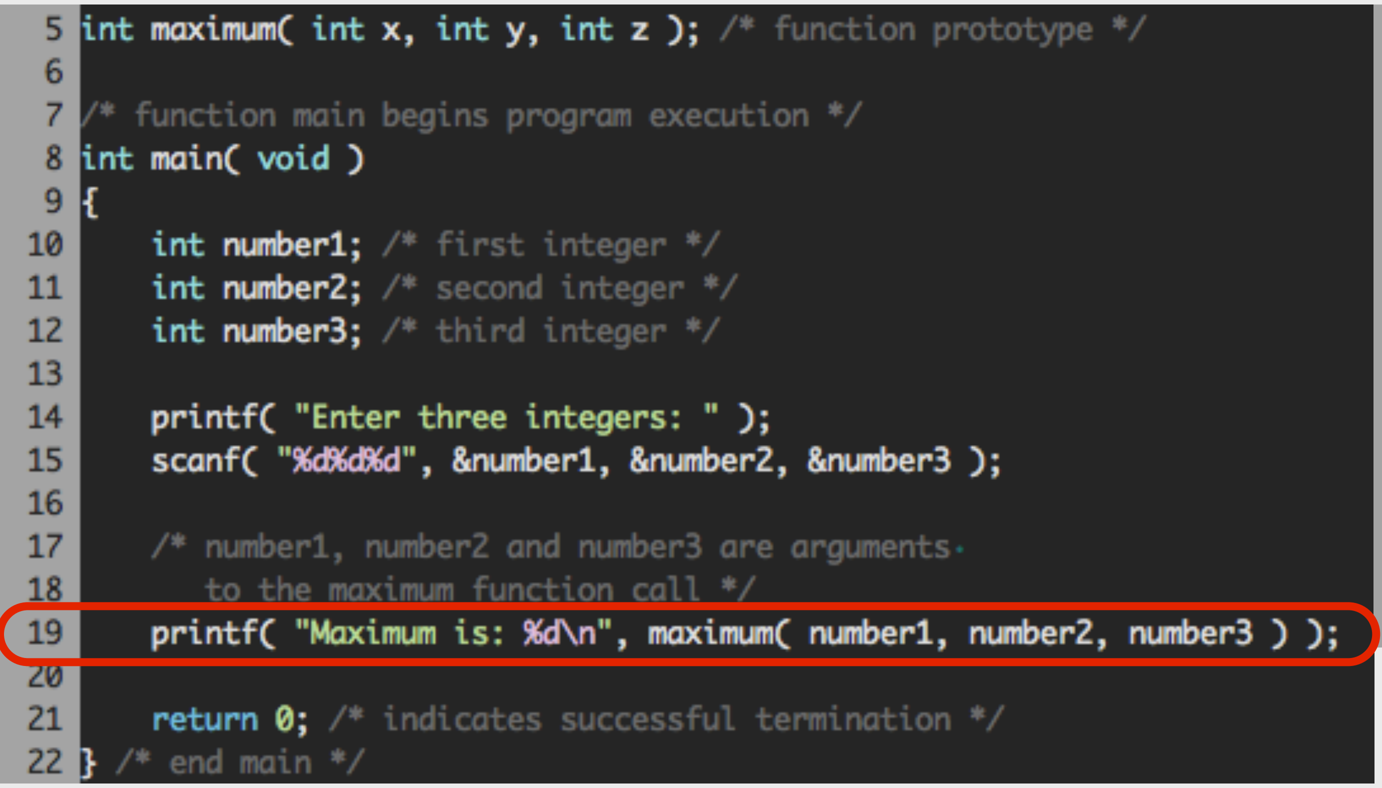
```c
int maximum( int x, int y, int z ); /* function prototype */

/* 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 */
```



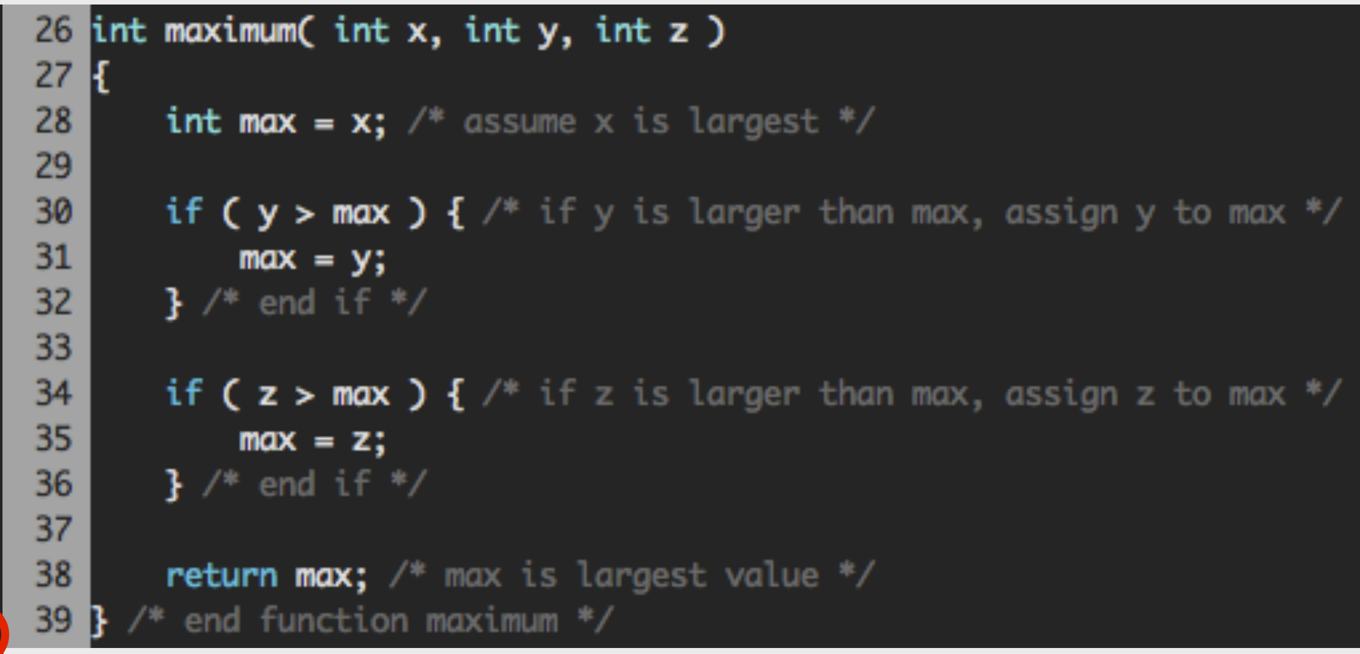
```c
int maximum( int x, int y, int z )
{
    int max = x; /* assume x is largest */

    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 */
```

Function Definitions (Cont.)

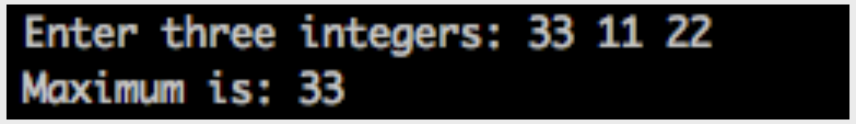
- Example: fig05_04.c



```
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 */
```



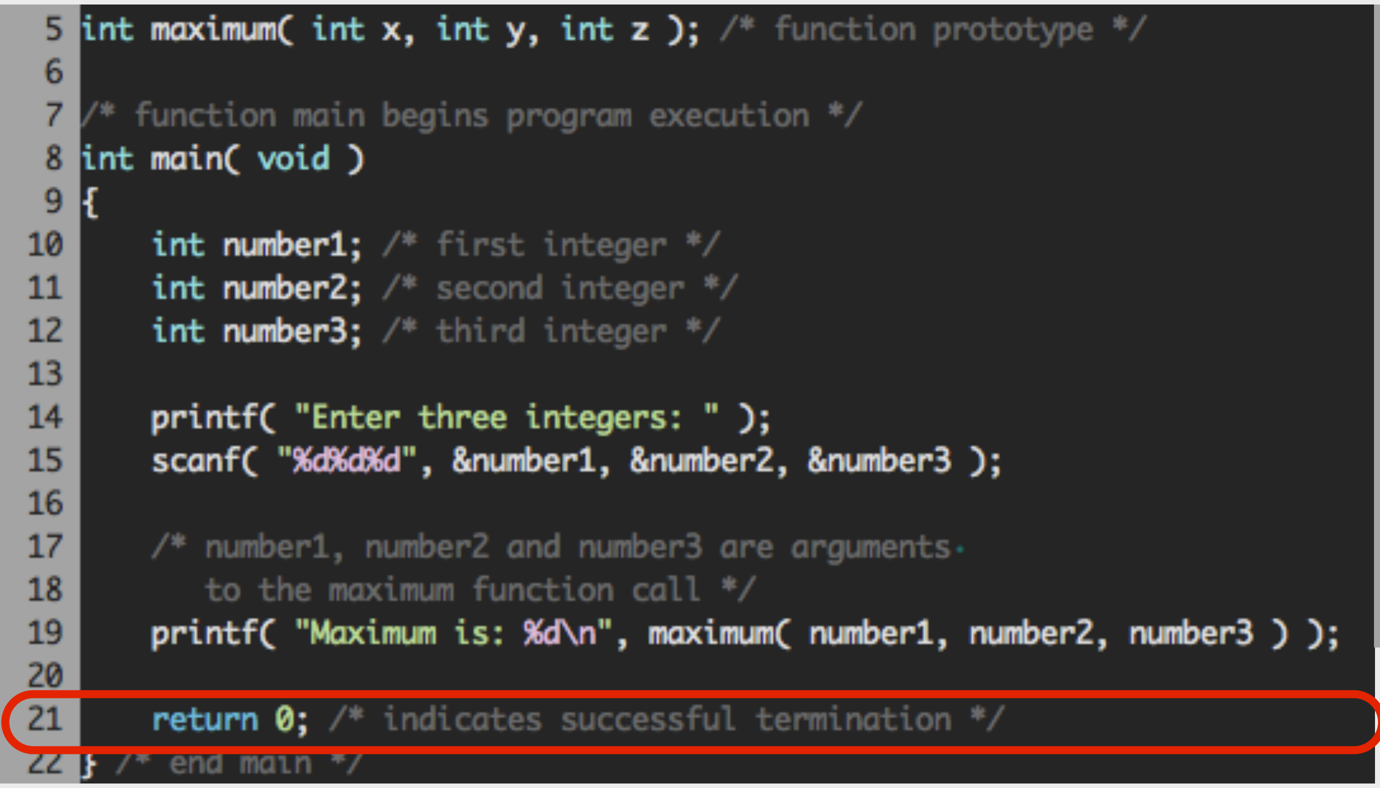
```
int maximum( int x, int y, int z )
{
    int max = x; /* assume x is largest */

    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 */
```



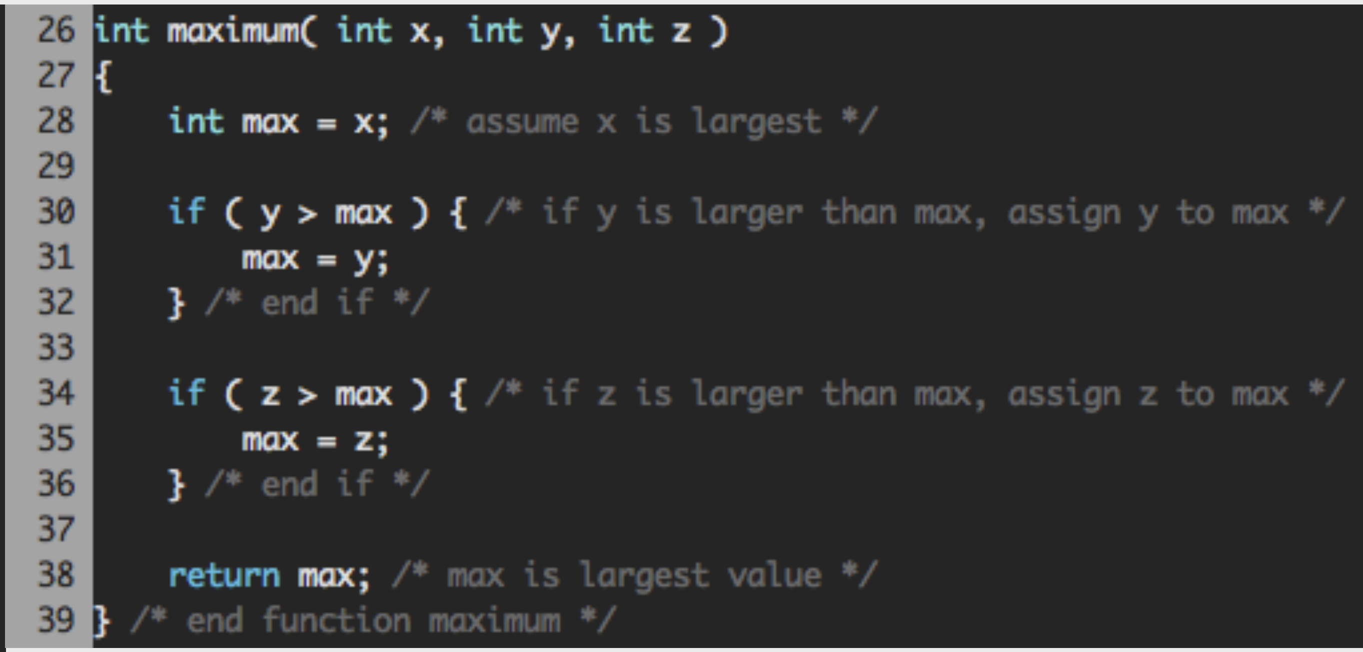
```
Enter three integers: 33 11 22
Maximum is: 33
```

Function Definitions (Cont.)

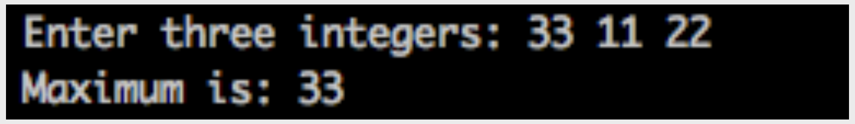
- Example: fig05_04.c



```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

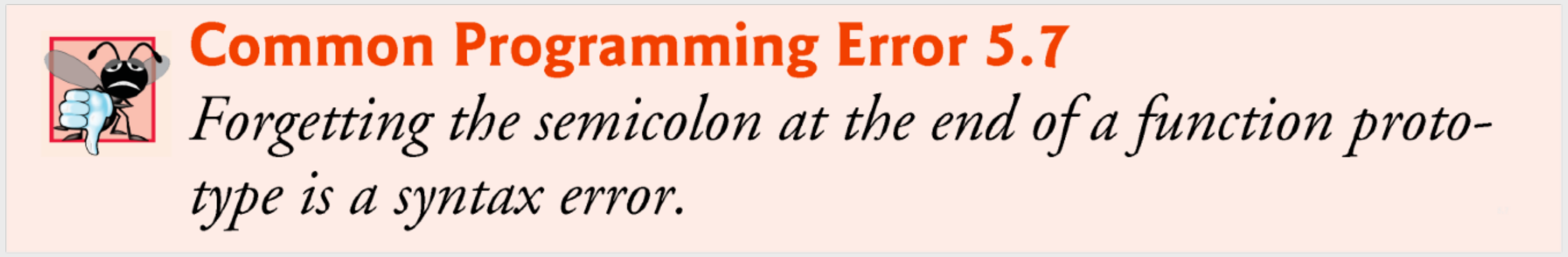
- One of the most important features of C is the function prototype.
- A function prototype tells the compiler the type of data returned by the function, the number of parameters the function expects to receive, the types of the parameters, and the order in which these parameters are expected.
- The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

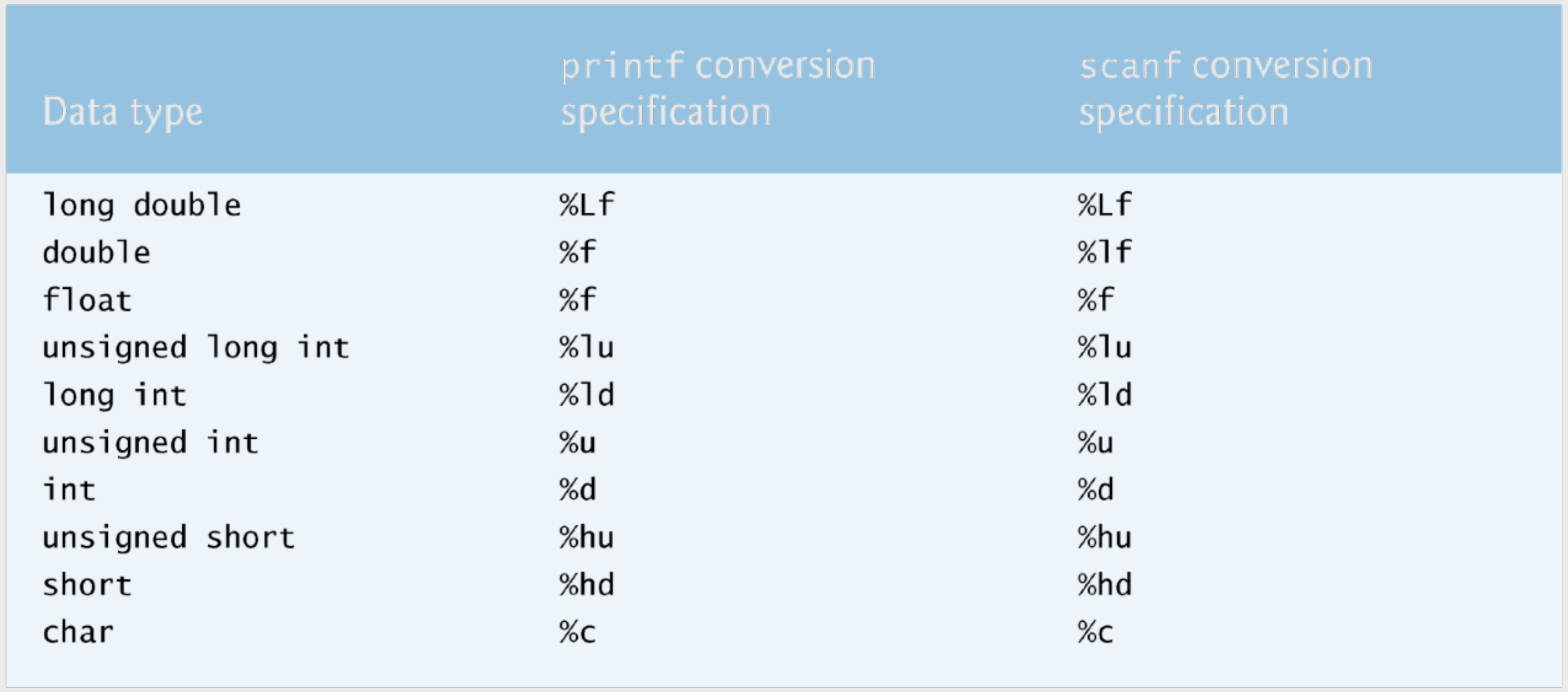
Forgetting the semicolon at the end of a function prototype is a syntax error.

Function Prototypes (Cont.)

- A function call that **does not match** the function prototype is a **compilation error**.
- An error is also generated if the function prototype and the function definition disagree.
- Another important feature of function prototypes is the coercion of arguments, i.e., the forcing of arguments to the appropriate type.
  - for instance, the **`sqrt( double )`** in **`<math.h>`**

Function Prototypes (Cont.)

- The data types in order from highest type to lowest type with each type's **`printf`** and **`scanf`** conversion specifications.

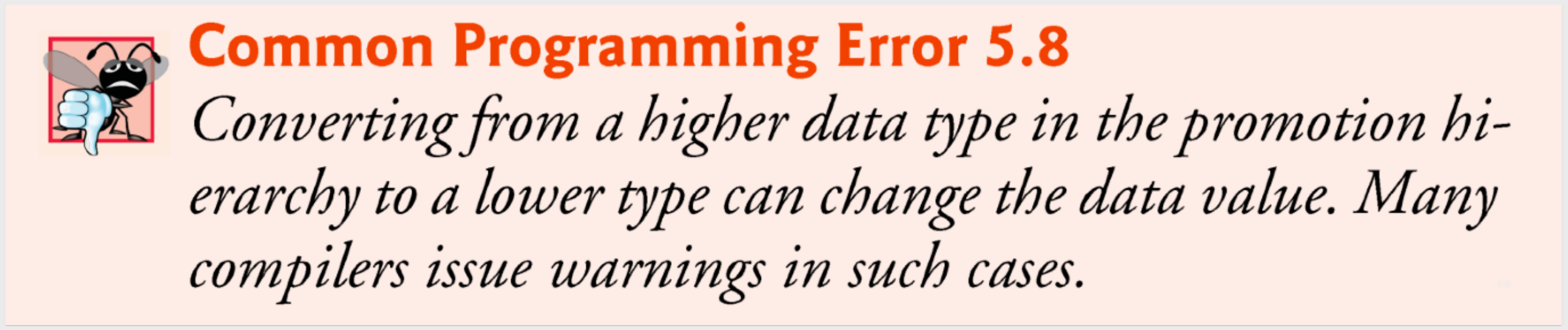


Data type	`printf` conversion specification	`scanf` conversion specification
`long double`	`%Lf`	`%Lf`
`double`	`%f`	`%lf`
`float`	`%f`	`%f`
`unsigned long int`	`%lu`	`%lu`
`long int`	`%ld`	`%ld`
`unsigned int`	`%u`	`%u`
`int`	`%d`	`%d`
`unsigned short`	`%hu`	`%hu`
`short`	`%hd`	`%hd`
`char`	`%c`	`%c`

Fig. 5.5 | Promotion hierarchy for data types.

Function Prototypes (Cont.)

- Converting values to lower types normally results in an incorrect value.
- Therefore, a value can be converted to a lower type only by explicitly assigning the value to a variable of lower type, or by using a cast operator.

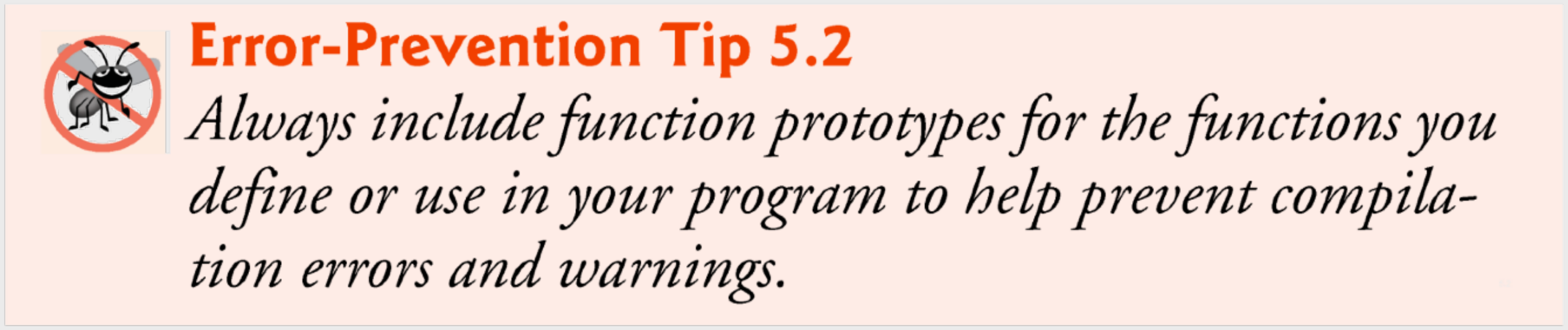


Common Programming Error 5.8

Converting from a higher data type in the promotion hierarchy to a lower type can change the data value. Many compilers issue warnings in such cases.

Function Prototypes (Cont.)

- If there is no function prototype for a function, the compiler forms its own function prototype using the first occurrence of the function—either the function definition or a call to the function.
- This typically leads to warnings or errors, depending on the compiler.



Error-Prevention Tip 5.2

Always include function prototypes for the functions you define or use in your program to help prevent compilation errors and warnings.

Function Call Stack and Activation Records

- To understand how C performs function calls, we first need to consider a data structure (i.e., collection of related data items) known as a **stack**.
- Stacks are known as **last-in, first-out (LIFO)** data structures—the last item pushed (inserted) on the stack is the first item popped (removed) from the stack.
- When a program calls a function, the called function must know how to return to its caller, so the return address of the calling function is pushed onto the **program execution stack** (sometimes referred to as the **function call stack**).

Function Call Stack and Activation Records (Cont.)

- This data, stored as a portion of the program execution stack, is known as the **activation record** or **stack frame** of the function call.
- When the function returns to its caller, the activation record for this function call is popped off the stack and those local variables are no longer known to the program.
- If more function calls occur than can have their activation records stored on the program execution stack, an error known as a **stack overflow** occurs.

Recursion

- A **recursive** function is a function that **calls itself** either directly or indirectly through another function.
- The factorial of a nonnegative integer n, written n! (and pronounced "n factorial"), is the product

  $n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot 1$

  with 1! equal to 1, and 0! defined to be 1.
- The factorial of an integer, number, greater than or equal to 0 can be calculated iteratively (**nonrecursively**) using a for statement as follows:

```
factorial = 1;
for ( counter = number; counter >= 1; counter-- )
     factorial *= counter;
```

Recursion (Cont.)

- A recursive definition of the factorial function is arrived at by observing the following relationship:
  $n! = n \cdot (n - 1)!$

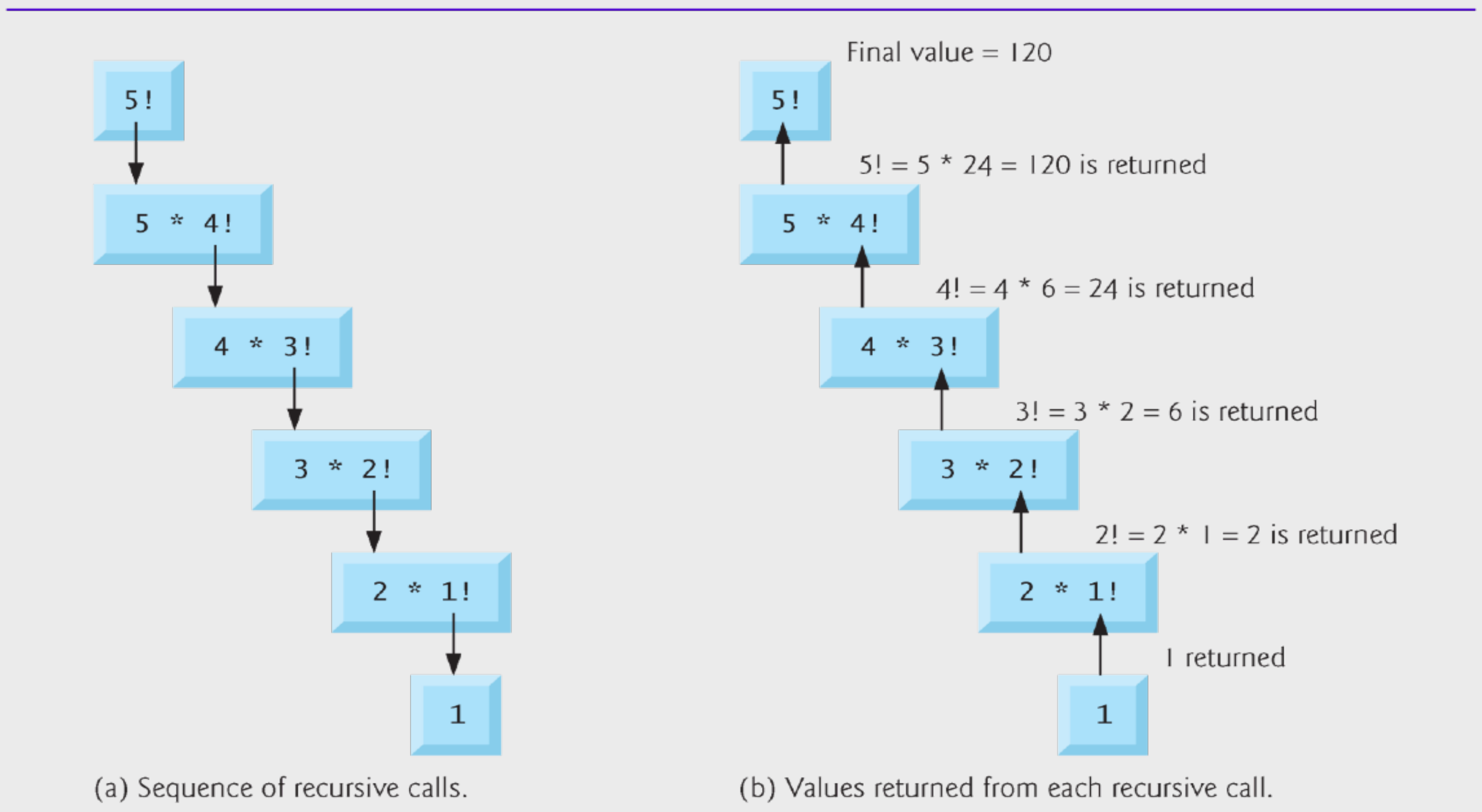
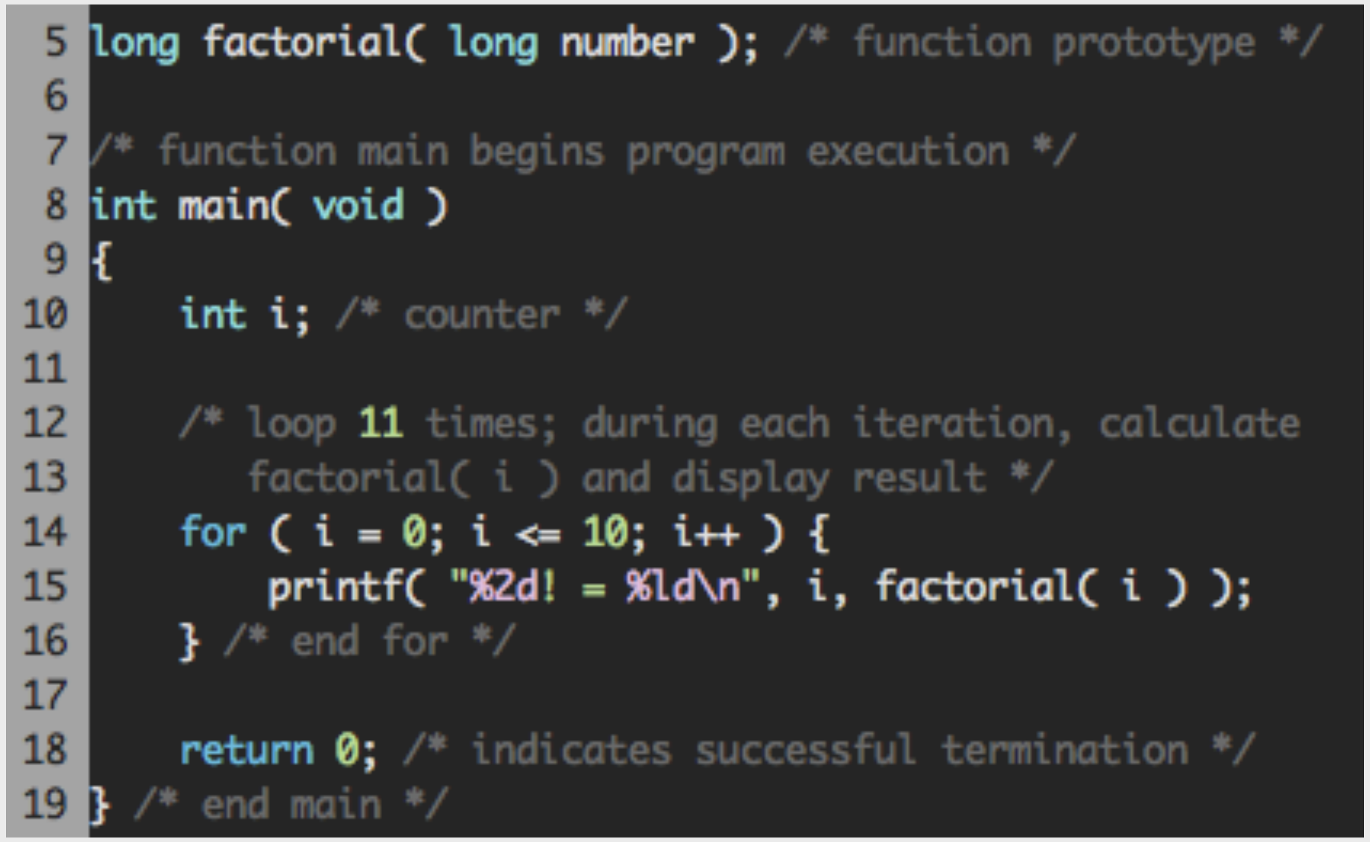

5!
5 * 4!
4 * 3!
3 * 2!
2 * 1!
1
(a) Sequence of recursive calls.
Final value = 120
5! = 5 * 24 = 120 is returned
4! = 4 * 6 = 24 is returned
3! = 3 * 2 = 6 is returned
2! = 2 * 1 = 2 is returned
1 returned
(b) Values returned from each recursive call.

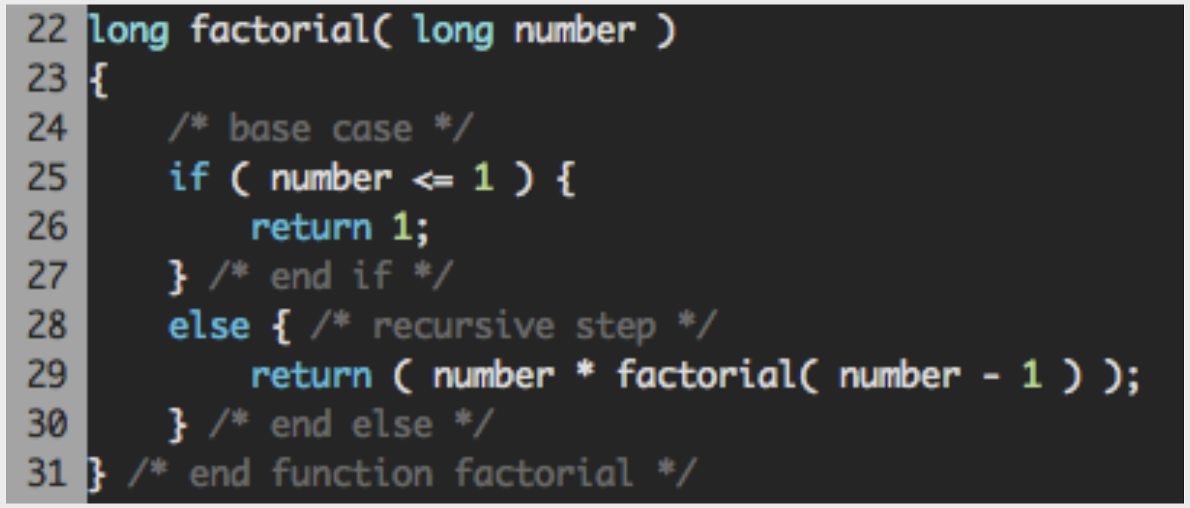
Fig. 5.13 | Recursive evaluation of 5!.

Recursion (Cont.)

- Example: fig05_14.c



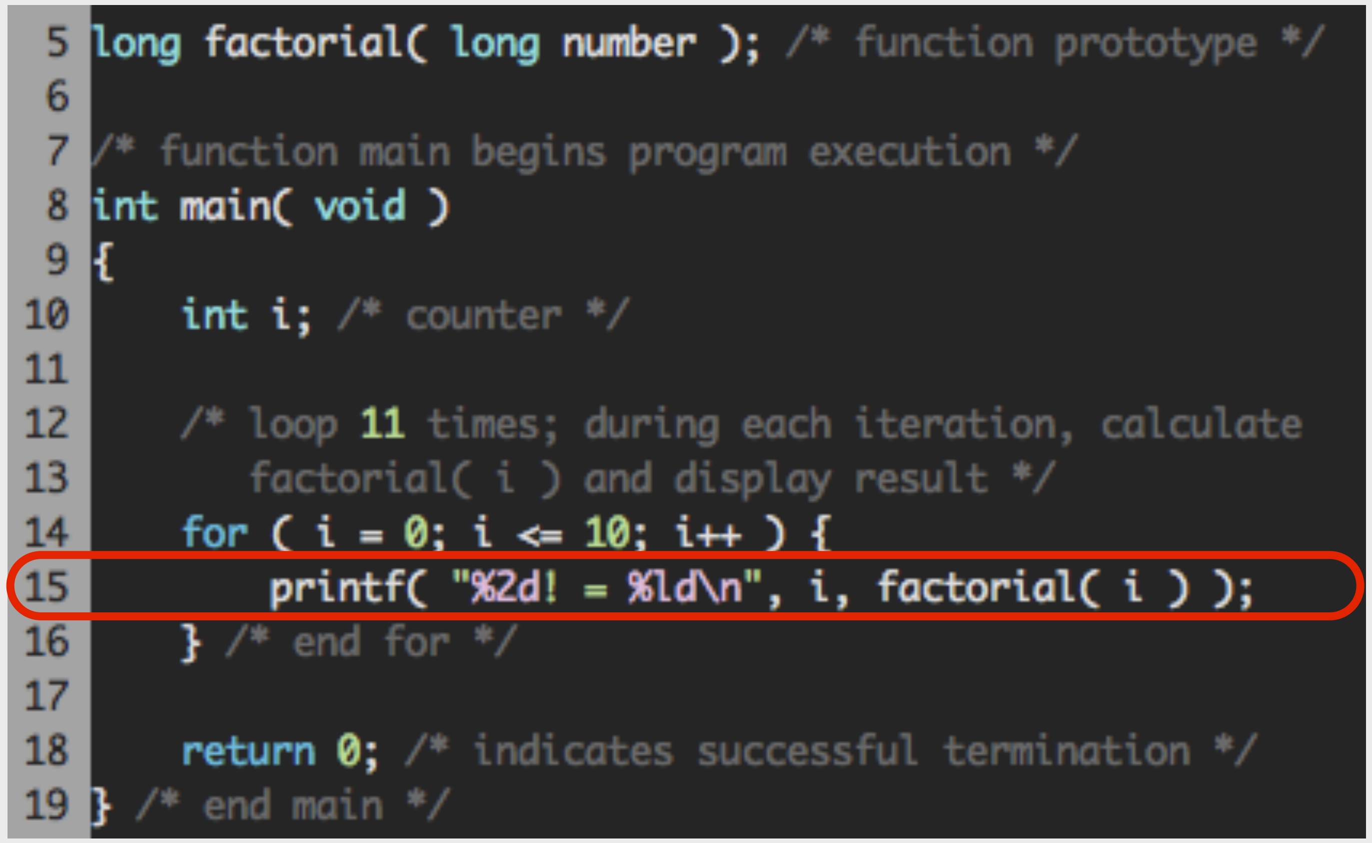
```
long factorial( long number ); /* function prototype */

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

    /* loop 11 times; during each iteration, calculate
       factorial( i ) and display result */
    for ( i = 0; i <= 10; i++ ) {
        printf( "%2d! = %ld\n", i, factorial( i ) );
    } /* end for */

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



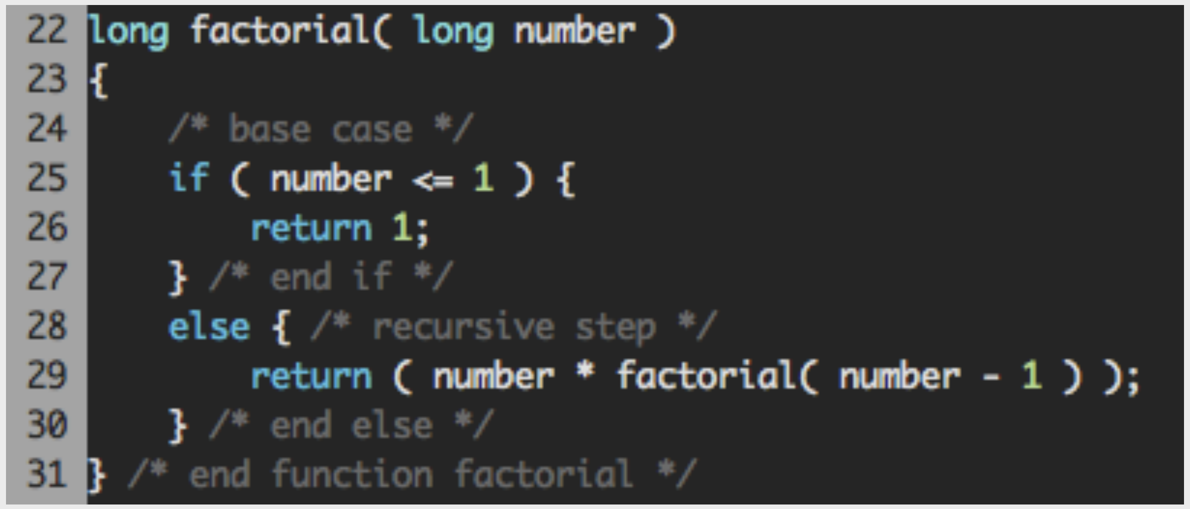
```
long factorial( long number )
{
    /* base case */
    if ( number <= 1 ) {
        return 1;
    } /* end if */
    else { /* recursive step */
        return ( number * factorial( number - 1 ) );
    } /* end else */
} /* end function factorial */
```

Recursion (Cont.)

- Example: fig05_14.c



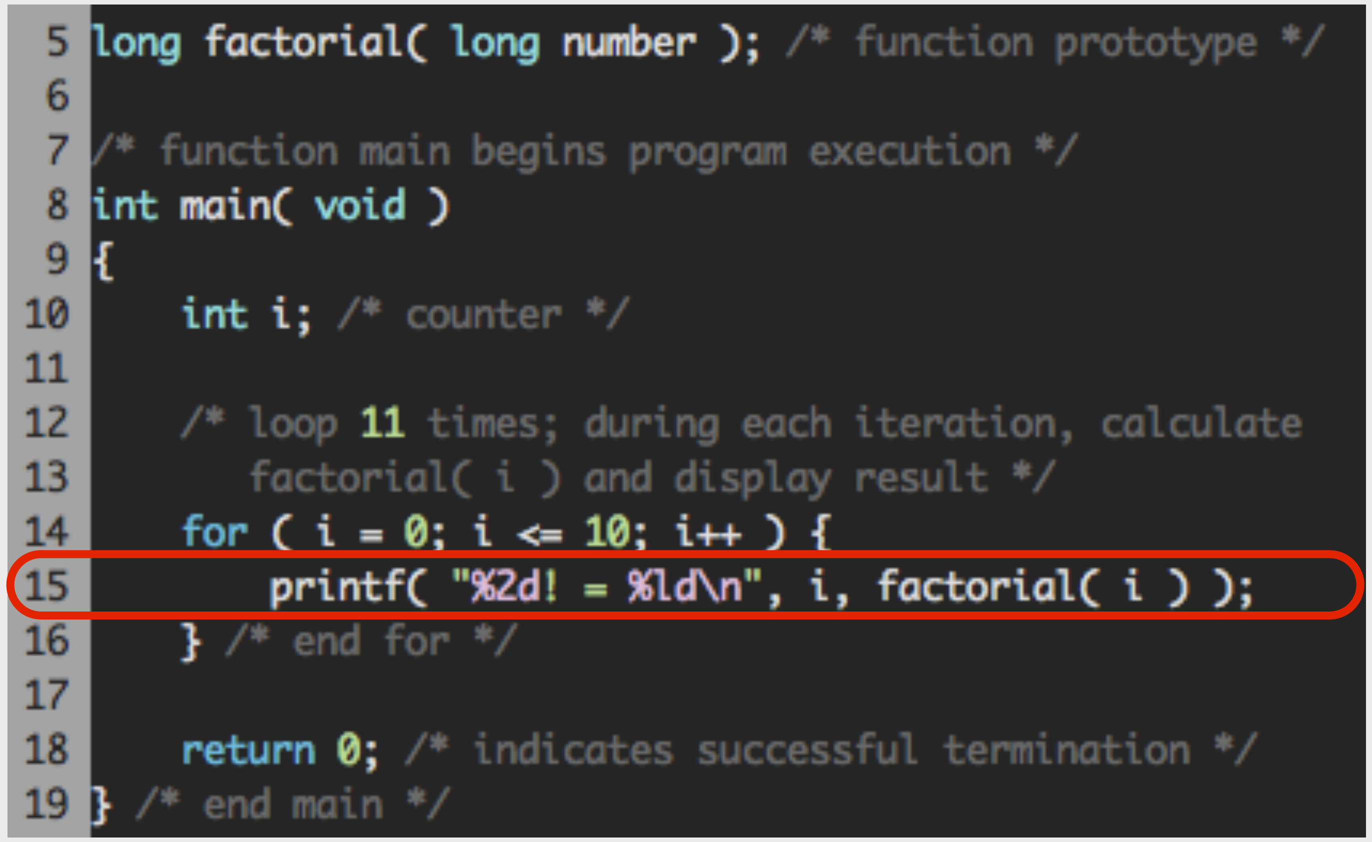
```
long factorial( long number ); /* function prototype */

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

    /* loop 11 times; during each iteration, calculate
       factorial( i ) and display result */
    for ( i = 0; i <= 10; i++ ) {
        printf( "%2d! = %ld\n", i, factorial( i ) );
    } /* end for */

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



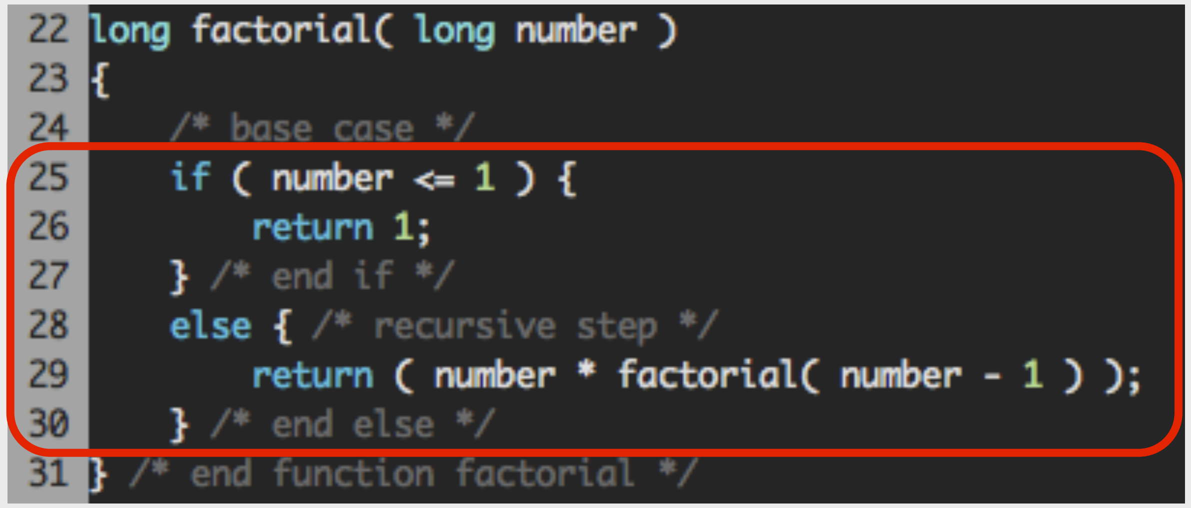
```
long factorial( long number )
{
    /* base case */
    if ( number <= 1 ) {
        return 1;
    } /* end if */
    else { /* recursive step */
        return ( number * factorial( number - 1 ) );
    } /* end else */
} /* end function factorial */
```

Recursion (Cont.)

- Example: fig05_14.c



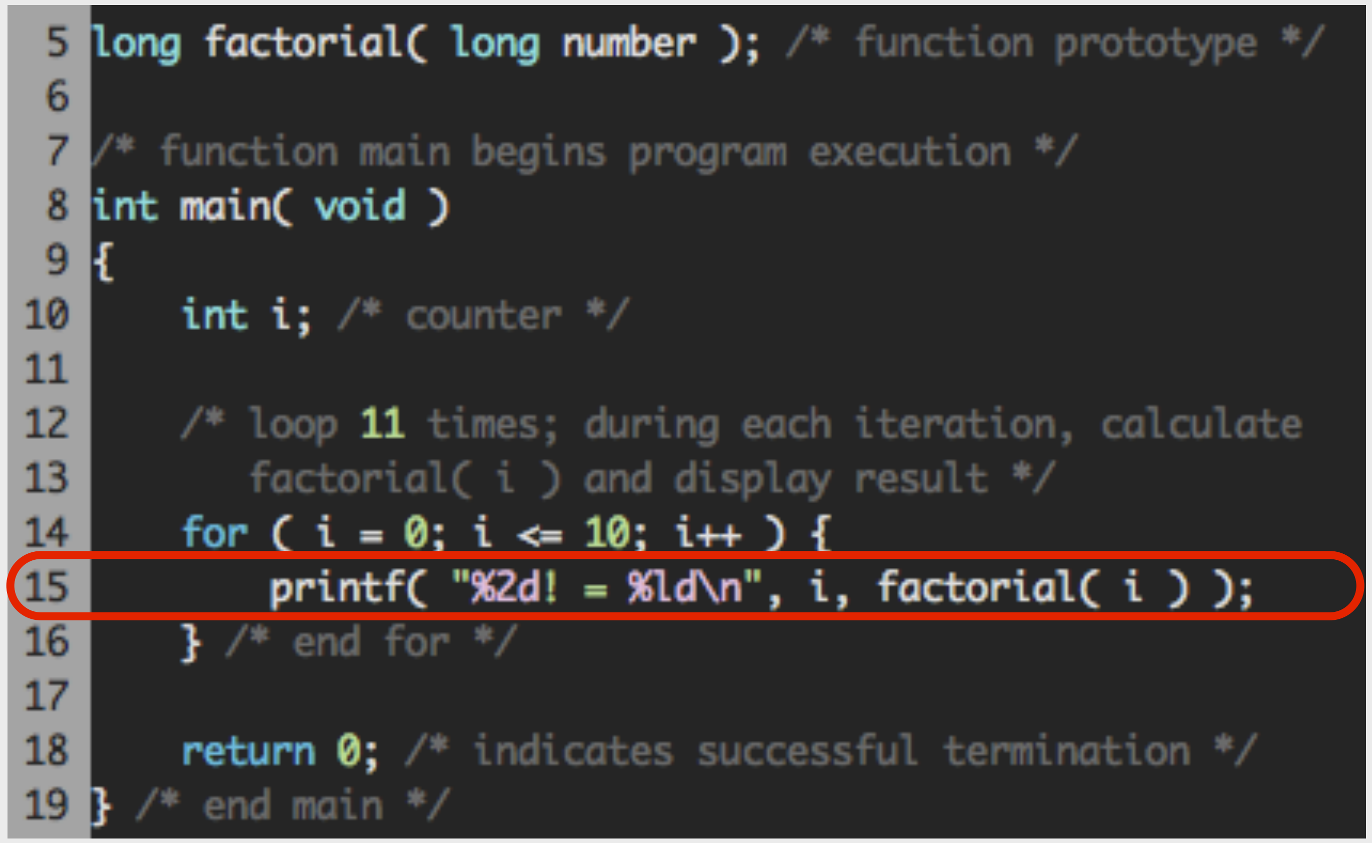
```
long factorial( long number ); /* function prototype */

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

    /* loop 11 times; during each iteration, calculate
       factorial( i ) and display result */
    for ( i = 0; i <= 10; i++ ) {
        printf( "%2d! = %ld\n", i, factorial( i ) );
    } /* end for */

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



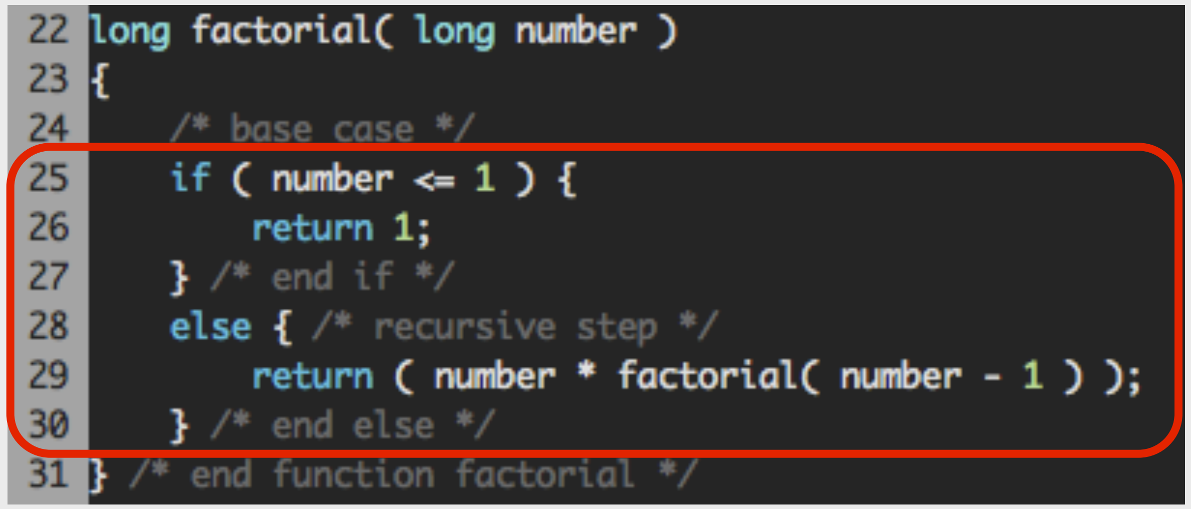
```
long factorial( long number )
{
    /* base case */
    if ( number <= 1 ) {
        return 1;
    } /* end if */
    else { /* recursive step */
        return ( number * factorial( number - 1 ) );
    } /* end else */
} /* end function factorial */
```

Recursion (Cont.)

- Example: fig05_14.c



```c
long factorial( long number ); /* function prototype */

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

    /* loop 11 times; during each iteration, calculate
       factorial( i ) and display result */
    for ( i = 0; i <= 10; i++ ) {
        printf( "%2d! = %ld\n", i, factorial( i ) );
    } /* end for */

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

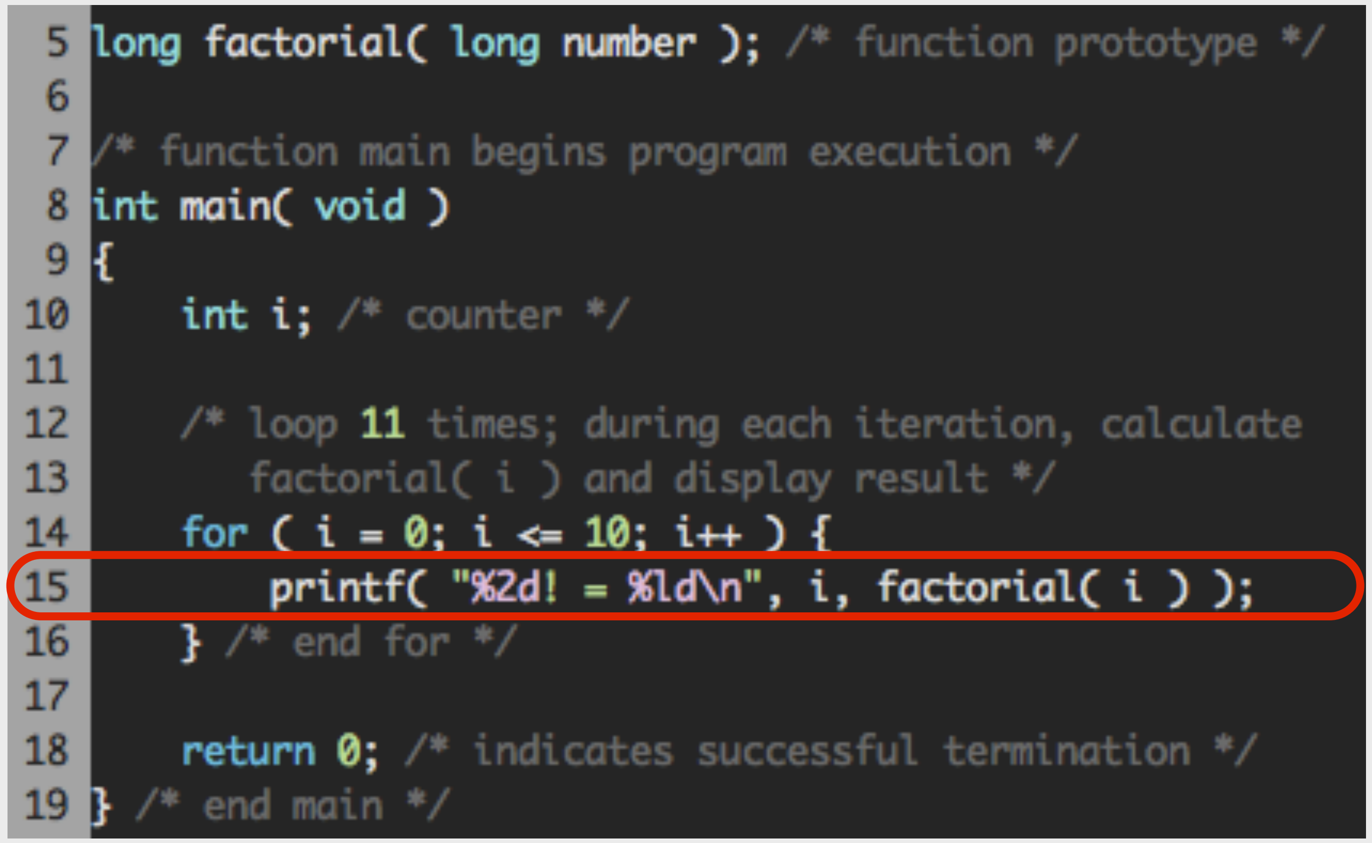


```c
long factorial( long number )
{
    /* base case */
    if ( number <= 1 ) {
        return 1;
    } /* end if */
    else { /* recursive step */
        return ( number * factorial( number - 1 ) );
    } /* end else */
} /* end function factorial */
```

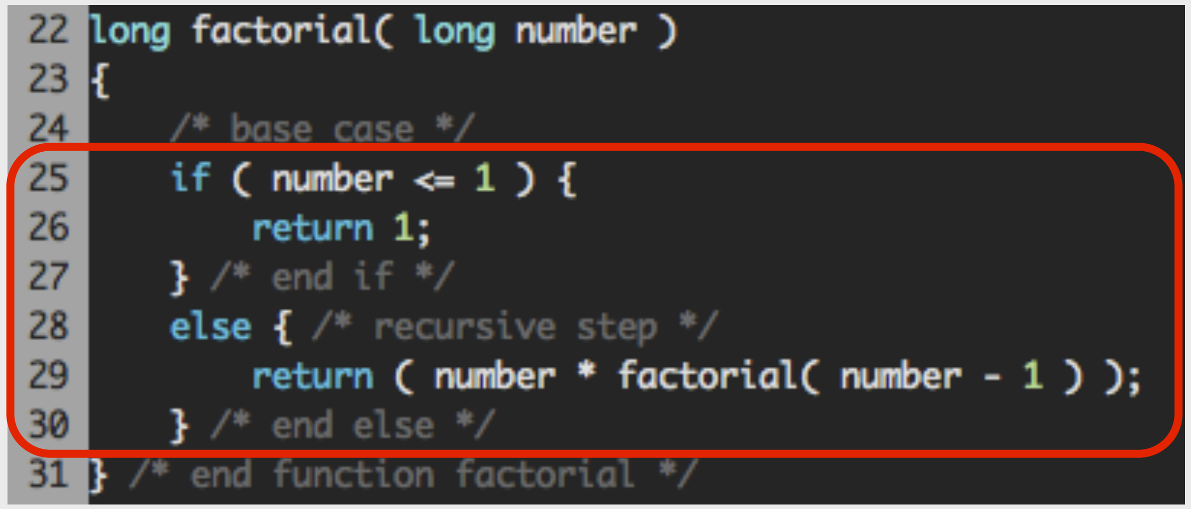
$$n! = \begin{cases} 1 & \text{if } n = 0, \\ (n-1)! \times n & \text{if } n > 0. \end{cases}$$

Recursion (Cont.)

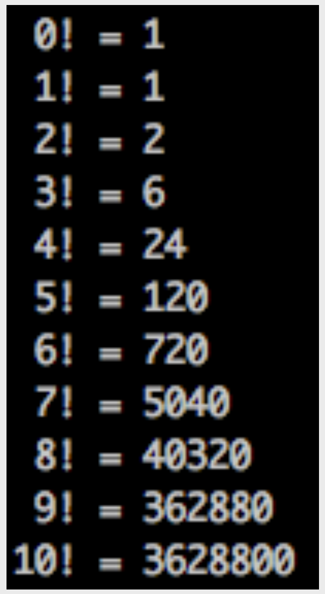
- Example: fig05_14.c



```c
long factorial( long number ); /* function prototype */

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

    /* loop 11 times; during each iteration, calculate
       factorial( i ) and display result */
    for ( i = 0; i <= 10; i++ ) {
        printf( "%2d! = %ld\n", i, factorial( i ) );
    } /* end for */

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



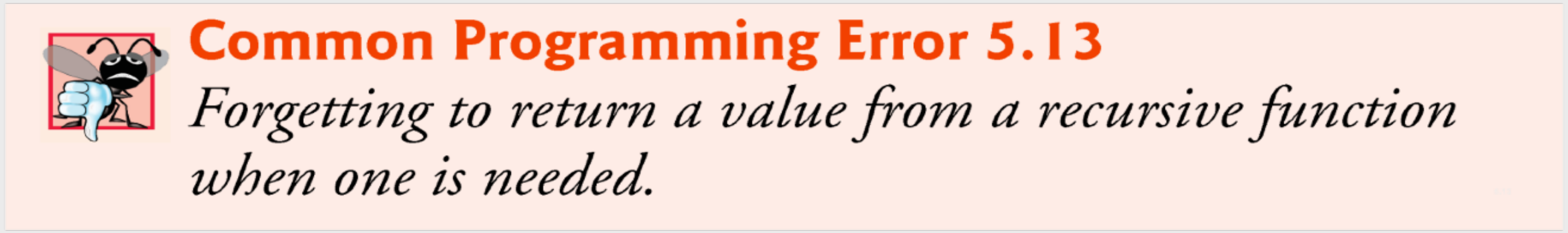
```c
long factorial( long number )
{
    /* base case */
    if ( number <= 1 ) {
        return 1;
    } /* end if */
    else { /* recursive step */
        return ( number * factorial( number - 1 ) );
    } /* end else */
} /* end function factorial */
```



```
 0! = 1
 1! = 1
 2! = 2
 3! = 6
 4! = 24
 5! = 120
 6! = 720
 7! = 5040
 8! = 40320
 9! = 362880
10! = 3628800
```

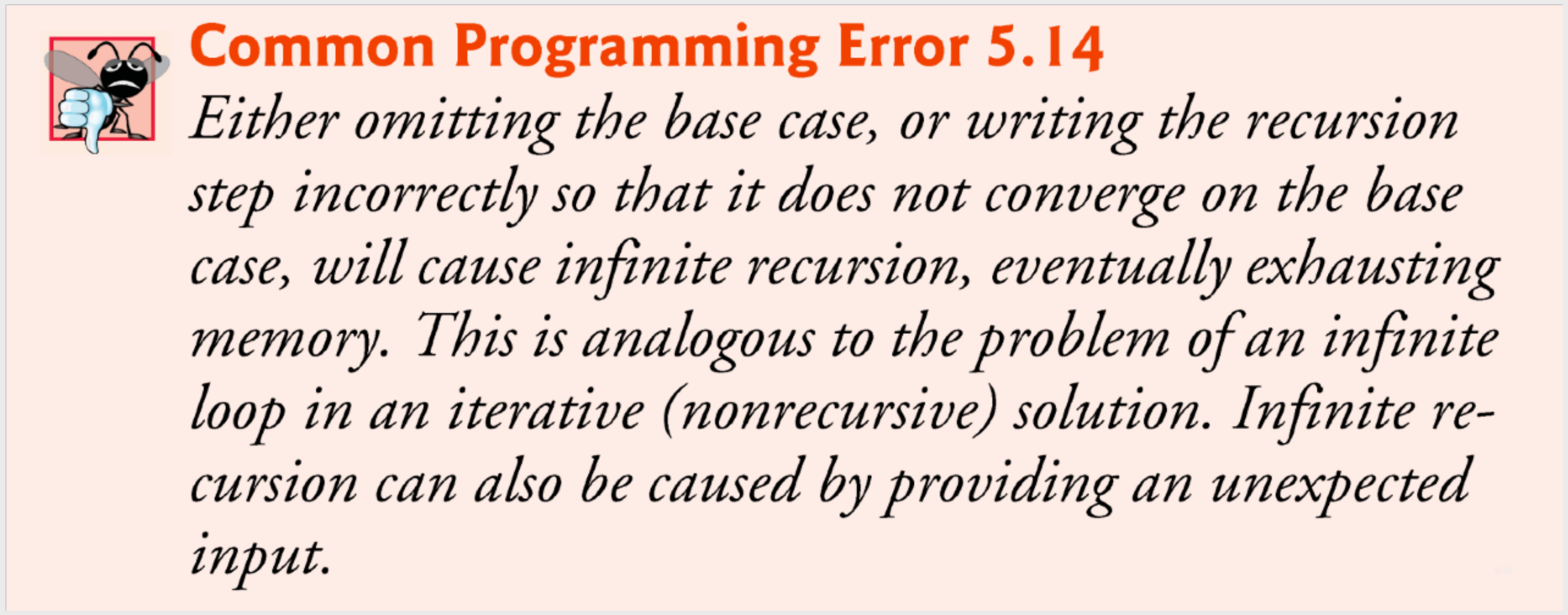
$$n! = \begin{cases} 1 & \text{if } n = 0, \\ (n-1)! \times n & \text{if } n > 0. \end{cases}$$

Recursion (Cont.)



Common Programming Error 5.13

Forgetting to return a value from a recursive function when one is needed.



Common Programming Error 5.14

Either omitting the base case, or writing the recursion step incorrectly so that it does not converge on the base case, will cause infinite recursion, eventually exhausting memory. This is analogous to the problem of an infinite loop in an iterative (nonrecursive) solution. Infinite recursion can also be caused by providing an unexpected input.

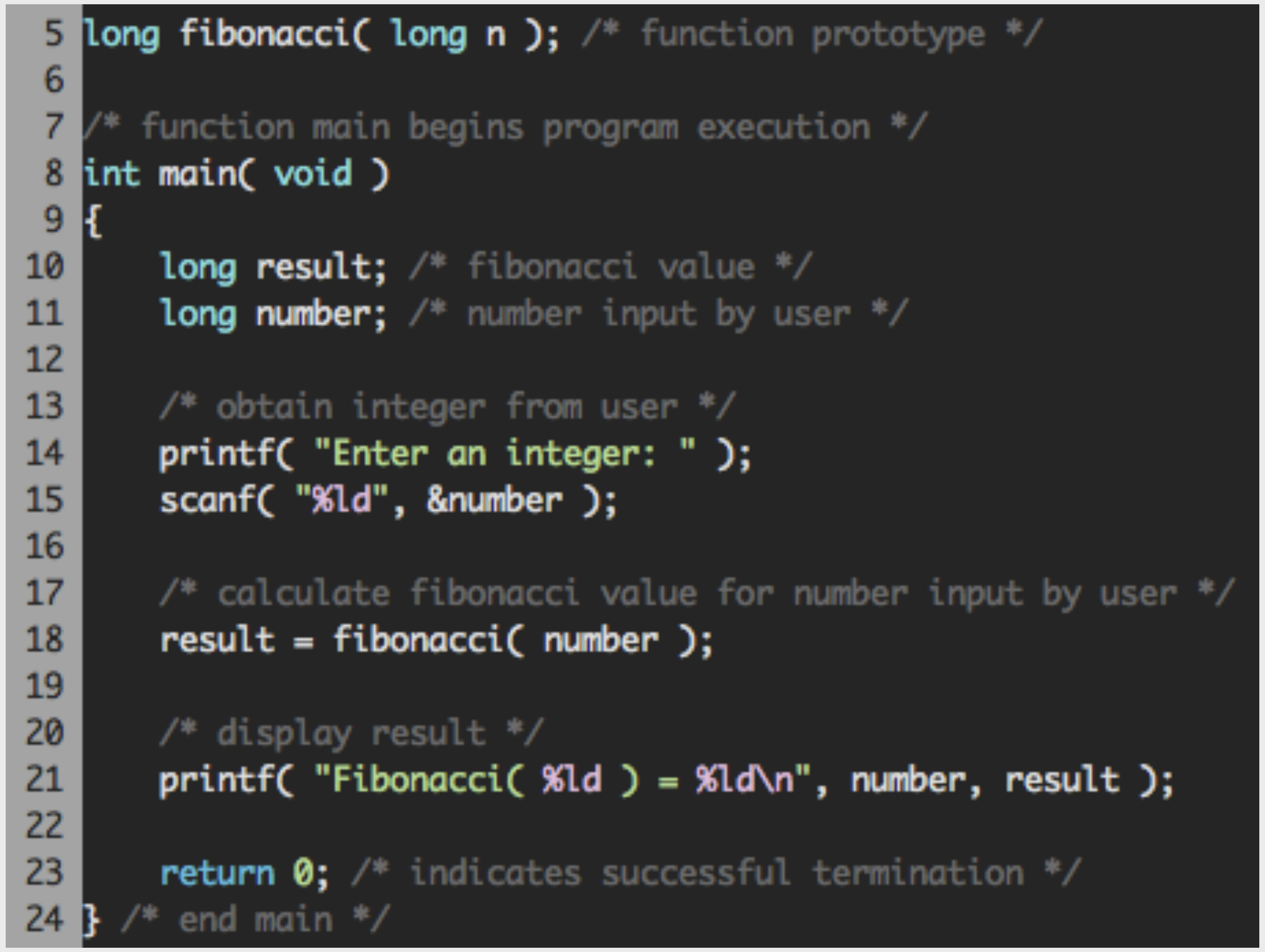
Example Using Recursion: Fibonacci Series

- The Fibonacci series
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, …
- The ratio of successive Fibonacci numbers converges to a constant value of **1.618**….
- The Fibonacci series may be defined recursively as follows:

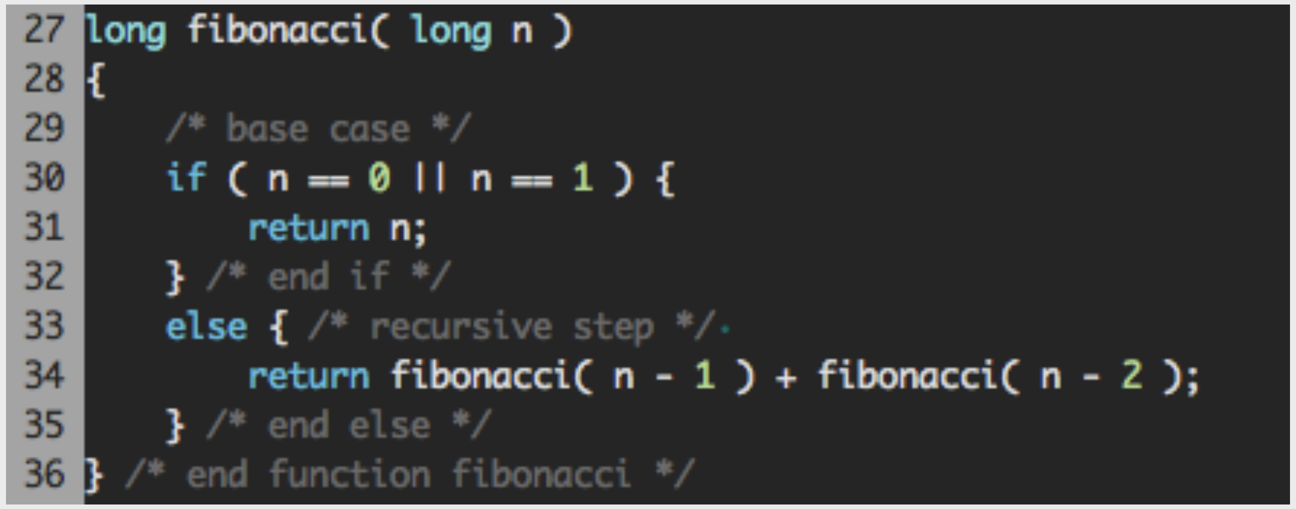
```
fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(n) = fibonacci(n – 1) + fibonacci(n –
2)
```

Example Using Recursion: Fibonacci Series (Cont.)

- Example: fig05_15.c



```c
long fibonacci( long n ); /* function prototype */

/* function main begins program execution */
int main( void )
{
    long result; /* fibonacci value */
    long number; /* number input by user */

    /* obtain integer from user */
    printf( "Enter an integer: " );
    scanf( "%ld", &number );

    /* calculate fibonacci value for number input by user */
    result = fibonacci( number );

    /* display result */
    printf( "Fibonacci( %ld ) = %ld\n", number, result );

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

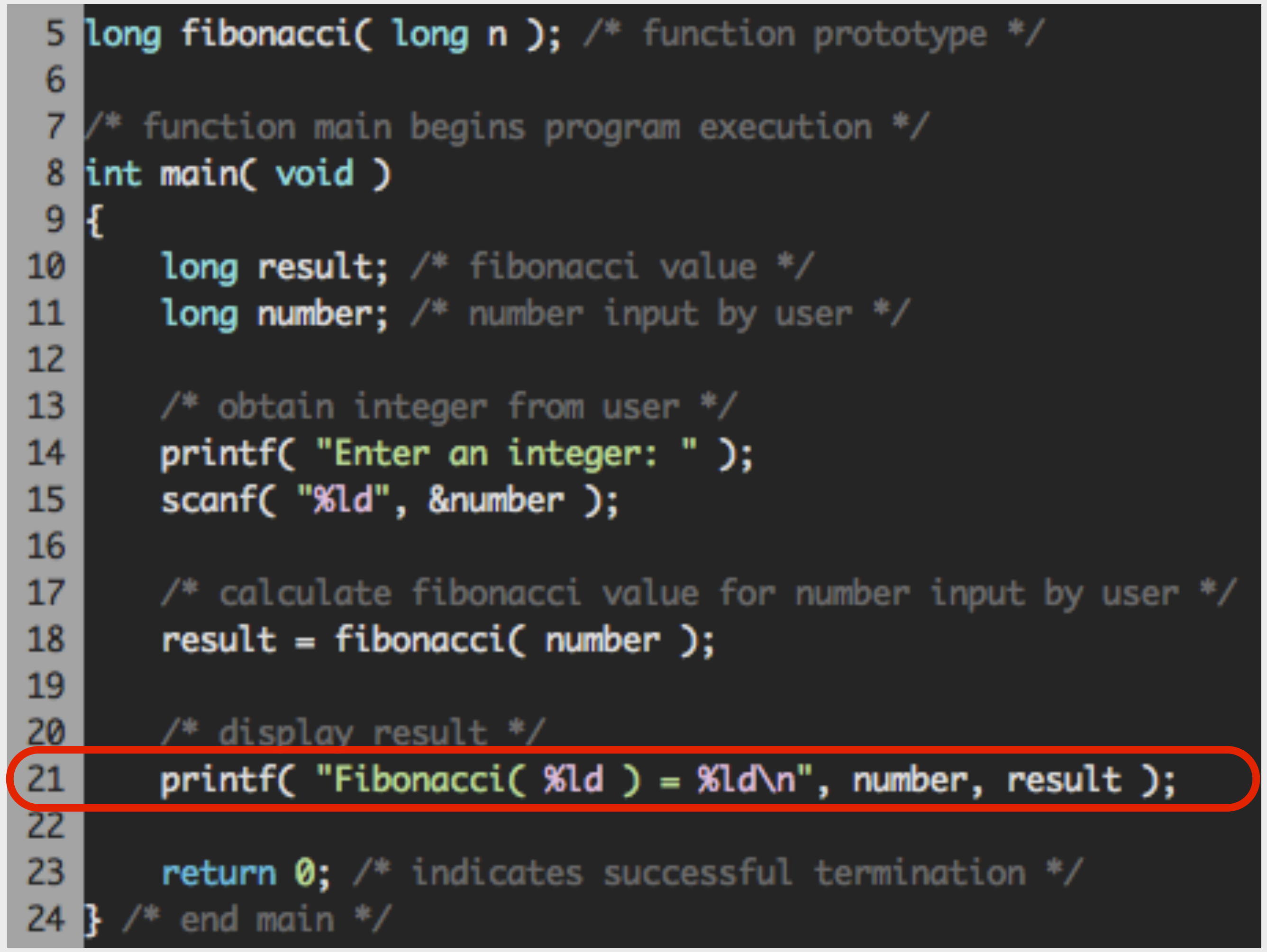


```c
long fibonacci( long n )
{
    /* base case */
    if ( n == 0 || n == 1 ) {
        return n;
    } /* end if */
    else { /* recursive step */
        return fibonacci( n - 1 ) + fibonacci( n - 2 );
    } /* end else */
} /* end function fibonacci */
```

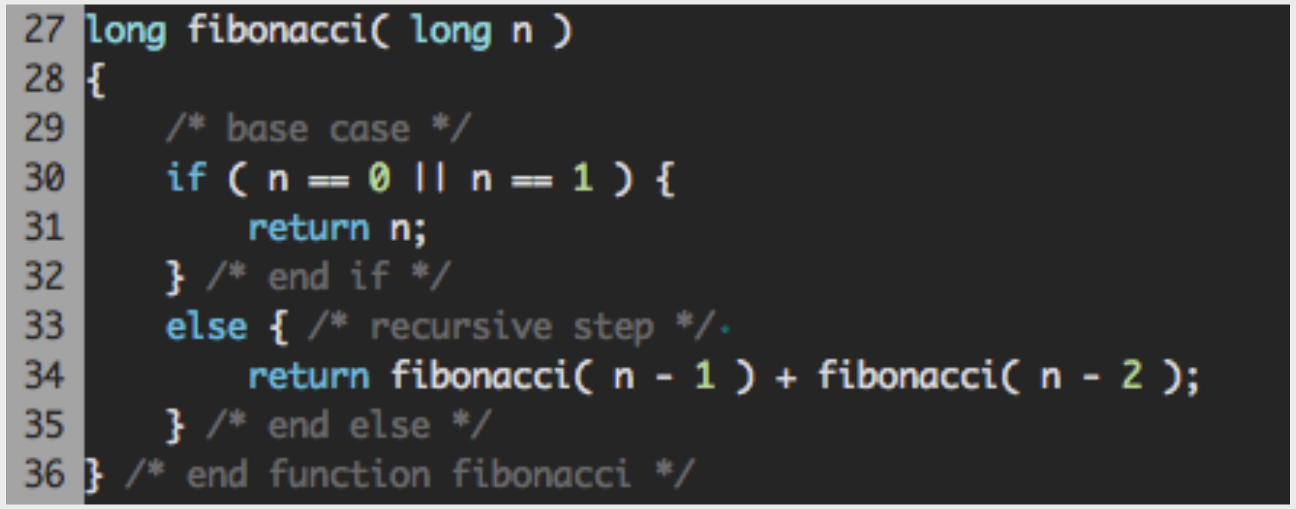
$$F_n = F_{n-1} + F_{n-2} \text{ if } n > 1$$

Example Using Recursion: Fibonacci Series (Cont.)

- Example: fig05_15.c



```c
long fibonacci( long n ); /* function prototype */

/* function main begins program execution */
int main( void )
{
    long result; /* fibonacci value */
    long number; /* number input by user */

    /* obtain integer from user */
    printf( "Enter an integer: " );
    scanf( "%ld", &number );

    /* calculate fibonacci value for number input by user */
    result = fibonacci( number );

    /* display result */
    printf( "Fibonacci( %ld ) = %ld\n", number, result );

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

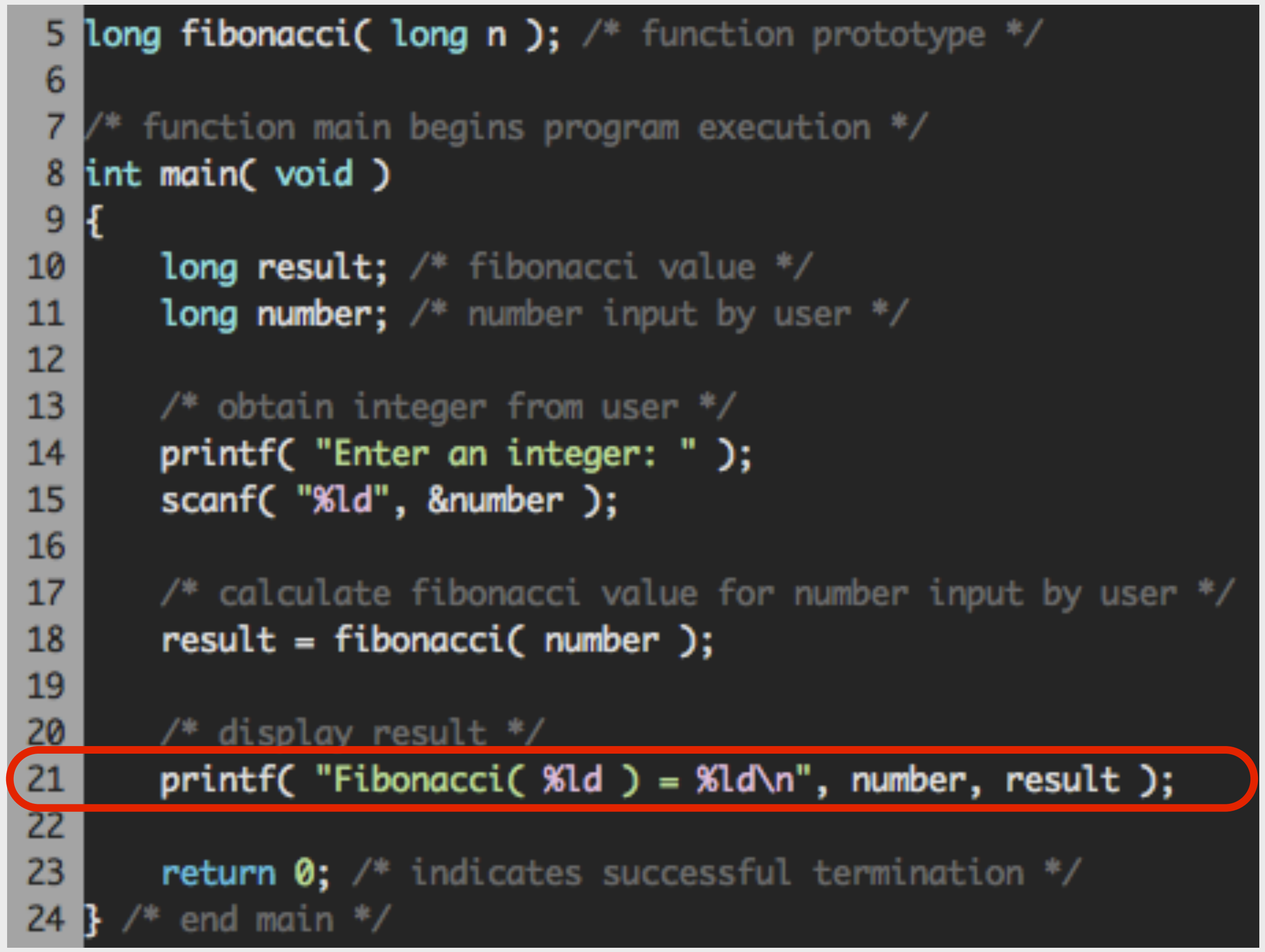


```c
long fibonacci( long n )
{
    /* base case */
    if ( n == 0 || n == 1 ) {
        return n;
    } /* end if */
    else { /* recursive step */
        return fibonacci( n - 1 ) + fibonacci( n - 2 );
    } /* end else */
} /* end function fibonacci */
```

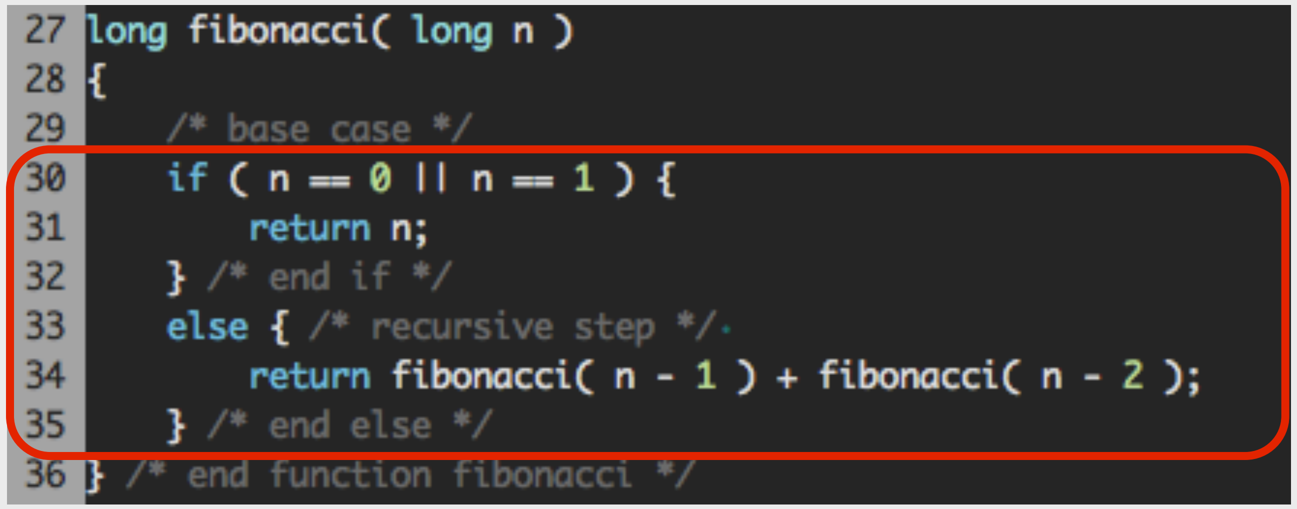
$$F_n = F_{n-1} + F_{n-2} \text{ if } n > 1$$

Example Using Recursion: Fibonacci Series (Cont.)

- Example: fig05_15.c



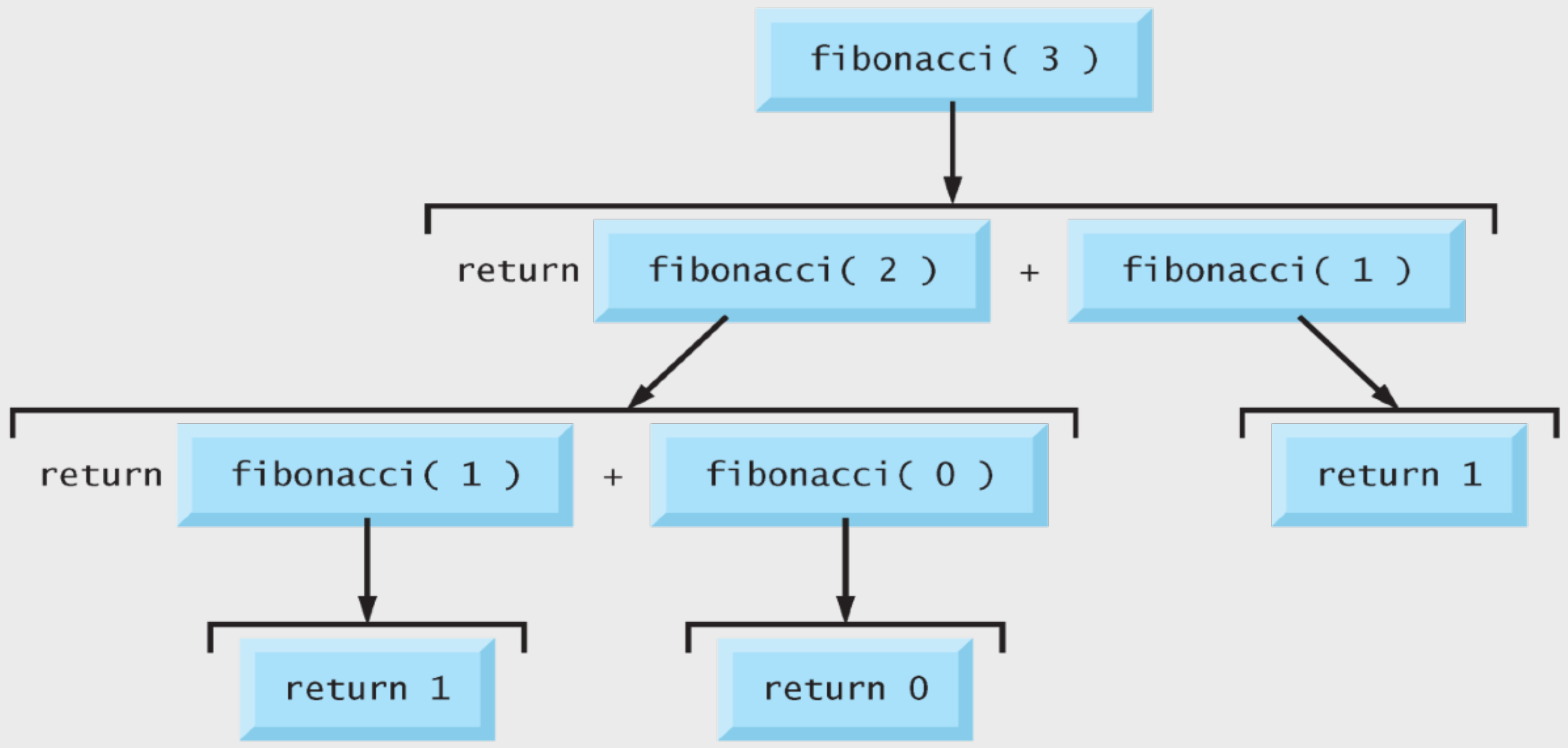
```c
long fibonacci( long n ); /* function prototype */

/* function main begins program execution */
int main( void )
{
    long result; /* fibonacci value */
    long number; /* number input by user */

    /* obtain integer from user */
    printf( "Enter an integer: " );
    scanf( "%ld", &number );

    /* calculate fibonacci value for number input by user */
    result = fibonacci( number );

    /* display result */
    printf( "Fibonacci( %ld ) = %ld\n", number, result );

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



```c
long fibonacci( long n )
{
    /* base case */
    if ( n == 0 || n == 1 ) {
        return n;
    } /* end if */
    else { /* recursive step */
        return fibonacci( n - 1 ) + fibonacci( n - 2 );
    } /* end else */
} /* end function fibonacci */
```

$$F_n = F_{n-1} + F_{n-2} \text{ if } n > 1$$

Example Using Recursion: Fibonacci Series (Cont.)

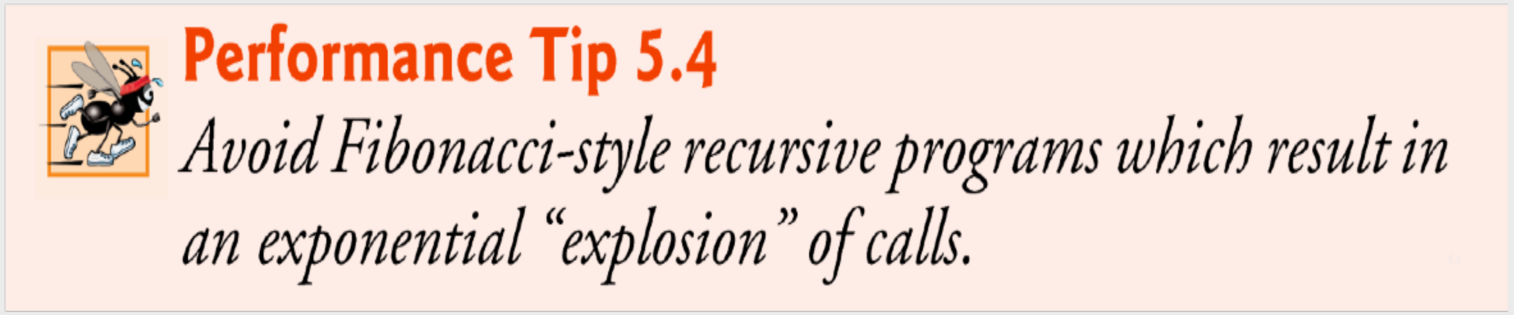


fibonacci(3)
return fibonacci(2) + fibonacci(1)
return fibonacci(1) + fibonacci(0)
return 1
return 1
return 0

Fig. 5.16 | Set of recursive calls for `fibonacci( 3 )`.

Example Using Recursion: Fibonacci Series (Cont.)

- Computer scientists refer to this as **exponential complexity**.
- Complexity issues in general, and exponential complexity in particular, are discussed in detail in the upper-level computer science curriculum course generally called "**Algorithms**."



Performance Tip 5.4

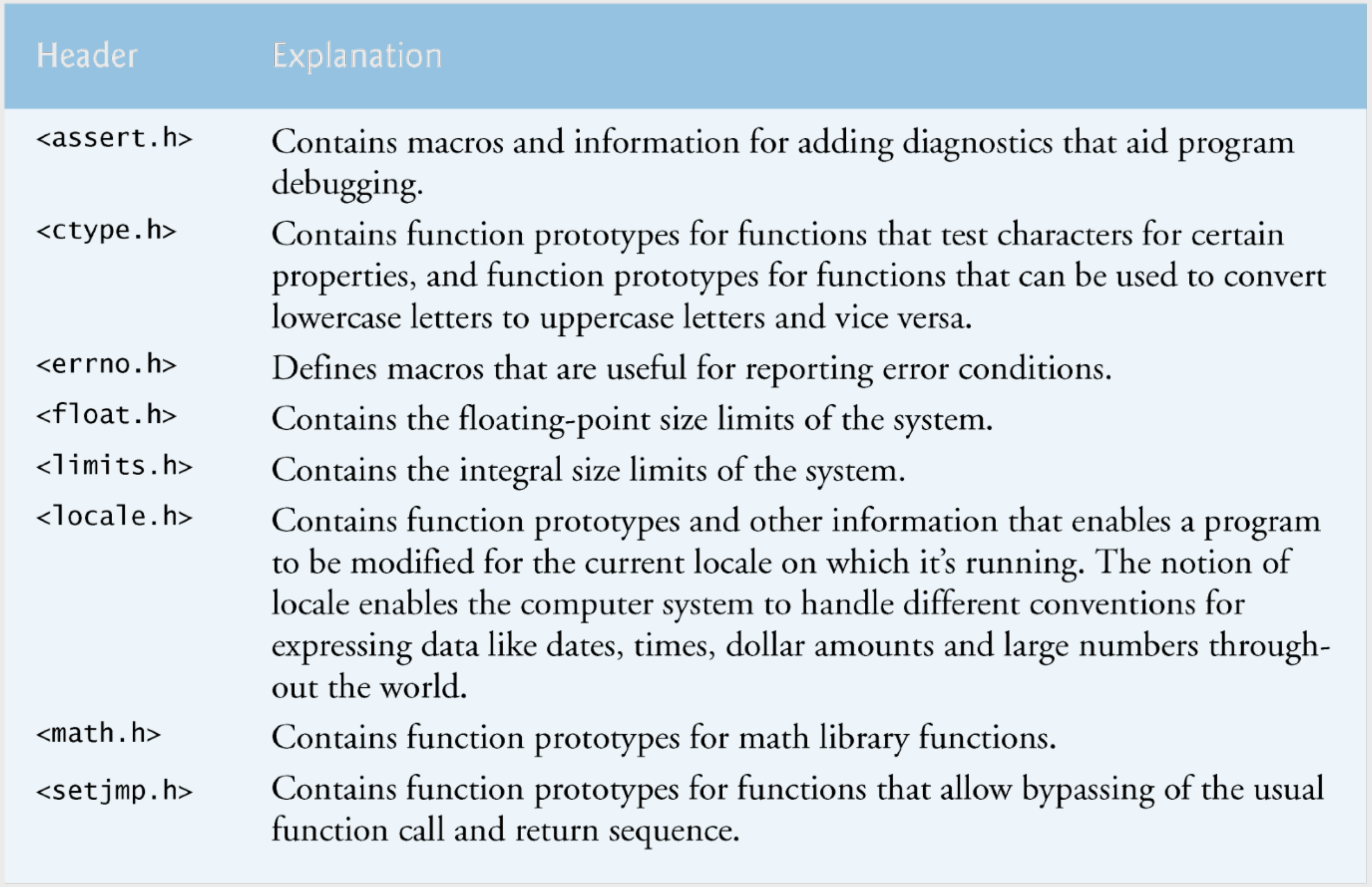
Avoid Fibonacci-style recursive programs which result in an exponential "explosion" of calls.

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"
```

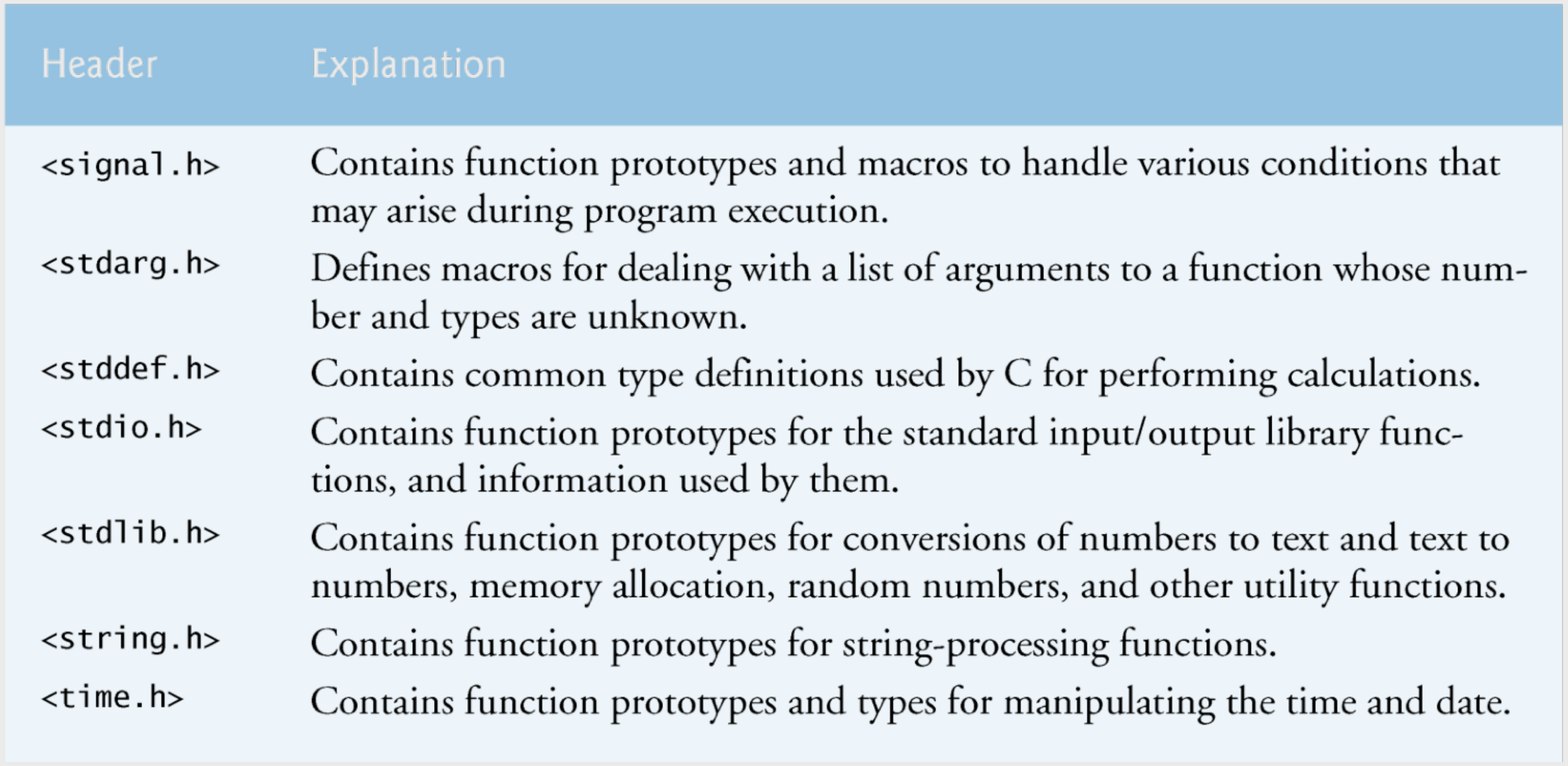
Headers (Cont.)



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.)

Vim Tips

- A minimalist Vim plugin manager
  - vim-plug
  - Useful links
    - Vim Awesome
    - 優秀 Vim 外掛幫你打造完美 IDE