BAKI ALİ NEFT MƏKTƏBİ
BAKU HIGHER OIL SCHOOL

Programming and Computer Applications-1

Functions

Instructor : PhD, Associate Professor Leyla Muradkhanli

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` that we've used in previous chapters are Standard Library functions.
- You can write your own functions to define tasks that may be used at many points in a program.
- These are sometimes referred to as programmer-defined functions.
- The statements defining the function are written only once, and the statements are hidden from other 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.)

- A common analogy for this is the hierarchical form of management.
- A boss (the calling function or caller) asks a worker (the called function) to perform a task and report back when the task is done (Fig. 5.1).
- For example, a function needing to display information on the screen calls the worker function `printf` to perform that task, then `printf` displays the information and reports back—or returns—to the calling function when its task is completed.
- The boss function does not know how the worker function performs its designated tasks.

Program Modules in C (Cont.)

- The worker may call other worker functions, and the boss will be unaware of this.
- 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`.
- Relationships among functions may differ from the hierarchical structure shown in this figure.

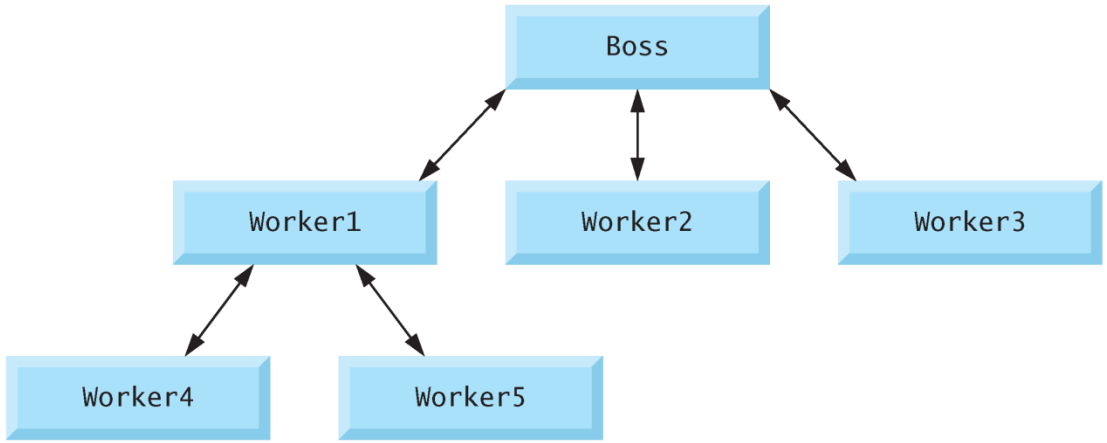


Fig. 5.1 | Hierarchical boss function/worker function relationship.

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, a programmer desiring to calculate and print the square root of `900.0` might write

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

- When this statement executes, the math library function `sqrt` is called to calculate the square root of the number contained in the parentheses (`900.0`).

Math Library Functions (Cont.)

- The number `900.0` is the argument of the `sqrt` function.
- The preceding statement would print `30.00`.
- The `sqrt` function takes an argument of type `double` and returns a result of type `double`.
- All functions in the math library that return floating point values return the data type `double`.
- Note that `double` values, like `float` values, can be output using the `%f` conversion specification.

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 ) );
```

- calculates and prints the square root of `13.0 + 3.0 * 4.0 = 25.0`, namely `5.00`.
- Some C math library functions are summarized in Fig. 5.2.
- In the figure, the variables `x` and `y` are of type `double`.

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.)

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.)

Math Library Functions (Cont.)

```c
#include <stdio.h>
#include <conio.h>
#include <math.h>
int main()
{
   // calculates and outputs the square root
   printf("sqrt(%.1f) = %.1f\n", 900.0, sqrt(900.0));
   printf("sqrt(%.1f) = %.1f\n", 9.0, sqrt(9.0));
   // calculates and outputs the exponential function e to the x
   printf("exp(%.1f) = %f\n", 1.0, exp(1.0));
   printf("exp(%.1f) = %f\n", 2.0, exp(2.0));
   // calculates and outputs the logarithm (base e)
   printf("log(%f) = %.1f\n", 2.718282, log(2.718282));
   printf("log(%f) = %.1f\n", 7.389056, log(7.389056));
   // calculates and outputs the logarithm (base 10)
   printf("log10(%.1f) = %.1f\n", 1.0, log10(1.0));
   printf("log10(%.1f) = %.1f\n", 10.0, log10(10.0));
   printf("log10(%.1f) = %.1f\n", 100.0, log10(100.0));
```

Math Library Functions (Cont.)

```
  // calculates and outputs the absolute value
printf("fabs(%.1f) = %.1f\n", 13.5, fabs(13.5));
printf("fabs(%.1f) = %.1f\n", 0.0, fabs(0.0));
printf("fabs(%.1f) = %.1f\n", -13.5, fabs(-13.5));
// calculates and outputs ceil(x)
printf("ceil(%.1f) = %.1f\n", 9.2, ceil(9.2));
printf("ceil(%.1f) = %.1f\n", -9.8, ceil(-9.8));
// calculates and outputs floor(x)
printf("floor(%.1f) = %.1f\n", 9.2, floor(9.2));
printf("floor(%.1f) = %.1f\n", -9.8, floor(-9.8));
```

Math Library Functions (Cont.)

```
// calculates and outputs pow(x, y)
   printf("pow(%.1f, %.1f) = %.1f\n", 2.0, 7.0, pow(2.0, 7.0));
   printf("pow(%.1f, %.1f) = %.1f\n", 9.0, 0.5, pow(9.0, 0.5));
   // calculates and outputs fmod(x, y)
   printf("fmod(%.3f/%.3f) = %.3f\n", 13.657, 2.333,
      fmod(13.657, 2.333));
   // calculates and outputs sin(x)
   printf("sin(%.1f) = %.1f\n", 0.0, sin(0.0));
   // calculates and outputs cos(x)
   printf("cos(%.1f) = %.1f\n", 0.0, cos(0.0));
   // calculates and outputs tan(x)
   printf("tan(%.1f) = %.1f\n", 0.0, tan(0.0));
   getch();
}
```

Function Definitions

- Each program we've presented has consisted of a function called `main` that called standard library functions to accomplish its tasks.
- We now consider how to write custom functions.
- Consider a program that uses a function `square` to calculate and print the squares of the integers from 1 to 10 (Fig. 5.3).

```c
/* Fig. 5.3: fig05_03.c
   Creating and using a programmer-defined function */
#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 */

```

Fig. 5.3 | Using a programmer-defined function. (Part 1 of 2.)

```
21  /* square function definition returns square of parameter */
22  int square( int y ) /* y is a copy of argument to function */
23  {
24     return y * y; /* returns square of y as an int */
25  } /* end function square */
```

```
1  4  9  16  25  36  49  64  81  100
```

Fig. 5.3 | Using a programmer-defined function. (Part 2 of 2.)

Function Definitions (Cont.)

- Function `square` is invoked or called in `main` within the `printf` statement (line 14)

```
printf( "%d  ", square( x ) ); /* function call */
```

- Function `square` receives a copy of the value of `x` in the parameter `y` (line 22).
- Then `square` calculates `y * y` (line 24).
- The result is passed back to function `printf` in `main` where `square` was invoked (line 14), and `printf` displays the result.
- This process is repeated 10 times using the `for` repetition statement.

Function Definitions (Cont.)

- The definition of function `square` shows that `square` expects an integer parameter `y`.
- The keyword `int` preceding the function name (line 22) indicates that `square` returns an integer result.
- The `return` statement in `square` passes the result of the calculation back to the calling function.
- Line 5

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

  is a function prototype.
- The `int` in parentheses informs the compiler that `square` expects to receive an integer value from the caller.

Function Definitions (Cont.)

- The `int` to the left of the function name `square` informs the compiler that `square` returns an integer result to the caller.
- The compiler refers to the function prototype to check that calls to `square` (line 14) contain the correct return type, the correct number of arguments, the correct argument types, and that the arguments are in the correct order.
- 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.)

- The *definitions* and *statements* within braces form the function body.
- The function body is also referred to as a block.
- Variables can be declared in any block, and blocks can be nested.
- A function cannot be defined inside another function.

Function Definitions (Cont.)

- There are three ways to return control from a called function to the point at which a function was invoked.
- If the function does not return a result, control is returned simply when the function-ending right brace is reached, or by executing the statement

```
return;
```

- If the function does return a result, the statement

```
return expression;
```

- returns the value of *expression to the caller.*

Function Definitions (Cont.)

- Our second example uses a programmer-defined function `maximum` to determine and return the largest of three integers (Fig. 5.4).
- Three integers are passed to `maximum` (line 19), which determines the largest integer.
- This value is returned to main by the `return` statement in `maximum` (line 37).

```c
/* Fig. 5.4: fig05_04.c
   Finding the maximum of three integers */
#include <stdio.h>

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

```

Fig. 5.4 | Finding the maximum of three integers. (Part 1 of 4.)

```
 7  /* function main begins program execution */
 8  int main( void )
 9  {
10     int number1; /* first integer */
11     int number2; /* second integer */
12     int number3; /* third integer */
13
14     printf( "Enter three integers: " );
15     scanf( "%d%d%d", &number1, &number2, &number3 );
16
17     /* number1, number2 and number3 are arguments
18        to the maximum function call */
19     printf( "Maximum is: %d\n", maximum( number1, number2, number3 ) );
20     return 0; /* indicates successful termination */
21  } /* end main */
22
```

Fig. 5.4 | Finding the maximum of three integers. (Part 2 of 4.)

```
23  /* Function maximum definition */
24  /* x, y and z are parameters */
25  int maximum( int x, int y, int z )
26  {
27     int max = x; /* assume x is largest */
28
29     if ( y > max ) { /* if y is larger than max, assign y to max */
30        max = y;
31     } /* end if */
32
33     if ( z > max ) { /* if z is larger than max, assign z to max */
34        max = z;
35     } /* end if */
36
37     return max; /* max is largest value */
38  } /* end function maximum */
```

Fig. 5.4 | Finding the maximum of three integers. (Part 3 of 4.)

```
Enter three integers: 22 85 17
Maximum is: 85
```

```
Enter three integers: 85 22 17
Maximum is: 85
```

```
Enter three integers: 22 17 85
Maximum is: 85
```

Fig. 5.4 | Finding the maximum of three integers. (Part 4 of 4.)

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 Definitions (Cont.)

- The function prototype for `maximum` in Fig. 5.4 (line 5) is

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

- This function prototype states that `maximum` takes three arguments of type `int` and returns a result of type `int`.
- Notice that the function prototype is the same as the first line of the function definition of `maximum`.

Function Definitions (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.
- For example, in Fig. 5.4, if the function prototype had been written

```
void maximum( int x, int y, int z );
```

- the compiler would generate an error because the `void` return type in the function prototype would differ from the `int` return type in the function header.

Function Definitions (Cont.)

- Another important feature of function prototypes is the coercion of arguments, i.e., the forcing of arguments to the appropriate type.
- For example, the math library function `sqrt` can be called with an integer argument even though the function prototype in `<math.h>` specifies a `double` argument, and the function will still work correctly.
- The statement

```
printf( "%.3f\n", sqrt( 4 ) );
```

  correctly evaluates `sqrt( 4 )`, and prints the value `2.000`.

Function Definitions (Cont.)

- The function prototype causes the compiler to convert the integer value `4` to the `double` value `4.0` before the value is passed to `sqrt`.
- In general, argument values that do not correspond precisely to the parameter types in the function prototype are converted to the proper type before the function is called.
- These conversions can lead to incorrect results if C's promotion rules are not followed.
- The promotion rules specify how types can be converted to other types without losing data.

Function Definitions (Cont.)

- In our `sqrt` example above, an `int` is automatically converted to a `double` without changing its value.
- However, a `double` converted to an `int` truncates the fractional part of the `double` value.
- Converting large integer types to small integer types (e.g., `long` to `short`) may also result in changed values.
- The promotion rules automatically apply to expressions containing values of two or more data types (also referred to as mixed-type expressions).

Function Definitions (Cont.)

- The type of each value in a mixed-type expression is automatically promoted to the "highest" type in the expression (actually a temporary version of each value is created and used for the expression—the original values remain unchanged).
- Figure 5.5 lists 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 Definitions (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.
- Function argument values are converted to the parameter types in a function prototype as if they were being assigned directly to variables of those types.
- If our `square` function that uses an integer parameter (Fig. 5.3) is called with a floating-point argument, the argument is converted to `int` (a lower type), and `square` usually returns an incorrect value.
- For example, `square( 4.5 )` returns `16`, not `20.25`.

Function Definitions (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.

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.
- Figure 5.6 lists alphabetically some of the standard library headers that may be included in programs.
- You can create custom headers.
- Programmer-defined headers should also use the `.h` filename extension.

Headers (Cont.)

- A programmer-defined header can be included by using the `#include` preprocessor directive.
- For example, if the prototype for our square function was located in the header `square.h`, we'd include that header in our program by using the following directive at the top of the program:

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

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.)

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.)

Recursion

- The programs we've discussed are generally structured as functions that call one another in a disciplined, hierarchical manner.
- For some types of problems, it's useful to have functions call themselves.
- A recursive function is a function that calls itself either directly or indirectly through another function.

Recursion (Cont.)

- If the function is called with a base case, the function simply returns a result.
- If the function is called with a more complex problem, the function divides the problem into two conceptual pieces: a piece that the function knows how to do and a piece that it does not know how to do.
- To make recursion feasible, the latter piece must resemble the original problem, but be a slightly simpler or slightly smaller version.

Recursion (Cont.)

- Because this new problem looks like the original problem, the function launches (calls) a fresh copy of itself to go to work on the smaller problem—this is referred to as a recursive call and is also called the recursion step.
- The recursion step also includes the keyword `return`, because its result will be combined with the portion of the problem the function knew how to solve to form a result that will be passed back to the original caller, possibly `main`.
- The recursion step executes while the original call to the function is still open, i.e., it has not yet finished executing.

Recursion (Cont.)

- The recursion step can result in many more such recursive calls, as the function keeps dividing each problem it's called with into two conceptual pieces.
- In order for the recursion to terminate, each time the function calls itself with a slightly simpler version of the original problem, this sequence of smaller problems must eventually converge on the base case.
- At that point, the function recognizes the base case, returns a result to the previous copy of the function, and a sequence of returns ensues all the way up the line until the original call of the function eventually returns the final result to `main`.

Recursion (Cont.)

- 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.
- For example, 5! is the product 5 * 4 * 3 * 2 * 1, which is equal to 120.
- 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:

$$n! = n \cdot (n - 1)!$$

- For example, 5! is clearly equal to 5 * 4! as is shown by the following:

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$
$$5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$$
$$5! = 5 \cdot (4!)$$

- The evaluation of 5! would proceed as shown in Fig. 5.13.

Recursion (Cont.)

- Figure 5.13(a) shows how the succession of recursive calls proceeds until 1! is evaluated to be 1, which terminates the recursion.
- Figure 5.13(b) shows the values returned from each recursive call to its caller until the final value is calculated and returned.
- Figure 5.14 uses recursion to calculate and print the factorials of the integers 0–10 (the choice of the type `long` will be explained momentarily).
- The recursive `factorial` function first tests whether a terminating condition is true, i.e., whether `number` is less than or equal to 1.

Recursion (Cont.)

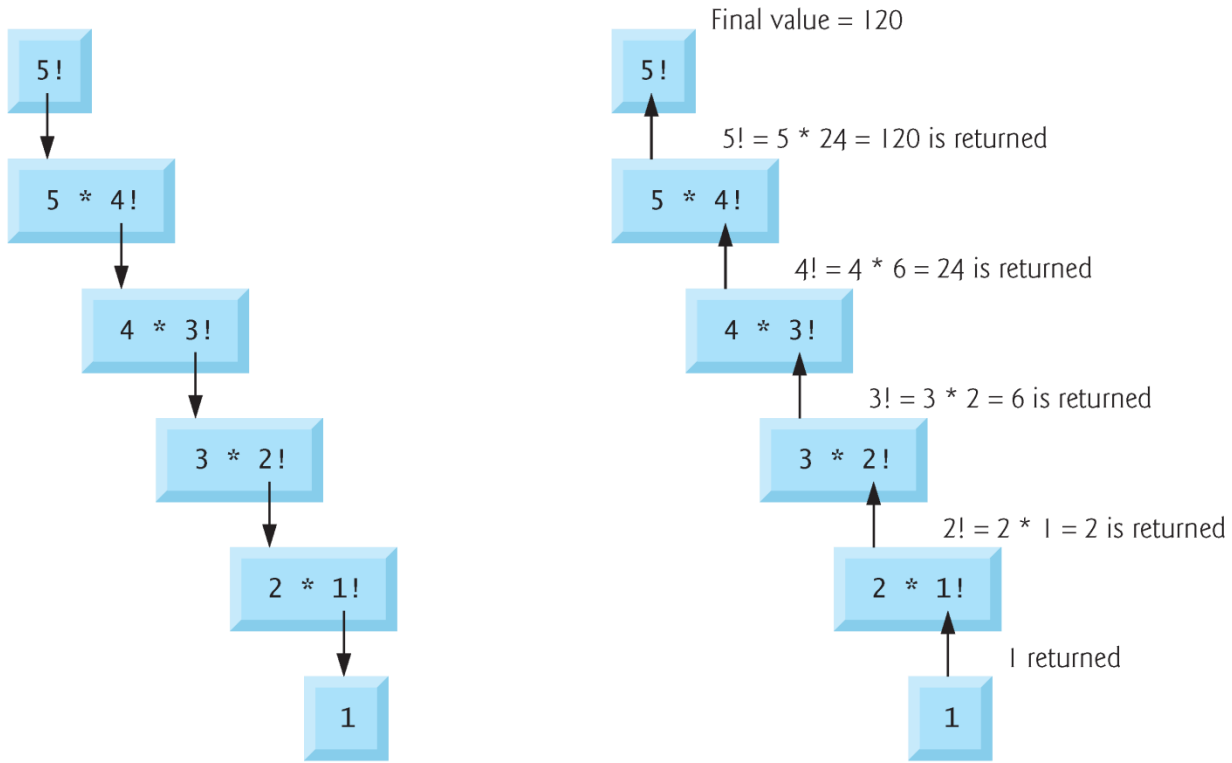
- If `number` is indeed less than or equal to 1, `factorial` returns 1, no further recursion is necessary, and the program terminates.
- If `number` is greater than 1, the statement

```
return number * factorial( number - 1 );
```

- expresses the problem as the product of `number` and a recursive call to `factorial` evaluating the factorial of `number - 1`.
- The call `factorial( number - 1 )` is a slightly simpler problem than the original calculation `factorial( number )`.

Recursion (Cont.)

- Function `factorial` (line 22) has been declared to receive a parameter of type `long` and return a result of type `long`.
- This is shorthand notation for `long int`.
- The C standard specifies that a variable of type `long int` is stored in at least 4 bytes, and thus may hold a value as large as +2147483647.
- As can be seen in Fig. 5.14, factorial values become large quickly.
- We've chosen the data type `long` so the program can calculate factorials greater than 7! on computers with small (such as 2-byte) integers.



(a) Sequence of recursive calls.

(b) Values returned from each recursive call.

Fig. 5.13 | Recursive evaluation of 5!.

```c
/* Fig. 5.14: fig05_14.c
   Recursive factorial function */
#include <stdio.h>

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 */

```

Fig. 5.14 | Calculating factorials with a recursive function. (Part 1 of 2.)

```
21  /* recursive definition of function factorial */
22  long factorial( long number )
23  {
24     /* base case */
25     if ( number <= 1 ) {
26        return 1;
27     } /* end if */
28     else { /* recursive step */
29        return ( number * factorial( number - 1 ) );
30     } /* end else */
31  } /* 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
```

Fig. 5.14 | Calculating factorials with a recursive function. (Part 2 of 2.)

Recursion (Cont.)

- The conversion specifier `%ld` is used to print `long` values.
- Unfortunately, the `factorial` function produces large values so quickly that even `long int` does not help us print many factorial values before the size of a `long int` variable is exceeded.
- As we'll explore in the exercises, `double` may ultimately be needed by the user desiring to calculate factorials of larger numbers.