Chapter 5 part 1
C Functions

C How to Program

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.

Program Modules in C (Cont.)

- 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.
- 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).

Program Modules in C (Cont.)

- 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.

- The worker may call other worker functions, and the boss will be unaware of this.

Program Modules in C (Cont.)

- Figure 5.1 shows a boss 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.

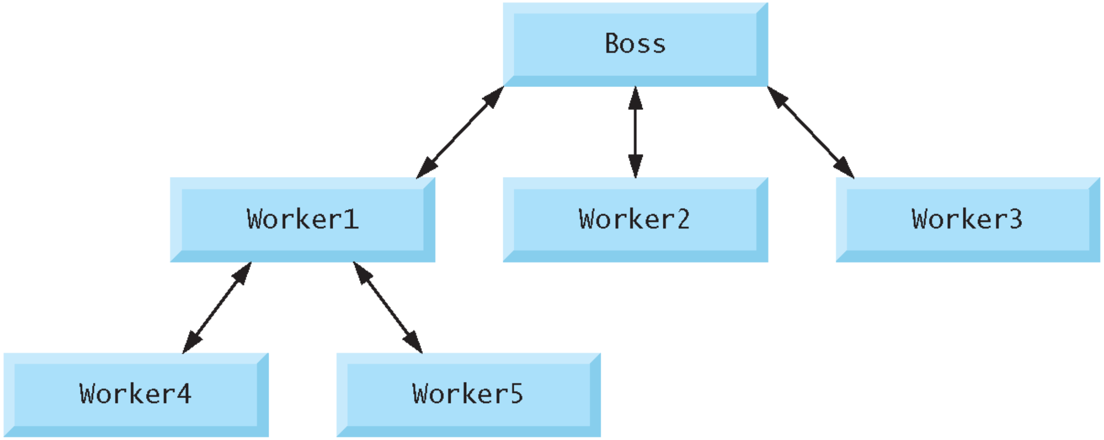

Fig. 5.1 | Hierarchical boss-function/worker-function relationship.

Math Library Functions

- Math library functions allow you to perform certain common mathematical calculations.

- Functions are 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.

- `double x, y = 900.0;`
- `x = sqrt(y);`

Math Library Functions

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

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

- The preceding statement would print `30.00`
- 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 `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`.

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`.

Function	Description	Example
`sqrt( x )`	square root of x	`sqrt( 900.0 )` is `30.0` `sqrt( 9.0 )` is `3.0`
`cbrt( x )`	cube root of x (C99 and C11 only)	`cbrt( 27.0 )` is `3.0` `cbrt( -8.0 )` is `-2.0`
`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 as a floating-point number	`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`

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

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

- All variables defined in function definitions are local variables—they can be accessed *only* in the function in which they're defined.
- 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

- 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

   puts( "" );
} // end main

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

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

Fig. 5.3 | Creating and using a programmer-defined function. (Part 2 of 2.)

Function square

- 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 21).

- Then `square` calculates `y * y`.
- The result is passed back returned 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` statement.

Function square (Cont.)

- The definition of function `square` shows that `square` expects an integer parameter `y`.
- The keyword `int` preceding the function name (line 21) indicates that `square` *returns* an integer result.
- The `return` statement in `square` passes the value of the expression `y * y` (that is, the result of the calculation) back to the calling function.

Function square (Cont.)

- Line 5

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

  is a function prototype.

- 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 any 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*.

Function Definitions (Cont.)

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

- The *function-name* is any valid identifier.
- The *return-value-type* is the data type of the result returned to the caller.
- The *parameter-list* is a comma-separated list that specifies the parameters received by the function when it's called.

Function Definitions (Cont.)

- The *return-value-type* `void` indicates that a function does not return a value.
- If a function does not receive any values, *parameter-list* is `void`.
- A type must be listed explicitly for each parameter.

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.

main's Return Type

- Notice that `main` has an `int` return type.
- The return value at the end of `main` is used to indicate whether the program executed correctly.

```
return 0;
```

—indicates to the Operating System that a program ran successfully.

You could explicitly return non-zero values from main to indicate that a problem occurred during your program's execution. These values are particular to each Operating System

Function Prototypes: A Deeper Look

- The compiler uses function prototypes to validate function calls.
- The function prototype for `square` is

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

- It states that `square` takes one argument of type `int` and returns a result of type `int`.

Function Prototypes: A Deeper Look (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.

Compilation Errors

- 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, if the function prototype had been written

```
void square( int y );
```

- 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 definition.

Arithmetic Conversion Rules

- 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` parameter, and the function will still work correctly.
- The statement

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

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

Arithmetic Conversion Rules (Cont.)

- The function prototype causes the compiler to convert a *copy* of the integer value `4` to the `double` value `4.0` before the *copy* 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.*

Arithmetic Conversion Rules (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, thus changing the original value.
- Converting large integer types to small integer types (e.g., `long` to `short`) may also result in changed values.

Arithmetic Conversion Rules (Cont.)

- Converting values to lower types normally results in an incorrect value.
- 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` returns an incorrect value.
- For example, `square( 4.5 )` returns `16`, not `20.25`.

Type the following program in Codeblocks

```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

   puts( "" );
} // end main

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

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

Self Review Exercise - Function maximum

- Create a new function `maximum` to determine and return the largest of three integers.
- The function must receive three integers as arguments.
- Write a program that reads three integers from the user
- Call the function `maximum` passing the three integers.
- Print the value of the largest integer returned to main by the `return` statement in `maximum`.

Self Review Exercise - Function maximum

- Use the following numbers to test your program.

- Enter integer one: 22
- Enter integer two: 85
- Enter integer three: 17

- Maximum is: 85

Function Call Stack

- 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.
- Think of a stack as analogous to a pile of dishes.
- When a dish is placed on the pile, it's normally placed at the top (referred to as pushing the dish onto the stack).
- Similarly, when a dish is removed from the pile, it's normally removed from the top (referred to as popping the dish off the 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.

Function Call Stack (Cont.)

- An important mechanism to understand is the function call stack (sometimes referred to as the program execution stack).
- This data structure—working "behind the scenes"—supports the function call/return mechanism.
- It also supports the creation, maintenance and destruction of each called function's automatic variables.

Function Call Stack and Stack Frames (Cont.)

- Of course, the amount of memory in a computer is finite, so only a certain amount of memory can be used to store stack frames on the function call stack.

- If more function calls occur than can have their stack frames stored on the function call stack, a *fatal* error known as a stack overflow occurs.

Function Call Stack in Action

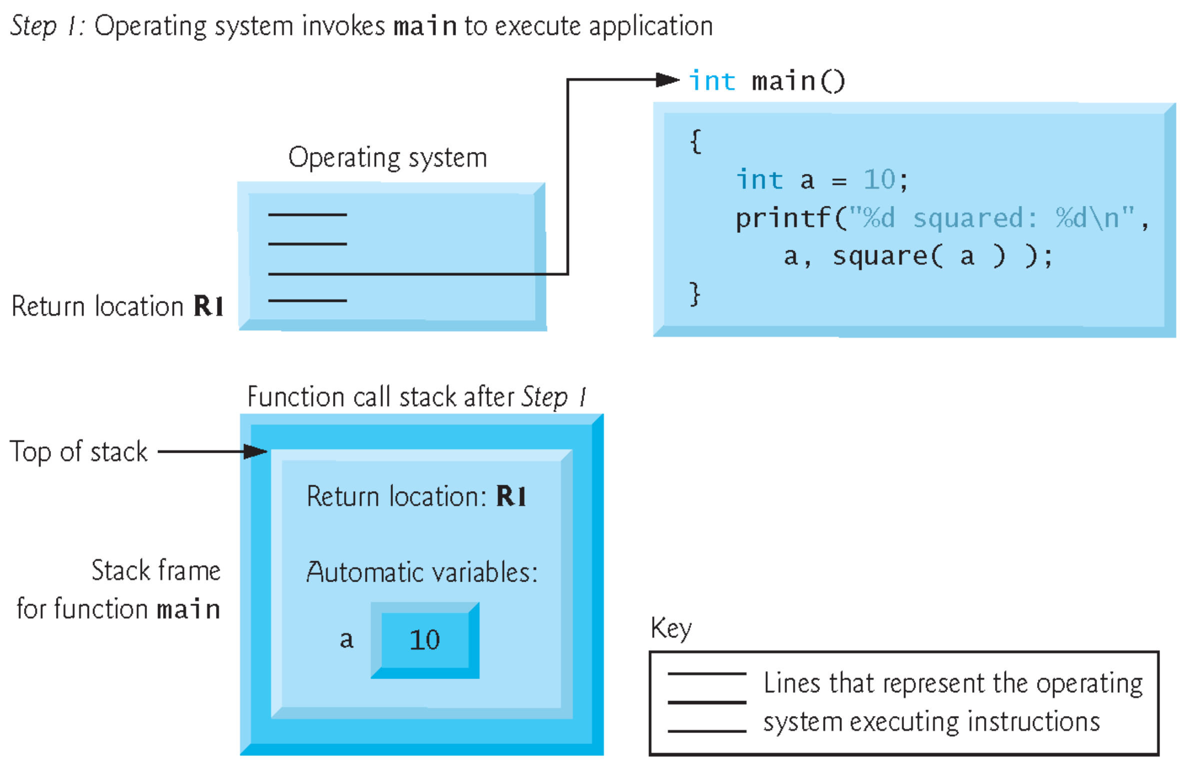

Fig. 5.7 | Function call stack after the operating system invokes main to execute the program.

Function Call Stack in Action

- Now let's consider how the call stack supports the operation of a square function called by `main`.
- First the operating system calls `main`—this pushes a stack frame onto the stack.
- The stack frame tells `main` how to return to the operating system (i.e., transfer to return address `R1`) and contains the space for `main`'s automatic variable (i.e., `a`, which is initialized to `10`).

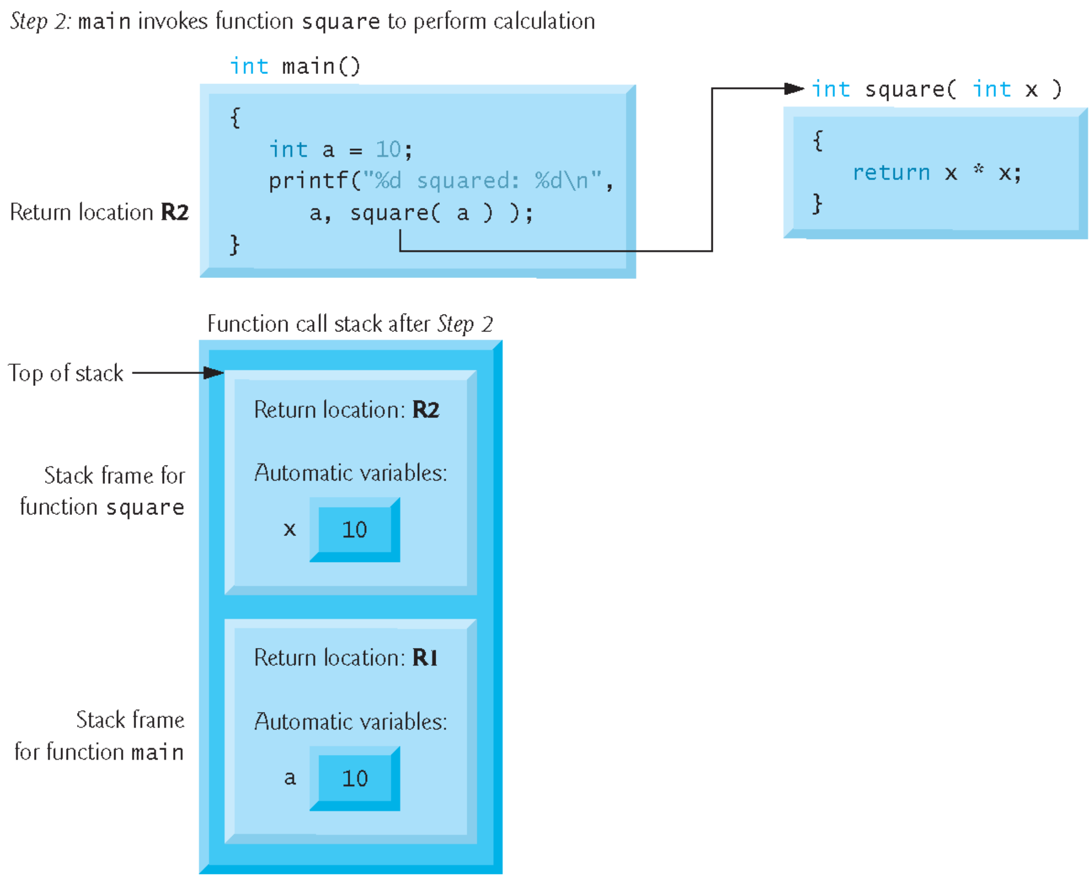

Fig. 5.8 | Function call stack after `main` invokes `square` to perform the calculation.

Function Call Stack in Action (Cont.)

- Function `main`—before returning to the operating system—now calls function `square`.
- This causes a stack frame for `square` to be pushed onto the function call stack (Fig. 5.8).
- This stack frame contains the return address that `square` needs to return to `main` (i.e., `R2`) and the memory for `square`'s automatic variable (i.e., `x`).

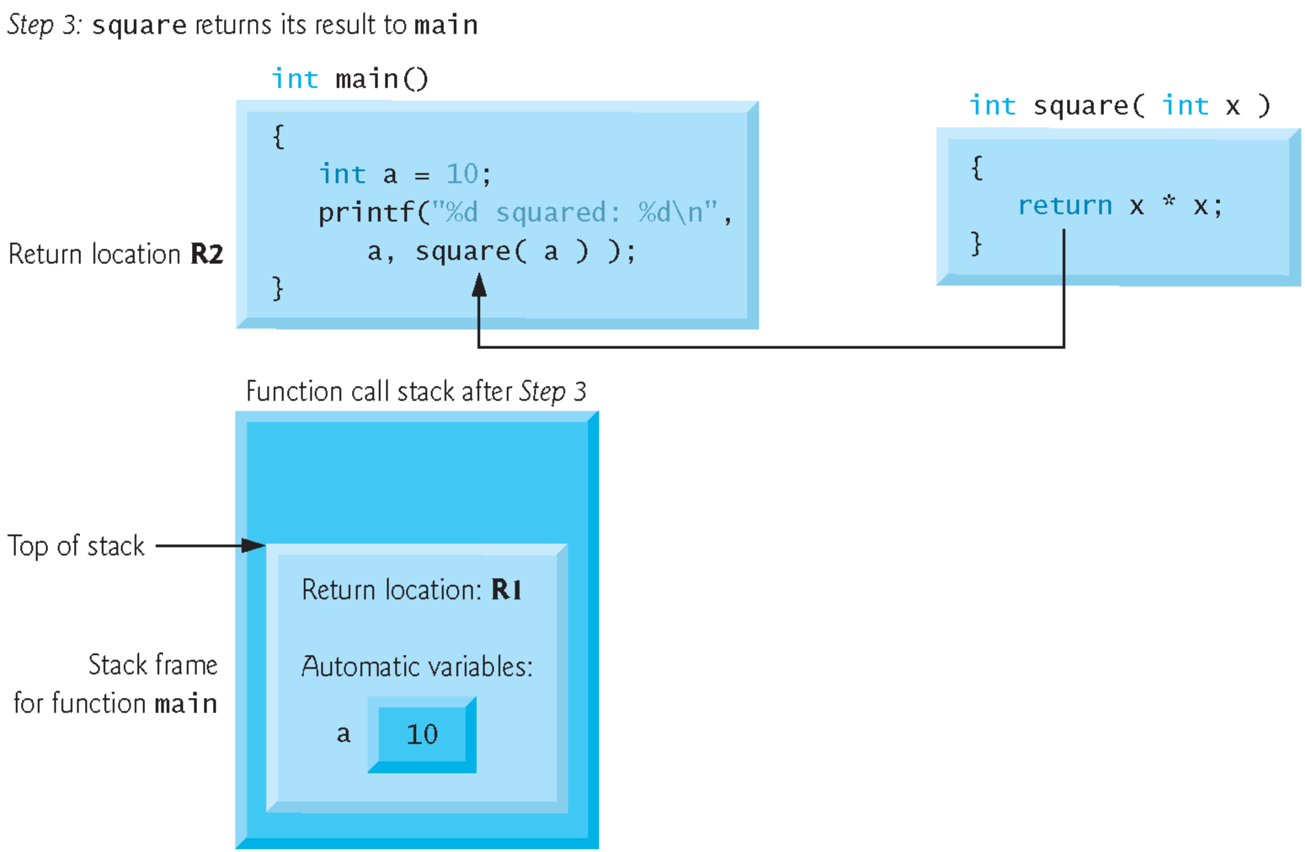

Fig. 5.9 | Function call stack after function `square` returns to `main`.

Function Call Stack in Action (Cont.)

- After `square` calculates the square of its argument, it needs to return to `main`—and no longer needs the memory for its automatic variable `x`.
- So the stack is popped—giving `square` the return location in `main` (i.e., `R2`) and losing `square`'s automatic variable.
- Figure 5.9 shows the function call stack after square's stack frame has been popped.

Function Call Stack in Action (Cont.)

- Function `main` now displays the result of calling `square`.
- Reaching the closing right brace of `main` causes its stack frame to be popped from the stack, gives `main` the address it needs to return to the operating system (i.e., `R1` in Fig. 5.7) and causes the memory for `main`'s automatic variable (i.e., `a`) to become unavailable.

Headers

- Each standard library has a corresponding header (.h file) containing the function prototypes for all the functions in that library and definitions of various data types and constants needed by those functions.
- Figure 5.10 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"
```

Passing Arguments By Value and By Reference

- There are two ways to pass arguments—pass-by-value and pass-by-reference.
- When arguments are *passed 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.
- When an argument is passed by reference, the caller allows the called function to modify the original variable's value.

Passing Arguments By Value and By Reference (Cont.)

- Pass-by-reference should be used only with trusted called functions that need to modify the original variable.
- So far all arguments we have used are passed by value.
- In Chapter 6, we'll see that array arguments are automatically passed by reference for performance reasons.

Self Review Exercise - Function seconds

- Create a new function `seconds` that returns the number of seconds since clock "struck 12" given input time as arguments hours h, minutes m, seconds s.
- Write a program that reads 2 times of the same day as hours, minutes, seconds.
- Call the function `seconds` passing the three values twice (once for each time entered by the user).
- Print the difference between the two times in seconds.