Cpt S 122 – Data Structures

Functions Review

Nirmalya Roy

School of Electrical Engineering and Computer Science
Washington State University

Topics

- Introduction
- Program Modules
- Functions
  - Definitions
  - Math Library Functions
  - Function Prototypes
    - Compilation Errors, Arguments coercion
  - Data Structure in Function Call
    - Stack and Stack Frame
  - Standard Library Headers
  - Arguments Passing By Value and By Reference
  - Storage Classes

Introduction

- Most computer programs that solve real-world problems are quite large.
- 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.
- Key features of the C language that facilitate the design, implementation, operation and maintenance of large programs.

Program Modules

- 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
  - perform common *mathematical calculations*, *string manipulations*, *character manipulations*, *input/output*, and many other useful operations.

Software Engineering Observations

- Familiarize yourself with the rich collections of functions in the C standard library.
- Avoid reinventing the wheel
  - When possible use C standard library functions instead of writing new functions.
- Using the functions in the C standard library helps make program more portable.

Program Modules (Cont.)

- The functions `printf`, `scanf` and `pow` 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
  - provides information (as arguments) that the called function needs to perform its designated task.

Program Modules (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
  - `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 (Cont.)

- The worker may call other worker functions, and the boss will be unaware of this.
- We’ll soon see how this “hiding” of implementation details promotes good software engineering.
- 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.

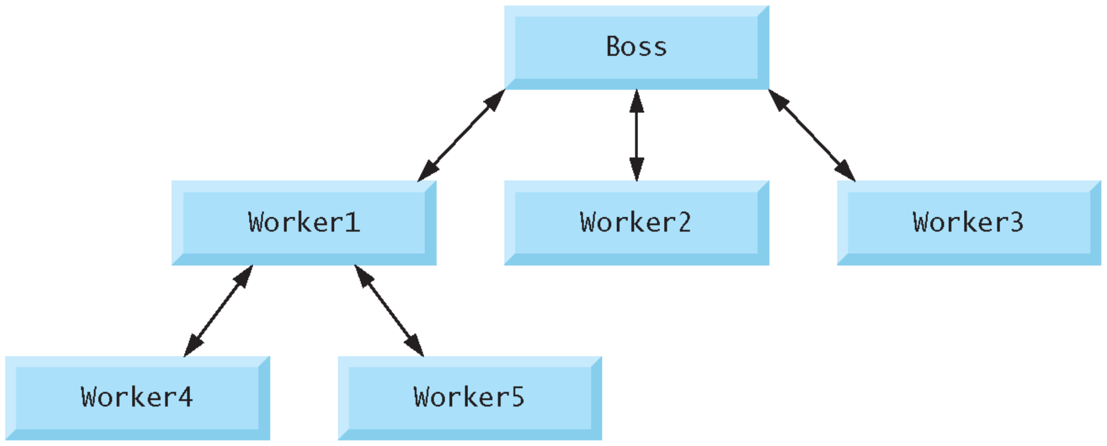


Fig. 5.1 | Hierarchical boss-function/worker-function relationship.

Functions

- Functions allow you to modularize a program.
- All variables defined in function definitions are local variables
  - They can be accessed *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.
- Software reusability is a major factor in the *object-oriented programming*
  - C++, Java and C# (pronounced “C sharp”).
- We use abstraction each time we use standard library functions like `printf`, `scanf` and `pow`.
- A third motivation is to avoid repeating code in a program.
- Packaging code as a function allows the code to be executed from other locations in a program simply by calling the function.

Software Engineering Observations

- In programs containing many functions, *main* is often implemented as a group of calls to functions that perform the bulk of the program’s work.
- Each function should be limited to perform a single, well-defined task, and function name should express that task
  - Facilitates abstraction and promotes software reusability.
- Break a multi-tasking function into several smaller functions
  - This is known as decomposition.

Function Definitions

- A program may consist 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

```

Fig. 5.3 | Creating and using a programmer-defined function. (Part 1 of 2.)

```cpp
// 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 Example (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 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 (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.
- 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 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*.
- 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.

Software Engineering Observations

- The function prototype, function header and function calls should all agree in the number, type, and order of arguments and parameters, and in the type of return value.

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.)

`main`'s Return Type

- Notice that `main` has an `int` return type.
- The return value of `main` is used to indicate whether the program executed correctly.
- In earlier versions of C, it's been explicitly placed

```
    return 0;
```

- at the end of `main`—`0` indicates that a program ran successfully.
- The C standard indicates that main implicitly returns 0.

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` you 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.)

- #include <math.h> when using functions from Math Library
- 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`.
- 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`

`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.)

Function Prototypes: A Deeper Look

- An important feature of C is the function prototype.
- This feature was borrowed from C++.
- The compiler uses function prototypes to validate function calls.

Function Example

Function `maximum`

- Our second example uses a programmer-defined function `maximum` to determine and return the largest of three integers (Fig. 5.4).
- Next, they’re passed to `maximum` (line 19), which determines the largest integer.
- This value is returned to main by the `return` statement in `maximum` (line 36).

```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

// function main begins program execution
int main( void )
{
   int number1; // first integer entered by the user
   int number2; // second integer entered by the user
   int number3; // third integer entered by the user

   printf( "%s", "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 ) );
} // end main

```

Fig. 5.4 | Finding the maximum of three integers. (Part 1 of 3.)

```
// Function maximum definition
// x, y and z are parameters
int maximum( int x, int y, int z )
{
   int max = x; // assume x is largest

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

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

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

Fig. 5.4 | Finding the maximum of three integers. (Part 2 of 3.)

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

```
Enter three integers: 47 32 14
Maximum is: 47
```

```
Enter three integers: 35 8 79
Maximum is: 79
```

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

Function Prototypes: A Deeper Look (Cont.)

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

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

- It 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 `maximum`'s function definition.

Function Prototypes: A Deeper Look (Cont.)

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, 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 Prototypes: A Deeper Look (Cont.)

Argument Coercion and "Usual 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`.

Function Prototypes: A Deeper Look (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*.
- These conversions can lead to incorrect results if C's usual arithmetic conversion rules are not followed.
- These rules specify how values can be converted to other types without losing data.

Function Prototypes: A Deeper Look (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.

Function Prototypes: A Deeper Look (Cont.)

- The usual arithmetic conversion rules automatically apply to expressions containing values of two or more data types (also referred to as mixed-type expressions) and are handled for you by the compiler.
- In a mixed-type expression, the compiler makes a temporary copy of the value that needs to be converted then converts the copy to the “highest” type in the expression—the original value remains unchanged.

Function Prototypes: A Deeper Look (Cont.)

- The usual arithmetic conversion rules for a mixed-type expression containing at least one floating-point value are:
  - If one of the values is a long double, the other is converted to a long double.
  - If one of the values is a double, the other is converted to a double.
  - If one of the values is a float, the other is converted to a float.

Data type	`printf` **conversion specification**	`scanf` **conversion specification**
Floating-point types		
`long double`	`%Lf`	`%Lf`
`double`	`%f`	`%lf`
`float`	`%f`	`%f`

Fig. 5.5 | Arithmetic data types and their conversion specifications. (Part 1 of 2.)

Data type	`printf` **conversion specification**	`scanf` **conversion specification**
Integer types		
`unsigned long long int`	`%llu`	`%llu`
`long long int`	`%lld`	`%lld`
`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 | Arithmetic data types and their conversion specifications. (Part 2 of 2.)

Function Call Stack and Stack Frames

- 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 and Stack Frames (Cont.)

- An important mechanism for computer science students 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.)

- As each function is called, it may call other functions, which may call other functions—all before any function returns.
- Each function eventually must return control to the function that called it.
- So, we must keep track of the return addresses that each function needs to return control to the function that called it.
- The function call stack is the perfect data structure for handling this information.

Function Call Stack and Stack Frames (Cont.)

- Each time a function calls another function, an entry is *pushed* onto the stack.
- This entry, called a stack frame, contains the *return address* that the called function needs in order to return to the calling function.
- If the called function returns, instead of calling another function before returning, the stack frame for the function call is popped, and control transfers to the return address in the *popped* stack frame.

Function Call Stack and Stack Frames (Cont.)

- The stack frames have another important responsibility.
- Most functions have *automatic variables*—parameters and some or all of their local variables.
- Automatic variables need to exist while a function is executing.
- But when a called function returns to its caller, the called function’s automatic variables need to “go away.”
- The called function’s stack frame is a perfect place to reserve the memory for *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.

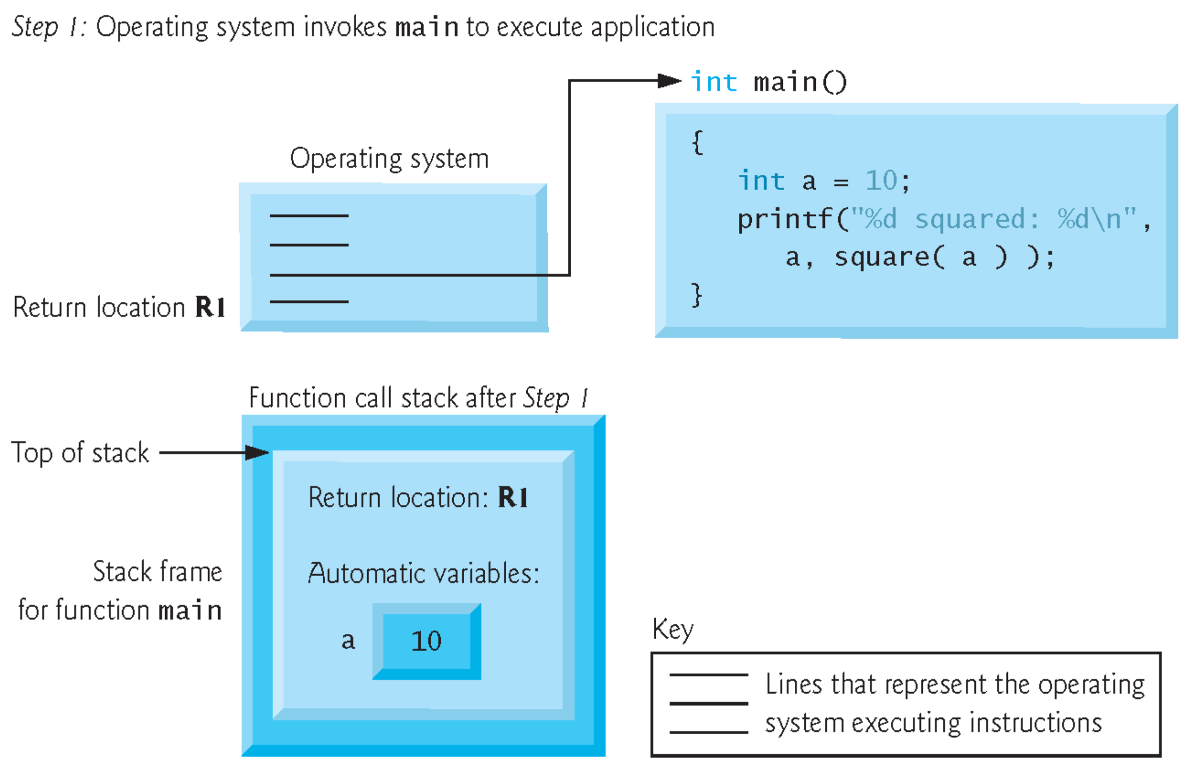
Function Call Stack and Stack Frames (Cont.)

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`).

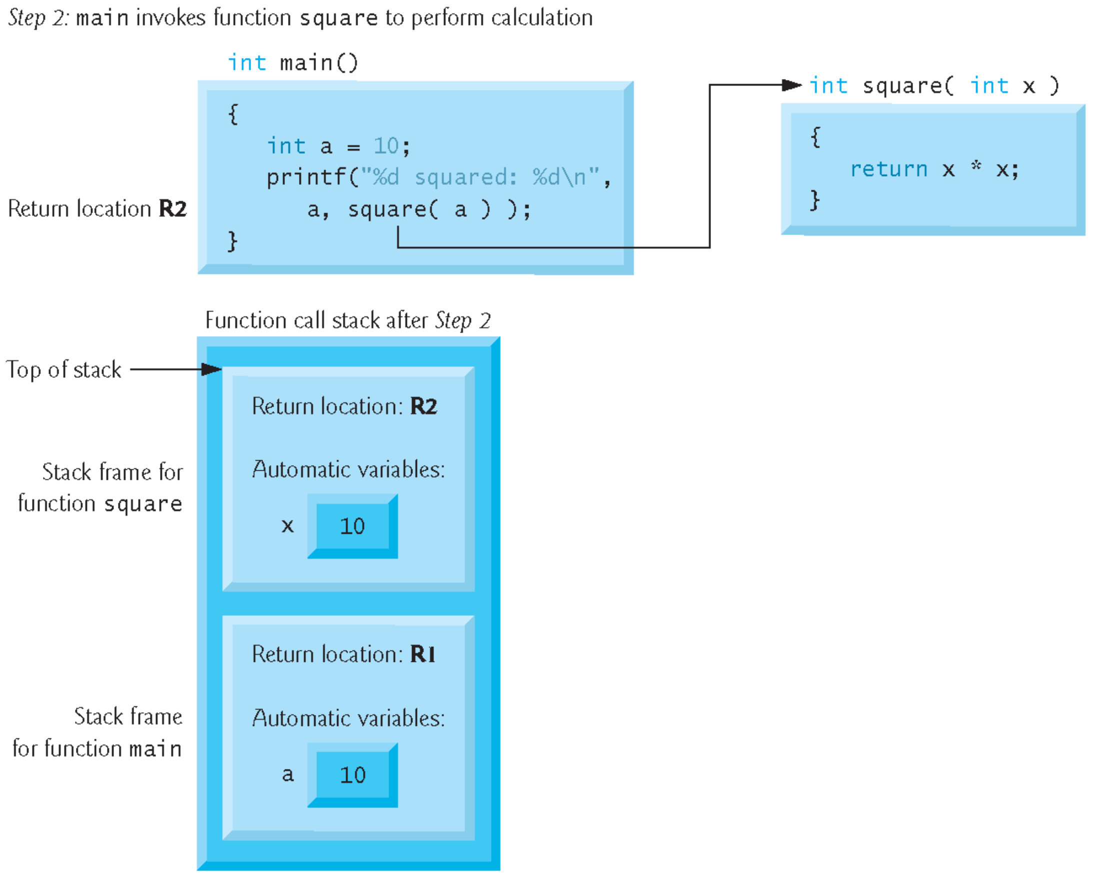
Function Call Stack and Stack Frames (Cont.)

- 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`).


Step 1: Operating system invokes main to execute application
Operating system
Return location R1
Function call stack after Step 1
Top of stack
Stack frame for function main
Return location: R1
Automatic variables:
a
10
Key
Lines that represent the operating system executing instructions

```
int main()
{
   int a = 10;
   printf("%d squared: %d\n",
      a, square( a ) );
}
```

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



Step 2: main invokes function square to perform calculation
Return location **R2**
Function call stack after *Step 2*
Top of stack
Stack frame for function square
Return location: **R2**
Automatic variables:
x
10
Stack frame for function main
Return location: **R1**
Automatic variables:
a
10

```
int main()
{
   int a = 10;
   printf("%d squared: %d\n",
      a, square( a ) );
}
```

```
int square( int x )
{
   return x * x;
}
```

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

Function Call Stack and Stack Frames (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.

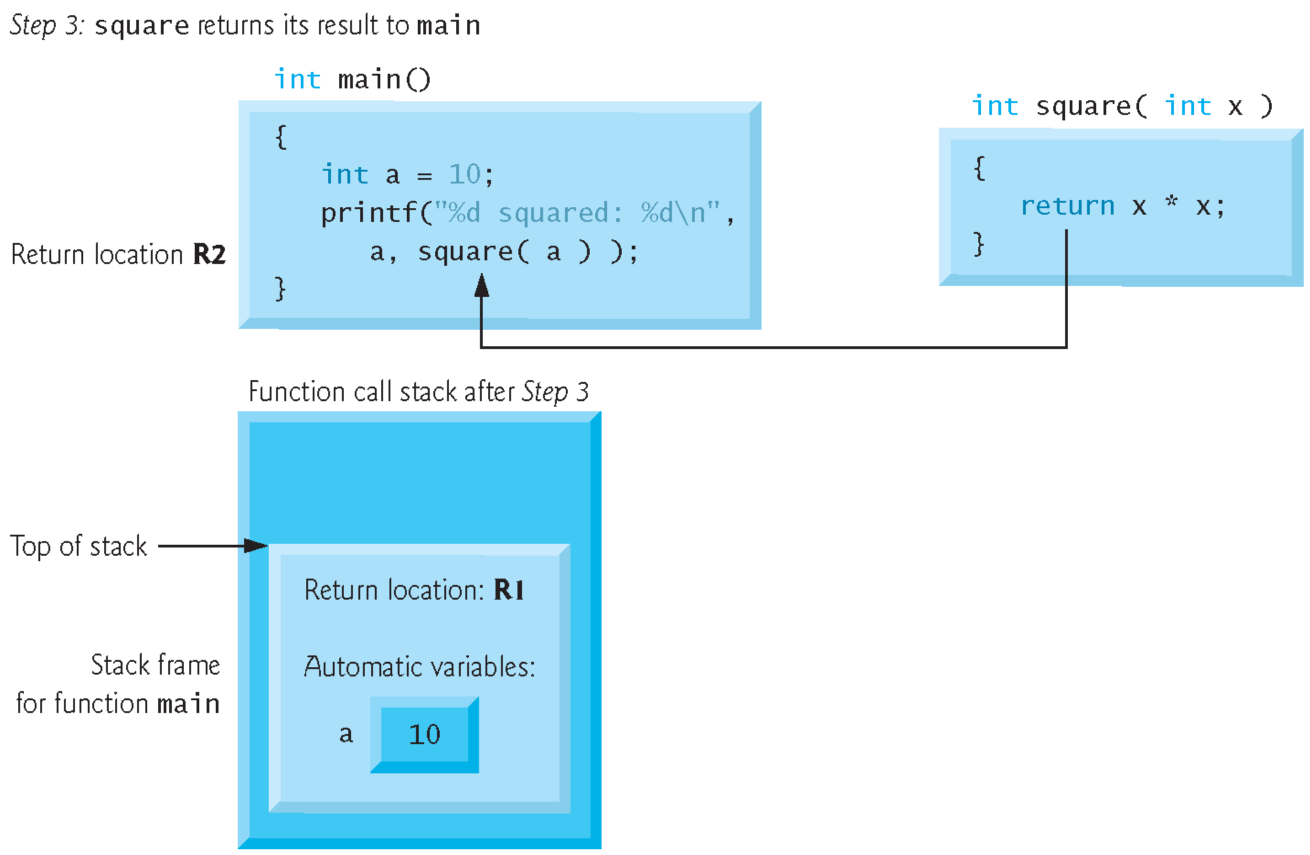


Fig. 5.9 | Function call stack after function square returns to main.

Function Call Stack and Stack Frames (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
  - contains the function prototypes for all the functions in that library.
  - definitions of various data types and constants needed by those functions.
- You can create custom headers.
  - A programmer-defined header can be included by using the `#include` preprocessor directive.
- For example, assume the prototype for our square function was located in the header `square.h`
  - we have to 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 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 such as dates, times, currency 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. 5.10 | 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. 5.10 | Some of the standard library headers. (Part 2 of 2.)

Passing Arguments By Value and By Reference

- In many programming languages, 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.
- Pass-by-value should be used whenever the called function does not need to modify the value of the caller's original variable.

Passing Arguments By Value and By Reference(Cont.)

- This prevents the accidental side effects (variable modifications) that so greatly hinder the development of correct and reliable software systems.
- Pass-by-reference should be used only with trusted called functions that need to modify the original variable.
- In C, all arguments are passed by value.
  - Simulate pass-by-reference by using the *address operator* and the *indirection operator*

Storage Classes

- We use identifiers for variable names.
- The attributes of variables include name, type, size and value.
- We also use identifiers as names for user-defined functions.
- Actually, each identifier in a program has other attributes, including storage class, storage duration, scope and linkage.
- C provides the storage class specifiers: `auto`, `register`, `extern` and static.
- An identifier's storage class determines its storage duration, scope and linkage.
- An identifier's storage duration is the period during which the identifier exists *in memory*.

Storage Classes (Cont.)

- Some exist briefly, some are repeatedly created and destroyed, and others exist for the program's entire execution.
- An identifier's scope is where the identifier can be referenced in a program.
- Some can be referenced throughout a program, others from only portions of a program.
- An identifier's linkage determines for a multiple-source-file program whether the identifier is known only in the current source file or in any source file with proper declarations.

Storage Classes (Cont.)

Local Variables

- Only variables can have automatic storage duration.
- A function’s local variables (those declared in the parameter list or function body) normally have automatic storage duration.
- Keyword `auto` explicitly declares variables of automatic storage duration.

Storage Classes (Cont.)

- Global variables are created by placing variable declarations *outside* any function definition, and they retain their values throughout the execution of the program.
- Global variables and functions can be referenced by any function that follows their declarations or definitions in the file.
- This is one reason for using function prototypes—when we include `stdio.h` in a program that calls `printf`, the function prototype is placed at the start of our file to make the name `printf` known to the rest of the file.

Storage Classes (Cont.)

- Local variables declared with the keyword `static` are still known only in the function in which they're defined, but unlike automatic variables, `static` local variables retain their value when the function is exited.
- The next time the function is called, the `static` local variable contains the value it had when the function last exited.
- The following statement declares local variable `count` to be `static` and initializes it to 1.
  - `static int count = 1;`

Conclusion

- Function Definitions and Prototypes
- Function Call Stack and Stack Frames
- Passing Arguments By Value and By Reference
- Storage Classes
- Live Code Examples