UEE1302 Introduction to Computers and Programming

C_Lecture 04:

Functions

C: How to Program 7th ed.

Agenda

- Introduction to Functions
- Predefined Functions
- Programmer-Defined Functions
  - Function Declaration (Prototype)
  - Function Definition
  - Function Call
- Function Call Stack

Program Modules in C

- Modulus 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.)

- Functions allow you to modularize a program by separating its tasks into self-contained units
- Functions you write are referred to as user-defined functions or programmer-defined functions
- The statements in function bodies are written only once, are reused from perhaps several locations in a program and are hidden from other functions

Program Modules in C (cont.)

- Motivations for "functionalizing" a program
  1. Divide-and-conquer makes program development more manageable
  2. Software reusability—using existing functions as building blocks to create new programs
     - With good function naming and definition, programs can be created from standardized functions that accomplish specific tasks, rather than being built by using customized code
  3. Avoid repeating code in a program
     - Packaging code as a function allows the code to be executed from different locations in a program simply by calling the function

Introduction to Functions

- Building blocks of C programs
  - named differently in other languages. Ex: procedures, subprograms, methods
  - in C, they are termed functions
- I-P-O of C functions
  - Input–Process–Output
  - use functions as basic subparts to any program
- Starting from predefined functions
  - What !? We've seen them before…

Predefined Functions

- Libraries are full of functions for our use!
- Two types of functions:
  - that return a value (**`int`**, **`float`**, **`char`**,…)
  - that do not return a value (**`void`**)
- In algebra, a function is defined as a *rule* or *correspondence* between values, called the function's **arguments**, and the unique value of the function associated with the arguments
  - Ex: f(x) = 2x+5, f(1) = 7, f(2) = 9, and f(3) = 11
  - 1, 2, and 3 are arguments
  - 7, 9, and 11 are the corresponding values

Predefined Functions (cont.)

- Predefined functions are organized into separate libraries => require **`#include`** `<XXX>`
- Examples:
  - `<stdio.h>`: for `printf`, `scanf` and other I/O
  - `<math.h>`: some Math functions in C
  - `<stdlib.h>`: standard general utilities in C
- Some examples of the predefined Math functions in `<math.h>` are:
  - `sqrt(x)`
  - `pow(x, y)`
  - `floor(x)`

* `<math.h>` ref: http://www.cplusplus.com/reference/cmath/

Power Function `pow(x, y)`

- Power function `pow(x, y)`
  - compute x^y (x to the power of y)
- Ex:

```c
double x = 3.0, y = 2.0;
double result = pow(x, y);
printf("%.2f", result);
```

=> 9.00 is displayed since $3.0^{2.0} = 9.00$

- `pow(x, y)` returns a value of type `double`
  - `x` and `y` are called the arguments of the function `pow(x, y)`
- Function `pow(x, y)` has *two* arguments

Square Root Function `sqrt(x)`

- Square root function `sqrt(x)`
  - compute the square root of x
- for x ≧ 0.0
  - return data of type `double`
  - require only one argument
- Ex:

```
double x = 2.25;
double result = sqrt(x);
printf("%.2f", result);
```

=> 1.50 is displayed since $\sqrt{2.25} = 1.50$

- What happen if giving it a negative value ?

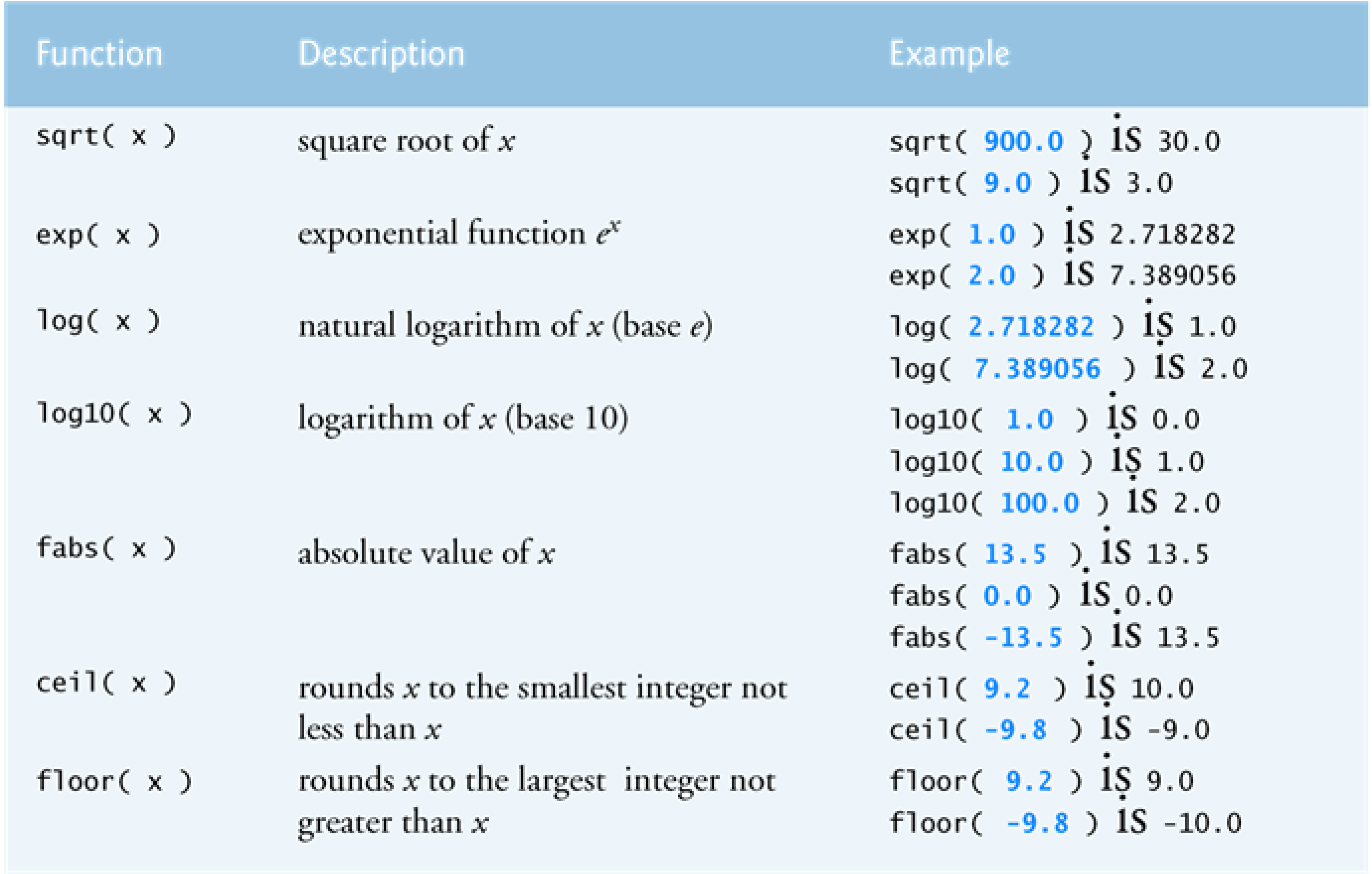
Round Functions: `ceil(x)` & `floor(x)`

- Round functions
  - `ceil(x)` returns the smallest integral value not less than x => ceil(x) ≥ x
  - `floor(x)` returns the largest integral value not greater than x => floor(x)≤ x
  - require only *one* argument
- Ex:

```
double x = 49.50;
double x_up = ceil(x);
double x_dn = floor(x);
printf("%.1f vs. %.1f", x_up, x_dn); ;
```

=> display on screen 50.0 vs. 49.0

Common Used Math Library Functions



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`

Example of Programmer-Defined Function

```c
// Fig. 5.3: fig05_03.c
// Creating and using a programmer-defined function.
#include <stdio.h>

int square ( int y ); // function prototype

int main ( void )
{
    int x; // counter
    // loop 10 times and calculate the square of x
    for ( x = 1; x <= 10; x++) {
        printf( "%d ", square(x) ); // function call
    }
    printf( "\n" );

    return 0;
}
```

Example of Programmer-Defined Function (cont.)

```
// square function definition returns square of an integer
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
```

- screen output:

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

Programmer-Defined Functions

- Programmer-defined functions
  - a.k.a. user-defined functions
- Building blocks of programs
  - divide & conquer
  - readability
  - re-use
- Your definition(s) can go in either:
  - the same file as `main()` resides, or
  - separate file(s) so others can use it (them) too

Using Programmer-Defined Functions

- Three pieces to using functions:

1. function declaration (a.k.a. prototype)
    - information for compiler
    - to properly interpret calls
2. function definition
    - actual implementation of C code for what function does
3. function call
    - transfer control to the function

1. Function Declaration

- Function declaration (prototype)
  - an informational declaration for compiler
- Guide compiler how to interpret calls
  - Syntax:

```
<return_type> FnName(<parameter-list>);
```

  - Example:

```
double totalCost(int number, double price);
```

- Placed before any calls
  - in *declaration space* of `main()`
  - or above `main()` in *global space*

Alternative Function Declaration

- Recall: function declaration only provides information or compiler
- Compiler only needs to know:
  - return type
  - function name
  - parameter list
- Formal parameter names not needed:

```
double totalCost(int, double);
```

2. Function Definition

- Implementation of function
  - just like implementing function `main()`
- Ex:

```
double totalCost(int number, double price) {
    const double TAXRATE = 0.05;
    double subtotal;
    subtotal = price * number;
    return (subtotal * (1 + TAXRATE));
}
```

  - Notice proper indentation

Function Definition Placement

- Placed after function `main()` if its prototype is placed at front
  - **NOT** "**inside**" function `main()`!
- Functions are "equals"
  - no function is ever "part" of another
- Formal parameters in definition
  - "Placeholders" for data sent in
  - "Variable name" used to refer to data in definition
- **`return`** statement
  - send data back to caller

3. Function Call

- Just like calling predefined function
  - Ex: `dBill = totalCost(num, cost);`
- Recall: `totalCost` returns **`double`** value
  - assigned to variable named "`dBill`"
- Two arguments here: `num` and `cost`
  - arguments can be literals, variables, expressions, or combination
  - in function call, arguments often as known as "actual arguments "

=>∵ they contain the "actual data" being sent

Parameter vs. Argument

- Terms often used interchangeably
- Formal parameters/arguments
    - in function declaration
    - in function definition's header
- Actual parameters/arguments
    - in function call
- Technically, parameter is formal piece while argument is actual piece.

Example of Functions with Multiple Parameters

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

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

int main ( void )
{
    int number1, number2, number3;

    printf( "Enter three integers: ");
    scanf( "%d%d%d", &number1, &number2, &number3);
    printf( "Maximum is %d\n",
           maximum(number1, number2, number3) );

    return 0;
}
```

Example of Functions with Multiple Parameters (cont.)

```
// 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
        max = y;  // assign y to max
    if ( z > max ) // if z is larger
        max = z;  // assign z to max

   return max;
} // end function maximum
```

Example of Functions with Multiple Parameters (cont.)

- screen output

```
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
```

Functions Calling Functions

- We're already doing this!
  - `main()` is a function!
- Only requirement:
  - function's declaration must appear first
- Function's definition typically elsewhere
  - after `main()`'s definition
  - or in the separate file
- Common for functions to call many other functions
- Function can even call itself => Recursion

Declaring Void Functions

- Similar to functions returning a value
  - return type specified as "**`void`**"
- Example:
  - Function declaration/prototype:

```
void showResults(double fDegrees,
                 double cDegrees);
```

  - return-type is "**`void`**"
  - nothing is really returned

Declaring Void Functions (cont.)

- Function definition:

```
void showResults(double fDegrees,
                 double cDegrees) {
    printf( "%.2f degrees Fahrenheit equals
             %.2f degrees Celsius\n",
             fDegrees, cDegrees );
}
```

- Notice: no **`return`** statement
  - optional for `void` functions

Calling Void Functions

- Same as calling predefined functions
- From some other function, like `main()`
  - `showResults(degreesF, degreesC);`
  - `showResults(32.5, 0.3);`
- Notice no assignment, since no value returned
- Actual arguments (`degreesF`, `degreesC`)
  - Passed to function
  - Function is called to "do its job" with the data passed in

More on `Return` Statements

- Transfers control back to "calling" function
  - For return type other than `void`, MUST have return statement
  - Typically the LAST statement in function definition
- **`return`** statement is optional for **`void`** functions
  - Closing } would implicitly return control from **`void`** function

Special Function `main()`

- Recall: `main()` is a function
- "Special" in that:
  - One and only one function called `main()` will exist in a program
- Who calls `main()` ?
  - Operating system (OS)
  - Tradition holds it should have **`return`** statement

    => return value to "caller" which is the operating system
  - should return "**`int`**" or "**`void`**"

Function Call Stack

- As each function is called, it may, in turn, call other functions, which may, in turn, call other functions—all before any of the functions return.
- Each function eventually must return control to the function that called it.
- So, somehow, we must keep track of the return addresses that each function needs to return control to the function that called it.

Function Call Stack (cont.)

- The function call stack is the perfect data structure for handling this information
- Each time a function calls another function, an entry is pushed onto the stack
- This entry, called a stack frame or an activation record, contains the return address that the called function needs in order to return to the calling function
- It also contains some additional information we'll soon discuss

Demonstrating the Function Call Stack

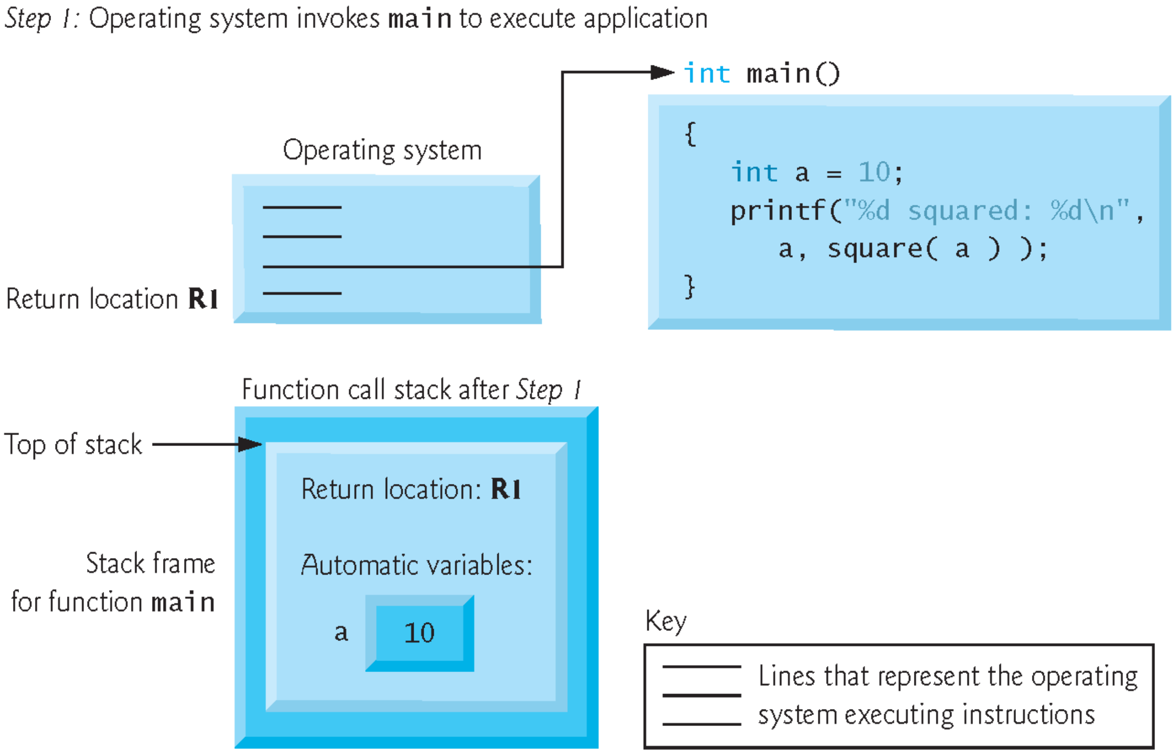
```
// Fig. 5.6: fig05_06.c
// square function.
#include <stdio.h>

int square ( int ); // function prototype

int main ()
{
    int a = 10;
    printf( "%d squared: %d", a, square(a) );
    return 0;
}
int square( int x ) // x is a local variable
{
    return x * x; // calculate square and return result
}
```

Demonstrating the Function Call Stack (cont.)

- Function call stack after the OS invokes `main` to execute the program

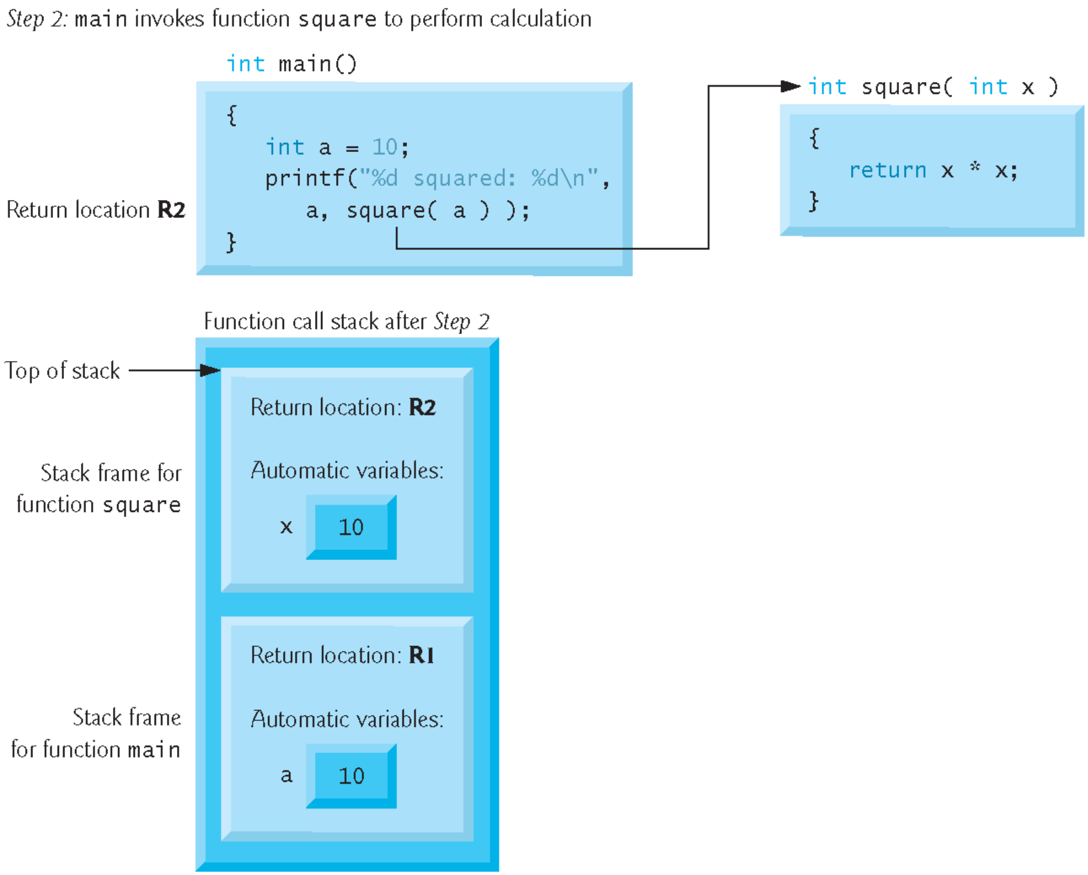


```
int main()
{
   int a = 10;
   printf("%d squared: %d\n",
      a, square( a ) );
}
```

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

Demonstrating the Function Call Stack (cont.)

- Function call stack after `main` invokes `square` to perform the calculation



Step 2: `main` invokes function `square` to perform calculation

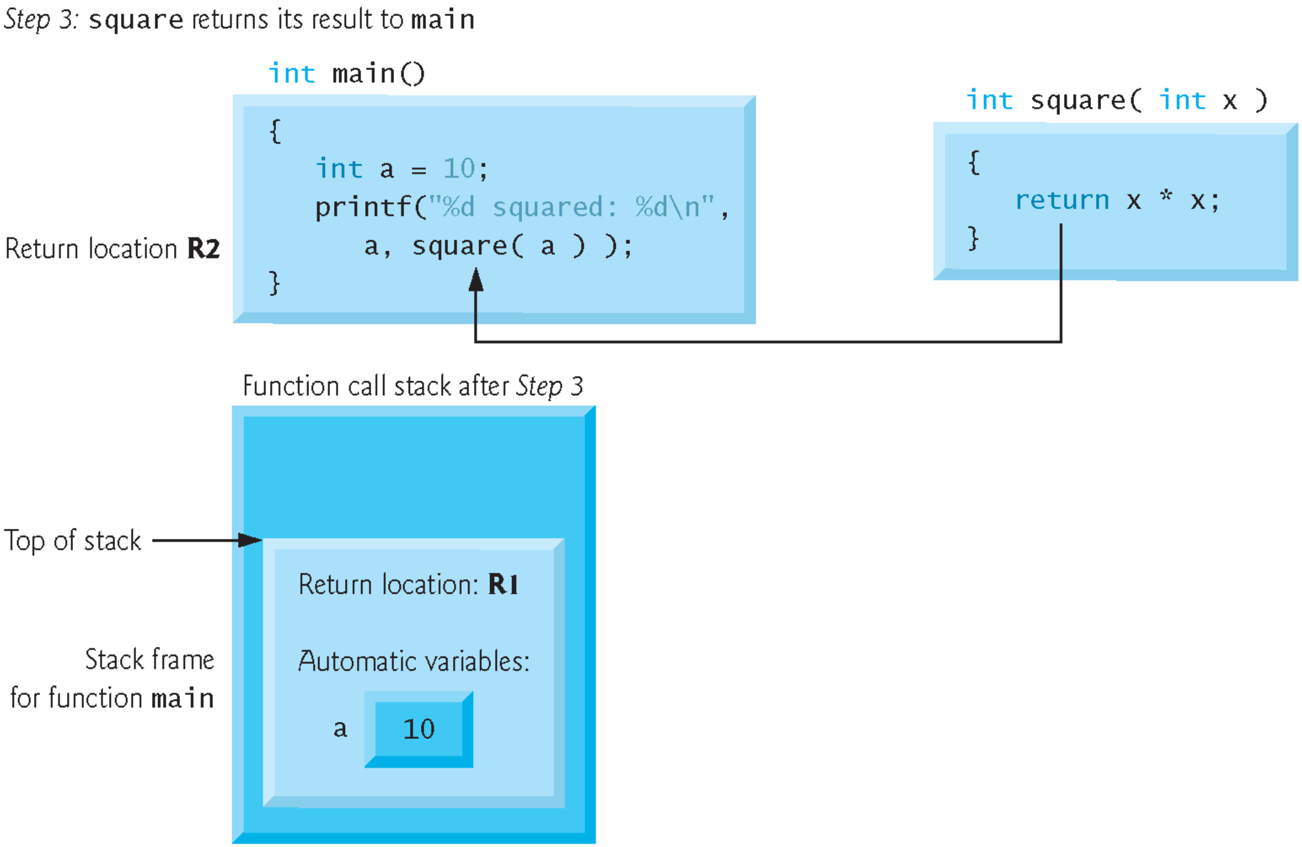
```
int main()
{
   int a = 10;
   printf("%d squared: %d\n",
      a, square( a ) );
}
```

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

Function call stack after *Step 2*

Demonstrating the Function Call Stack (cont.)

- Function call stack after function `square` returns to `main`



Step 3: `square` returns its result to `main`

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

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

Function call stack after *Step 3*

Summary

- Functions should be “black boxes”
  - Hide “how” details
  - Declare own local data
- Programmer-Defined Functions
  - Function Declaration (Prototype)
  - Function Definition
  - Function Call

Exercise (1)

- Give the function prototype for each of the following functions.
  a) Function `unique` that takes two integer arguments, `a` and `n`, and returns an integer result.
  b) Function `compare` that takes two floating point arguments, `n` and `m`, and does not return a value.
  c) Function `findMax` that takes three integer arguments, `a`, `b`, and `c`, and returns an integer result.

Exercise (2)

- Write a function `check(x, y, n)` that returns 1 if both `x` and `y` fall between 0 and `n - 1`, inclusive. The function should return `0` otherwise. Assume that `x`, `y`, and `n` are all of type `int`.

Exercise (3)

- Write a function `digit(n, k)` that returns the `k`th digit (from the right) in `n` (a positive integer). For example digit(829, 1) return 9 and digit(829, 2) return 2. If `k` is greater than the number of digits in `n`, have the function return `0`.