Chapter 5

Functions

Chapter 5 - Functions

Outline

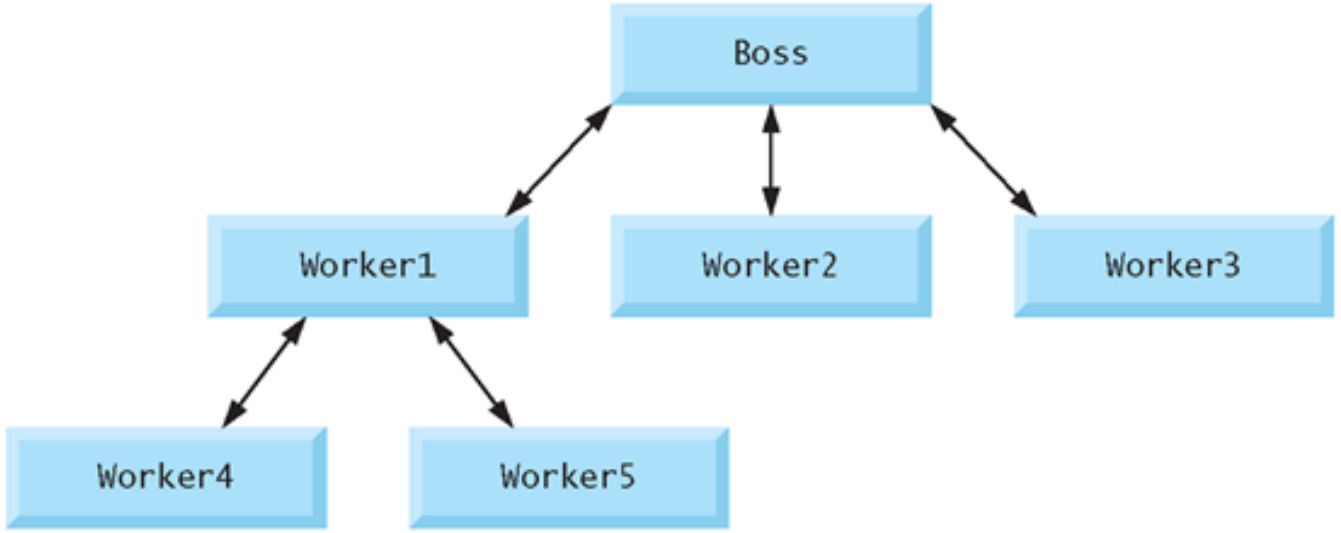
5.2 Program Modules in C
5.3 Math Library Functions
5.4 Functions
5.5 Function Definitions
5.6 Function Prototypes
5.7 Header Files
5.8 Calling Functions: Call by Value and Call by Reference
5.9 Random Number Generation
5.10 Examples: Coin Toss Simulation, Dice Simulation
5.11 Storage Classes
5.12 Scope Rules
5.13 Recursion
5.14 Examples Using Recursion: Factorial, Fibonacci Series
5.15 Recursion vs. Iteration

5.2 Program Modules in C

- Functions
  - Modules in C
  - Programs combine user-defined functions with library functions
    - C standard library has a wide variety of functions
- Function calls
  - Invoking functions
    - Provide function name and arguments (data)
    - Function performs operations or manipulations
    - Function returns results
  - Function call analogy:
    - Boss asks worker to complete task
      - Worker gets information, does task, returns result
      - Information hiding: boss does not know details

5.2 Program Modules in C

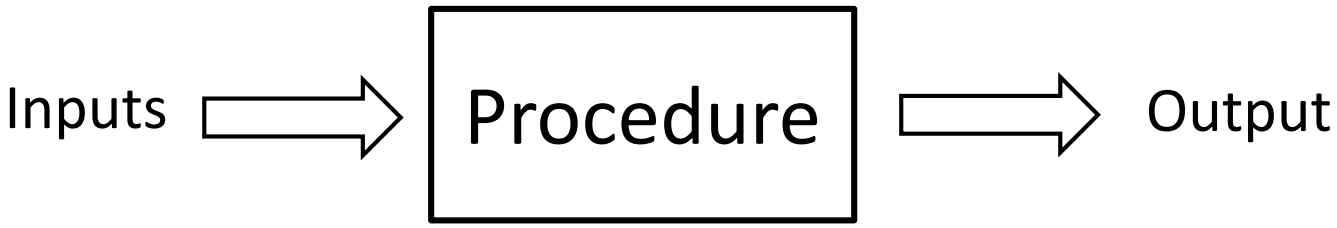
Fig. 5.1 Hierarchical boss function/worker function relationship.



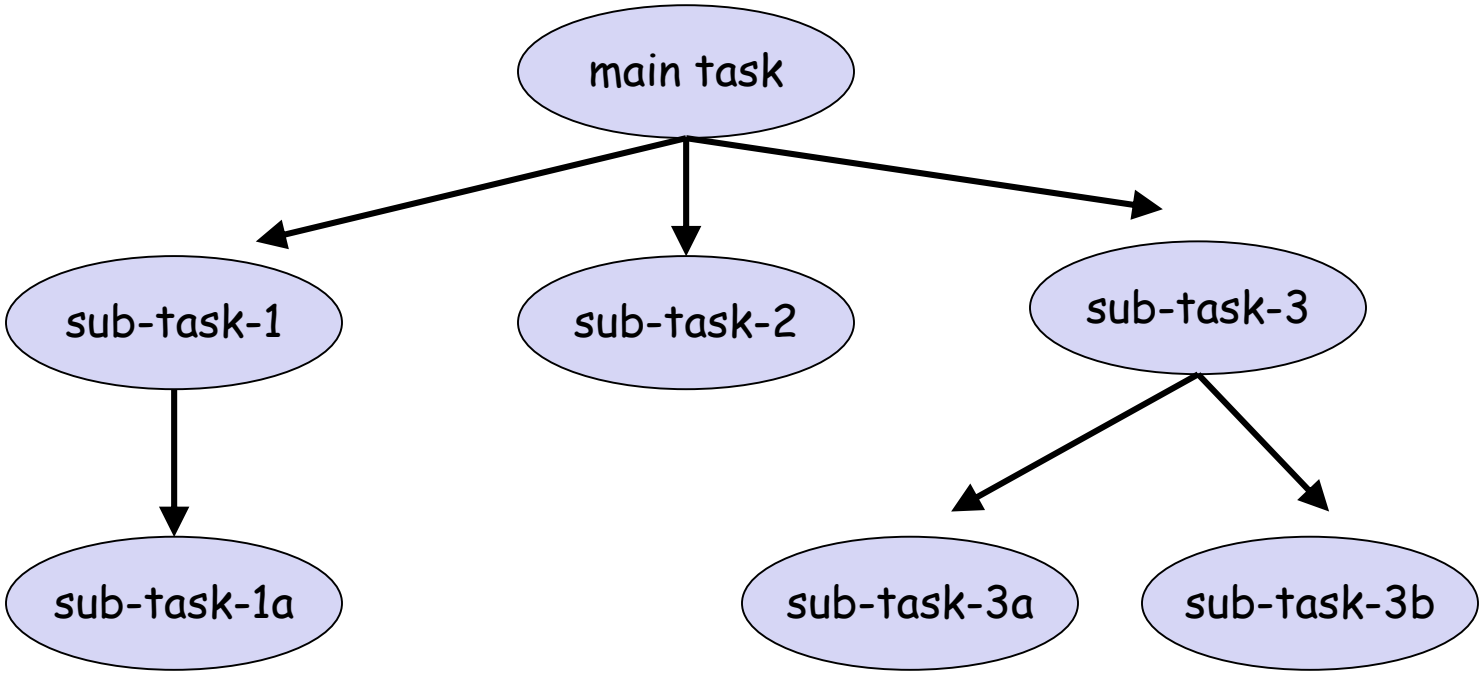
Abstraction

- Divide the main task to sub-tasks
- Consider each sub-task as a main task, and divide into sub-sub-tasks, ... (Divide-and-conquer)
- Top-down design
- Each task is implemented by a procedure (C function)

- A procedure may have parameters:
  - Input parameters: which data will the procedure work on?
  - Output parameter: what value will the procedure produce?



Abstraction Example



Advantages of Abstraction

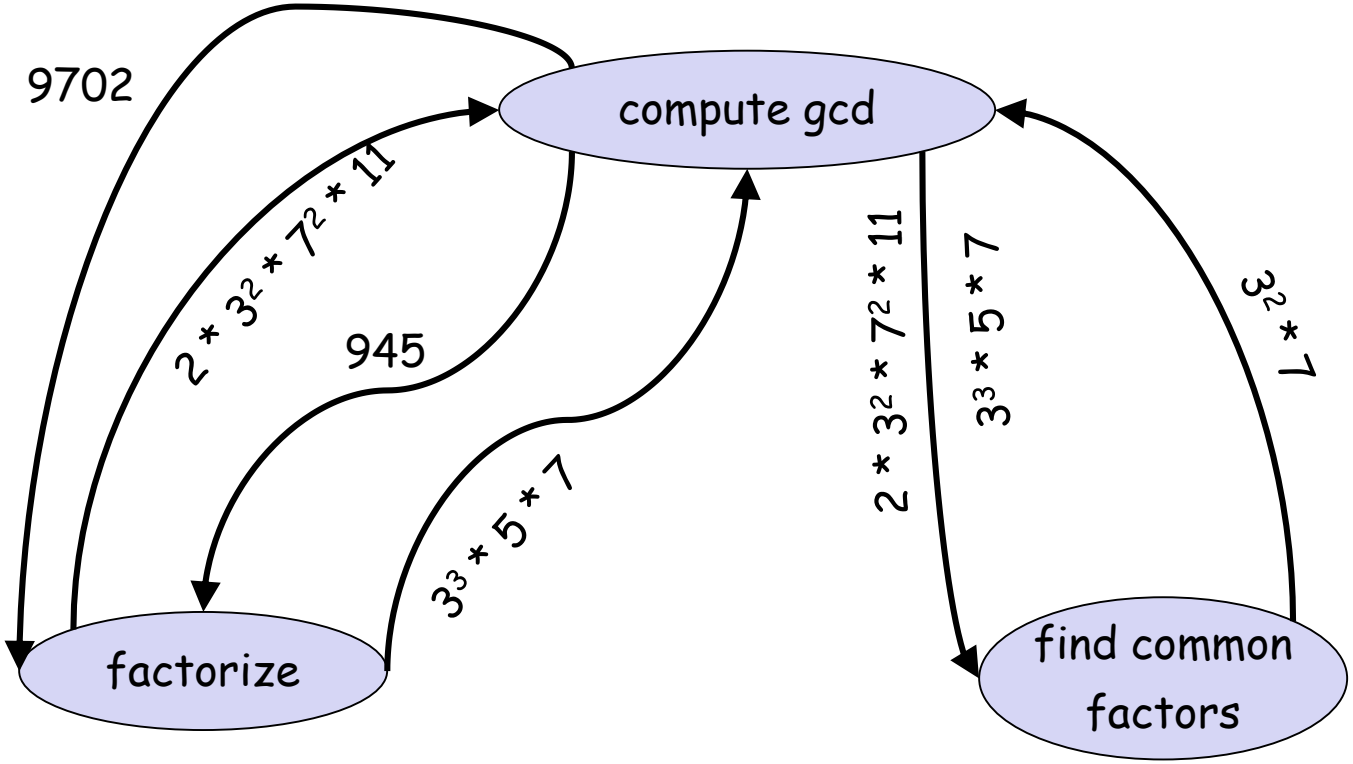
- Procedures are only interested in WHAT sub-procedures are doing, not HOW they are doing it
- Smaller units are easier to manage
- Maintaining is easier
  - if the HOW of the sub-procedure changes, the super-procedure is not affected

Abstraction Example: Greatest Common Divisor

- Find the Greatest Common Divisor (GCD) of two integer numbers.
- Algorithm (abstraction):
  1) decompose the first number to its prime factors
  2) decompose the second number to its prime factors
  3) find the common factors of both numbers
  4) compute the gcd from the common factors

- Sample numbers: 9702 and 945
  1) $9702 = 2 * 3^2 * 7^2 * 11$
  2) $945 = 3^3 * 5 * 7$
  3) $3^2 * 7$
  4) 63

Abstraction Example: Greatest Common Divisor



5.3 Math Library Functions

- Math library functions
  - perform common mathematical calculations
  - `#include <math.h>`
- Format for calling functions
  - `FunctionName( argument );`
    - If multiple arguments, use comma-separated list
  - `printf( "%.2f", sqrt( 900.0 ) );`
    - Calls function `sqrt`, which returns the square root of its argument
    - All math functions return data type `double`
  - Arguments may be constants, variables, or expressions

5.3 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( 5.0 ) is 5.0`
		`fabs( 0.0 ) is 0.0`
		`fabs( -5.0 ) is 5.0`

`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`

5.3 Math Library Functions

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.

Formula for converting degrees (0-360) to radian :
Radian = Degree * 3.14 / 180

5.4 Functions

- Functions
  - Modularize a program
  - All variables defined inside functions are local variables
    - Known only in function defined
  - Parameters
    - Communicate information between functions
    - Local variables
- Benefits of functions
  - Divide and conquer
    - Manageable program development
  - Software reusability
    - Use existing functions as building blocks for new programs
    - Abstraction - hide internal details (library functions)
  - Avoid code repetition

5.5 Function Definitions

- Function definition format

return-value-type function-name (parameter-list)
{
declarations and statements
}

  - **Function-name:** any valid identifier
  - **Parameter-list:** comma separated list, declares parameters
    - A type must be listed explicitly for each parameter, unless the parameter is of type `int`
  - **Definitions and statements:** function body (block)
    - Variables can be defined inside blocks (can be nested)
    - Important: Functions can not be defined inside other functions!

5.5 Function Definitions

- **Return control:** There are three ways to return control from a called function to the point at which a function was invoked.

1) If the function does *not* return a result, control is returned simply when the function-ending right brace is reached.

2) Whenever in function, by executing the statement

```
return;
```

3) If the function *does* return a result, the statement

```
return expression;
```

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

5.5 Function Definitions

- **Return-value-type:**
  - Data type of the result **(default is `int`)**
  - The return statement can be used anywhere in function many times.
  - Usage of return statement is optional, you may not use it at all.
  - Return statement can be used with aritmetic expressions.

    return (3*a+b);
  - `void` – indicates that the function returns no values
  - If the return-value-type is void, you can still use the return statement, but without a value.

    return;

5.5 Function Definitions

- **Main function:**
  - The main is a function called by the operating system.
  - There are many ways to declare the main function.
  - If return-value-type is omitted, it is considered as int by default.
  - If no arguments are specified, they are considered as void.

```
int main() {
  return 0;
}
```

```
main() {
  return 0;
}
```

```
void main() {
  return;
}
```

```
int main(void) {
}
```

```
void main(void) {
}
```

```
int main(int argc, char * argv[] )  {
}
```

How to obtain the return value of main?

- In Unix operating system, enter following:

  echo $?

- In Windows operating system (command line)

  echo %ERRORLEVEL%

```
int main()
{
  return 5;
}
```

```
C:\>myprog.exe

C:\>echo %ERRORLEVEL%

   5
```

Example1: Calling a 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()
{
   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 main


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

} // end function square
```

Program Output

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

Example2: Calling a function

```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

int main()
{
   int number1; // first integer
   int number2; // second integer
   int number3; // third integer

   printf( "Enter three integers: " );
   scanf( "%d%d%d", &number1, &number2, &number3 );

   /* number1, number2 and number3 are arguments
      to the maximum function call */
   printf( "Maximum is: %d\n", maximum( number1, number2, number3 ) );
} // end main
```

Part 1 of 2

Example2: Calling a function

Part 2 of 2

```
/* 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, assign y to max
      max = y;
   }

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

   return max;       // max is largest value

} // end function maximum
```

Program 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
```

5.6 Function Prototypes

- Function prototype
  - Function definition line (signature) without function body.
  - Function name
  - Parameters – what the function takes in
  - Return type – data type function returns (default `int`)
  - Used to validate functions
  - Important: Prototype only needed if function definition comes after use in program.

  - The function with the prototype

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

    - Takes in 3 `int`s
    - Returns an `int`

5.7 Header Files

- Header files
  - Contain function prototypes for library functions
  - `<stdlib.h>` , `<math.h>` , etc
  - Load with `#include <filename>`

```
#include <math.h>
```

- Custom header files *(i.e. written by programmer)*
  - Create file with functions
  - Save as `filename.h`
  - Load in other files with `#include "filename.h"`
  - Reuse functions

5.7 Header Files
(Containing built-in function prototypes)

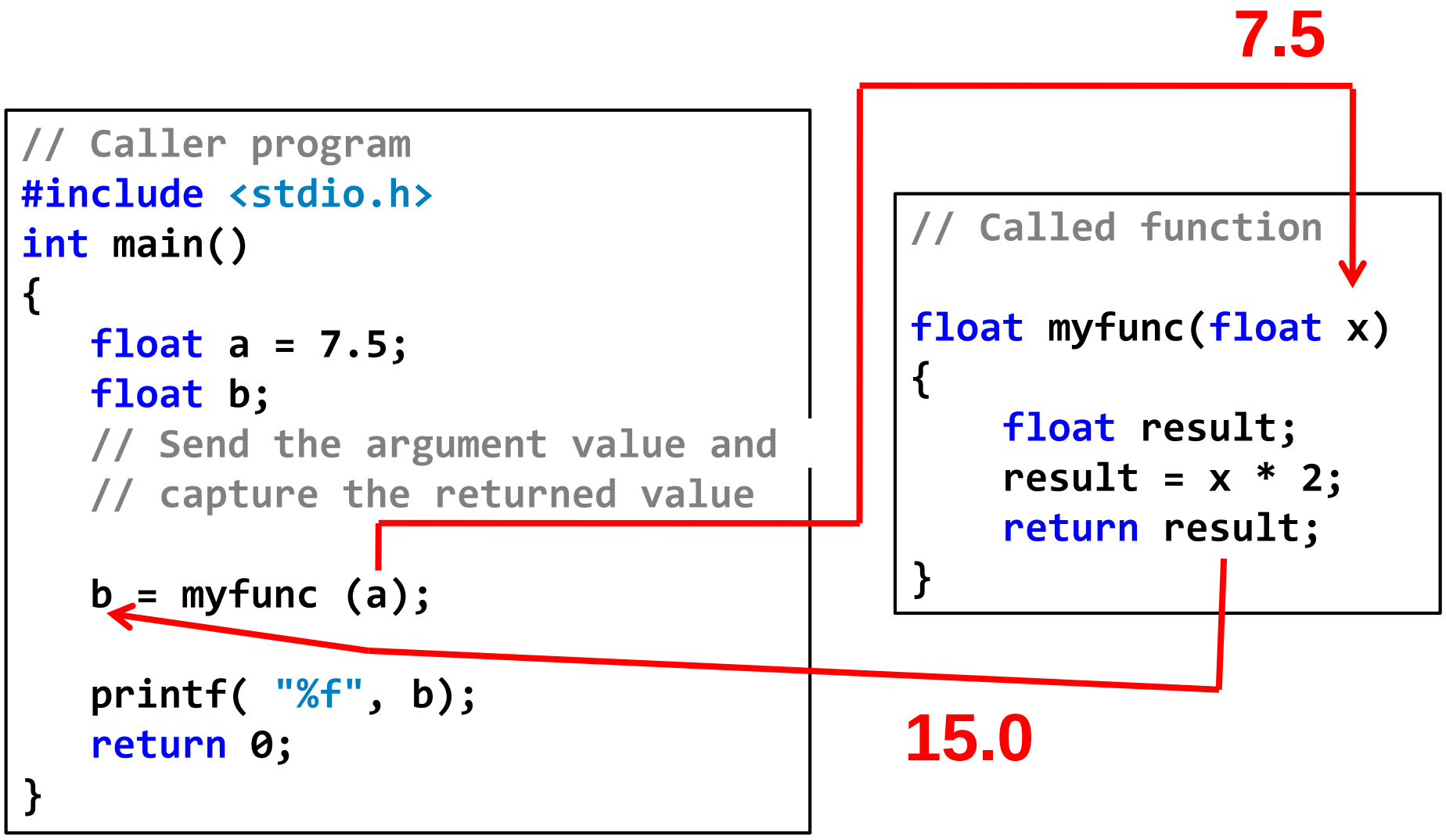
Standard library header	Explanation
`<ctype.h>`	Contains function prototypes for functions that test characters for certain properties, and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa.
`<limits.h>`	Contains the integral size limits 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 is 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.
`<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 header.

5.8 Calling Functions: Call by Value and Call by Reference

- Call by value
  - Copy of argument passed to function
  - **Changes in function do not effect original**
  - Use when function does not need to modify argument
    - Avoids accidental changes
- Call by reference
  - Passes original argument
  - **Changes in function effect original**
  - Only used with trusted functions
- For now, we focus on call-by-value

Example: Function Call by Value



5.9 Random Number Generation

- **rand ( ) function**
  - Load <stdlib.h>
  - Returns "random" number between 0 and RAND_MAX (at least 32767)

```
i = rand();
```

  - Pseudorandom
    - Preset sequence of "random" numbers
    - Same sequence for every function call
- Scaling
  - To get a random number between 1 and n

```
1 + ( rand() % n )
```

    - rand() % n returns a number between 0 and n - 1
    - Add 1 to make random number between 1 and n

```
1 + ( rand() % 6)
```

      - number between 1 and 6

5.9 Random Number Generation

- General formula to get a random number between a and b is :

```
number = a + ( rand() % (b-a+1) )
```

5.9 Random Number Generation

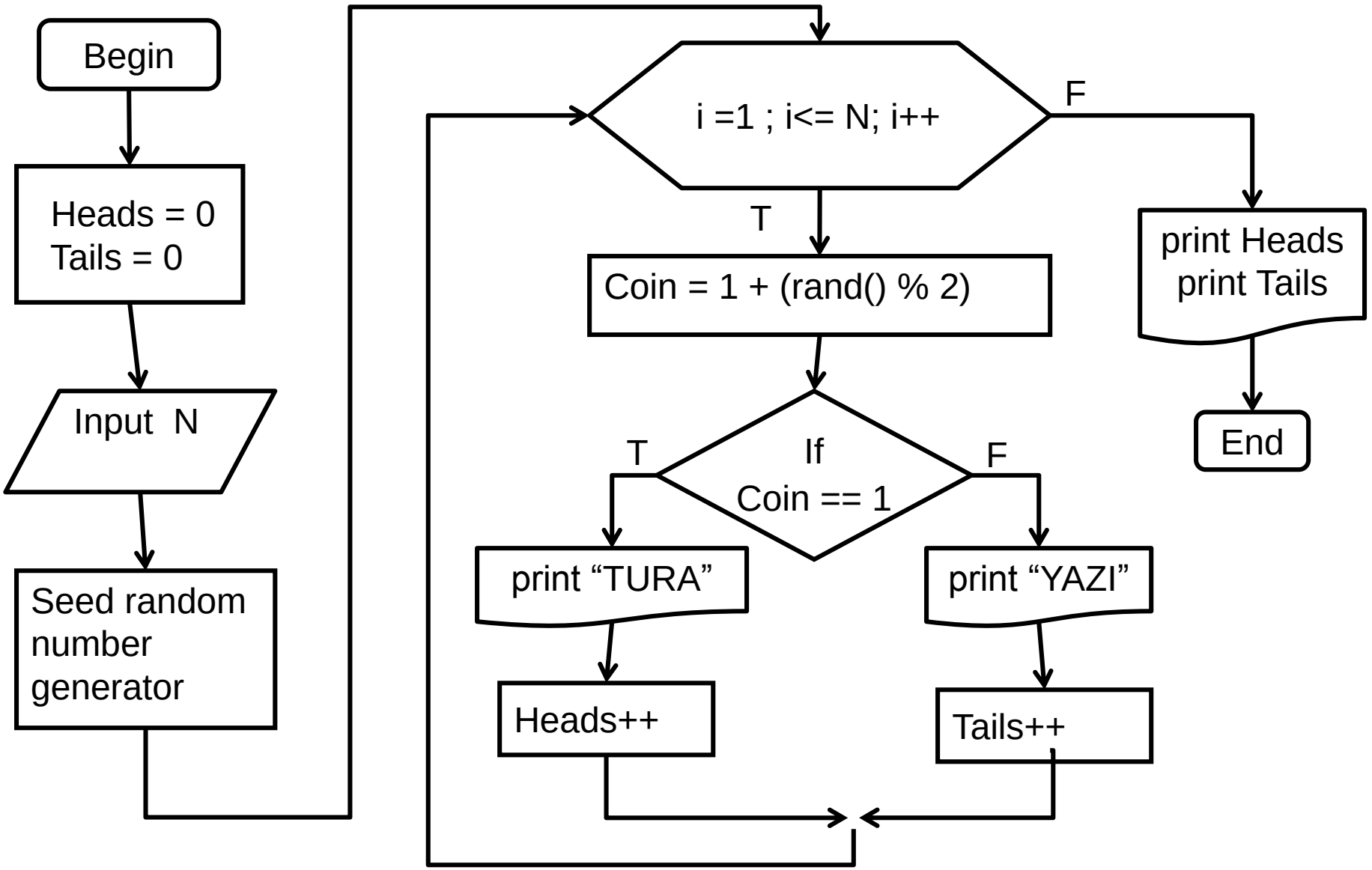
- **`srand()` function**
  - `<stdlib.h>`
  - Takes an integer seed and jumps to that location in its "random" sequence

    `srand(` *seed* **`);`**

  - Used to generate different random sequence for every rand() function call.

  - **`srand( time( NULL ) );`** `//load <time.h>`
    - `time( NULL )`
      - Returns the time (in seconds) at which the program was executed
      - “Randomizes" the seed which guarantees complete randomness

Example: Coin Toss Simulation



Example: Coin Toss Simulation

```c
#include <stdio.h>
#include <stdlib.h> // srand,rand
#include <time.h>   // time

int main()
{
    int N;              // Number of simulations
    int i;              // Loop counter
    int Coin;           // Random number (1 or 2)
    int heads = 0, tails = 0;   // Counters for heads and tails

    printf("Kaç kere yazı-tura simülasyonu yapılacak ? ");
    scanf("%d", &N);

    srand(time(NULL));  // Seed random number generator
```

Part 1 of 2

Example: Coin Toss Simulation

Part 2 of 2

```
for (i = 1;  i <= N;  i++) {
        // Random sayı oluştur.
        Coin = 1 + (rand() % 2);

        if (Coin == 1) {
            printf("* TURA *\n\n");
            heads++;
        } else {
            printf(" YAZI \n\n");
            tails++;
        }
    }

    printf("Tura sayısı: %d   Yüzdesi: %.f \n",
           heads, 100.0 * heads / N);

    printf("Yazı sayısı: %d   Yüzdesi: %.f \n",
          tails, 100.0 * tails / N);

} // end main
```

Program Output

```
Kaç kere yazı-tura simülasyonu yapılacak ? 5
 YAZI

 YAZI

 YAZI

* TURA *

 YAZI

Tura sayısı: 1   Yüzdesi: 20
Yazı sayısı: 4   Yüzdesi: 80
```

Example: Dice Simulation

```c
/* Fig. 5.7: fig05_07.c
   Space shifted, scaled integers produced by 1 + rand() % 6 */
#include <stdio.h>
#include <stdlib.h>

int main()
{
   int i; // counter

   // loop 20 times
   for ( i = 1; i <= 20; i++ ) {

      // pick random number from 1 to 6 and output it
      printf( "%10d", 1 + ( rand() % 6 ) );

      // if counter is divisible by 5, begin new line of output
      if ( i % 5 == 0 ) {
         printf( "\n" );
      }

   } // end for

} // end main
```

Program Output

```
6 6 5 5 6
5 1 1 5 3
6 6 2 4 2
6 2 3 4 1
```

Example: Dice Simulation (with frequency counting)

```c
/* Fig. 5.8: fig05_08.c
   Roll a six-sided die 6000 times */
#include <stdio.h>
#include <stdlib.h>

int main()
{
   int frequency1 = 0; // rolled 1 counter
   int frequency2 = 0; // rolled 2 counter
   int frequency3 = 0; // rolled 3 counter
   int frequency4 = 0; // rolled 4 counter
   int frequency5 = 0; // rolled 5 counter
   int frequency6 = 0; // rolled 6 counter

   int roll; // roll counter, value 1 to 6000
   int face; // represents one roll of the die, value 1 to 6
```

Part 1 of 3

Part 2 of 3

```
// loop 6000 times and summarize results
  for ( roll = 1; roll <= 6000; roll++ ) {
     face = 1 + rand() % 6; // random number from 1 to 6
     // determine face value and increment appropriate counter
     switch ( face ) {
        case 1: // rolled 1
           ++frequency1;
           break;

        case 2: // rolled 2
           ++frequency2;
           break;

        case 3: // rolled 3
           ++frequency3;
           break;

        case 4: // rolled 4
           ++frequency4;
           break;

        case 5: // rolled 5
           ++frequency5;
           break;

        case 6: // rolled 6
           ++frequency6;
           break; // optional
     } // end switch
  } // end for
```

Part 3 of 3

```
 // display results in tabular format
   printf( "%s%13s\n", "Face", "Frequency" );
   printf( "   1%13d\n", frequency1 );
   printf( "   2%13d\n", frequency2 );
   printf( "   3%13d\n", frequency3 );
   printf( "   4%13d\n", frequency4 );
   printf( "   5%13d\n", frequency5 );
   printf( "   6%13d\n", frequency6 );

} // end main
```

Program Output

```
Face      Frequency
   1           1003
   2           1017
   3            983
   4            994
   5           1004
   6            999
```

5.11 Storage Classes

- Storage duration : how long an object exists in memory
- Static storage
  - Variables exist for entire program execution
  - `static`: local variables defined in functions.
    - Keep value after function ends
    - Only known in their own function

```
static int x, y;
```

  - `extern`: default for global variables and functions
    - Known in any function

```
extern int x, y;
```

5.12 Scope Rules

- Scope : where object can be referenced in program
- File scope (i.e. Global)
  - Identifier defined outside function, known in all functions
  - Used for global variables, function definitions, function prototypes
- Function scope (i.e. Local)
  - Can only be referenced inside a function body
  - Used for local variables of a function
  - Function prototype scope
    - Used for variable identifiers in parameter list

5.12 Scope Rules

- Block scope (i.e. Local)
  - Identifier declared inside a block
    - Block scope begins at definition, ends at right brace
  - Outer blocks "hidden" from inner blocks if there is a variable with the same name in the inner block

Example: Variable scopes

Part 1 of 3

```c
/* Fig. 5.12: fig05_12.c
   A scoping example */
#include <stdio.h>

// function prototypes
void useLocal( void );
void useStaticLocal( void );
void useGlobal( void );

int x = 1; // global variable

// function main begins program execution
int main()
{
   int x = 5; // local variable to main

   printf("local x in outer scope of main is %d\n", x );

   { // start new scope
      int x = 7; // local variable to new scope

      printf( "local x in inner scope of main is %d\n", x );
   } // end new scope

   printf( "local x in outer scope of main is %d\n", x );
```

Example: Variable scopes

Part 2 of 3

```
   useLocal();       // useLocal has automatic local x
   useStaticLocal(); // useStaticLocal has static local x
   useGlobal();      // useGlobal uses global x
   useLocal();       // useLocal reinitializes automatic local x
   useStaticLocal(); // static local x retains its prior value
   useGlobal();      // global x also retains its value

   printf( "\nlocal x in main is %d\n", x );

} // end main
```

```
/* useLocal reinitializes local variable x during each call */
void useLocal( void )
{
   int x = 25;  // initialized each time useLocal is called

   printf( "\nlocal x in useLocal is %d after entering useLocal\n", x );
   x++;
   printf( "local x in useLocal is %d before exiting useLocal\n", x );
} // end function useLocal
```

Example: Variable scopes

Part 3 of 3

```
/* useStaticLocal initializes static local variable x only the first time
   the function is called; value of x is saved between calls to this function */
void useStaticLocal( void )
{
   // initialized only first time useStaticLocal is called
   static int x = 50;

   printf( "\nlocal static x is %d on entering useStaticLocal\n", x );
   x++;
   printf( "local static x is %d on exiting useStaticLocal\n", x );
} // end function useStaticLocal
```

```
/* function useGlobal modifies global variable x during each call */
void useGlobal( void )
{
   printf( "\nglobal x is %d on entering useGlobal\n", x );
   x *= 10;
   printf( "global x is %d on exiting useGlobal\n", x );
} // end function useGlobal
```

Program Output

```
local x in outer scope of main is 5
local x in inner scope of main is 7
local x in outer scope of main is 5

local x in a is 25 after entering a
local x in a is 26 before exiting a

local static x is 50 on entering b
local static x is 51 on exiting b

global x is 1 on entering c
global x is 10 on exiting c

local x in a is 25 after entering a
local x in a is 26 before exiting a

local static x is 51 on entering b
local static x is 52 on exiting b

global x is 10 on entering c
global x is 100 on exiting c
local x in main is 5
```

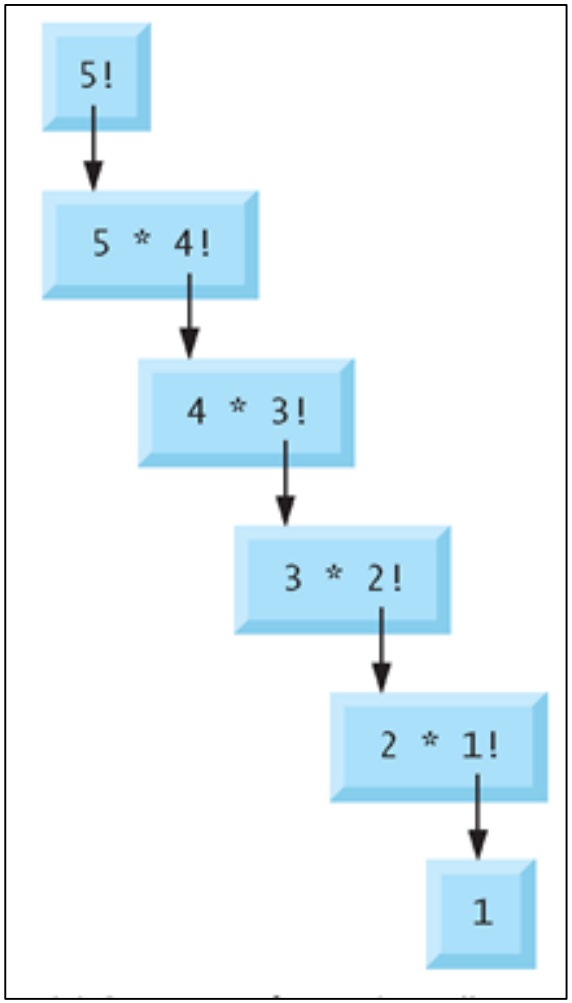
5.13 Recursion

- Recursive functions
  - **Functions that call themselves are recursive**
  - Can only solve a base case
  - Divide a problem up into
    - What it can do **(base part)**
    - What it cannot do **(recursive part)**
      - What it cannot do resembles original problem
      - The function launches a new copy of itself (recursion step) to solve what it cannot do
  - Eventually base case gets solved
    - Gets plugged in, works its way up and solves whole problem

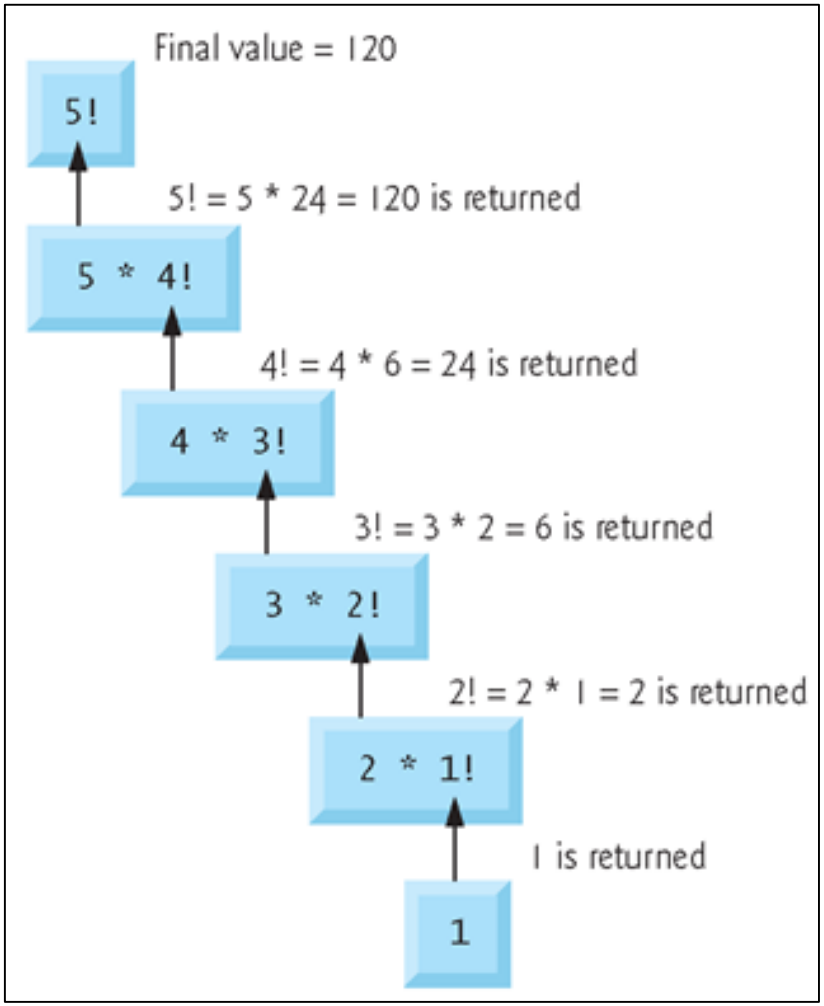
5.13 Recursion Example: Factorial

- Example: Factorial
  - `5! = 5 * 4 * 3 * 2 * 1`
  - Notice that
    - `5! = 5 * 4!`
    - `4! = 4 * 3! ...`
  - Can compute factorials recursively
  - Solve base case (`1! = 1`) then plug in
    - `2! = 2 * 1! = 2 * 1 = 2;`
    - `3! = 3 * 2! = 3 * 2 = 6;`

5.13 Recursion Example: Factorial



(a) Sequence of recursive calls



(b) Values returned from each recursive call

Example: Recursive factorial function

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

long factorial( long number ); // function prototype

int main()
{
   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 main
```

Example: Recursive factorial function

Part 2 of 2

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

} // end function factorial
```

Program Output

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

Infinite Recursion

- If the termination condition of a recursive function call is coded with logical errors, the program may run inifinetely.
- The operating system can detect and stop an infinitely recursive program, by displaying a run-time error message such as "Stack Overflow".

```c
#include <stdio.h>

int main()
{
  main(); // Recursively infinite call
  printf("Hello");
}
```

Program Output