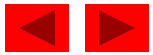
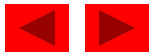
Chapter 5 - Functions

Outline



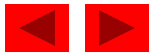
Objectives

- In this chapter, you will learn:
  - To understand how to construct programs modularly from small pieces called functions..
  - To introduce the common math functions available in the C standard library.
  - To be able to create new functions.
  - To understand the mechanisms used to pass information between functions.
  - To introduce simulation techniques using random number generation.
  - To understand how to write and use functions that call themselves.



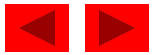
5.1 Introduction

- Divide and conquer
  - Construct a program from smaller pieces or components
    - These smaller pieces are called modules
  - Each piece more manageable than the original program



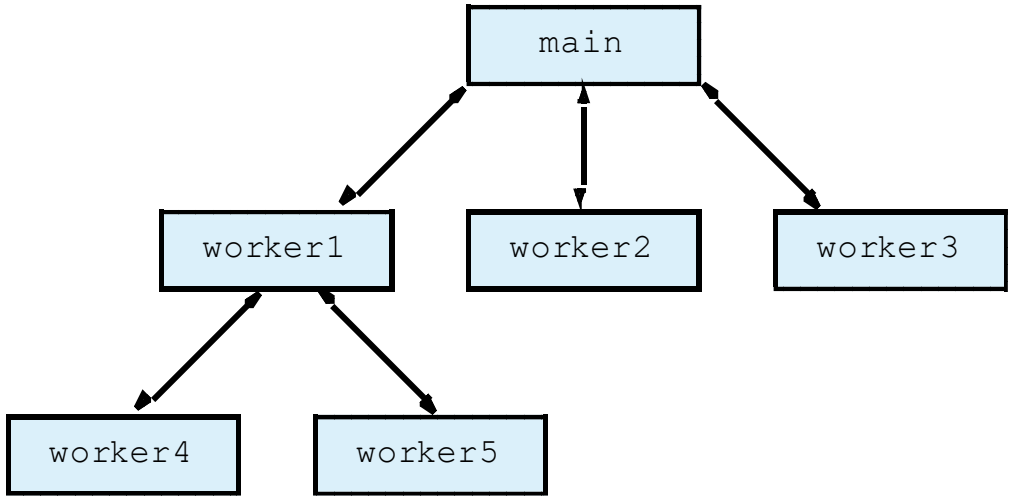
5.2 Program Modules in C

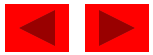
- Functions
  - Modules in C
  - Programs combine user-defined functions with library functions
    - C standard library has a wide variety of functions
- Function calls
  - Invoking functions
    - Provide function name and arguments (data)
    - Function performs operations or manipulations
    - Function 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

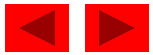
Fig. 5.1 Hierarchical boss function/worker function relationship.





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

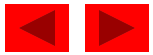
Function	Description	Example
`sqrt( x )`	square root of *x*	`sqrt( 900.0 ) is 30.0` `sqrt( 9.0 ) is 3.0`
`exp( x )`	exponential function e^x	`exp( 1.0 ) is 2.718282` `exp( 2.0 ) is 7.389056`
`log( x )`	natural logarithm of *x* (base *e*)	`log( 2.718282 ) is 1.0` `log( 7.389056 ) is 2.0`

`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`

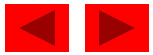
`ceil( x )`	rounds *x* to the smallest integer not less than *x*	`ceil( 9.2 ) is 10.0` `ceil( -9.8 ) is -9.0`
`floor( x )`	rounds *x* to the largest integer not greater than *x*	`floor( 9.2 ) is 9.0` `floor( -9.8 ) is -10.0`
`pow( x, y )`	*x* raised to power *y* (x^y)	`pow( 2, 7 ) is 128.0` `pow( 9, .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.



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
  }
  ```

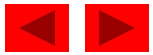
  – Function-name: any valid identifier

  – Return-value-type: data type of the result (default `int`)

    - `void` – indicates that the function returns nothing

  – Parameter-list: comma separated list, declares parameters

    - A type must be listed explicitly for each parameter unless, the parameter is of type `int`

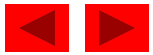


5.5 Function Definitions

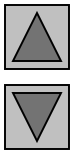
- Function definition format (continued)

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

  – Definitions and statements: function body (block)
    - Variables can be defined inside blocks (can be nested)
    - Functions can not be defined inside other functions
  – Returning control
    - If nothing returned
      – `return;`
      – or, until reaches right brace
    - If something returned
      – `return` *expression*`;`



```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()
{
   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 */
```

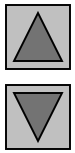


Outline

fig05_03.c (Part 1 of 2)

```
23 /* square function definition returns square of an integer */
24 int square( int y ) /* y is a copy of argument to function */
25 {
26    return y * y; /* returns square of y as an int */
27
28 } /* end function square */
```

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

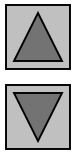


Outline

fig05_03.c (Part 2 of 2)

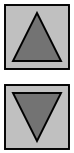
Program Output

```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()
{
   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 ) );

   return 0; /* indicates successful termination */

} /* end main */

```



Outline

fig05_04.c (Part 1 of 2)



Outline

```
25 /* Function maximum definition */
26 /* x, y and z are parameters */
27 int maximum( int x, int y, int z )
28 {
29    int max = x;      /* assume x is largest */
30
31    if ( y > max ) { /* if y is larger than max, assign y to max */
32       max = y;
33    } /* end if */
34
35    if ( z > max ) { /* if z is larger than max, assign z to max */
36       max = z;
37    } /* end if */
38
39    return max;       /* max is largest value */
40
41 } /* end function maximum */
```

fig05_04.c (Part 2 of 2)

```
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
```

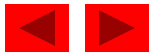
Program Output

5.6 Function Prototypes

- Function prototype
  - Function name
  - Parameters – what the function takes in
  - Return type – data type function returns (default `int`)
  - Used to validate functions
  - Prototype only needed if function definition comes after use in program
  - The function with the prototype

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

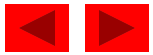
    - Takes in 3 `int`s
    - Returns an `int`
- Promotion rules and conversions
  - Converting to lower types can lead to errors



5.6 Function Prototypes

Data types	`printf` conversion specifications	`scanf` conversion specifications
`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`
`short`	`%hd`	`%hd`
`char`	`%c`	`%c`

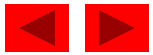
Fig. 5.5 Promotion hierarchy for data types.



5.7 Header Files

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

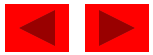
    ```
    #include <math.h>
    ```
- Custom header files
  - Create file with functions
  - Save as `filename.h`
  - Load in other files with `#include "filename.h"`
  - Reuse functions



5.7 Header Files

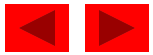
Standard library header	Explanation
<assert.h>	Contains macros and information for adding diagnostics that aid program debugging.
<ctype.h>	Contains function prototypes for functions that test characters for certain properties, and function prototypes for functions that can be used to convert lowercase 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 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.
<setjmp.h>	Contains function prototypes for functions that allow bypassing of the usual function call and return sequence.
<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 definitions of types used by C for performing certain 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 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



5.9 Random Number Generation

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

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

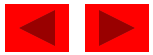
  - Pseudorandom
    - Preset sequence of "random" numbers
    - Same sequence for every function call
- Scaling
  - To get a random number between `1` and `n`

```
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`

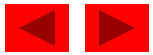


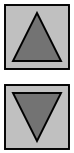
5.9 Random Number Generation

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

    ```
    srand( seed );
    ```

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



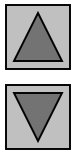


Outline

fig05_07.c

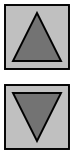
```
1  /* Fig. 5.7: fig05_07.c
2     Shifted, scaled integers produced by 1 + rand() % 6 */
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  /* function main begins program execution */
7  int main()
8  {
9     int i; /* counter */
10
11    /* loop 20 times */
12    for ( i = 1; i <= 20; i++ ) {
13
14       /* pick random number from 1 to 6 and output it */
15       printf( "%10d", 1 + ( rand() % 6 ) );
16
17       /* if counter is divisible by 5, begin new line of output */
18       if ( i % 5 == 0 ) {
19          printf( "\n" );
20       } /* end if */
21
22    } /* end for */
23
24    return 0; /* indicates successful termination */
25
26 } /* end main */
```

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



Outline

Program Output

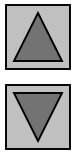


Outline

fig05_08.c (Part 1 of 3)

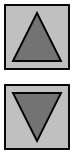
```
1  /* Fig. 5.8: fig05_08.c
2     Roll a six-sided die 6000 times */
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  /* function main begins program execution */
7  int main()
8  {
9     int frequency1 = 0; /* rolled 1 counter */
10    int frequency2 = 0; /* rolled 2 counter */
11    int frequency3 = 0; /* rolled 3 counter */
12    int frequency4 = 0; /* rolled 4 counter */
13    int frequency5 = 0; /* rolled 5 counter */
14    int frequency6 = 0; /* rolled 6 counter */
15
16    int roll; /* roll counter */
17    int face; /* represents one roll of the die, value 1 to 6 */
18
19    /* loop 6000 times and summarize results */
20    for ( roll = 1; roll <= 6000; roll++ ) {
21       face = 1 + rand() % 6; /* random number from 1 to 6 */
22
```

```
23      /* determine face value and increment appropriate counter */
24      switch ( face ) {
25
26         case 1:          /* rolled 1 */
27            ++frequency1;
28            break;
29
30         case 2:          /* rolled 2 */
31            ++frequency2;
32            break;
33
34         case 3:          /* rolled 3 */
35            ++frequency3;
36            break;
37
38         case 4:          /* rolled 4 */
39            ++frequency4;
40            break;
41
42         case 5:          /* rolled 5 */
43            ++frequency5;
44            break;
45
```



Outline

fig05_08.c (Part 2 of 3)



Outline

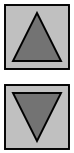
fig05_08.c (Part 3 of 3)

```
45
46          case 6:          /* rolled 6 */
47             ++frequency6;
48             break;
49       } /* end switch */
50
51    } /* end for */
52
53    /* display results in tabular format */
54    printf( "%s%13s\n", "Face", "Frequency" );
55    printf( "   1%13d\n", frequency1 );
56    printf( "   2%13d\n", frequency2 );
57    printf( "   3%13d\n", frequency3 );
58    printf( "   4%13d\n", frequency4 );
59    printf( "   5%13d\n", frequency5 );
60    printf( "   6%13d\n", frequency6 );
61
62    return 0; /* indicates successful termination */
63
64 } /* end main */
```

Program Output

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

```c
/* Fig. 5.9: fig05_09.c
   Randomizing die-rolling program */
#include <stdlib.h>
#include <stdio.h>

/* function main begins program execution */
int main()
{
   int i;         /* counter */
   unsigned seed; /* number used to seed random number generator */

   printf( "Enter seed: " );
   scanf( "%u", &seed );

   srand( seed ); /* seed random number generator */

   /* loop 10 times */
   for ( i = 1; i <= 10; i++ ) {

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

```

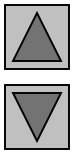


Outline

fig05_09.c (Part 1 of 2)

```
23       /* if counter is divisible by 5, begin a new line of output */
24       if ( i % 5 == 0 ) {
25          printf( "\n" );
26       } /* end if */
27
28    } /* end for */
29
30    return 0; /* indicates successful termination */
31
32 } /* end main */
```

```
Enter seed: 67
        6       1       4       6       2
        1       6       1       6       4
Enter seed: 867
        2       4       6       1       6
        1       1       3       6       2
Enter seed: 67
        6       1       4       6       2
        1       6       1       6       4
```



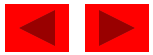
Outline

fig05_09.c (Part 2 of 2)

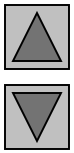
Program Output

5.10 Example: A Game of Chance

- Craps simulator
- Rules
  - Roll two dice
    - 7 or 11 on first throw, player wins
    - 2, 3, or 12 on first throw, player loses
    - 4, 5, 6, 8, 9, 10 - value becomes player's "point"
  - Player must roll his point before rolling 7 to win



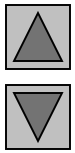
```
1  /* Fig. 5.10: fig05_10.c
2     Craps */
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <time.h> /* contains prototype for function time */
6
7  /* enumeration constants represent game status */
8  enum Status { CONTINUE, WON, LOST };
9
10 int rollDice( void ); /* function prototype */
11
12 /* function main begins program execution */
13 int main()
14 {
15    int sum;        /* sum of rolled dice */
16    int myPoint;    /* point earned */
17
18    enum Status gameStatus; /* can contain CONTINUE, WON, or LOST */
19
20    /* randomize random number generator using current time */
21    srand( time( NULL ) );
22
23    sum = rollDice(  ); /* first roll of the dice */
24
```



Outline

fig05_10.c (Part 1 of 4)

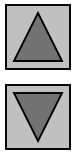
```
25    /* determine game status based on sum of dice */
26    switch( sum ) {
27
28       /* win on first roll */
29       case 7:
30       case 11:
31          gameStatus = WON;
32          break;
33
34       /* lose on first roll */
35       case 2:
36       case 3:
37       case 12:
38          gameStatus = LOST;
39          break;
40
41       /* remember point */
42       default:
43          gameStatus = CONTINUE;
44          myPoint = sum;
45          printf( "Point is %d\n", myPoint );
46          break; /* optional */
47    } /* end switch */
48
```



Outline

fig05_10.c (Part 2 of 4)

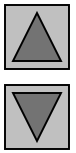
```
49    /* while game not complete */
50    while ( gameStatus == CONTINUE ) {
51       sum = rollDice(  ); /* roll dice again */
52
53       /* determine game status */
54       if ( sum == myPoint ) { /* win by making point */
55          gameStatus = WON;
56       } /* end if */
57       else {
58
59          if ( sum == 7 ) { /* lose by rolling 7 */
60             gameStatus = LOST;
61          } /* end if */
62
63       } /* end else */
64
65    } /* end while */
66
67    /* display won or lost message */
68    if ( gameStatus == WON ) {
69       printf( "Player wins\n" );
70    } /* end if */
71    else {
72       printf( "Player loses\n" );
73    } /* end else */
74
```



Outline

fig05_10.c (Part 3 of 4)

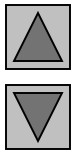
```
75    return 0; /* indicates successful termination */
76
77 } /* end main */
78
79 /* roll dice, calculate sum and display results */
80 int rollDice( void )
81 {
82    int die1;    /* first die */
83    int die2;    /* second die */
84    int workSum; /* sum of dice */
85
86    die1 = 1 + ( rand() % 6 ); /* pick random die1 value */
87    die2 = 1 + ( rand() % 6 ); /* pick random die2 value */
88    workSum = die1 + die2;     /* sum die1 and die2 */
89
90    /* display results of this roll */
91    printf( "Player rolled %d + %d = %d\n", die1, die2, workSum );
92
93    return workSum; /* return sum of dice */
94
95 } /* end function rollRice */
```



Outline

fig05_10.c (Part 4 of 4)

```
Player rolled 5 + 6 = 11
Player wins

Player rolled 4 + 1 = 5
Point is 5
Player rolled 6 + 2 = 8
Player rolled 2 + 1 = 3
Player rolled 3 + 2 = 5
Player wins

Player rolled 1 + 1 = 2
Player loses

Player rolled 1 + 4 = 5
Point is 5
Player rolled 3 + 4 = 7
Player loses
```



Outline

Program Output

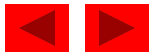
5.11 Storage Classes

- Storage class specifiers
  - Storage duration – how long an object exists in memory
  - Scope – where object can be referenced in program
  - Linkage – specifies the files in which an identifier is known (more in Chapter 14)
- Automatic storage
  - Object created and destroyed within its block
  - `auto:` default for local variables

```
auto double x, y;
```

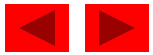
  - `register:` tries to put variable into high-speed registers
    - Can only be used for automatic variables

```
register int counter = 1;
```

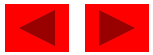
5.11 Storage Classes

- Static storage
  - Variables exist for entire program execution
  - Default value of zero
  - `static:` local variables defined in functions.
    - Keep value after function ends
    - Only known in their own function
  - `extern:` default for global variables and functions
    - Known in any function



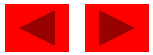
5.12 Scope Rules

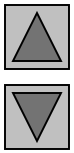
- File scope
  - Identifier defined outside function, known in all functions
  - Used for global variables, function definitions, function prototypes
- Function scope
  - Can only be referenced inside a function body
  - Used only for labels (`start:`, `case:`, etc.)



5.12 Scope Rules

- Block scope
  - Identifier declared inside a block
    - Block scope begins at definition, ends at right brace
  - Used for variables, function parameters (local variables of function)
  - Outer blocks "hidden" from inner blocks if there is a variable with the same name in the inner block
- Function prototype scope
  - Used for identifiers in parameter list

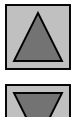




Outline

fig05_12.c (Part 1 of 3)

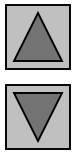
```
1  /* Fig. 5.12: fig05_12.c
2     A scoping example */
3  #include <stdio.h>
4
5  void useLocal( void );       /* function prototype */
6  void useStaticLocal( void ); /* function prototype */
7  void useGlobal( void );      /* function prototype */
8
9  int x = 1; /* global variable */
10
11 /* function main begins program execution */
12 int main()
13 {
14    int x = 5; /* local variable to main */
15
16    printf("local x in outer scope of main is %d\n", x );
17
18    { /* start new scope */
19       int x = 7; /* local variable to new scope */
20
21       printf( "local x in inner scope of main is %d\n", x );
22    } /* end new scope */
23
24    printf( "local x in outer scope of main is %d\n", x );
25
```



Outline

g05_12.c (Part 2 f 3)

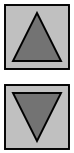
```
26    useLocal();       /* useLocal has automatic local x */
27    useStaticLocal(); /* useStaticLocal has static local x */
28    useGlobal();      /* useGlobal uses global x */
29    useLocal();       /* useLocal reinitializes automatic local x */
30    useStaticLocal(); /* static local x retains its prior value */
31    useGlobal();      /* global x also retains its value */
32
33    printf( "local x in main is %d\n", x );
34
35    return 0; /* indicates successful termination */
36
37 } /* end main */
38
39 /* useLocal reinitializes local variable x during each call */
40 void useLocal( void )
41 {
42    int x = 25;  /* initialized each time useLocal is called */
43
44    printf( "\nlocal x in a is %d after entering a\n", x );
45    x++;
46    printf( "local x in a is %d before exiting a\n", x );
47 } /* end function useLocal */
48
```



Outline

fig05_12.c (Part 3 of 3)

```
49 /* useStaticLocal initializes static local variable x only the first time
50    the function is called; value of x is saved between calls to this
51    function */
52 void useStaticLocal( void )
53 {
54    /* initialized only first time useStaticLocal is called */
55    static int x = 50;
56
57    printf( "\nlocal static x is %d on entering b\n", x );
58    x++;
59    printf( "local static x is %d on exiting b\n", x );
60 } /* end function useStaticLocal */
61
62 /* function useGlobal modifies global variable x during each call */
63 void useGlobal( void )
64 {
65    printf( "\nglobal x is %d on entering c\n", x );
66    x *= 10;
67    printf( "global x is %d on exiting c\n", x );
68 } /* end function useGlobal */
```



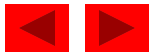
Outline

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
```

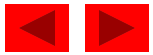
5.13 Recursion

- Recursive functions
  - Functions that call themselves
  - Can only solve a base case
  - Divide a problem up into
    - What it can do
    - What it cannot do
      - 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

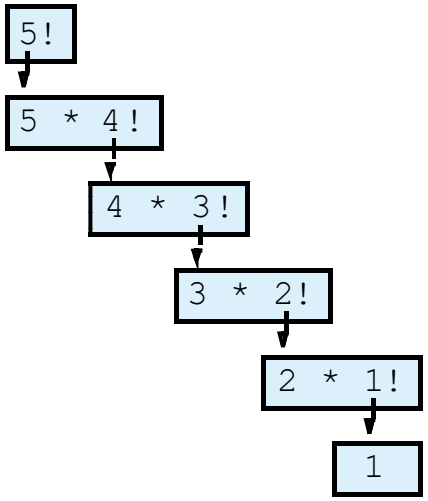


5.13 Recursion

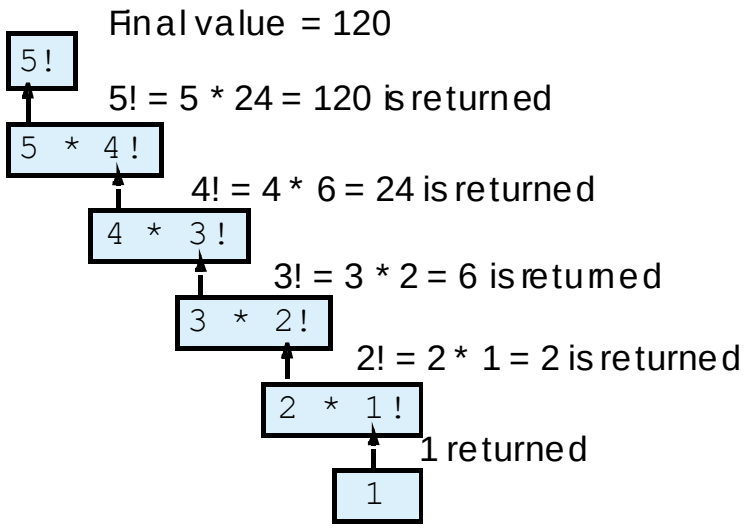
- Example: factorials
  - `5! = 5 * 4 * 3 * 2 * 1`
  - Notice that
    - `5! = 5 * 4!`
    - `4! = 4 * 3! ...`
  - Can compute factorials recursively
  - Solve base case (`1! = 0! = 1`) then plug in
    - `2! = 2 * 1! = 2 * 1 = 2;`
    - `3! = 3 * 2! = 3 * 2 = 6;`



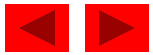
5.13 Recursion

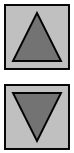


(a) Sequence of recursive calls.



(b) Values returned from each recursive call.

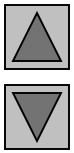




Outline

fig05_14.c (Part 1 of 2)

```
1  /* Fig. 5.14: fig05_14.c
2     Recursive factorial function */
3  #include <stdio.h>
4
5  long factorial( long number ); /* function prototype */
6
7  /* function main begins program execution */
8  int main()
9  {
10    int i; /* counter */
11
12    /* loop 10 times.  During each iteration, calculate
13       factorial( i ) and display result */
14    for ( i = 1; i <= 10; i++ ) {
15       printf( "%2d! = %ld\n", i, factorial( i ) );
16    } /* end for */
17
18    return 0; /* indicates successful termination */
19
20 } /* end main */
21
```



Outline

fig05_14.c (Part 2 of 2)

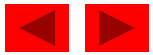
```
22 /* recursive definition of function factorial */
23 long factorial( long number )
24 {
25    /* base case */
26    if ( number <= 1 ) {
27       return 1;
28    } /* end if */
29    else { /* recursive step */
30       return ( number * factorial( number - 1 ) );
31    } /* end else */
32
33 } /* end function factorial */
```

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

5.14 Example Using Recursion: The Fibonacci Series

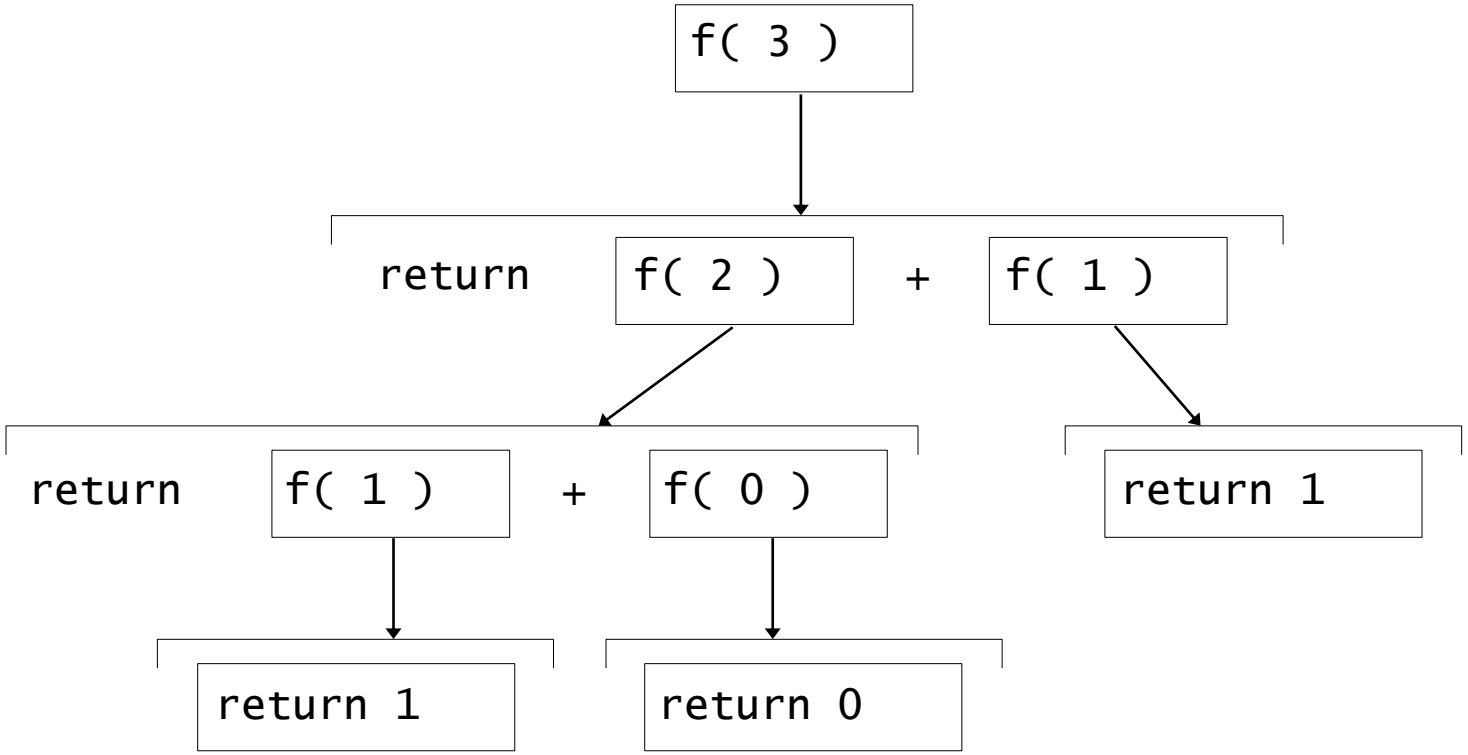
- Fibonacci series: 0, 1, 1, 2, 3, 5, 8...
  - Each number is the sum of the previous two
  - Can be solved recursively:
    - `fib( n ) = fib( n - 1 ) + fib( n - 2 )`
  - Code for the `fibonacci` function

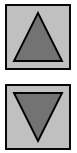
```
long fibonacci( long n )
{
  if (n == 0 || n == 1)  // base case
    return n;
  else
    return fibonacci( n - 1) +
      fibonacci( n - 2 );
}
```

5.14 Example Using Recursion: The Fibonacci Series

- Set of recursive calls to function `fibonacci`



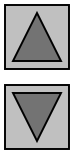


Outline

fig05_15.c (Part 1 of 2)

```
1  /* Fig. 5.15: fig05_15.c
2     Recursive fibonacci function */
3  #include <stdio.h>
4
5  long fibonacci( long n ); /* function prototype */
6
7  /* function main begins program execution */
8  int main()
9  {
10    long result; /* fibonacci value */
11    long number; /* number input by user */
12
13    /* obtain integer from user */
14    printf( "Enter an integer: " );
15    scanf( "%ld", &number );
16
17    /* calculate fibonacci value for number input by user */
18    result = fibonacci( number );
19
20    /* display result */
21    printf( "Fibonacci( %ld ) = %ld\n", number, result );
22
23    return 0; /* indicates successful termination */
24
25 } /* end main */
26
```

```
27 /* Recursive definition of function fibonacci */
28 long fibonacci( long n )
29 {
30    /* base case */
31    if ( n == 0 || n == 1 ) {
32       return n;
33    } /* end if */
34    else { /* recursive step */
35       return fibonacci( n - 1 ) + fibonacci( n - 2 );
36    } /* end else */
37
38 } /* end function fibonacci */
```



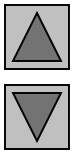
Outline

fig05_15.c (Part 2 of 2)

Program Output

```
Enter an integer: 0
Fibonacci( 0 ) = 0

Enter an integer: 1
Fibonacci( 1 ) = 1

Enter an integer: 2
Fibonacci( 2 ) = 1

Enter an integer: 3
Fibonacci( 3 ) = 2

Enter an integer: 4
Fibonacci( 4 ) = 3
```

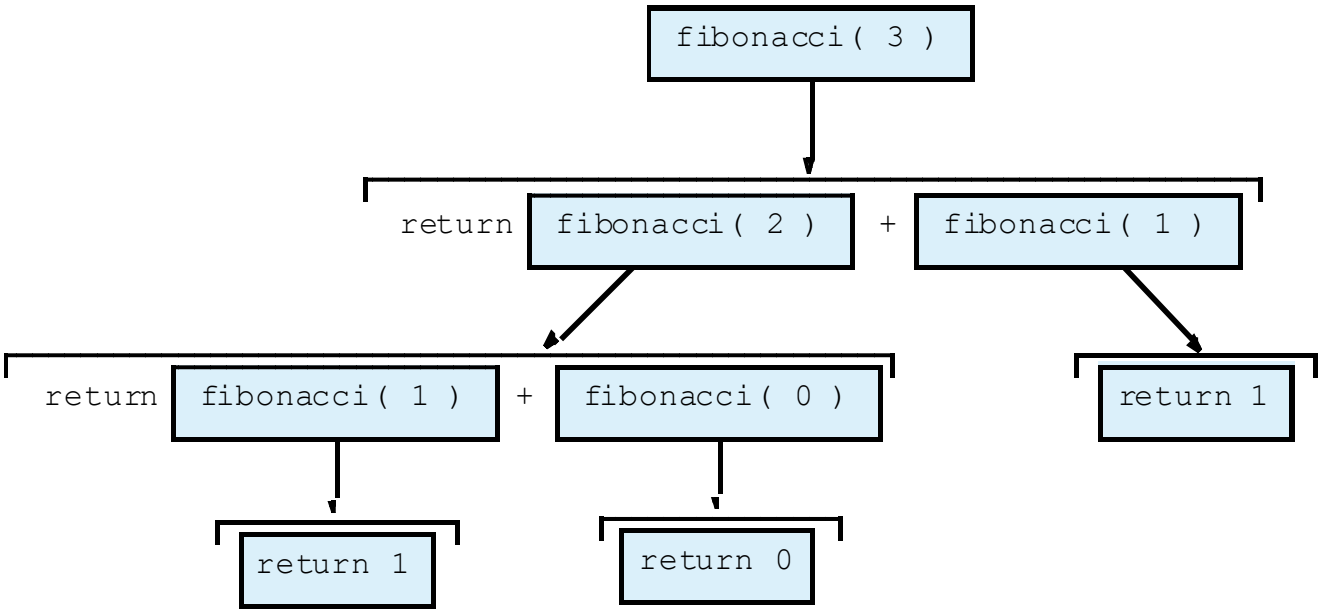
```
Enter an integer: 5
Fibonacci( 5 ) = 5

Enter an integer: 6
Fibonacci( 6 ) = 8

Enter an integer: 10
Fibonacci( 10 ) = 55

Enter an integer: 20
Fibonacci( 20 ) = 6765

Enter an integer: 30
Fibonacci( 30 ) = 832040

Enter an integer: 35
Fibonacci( 35 ) = 9227465
```

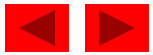


Outline

Program Output (continued)

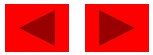
5.14 Example Using Recursion: The Fibonacci Series





5.15 Recursion vs. Iteration

- Repetition
  - Iteration: explicit loop
  - Recursion: repeated function calls
- Termination
  - Iteration: loop condition fails
  - Recursion: base case recognized
- Both can have infinite loops
- Balance
  - Choice between performance (iteration) and good software engineering (recursion)



5.15 Recursion vs. Iteration

Chapter	Recursion Examples and Exercises
Chapter 5	Factorial function Fibonacci functions Greatest common divisor Sum of two integers Multiply two integers Raising an integer to an integer power Towers of Hanoi Recursive `main` Printing keyboard inputs in reverse Visualizing recursion
Chapter 6	Sum the elements of an array Print an array Print an array backwards Print a string backwards Check if a string is a palindrome Minimum value in an array Selection sort Quicksort Linear search Binary search
Chapter 7	Eight Queens Maze traversal
Chapter 8	Printing a string input at the keyboard backwards
Chapter 12	Linked list insert Linked list delete Search a linked list Print a linked list backwards Binary tree insert Preorder traversal of a binary tree Inorder traversal of a binary tree Postorder traversal of a binary tree

Fig. 5.17 Summary of recursion examples and exercises in the text.