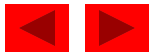
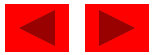
Chapter 5 - Functions

Outline



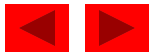
5.1 Introduction

- 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



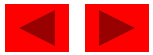
5.2 Program Modules in C

- 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



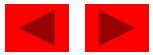
5.3 Math Library Functions

- 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



5.4 Functions

- Functions
  - Modularize a program
  - All variables declared 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

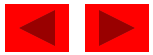


5.5 Function Definitions

- 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`**

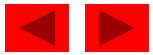


5.5 Function Definitions

- Function definition format (continued)

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

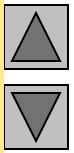
  – Declarations and statements: function body (block)
    - Variables can be declared inside blocks (can be nested)
    - Functions can not be defined inside other functions
  – Returning control
    - If nothing returned
      – **`return;`**
      – or, until reaches right brace
    - If something returned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



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

int maximum( int, int, int );   /* function prototype */

int main()
{
   int a, b, c;

   printf( "Enter three integers: " );
   scanf( "%d%d%d", &a, &b, &c );
   printf( "Maximum is: %d\n", maximum( a, b, c ) );

   return 0;
}

/* Function maximum definition */
int maximum( int x, int y, int z )
{
   int max = x;

   if ( y > max )
      max = y;

   if ( z > max )
      max = z;

   return max;
}
```

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

Program Output



Outline

1. Function prototype (3 parameters)

2. Input values

2.1 Call function

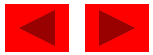
3. Function definition

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, int, int );
    ```

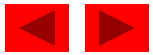
    - Takes in 3 **`int`**s
    - Returns an **`int`**
- Promotion rules and conversions
  - Converting to lower types can lead to errors



5.7 Header Files

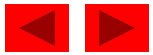
- Header files
  - Contain function prototypes for library functions
  - **<stdlib.h>** , **<math.h>** , etc
  - Load with **#include <filename>**

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



5.8 Calling Functions: Call by Value and Call by Reference

- Used when invoking functions
- 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();
```

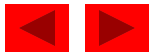
  - Pseudorandom
    - Preset sequence of "random" numbers
    - Same sequence for every function call
- Scaling
  - To get a random number between **`1`** and **`n`**

```
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)
```

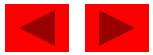
      - number between **`1`** and **`6`**



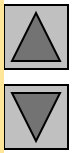
5.9 Random Number Generation

- **`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



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

int main()
{
   int i;
   unsigned seed;

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

   for ( i = 1; i <= 10; i++ ) {
      printf( "%10d", 1 + ( rand() % 6 ) );

      if ( i % 5 == 0 )
         printf( "\n" );
   }

   return 0;
}
```



Outline

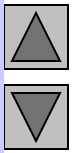
1. Initialize seed

2. Input value for seed

2.1 Use `srand` to change random sequence

2.2 Define Loop

3. Generate and output random numbers



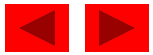
Outline

Program Output

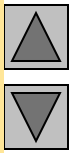
```
Enter seed: 67
        6       1       4       6       2
        1       6       1       6       4

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
```

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>
6
7  int rollDice( void );
8
9  int main()
10 {
11    int gameStatus, sum, myPoint;
12
13    srand( time( NULL ) );
14    sum = rollDice();             /* first roll of the dice */
15
16    switch ( sum ) {
17       case 7: case 11:           /* win on first roll */
18          gameStatus = 1;
19          break;
20       case 2: case 3: case 12:   /* lose on first roll */
21          gameStatus = 2;
22          break;
23       default:                   /* remember point */
24          gameStatus = 0;
25          myPoint = sum;
26          printf( "Point is %d\n", myPoint );
27          break;
28    }
29
30    while ( gameStatus == 0 ) {    /* keep rolling */
31       sum = rollDice();
32
```



Outline

1. `rollDice` prototype

1.1 Initialize variables

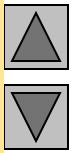
1.2 Seed `srand`

2. Define `switch` statement for win/loss/continue

2.1 Loop

```
33       if ( sum == myPoint )       /* win by making point */
34          gameStatus = 1;
35       else
36          if ( sum == 7 )          /* lose by rolling 7 */
37             gameStatus = 2;
38    }
39
40    if ( gameStatus == 1 )
41       printf( "Player wins\n" );
42    else
43       printf( "Player loses\n" );
44
45    return 0;
46 }
47
48 int rollDice( void )
49 {
50    int die1, die2, workSum;
51
52    die1 = 1 + ( rand() % 6 );
53    die2 = 1 + ( rand() % 6 );
54    workSum = die1 + die2;
55    printf( "Player rolled %d + %d = %d\n", die1, die2, workSum );
56    return workSum;
57 }
```

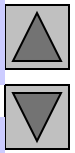
```
Player rolled 6 + 5 = 11
Player wins
```



Outline

2.2 Print win/loss

Program Output



Outline

Program Output

```
Player rolled 6 + 6 = 12
Player loses
```

```
Player rolled 4 + 6 = 10
Point is 10
Player rolled 2 + 4 = 6
Player rolled 6 + 5 = 11
Player rolled 3 + 3 = 6
Player rolled 6 + 4 = 10
Player wins
```

```
Player rolled 1 + 3 = 4
Point is 4
Player rolled 1 + 4 = 5
Player rolled 5 + 4 = 9
Player rolled 4 + 6 = 10
Player rolled 6 + 3 = 9
Player rolled 1 + 2 = 3
Player rolled 5 + 2 = 7
Player loses
```

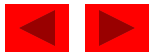
5.11 Storage Classes

- 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;
```

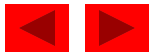
  - **`register`**: tries to put variable into high-speed registers
    - Can only be used for automatic variables

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



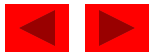
5.11 Storage Classes

- 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



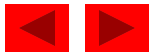
5.12 Scope Rules

- 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.)

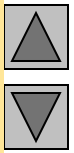


5.12 Scope Rules

- Block scope
  - Identifier declared inside a block
    - Block scope begins at declaration, 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



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

void a( void );   /* function prototype */
void b( void );   /* function prototype */
void c( void );   /* function prototype */

int x = 1;        /* global variable */

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;

      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 );

   a();         /* a has automatic local x */
   b();         /* b has static local x */
   c();         /* c uses global x */
   a();         /* a reinitializes automatic local x */
   b();         /* static local x retains its previous value */
   c();         /* global x also retains its value */
```



Outline

1. Function prototypes

1.1 Initialize global variable

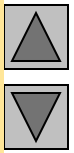
1.2 Initialize local variable

1.3 Initialize local variable in block

2. Call functions

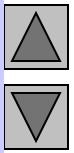
3. Output results

```
31
32    printf( "local x in main is %d\n", x );
33    return 0;
34 }
35
36 void a( void )
37 {
38    int x = 25;  /* initialized each time a is called */
39
40    printf( "\nlocal x in a is %d after entering a\n", x );
41    ++x;
42    printf( "local x in a is %d before exiting a\n", x );
43 }
44
45 void b( void )
46 {
47     static int x = 50;  /* static initialization only */
48                         /* first time b is called */
49     printf( "\nlocal static x is %d on entering b\n", x );
50     ++x;
51     printf( "local static x is %d on exiting b\n", x );
52 }
53
54 void c( void )
55 {
56    printf( "\nglobal x is %d on entering c\n", x );
57    x *= 10;
58    printf( "global x is %d on exiting c\n", x );
59 }
```



Outline

3.1 Function definitions



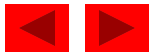
Outline

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
```

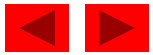
5.13 Recursion

- 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

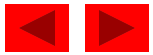
- Example: factorials
  - `5! = 5 * 4 * 3 * 2 * 1`
  - Notice that
    - `5! = 5 * 4!`
    - `4! = 4 * 3!` ...
  - Can compute factorials recursively
  - Solve base case (`1! = 0! = 1`) then plug in
    - `2! = 2 * 1! = 2 * 1 = 2;`
    - `3! = 3 * 2! = 3 * 2 = 6;`



5.14 Example Using Recursion: The Fibonacci Series

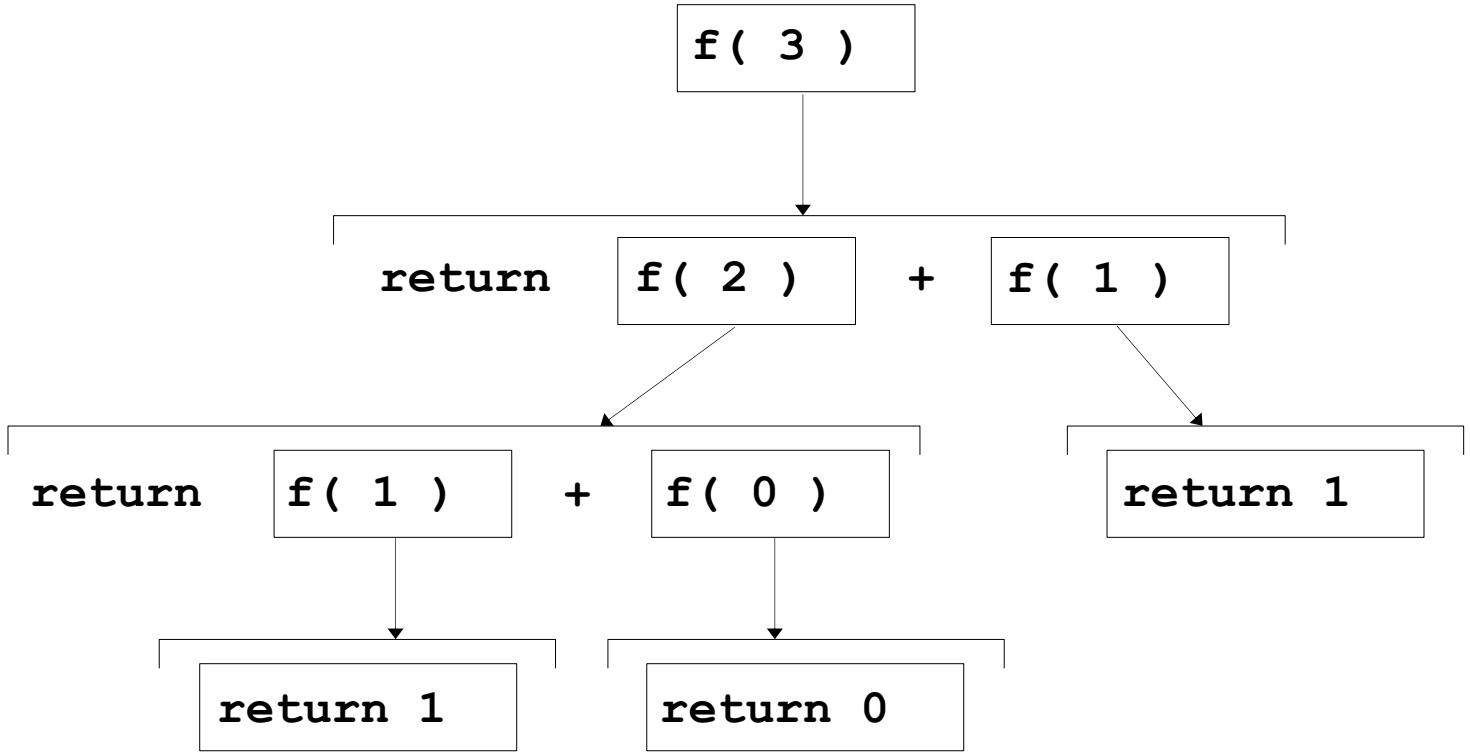
- Fibonacci series: 0, 1, 1, 2, 3, 5, 8...
  - Each number is the sum of the previous two
  - Can be solved recursively:
    - **fib(n) = fib(n - 1) + fib(n - 2)**
  - Code for the **fibaonacci** function

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



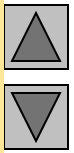
5.14 Example Using Recursion: The Fibonacci Series

- Set of recursive calls to function **`fibonacci`**



```c
/* Fig. 5.15: fig05_15.c
   Recursive fibonacci function */
#include <stdio.h>

long fibonacci( long );

int main()
{
   long result, number;

   printf( "Enter an integer: " );
   scanf( "%ld", &number );
   result = fibonacci( number );
   printf( "Fibonacci( %ld ) = %ld\n", number, result );
   return 0;
}

/* Recursive definition of function fibonacci */
long fibonacci( long n )
{
   if ( n == 0 || n == 1 )
      return n;
   else
      return fibonacci( n - 1 ) + fibonacci( n - 2 );
}
```

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

Enter an integer: 1
Fibonacci(1) = 1
```



Outline

1. Function prototype

1.1 Initialize variables

2. Input an integer

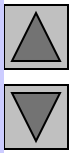
2.1 Call function `fibonacci`

2.2 Output results.

3. Define `fibonacci` recursively

Program Output

```
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

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)