Chapter 5
Functions

Associate Prof. Yuh-Shyan Chen
Dept. of Computer Science and Information Engineering
National Chung-Cheng University

Chapter 5 - Functions

Outline
5.1 Introduction
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 Example: A Game of Chance
5.11 Storage Classes
5.12 Scope Rules
5.13 Recursion
5.14 Example Using Recursion: The Fibonacci Series
5.15 Recursion vs. Iteration

5.1 Introduction
- Divide and conquer
  - Construct a program from smaller pieces or components
  - Each piece more manageable than the original program
- This chapter describes the features of the C language that facilitate the design, implementation, operation, and maintenance of large programs.

5.2 Program Modules in C
- Functions
  - Modules in C
    - Programs written by combining user-defined functions with library functions
    - C standard library ("pre-packaged" functions) has a wide variety of functions
      - Common mathematical calculations
      - String manipulations
      - Character manipulations
      - Input/output
    - Makes programmer's job easier - avoid reinventing the wheel
5.2 Program Modules in C (II)

- Function calls
  - Invoking functions
    - Provide function name and arguments (data)
    - Function performs operations or manipulations
    - Function returns results
  - 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
Example

```c
printf("%.2f", sqrt ( c1 + d * f) );
```

--

**Fig. 5.2**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqrt(x)</td>
<td>square root of x</td>
<td>sqrt(9.0) = 3.0</td>
</tr>
<tr>
<td>exp(x)</td>
<td>exponential function e&lt;sup&gt;x&lt;/sup&gt;</td>
<td>exp(1.0) = 2.71828</td>
</tr>
<tr>
<td>log(x)</td>
<td>natural logarithm of x (base e)</td>
<td>log(2.71828) = 1.0</td>
</tr>
<tr>
<td>log10(x)</td>
<td>logarithm of x (base 10)</td>
<td>log10(1.0) = 0.0</td>
</tr>
</tbody>
</table>

**5.4 Functions**

- Modularize a program
- All variables declared inside functions are local variables
  - Known only in function defined
- Parameters
  - Communicate information between functions
  - Local variables
  - Abstraction - hide internal details (library functions)
- Avoids code repetition
5.5 Function Definitions

Function definition format

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

- Function-name: any valid identifier
- Return-value-type: data type of the result (default `int`)
- `void` - function returns nothing

Parameter-list: comma separated list, declares parameters (default `int`)

Function definition format (II)

```
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)
- Function can not be defined inside another function
Returning control
- If nothing returned
  
  \textbf{return};
  
  - or, until reaches right brace
- If something returned
  \textbf{return expression};

\subsection*{5.6 Function Prototypes}

A function prototype tells the compiler
- The type of data returned by the function
- The number of parameters the function expects to receive
- The type of the parameters
- The order in which these parameters are expected

\section*{Function prototype}
- Function name
- Parameters - what the function takes in
- Return type - data type function returns (default int)

\textbf{Used to validate} functions
- Prototype only needed if \textbf{function definition} comes after use in program
  \begin{verbatim}
  int maximum(int, int, int);
  \end{verbatim}
  - Takes in 3 \texttt{int}s
  - Returns an \texttt{int}

5.6 Function Prototypes
Important feature of function prototypes is
- The coercion of arguments
  - For example,
    ```c
    printf("%.3f\n", sqrt(4));
    // All data type of math function is double
    ```
- Promotion rules and conversions
  - Converting to lower types can lead to errors

5.7 Header Files
- Header files
  - Each standard library has a corresponding header file containing the function prototypes for all the functions library functions
  - `<stdlib.h>`, `<math.h>`, etc
  - Load with `#include <filename>`
    ```c
    #include <math.h>
    ```

Custom header files (Programmer-defined header file)
- 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
    \[1 + \left( \text{rand()} \mod n \right)\]
  - `rand % n` returns a number between 0 and n-1
  - Add 1 to make random number between 1 and n
    `1 + ( \text{rand()} \mod 6)` //number between 1 and 6

5.9 Random Number Generation (II)

- **srand function**
  - `<stdlib.h>`
  - Takes an integer seed - jumps to location in "random" sequence
    `srand( seed );`
  - `srand( time( NULL ) );`
    // load `<time.h>`
  - `time( NULL )` - time program was compiled in seconds
    "Randomizes" the seed
Producing a value between \( a \) and \( b \)

\[ n = a + \text{rand}() \% b; \]

Where \( a \) is the shifting value, and \( b \) is the scaling factor

---

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
# Fig. 5.10: fig05_10.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int rollDice( void );

int main()
{
    int gameStatus, sum, myPoint;

    srand( time( NULL ) );
    sum = rollDice();            /* first roll of the dice */
    sum = rollDice();            /* first roll of the dice */

    switch ( sum ) {
        case 7: case 11:          /* win on first roll */
            gameStatus = 1;
            break;
        case 2: case 3: case 12:  /* lose on first roll */
            gameStatus = 2;
            break;
        default:                  /* remember point */
            gameStatus = 0;
            myPoint = sum;
            printf( "Point is %d\n", myPoint );
            break;
    }

    while ( gameStatus == 0 ) {    /* keep rolling */
        sum = rollDice();
        if ( sum == myPoint )       /* win by making point */
            gameStatus = 1;
        else
            if ( sum == 7 )          /* lose by rolling 7 */
                gameStatus = 2;
    }

    if ( gameStatus == 1 )
        printf( "Player wins\n" );
    else
        printf( "Player loses\n" );

    return 0;
}

int rollDice( void )
{
    int die1, die2, workSum;

    die1 = 1 + ( rand() % 6 );
    die2 = 1 + ( rand() % 6 );
    workSum = die1 + die2;
    printf( "Player rolled %d + %d = %d\n", die1, die2, workSum );
    return workSum;
}
```

## Program Output

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

Player rolled 6 + 6 = 12
Player loses

Player rolled 4 + 6 = 10
Point is 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 wins

Player rolled 1 + 2 = 3
Player rolled 5 + 2 = 7
Player loses

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

## 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** - what files an identifier is known (more in Chapter 14)
5.11 Storage Classes (II)

- **Static** storage
  - Variables exist for entire program execution
  - Default value of zero
  - (More detail in Chapter 14)
  - **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 (II)

- **File scope**
  - Identifier defined outside function, known in all functions
  - Global variables, function definitions, function prototypes

- **Function scope**
  - Can only be referenced inside a function body
  - Only labels (**start**, **case**, etc.)

- **Block scope**
  - Identifier declared inside a block
  - Block scope begins at declaration, ends at right brace
  - Variables, function parameters (local variables of function)
  - Outer blocks "hidden" from inner blocks if same variable name
Function prototype scope

- Identifiers in parameter list
- Names in function prototype optional, and can be used anywhere

```c
#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 );

  {
    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 */

  printf("local x in main is %d\n", x );
  return 0;
}

void a( void )
{
  int x = 25;  /* initialized each time a is called */

  printf( "local x in a is %d after entering a\n", x );
  ++x;
  printf( "local x in a is %d before exiting a\n", x );
}

void b( void )
{
  static int x = 50;  /* static initialization only */
  /* first time b is called */
  printf( "local static x is %d on entering b\n", x );
  ++x;
  printf( "local static x is %d on exiting b\n", x );
}

void c( void )
{
  printf("global x is %d on entering c\n", x );
  x *= 10;
  printf("global x is %d on exiting c\n", x );
}
```

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 x in b is 50 on entering b
local x in c is 1 on entering c
local x in c is 10 on entering c
local x in 25 before exiting a
local x in c is 100 on entering c
local x in c is 100 on exiting c
local x in main is 5
```
5.13 Recursion

- Recursive functions
  - Function that calls itself
  - Can only solve a base case
  - Divides up problem into
    - What it can do
    - What it cannot do - resembles original problem
      - Launches a new copy of itself (recursion step)

Eventually base case gets solved
- Gets plugged in, works its way up and solves whole problem

5.13 Recursion (II) 找出遞迴的規則性

- Example: factorial: \( n! = n \times (n-1)! \)
  - Notice that \( 5! = 5 \times 4! \)
  - \( 4! = 4 \times 3! \ldots \)

- Can compute factorials recursively
- Solve base case \((1! = 0! = 1)\) then plug in
  - \( 2! = 2 \times 1! = 2 \times 1 = 2 \)
  - \( 3! = 3 \times 2! = 3 \times 2 = 6 \)

Recursive evaluation of 5!

- Process of recursive calls
- Values returned from each recursive call:
  - Final value = 120
  - \( 5 = 5 \times 4 \) is returned
  - \( 4 = 4 \times 3 \) is returned
  - \( 3 = 3 \times 2 \) is returned
  - \( 2 = 2 \times 1 \) is returned
  - \( 1 \) is returned

Fig. 5.13 Recursive evaluation of 5!
Example

long factorial (long number)
{
    if (number <= 1) //base case
        return 1;
    else return (number * factorial (number - 1));
}

5.14 Example Using Recursion: The Fibonacci Series

Fibonacci series: 0, 1, 1, 2, 3, 5, 8...

Each number sum of the previous two
fib(n) = fib(n-1) + fib(n-2)
// recursive formula

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 (II)

f(3)
  f(2) + f(1)
    return 1

f(1) + f(0)
  return 0

long fibonacci( long )
{
    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

#include <stdio.h>

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
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)