Chapter 5
C Functions

C How to Program

5.1 Introduction

- In practice, most computer programs are very large.
- Functions allow you to **modularize** a program.
  - This technique is called divide and conquer.
  - Increase reusability
  - Avoid repeating code in a program
- Modules in C are called functions.

5.2 Program Modules in C

- C programs are typically written by combining new functions you write with *prepackaged* functions available in the C standard library.
- C Standard Library.
  - common mathematical calculations: pow
  - string manipulations
  - character manipulations
  - input/output: printf, scanf, getchar
  - many other useful operations
- Programmer-defined functions
  - Functions defined by you

5.3 Math Library Functions

- double sqrt(double)

```
printf( "%.2f", sqrt( 900.0 ) );
```

  - The `sqrt` function takes an argument of type `double` and returns a result of type `double`.
  - Include the header: **#include <math.h>**

- Function arguments may be constants, variables, or expressions.
  - `printf( "%.2f", sqrt( c1 + d * f ) );`

Function	Description	Example
`sqrt(x)`	square root of x	`sqrt(900.0)` is `30.0` `sqrt(9.0)` is `3.0`
`cbrt(x)`	cube root of x (C99 and C11 only)	`cbrt(27.0)` is `3.0` `cbrt(-8.0)` is `-2.0`
`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 as a floating-point number	`fabs(13.5)` is `13.5`
		`fabs(0.0)` is `0.0`
		`fabs(-13.5)` is `13.5`

`ceil(x)`	rounds x to the smallest integer not less than x	`ceil(9.2)` is `10.0` `ceil(-9.8)` is `-9.0`

Fig. 5.2 | Commonly used math library functions. (Part 1 of 2.)

Function	Description	Example
`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. (Part 2 of 2.)

5.4 Functions

- Functions allow you to modularize a program.
- All variables defined in function definitions are local variables—they can be accessed *only* in the function in which they're defined.
- Most functions have a list of parameters that provide the means for communicating information between functions.
- A function's parameters are also local variables of that function.

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

int square(int y); // function prototype

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

   puts("");
}

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

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

function prototype
宣告
注意結尾加分號

呼叫square function,
傳入x,
計算完回傳結果

輸出結果為整數

傳入參數為整數

定義回傳值

Fig. 5.3 | Creating and using a programmer-defined function.

5.5 Function Definitions (Cont.)

- The format of a function definition is

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

- The *function-name* is any valid identifier.
- *return-value-type*
  - the data type of the result returned to the caller.
  - The *return-value-type* `void` indicates that a function does not return a value.

5.5 Function Definitions (Cont.)

- *parameter-list*
  - a comma-separated list that specifies the parameters received by the function when it's called.
  - If a function does not receive any values, *parameter-list* is `void`.
  - A type must be listed explicitly for each parameter.

- A function cannot be defined inside another function.

5.5 Function Definitions (Cont.)

main's Return Type

- Notice that `main` has an `int` return type.
- The return value of `main` is used to indicate whether the program executed correctly.
- In earlier versions of C, we'd explicitly place

```
return 0;
```

- at the end of `main`—`0` indicates that a program ran successfully.
- The C standard indicates that main implicitly returns 0 if you omit the "return 0" statement—as we've done throughout this book.

5.5 Function Definitions (Cont.)

Function maximum

```
1  // Fig. 5.4: fig05_04.c
2  // Finding the maximum of three integers.
3  #include <stdio.h>
4
5  int maximum(int x, int y, int z); // function prototype
6
7  int main(void)
8  {
9     int number1; // first integer entered by the user
10    int number2; // second integer entered by the user
11    int number3; // third integer entered by the user
12
13    printf("%s", "Enter three integers: ");
14    scanf("%d%d%d", &number1, &number2, &number3);
15
16    // number1, number2 and number3 are arguments
17    // to the maximum function call
18    printf("Maximum is: %d\n", maximum(number1, number2, number3));
19 }
20
```

Prototype

呼叫

Function 定義須與function prototype 宣告一致

Define the function

Local variable

```
21  // Function maximum definition
22  // x, y and z are parameters
23  int maximum(int x, int y, int z)
24  {
25     int max = x; // assume x is largest
26
27     if (y > max) { // if y is larger than max,
28        max = y; // assign y to max
29     }
30
31     if (z > max) { // if z is larger than max,
32        max = z; // assign z to max
33     }
34
35     return max; // max is largest value
36  }
```

Fig. 5.4 | Finding the maximum of three integers. (Part 2 of 3.)

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

```
Enter three integers: 47 32 14
Maximum is: 47
```

```
Enter three integers: 35 8 79
Maximum is: 79
```

Fig. 5.4 | Finding the maximum of three integers. (Part 3 of 3.)

5.6 Function Prototypes: A Deeper Look

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

- The compiler uses function prototypes to validate function calls.

5.6 Function Prototypes: A Deeper Look (Cont.)

- The function prototype for `maximum` in Fig. 5.4 (line 5) is

```
// function prototype
int maximum( int x, int y, int z );
```

- It states that `maximum` takes three arguments of type `int` and returns a result of type `int`.
- Notice that the function prototype is the same as the first line of `maximum`'s function definition.

5.6 Function Prototypes: A Deeper Look (Cont.)

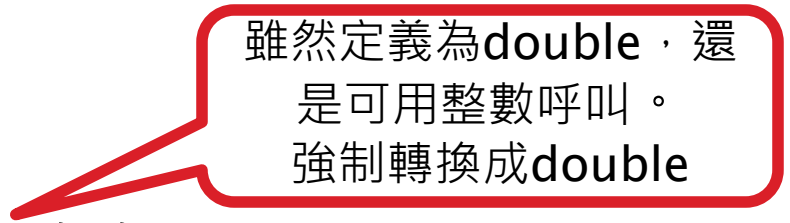
Argument Coercion and "Usual Arithmetic Conversion Rules"

- Another important feature of function prototypes is the coercion of arguments, i.e., the forcing of arguments to the appropriate type.

- For example

```
printf( "%.3f\n", sqrt( 4 ) );
```

correctly evaluates `sqrt( 4 )` and prints the value `2.000`.



5.6 Function Prototypes: A Deeper Look (Cont.)

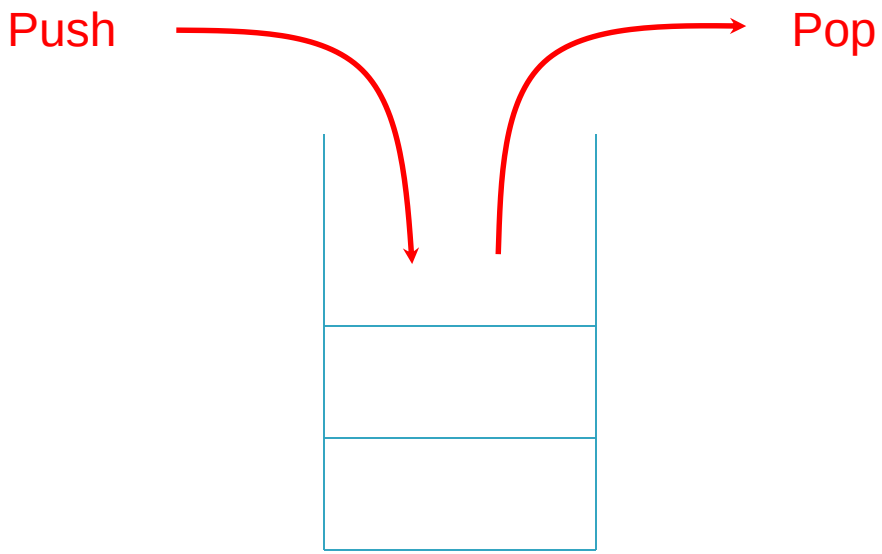
- In our `sqrt` example above, an `int` is automatically converted to a `double` without changing its value.
- However, a `double` converted to an `int` *truncates* the fractional part of the `double` value, thus changing the original value.
- Converting large integer types to small integer types (e.g., `long` to `short`) may also result in changed values.

Data type	printf conversion specification	scanf conversion specification
Floating-point types		
`long double`	`%Lf`	`%Lf`
`double`	`%f`	`%lf`
`float`	`%f`	`%f`
Integer types		
`unsigned long long int`	`%llu`	`%llu`
`long long int`	`%lld`	`%lld`
`unsigned long int`	`%lu`	`%lu`
`long int`	`%ld`	`%ld`
`unsigned int`	`%u`	`%u`
`int`	`%d`	`%d`
`unsigned short`	`%hu`	`%hu`
`short`	`%hd`	`%hd`
`char`	`%c`	`%c`

Fig. 5.5 | Arithmetic data types and their conversion specifications.

5.7 Function Call Stack and Stack Frames

- To understand how C performs function calls, we first need to consider a data structure (i.e., collection of related data items) known as a stack.



- Stacks are known as last-in, first-out (LIFO) data structures

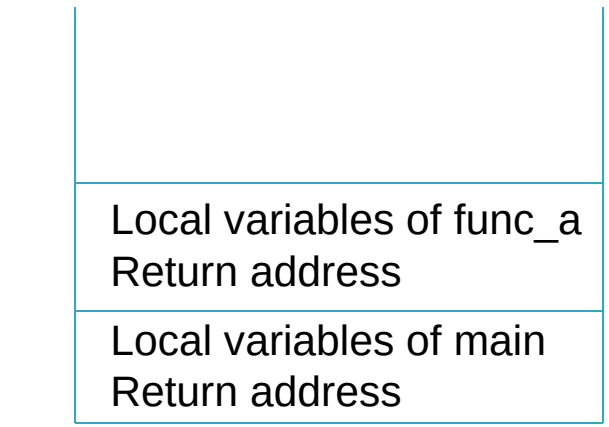
5.7 Function Call Stack and Stack Frames (Cont.)

- When a program calls a function, the called function must know how to return to its caller. The return address of the calling function is pushed onto the program execution stack

```
int main()
{
   …
   func_a();
   …
}
```

```
void  func_a()
{
   …
   func_b();
   …
}
```

```
void  func_b()
{
   …
}
```



5.8 Headers

- Each standard library has a corresponding header containing
  - the function prototypes for all the functions in that library
  - definitions of various data types and constants needed by those functions.

- You can create custom headers.
  - `#include "square.h“`

Header	Explanation
`<assert.h>`	Contains 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's running. The notion of locale enables the computer system to handle different conventions for expressing data such as dates, times, currency 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.

Fig. 5.10 | Some of the standard library headers. (Part 1 of 2.)

Header	Explanation
`<stdarg.h>`	Defines macros for dealing with a list of arguments to a function whose number and types are unknown.
`<stddef.h>`	Contains common type definitions used by C for performing 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. 5.10 | Some of the standard library headers. (Part 2 of 2.)

5.9 Passing Arguments By Value and By Reference

- In many programming languages, there are two ways to pass arguments—pass-by-value and pass-by-reference.
- pass-by-value: When arguments are *passed by value*, a *copy* of the argument's value is made and passed to the called function.
  - Changes to the copy do *not* affect an original variable's value in the caller.
- pass-by-reference: When an argument is passed by reference, the caller allows the called function to modify the original variable's value.

- In C, all arguments are passed by value.

5.10 Random Number Generation

- C standard library function `rand` from the `<stdlib.h>` header.

- Consider the following statement:

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

- The `rand` function generates an integer between 0 and `RAND_MAX` (a symbolic constant defined in the `<stdlib.h>` header).

5.10 Random Number Generation (Cont.)

- A dice-rolling program that simulates a six-sided die would require random integers from 1 to 6.

Rolling a Six-Sided Die

- Produce integers in the range 0 to 5.

```
rand() % 6
```

```c
// Fig. 5.11: fig05_11.c
// Shifted, scaled random integers produced by 1 + rand() % 6.
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   // loop 20 times
   for (unsigned int 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) {
         puts("");
      }
   }
}
```

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

Fig. 5.11 | Shifted, scaled random integers produced by `1 + rand() % 6`.

5.10 Random Number Generation (Cont.)

Randomizing the Random Number Generator

- Run the program again
  - The same results will be generated.
  - Function `rand` actually generates pseudorandom numbers.

- Function `srand` takes an `unsigned` integer argument and seeds function `rand` to produce a different sequence of random numbers for each execution of the program.
  - `srand` is defined in `<stdlib.h>`.

```c
// Fig. 5.13: fig05_13.c
// Randomizing the die-rolling program.
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   unsigned int seed; // number used to seed the random number generator

   printf("%s", "Enter seed: ");
   scanf("%u", &seed); // note %u for unsigned int

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

```

Fig. 5.13 | Randomizing the die-rolling program. (Part 1 of 3.)

```
15      // loop 10 times
16      for (unsigned int i = 1; i <= 10; ++i) {
17
18         // pick a random number from 1 to 6 and output it
19         printf("%10d", 1 + (rand() % 6));
20
21         // if counter is divisible by 5, begin a new line of output
22         if (i % 5 == 0) {
23            puts("");
24         }
25      }
26   }
```

Fig. 5.13 | Randomizing the die-rolling program. (Part 2 of 3.)

```
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
```

Fig. 5.13 | Randomizing the die-rolling program. (Part 3 of 3.)

5.10 Random Number Generation (Cont.)

- To randomize without entering a seed each time, use a statement like

```
srand( time( NULL ) );
```

- This causes the computer to read its clock to obtain the value for the seed automatically.

- Function `time` returns the number of seconds that have passed since midnight on January 1, 1970.
  - The function prototype for `time` is in `<time.h>`.

5.11 Example: A Game of Chance

- One of the most popular games of chance is a dice game known as “craps.” The rules of the game are simple.
  - A player rolls two dice. Each die has six faces. These faces contain 1, 2, 3, 4, 5, and 6 spots. After the dice have come to rest, the sum of the spots on the two upward faces is calculated. If the sum is 7 or 11 on the first throw, the player wins. If the sum is 2, 3, or 12 on the first throw (called “craps”), the player loses (i.e., the “house” wins). If the sum is 4, 5, 6, 8, 9, or 10 on the first throw, then that sum becomes the player’s “point.” To win, you must continue rolling the dice until you “make your point.” The player loses by rolling a 7 before making the point.

```
1  // Fig. 5.14: fig05_14.c
2  // Simulating the game of 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 int main(void)
13 {
14    // randomize random number generator using current time
15    srand(time(NULL));
16
17    int myPoint; // player must make this point to win
18    enum Status gameStatus; // can contain CONTINUE, WON, or LOST
19    int sum = rollDice(); // first roll of the dice
20
```

Fig. 5.14 | Simulating the game of craps. (Part 1 of 4.)

```
21      // determine game status based on sum of dice
22      switch(sum) {
23
24         // win on first roll
25         case 7: // 7 is a winner
26         case 11: // 11 is a winner
27            gameStatus = WON;
28            break;
29
30         // lose on first roll
31         case 2: // 2 is a loser
32         case 3: // 3 is a loser
33         case 12: // 12 is a loser
34            gameStatus = LOST;
35            break;
36
37         // remember point
38         default:
39            gameStatus = CONTINUE; // player should keep rolling
40            myPoint = sum; // remember the point
41            printf("Point is %d\n", myPoint);
42            break; // optional
43      }
44
```

Fig. 5.14 | Simulating the game of craps. (Part 2 of 4.)

```
45      // while game not complete
46      while (CONTINUE == gameStatus) { // player should keep rolling
47         sum = rollDice(); // roll dice again
48
49         // determine game status
50         if (sum == myPoint) { // win by making point
51            gameStatus = WON;
52         }
53         else {
54            if (7 == sum) { // lose by rolling 7
55               gameStatus = LOST;
56            }
57         }
58      }
59
60      // display won or lost message
61      if (WON == gameStatus) { // did player win?
62         puts("Player wins");
63      }
64      else { // player lost
65         puts("Player loses");
66      }
67   }
68
```

Fig. 5.14 | Simulating the game of craps. (Part 3 of 4.)

```
69  // roll dice, calculate sum and display results
70  int rollDice(void)
71  {
72     int die1 = 1 + (rand() % 6); // pick random die1 value
73     int die2 = 1 + (rand() % 6); // pick random die2 value
74
75     // display results of this roll
76     printf("Player rolled %d + %d = %d\n", die1, die2, die1 + die2);
77     return die1 + die2; // return sum of dice
78  }
```

Fig. 5.14 | Simulating the game of craps. (Part 4 of 4.)

Player wins on the first roll

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

Player wins on a subsequent roll

```
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
```

Fig. 5.15 | Sample runs for the game of craps. (Part 1 of 2.)

Player loses on the first roll

```
Player rolled 1 + 1 = 2
Player loses
```

Player loses on a subsequent roll

```
Player rolled 6 + 4 = 10
Point is 10
Player rolled 3 + 4 = 7
Player loses
```

Fig. 5.15 | Sample runs for the game of craps. (Part 2 of 2.)

5.11 Example: A Game of Chance (Conf.)

- Enumerate

  enum Status {CONTINUE, WON, LOSE}

  - `enum`: a set of integer constants represented by identifiers.

- Values in an `enum` start with `0` and are incremented by `1`.

- The identifiers in an enumeration must be **unique**

5.12 Storage Classes

- C provides the storage class specifiers: `auto`, `register`, `extern` and `static`.

- An identifier's storage class determines its storage duration, scope and linkage.
  - storage duration is the period during which the identifier exists in memory
  - scope is where the identifier can be referenced in a program
  - linkage determines for a multiple-source-file program whether the identifier is known only in the current source file or in any source file with proper declarations.

5.12 Storage Classes (Cont.)

- The storage-class specifiers can be split into automatic storage duration and static storage duration.
- For example

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

  - Variables with automatic storage duration are created when the block in which they're defined is entered; they exist while the block is active, and they're destroyed when the block is exited.

- Local variables have automatic storage duration by *default*, so keyword `auto` is rarely used.

5.12 Storage Classes (Cont.)

- The following declaration suggests that the integer variable `counter` be placed in one of the computer's registers and initialized to 1:
  - `register int counter = 1;`

- If no register is available, compiler may ignore `register` declarations.

5.12 Storage Classes (Cont.)

Static Storage Class

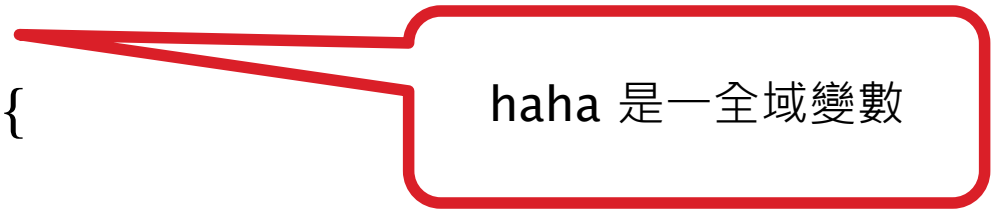
- `extern` and `static` are used in the declarations of identifiers for variables and functions of static storage duration.
- Identifiers of static storage duration exist from the time at which the program begins execution until the program terminates.
  - storage is allocated and initialized *only once*, *before* the program begins execution.
- For functions, the name of the function exists when the program begins execution.

5.12 Storage Classes (Cont.)

- There are several types of identifiers with static storage duration:
  - *external identifiers* (such as global variables and function names)
  - local variables declared with the storage-class specifier `static`.

- Global variables and function names are of storage class `extern` by default.

```
1. #include <stdio.h>
2. int   haha;
3. int main () {
4.       …
5. }
```



5.12 Storage Classes (Cont.)

```
1. void func(){
2.    static int count = 1;
3.    printf("count=%d\n");
4.    count++;
5. }

6.  int main(){
7.     func();
8.     func();
9.  }
```

5.13 Scope Rules

- The scope of an identifier is the portion of the program in which the identifier can be referenced.

- For example, when we define a local variable in a block, it can be referenced only following its definition in that block or in blocks nested within that block.

```c
// Fig. 5.16: fig05_16.c
// Scoping.
#include <stdio.h>

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

int x = 1; // global variable

int main(void)
{
   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);

```

Fig. 5.16 | Scoping. (Part 1 of 4.)

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

Fig. 5.16 | Scoping. (Part 2 of 4.)

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

Fig. 5.16 | Scoping. (Part 3 of 4.)

```
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 useLocal is 25 after entering useLocal
local x in useLocal is 26 before exiting useLocal

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

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

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

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

global x is 10 on entering useGlobal
global x is 100 on exiting useGlobal

local x in main is 5
```

Fig. 5.16 | Scoping. (Part 4 of 4.)

5.14 Recursion

- A recursive function is a function that calls itself either directly or indirectly through another function.

- A *recursive* definition of the factorial function is arrived at by observing the following relationship:

  $n! = n \cdot (n - 1)!$

- For example, 5! is clearly equal to 5 * 4! as is shown by the following:

  $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$

  $5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$

  $5! = 5 \cdot (4!)$

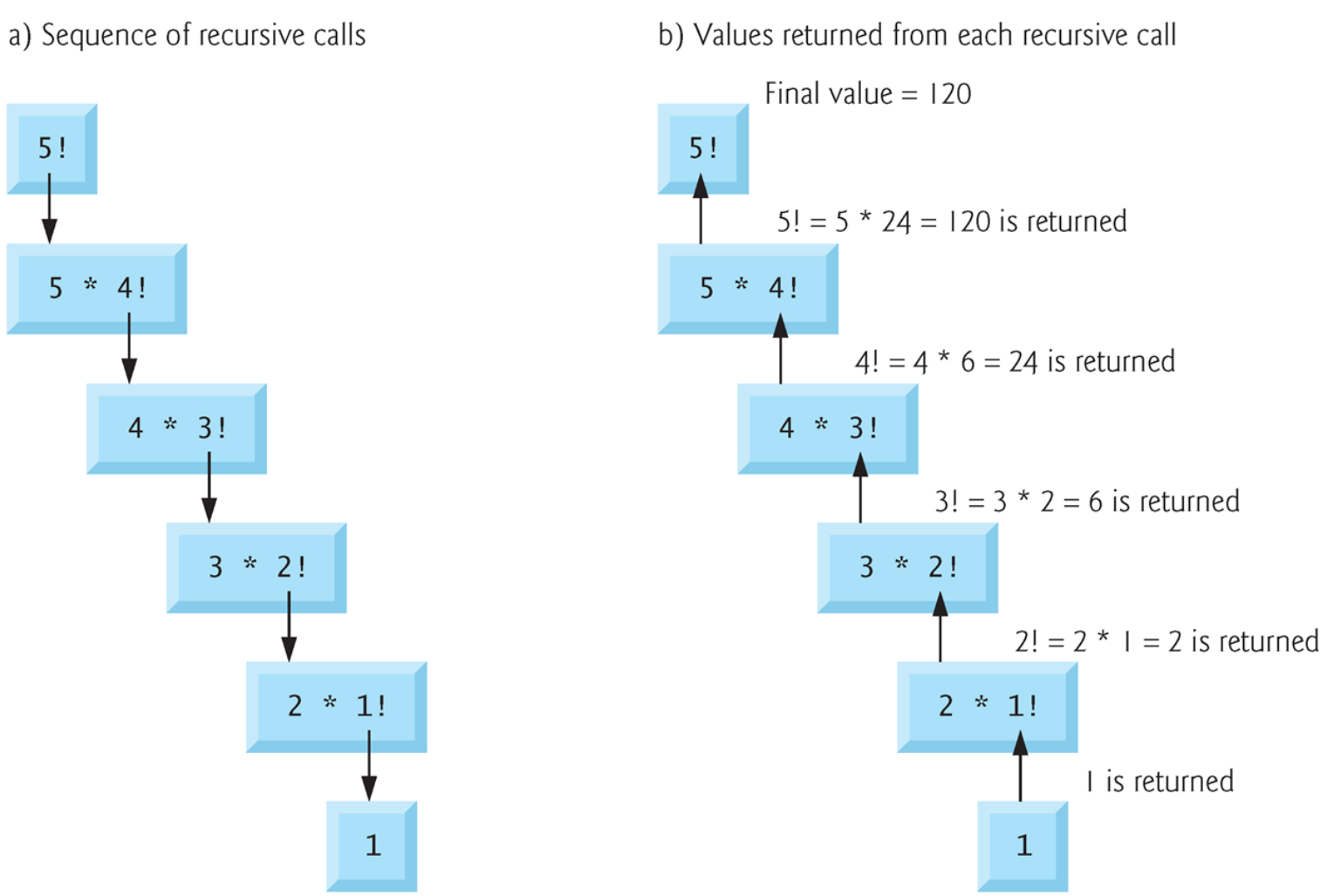


Fig. 5.17 | Recursive evaluation of 5!.

```c
// Fig. 5.18: fig05_18.c
// Recursive factorial function.
#include <stdio.h>

unsigned long long int factorial(unsigned int number);

int main(void)
{
   // during each iteration, calculate
   // factorial(i) and display result
   for (unsigned int i = 0; i <= 21; ++i) {
      printf("%u! = %llu\n", i, factorial(i));
   }
}

```

Fig. 5.18 | Recursive factorial function. (Part 1 of 3.)

```
16  // recursive definition of function factorial
17  unsigned long long int factorial(unsigned int number)
18  {
19     // base case
20     if (number <= 1) {
21        return 1;
22     }
23     else { // recursive step
24        return (number * factorial(number - 1));
25     }
26  }
```

Fig. 5.18 | Recursive factorial function. (Part 2 of 3.)

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
11! = 39916800
12! = 479001600
13! = 6227020800
14! = 87178291200
15! = 1307674368000
16! = 20922789888000
17! = 355687428096000
18! = 6402373705728000
19! = 121645100408832000
20! = 2432902008176640000
21! = 14197454024290336768
```

Fig. 5.18 | Recursive factorial function. (Part 3 of 3.)

5.15 Example Using Recursion: Fibonacci Series

- The Fibonacci series
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, …
- begins with 0 and 1 and has the property that each subsequent Fibonacci number is the sum of the previous two Fibonacci numbers.

- The Fibonacci series may be defined recursively as follows:

```
fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2)
```

```c
// Fig. 5.19: fig05_19.c
// Recursive fibonacci function
#include <stdio.h>

unsigned long long int fibonacci(unsigned int n); // function prototype

int main(void)
{
   unsigned int number; // number input by user

   // obtain integer from user
   printf("%s", "Enter an integer: ");
   scanf("%u", &number);

   // calculate fibonacci value for number input by user
   unsigned long long int result = fibonacci(number);

   // display result
   printf("Fibonacci(%u) = %llu\n", number, result);
}
```

Fig. 5.19 | Recursive fibonacci function. (Part 1 of 3.)

```
22  // Recursive definition of function fibonacci
23  unsigned long long int fibonacci(unsigned int n)
24  {
25     // base case
26     if (0 == n || 1 == n) {
27        return n;
28     }
29     else { // recursive step
30        return fibonacci(n - 1) + fibonacci(n - 2);
31     }
32  }
```

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

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

```
Enter an integer: 2
Fibonacci(2) = 1
```

Fig. 5.19 | Recursive fibonacci function. (Part 2 of 3.)

```
Enter an integer: 3
Fibonacci(3) = 2
```

```
Enter an integer: 10
Fibonacci(10) = 55
```

```
Enter an integer: 20
Fibonacci(20) = 6765
```

```
Enter an integer: 30
Fibonacci(30) = 832040
```

```
Enter an integer: 40
Fibonacci(40) = 102334155
```

Fig. 5.19 | Recursive fibonacci function. (Part 3 of 3.)

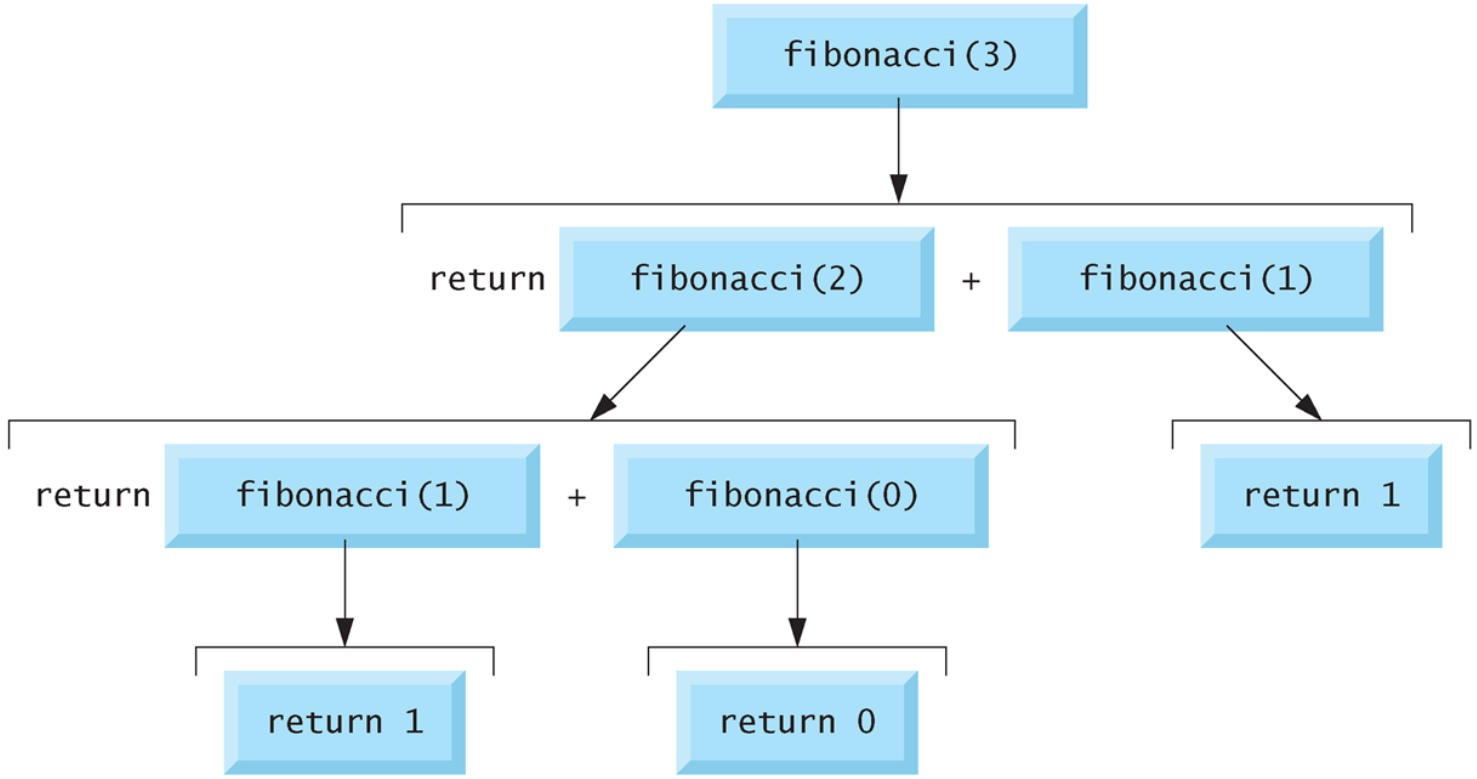


Fig. 5.20 | Set of recursive calls for `fibonacci(3)`.

5.15 Example Using Recursion: Fibonacci Series (Cont.)

- For optimization reasons, C does not specify the order in which the operands of most operators (including +) are to be evaluated.
- Therefore, you should make no assumption about the order in which these calls will execute.
- The calls could in fact execute `fibonacci(2)` first and then `fibonacci(1)`, or the calls could execute in the reverse order, `fibonacci(1)` then `fibonacci(2)`.
- In this program and in most other programs, the final result would be the same.

5.15 Example Using Recursion: Fibonacci Series (Cont.)

- But in some programs the evaluation of an operand may have side effects that could affect the final result of the expression.
- C specifies the order of evaluation of the operands of only four operators—namely `&&`, `||`, the comma (`,`) operator and `?:`.
- The first three of these are binary operators whose operands are guaranteed to be evaluated left to right.

5.15 Example Using Recursion: Fibonacci Series (Cont.)

Exponential Complexity

- Each level of recursion in the `fibonacci` function has a doubling effect on the number of calls

- Calculating the 20^{th} Fibonacci number would require on the order of 2^{20} or about a million calls
- Calculating the 30^{th} Fibonacci number would require on the order of 2^{30} or about a billion calls.

- Complexity => Algorithm
  - Fibonacci number is on the order of 2^n.

5.16 Recursion vs. Iteration

- Both iteration and recursion are based on a control structure: Iteration uses a repetition structure; recursion uses a *selection structure*.

- Both iteration and recursion involve repetition: Iteration explicitly uses a repetition structure; recursion achieves repetition through *repeated function calls*.

- Recursion has many negatives.
  - It *repeatedly* invokes the mechanism, and consequently the *overhead, of function calls*.
  - This can be expensive in both processor time and memory space.

5.16 Recursion vs. Iteration (Cont.)

- Each recursive call causes *another copy* of the function (actually only the function's variables) to be created; this can consume *considerable memory*.
- Iteration normally occurs within a function, so the overhead of repeated function calls and extra memory assignment is omitted.
- So why choose recursion?

Recursion examples and exercises	
Chapter 5	*Chapter 7*
Factorial function	Maze traversal
Fibonacci function	*Chapter 8*
Greatest common divisor	Printing a string input at the keyboard backward
Multiply two integers	
Raising an integer to an integer power	*Chapter 12*
Towers of Hanoi	Search a linked list
Recursive `main`	Print a linked list backward
Visualizing recursion	Binary tree insert
Chapter 6	Preorder traversal of a binary tree
Sum the elements of an array	Inorder traversal of a binary tree
Print an array	Postorder traversal of a binary tree
Print an array backward	Printing trees
Print a string backward	*Appendix D*
Check whether a string is a palindrome	Selection sort
Minimum value in an array	Quicksort
Linear search	*Appendix E*
Binary search	Fibonacci function
Eight Queens	

Fig. 5.21 | Recursion examples and exercises in the text.