

Chapter 8 - Functions

Outline

1. Introduction
2. Program Modules in C
3. Math Library Functions
4. Functions
5. Header Files
6. Calling Functions: Call by Value and Call by Reference
7. Recursion
8. Recursion vs. Iteration



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.



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



5.3 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



Math Library Functions

Function	Description	Example
<code>sqrt(x)</code>	square root of x	<code>sqrt(900.0)</code> is 30.0 <code>sqrt(9.0)</code> is 3.0
<code>exp(x)</code>	exponential function e^x	<code>exp(1.0)</code> is 2.718282 <code>exp(2.0)</code> is 7.389056
<code>log(x)</code>	natural logarithm of x (base e)	<code>log(2.718282)</code> is 1.0 <code>log(7.389056)</code> is 2.0
<code>log10(x)</code>	logarithm of x (base 10)	<code>log10(1.0)</code> is 0.0 <code>log10(10.0)</code> is 1.0 <code>log10(100.0)</code> is 2.0
<code>fabs(x)</code>	absolute value of x	<code>fabs(5.0)</code> is 5.0 <code>fabs(0.0)</code> is 0.0 <code>fabs(-5.0)</code> is 5.0
<code>ceil(x)</code>	rounds x to the smallest integer not less than x	<code>ceil(9.2)</code> is 10.0 <code>ceil(-9.8)</code> is -9.0
<code>floor(x)</code>	rounds x to the largest integer not greater than x	<code>floor(9.2)</code> is 9.0 <code>floor(-9.8)</code> is -10.0
<code>pow(x, y)</code>	x raised to power y (x^y)	<code>pow(2, 7)</code> is 128.0 <code>pow(9, .5)</code> is 3.0
<code>fmod(x, y)</code>	remainder of x/y as a floating point number	<code>fmod(13.657, 2.333)</code> is 1.992
<code>sin(x)</code>	trigonometric sine of x (x in radians)	<code>sin(0.0)</code> is 0.0
<code>cos(x)</code>	trigonometric cosine of x (x in radians)	<code>cos(0.0)</code> is 1.0
<code>tan(x)</code>	trigonometric tangent of x (x in radians)	<code>tan(0.0)</code> is 0.0
Fig. 5.2 Commonly used math library functions.		



Function Definitions

- Function definition format

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



```
1  /* Fig. 5.3: fig05_03.c
2     Creating and using a programmer-defined function */
3  #include <stdio.h>
4
5  int square( int y ); /* function prototype */
6
7  /* function main begins program execution */
8  int main()
9  {
10     int x; /* counter */
11
12     /* loop 10 times and calculate and output square of x each time */
13     for ( x = 1; x <= 10; x++ ) {
14         printf( "%d ", square( x ) ); /* function call */
15     } /* end for */
16
17     printf( "\n" );
18
19     return 0; /* indicates successful termination */
20
21 } /* end main */
22
```



Outline

```
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



Outline

```
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  /* function main begins program execution */
8  int main()
9  {
10     int number1; /* first integer */
11     int number2; /* second integer */
12     int number3; /* third integer */
13
14     printf( "Enter three integers: " );
15     scanf( "%d%d%d", &number1, &number2, &number3 );
16
17     /* number1, number2 and number3 are arguments
18        to the maximum function call */
19     printf( "Maximum is: %d\n", maximum( number1, number2, number3 ) );
20
21     return 0; /* indicates successful termination */
22
23 } /* end main */
24
```



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 */
```

```
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
```

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.



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



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



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

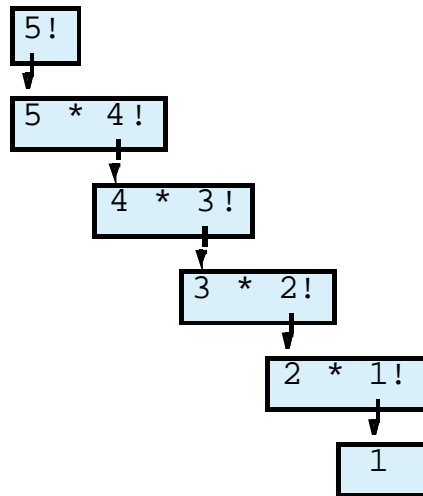


Recursion

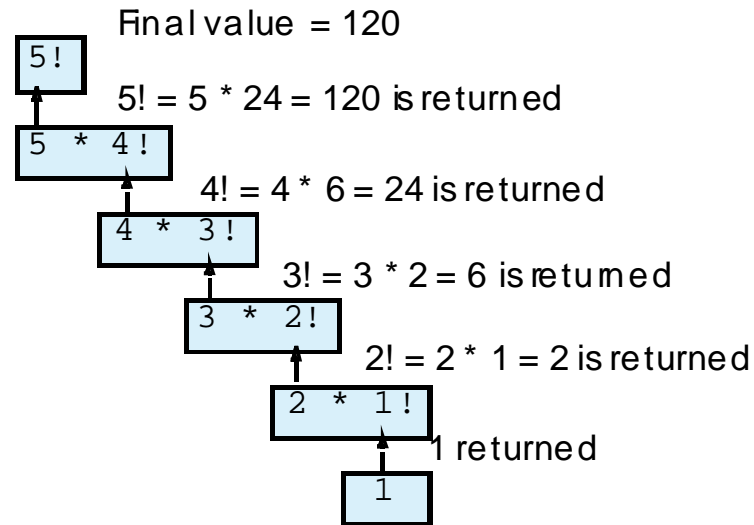
- Example: factorials
 - $5! = 5 * 4 * 3 * 2 * 1$
 - Notice that
 - $5! = 5 * 4!$
 - $4! = 4 * 3! \dots$
 - Can compute factorials recursively
 - Solve base case ($1! = 0! = 1$) then plug in
 - $2! = 2 * 1! = 2 * 1 = 2;$
 - $3! = 3 * 2! = 3 * 2 = 6;$



5.13 Recursion



(a) Sequence of recursive calls



(b) Values returned from each recursive call.





Outline

```
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
```

```
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
```



Outline

Example Using Recursion: The Fibonacci Series

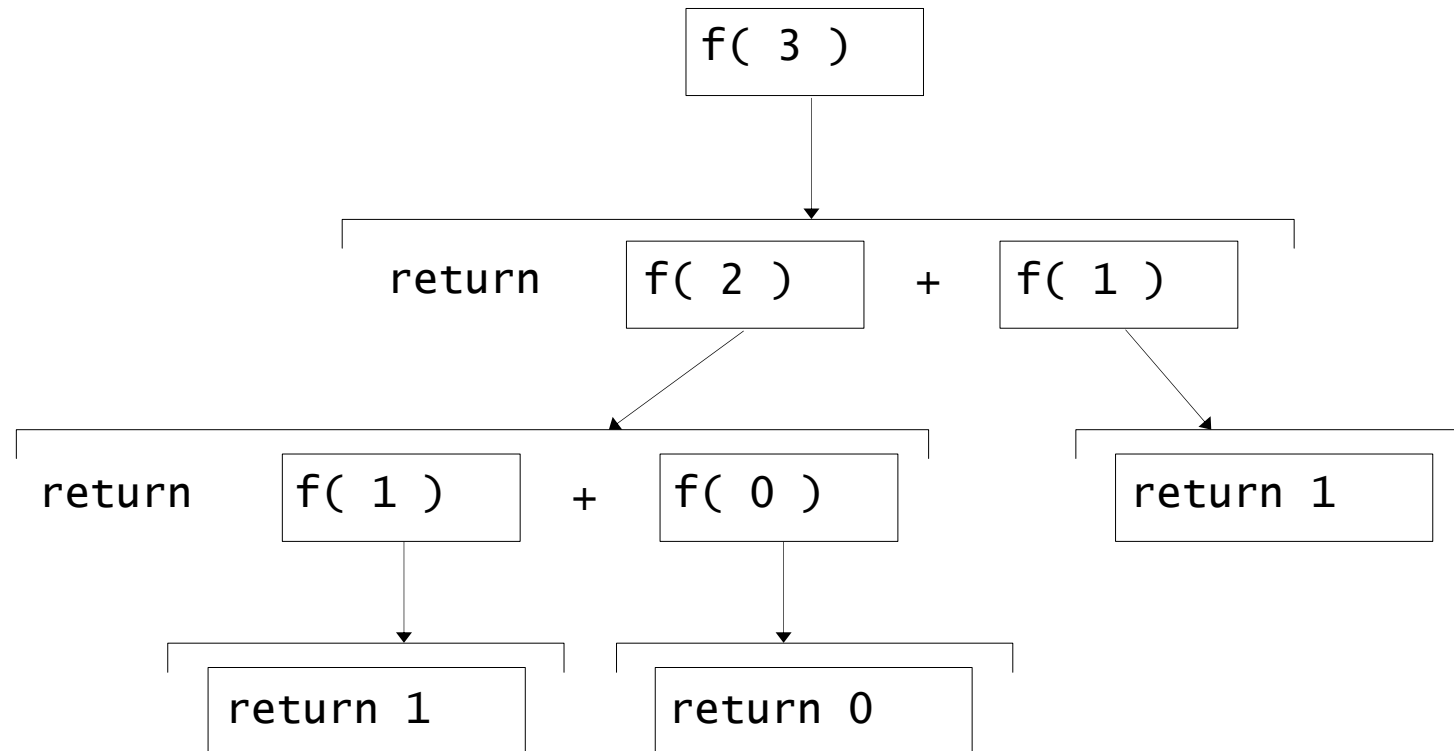
- Fibonacci series: 0, 1, 1, 2, 3, 5, 8...
 - Each number is the sum of the previous two
 - Can be solved recursively:
 - $\text{fib}(n) = \text{fib}(n - 1) + \text{fib}(n - 2)$
 - Code for the fibonacci function

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



Example Using Recursion: The Fibonacci Series

- Set of recursive calls to function fibonacci





Outline

```
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 */
```

```
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
```



Outline



Outline

Program Output (continued)

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

```
Enter an integer: 6  
Fibonacci( 6 ) = 8
```

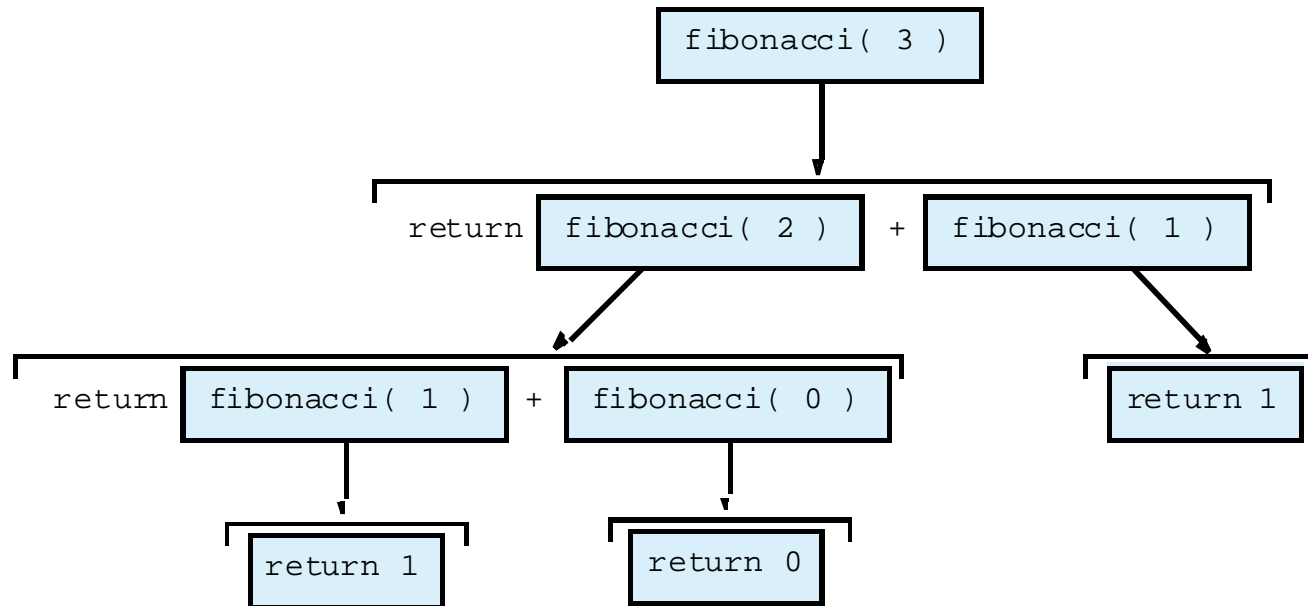
```
Enter an integer: 10  
Fibonacci( 10 ) = 55
```

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

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

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

Example Using Recursion: The Fibonacci Series



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)

