SINGAPORE INSTITUTE OF TECHNOLOGY

ENG1008 Programming

- C Functions

Dr Kwee Hiong Lee

kweehiong.lee@singaporetech.edu.sg

Objectives

- Construct programs *modularly* from small pieces called *functions*
- Use *common math functions* in the C standard library as well as creating new *(user-defined) functions*
- Use the mechanisms that *pass information between functions*
- Understand how *function call/return* is supported by the function call *stack*
- Write and use *functions* that *call themselves*

Introduction

- Computer programs that solve real-world problems tend to be *larger* and more *complex* than programs presented so far
- The best way to develop and maintain a large program is to build it from *smaller pieces* or modules
- This chapter describes some important features of the C language that supports the design, implementation, operation and maintenance of large programs
- Modules in C are called functions
  - C programs can be written by combining *C standard library* prepackaged functions with new *user-defined/user-created* functions

❖ C Standard Library Functions

C Standard Library Functions

SINGAPORE INSTITUTE OF TECHNOLOGY

- **C standard library functions**
  - should be used instead of developing new functions
  - can *reduce* program *development time*
  - will make the program more *portable*
  - like *printf*, *scanf*, and *pow* are standard library functions
- The **C standard library** provides a rich collection of functions that can perform
  - *common mathematical* calculations
  - *string manipulations*
  - *character manipulations*
  - *input and output (I/O)* and other functions

Printf and Scanf

SINGAPORE INSTITUTE OF TECHNOLOGY

Data type	printf conversion specification	scanf conversion specification
Floating-point types		
long double	%Lf	%Lf
double	%f	%lf
float	%f	%f
Data type	**printf conversion specification**	**scanf conversion specification**
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. (Part 2 of 2.)

Math Library Functions

SINGAPORE INSTITUTE OF TECHNOLOGY

- To perform certain common *mathematical calculations* instead of implementing it using your own code
- To include the math header #include <math.h>
- Functions are normally used by writing the name of the function followed by a *left parenthesis*, the argument (or comma-separated list of arguments) of the function followed by the *right parenthesis*
  - For example, to calculate and print the **square root** of 900.0
    - **printf("%.2f", sqrt(900.0));**
    - When this is executed, the math library function **sqrt** is called and will evaluate the square root of the number given in the parentheses (i.e. 900.0 is the argument)
    - The result of 30.00 will be displayed
  - The **sqrt** function takes an **argument** of type **double** and **returns** a **result** of type **double**

Math Library Functions

SINGAPORE INSTITUTE OF TECHNOLOGY

- Function arguments may be constants, variables or expressions
- Commonly used math library functions

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.)

Math Library Functions

SINGAPORE INSTITUTE OF TECHNOLOGY

- Commonly used math library functions

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.)

❖ User-defined Functions

User-defined Functions

SINGAPORE INSTITUTE OF TECHNOLOGY

- A function helps to *modularise* a program (divide-and-conquer)
- Each *function* should be limited to perform a *specific well-defined task* and the *function name* should reflect that task
  - it is advisable to break a *complex* function into *several smaller* functions (this process is sometimes called decomposition)
- Motivation of using functions
  - makes program development more manageable
  - *software reusability* - using existing functions as building block to implement new functionality
  - *Object-oriented programming*
  - avoids repeating code

User-defined Functions

SINGAPORE INSTITUTE OF TECHNOLOGY

- **Writing your own functions**
  - user-defined functions
  - functions are invoked by a function call, which specifies the *function name* and provides the *parameters* (i.e. *arguments*) that is used by the *called function* to perform specific tasks
  - E.g. a function that needs to display information onto the screen will call the printf function to perform that task. The printf function will display the information and revert back (i.e. return) to the *calling function* when its task is completed
- All *variables* defined in the function are local variables – they can only be accessed within the function in which they are defined
- Most functions have a set of parameters / arguments that provide the mechanism for *communicating information* between functions
- A function's *parameters* are also *local variables* of that function

User-defined Functions

- The **Main** function
  - each program shown so far has a function called **main**
  - is itself a function
  - can be implemented as a group of calls to functions (standard library or user-defined) that perform the bulk of the work (of the program)
- Let's visit a program that uses a function *square* to calculate and print the squares of the integers from 1 to 10 (refer Fig 5.3)
- Function square is invoked / called in main within the printf statement (line 14) - printf("%d ", square(x)); // function call
  - Function square receives a *copy* of the value of x in the parameter y (line 21)
  - Then function square calculates y*y
  - The result y*y is *passed back / returned* to function printf in main (where square was *invoked*) and printf displays the result
  - This is repeated 10 times using the for statement

User-defined Functions

SINGAPORE INSTITUTE OF TECHNOLOGY

```
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( void )
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    puts( "" );
18 } // end main
19
20 // square function definition returns the square of its parameter
21 int square( int y ) // y is a copy of the argument to the function
22 {
23    return y * y; // returns the square of y as an int
24 } // end function square
```

Function prototype

Function call

Function header

Function definition

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

Fig. 5.3 | Creating and using a programmer-defined function. (Part 2 of 2.)

Functions Definitions

- Line 5 is the function prototype for the function square
  - *int square (int y); // function prototype*
  - the (int y) *in parentheses* informs the compiler that the function square expects to receive an integer value from the *calling function*
  - The int *preceding the function name square* informs the compiler that square returns an integer result to the caller
  - The compiler refers to the function prototype to check that any calls to square contains the *correct return type*, *correct number of arguments and type*, and the *arguments* are in the *correct order*
- The definition of function square (line 21)
  - Square expects an integer parameter y
  - Keyword int preceding the function name square indicates that the function returns an integer result
  - The return statement in square passes the (computed) value of the expression y * y back to the *calling function*

Functions Definitions

SINGAPORE INSTITUTE OF TECHNOLOGY

- Format of a function definition
  - *return-value-type function-name (parameter-list)*
    {
    definitions
    statements
    }
  - *Function-name is any valid identifier*
  - The *return-value-type* is the data type of the result returned to the caller
  - The return-value-type void indicates that a function **does not return** a value
  - The *return-value-type*, *function-name* and *parameter-list* are sometimes referred to as the function header

Functions Definitions

SINGAPORE INSTITUTE OF TECHNOLOGY

- Parameter-list
  - The *parameter-list* is a *comma-separated* list that specifies the **parameters / arguments** to be received by the function when it is called
  - If the *parameter-list* is *void or empty*, the function does not receive any parameters in the function call
  - A type is explicity listed for each parameter
    - E.g. *double x, double y*

      However, *double x, y* is a syntax error
  - Defining a parameter again as a local variable in the function is an error
- The definitions and statements within the braces form the body of the function

Functions Definitions

SINGAPORE INSTITUTE OF TECHNOLOGY

- **Note**
  - *Defining* a function *inside* another function is a syntax error
  - Programs should be written as *collections* of *small functions*. This makes the program easier to write, debug, maintain and modify
  - A function requiring a *large number* of *parameters* may be performing too many tasks
    - breaking the above function into *smaller functions* with each function performing a specific task
    - The *function header* should normally fit on one line
- The *function prototype*, *function header* and *function calls* should all agree in the number, type, and order of *parameters / arguments*, and in the type of the *return value*

Functions Definitions

SINGAPORE INSTITUTE OF TECHNOLOGY

- Return from a function call
  - *Control* is *returned* from the called function to the *point* at which the *function* was *initially invoked*
  - If the function does not return a value, the statement is
    - return;
  - If a function does return a value, the statement is
    - return expression;
    - the *expression* computed is returned to the caller
- The main function
  - Main has an int return type
  - Return value can be used to indicate whether the program did execute correctly
  - **return 0** can be explicitly placed
  - **0** indicates that the program *ran without any error*
  - main implicitly returns 0 if it is omitted

Functions Definitions

SINGAPORE INSTITUTE OF TECHNOLOGY

- Let's visit another Function maximum (Fig 5.4)
  - User-defined function *maximum* to determine and return the largest of three integers
  - Three integers are *passed* (as *parameters/arguments*) to the function maximum (*line 19*) which determines the largest integer
  - The *value* of the largest integer is *returned* to main by the *return* statement (*line 36*)

Functions Definitions

```
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( void )
9  {
10    int number1; // first integer entered by the user
11    int number2; // second integer entered by the user
12    int number3; // third integer entered by the user
13
14    printf( "%s", "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 } // end main
21
```

Functions Definitions

SINGAPORE INSTITUTE OF TECHNOLOGY

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

```
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.)

Functions Prototypes

SINGAPORE INSTITUTE OF TECHNOLOGY

- C compiler uses the function prototypes to validate function calls
  - Include function prototypes for all functions
  - C's *type-checking* capability
  - Use #include preprocessor directives to obtain function prototypes for the standard library function from the headers for the relevant libraries, or
  - to obtain headers containing function prototypes for user developed functions
- The function prototype for *maximum* (line 5) is
  - *int maximum (int x, int y, int z);*
  - It states that the maximum takes *three arguments* of type int, and *returns* a result of type int
  - Note that the *function prototype* is the same as the first line of maximum's *function definition* and with a *semicolon* at the end
  - Parameter names included are ignored by the compiler

Functions Prototypes

SINGAPORE INSTITUTE OF TECHNOLOGY

Compilation errors

- A *function call* that **does not match** the *function prototype*
- If the *function prototype* and the *function definition* **disagree**
- For e.g., void maximum (int x, int y, int z);
- The compiler will generate an error because the void return type in the function prototype would differ from the int return type in the function definition

Coercion of arguments

- **Forcing** the arguments to the appropriate type
- In general, argument values that do not correspond precisely to the parameter types in the function prototype are **converted** to the *proper type* before the function is called
- For e.g., printf("%.3f\n", sqrt(4)); ← prints the value of 2.000
- Note that int is automatically converted to a double without changing its value but a double converted to an int truncates the functional part of the double value

Functions Prototypes

SINGAPORE INSTITUTE OF TECHNOLOGY

- Mixed-type expression
  - The compiler makes a *temporary copy* of the value that needs to be converted and then *converts the copy* to the "highest" type in the expression (original value remains unchanged)
  - If one of the values is a long double/double/float, the other is converted to a long double/double/float
- If there is no function prototype for a function, the compiler forms its own function prototype using the first occurrence of the function; typically leads to warnings or errors (depends on compiler)
- A *function prototype* placed *outside* any function definition *applies* to *all calls* to the function *appearing after* the function prototype in the file
- *A function prototype placed in* a function *applies only to calls make in* that function

Functions

SINGAPORE INSTITUTE OF TECHNOLOGY

- **Headers**
- **Pass by value and Pass by reference**
- **Function call stack**

Headers

- Each standard library has a corresponding header containing the *function prototypes* for *all the functions* in that *library* and *definitions of various data types and constants* needed by those functions
- **Figure 5.10** lists alphabetically some of the standard library headers that may be included in programs
- One can create custom headers – programmer-defined headers should use the .h filename extension
- A *programmer-defined header* can be included by using the #include preprocessor directive, for example
  - *#include "myown.h"*
  - and in the file myown.h, it contains a collection of programmer's function prototypes:

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

Headers

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.
`<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.)

Passing Arguments by Value and by Reference

SINGAPORE INSTITUTE OF TECHNOLOGY

- Two ways to pass arguments
  - *pass-by-value* and *pass-by-reference*

- *Pass-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*
  - The caller allows the called function to *modify* the original variable's value

Passing Arguments by Value and by Reference

SINGAPORE INSTITUTE OF TECHNOLOGY

- *Pass-by-value* should be used whenever the called function *does not need to modify* the value of the caller's original variable

- *Pass-by-reference* should be used with functions that need to *modify* the original variable

- *In C, all arguments are passed by value*
  - *Refer to program (Fig. 5.4)*
    - *maximum(number1,number2,number3) // function call*
    - *int maximum(int x, int y, int z) // function definition*

- *It is possible to simulate pass-by-reference by using the address operator (&) and the indirection operator (*)*

Function Call Stack

SINGAPORE INSTITUTE OF TECHNOLOGY

- Stack data structure
  - *Last-in, first-out* (LIFO) or *First-in, last-out* (FILO)
  - Last item pushed (inserted) on the stack is the first item popped (removed) from the stack
- Each *function may call other functions*, which may call other functions – *all before any function returns*
  - Function call stack is used to keep track of the *return addresses* that each function needs to return control to the function that called it
- Each called function always finds the information it needs to return to its caller at the *top* of the stack

Function Call Stack

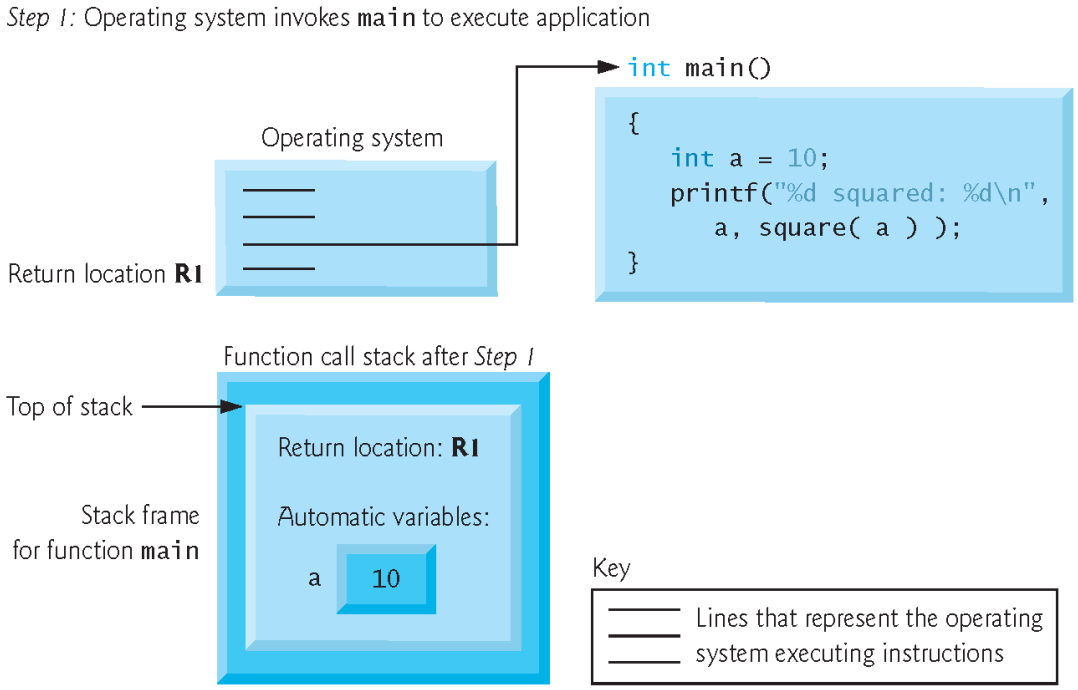
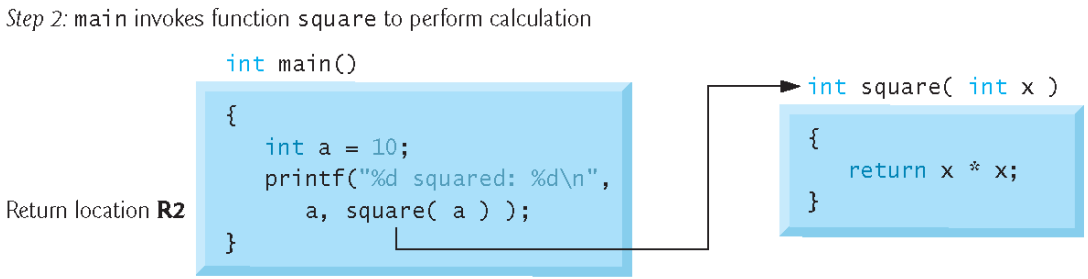


Fig. 5.7 | Function call stack after the operating system invokes `main` to execute the program.



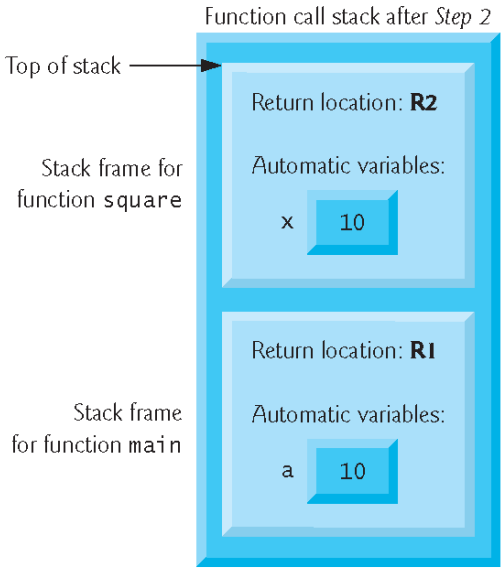


Fig. 5.8 | Function call stack after `main` perform the calculation.

Functions

SINGAPORE INSTITUTE OF TECHNOLOGY

- **Storage Classes**
- **Scope Rules**

Storage Classes

SINGAPORE INSTITUTE OF TECHNOLOGY

- **Local variables**
  - Only *variables* can have automatic storage duration
  - Keyword *auto* explicitly declares variables of automatic storage duration
  - A function's *local variables* (those declared in the parameter list or function body) normally have automatic storage duration by default, so keyword *auto* is rarely used
  - Conserves memory – *created* when a function is *entered* and *destroyed* when the function is *exited*

- **Static Storage Class**
  - Keywords *extern* and *static* are used
  - Identifiers of *static storage duration* exist from the time at which the program begins execution until the program terminates
  - For *static* variables, storage is allocated and initialized *only once, before* the program begins execution
  - Types of identifiers with static storage : storage class extern (e.g. global variables and function names) and local variables declared with static

Storage Classes

SINGAPORE INSTITUTE OF TECHNOLOGY

- **Global variables and function names**
  - Storage class *extern* by default
  - *Global variables* are created by placing variable declarations *outside* any function definition, and they retain their values throughout the execution of the program
  - *Global variables* and *functions* can be referenced by any function that follows their declarations or definitions in the file
  - This is *one reason* for using *function prototypes*—when we include stdio.h in a program that calls printf, the function prototype is placed at the start of our file to make the name printf known to the rest of the file
- Variables used only in a particular function should be defined as *local variables* in that function vs as *extern variables*

Storage Classes

SINGAPORE INSTITUTE OF TECHNOLOGY

- **Local variables declared with the keyword static**
  - Still known only in the function in which they're defined
  - Unlike automatic variables, static local variables *retain* their value when the *function is exited*
  - The next time the function is called, the static local variable contains the value it had when the function last exited
  - e.g. static int count = 1;
- All numeric variables of static storage duration are initialized to zero by default if you do not explicitly initialize them

Scope Rules

SINGAPORE INSTITUTE OF TECHNOLOGY

- The scope of an identifier is the portion of the program in which the identifier can be referenced
- The four identifier scopes are function scope, file scope, block scope, and function-prototype scope
- An identifier declared outside any function has file scope
  - Such an identifier is “known” (i.e., accessible) in all functions from the point at which the identifier is declared until the end of the file
  - Global variables, function definitions, and function prototypes placed outside a function all have file scope
- Identifiers defined inside a block have block scope
  - Block scope ends at the *terminating right brace (}) of the block*
  - *Local variables* defined at the beginning of a function have block scope as do *function parameters*, which are considered local variables by the function

Scope Rules

SINGAPORE INSTITUTE OF TECHNOLOGY

- When blocks are nested, and an identifier in an outer block has the same name as an identifier in an inner block, the identifier in the outer block is "*hidden*" until the inner block terminates
- This means that while executing in the inner block, the inner block sees the value of its own local identifier and not the value of the identically named identifier in the enclosing block

➢ Local variables declared static still have block scope, even though they exist from before program startup

➢ The only identifiers with function-prototype scope are those used in the parameter list of a function prototype

- As mentioned previously, function prototypes do not require names in the parameter list—only types are required
- If a name is used in the parameter list of a function prototype, the compiler ignores the name
- Identifiers used in a function prototype can be reused elsewhere in the program without ambiguity

Functions

SINGAPORE INSTITUTE OF TECHNOLOGY

- **Recursion**

Recursion

SINGAPORE INSTITUTE OF TECHNOLOGY

- A recursive function is a function that calls itself either directly or indirectly through another function
- If the function is called with a *base case*, the function simply *returns a result*
- When the function is called with a more complex problem, the function divides the problem into two conceptual pieces: a piece that the function knows how to do and a piece that it does not know how to do
- Because this new problem looks like the original problem, the *function launches (calls) a fresh copy of itself to go to work on the smaller problem*—this is referred to as a recursive call or the recursion step

Recursion

SINGAPORE INSTITUTE OF TECHNOLOGY

Recursively Calculating Factorials

- The factorial of a nonnegative integer *n*, written *n!* (pronounced "*n* factorial"), is the product

  $n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$

- with 1! equal to 1, and 0! defined to be 1
- For example, 5! is the product 5 * 4 * 3 * 2 * 1, which is equal to 120
- The factorial of an integer number greater than or equal to 0 can be calculated iteratively (nonrecursively) using a for statement as follows:
  - `factorial = 1;`
  - ```
    for ( counter = number; counter >= 1; counter-- )
       factorial *= counter;
    ```
- A *recursive* definition of the factorial function is arrived at by observing the following relationship:
  - $n! = n \cdot (n-1)!$
    ```

Recursion

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

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

- The evaluation of 5! would proceed as shown in Fig. 5.17.
- Figure 5.17(a) shows how the succession of recursive calls proceeds until 1! is evaluated to be 1 (i.e., the *base* case), which terminates the recursion.
- Figure 5.17(b) shows the values returned from each recursive call to its caller until the final value is calculated and returned.

Recursion

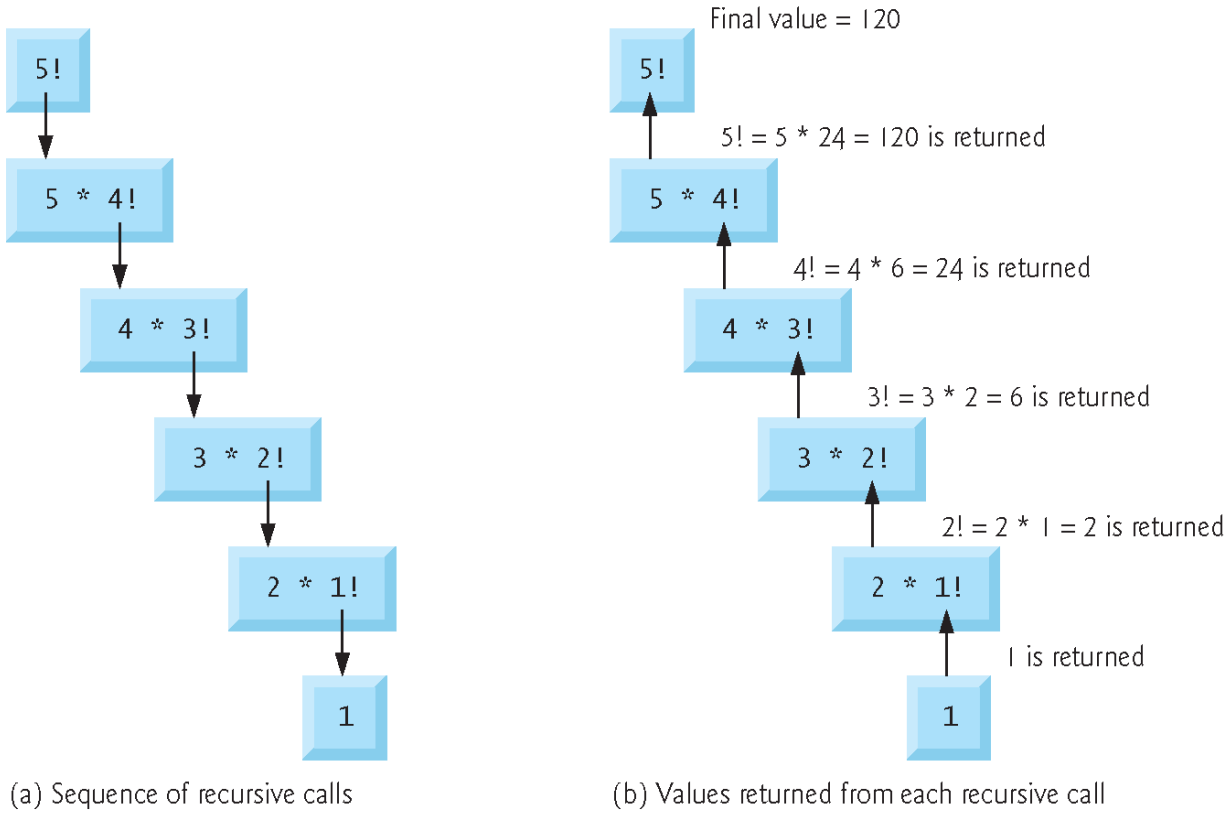


Fig. 5.17 | Recursive evaluation of 5!.

Recursion

```
1  // Fig. 5.18: fig05_18.c
2  // Recursive factorial function.
3  #include <stdio.h>
4
5  unsigned long long int factorial( unsigned int number );
6
7  // function main begins program execution
8  int main( void )
9  {
10    unsigned int i; // counter
11
12    // during each iteration, calculate
13    // factorial( i ) and display result
14    for ( i = 0; i <= 21; ++i ) {
15       printf( "%u! = %llu\n", i, factorial( i ) );
16    } // end for
17 } // end main
18
19 // recursive definition of function factorial
20 unsigned long long int factorial( unsigned int number )
21 {
22    // base case
23    if ( number <= 1 ) {
24       return 1;
25    } // end if
26    else { // recursive step
27       return ( number * factorial( number - 1 ) );
28    } // end else
29 } // end function factorial
```

factorial(4)

$5! = 5 \times 4!$

$4 \times$ factorial(3)

:

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

Recursion

SINGAPORE INSTITUTE OF TECHNOLOGY

- If number is indeed less than or equal to 1, factorial returns 1, no further recursion is necessary, and the program terminates.
- If number is greater than 1, the statement

  return number * factorial(number - 1);

  - expresses the problem as the product of number and a recursive call to factorial evaluating the factorial of number - 1.
- The call factorial(number - 1) is a slightly simpler problem than the original calculation factorial(number).

Recursion

```
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.)