FUNCTION

30 October 2024 16:00

FUNCTION:

USEFUL FUNCTIONS IN C MATH LIBRARY

`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
`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

ALSO FMOD(X,Y) gives you the remainder of x/y as a floating point number

FUNCTION DEFINITION

Always include function prototype
If return nothing void
If one of input is array u need to put *
Ex.
Void searchnum(int *array, int search)

-aritmetic conversion is handled by compiler

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`

A value can be converted to a lower type only by explicitly assigning the value to a variable of lower type or by using a cast operator. So, if we pass a double to our square function in Fig. 5.1, the double is converted to int (a lower type), and square usually returns an incorrect value. For example, square(4.5) returns 16, not 20.25.

Obtaining a random integer value:

Int value = rand()

Generates a num between 0 and rand_max()
This is pseudo random

Storage classes:

C provides the storage-class specifiers auto, register, 5 extern and static. 6 A storage class determines an identifier's storage duration, scope and linkage. Storage duration is the period during which an identifier exists in memory. Some exist briefly, some are repeatedly created and destroyed, and others exist for the entire program execution. Scope determines where a program can reference an identifier. Some can be referenced throughout a program, others from only portions of a program. For a multiplesource-file program, an identifier's linkage determines whether the identifier is known only in the current source file or in any source file with proper declarations.

The storage-class specifiers are split between automatic storage duration and static storage duration. The auto keyword declares that a variable has automatic storage duration (only variable)
static storage duration. Identifiers of static storage duration exist from the time at which the program begins execution until it terminates. For static variables, 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.

SCOPE RULE
The scope of an identifier is the portion of the program in which the identifier can be referenced. For example, a local variable in a block can be referenced only following its definition in that block or in blocks nested within that block. The four identifier scopes are function scope, file scope, block scope and function-prototype scope.

FUNCTION SCOPE
Labels are identifiers followed by a colon such as start:. Labels are the only identifiers with function scope. Labels can be used anywhere in the function in which they appear, but they cannot be referenced outside the function body.

FILE 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.
(e. Global variables, function definitions and function prototypes placed outside a function all have file scope.)

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. Any block may contain variable definitions.

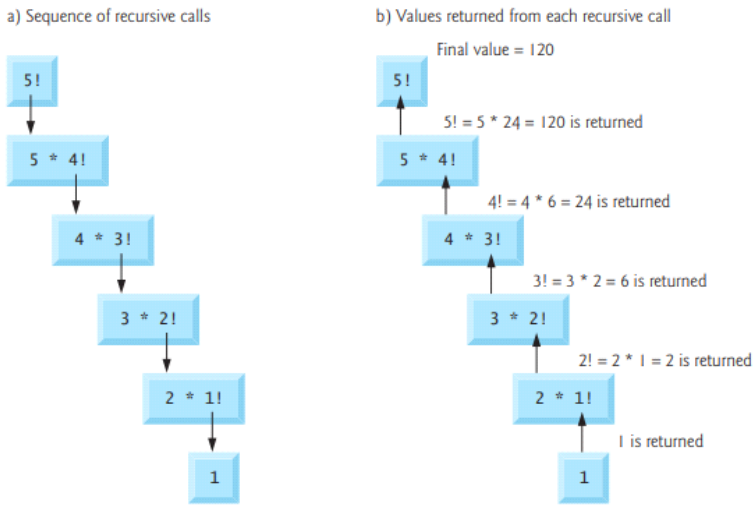
FUNCTINO PROTOTYPE SCOPE
The only identifiers with function-prototype scope are those used in the parameter list of a function prototype

SCOPING EXAMPLE

```c
// fig05_08.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);

   useLocal(); // useLocal has automatic local x
   useStaticLocal(); // useStaticLocal has static local x
   useGlobal(); // useGlobal uses global x
   useLocal(); // useLocal reinitializes automatic local x
   useStaticLocal(); // static local x retains its prior value
   useGlobal(); // global x also retains its value

   printf("\nlocal x in main is %d\n", x);
}

// useLocal reinitializes local variable x during each call
void useLocal(void) {
   int x = 25; // initialized each time useLocal is called

   printf("\nlocal x in useLocal is %d after entering useLocal\n", x);
   ++x;
```

ig. 5.8 | Scoping. (Part 1 of 2.)

RECURSION:



```c
// fig05_09.c
// Recursive factorial function.
#include <stdio.h>

unsigned long long int factorial(int number);

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

// recursive definition of function factorial
unsigned long long int factorial(int number) {
   if (number <= 1) { // base case
      return 1;
   }
   else { // recursive step
      return (number * factorial(number - 1));
   }
}
```

Fig. 5.9 | Recursive factorial function. (Part 1 of 2.)

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:

```c
unsigned long long int factorial = 1;
for (int counter = number; counter > 1; --counter)
    factorial *= counter;
```

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

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

For example, 5! is clearly equal to 5 * 4! as 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 recursive factorial function first tests whether a terminating condition is true, i.e., whether number is less than or equal to 1.
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).