Chapter 5: Functions

* Experience has shown that the best way to develop and maintain a large program is to construct it from a smaller pieces or modules each of which is more manageable than the original program.

Program Modules in C

* Modules in C are called functions/
* C programs are typically written by combining new functions the programmer writes with "pre-packaged" functions available in the C standard library.
* The programmer can write functions to define specific tasks that may be used at many points in a program.
* The actual statements defining the function are written only once, and the statements are hidden from other functions.
* Functions are invoked by a function call. The function call specifies:
    1. function name
    2. information (arguments) that the called function needs in order to perform its designed task.
* The called function reports back - or returns - to the calling function when its task is completed, with a return value.

Math Library Functions

* Math library functions allow the programmer to perform certain common mathematical calculations.
* Functions are normally used in a program by writing the name of the function followed by a left parenthesis followed by the argument (or a comma separated list of arguments) of the function followed by a right parenthesis.
* Commonly used math library functions:
    1. sqrt(x) - square root of x
    2. exp(x) - exponential function of e
    3. log(x) - natural logarithm of x (base e)
    4. log10(x) - logarithm of x (base 10)
    5. fabs(x) - absolute value of x
    6. ceil(x) - rounds x to the smallest integer not less that x
    7. floor(x) - round x to the largest integer not greater than x
    8. pow(x,y) - x raised to power y
    9. fmod(x,y) - remainder of x/y as a floating point number
    10. sin(x) - trigonometric sine of x (x in radians)
    11. cos(x) - trigonometric cosine of x (x in radians)
    12. tan(x) - trigonometric tangent of x (x in radians)

Functions

* All variable declared in function definitions are local variables - they are known only in the function in which they are defined.
* Most functions have a list of parameters.
* The parameters provide the means for communicating information between functions.
* A function's parameters are also local variables (with initial values).
* There are several motivations for "dividing" code into procedures:
    1. more manageable.
    2. software reusability
    3. to avoid repeating code

Function Definitions

----

* E.g.

```
        #include <stdio.h>
        int square(int);
        main()
        {
            int x;

            for (x=1; x<=10; x++)
                printf("%d  ",square(x));
            printf("\n");
            return 0;
        }

        int square(int y)
        {
            return y*y;
        }
```

* Function "square" is invoked or called in "main" within the "printf" statement.
* Function "square" receives a copy of the value of "x" in the parameter "y".
* The result is passed back to the "printf" function in "main" where "square" was invoked, and "printf" displays the result.

* The definition of "square" shows that "square" expects an integer parameter "y".
* The keyword "int" preceding the function name indicates that "square" returns an integer result.
* The "return" statement in "square" passes the result of the calculation back to the calling function.

* The line "int square(int);" is a function prototype.
* The compiler refers to the function prototype to check that calls to "square" contain the correct return type, the correct number of arguments, the correct argument types, and that the arguments are in the correct order.

* The format of a function definition is:

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

            statements
        }
```

* The return-value-type "void" indicates that a function does not return a value.
* An unspecified return-value-type is always assumed by the compiler to be "int".
* The parameter-list is a comma-separated list containing the declarations of the parameter received by the function when it is called.
* A function cannot be defined inside another function under any circumstances.

* There are three ways to return control to the point at which a function was invoked:
    1. the function-ending right brace is reached, if it does not

    return a result.
    2. "return;", if it does not return a result.
    3. "return expression;", returns the value of expression to the
    caller.
* E.g.

```
        #include <stdio.h>
        int maximum(int,int,int);
        main()
        {
            int a, b, c;
            printf("Enter three integers: ");
            scanf("%d%d%d", &a, &b, &c);
            printf("Maximum is: %d\n",maximum(a, b, c));

            return 0;
        }

        int maximum(int x, int y, int z)
        {
            int max;

            max = x;
            if (y > max)
                max = y;
            if (z > max)
                max = z;
            return max;
        }
```

Header Files

* Each standard library has a corresponding header file containing:
    1. the function prototypes for all the functions in that library
    2. definitions of various data types
    3. constants needed by those functions.
* E.g.
    <math.h> - Contains function prototypes for math library
    functions.
    <stdio.h> - Contains function prototypes for the standard I/O
    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.

* Programmer-defined header files should also end in ".h".
* A programmer-defined header file can be included by using the
"#include" preprocessor directive.

Calling Functions: Call by Value and Call by Reference

* When arguments are passed call 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.
* When an argument is passed call by reference, the caller actually

allows the called function to modify the original variable's value.
* Call by value prevents the accidental side effects by modifying the value of the caller's original variable.

* In C, all calls are call by value.
* It is possible to simulate call by reference by using address operators and indirection operators (call by address).

* E.g.
```
        #include <stdio.h>
        #include <stdlib.h>

        main()
        {
            int i;

            srand(time(NULL));
            for (i=1; i<=20; i++)
            {
                printf("%10d",1+(rand()%6));
                if (i%5 == 0)
                    printf("\n");
            }

            return 0;
        }
```

Storage Classes
----
* Each identifier in a program has other attributes including storage class, storage duration, scope and linkage.

* C provides four storage classes:
    1. auto
    2. register
    3. extern
    4. static

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

* The four storage class specifiers can be splitted into two storage duration:
    1. automatic storage duration : auto, register
    2. static storage duration: extern, static

* Variables with automatic storage duration are
    1. created when the block in which they are declared is entered
    2. exist while the block is active
    3. destroyed when the block is exited.
* A function's local variables normally have automatic storage duration. E.g.

```
        auto float x, y;
```
* Local variables have automatic storage duration by default, so the "auto" keyword is rarely used.

* The storage class specifier "register" suggest that the compiler maintain the variable in one of the computer's high-speed hardware registers.
* The compiler may ignore "register" declarations.
* E.g.
```
        register int counter = 1;
```
* The "register" keyword can be used only with variables of automatic storage duration.

* Identifiers of static storage duration exist from the point at which the program begins execution.
* For variables, storage is allocatedand initialized once when the program begins execution.
* Even though the variables exist, this does not mean that these identifiers can be used throughout the program.
* Two type of identifiers with static storage duration:
    1. external identifiers (such as global variables and function names)
    2. local variables declared with the storage class specifier "static".
* 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 definition in the file.

* Local variables declared with the keyword "static" are still known only in the function in which they are defined.
* 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;
```

Scope Rules
----
* The scope of an identifier is the portion of the program in which the identifier can be refernced.
* The four scopes for an identifier are:
    1. function scope
    2. file scope
    3. block scope
    4. function-prototype scope

* Labels (an identifier followed by a colon such as "start:") are the only identifiers with function scope.
* Labels can be used anywhere in the function in which they appear, but cannot be referenced outside the function body.

* An identifier declared outside any function has file scope.
* Such an identifier is "known" in all functions from the point at which the identifier is declared until the end of the file.
* E.g. Global variables, function definitions, etc.

* Identifiers declared inside a block have block scope.
* Block scope ends at the terminating right brace ("}") of the block.

* E.g. Local variables, function parameters.
* Any block may contain variable declarations.
* 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.

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

* E.g.

```
        #include <stdio.h>
        void a(void);
        void b(void);
        void c(void);

        int x = 1;

        main()
        {
            int x = 5;

            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);
            }

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

        void a(void)
        {
            int x = 25;

            printf("\nlocal 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;

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

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

Recursion
----
* A recursive function is a function that calls itself either directly or indirectly through another function.
* A recursive function is called to solved a problem.
* The function actually knows how to solve only the simplest case(s), or so-called base case(s).
* If called with a base case, the function stops recursively calling itself and simply returns to its caller.
* If called with a more complex problem, the function divides the problem into two conceptual pieces: a piece that it knows how to do and a piece that it does not know how to do.
* To make recursion feasible, the latter piece must resemble the original problem, but be a slightly simpler or slightly smaller version of the original problem.
* 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 and is also called the recursion step.
* The recursion step executes while the original call to the function is still open, i.e., it has not yet finished executing.
* At a point, the function recognizes the base case, return a result to the previous copy of the function, and a sequence of return ensues all the way up the line until the original call of the function eventually returns the final result.
* E.g.

```
        #include <stdio.h>
        long factorial(long);
        main()
        {
            int i;

            for (i=1; i<=10; i++)
                printf("%2d! = %ld\n", i, factorial(i));
            return 0;
        }

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

Random Number Generation
----
* The element of chance can be introduced into computer applications by using the "rand" function in the C standard library.
* E.g.

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

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

* The function prototype for the "rand" function can be found in <stdio.h>.

* We can use the modulus operator (%) in conjunction with "rand" as follows

```
        rand() % 6
```

to produce integers in the range 0 to 5.
* This is called scaling. The number "6" is called the scaling factor.
* We then shift the range of numbers produced by adding an integer.

* The sequence of pseudo-random numbers repeats itself each time the program is executed.
* The "srand" function takes an "unsigned" integer argument and seeds the "rand" function to produce a different sequence of random numbers for each execution of the program.

* A two-byte "unsigned int" can have only positive values in the range 0 to 65535.
* The conversion specifier "%u" is used for "unsigned" in "printf" and "scanf".

* We may use statement like

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

to randomize without the need for entering a seed each time.
* The "time" function returns the current time of day in seconds.
* The function prototype of "time" is in <time.h>.

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

        main()
        {
            int i;
            unsigned seed;

            srand(time(NULL));
            for (i = 1; i <= 10; i++)
            {
                printf("%10d", 1 + (rand() % 6));

                if (i % 5 == 0)
                    printf("\n");
            }
            return 0;
        }
```

* Execrise
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 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. 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.