Agenda

C Functions
Introduction

- Most computer programs that solve real-world problems are much larger than the programs presented in the first few chapters. Experience has shown that the best way to develop and maintain a large program is to construct it from smaller pieces or modules, each of which is more manageable than the original one.

C Functions

Program Modules in C

- Modules in C are called functions. C programs are typically writtern by combining new functions you write with prepackaged functions available in C standard library.

- The C standard library provides a rich collection of functions for performing common mathematical calculations, string manipulations, characters manipulations, input/output, and many other useful operations.

Software Engineering Observation

Avoid reinventing the wheel. When possible, use C standard library functions instead of writing new functions. Thic can reduce program development time.

C Functions

Program Modules in C

- You can write functions to define specific tasks that may be used at many points in a program. These are sometimes referred to as programmer-defined functions. The actual statements defined the function are written only once, and the statements are hidden from other functions.

- Functions are invoked by a function call, which specifies the function name and provides information (as arguments) that the function needs to perform its designated task. A common analogy for this is the hierarchical form of management. A boss (the calling function or caller) asks a worker (the called function) to perform a task an report back when the task is done.

- For example, a function needing to display information on the screen calls the worker function printf to perform that task, the printf displays the information and reports back--or returns--to the calling function when its task is completd.

The boss function does not know how the worker function performs its designated tasks. The worker may call other worker functions, and the boss will be unavware of this. Figure 1.1 shows a boss function communicating with several worker functions in a hierarchical manner. Note that Worker1 acts as a boss function to worker4 and worker5. Relationships among functions may differ from the hiearchical structure shown in this figure.

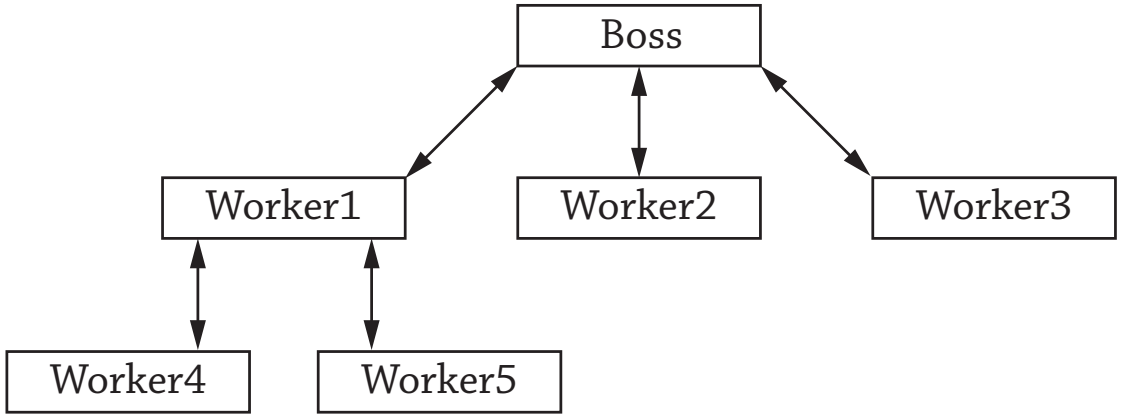


Fig 1.1 | Hierarchical boss-function/worker-function relationship.

Math Library Functions

- 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. For example, to calculate and print the square root of 900.0 you might write

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

When this statement executes, the math library function sqrt is called to calculate the square root of the number contained in the parentheses. The number 900 is the argument of the sqrt function.

- All functions in the math library that return floating-point values return the data type double. Note that double values, like float values, can be output using the %f conversion specification.

- Function arguments may be constants, variables, or expressions. If c1 = 13.0, d = 3.0 and f = 4.0, then the statement

```
printf( "%.2f", sqrt( c1 + d * f ) );
```

calculates and prints the square root of 13.0 + 3.0 * 4.0 = 25.0, namely 5.00.

Error-Prevention Tip

Include the math header by using the preprocessor directive #include <math.h> when using functions in the math library.

Figure 1.2 summarizes a small sample of the C math library functions. In the figure, the variables x and y are of type double. The C11 standard adds a wide range of floating-point and complex-number capabilities.

sqrt(x)	square root of x	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 ex	exp(1.0) is 2.718282	exp(2.0) is 7.389056
log(x)	natural logorithm 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
fabs(x)	absolute value of x as a floating-point number	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(-9.8) is -9.0	ceil(9.2) is 10.0
floor(x)	rounds x to the largest integer not greater than x	floor(-9.8) is -10.0	floor(9.2) is 9.0
pow(x, y)	x raised to power y (xy)	pow(2, 7) is 128.0	pow(9, .5) is 3.0
fmod(x)	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 1.2 | Commonly used math library functions.

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.

Software Engineering Observation

In programs containing many functions, main is often implemented as a group of calls to functions that perform the bulk of the program's work.

There are several motivations for "functionalizing" a program. The divide-and-conquer approach makes program development more manageable. Another motivation is software reusability--using existing functions as building blocks to create new programs. With good function naming and definition, programs can be created from standardized functions that accomplish specific tasks, rather than being built by using customized code. This is known as abstraction.

- A third motivation is to avoid repeating code in a program. Packaging code as a function allows the code to be executed from other locations in a program simply by calling the function.

Software Engineering Observation
Each function should be limited to performing a single, well-defined task, and the function name should express that task. This facilitates abstraction and promotes software reusability.

Software Engineering Observation
If you cannot choose a concise name that expresses what the function does, it's possible that your function is attempting to perform too many diverse tasks. It's usually best to break such a function into several smaller functions--this is sometimes called decomposition.

Function Definitions

- Each program we've presented has consisted of a function called main that called standard library functions to accomplish its tasks. We no consider how to write custom functions. Consider a program that uses a function square to calculate and print the squares of the integers from 1 to 10.

```
1   // Creating and using a programmer-defined function.
2   #include <stdio.h>
3
4   int square( int y ); // function prototype
5
6   // function main begins program execution
7   int main( void )
8   {
9     int x; // counter
10    for( x = 1; x <= 10; ++x ){
11      printf( "%d ", square( x ) ); // function call
12    } // end for
13    puts( "" );
14  } // end main
15
16  // square function definition returns the square of its parameter
17  int square( int y ) // y is a copy of the argument to the function
18  {
19    return y * y; // returns the square of y as an int
20  }
```

- The definition of function 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 passess the value of the expression y * y back to the calling function.

- int square(int y); // function prototype is a function prototype. The int in parentheses informs the compiler that square expects to receive an integer value from the caller. The int to the left of the function name square informs the compiler that square returns and integer result to the caller.

- The compiler refers to the function prototype to check that any calls to square contain the correct return type, the correct number of arguments and the correct argument types, and the arguments are in the correct order. The format of a function definition is

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

- The 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. Together, the return-value-type, function-name and parameter-list are sometimes referred to as the function header.

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

- The definitions and statements within braces form the function body, which is also referred to as a block. Variables can be declared in any block, and blocks can be nested.

Software Engineering Observation
Programs should be written as collections of small functions. This makes programs easier to write, debug, maintain and modify.

Function Definitions

- There are three ways to return control from a called function to the point at which a function was invoked. If the function does not return a result, control is returned simply when the function-ending right brace is reached, or by executing the statement

  return;

- If the function does return a result, the statement

  return expression;

  returns the value of expression to the caller.

Software Engineering Observation

A function requiring a large number of parameters may be performing too many tasks. Consider dividing the function into smaller functions that perform the separate tasks. The function header should fit on one line if possible.

Software Engineering Observation

The function prototype, function header and function calls should all agree in the number, type, and oder of arguments and parameter, and in the type of return value.

Function Definitions--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 explicity place

  return 0;

  at the end of main--0 indicates that a program ran successfully. The C standard indicates that main implicity returns 0 if you to moit the preceeding statement. You can explicity return non-zero values from main to indicate that a problem occured during your program's execution.

Function Definitions--Function maximum

- Our second example uses a progremmer-defined function maximum to determine and return the largest of three integer.

```
1   // Finding the maximum of three integers.
2   #include <stdio.h>
3
4   int maximum( int x, int y, int z ); // function prototype
5
6   // function main begins program execution
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 to the maximum function call
17    printf( “Maximum is: %d\n”, maximum( number1, number2, number3 ) );
18  } // end main
```

Function Definitions--Function maximum

```
19  // Function maximum definition x, y and z are paraemters
20  int maximum( int x, int y, int z )
21  {
22     int max = x; // assume x is largest
23     if( y > max ){ // if y is larger than max
24        max = y; // assign y to max
25     } // end if
26     if( z > max ){ // if z is larger than max
27        max = z; // assign y to max
28     } // end if
29     return max; // max is largest value
30  } // end function maximum
```

Function Prototypes: A Deeper Look

- The compiler uses function prototypes to validate function calls. Early versions of C did not perform this kind of checking, so it was possible to call functions improperly without the compiler detecing the errors. Such calls could result in fatal execution-time errors or nonfatal errors that caused subtle, difficult-to-detect problems. Function prototypes correct this deficiency.

Good Programming Practice

Include function prototypes for all functions to take advantage of C's type-chekcing capabilities. Use #include preprocessor directives to obtain function prototypes for the standard library function from the headers for the appropriate libraries, or to obtain headers containing function prototypes for functions developed by you and/or your gruop members.

- The function prototype for maximum is

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

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 definition.

Function Prototypes: A Deeper Look--Compilation Errors

- A function call that does not match the function prototype is a compilation error. An error is also generated if the function prototype and the function definition disagree. For example, if the function prototype had been written

```
void maximum( int x, int y, int z );
```

the compiler would generate an error because the void return type in the function prototype would differ from the int return type in the function header.

Function Prototypes: A Deeper Look--Argument Coercion and Usual Arithmetic Conversion

- Another important feature of function prototypes is the coercion of arguments, i.e., the forcing of arguments to the appropriate type. For example, the math library function sqrt can be called with an integer argument event though the function prototype in <math.h> specifies a double parameter, and the function will still work correctly.

- The statement

  printf(“%.3f\n”, sqrt(4));

  correctly evaluates sqrt(4) and prints the value 2.000. The function prototype causes the compiler to convert a copy of the integer value 4 to the double value 4.0 before the copy is passed to sqrt.

- In general, argument value that do not correspond precisely to the parameter types in the function prototype are converted to the proper type before the function is called. These conversions can lead to incorrect results if C’s usual artihmetic conversion rules are not followed.

- These rules specify how values can be convered to other types without losing data. In our sqrt example above, an int is automatically converted to a double without changing it value. However, a double converted to an int truncates the fractional part of the double value, thus changing the original value.

Function Prototypes: A Deeper Look--Argument Coercion and Usual Arithmetic Conversion

- The usual arithmetic conversion rules automatically apply to expressions containing values of two data types (also referred to as mixed-type expressions), and are handled for you by the compiler. In a mixed-type expression, the compiler makes a temporary copy of the value that needs to be converted then converts the copy to the highest type in the expression--the original value remains unchanged.

- The usual arithmetic conversion rules for a mixed-type expression containing at least one floating-point value are:

  If one of the values is a long double, the other is converted to a long double.

  If one of the values is a double, the other is converted to a double.

  If one of the values is a float, the other is converted to a float.

- If the mixed-type expression contains only integer types, then the usual arithmetic conversion specify a set of integer promotion rules. In most cases, the integer types lower are converted to types higher.

- Figure 1.3 lists the floating-point and integer data types with each type's printf and scanf conversion specification.

Floating-point types	**printf conversion specification**	**scanf conversion specification**
long double	%Lf	%Lf
double	%f	%f
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

Function Prototypes: A Deeper Look--Argument Coercion and Usual Arithmetic Conversion

- Converting values to lower types can result in incorrect values, so the compiler typically issues warnings for such cases. 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.

- Arguments in a function call are converted to the parameter types specified in a function prototype as if the arguments were being assigned directly to variables of those types. If our square function that uses an int parameter is called with a floating-point argument, the argument is converted to int (a lower type), and square usually returns an incorrect value.

Common Programming Error

Converting from a higher data type in the promotion hierarchy to a lower type can change the data value. Many compilers issue warnings in such cases.

Function Prototypes: A Deeper Look--Argument Coercion and Usual Arithmetic Conversion

- If there's no function prototype for a function, the compiler forms its own function prototype using the first occurence of the function--either the function definition or a call to the function. This typically leads to warnings or errors, depending on the compilier.

Error-Prevention Tip

Always include function prototypes for the functions you define or use in your program to help prevent compilation errors and warnings.

Function Call Stack and Stack Frames

- To understand how C performs function calls, we first need to consider a data structure known as a stack. Think of a stack as analogous to a pile of dishes. When a dish is placed on the pile, it's normally placed at the top (referred to as pushing the dish onto the stack.) Similarly, when a dish is removed from the pile, it's normally removed from the top (referred to as popping the dish of the stack).

- Stacks are known as last-in, first-out (LIFO) data structures--the last item pushed (inserted) on the stack is the first item popped (removed) from the stack.

- An important mechanism for computer science students to understand is the function call stack (sometimes referred to as the program execution stack). This data structure--working behind the scenes--supports the function call/return mechanism. It also supports the creation, maintenance and destruction of each called function's automatic variables.

Function Call Stack and Stack Frames

- Each function eventually must return control to the function that called it. So, we must keep track of the return addresses that each function needs to return control to the function that called it. Each time a function calls another function, an entry is pushed onto the stack. This entry, called a stack frame, contains the return address that the called function needs in order to return to the calling function.

- If the called function returns, instead of calling another function before returning, the stack frame for the function call is popped, and control transfers to the return address in the popped stack frame.

- Each called function always finds the information it needs to return to its caller at the top of the call stack. And, if a function makes a call to another function, a stack frame for the new function call is simply pushed onto the call stack. Thus, the return address required by the newly called function to return to its caller is now located at the top of the stack.

- The stack frames have another important responsibility. Most functions have automatic variables--parameters and some or all of their local variables. Automatic variables need to exist while a function is executing. They need to remain active if the function makes calls to other functions. But when a called function returns to its caller, the called function's automatic variables need to go away.

- The called function's stack frame is a perfect place to reserve the memory for automatic variables. The stack frame exists only as long as the called function is active. When that function returns--and no longer needs its local automatic variables--its stack frame is popped from the stack, and those local automatic variables are no longer known to the program.

- Of course, the amount of memory in a computer is finite, so only a certain amount of memory can be used to store stack frames on the function call stack. If more function calls occur than can have their stack frames stored on the function call stack, a fatal error known as stack overflow occurs.

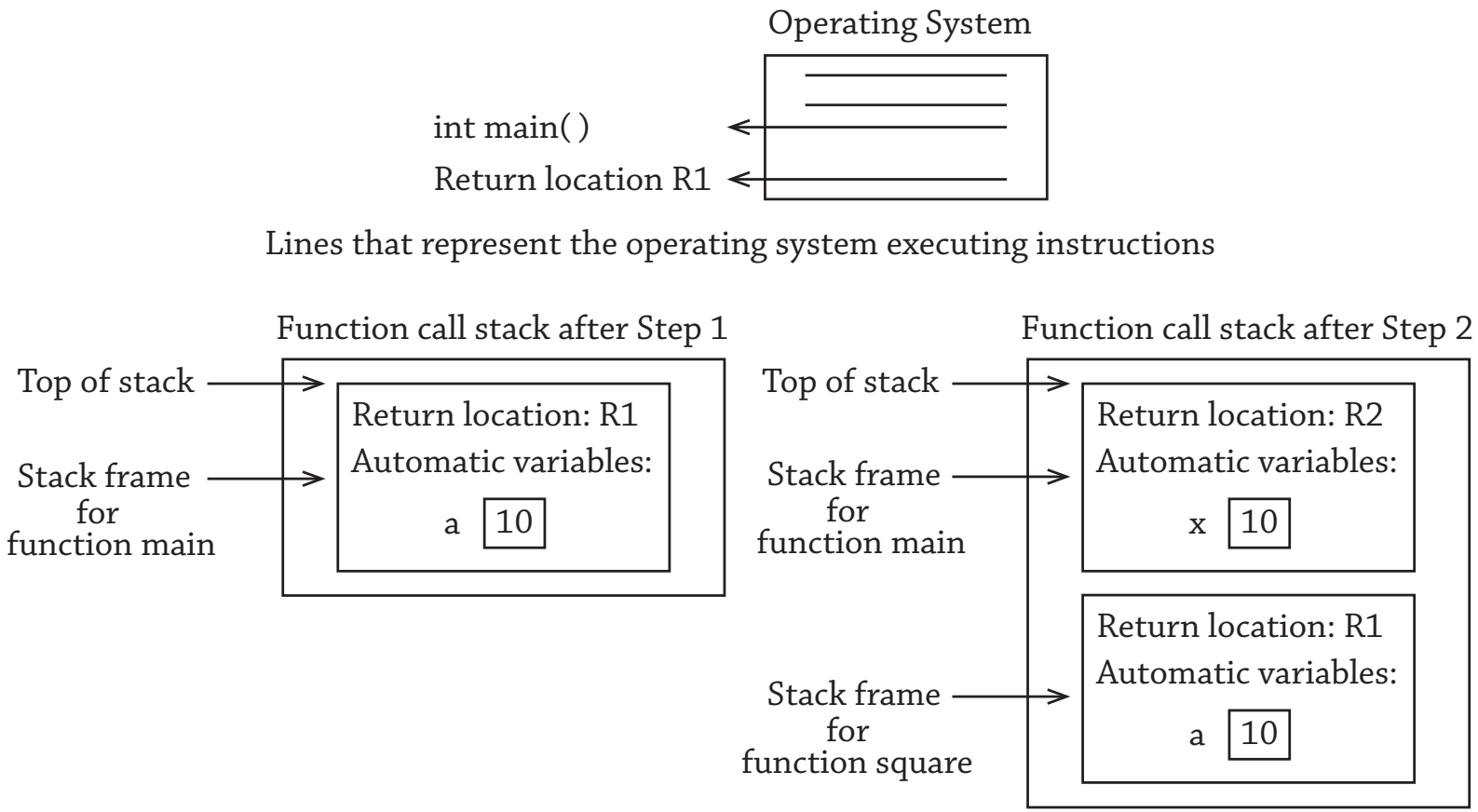
Function Call Stack and Stack Frames--Function Call Stack in Action

- Now let's consider how the call stack supports the operation of a square function called by main. First the operating system calls main--this pushes a stack frame onto the stack. The stack frame tells main how to return to the operating system and contains the space for main's automatic variable.

```
1   // Demonstrating the function call stack and stack frames using as function square
2   #include <stdio.h>
3
4   int square( int ); // prototype for function square
5
6   // function main begins program execution
7   int main( void )
8   {
9     int a = 10; // value to square (local automatic variable in main)
10    printf( "%d squared: %d\n", square( a ) );
11  } // end main
12
13  // returns the square of an integer
14   int square( int x )
15   {
16     return x * x; // calculate square and return result
17   } // end function square
```

Function Call Stack and Stack Frames--Function Call Stack in Action

- Function main--before returning to the operating system--now calls function square. This causes a stack frame for square to be pushed onto the function call stack. This stack frame contains the return address that square needes to return to main and the memory for square's automatic variable.



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. When arguments are passed by value, a copy of the arguments'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 by reference, the caller allows the called function to modify the original variable's value.

- Pass-by-value should be used whenever the called function does not need to modify the value of the caller's original variable. This prevents the accidental side effects (variable modifications) that so greatly hinder the development of correct and reliable software systems. Pass-by-reference should be used only with trusted called functions that need to modify the original variable.

- In C, all arguments are passed by value. It's possible to simulate pass-by-reference by using the address operator and inderection operator.

Random Number Generation

- The element of chance can be introduced into computer applications by using the 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). Standard C states that the value of RAND_MAX must be at least 32767, which is the maximum value for a two-byte integer. If rand truly produces integer at random, every number between 0 and RAND_MAX has an equal chance (or probability) of being chosen each time rand is called.

- The range of values produced directly by rand is often different from what's needed in a specific application. For example, a program that simulates coin tossing might require only 0 for heads and 1 for tails. A dice-rolling program that simulates a six-sided die would require random integer from 1 to 6.

Random Number Generation--Rolling a Six-Sided Die

- To demonstrate rand, let's develop a program to simulate 20 rolls of a six-sided die and print the value of each roll. The function prototype for function rand is in <stdlib.h>. We use the remainder operator (%) in conjunction with rand as follows

rand() % 6

to produce integers in the range 0 and 5. This is called scaling. The number 6 is called scaling factor. We then shift the range of numbers produced by adding 1 to our previous result.

```
1  // Shifted, scaled random integers produced by 1 + rand() % 6.
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  // function main begins program execution
6  int main( void )
7  {
8     unsinged int i; // counter
9
10    // loop 20 times
11    for( i = 1; i <= 20; ++i ){
12       printf( "%10d", 1 + rand() % 6 ) );
13       // if counter is divisible by 5, begin new line of output
14       if( i % 5 ==  0 ){
15          puts( "" );
16       } // end if
17    } // end for
18 } // end main
```

Random Number Generation--Randomizing the Random Number Generator

- Function rand actually generates pseudorandom numbers. Calling rand repeatedly produces a sequence of numbers that appears to be random. However, the sequence repeats itself each time the program is executed. Once a program has been thoroughly debugged, it can be conditioned to produce a different sequence of random numbers for each execution. This is called randomizing and is accomplished with the standard library function srand. Function srand takes an unsinged integer argument and seeds function rand to produce a different sequence of random numbers for each execution of the program.

- Function srand takes and unsigned int value as an argument. The conversion specifier %u is used to read an unsigned int value with scanf. The function prototype for srand is found in <stdlib.h>.

- 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. This value is converted to an unsigned integer and used as the seed to the random number generator. The function prototype for time is in <time.h>.

```
1  // Shifted, scaled random integers produced by 1 + rand() % 6.
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  // function main begins program execution
6  int main( void )
7  {
8     unsinged int i; // counter
9     unsinged int seed; // note %u for usinged int
10
11    printf( “%s”, “Enter seed: ” );
12    scanf( “%u”, &seed ); // seed the random number generator
13
14   srand( seed )
15    // loop 20 times
16    for( i = 1; i <= 20; ++i ){
17       printf( “%10d”, 1 + rand() % 6 ) );
18       // if counter is divisible by 5, begin new line of output
19       if( i % 5 ==  0 ){
20          puts( “” );
21       } // end if
22    } // end for
23 } // end main
```

Random Number Generation--Generalized Scaling and Shifting of Random Number

- The values produced directly by rand are always in the range 0 <= rand() <= RAND_MAX. As you know, the following statement simulates rolling a six-sided die face = 1 + rand() % 6. This statement always assigns an integer value (at random) to the variable face in the range 1 <= face <= 6. The width of this range is 6 and the starting number in the range is 1.

- Refererring to the preceding statement, we see that the width of the range is determined by the number used to scale rand with the remainder operator, and the starting number of the range is equal to the number that's added to rand %6. We can generalize this result as follows:

  n = a + rand() % b;

  where a is the shifting value (which is equal to the first number in the desired range of consequtive integers) and b is the scaling factor (which is equal to the width of the desired range of consecutive integers).

- The attributes of variables include name, type, size and value. We also use identifiers as names for user-defined functions. Actually, each identifier in a program has other attributes, including storage class, storage duration, scope and linkage.

- C provides the storage class specifiers auto, register, extern and static. An identifier's storage class determines its storage duration, scope and linkage. An identifier's storage duration is the period during which the identifier exists in memory. An identifier's scope is where the identifier can be referenced in a program. An identifier's 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.

- The storage class specifiers can be split automatic storage duration and static storage duration. Keyword auto is used to declare variables of automatic storage duration. 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.

- Only variables can have automatic storage duration. A function's local variables (those declared in the parameter list or function body) normally have automatic storage duration. Keyword auto explicitly declares variables of automatic storage duration.

- Local variables have automatic storage duration by default, so keyword auto is rarely used. For the remainder of the text, we'll refer to variables with automatic storage duration simply as automatic variables.

Performance Tip

Automatic storage is a means of conserving memory, because automatic variables exist only when they're needed. They're created when a function is entered and destroyed when the function is exited.

Storage Classes--Static Storage Class

- Keyword 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.

- 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 progrm begins execution. However, even though the variables and the function names exists from the start of program execution, this does not means these identifiers can be accessed throughout the program.

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

Software Engineering Observation
Defining a variable as global rather than local allows unintended side effects to occur when a function that does not need access to the variable accidentally or maliciously modifies it. In general, global variables should be avoided except in certain situations with unique performance requirements.

Software Engineering Observation
Variables used only in a particular function should be defined as local variables in that function rather than as external variables.

Storage Classes--Static Storage Class

- Local variables declared with the keyword static are still known only in the function in which they're defined, but 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. The following statement declares local variable could to be static and initializes it to 1.

```
static int count = 1;
```

- All numeric variables of static storage duration are initialized to zero by default if you do not explicity initialize them. Keywords extern and static have special meaning when explicitly applied to external identifiers.

- 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.

- The four identifier scopes are function scope, file scope, block scope, and function-prototype scope. Labels (identifiers 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, bu cannot be referenced outside the function body. Labels are used in switch statements (as case labels) and in goto statements.

- Labels are implementation details that functions hide from one another. This hiding--more formally called information hiding--is a means of implementing the principle of least privilage--a fundamental principle of good software engineering. In the context of an application, the principle states that code should be granted only the amount of privilege and access that it needes to accomplish its designated task, but no more.

- 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. 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 varibles defined at the beginning of a function have block scope, as do function parameters, which are considered local variables by the function.

- When blocks are nested, and an identifiers 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 identifiers 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. Thus, storage duration does not affect the scope of an identifier.

- 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.

```
1  // Scoping.
2  #include <stdio.h>
3
4  void useLocal( void ); // function prototype
5  void useGlobal( void ); // function prototype
6  void useStaticLocal( void ); // function prototype
7
8  int x = 1; // global variable
9
10 // function main begins program execution
11 int main( void )
12 {
13    int x = 5; // local variable to main
14    printf( "local x in outer scope of main is %d\n", x );
15
16    { // start new scope
17       int x = 7; // local variable to new scope
18       printf( "local x in inner scope of main is %d\n", x );
19    } // end of scope
20
21    printf( "local x in outer scope of main is %d\n", x );
22    useLocal(); // useLocal has automatic local x
23    useStaticLocal(); // useStaticLocal has static local x
24    useGlobal(); // useGlobal uses global x
25    useLocal(); // useLocal reinitializes automatic local x
26    useStaticLocal(); // static local x retains its prior value
27    useGlobal(); // gloal x also retains its value
28    printf( "local x in outer scope of main is %d\n", x );
29 } // end of scope
```

```
30  void useLocal( void )
31  {
32    int x = 25; // initialized each time useLocal is called
33    printf( "\nlocal x in useLocal is %d after entering useLocal\n", x );
34    ++x;
35    printf( "\nlocal x in useLocal is %d before exiting useLocal\n", x );
36  } // end function useLocal
37  void useStaticLocal( void )
38  {
39    static int x = 50; // initialized once before program startup
40    printf( "\nlocal static x is %d on entering useStaticLocal\n", x)
41    ++x;
42    printf( "\nlocal static x is %d on exiting useStaticLocal\n", x)
43  } // end function useStaticLocal
44  void useGlobal( void )
45  {
46    printf( "\nglobal x is %d on entering useGlobal\n", x );
47    x *= 10;
48    printf( "\nglobal x is %d on exiting useGlobal\n", x );
49  } // end function useGlobal
```

- A recursive function is a function that calls itself either directly or indirectly through another function. Recursion is a complex topic discussed at length in upper-level computer science courses.

- A recursive function is called to solve a problem. The function actually knows how to solve only the simplest case(s), or so-called base case(s). If the function is called with a base case, the function simply returns a result.

- If the function is called with a more complex problem, the function divides the problem into two conceptional pieces: a piece that the function knows how to do and a piece that is does not know how to do. To make recursion feasible, the latter piece must resemble the original problem, but be a slightly simpler or smaller version. Because this new problem looks like the original problem, the function launches a free copy of itself to go to work on the smaller problem--this is referred to as a recursive call or recursive step.

- The recursion step also includes the keyword return, because its result will be combined with the portion of the problem the function knew hwo to solve to form a result that will be passed back to the original caller.

- The recursion step executes while the original call to the function has not yet finished executing. The recursion step can result in many more such recursive calls, as the function keeps dividing each problem it's called with into two conceptual pieces.

- For the recursion to terminate, each time the function calls itself with a slightly simpler version of the original problem, this sequence of smaller problems must eventually converge on the base case. When the function recognizes the base case, it returns a result to the previous copy of the function, and a sequence of returns ensues all the way up the line until the original call of the function eventually returns the final result to main.

Recursion--Recursively Calculating Factorials

- 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 . (n-1)!. For example, 5! is clearly equal to 5 * 4!. The evaluation of 5! would proceed until 1! is evaluated to be 1 (i.e., the base case), which terminates the recursion.

Recursion--Recursively Calculating Factorials

- Figure below uses recursion to calculate and print the factorials of the integer 0-21.

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

- We've chosen the data type unsigned long long int so the program calculate larger factorial values. The conversion specifier %llu is used to print unsigned long long int values. Unfortunately, the factorial function produces large values so quickly that even unsigned long long int does not help us print very many factorial values before the maximum value of a unsigned long long int variable is exceeded. Event when we use unsigned long long int, we still can't calculate factorials beyond 21!.

Common Programming Error
Forgetting to return a value from a recursive function when one is needed.

Common Programming Error
Either omitting the base case, or writing the recursion step incorrectly so that it does not converge on the base case, will cause infinite recursion, eventually exhausting memory. This is analogous to the problem of an infinite loop in an iterative solution. Infinite recursion can also be caused by providing an unexpected input.

Recursion--Fibonacci Series

- The Fibonacci series 0, 1, 1, 2, 3, 5, 8, 13, 21, ..., begins with 0 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)

- Figure below calculates the nth Fibonacci number recursively using function fibonacci.

```
1   // Recursive fibonacci function.
2   #include <stdio.h>
3
4   unsigned long long int fibonacci(unsigned int n );
5
6   // function main begins program execution
7   int main( void )
8   {
9     unsigned int number; // counter
10    unsinged long long int resut; // fibonacci value
11
12    printf( “%s”, “Enter an integer: ” );
13    scanf( “%u”, &number );
14    result = fibonacci( number );
15    printf( “Fibonacci( %u ) = %llu\n”, number, result );
16  } // end main
17
18  unsigned long long int fibonacci( unsigned int number )
19    // base case
20    if( 0 == number || 1 == number ){
21      return number;
22    } // end if
23    else{ //recursive step
24      return fibonacci( number - 1 ) + fibonacci( number - 2 );
25    } // end else
26  } // end function factorial
```

Recursion--Fibonacci Series

- The call to fibonacci from main is not a recursive call, but all subsequent calls to fibonacci are recursive. Each time fibonacci is invoked, it immediately tests fro the base case--n is equal to 0 or 1. If this is true, n is returned. Interestingly, if n is greater then 1, the recursion step generates two recursive calls, each a slightly simpler problem than the original call to fibonacci.

- For fibonacci(3), two recusive calls will be made, namely fibonacci(2) and fibonacci(1). But in what order will these calls be made? You might simply assume the operands will be evaluated left to right. For optimization reasons, C does not specify the order in which the operands of most operators 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 then fibonacci(1), or the calls could execute in the reverse order, fibonacci(1) then fibonacci(2).

- 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 guareanteed to be evaluated left to right. The last operator is C's only ternary operator. Its leftmost operand is always evaluated first; if the leftmost operand evaluates to nonzer, the midde operand is evaluated next and the last operand is ignored; if the leftmost operand evaluates to zero, the third operand is evaluated next and the middle operand is ignored.

Recursion--Fibonacci Series--Exponential Complexity

- A word of caution is in order about recursive programs like the one we use here to generate Fibonacci numbers. Each level of recursion in the fibonacci function has a doubling effect on the number of calls--the number of recursive calls that will be executed to calcuate the nth fibonacci number is on the order of 2^n.

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

Recursion--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 statement; recursion achieves repetition through repeated function calls.
- Iteration and recursion each involve a termination test: Iteration terminates when the loop-continuation condition fails, recursion when a base case is recognized.
- Both iteration and recursion can occur infinitely: An infinite loop occurs with iteration if the loop-continuation test never becomes false; infinite recursion occurs if the recursion step does not reduce the problem each time in a manner that converges on the base case.

- 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.

- 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.

Software Engineering Observation
Any problem that can be solved recursively can also be solved iteratively (nonrecursively). A recursive approach is normally chosen in preference to an iterative approach when the recursive approach more naturally mirrors the problem and results in a programs that's easier to understand and debug. Another reason to choose a recusive solution is that an iterative soltion may not be apparent.