

# مبانی برنامه نویسی به زبان سی

۹، ۱۱ و ۱۳ آذر ۱۳۹۹

جلسه یازدهم، دوازدهم و سیزدهم

ملکی مجد

## مباحث این هفته:

- تقسیم برنامه به قطعات کوچکتر به نام ماژول
- تابع
  - تابع های کتابخانه ای
  - تابع های که خودمان تعریف می کنیم.

Function Call Stack and Activation Records •

Call by value •

تولید عدد رندم •

مثالی از بازی با تاس و آشنایی با enum •

Scope rules •

## divide and conquer

- Most computer programs are large
- 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 program.
- This technique is called **divide and conquer**.

## 5.2 Program Modules in C

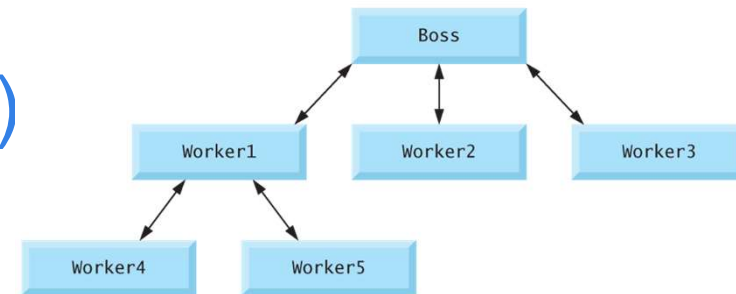
- Modules in C are called **functions**.
- C programs are typically written by combining new functions you write with
  - referred to as **programmer-defined functions**
  - Learn soon
- “prepackaged” functions available in the **C Standard Library**.
  - E.g., functions `printf`, `scanf` and `pow`

## 5.2 Program Modules in C (Cont.)

- You can write your own functions to define tasks that may be used at many points in a program.
- The statements defining 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 called function needs to perform its designated task.

## 5.2 Program Modules in C (Cont.)

- Consider a hierarchical form of management.

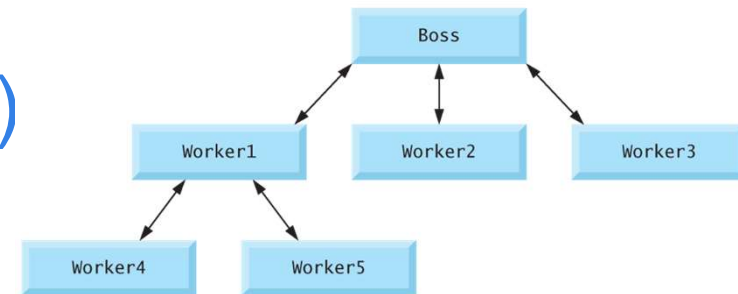


| Hierarchical boss function/worker function relationship.

A boss (the **calling function** or **caller**) asks a worker (the **called function**) to perform a task and report back when the task is done. The boss function does not know how the worker function performs its designated tasks.

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

## 5.2 Program Modules in C (Cont.)



| Hierarchical boss function/worker function relationship.

- The worker may call other worker functions, and the boss will be unaware of this.
  - Note that `worker1` acts as a boss function to `worker4` and `worker5`.

“hiding” of implementation details promotes good software engineering. Relationships among functions may differ from the hierarchical structure shown in this figure.

## 5.3 Math Library Functions

- Math library functions allow you 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.



## 5.3 Math Library Functions (Cont.)

- For example : `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 (900.0).
- Function arguments may be constants, variables, or expressions.
- For example:

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.

the variables `x` and `y` are of type `double`

Function	Description	Example
<code>sqrt( x )</code>	square root of $x$	<code>sqrt( 900.0 )</code> IS 30.0 <code>sqrt( 9.0 )</code> IS 3.0
<code>exp( x )</code>	exponential function $e^x$	<code>exp( 1.0 )</code> IS 2.718282 <code>exp( 2.0 )</code> IS 7.389056
<code>log( x )</code>	natural logarithm of $x$ (base $e$ )	<code>log( 2.718282 )</code> IS 1.0 <code>log( 7.389056 )</code> IS 2.0
<code>log10( x )</code>	logarithm of $x$ (base 10)	<code>log10( 1.0 )</code> IS 0.0 <code>log10( 10.0 )</code> IS 1.0 <code>log10( 100.0 )</code> IS 2.0
<code>fabs( x )</code>	absolute value of $x$	<code>fabs( 13.5 )</code> IS 13.5 <code>fabs( 0.0 )</code> IS 0.0 <code>fabs( -13.5 )</code> IS 13.5
<code>ceil( x )</code>	rounds $x$ to the smallest integer not less than $x$	<code>ceil( 9.2 )</code> IS 10.0 <code>ceil( -9.8 )</code> IS -9.0
<code>floor( x )</code>	rounds $x$ to the largest integer not greater than $x$	<code>floor( 9.2 )</code> IS 9.0 <code>floor( -9.8 )</code> IS -10.0

**Fig. 5.2** | Commonly used math library functions. (Part I of 2.)

Function	Description	Example
<code>pow( x, y )</code>	$x$ raised to power $y$ ( $x^y$ )	<code>pow( 2, 7 )</code> is 128.0 <code>pow( 9, .5 )</code> is 3.0
<code>fmod( x, y )</code>	remainder of $x/y$ as a floating-point number	<code>fmod( 13.657, 2.333 )</code> is 1.992
<code>sin( x )</code>	trigonometric sine of $x$ ( $x$ in radians)	<code>sin( 0.0 )</code> is 0.0
<code>cos( x )</code>	trigonometric cosine of $x$ ( $x$ in radians)	<code>cos( 0.0 )</code> is 1.0
<code>tan( x )</code>	trigonometric tangent of $x$ ( $x$ in radians)	<code>tan( 0.0 )</code> is 0.0

**Fig. 5.2** | Commonly used math library functions. (Part 2 of 2.)

## 5.4 Functions

- Functions allow you to modularize a program.
- All variables defined in function definitions are **local variables**—**they're known 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.

## 5.4 Functions (Cont.)

### motivations for “functionalizing” a program

- Divide-and-conquer approach makes program development more **manageable**.
- **Software reusability**—using existing functions as building-blocks to create new programs.
  - Software reusability is a major factor in the object-oriented programming movement that you’ll learn more about when you study languages derived from C, such as C++, Java and C# (pronounced “C sharp”).
  - We use abstraction each time we use standard library functions like `printf`, `scanf` and `pow`.
- **Avoid repeating code** in a program.
  - Packaging code as a function allows the code to be executed from several locations in a program simply by calling the function.

## 5.4 Math Library Functions (Cont.)

### Abstraction

- 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.
- We use abstraction each time we use standard library functions like `printf`, `scanf` and `pow`.

## how to write custom functions

```
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     printf( "\n" );
18     return 0; /* indicates successful termination */
19 } /* end main */
20
21 /* square function definition returns square of parameter */
22 int square( int y ) /* y is a copy of argument to function */
23 {
24     return y * y; /* returns square of y as an int */
25 } /* end function square */
```

1 4 9 16 25 36 49 64 81 100

**Fig. 5.3** | Using a programmer-defined function. (Part 2 of 2.)

## how to write custom functions

```
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     printf( "\n" );
18     return 0; /* indicates successful termination */
19 } /* end main */
20
21 /* square function definition returns square of parameter */
22 int square( int y ) /* y is a copy of x */
23 {
24     return y * y; /* returns square of y */
25 } /* end function square */
```

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.

**square** receives a copy of the value of **x** in the parameter **y**

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

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

**Fig. 5.3** | Using a programmer-defined function. (Part 2 of 2.)



## 5.5 Function Definitions (Cont.)

- 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**.

## 5.5 Function Definitions (Cont.)

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

### Common Programming Error 5.3

*Specifying function parameters of the same type as `double x, y` instead of `double x, double y` results in a compilation error.*

## 5.5 Function Definitions (Cont.)

- The *definitions* and *statements* within braces form the **function body**.
- The function body is also referred to as a **block**.
- Variables can be declared in any block, and blocks can be nested.
- A function cannot be defined inside another function.



#### **Good Programming Practice 5.4**

*Choosing meaningful function names and meaningful parameter names makes programs more readable and helps avoid excessive use of comments.*



### **Software Engineering Observation 5.10**

*The function prototype, function header and function calls should all agree in the number, type, and order of arguments and parameters, and in the type of return value.*

## 5.5 Function Definitions (Cont.)

- There are three ways to return control from a called function to the point at which a function was invoked.
  1. If the function does not return a result, control is returned simply when the function-ending right brace is reached, or
  2. by executing the statement
    1. `return;`
  3. If the function does return a result, the statement
    - `return expression;`returns the value of *expression to the caller*.

```

void printInt(int x){
    printf("PrintInt(1):\t%d\n",x);
    x++;
    printf("PrintInt(2):\t%d\n",x);
}
void printSomeInt(int x){
    printf("printSomeInt(1):\t%d\n",x);
    x++;
    if(x>2)
        return;
    printf("printSomeInt(2):\t%d\n",x);
    return;//optional!
}
int printAndRerunNewOne(int x){
    printf("printAndRerunNewOne:\t%d\n",x);
    x++;
    return x;
}

```

```

int main(){
    int i = 2;
    printInt(i);
    printSomeInt(i);
    i = printAndRerunNewOne(i);
    printf("main:\t%d\n",i);
    return 0;
}

```

```

PrintInt(1):    2
PrintInt(2):    3
printSomeInt(1):    2
printAndRerunNewOne:    2
main:    3

```

determine and return the largest of three integers

```
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 */
11     int number2; /* second integer */
12     int number3; /* third integer */
13
14     printf( "Enter three integers: " );
15     scanf( "%d%d%d", &number1, &number2, &number3 );
16
17     /* number1, number2 and number3 are arguments
18        to the maximum function call */
19     printf( "Maximum is: %d\n", maximum( number1, number2, number3 ) );
20     return 0; /* indicates successful termination */
21 } /* end main */
22
23 /* Function maximum definition */
24 /* x, y and z are parameters */
25 int maximum( int x, int y, int z )
26 {
27     int max = x; /* assume x is largest */
28
29     if ( y > max ) { /* if y is larger than max, assign y to max */
30         max = y;
31     } /* end if */
32
33     if ( z > max ) { /* if z is larger than max, assign z to max */
34         max = z;
35     } /* end if */
36
37     return max; /* max is largest value */
38 } /* end function maximum */
```

Enter three integers: 22 85 17  
Maximum is: 85

Enter three integers: 85 22 17  
Maximum is: 85

Enter three integers: 22 17 85  
Maximum is: 85



## 5.6 Function Prototypes

- A function prototype tells the compiler
  - the type of data returned by the function,
  - the number of parameters the function expects to receive,
  - the types of the parameters, and
  - the order in which these parameters are expected.
- The compiler uses function prototypes to validate function calls.
- If there is no function prototype for a function, the compiler forms its own function prototype using the first occurrence of the function—either the function definition or a call to the function.
  - This typically leads to warnings or errors, depending on the compiler.

The function prototype for `maximum` in Fig. 5.4 (line 5) is

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

This function prototype states that `maximum` takes three arguments of type `int` and returns a result of type `int`.

## 5.6 Function Prototypes (Cont.)

- 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, in Fig. 5.4, 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.

## 5.6 hierarchy in data types (Cont.)

- The type of each value in a mixed-type expression is automatically promoted to the “highest” type in the expression (**actually a temporary version of each value is created and used for the expression—the original values remain unchanged**).

Data type	printf conversion specification	scanf conversion specification
long double	%Lf	%Lf
double	%f	%lf
float	%f	%f
unsigned long int	%lu	%lu
long int	%ld	%ld
unsigned int	%u	%u
int	%d	%d
unsigned short	%hu	%hu
short	%hd	%hd
char	%c	%c

**Fig. 5.5** | Promotion hierarchy for data types.

## 5.6 Function Definitions (Cont.)

- Converting values to lower types normally results in an incorrect value.
- Therefore, 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**.
- Function argument values are converted to the parameter types in a function prototype as if they were being assigned directly to variables of those types.
- If our `square` function that uses an integer parameter (Fig. 5.3) is called with a floating-point argument, the argument is converted to `int` (a lower type), and `square` usually returns an incorrect value.
  - For example, `square( 4.5 )` returns 16, not 20.25.

## 5.7 Function Call Stack and Activation Records

- To understand how C performs function calls, we first need to consider a data structure (i.e., collection of related data items) known as a **stack**.
  - can 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 always removed from the top (referred to as **popping** the dish off 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.

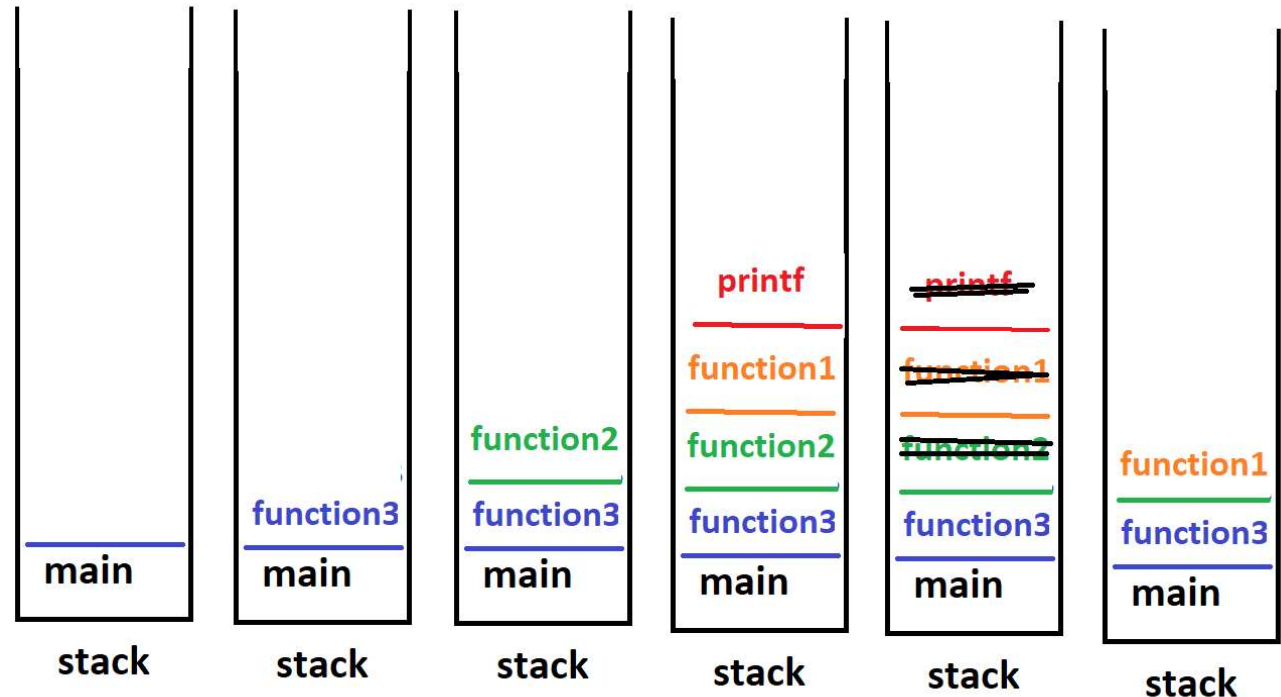
## 5.7 Function Call Stack and Activation Records (Cont.)

- When a program calls a function, the called function must know how to return to its caller, so the return address of the calling function is pushed onto the **program execution stack** (sometimes referred to as the **function call stack**).
- If a series of function calls occurs, the successive return addresses are pushed onto the stack in last-in, first-out order so that each function can return to its caller.
- The program execution stack also contains the memory for the local variables used in each invocation of a function during a program's execution.

```

5 void function1(){
6     printf("function1.");
7 }
8 void function2(){
9     function1();
10    printf("function2.");
11 }
12 void function3(){
13     function2();
14     printf("function3.");
15     function1();
16 }
17 int main(){
18     function3();
19     printf("main.");
20     return 0;
21 }

```



- This data, stored as a portion of the program execution stack, is known as the **activation record** or **stack frame** of the function call.
- When a function call is made, the activation record for that function call is pushed onto the program execution stack.
- When the function returns to its caller, the activation record for this function call is popped off the stack and those local variables are no longer known to the program.

## 5.7 Function Call Stack and Activation Records (Cont.)

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



## 5.8 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.

Header	Explanation
<code>&lt;assert.h&gt;</code>	Contains macros and information for adding diagnostics that aid program debugging.
<code>&lt;ctype.h&gt;</code>	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.
<code>&lt;errno.h&gt;</code>	Defines macros that are useful for reporting error conditions.
<code>&lt;float.h&gt;</code>	Contains the floating-point size limits of the system.
<code>&lt;limits.h&gt;</code>	Contains the integral size limits of the system.
<code>&lt;locale.h&gt;</code>	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 like dates, times, dollar amounts and large numbers throughout the world.
<code>&lt;math.h&gt;</code>	Contains function prototypes for math library functions.
<code>&lt;setjmp.h&gt;</code>	Contains function prototypes for functions that allow bypassing of the usual function call and return sequence.

**Fig. 5.6** | Some of the standard library headers. (Part 1 of 2.)

Header	Explanation
<code>&lt;signal.h&gt;</code>	Contains function prototypes and macros to handle various conditions that may arise during program execution.
<code>&lt;stdarg.h&gt;</code>	Defines macros for dealing with a list of arguments to a function whose number and types are unknown.
<code>&lt;stddef.h&gt;</code>	Contains common type definitions used by C for performing calculations.
<code>&lt;stdio.h&gt;</code>	Contains function prototypes for the standard input/output library functions, and information used by them.
<code>&lt;stdlib.h&gt;</code>	Contains function prototypes for conversions of numbers to text and text to numbers, memory allocation, random numbers, and other utility functions.
<code>&lt;string.h&gt;</code>	Contains function prototypes for string-processing functions.
<code>&lt;time.h&gt;</code>	Contains function prototypes and types for manipulating the time and date.

**Fig. 5.6** | Some of the standard library headers. (Part 2 of 2.)

## 5.9 Calling Functions By Value and By Reference

- There are two ways to invoke functions in many programming languages—  
    call-by-value and  
    call-by-reference.
- When arguments are **passed 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. This prevents the accidental **side effects** (variable modifications) that so greatly hinder the development of correct and reliable software systems. . Call-by-value should be used whenever the called function does not need to modify the value of the caller's original variable.
- When an argument is **passed by reference**, the caller allows the called function to **modify the original** variable's value. Call-by-reference should be used only with **trusted called functions** that need to modify the original variable.

## 5.10 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 (i.e., 16-bit) integer.

## 5.10 Random Number Generation (Cont.)

- If `rand` truly produces integers 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 is 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 integers from 1 to 6.

## 5.10 Random Number Generation (Cont.)

- To demonstrate `rand`, let's develop a program to simulate 20 rolls of a six-sided die and print the value of each roll.
- We use the remainder operator (%) in conjunction with `rand` as follows  
`rand() % 6`
- to produce integers in the range 0 to 5. (is called **scaling**)
- We then **shift** the range of numbers produced by adding 1 to our previous result.

---

```

1  /* Fig. 5.7: fig05_07.c
2     Shifted, scaled integers produced by 1 + rand() % 6 */
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  /* function main begins program execution */
7  int main( void )
8  {
9     int i; /* counter */
10
11    /* loop 20 times */
12    for ( i = 1; i <= 20; i++ ) {
13
14        /* pick random number from 1 to 6 and output it */
15        printf( "%10d", 1 + ( rand() % 6 ) );
16
17        /* if counter is divisible by 5, begin new line of output */
18        if ( i % 5 == 0 ) {
19            printf( "\n" );
20        } /* end if */
21    } /* end for */
22
23    return 0; /* indicates successful termination */
24 } /* end main */

```

---

**Fig. 5.7** | Shifted, scaled random integers produced by  $1 + \text{rand}() \% 6$ . (Part 1 of 2.)

6	6	5	5	6
5	1	1	5	3
6	6	2	4	2
6	2	3	4	1

**Fig. 5.7** | Shifted, scaled random integers produced by  $1 + \text{rand}() \% 6$ . (Part 2 of 2.)



## 5.10 Random Number Generation (Cont.)

- To show that these numbers occur approximately with equal likelihood, let's simulate 6000 rolls of a die
  - Each integer from 1 to 6 should appear approximately 1000 times.

---

```
1  /* Fig. 5.8: fig05_08.c
2     Roll a six-sided die 6000 times */
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  /* function main begins program execution */
7  int main( void )
8  {
9      int frequency1 = 0; /* rolled 1 counter */
10     int frequency2 = 0; /* rolled 2 counter */
11     int frequency3 = 0; /* rolled 3 counter */
12     int frequency4 = 0; /* rolled 4 counter */
13     int frequency5 = 0; /* rolled 5 counter */
14     int frequency6 = 0; /* rolled 6 counter */
15
16     int roll; /* roll counter, value 1 to 6000 */
17     int face; /* represents one roll of the die, value 1 to 6 */
18
```

---

**Fig. 5.8** | Rolling a six-sided die 6000 times. (Part I of 4.)

```

19  /* loop 6000 times and summarize results */
20  for ( roll = 1; roll <= 6000; roll++ ) {
21      face = 1 + rand() % 6; /* random number from 1 to 6 */
22
23      /* determine face value and increment appropriate counter */
24      switch ( face ) {
25
26          case 1: /* rolled 1 */
27              ++frequency1;
28              break;
29
30          case 2: /* rolled 2 */
31              ++frequency2;
32              break;
33
34          case 3: /* rolled 3 */
35              ++frequency3;
36              break;
37
38          case 4: /* rolled 4 */
39              ++frequency4;
40              break;
41
42          case 5: /* rolled 5 */
43              ++frequency5;
44              break;
45
46          case 6: /* rolled 6 */
47              ++frequency6;
48              break; /* optional */
49      } /* end switch */
50  } /* end for */
51
52  /* display results in tabular format */
53  printf( "%5s%13s\n", "Face", "Frequency" );
54  printf( " 1%13d\n", frequency1 );
55  printf( " 2%13d\n", frequency2 );
56  printf( " 3%13d\n", frequency3 );
57  printf( " 4%13d\n", frequency4 );
58  printf( " 5%13d\n", frequency5 );
59  printf( " 6%13d\n", frequency6 );
60  return 0; /* indicates successful termination */
61 } /* end main */

```

Face	Frequency
1	1003
2	1017
3	983
4	994
5	1004
6	999

**Fig. 5.8** | Rolling a six-sided die 6000 times. (Part 4 of 4.)

## 5.10 Random Number Generation (Cont.)

- Executing the program of Fig. 5.7 **again produces exactly the same sequence of values.**
- How can these be random numbers? Ironically, this repeatability is an important characteristic of function `rand`.

## 5.10 Random Number Generation (Cont.)

- 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. )

## 5.10 Random Number Generation (Cont.)

- This is called **randomizing** and is accomplished with the standard library function `srand`.
- Function `srand` takes an **unsigned** integer argument and **seeds** function `rand` to produce a different sequence of random numbers for each execution of the program.
  - The function prototype for `srand` is found in `<stdlib.h>`.
  - The conversion specifier `%u` is used to read an **unsigned** value with `scanf`.

```

2      Randomizing die-rolling program */
3  #include <stdlib.h>
4  #include <stdio.h>
5
6  /* function main begins program execution */
7  int main( void )
8  {
9      int i; /* counter */
10     unsigned seed; /* number used to seed random number generator */
11
12     printf( "Enter seed: " );
13     scanf( "%u", &seed ); /* note %u for unsigned */
14
15     srand( seed ); /* seed random number generator */
16
17     /* loop 10 times */
18     for ( i = 1; i <= 10; i++ ) {
19
20         /* pick a random number from 1 to 6 and output it */
21         printf( "%10d", 1 + ( rand() % 6 ) );
22
23         /* if counter is divisible by 5, begin a new line of output */
24         if ( i % 5 == 0 ) {
25             printf( "\n" );
26         } /* end if */
27     } /* end for */
28
29     return 0; /* indicates successful termination */
30 } /* end main */

```

Enter seed: 67					
6	1	4	6	2	
1	6	1	6	4	

Enter seed: 867					
2	4	6	1	6	
1	1	3	6	2	

Enter seed: 67					
6	1	4	6	2	
1	6	1	6	4	

## 5.10 Random Number Generation (Cont.)

- 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.
- Function `time` takes `NULL` as an argument (`time` is capable of providing you with a string representing the value it returns; `NULL` disables this capability for a specific call to `time`).
- The function prototype for `time` is in `<time.h>`.



## 5.10 Random Number Generation (Cont.)

- Produce a range of random numbers
- the width of the range is determined by the number used to scale `rand` with the remainder operator (i.e., 6),
- and the starting number of the range is equal to the number (i.e., 1) that is added to `rand % w`.
- We can generalize this result as follows
$$n = a + \text{rand}() \% b;$$
  - where `a` is the **shifting value** (which is equal to the first number in the desired range of consecutive integers) and `b` is the scaling factor (which is equal to the width of the desired range of consecutive integers).

- How to produce a random number between 11 and 31?

- How to produce a random number between 11 and 31?

$a + \text{rand}() \% b$

$11 + \text{rand}() \% 21$

## 5.11 Example: A Game of Chance

- Consider Following Game:
  - 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 of the 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.

```

3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <time.h> /* contains prototype for function time */
6
7  /* enumeration constants represent game status */
8  enum Status { CONTINUE, WON, LOST };
9
10 int rollDice( void ); /* function prototype */
11
12 /* function main begins program execution */
13 int main( void )
14 {
15     int sum; /* sum of rolled dice */
16     int myPoint; /* point earned */
17
18     enum Status gameStatus; /* can contain CONTINUE, WON, or LOST */
19
20     /* randomize random number generator using current time */
21     srand( time( NULL ) );
22
23     sum = rollDice(); /* first roll of the dice */

```

```

24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47

```

```

/* determine game status based on sum of dice */
switch( sum ) {

    /* win on first roll */
    case 7:
    case 11:
        gameStatus = WON;
        break;

    /* lose on first roll */
    case 2:
    case 3:
    case 12:
        gameStatus = LOST;
        break;

    /* remember point */
    default:
        gameStatus = CONTINUE;
        myPoint = sum;
        printf( "Point is %d\n", myPoint );
        break; /* optional */
} /* end switch */

```

Player rolled 5 + 6 = 11  
Player wins

Player rolled 4 + 1 = 5  
Point is 5  
Player rolled 6 + 2 = 8  
Player rolled 2 + 1 = 3  
Player rolled 3 + 2 = 5  
Player wins

Player rolled 1 + 1 = 2  
Player loses

Player rolled 6 + 4 = 10  
Point is 10  
Player rolled 3 + 4 = 7  
Player loses

```

48
49  /* while game not complete */
50  while ( gameStatus == CONTINUE ) {
51      sum = rollDice(); /* roll dice again */
52
53      /* determine game status */
54      if ( sum == myPoint ) { /* win by making point */
55          gameStatus = WON; /* game over, player won */
56      } /* end if */
57      else {
58          if ( sum == 7 ) { /* lose by rolling 7 */
59              gameStatus = LOST; /* game over, player lost */
60          } /* end if */
61      } /* end else */
62  } /* end while */
63
64  /* display won or lost message */
65  if ( gameStatus == WON ) { /* did player win? */
66      printf( "Player wins\n" );
67  } /* end if */
68  else { /* player lost */
69      printf( "Player loses\n" );
70  } /* end else */
71
72  return 0; /* indicates successful termination */
73 } /* end main */
74

```

Player rolled 5 + 6 = 11  
Player wins

Player rolled 4 + 1 = 5  
Point is 5  
Player rolled 6 + 2 = 8  
Player rolled 2 + 1 = 3  
Player rolled 3 + 2 = 5  
Player wins

Player rolled 1 + 1 = 2  
Player loses

Player rolled 6 + 4 = 10  
Point is 10  
Player rolled 3 + 4 = 7  
Player loses

```

75  /* roll dice, calculate sum and display results */
76  int rollDice( void )
77  {
78      int die1; /* first die */
79      int die2; /* second die */
80      int workSum; /* sum of dice */
81
82      die1 = 1 + ( rand() % 6 ); /* pick random die1 value */
83      die2 = 1 + ( rand() % 6 ); /* pick random die2 value */
84      workSum = die1 + die2; /* sum die1 and die2 */
85
86      /* display results of this roll */
87      printf( "Player rolled %d + %d = %d\n", die1, die2, workSum );
88      return workSum; /* return sum of dice */
89  } /* end function rollDice */

```

## 5.11 Example: A Game of Chance (Conf.)

### enum

- Variable `gameStatus`, defined to be of a new type—`enum Status`—stores the current status.
- Line 8 creates a programmer-defined type called an `enumeration`.
- An enumeration, introduced by the keyword `enum`, is a set of integer constants represented by identifiers.
- `Enumeration constants` are sometimes called symbolic constants.
- Values in an `enum` start with `0` and are incremented by `1`.
- In line 8, the constant `CONTINUE` has the value `0`, `WON` has the value `1` and `LOST` has the value `2`.
- It's also possible to assign an integer value to each identifier in an `enum`.
- The identifiers in an enumeration must be unique, but the values may be duplicated.
- *enum cover completely later!*

## 5.12 Storage Classes

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

An identifier's **scope** is where the identifier can be referenced in a program.



## 5.12 Global variables

- 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.

## 5.12 Static local variables

- Local variables declared with the keyword `static` are known **only in the function in which they're defined**,
- `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 `count` to be `static` and to be initialized to 1.
  - `static int count = 1;`
- All numeric variables of static storage duration are initialized to zero if you do not explicitly initialize them.

## 5.13 Scope Rules

- 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 (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.
  - Labels are used in **switch** statements (as **case** labels) and in **goto** statements.

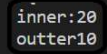
## 5.13 Scope Rules (Cont.)

- 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.

## 5.13 Scope Rules (Cont.)

- 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.
  - Any block may contain variable definitions.
- 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.

```
5  int main(){
6      int i = 10;
7      {
8          int i = 20;
9          printf("inner:%d\n",i);
10     }
11     printf("outter%d",i);
12     return 0;
13 }
```



## 5.13 Scope Rules (Cont.)

- Local variables declared `static` still have block scope, even though they exist from the time the program begins execution.
- The only identifiers with **function-prototype scope** are those used in the parameter list of a function prototype.
- If a name is used in the parameter list of a function prototype, the compiler ignores the name.

scoping issues with global variables, automatic local variables, and **static** local variables

```
2  A scoping example
3  #include <stdio.h>
4
5  void useLocal( void ); /* function prototype */
6  void useStaticLocal( void ); /* function prototype */
7  void useGlobal( void ); /* function prototype */
8
9  int x = 1; /* global variable */
10
11 /* function main begins program execution */
12 int main( void )
13 {
14     int x = 5; /* local variable to main */
15
16     printf("local x in outer scope of main is %d\n", x );
17
18     { /* start new scope */
19         int x = 7; /* local variable to new scope */
20
21         printf( "local x in inner scope of main is %d\n", x );
22     } /* end new scope */
23 }
```

```
24     printf( "local x in outer scope of main is %d\n", x );
25
26     useLocal(); /* useLocal has automatic local x */
27     useStaticLocal(); /* useStaticLocal has static local x */
28     useGlobal(); /* useGlobal uses global x */
29     useLocal(); /* useLocal reinitializes automatic local x */
30     useStaticLocal(); /* static local x retains its prior value */
31     useGlobal(); /* global x also retains its value */
32
33     printf( "\nlocal x in main is %d\n", x );
34     return 0; /* indicates successful termination */
35 } /* end main */
36
37 /* useLocal reinitializes local variable x during each call */
38 void useLocal( void )
39 {
40     int x = 25; /* initialized each time useLocal is called */
41
42     printf( "\nlocal x in useLocal is %d after entering useLocal\n", x );
43     x++;
44     printf( "local x in useLocal is %d before exiting useLocal\n", x );
45 } /* end function useLocal */
46
```

```

47  /* useStaticLocal initializes static local variable x only the first time
48     the function is called; value of x is saved between calls to this
49     function */
50  void useStaticLocal( void )
51  {
52      /* initialized only first time useStaticLocal is called */
53      static int x = 50;
54
55      printf( "\nlocal static x is %d on entering useStaticLocal\n", x );
56      x++;
57      printf( "local static x is %d on exiting useStaticLocal\n", x );
58  } /* end function useStaticLocal */
59
60  /* function useGlobal modifies global variable x during each call */
61  void useGlobal( void )
62  {
63      printf( "\nglobal x is %d on entering useGlobal\n", x );
64      x *= 10;
65      printf( "global x is %d on exiting useGlobal\n", x );
66  } /* end function useGlobal */

```

```

local x in outer scope of main is 5
local x in inner scope of main is 7
local x in outer scope of main is 5

```

```

local x in useLocal is 25 after entering useLocal
local x in useLocal is 26 before exiting useLocal

```

```

local static x is 50 on entering useStaticLocal
local static x is 51 on exiting useStaticLocal

```

```

global x is 1 on entering useGlobal
global x is 10 on exiting useGlobal

```

```

local x in useLocal is 25 after entering useLocal
local x in useLocal is 26 before exiting useLocal

```

```

local static x is 51 on entering useStaticLocal
local static x is 52 on exiting useStaticLocal

```

```

global x is 10 on entering useGlobal
global x is 100 on exiting useGlobal

```

```

local x in main is 5

```



## 5.13 Scope Rules (Cont.)

- This global variable is hidden in any block (or function) in which a variable named `x` is defined.
- The variable `x` with value 7 is automatically destroyed when the block is exited,
- the local variable `x` in the outer block of `main` is printed again to show that it's no longer hidden.
- Each time this function is called, automatic variable `x` is reinitialized to 25.
- Local variables declared as `static` retain their values even when they're out of scope.
- In the next call to this function, `static` local variable `x` will contain the value 51.