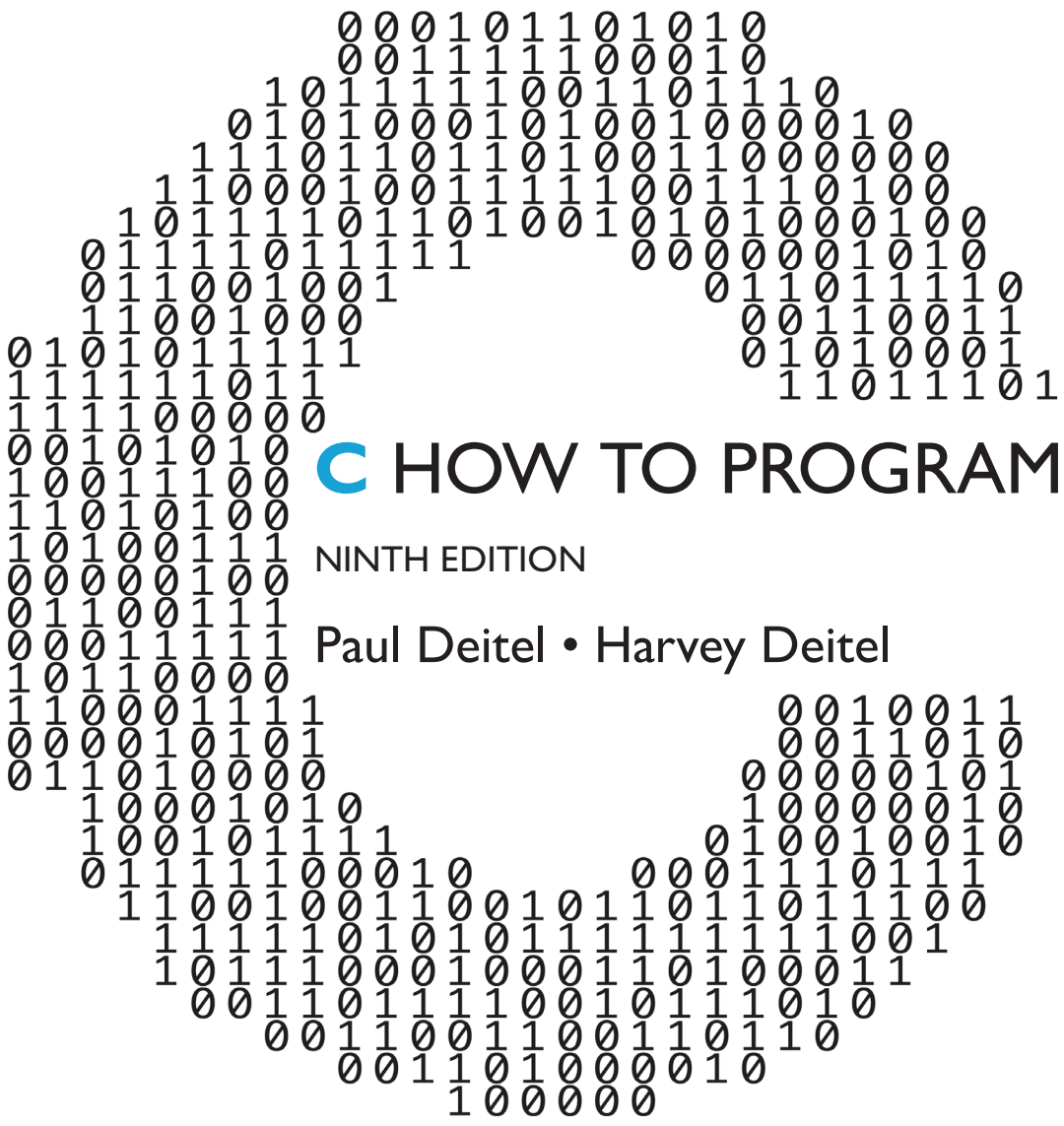
GLOBAL EDITION



C HOW TO PROGRAM

NINTH EDITION

Paul Deitel • Harvey Deitel

with case studies introducing

Applications Programming and

Systems Programming

DEITEL®

HOW TO PROGRAM

NINTH EDITION

GLOBAL EDITION

with

Case Studies Introducing

Applications Programming and

Systems Programming

PAUL DEITEL
HARVEY DEITEL

2 *(Fill-In)* The _____ header contains information for adding diagnostics that aid program debugging.
Answer: `<assert.h>`.

5.9 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 an argument is passed by value, a copy of the argument's value is made and passed to the 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 accidental **side effects** (variable modifications) that can 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. In **Chapter 7, Pointers**, we'll show how to achieve pass-by-reference. In Chapter 6, we'll see that array arguments are automatically passed by reference for performance reasons. We'll see in Chapter 7 that this is not a contradiction. For now, we concentrate on pass-by-value.

Self Check

1 *(True/False)* When an argument is passed by value, a copy of the argument's value is made and passed to the function. Changes to the copy also are applied to the original variable's value in the caller.
Answer: *False*. With pass-by-value, changes to the copy do not affect an original variable's value in the caller.

2 *(True/False)* Pass-by-reference should be used only with *trusted* called functions that need to modify the original variable.
Answer: *True*.

5.10 Random-Number Generation

We now take a brief and, hopefully, entertaining diversion into *simulation* and *game playing*. In this and the next section, we'll develop a nicely structured game-playing program that includes multiple custom functions. The program uses functions and several of the control statements we've studied. The *element of chance* can be introduced into computer applications by using the C standard library function `rand`[4] from the `<stdlib.h>` header.

4. C standard library function `rand` is known to be "predictable," which can create security breach opportunities. Each of our preferred platforms offers a non-standard secure random-number generator. We'll mention these in Section 5.17, Secure C Programming—Secure Random-Number Generation.

Obtaining a Random Integer Value

Consider the following statement:

```
int value = rand();
```

The `rand` function generates an integer between 0 and `RAND_MAX` (a symbolic constant defined in the `<stdlib.h>` header). The C standard states that `RAND_MAX`'s value must be at least 32,767, which is the maximum value for a two-byte (i.e., 16-bit) integer. The programs in this section were tested on Microsoft Visual C++ with a maximum `RAND_MAX` value of 32,767, and on GNU `gcc` and Xcode Clang with a maximum `RAND_MAX` value of 2,147,483,647. 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'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 integers from 1 to 6.

Rolling a Six-Sided Die

To demonstrate `rand`, let's develop a program (Fig. 5.4) to simulate 10 rolls of a six-sided die and print each roll's value.

```
1  // fig05_04.c
2  // Shifted, scaled random integers produced by 1 + rand() % 6.
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  int main(void) {
7
8     for (int i = 1; i <= 10; ++i) {
9        printf("%d  ", 1 + (rand() % 6)); // display random die value
10    }
11
12    puts("");
13 }
```

```
6  6  5  5  6  5  1  1  5  3
```

Fig. 5.4 | Shifted, scaled random integers produced by `1 + rand() % 6`.

The `rand` function's prototype is in `<stdlib.h>`. In line 9, we use the remainder 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 1 to our previous result. The output confirms that the results are in the range 1 to 6—the order in which these random values are chosen might vary by compiler.

Rolling a Six-Sided Die 60,000,000 Times

To show that these numbers occur approximately with *equal likelihood*, let's simulate 60,000,000 rolls of a die with the program of Fig. 5.5. Each integer from 1 to 6 should appear approximately 10,000,000 times.

```
1  // fig05_05.c
2  // Rolling a six-sided die 60,000,000 times.
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  int main(void) {
7     int frequency1 = 0; // rolled 1 counter
8     int frequency2 = 0; // rolled 2 counter
9     int frequency3 = 0; // rolled 3 counter
10    int frequency4 = 0; // rolled 4 counter
11    int frequency5 = 0; // rolled 5 counter
12    int frequency6 = 0; // rolled 6 counter
13
14    // loop 60000000 times and summarize results
15    for (int roll = 1; roll <= 60000000; ++roll) {
16       int face = 1 + rand() % 6; // random number from 1 to 6
17
18       // determine face value and increment appropriate counter
19       switch (face) {
20          case 1: // rolled 1
21             ++frequency1;
22             break;
23          case 2: // rolled 2
24             ++frequency2;
25             break;
26          case 3: // rolled 3
27             ++frequency3;
28             break;
29          case 4: // rolled 4
30             ++frequency4;
31             break;
32          case 5: // rolled 5
33             ++frequency5;
34             break;
35          case 6: // rolled 6
36             ++frequency6;
37             break; // optional
38       }
39    }
40
41    // display results in tabular format
42    printf("%s%13s\n", "Face", "Frequency");
43    printf("   1%13d\n", frequency1);
44    printf("   2%13d\n", frequency2);
45    printf("   3%13d\n", frequency3);
46    printf("   4%13d\n", frequency4);
```

Fig. 5.5 | Rolling a six-sided die 60,000,000 times. (Part 1 of 2.)

```
47     printf("   5%13d\n", frequency5);
48     printf("   6%13d\n", frequency6);
49  }
```

```
Face    Frequency
   1      9999294
   2     10002929
   3      9995360
   4     10000409
   5     10005206
   6      9996802
```

Fig. 5.5 | Rolling a six-sided die 60,000,000 times. (Part 2 of 2.)

As the program output shows, by scaling and shifting, we've used the `rand` function to realistically simulate the rolling of a six-sided die. Note the use of the `%s` conversion specification to print the character strings `"Face"` and `"Frequency"` as column headers (line 42). After we study arrays in Chapter 6, we'll show how to replace this 20-line `switch` statement elegantly with a single-line statement.

Randomizing the Random-Number Generator

Executing the program of Fig. 5.4 again produces

```
6  6  5  5  6  5  1  1  5  3
```

This is the exact sequence of values we showed in Fig. 5.4. How can these be random numbers? Ironically, this *repeatability* is an important characteristic of function `rand`. When debugging a program, this repeatability is essential for proving that corrections to a program work properly.

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 `int` argument and **seeds** function `rand` to produce a different sequence of random numbers for each program execution.

We demonstrate function `srand` in Fig. 5.6. The function prototype for `srand` is found in `<stdlib.h>`.

```
1  // fig05_06.c
2  // Randomizing the die-rolling program.
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  int main(void) {
7     printf("%s", "Enter seed: ");
```

Fig. 5.6 | Randomizing the die-rolling program. (Part 1 of 2.)

```
 8     int seed = 0; // number used to seed the random-number generator
 9     scanf("%d", &seed);
10
11     srand(seed); // seed the random-number generator
12
13     for (int i = 1; i <= 10; ++i) {
14        printf("%d  ", 1 + (rand() % 6)); // display random die value
15     }
16
17     puts("");
18  }
```

```
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
```

Fig. 5.6 | Randomizing the die-rolling program. (Part 2 of 2.)

Let's run the program several times and observe the results. Notice that a different sequence of random numbers is obtained each time the program is run, provided that a different seed is supplied. The first and last outputs use the same seed value, so they show the same results.

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 integer and used as the seed to the random-number generator. The function prototype for `time` is in `<time.h>`. We'll say more about `NULL` in Chapter 7.

Generalized Scaling and Shifting of Random Numbers

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:

```
int face = 1 + rand() % 6;
```

This statement always assigns an integer value (at random) to the variable `face` in the range $1 \leq \texttt{face} \leq 6$. The *width* of this range (i.e., the number of consecutive integers in the range) is 6, and the starting number in the range is 1. Referring 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 (i.e., 6), and the starting number of the range is equal to the number (i.e., 1) that's added to `rand % 6`. We can generalize this result as follows:

```
int n = a + 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).

In the exercises, you'll choose integers at random from sets of values other than ranges of consecutive integers.

Self Check

1 ***(Fill-In)*** _____ is an important characteristic of function `rand`. When we're debugging a program, this characteristic is essential for proving that corrections to a program work properly.
Answer: Repeatability.

2 ***(True/False)*** 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.
Answer: *True*.

3 ***(Fill-In)*** 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 _____.
Answer: `srand`.

5.11 Game Simulation Case Study: Rock, Paper, Scissors

In this section, we simulate a popular hand game; rock, paper, scissors. The game is played between two people. The rules of the game are straightforward:

> *Each player simultaneously forms one of three shapes with an outstretched hand. The shapes are "rock", "paper", and "scissors". A simultaneous, zero-sum game, it has three possible outcomes: a draw, a win, or a loss. "rock" breaks (wins) "scissors", "scissors" cuts (wins) "paper", and "paper" covers (wins) "rock". If both players choose the same shape, the game is a tie.*

Figure 5.7 simulates the game and shows several sample executions.