Topic 3
Lecture 3a
Functions and an Introduction to Recursion

CSCI 140: C++ Language & Objects

Prof. Dominick Atanasio

Book: C++: How to Program 10 ed.

Agenda

- Construct programs modularly from functions.
- Use common math library functions.
- Use function prototypes to declare functions.
- Use random-number generation to implement game playing applications.
- Use C++14 digit separators to make numeric literals more readable.
- Visibility of identifiers (Scope).
- Function call/return mechanism
- Passing data to functions and returning results.
- Inline function, references, and default arguments.
- Overloaded Functions.
- Function templates.
- Recursive functions.

6.1 Introduction

- Develop and maintain a large program by constructing it from small, simple pieces, or components.
  - divide and conquer.
- Emphasize how to declare and use functions to facilitate the design, implementation, operation and maintenance of large programs.
- Function prototypes and how the compiler uses them to convert the type of an argument in a function call to the type specified in a function's parameter list, if necessary.
- Simulation techniques with random number generation.
- C++'s scope rules.

6.1 Introduction (cont.)

- How C++ keeps track of which function is currently executing, how parameters and other local variables of functions are maintained in memory and how a function knows where to return after it completes execution.
- Function overloading.
- Function templates.
- Recursion.

6.2 Program Components in C++

- C++ programs are typically written by combining “prepackaged” functions and classes available in the C++ Standard Library with new functions and classes you write.
- The C++ Standard Library provides a rich collection of functions.
- Functions allow you to modularize a program by separating its tasks into self-contained units.

Software Engineering Observation 6.1

To promote software reusability, every function should be limited to performing a single, well-defined task, and the name of the function should express that task effectively.

6.3 Math Library Functions

- Sometimes functions are not members of a class.
  - Called global functions.
- The <cmath> header provides a collection of functions that enable you to perform common mathematical calculations.
- All functions in the <cmath> header are global functions—therefore, each is called simply by specifying the name of the function followed by parentheses containing the function's arguments.

6.3 Math Library Functions (cont.)

- Some math library functions are summarized in Fig. 6.2.
  - In the figure, the variables x and y are of type double.

Function	Description	Example
`ceil(x)`	rounds x to the smallest integer not less than x	`ceil(9.2)` is `10.0` `ceil(-9.8)` is `-9.0`
`cos(x)`	trigonometric cosine of x (x in radians)	`cos(0.0)` is `1.0`
`exp(x)`	exponential function e^x	`exp(1.0)` is `2.718282` `exp(2.0)` is `7.389056`

`fabs(x)`	absolute value of x	`fabs(5.1)` is `5.1`
		`fabs(0.0)` is `0.0`
		`fabs(-8.76)` is `8.76`

`floor(x)`	rounds x to the largest integer not greater than x	`floor(9.2)` is `9.0` `floor(-9.8)` is `-10.0`
`fmod(x, y)`	remainder of x/y as a floating-point number	`fmod(2.6, 1.2)` is `0.2`
`log(x)`	natural logarithm of x (base e)	`log(2.718282)` is `1.0` `log(7.389056)` is `2.0`
`log10(x)`	logarithm of x (base 10)	`log10(10.0)` is `1.0` `log10(100.0)` is `2.0`
`pow(x, y)`	x raised to power y (x^y)	`pow(2, 7)` is `128` `pow(9, .5)` is `3`

Fig. 6.2 | Math library functions. (Part 1 of 2.)

Function	Description	Example
`sin(x)`	trigonometric sine of x (x in radians)	`sin(0.0)` is `0`
`sqrt(x)`	square root of x (where x is a nonnegative value)	`sqrt(9.0)` is `3.0`
`tan(x)`	trigonometric tangent of x (x in radians)	`tan(0.0)` is `0`

Fig. 6.2 | Math library functions. (Part 2 of 2.)

6.4 Function Prototypes

- In preceding chapters, we created classes with various member functions.
- We defined each class in a header (.h) and included it before main in a program's source-code file.
- Doing this ensures that the class (and thus its member functions) is defined before main creates and manipulates objects of that class.
- The compiler then can ensure that we call each class's constructors and member functions correctly—for example, passing each the correct number and types of arguments.

6.4 Function Prototypes (cont.)

- For a function not defined in a class, you must either define the function before using it or you must declare that the function exists, as we do in line 7 of Fig. 6.3:
  - int maximum(int x, int y, int z); // function prototype
- This is a function prototype, which describes the maximum function without revealing its implementation.
- A function prototype is a declaration of a function
  - tells the compiler the function's name, its return type and the types of its parameters.
- This function prototype indicates that the function returns an int, has the name maximum and requires three int parameters to perform its task.
- The function prototype is the same as the first line of the corresponding function definition (line 20) but ends with a required semicolon.

6.4 Function Prototypes (cont.)

- When compiling the program, the compiler uses the prototype to
  - Ensure that maximum's first line matches its prototype.
  - Check that the call to maximum contains the correct number and types of arguments, and that the types of the arguments are in the correct order.
  - Ensure that the value returned by the function can be used correctly in the expression that called the function.
  - Ensure that each argument is consistent with the type of the corresponding parameter.
    - If the arguments passed to a function do not match the types specified in the function's prototype, the compiler attempts to convert the arguments to those types.

Common Programming Error 6.2

Compilation errors occur if the function prototype, header and calls do not all agree in the number, type and order of arguments and parameters, and in the return type.

6.4 Function Prototypes (cont.)

- Multiple parameters are specified in both the function prototype and the function header as a comma-separated list, as are multiple arguments in a function call.

Portability Tip 6.2

Sometimes when a function's arguments are expressions, such as those with calls to other functions, the order in which the compiler evaluates the arguments could affect the values of one or more of the arguments. If the evaluation order changes between compilers, the argument values passed to the function could vary, causing subtle logic errors.

6.4 Function Definitions with Multiple Parameters (cont.)

- In a function that does not return a result (i.e., it has a void return type), we showed that control returns when the program reaches the function-ending right brace.
- You also can explicitly return control to the caller by executing the statement
  - return;

6.5 Function-Prototypes and Argument-Coercion Notes

- A function prototype is required unless the function is defined before it's used.
- When you use a standard library function like sqrt, you do not have access to the function's definition, therefore it cannot be defined in your code before you call the function.
- Instead, you must include its corresponding header (<cmath>), which contains the function's prototype.

Software Engineering Observation 6.3

Always provide function prototypes, even though it's possible to omit them when functions are defined before *they're used. Providing the prototypes avoids tying the code to the order in which functions are defined (which can easily change as a program evolves).*

6.5.1 Function Signatures and Function Prototypes

- The portion of a function prototype that includes the name of the function and the types of its arguments is called the function signature or simply the signature.
  - Signature does not specify the function's return type.
- The scope of a function is the region of a program in which the function is known and accessible.

6.5.3 Argument-Promotion Rules and Implicit Conversions

- An important feature of function prototypes is argument coercion
  - forcing arguments to the appropriate types specified by the parameter declarations.
  - These conversions occur as specified by C++'s promotion rules.
- The promotion rules indicate how to convert between types without losing data.
- The promotion rules apply to expressions containing values of two or more data types
  - also referred to as mixed-type expressions.
- The type of each value in a mixed-type expression is promoted to the "highest" type in the expression.

6.5.3 Argument-Promotion Rules and Implicit Conversions

- Figure 6.4 lists the arithmetic data types in order from “highest type” to “lowest type.”
- Converting values to lower fundamental types can result in incorrect values.
- Therefore, a value can be converted to a lower fundamental type only by explicitly assigning the value to a variable of lower type or by using a cast operator.

Data types	
`long double`	
`double`	
`float`	
`unsigned long long int`	(synonymous with `unsigned long long`)
`long long int`	(synonymous with `long long`)
`unsigned long int`	(synonymous with `unsigned long`)
`long int`	(synonymous with `long`)
`unsigned int`	(synonymous with `unsigned`)
`int`	
`unsigned short int`	(synonymous with `unsigned short`)
`short int`	(synonymous with `short`)
`unsigned char`	
`char` and `signed char`	
`bool`	

Fig. 6.4 | Promotion hierarchy for arithmetic data types; the "highest" types are at the top.

Common Programming Error 6.4

A compilation error occurs if the arguments in a function call cannot be implicitly converted to the expected types specified in the function's prototype.

6.6 C++ Standard Library Header Files

- The C++ Standard Library is divided into many portions, each with its own header file.
- The header files contain the function prototypes for the related functions that form each portion of the library.
- The header files also contain definitions of various class types and functions, as well as constants needed by those functions.
- A header file “instructs” the compiler on how to interface with library and user-written components.
- Figure 6.5 lists some common C++ Standard Library header files, most of which are discussed later in the book.

Standard Library header	Explanation
`<iostream>`	Contains function prototypes for the C++ standard input and output functions, introduced in Chapter 2, and is covered in more detail in Chapter 13, Stream Input/Output: A Deeper Look.
`<iomanip>`	Contains function prototypes for stream manipulators that format streams of data. This header is first used in Section 4.10 and is discussed in more detail in Chapter 13, Stream Input/Output: A Deeper Look.
`<cmath>`	Contains function prototypes for math library functions (Section 6.3).
`<cstdlib>`	Contains function prototypes for conversions of numbers to text, text to numbers, memory allocation, random numbers and various other utility functions. Portions of the header are covered in Section 6.7; Chapter 11, Operator Overloading; Class `string`; Chapter 17, Exception Handling: A Deeper Look; Chapter 22, Bits, Characters, C Strings and `structs`; and Appendix F, C Legacy Code Topics.
`<ctime>`	Contains function prototypes and types for manipulating the time and date. This header is used in Section 6.7.

Fig. 6.5 | C++ Standard Library headers. (Part 1 of 4.)

Standard Library header	Explanation
`<array>`, `<vector>`, `<list>`, `<forward_list>`, `<deque>`, `<queue>`, `<stack>`, `<map>`, `<unordered_map>`, `<unordered_set>`, `<set>`, `<bitset>`	These headers contain classes that implement the C++ Standard Library containers. Containers store data during a program's execution. The `<vector>` header is first introduced in Chapter 7, Class Templates `array` and `vector`; Catching Exceptions. We discuss all these headers in Chapter 15, Standard Library Containers and Iterators. `<array>`, `<forward_list>`, `<unordered_map>` and `<unordered_set>` were all introduced in C++11.
`<cctype>`	Contains function prototypes for functions that test characters for certain properties (such as whether the character is a digit or a punctuation), and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa. These topics are discussed in Chapter 22, Bits, Characters, C Strings and `structs`.
`<cstring>`	Contains function prototypes for C-style string-processing functions.
`<typeinfo>`	Contains classes for runtime type identification (determining data types at execution time). This header is discussed in Section 12.9.

Fig. 6.5 | C++ Standard Library headers. (Part 2 of 4.)

11

Standard Library header	Explanation
`<exception>`, `<stdexcept>`	These headers contain classes that are used for exception handling (discussed in Chapter 17, Exception Handling: A Deeper Look).
`<memory>`	Contains classes and functions used by the C++ Standard Library to allocate memory to the C++ Standard Library containers. This header is used in Chapter 17, Exception Handling: A Deeper Look.
`<fstream>`	Contains function prototypes for functions that perform input from and output to files on disk (discussed in Chapter 14, File Processing).
`<string>`	Contains the definition of class `string` from the C++ Standard Library (discussed in Chapter 21, Class `string` and String Stream Processing).
`<sstream>`	Contains function prototypes for functions that perform input from strings in memory and output to strings in memory (discussed in Chapter 21, Class `string` and String Stream Processing).
`<functional>`	Contains classes and functions used by C++ Standard Library algorithms. This header is used in Chapter 16.
`<iterator>`	Contains classes for accessing data in the C++ Standard Library containers. This header is used in Chapter 15.

Fig. 6.5 | C++ Standard Library headers. (Part 3 of 4.)

Standard Library header	Explanation
`<algorithm>`	Contains functions for manipulating data in C++ Standard Library containers. This header is used in Chapter 15.
`<cassert>`	Contains macros for adding diagnostics that aid program debugging. This header is used in Appendix E, Preprocessor.
`<cfloat>`	Contains the floating-point size limits of the system.
`<climits>`	Contains the integral size limits of the system.
`<cstdio>`	Contains function prototypes for the C-style standard input/output library functions.
`<locale>`	Contains classes and functions normally used by stream processing to process data in the natural form for different languages (e.g., monetary formats, sorting strings, character presentation, etc.).
`<limits>`	Contains classes for defining the numerical data type limits on each computer platform—this is C++'s version of `<climits>` and `<cfloat>`.
`<utility>`	Contains classes and functions that are used by many C++ Standard Library headers.

Fig. 6.5 | C++ Standard Library headers. (Part 4 of 4.)

6.7 Case Study: Random Number Generation

- The element of chance can be introduced into computer applications by using the C++ Standard Library function rand.
- The function rand generates an unsigned integer between 0 and RAND_MAX (a symbolic constant defined in the <cstdlib> header file).

rand and srand

- srand and rand are very simple implementations of a Pseudorandom Number Generator (PRNG)
- Here is the sample implementation from the C standard:

```
static unsigned long int next = 1;

int rand(void) // RAND_MAX assumed to be 32767
{
    next = next * 1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}

void srand(unsigned int seed)
{
    next = seed;
}
```

6.7 Case Study: Random Number Generation

- The value of RAND_MAX must be at least 32767—the maximum positive value for a two-byte (16-bit) integer.
- For GNU C++, the value of RAND_MAX is 2147483647; for Visual Studio, the value of RAND_MAX is 32767.
- 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.

6.7.1 Rolling a Six-Sided Die

- The function prototype for the rand function is in <cstdlib>.
- To produce integers in the range 0 to 5, we use the modulo operator (%) with rand:
  - rand() % 6
  - This is called scaling.
  - The number 6 is called the scaling factor. Six values are produced.
- We can shift the range of numbers produced by adding a value.

```
1  // Fig. 6.6: fig06_06.cpp
2  // Shifted, scaled integers produced by 1 + rand() % 6.
3  #include <iostream>
4  #include <iomanip>
5  #include <cstdlib> // contains function prototype for rand
6  using namespace std;
7
8  int main() {
9     // loop 20 times
10    for (unsigned int counter{1}; counter <= 20; ++counter) {
11       // pick random number from 1 to 6 and output it
12       cout << setw(10) << (1 + rand() % 6);
13
14       // if counter is divisible by 5, start a new line of output
15       if (counter % 5 == 0) {
16          cout << endl;
17       }
18    }
19 }
```

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

Fig. 6.6 | Shifted, scaled integers produced by `1 + rand() % 6`.

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

- To show that the numbers produced by rand occur with approximately equal likelihood, Fig. 6.7 simulates 60,000,000 rolls of a die.
- Each integer in the range 1 to 6 should appear approximately 10,000,000 times.

```
1  // Fig. 6.7: fig06_07.cpp
2  // Rolling a six-sided die 60,000,000 times.
3  #include <iostream>
4  #include <iomanip>
5  #include <cstdlib> // contains function prototype for rand
6  using namespace std;
7
8  int main() {
9     unsigned int frequency1{0}; // count of 1s rolled
10    unsigned int frequency2{0}; // count of 2s rolled
11    unsigned int frequency3{0}; // count of 3s rolled
12    unsigned int frequency4{0}; // count of 4s rolled
13    unsigned int frequency5{0}; // count of 5s rolled
14    unsigned int frequency6{0}; // count of 6s rolled
15    int face; // stores each roll of the die
16
17    // summarize results of 60,000,000 rolls of a die
18    for (unsigned int roll{1}; roll <= 60'000'000; ++roll) {
19       face = 1 + rand() % 6; // random number from 1 to 6
20
```

Fig. 6.7 | Rolling a six-sided die 60,000,000 times. (Part 1 of 3.)

```
21          // determine roll value 1-6 and increment appropriate counter
22          switch (face) {
23             case 1:
24                ++frequency1; // increment the 1s counter
25                break;
26             case 2:
27                ++frequency2; // increment the 2s counter
28                break;
29             case 3:
30                ++frequency3; // increment the 3s counter
31                break;
32             case 4:
33                ++frequency4; // increment the 4s counter
34                break;
35             case 5:
36                ++frequency5; // increment the 5s counter
37                break;
38             case 6:
39                ++frequency6; // increment the 6s counter
40                break;
41             default: // invalid value
42                cout << "Program should never get here!";
43          }
44       }
```

Fig. 6.7 | Rolling a six-sided die 60,000,000 times. (Part 2 of 3.)

```
45
46      cout << "Face" << setw(13) << "Frequency" << endl; // output headers
47      cout << "   1" << setw(13) << frequency1
48         << "\n   2" << setw(13) << frequency2
49         << "\n   3" << setw(13) << frequency3
50         << "\n   4" << setw(13) << frequency4
51         << "\n   5" << setw(13) << frequency5
52         << "\n   6" << setw(13) << frequency6 << endl;
53   }
```

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

Fig. 6.7 | Rolling a six-sided die 60,000,000 times. (Part 3 of 3.)

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

- Prior to C++14, you'd represent the integer value 60,000,000 as 60000000 in a program.
- To make numeric literals more readable, C++14 allows you to insert between groups of digits in numeric literals the digit separator ' (a single-quote character)
  - 60'000'000 represents the integer value 60,000,000.

6.7.3 Randomizing the Random-Number Generator with srand

- Executing the program of Fig. 6.6 again produces exactly the same sequence of values.
  - Repeatability is an important characteristic of rand.
  - Can help with testing and debugging.
- rand generates pseudorandom numbers.
  - a sequence of numbers that appears to be random.
  - sequence repeats itself each time the program executes.

Error-Prevention Tip 6.5

When debugging a simulation program, random-number repeatability is essential for proving that corrections to the program work properly.

6.7.3 Randomizing the Random-Number Generator with srand

- Once a program has been debugged, it can be conditioned to produce a different sequence of random numbers for each execution.
- This is called seeding, or randomizing, and is accomplished with the C++ Standard Library function srand.
  - Takes an unsigned integer argument and seeds rand to produce a different sequence of random numbers for each execution.

6.7 Case Study: Random Number Generation (cont.)

- Seeding the Random Number Generator with srand
- Figure 6.8 demonstrates function srand.

```
1  // Fig. 6.8: fig06_08.cpp
2  // Randomizing the die-rolling program.
3  #include <iostream>
4  #include <iomanip>
5  #include <cstdlib> // contains prototypes for functions srand and rand
6  using namespace std;
7
8  int main() {
9     unsigned int seed{0}; // stores the seed entered by the user
10
11    cout << "Enter seed: ";
12    cin >> seed;
13    srand(seed); // seed random number generator
14
```

Fig. 6.8 | Randomizing the die-rolling program. (Part 1 of 3.)

```
15      // loop 10 times
16      for (unsigned int counter{1}; counter <= 10; ++counter) {
17         // pick random number from 1 to 6 and output it
18         cout << setw(10) << (1 + rand() % 6);
19
20         // if counter is divisible by 5, start a new line of output
21         if (counter % 5 == 0) {
22            cout << endl;
23         }
24      }
25   }
```

Fig. 6.8 | Randomizing the die-rolling program. (Part 2 of 3.)

Software Engineering Observation 6.4

Ensure that your program seeds the random-number generator differently *(and* only once*) each time the program executes; otherwise, an attacker would easily be able to determine the sequence of pseudorandom numbers that would be produced.*

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

```
Enter seed: 432
         4          6          3          1          6
         3          1          5          4          2
```

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

Fig. 6.8 | Randomizing the die-rolling program. (Part 3 of 3.)

6.7.4 Seeding the Random-Number Generator with the Current Time

- To randomize without having to enter a seed
  - srand(static_cast<unsigned int>(time(0)));
- Causes the computer to read its clock to obtain the value for the seed.
- Function time (with the argument NULL or an argument of 0 as written in the preceding statement) returns the current time as the number of seconds since January 1, 1970, at midnight Greenwich Mean Time (GMT). This is called the Epoch.
- The function prototype for time is in <ctime>.

6.7.5 Scaling and Shifting Random Numbers

- To produce random numbers in a specific range use:
  - type variableName = shiftingValue + rand() % scalingFactor;
- shiftingValue is equal to the first number in the desired range of consecutive integers
- scalingFactor is equal to the width of the de-sired range of consecutive integers.

6.8 Case Study: Game of Chance; Introducing enum

- Rules of Craps
  - A player rolls two dice. Each die has six faces.
  - 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 roll, the player wins.
    - If the sum is 2, 3 or 12 on the first roll (called “craps”), the player loses (i.e., the “house” wins).
    - If the sum is 4, 5, 6, 8, 9 or 10 on the first roll, then that sum becomes the player’s “point.”
  - To win, continue rolling until you “make your point.”
  - You lose by rolling a 7 before making the point.
- The program in Fig. 6.9 simulates the game.

```
1  // Fig. 6.9: fig06_09.cpp
2  // Craps simulation.
3  #include <iostream>
4  #include <cstdlib> // contains prototypes for functions srand and rand
5  #include <ctime> // contains prototype for function time
6  using namespace std;
7
8  unsigned int rollDice(); // rolls dice, calculates and displays sum
9
10 int main() {
11    // scoped enumeration with constants that represent the game status
12    enum class Status {CONTINUE, WON, LOST}; // all caps in constants
13
14    // randomize random number generator using current time
15    srand(static_cast<unsigned int>(time(0)));
16
17    unsigned int myPoint{0}; // point if no win or loss on first roll
18    Status gameStatus; // can be CONTINUE, WON or LOST
19    unsigned int sumOfDice{rollDice()}; // first roll of the dice
20
```

Fig. 6.9 | Craps simulation. (Part 1 of 5.)

```
21      // determine game status and point (if needed) based on first roll
22      switch (sumOfDice) {
23         case 7: // win with 7 on first roll
24         case 11: // win with 11 on first roll
25            gameStatus = Status::WON;
26            break;
27         case 2: // lose with 2 on first roll
28         case 3: // lose with 3 on first roll
29         case 12: // lose with 12 on first roll
30            gameStatus = Status::LOST;
31            break;
32         default: // did not win or lose, so remember point
33            gameStatus = Status::CONTINUE; // game is not over
34            myPoint = sumOfDice; // remember the point
35            cout << "Point is " << myPoint << endl;
36            break; // optional at end of switch
37      }
38
```

Fig. 6.9 | Craps simulation. (Part 2 of 5.)

```
39      // while game is not complete
40      while (Status::CONTINUE == gameStatus) { // not WON or LOST
41         sumOfDice = rollDice(); // roll dice again
42
43         // determine game status
44         if (sumOfDice == myPoint) { // win by making point
45            gameStatus = Status::WON;
46         }
47         else {
48            if (sumOfDice == 7) { // lose by rolling 7 before point
49               gameStatus = Status::LOST;
50            }
51         }
52      }
53
54      // display won or lost message
55      if (Status::WON == gameStatus) {
56         cout << "Player wins" << endl;
57      }
58      else {
59         cout << "Player loses" << endl;
60      }
61   }
62
```

Fig. 6.9 | Craps simulation. (Part 3 of 5.)

```
63  // roll dice, calculate sum and display results
64  unsigned int rollDice() {
65     int die1{1 + rand() % 6}; // first die roll
66     int die2{1 + rand() % 6}; // second die roll
67     int sum{die1 + die2}; // compute sum of die values
68
69     // display results of this roll
70     cout << "Player rolled " << die1 << " + " << die2
71        << " = " << sum << endl;
72     return sum;
73  }
```

```
Player rolled 2 + 5 = 7
Player wins
```

```
Player rolled 6 + 6 = 12
Player loses
```

Fig. 6.9 | Craps simulation. (Part 4 of 5.)

```
Player rolled 1 + 3 = 4
Point is 4
Player rolled 4 + 6 = 10
Player rolled 2 + 4 = 6
Player rolled 6 + 4 = 10
Player rolled 2 + 3 = 5
Player rolled 2 + 4 = 6
Player rolled 1 + 1 = 2
Player rolled 4 + 4 = 8
Player rolled 4 + 3 = 7
Player loses
```

```
Player rolled 3 + 3 = 6
Point is 6
Player rolled 5 + 3 = 8
Player rolled 4 + 5 = 9
Player rolled 2 + 1 = 3
Player rolled 1 + 5 = 6
Player wins
```

Fig. 6.9 | Craps simulation. (Part 5 of 5.)

6.8 Case Study: Game of Chance; Introducing enum (cont.)

- A scoped enumeration (C++11) introduced by enum class and followed by a type name, is a set of identifiers representing integer constants.
- The values of these enumeration constants start at 0, unless specified otherwise, and increment by 1.
- The identifiers must be unique, but separate enumeration constants can have the same value.
- Variables of an enum class can be assigned only one of the values declared in the enumeration.

Good Programming Practice 6.2

By convention, you should capitalize the first letter of an `enum class`*'s name.*

Good Programming Practice 6.3

Use only uppercase letters in enumeration constant names. This makes these constants stand out in a program and reminds you that enumeration constants are not *variables.*

6.8 Case Study: Game of Chance; Introducing enum (cont.)

- To reference a scoped enum constant, qualify it with the type name and the scope-resolution operator (::), as in Status::CONTINUE.

Error-Prevention Tip 6.6

Qualifying an `enum class`*'s constant with its typename and* `::` *explicitly identifies the constant as being in the scope of the specified* `enum class`*. If another* `enum class` *contains the same identifier for one of its constants, it's always clear which version of the constant is being used, because the typename and* `::` *are required.*

6.8 Case Study: Game of Chance; Introducing enum (cont.)

- Another popular enumeration is
  - enum class Months { JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC };
  - creates user-defined type Months with enumeration constants representing the months of the year.
  - The first value in the preceding enumeration is explicitly set to 1, so the remaining values increment from 1, resulting in the values 1 through 12.
- Any enumeration constant can be assigned an integer value in the enumeration definition.

Error-Prevention Tip 6.7

Use unique values for an `enum`*'s constants to help prevent hard-to-find logic errors.*

6.8 Case Study: Game of Chance; Introducing enum (cont.)

- Prior to C++11, enumerations were defined with enum (unscoped enums)
- Problem: multiple enums may contain the same identifiers.
- Using such enums in the same program can lead to naming collisions and logic errors.

Error-Prevention Tip 6.8t

Use scoped `enum`*s to avoid the potential naming conflicts that can occur with unscoped* `enum` *constants.*

6.8 Case Study: Game of Chance; Introducing enum (cont.)

- Enumeration constants are represented as integers.
- An unscoped enum's underlying integral type depends on its constants' values—the type is guaranteed to be large enough to store the constant values specified.
- A scoped enum's underlying integral type is int.
- You to specify a scoped enum's underlying integral type by following the enum's type name with a colon (:) and the integral type.
- We can specify that the constants in the enum class Status should have type unsigned int, as in
  - enum class Status : unsigned int {CONTINUE, WON, LOST};

Common Programming Error 6.5

A compilation error occurs if an `enum` *constant's value is outside the range that can be represented by the* `enum`*'s underlying type.*

6.9 C++11 Random Numbers

- According to CERT, function rand does not have “good statistical properties” and can be predictable, which makes programs that use rand less secure (CERT guideline MSC30-CPP).
- C++11 provides a new, more secure library of random-number capabilities that can produce nondeterministic random numbers that can’t be predicted.
- Such random-number generators are used in simulations and security scenarios where predictability is undesirable.
  - New capabilities are located in the C++ Standard Library’s <random> header.

6.9 C++11 Random Numbers (cont.)

- For flexibility based on how random numbers are used in programs, C++11 provides many classes that represent various random-number generation engines and distributions.
  - An engine implements a random-number generation algorithm that produce pseudorandom numbers.
  - A distribution controls the range of values produced by an engine, the types of those values (e.g., int, double, etc.) and the statistical properties of the values.
- uniform_int_distribution evenly distributes pseudorandom integers over a specified range of values.

6.9 C++11 Random Numbers (cont.)

- Rolling a Six-Sided Die
- Figure 6.10 uses the default_random_engine and the uniform_int_distribution to roll a six-sided die.
  - default_random_engine object named engine.
  - Its constructor argument seeds the random-number generation engine with the current time.
  - If you don't pass a value to the constructor, the default seed will be used and the program will produce the same sequence of numbers each time it executes.
  - randomInt—a uniform_int_distribution object that produces unsigned int values (as specified by <unsigned int>) in the range 1 to 6 (as specified by the constructor arguments).
  - The expression randomInt(engine) returns one unsigned int value in the range 1 to 6.

6.9 C++11 Random Numbers (cont.)

- The notation <unsigned int> indicates that uniform_int_distribution is a class template.
- In this case, any integer type can be specified in the angle brackets (< and >).
- In Chapter 18, we discuss how to create class templates and various other chapters show how to use existing class templates from the C++ Standard Library.
- For now, you should feel comfortable using class template uniform_int_distribution by mimicking the syntax shown in the example.

```
1  // Fig. 6.10: fig06_10.cpp
2  // Using a C++11 random-number generation engine and distribution
3  // to roll a six-sided die.
4  #include <iostream>
5  #include <iomanip>
6  #include <random> // contains C++11 random number generation features
7  #include <ctime>
8  using namespace std;
9
10 int main() {
11    // use the default random-number generation engine to
12    // produce uniformly distributed pseudorandom int values from 1 to 6
13    default_random_engine engine{static_cast<unsigned int>(time(0))};
14    uniform_int_distribution<unsigned int> randomInt{1, 6};
15
```

Fig. 6.10 | Using a C++11 random-number generation engine and distribution to roll a six-sided die. (Part 1 of 2.)

```
16      // loop 10 times
17      for (unsigned int counter{1}; counter <= 10; ++counter) {
18         // pick random number from 1 to 6 and output it
19         cout << setw(10) << randomInt(engine);
20
21         // if counter is divisible by 5, start a new line of output
22         if (counter % 5 == 0) {
23            cout << endl;
24         }
25      }
26   }
```

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

Fig. 6.10 | Using a C++11 random-number generation engine and distribution to roll a six-sided die. (Part 2 of 2.)