 Answers to review questions from Chapter 2

1. Explain in your own words the difference between a *function* and a *program*.

**A *function* computes a value or performs some operation on behalf of the code for a program. A *program* executes computation on behalf of a user. A program typically consists of many functions.**

2. Define the following terms as they apply to functions: *call*, *argument*, *return*.

**In programming terminology, the process of invoking a function is referred to as *calling* that function. In the process of making a call, the caller can provide data to the function in the form of *arguments,* which are a set of local variables that are initialized from the values written inside the parentheses that designate the call. When the function completes its work, it *returns* to its caller, often passing back a value as a result.**

3. True or false: Every function in a C++ program requires a prototype.

**False. If you define a function before you call it, prototypes are not required. This book defines prototypes for all functions so that they can be defined and called in any order.**

4. What is the prototype of the function **sqrt** in the **<cmath>** library?

double sqrt(double x);

5. Can there be more than one **return** statement in the body of a function?

**Yes. A function can include any number of return statements.**

6. What is a *predicate function?*

**A *predicate function* is a function that returns a Boolean value.**

7. What is meant by the term *overloading?* How does the C++ compiler use *signatures* to implement overloading?

**In C++, the term *overloading* refers to the fact that you can define several different functions with the same name, as long as each function has a different *signature,* which indicates the number and types of the arguments. When the C++ compiler sees a call to an overloaded function with a particular name, it examines the argument values to see which version of that function is required.**

8. How do you specify a default value for a parameter?

**By including an equal sign and the default value after the name of the parameter**

9. True or false: It is possible to specify a default value for the first parameter to a function without specifying a default value for the second.

**False. In C++, default values may appear only at the end of the argument list.**

10. What is a *stack frame?*

**A *stack frame* is the region of memory used to hold the values of local variables during a function call. The frame is created when the function is called and deleted when the function returns.**

11. Describe the process by which *arguments* are associated with *parameters.*

**Arguments and parameters are matched by their order in the argument list. The names of the parameters have no bearing on the association process.**

12. Variables declared within a function are said to be *local variables*. What is the significance of the word *local* in this context?

**Local variables can be used only within that function. Neither the caller nor any function called from inside a function has access to those local variables.**

13. What does the term *call by reference* mean?

**In C++, *call by reference* specifies a sharing relationship between a parameter variable and the corresponding argument in the caller. Changing the value of the parameter inside the function changes that value in the caller as well, since the two values are shared.**

14. How do you indicate *call by reference* in a C++ program?

**By writing an ampersand (&) before the parameter name.**

15. Define the following terms in the context of libraries: *client, implementation, interface.*

**The part of a program that uses the library (along with the programmer writing that code) is called a *client* of the library. The code for the library itself is called the *implementation.* The *interface* represents the information that the client and the implementation share.**

16. If you were writing an interface called **mylib.h**, what lines would you include as the interface boilerplate?

**At the beginning of the interface file, you would use the following boilerplate lines:**

#ifndef \_mylib\_h

#define \_mylib\_h

**The following line would appear at the end of the interface file:**

#endif

17. Describe the process used to export a constant definition as part of an interface.

**The implementation file includes the definition of the constant, including the initializer value; the interface file specifies only the name and type of the constant, without the value. Unlike constants that are local to a single source file, exported constants must include the keyword extern in both the implementation and interface files.**

18. What criteria are identified in this chapter as central to the process of interface design?

**Interfaces should be *unified, simple, sufficient, general,* and *stable*.**

19. Why is it important for an interface to be stable?

**Libraries typically have a large number of clients that depend on being able to invoke the services of that library in a particular form. If the interface changes in an incompatible way, clients will be forced to update their own code.**

20. What is meant by the term *pseudorandom number?*

**A *pseudorandom number* is produced by a computational process in such a way that it “appears” to be random, in the sense that it is difficult for the user to predict and that it passes statistical tests for randomness.**

21. On most computers, how is the value of **RAND\_MAX** chosen?

**The value of RAND\_MAX is typically the largest positive value of type int.**

22. What four steps are necessary to convert the result of **rand** into an integer value with a different range?

**1. *Normalize* the integer result from rand by converting it into a floating‑point number *d* in the range 0 ≤ *d* < 1.**

**2. *Scale* the value *d* by multiplying it by the size of the desired range, so that it spans the correct number of integers.**

**3. *Translate* the value by adding in the lower bound so that the range begins at the desired point.**

**4. *Convert* *the number to an integer* by calling the function floor from <cmath>, which returns the largest integer that is smaller than its argument.**

23. How would you use the **randomInteger** function to generate a pseudorandom number between 1 and 100?

You would call the function **randomInteger(1, 100)**

24. By executing each of the statements in the implementation by hand, determine whether the **randomInteger** function works with negative arguments. What are the possible results of calling the function **randomInteger(-5,** **5)**?

**The randomInteger function works correctly with negative values. The possible values of randomInteger(-5,** **5) are therefore −5, −4, −3, −2, −1, 0, 1, 2, 3, 4, and 5.**

25. Assuming that **d1** and **d2** have already been declared as variables of type **int**, could you use the multiple assignment statement

d1 = d2 = RandomInteger(1, 6);

to simulate the process of rolling two dice?

**No. This statement would set d1 and d2 to the same random value between 1 and 6.**

26. True or false: The **rand** function ordinarily generates the same sequence of random numbers every time a program is run.

**True.**

27. What is meant by the term *seed* in the context of random numbers?

**The *seed* is the starting value for the random number sequence.**

28. What suggestion does this chapter offer for debugging a program involving random numbers?

**When you use the random.h interface, it makes sense to call setRandomSeed(1) so that the behavior of the program is repeatable. When the program is working, you can remove this statement.**

29. What functions are defined in the final version of the **random.h** interface? In what context would you use each function?

**The random.h interface exports the functions randomInteger, randomReal, randomChance, and setRandomSeed. The randomInteger function is useful when you need a random integer in a certain range, as in simulating a die roll. The randomReal function is useful if you need to choose a value that can vary continuously within a specified range. The randomChance function simulates an event that happens with a particular probability. The setRandomSeed function allows you to set a specified starting point for the random number sequence and exists primarily to support debugging.**