 Answers to review questions from Chapter 3

1. What is the difference between a *character* and a *string?*

**A *character* is a data value that represents a single character of text. A *string* is a sequence of characters.**

2. True or false: If you execute the lines

string line;

cin >> line;

the program will read an entire line of data from the user and store it in the variable **line**.

**False. The program will read a string up to the next whitespace character.**

3. Which arguments to the **getline** function are passed by reference?

**Both arguments are passed by reference.**

4. What is the difference between a *method* and a *free function?*

**A *method* is a function that is part of an object. In C++, a *free function* is defined at the top level of a file and is not part of any class definition.**

5. True or false: In C++, you can determine the length of the string stored in the variable **str** by calling **length(str)**.

**False. In C++, length is a method, so the call is str.length().**

6. If you call **s1.replace(0, 1, s2)**, which string is the *receiver*?

**s1**

7. What is the effect of the **+** operator when it is used with two string operands?

**The + operator signifies *concatenation,* which joins the strings together.**

8. When C++ evaluates the expression **s1 < s2**, what rule does the **string** class use to compare the string values?

**C++ uses *lexicographic order,* which is the order imposed by the ASCII values of the characters.**

9. What two syntactic forms does this chapter describe for selecting an individual character from a string? How do these two syntactic forms differ in their implementation?

**C++ supports both bracket selection and the at method. The at method checks to make sure that the index is in range.**

10. When you select an individual character from a C++ string, you can use either the **at** method or the standard subscript notation in which the index is enclosed in square brackets. From the client’s perspective, what is the difference between these two options?

**As noted in the previous answer, the at method checks to make sure that the index is in range. Despite the fact that the at method is safer, bracket selection is much more common.**

11. True or false: If you assign the value of the string variable **s1** to the string variable **s2**, the **string** class copies the characters so that subsequent changes to the characters in one string will not affect the characters in the other.

**True**

12. True or false: The index positions in a string begin at 0 and extend up to the length of the string minus 1.

**True**

13. What are the arguments to the **substr** method? What happens if you omit the second argument?

**The first argument to substr is the starting position; the second is the desired number of characters. If the second argument is missing, substr returns the rest of the string after the starting position.**

14. Describe how the **compare** method uses the return value to indicate the relative ordering of two strings. Why is this method rarely used in practice?

**The compare method returns 0 if the strings are equal, a positive value if the receiver string is greater than the argument, and a negative value if the receiver string is smaller than the argument. In most cases, it is more convenient to use the relational operators to compare strings.**

15. What value does the **find** method return to indicate that the search string does not appear?

**string::npos**

16. What is the significance of the optional second argument to the **find** method?

**The second argument represents the starting position for the search.**

17. Suppose that you have declared and initialized the variables **s** and **t** like this:

string s = "ABCDE";

string t = "";

Given these declarations, what is the effect of each of the following calls:

a. **s.length()**→**5**

b. **t.length()**→**0**

c. **s[2]**→**'C'**

d. **s + t**→**"ABCDE"**

e. **t += 'a'**→**"a"**

f. **s.replace(0, 2, "Z")**→**"ZCDE"** (the variable **s** changes to this value as well)

g. **s.substr(0, 3)**→**"ABC"**

h. **s.substr(4)**→**"E"**

i. **s.substr(3, 9)**→**"DE"**

j. **s.substr(3, 3)**→**"DE"**

18. What is the pattern for iterating through each character in a string?

for (int i = 0; i < str.length(); i++) {

. . . *body of loop that manipulates* str[i] . . .

}

19. How does the pattern in question 18 change if you want to iterate through the characters in reverse order, starting with the last character and ending with the first?

for (int i = str.length() - 1; i >= 0; i--) {

. . . *body of loop that manipulates* str[i] . . .

}

20. What is the pattern for growing a string through concatenation?

string str = "";

for (*whatever loop header line fits the application*) {

str += *the next substring or character*;

}

21. What is the result of each of the following calls to the **<cctype>** library:

a. **isdigit(7)**→**false**

b. **isdigit('7')**→**true**

c. **isalnum(7)**→**false**

d. **toupper(7)**→**7**

e. **toupper('A')**→**'A'**

f. **tolower('A')**→**'a'**

22. Why does C++ support both a **string** class and a more primitive string type?

**C++ was designed to be backward-compatible with C. It therefore supports primitive strings that match those in C.**

23. How can you convert a primitive string value to a C++ string? How can you specify a conversion in the opposite direction?

**Convert a primitive string to a C++ string: string(cstr)**

**Convert a C++ string to a primitive string: str.c\_str()**