 Answers to review questions from Chapter 12

1. What are the three allocation mechanisms described in this chapter?

**Static allocation, automatic allocation, and dynamic allocation.**

2. What is the *heap?*

**The *heap* is the pool of unassigned memory between the static area and the stack, which is available for dynamic allocation.**

3. Why does it make sense to start the heap and the stack at opposite ends of memory and have them grow toward each other?

**This strategy ensures that memory is always available for both the heap and the stack until all of the available memory is exhausted.**

4. What declaration would you use to create and initialize each of the following variables:

a) A pointer **bp** that points to a Boolean value

bool \*bp = new bool;

b) A pointer named **pp** that points to a **Point** with the coordinates (3, 4)

Point \*pp = new Point(3, 4);

c) A dynamic array called **names** capable of holding 100 C++ strings?

const int MAX\_NAMES = 100;

string \*names = new string[MAX\_NAMES];

5. What statements would you use to free the storage allocated in the preceding exercise?

delete bp;

delete pp;

delete[] names;

6. Define the terms *cell* and *link* as they are used in the context of the linked list data structure.

**A *cell* is a structure that stores one item in a linked list. It has two components: a *value* field whose type and structure depend on the data stored in the list and a *link* field, which is a pointer to the next cell in the list.**

7. What is the standard technique for marking the end of a linked list?

**The end of the list is marked by the special pointer value NULL.**

8. What structure definition would you use to define the cell type for a linked list of integers?

struct Cell {

int value;

Cell \*link;

};

9. Given your definition in the preceding exercise, how would you write a **for** loop to step through every element of a linked list stored in the variable **list**?

for (Cell \*cp = list; cp != NULL; cp = cp->link)

10. What is a *memory leak?*

**A *memory leak* occurs when an application allocates memory but fails to free it when it is no longer needed.**

11. True or false: C++ uses garbage collection to manage memory.

**False. In C++, programmers are responsible for freeing any memory they allocate.**

12. What is a *destructor?* What is its most important role?

**A *destructor* is a special method that is called whenever an object is explicitly deleted or goes out of scope. The primary purpose of a destructor is to free any heap memory that the object has acquired.**

13. If you create a class named **IntArray**, how would you write the prototype for its destructor?

~IntArray()

14. What does it mean for a variable to *go out of scope?*

**An automatic variable *goes out of scope* when the function that declares it returns.**

15. True or false: Destructors can be invoked even on temporary values that are never assigned to local variables.

**True. The compiler allocates memory for those temporary values as if they were automatic variables.**

16. How is it possible for the **CharStack** class to expand its capacity dynamically even though it uses arrays whose size is fixed at the time they are allocated?

**The CharStack class allocates new memory whenever the existing space is exhausted and then copies the existing data into the larger memory block.**

17. Describe the purpose of each of the instance variables in the **CharStack** class.

|  |  |
| --- | --- |
| char \*array; | ***This variable holds the dynamic array used to store the elements.*** |
| int capacity; | ***This variable keeps track of the allocated size of the array.*** |
| int count; | ***This variable keeps track of how many characters are on the stack.*** |

18. Explain each of the statements in the implementation of **expandCapacity** in Figure 12‑8.

|  |  |
| --- | --- |
| void CharStack::expandCapacity() { |  |
| char \*oldArray = array; | ***Saves the pointer to the old array.*** |
| capacity \*= 2; | ***Doubles the stack capacity.*** |
| array = new char[capacity]; | ***Allocates a new array with that larger size.*** |
| for (int i = 0; i < count; i++) { | ***Cycles through all characters in the stack.*** |
| array[i] = oldArray[i]; | ***Copies the old value to the new array.*** |
| } |  |
| delete[] oldArray; | ***Frees the heap memory in the old array.*** |
| } |  |

19. Suppose that, instead of doubling the capacity of the array, **expandCapacity** simply added one more element to the array. Would the **push** method still have an average computational complexity of *O*(1)? Why or why not?

**This redesign would change the average computational complexity of push to *O*(*N*). Once the original capacity was reached, every push operation would incur the linear‑time cost of copying the old array.**

20. When is new memory added to the stack side of a heap‑stack diagram? When does that memory get reclaimed?

**New stack frames are added to the stack side of the diagram whenever a function or method is called. That memory is reclaimed when the function returns.**

21. When is new memory added to the heap side of a heap‑stack diagram and when is it reclaimed?

**New heap memory is allocated only when the programmer invokes the new operator. That memory is freed by calling delete, typically in the implementation of a destructor.**

22. What reason does the chapter give for including the *overhead word* in heap‑stack diagrams?

**The stack frame includes additional data beyond the variables, such as the address of the instruction that follows the call. Including a placeholder for this information in the diagrams makes it easier to see the boundaries of the different stack frames.**

23. How do you represent a reference parameter in a heap‑stack diagram?

**In the heap‑stack diagrams used in this text, reference parameters are marked with an ampersand (&), just as they are in the code. Reference parameters are always a single memory word, which is used to hold the address of the actual argument.**

24. What additional local variable gets added to a stack frame when you call a method as opposed to a function?

**All method calls generate a stack frame entry called this, which holds the address of the receiving object.**

25. What is the implication of the word *unit* in the phrase *unit test?*

**A *unit test* checks the code for a particular piece of the code (typically a class definition or a library implementation) without making assumptions about the context in which that code is used.**

26. What is the difference between a *shallow copy* and a *deep copy?* Which of these two strategies does C++ use by default?

**In a *shallow copy,* any pointer values in the source data are copied as pointers to the target and therefore refer to the same memory. In a *deep copy,* the data structures to which those pointers refer are also copied. C++ uses shallow copying by default.**

27. What methods must you override to change how C++ copies an object?

**The copy constructor and the assignment operator.**

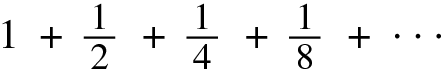
28. How does *constant call by reference* differ from the more familiar parameter passing paradigms of *call by value* and *call by reference?*

**If you mark a reference parameter with the keyword const, the compiler takes that as an indication that the function or method cannot change the value of that parameter and checks for any violation of that rule. Actual arguments that are passed using constant call by reference need not be lvalues.**

29. What does it mean for a class to be **const** correct?

**A class is const correct if all of its methods indicate which of the parameters are subject to change by the method and whether that method can change the object.**

30. The argument that the amortized complexity of the **push** operation is *O*(1) depends on the claim that the sum of the series



can never exceed 2 no matter how many terms you include. In your own words, try to explain why. (If you have trouble, you might try looking up *Zeno’s Paradox* on the web and then giving it another go.)

**Each new term in the sum takes you half of the remaining distance toward 2. No matter how many terms you use, the sum will still be less than 2.**