 Answers to review questions from Chapter 19

1. In C++, what header line would you use to define a class named **Sub** that inherited the public behavior from a class named **Super**?

class Sub : public Super

2. True or false: The superclass specification in a new class definition may not be a template class with specific instantiation of the template types.

**False.**

3. True or false: As in most object‑oriented languages, a new definition of a method in a C++ subclass automatically overrides the definition of that method in its superclass.

**False. In C++, overriding happens only for virtual methods.**

4. In your own words, describe the effect of the **virtual** keyword.

**The virtual keyword signifies that the behavior of the associated method can be overridden by subclasses.**

5. What is a *pure virtual method?* Why is such a construct useful?

**A *pure virtual method* is a method that has no definition in the superclass but which must be overridden by every concrete subclass. Pure virtual methods make it possible to define abstract classes that cannot be instantiated on their own but which provide a useful base for a class hierarchy.**

6. What syntactic marker does C++ use to indicate that a method is pure virtual?

**The header line is followed by the marker = 0 before the closing semicolon.**

7. What is an *abstract class?* Is it possible for an abstract class to provide its own implementation of its exported methods?

**An *abstract class* is a class that has no objects that are not also instances of a concrete subclass. Abstract classes usually define their own implementations for exported methods, which are then shared by any subclasses further down in the hierarchy.**

8. What is meant by the term *slicing?*

**C++ allows you to assign an object to a variable of its superclass type, but the behavior of that operation is not what one would expect from other object‑oriented languages. In general, a subclass object is assigned more space than objects of its superclass. C++ simply deletes the extra information when it copies the object. This behavior is known as *slicing.***

9. When you store values from an inheritance hierarchy in a collection, does it make more sense to store the values themselves or pointers to those values allocated elsewhere?

**Pointers are generally much more useful because they avoid the loss of information caused by slicing.**

10. What classes and methods are virtual in the **GObject** class hierarchy shown in Figure 19‑3?

**The GObject class is virtual, as is its draw method.**

11. How does the visibility of entries in the **protected** section of a class differ from those in its **public** and **private** sections?

**Entries in the protected section are visible to subclasses but not to classes that do not have a subclass relationship. In essence, protected entries act as if they were public for subclasses and as if they were private for all other classes.**

12. What is an *initializer list?* Where do initializer lists appear in a C++ program?

**An *initializer* *list* is a comma‑separated list of specifications, which must be either a call to a superclass constructor or instance variables followed by an expression in parentheses. Initializer lists appear in the header line of a constructor after a colon but before the brace that begins the body.**

13. What is the difference between an *interpreter* and a *compiler?*

**A *compiler* translates a program from a source language into the target language for the machine, which allows it to be executed directly on that hardware. An *interpreter* executes a program by simulating its operation. Interpreters are typically easier to write than compliers but run far more slowly.**

14. What is a *read-eval-print loop?*

**A *read‑eval‑print loop* is a program that reads some program text—such as the arithmetic expressions introduced in this chapter—and then interprets that program to produce a value, which is then printed on the console.**

15. What are the three phases involved in reading an expression?

**1. *Input***

**2. *Lexical analysis***

**3. *Parsing***

16. What is an *exception?*

**An *exception* is an event that occurs outside of normal program operation and is therefore handled in a special way.**

17. In its simplest form that catches only one exception type, what is the syntax of the **try** statement in C++?

try {

*code under the control of the try statement*

} catch (*type* *var*) {

*code to respond to an exception with the specified value type*

}

18. State the recursive definition for an expression as given in this chapter.

**A sequence of symbols is an *expression* if it has one of the following forms:**

**1. An integer constant**

**2. A variable name**

**3. An expression enclosed in parentheses**

**4. A sequence of two expressions separated by an operator**

19. Identify which of the following lines constitutes an expression according to the definition used in this chapter:

a. **(((0))) Yes**

b. **2x + 3y No; it is missing the \* operator**

c. **x - (y \* (x / y)) Yes**

d. **-y No; unary minus is not an operator**

e. **x = (y = 2 \* x - 3 \* y) Yes**

f. **10 - 9 + 8 / 7 \* 6 - 5 + 4 \* 3 / 2 - 1 Yes**

20. For each of the legal expressions in the preceding question, draw a parse tree that reflects the standard precedence assumptions of mathematics.

|  |  |
| --- | --- |
| **a. (((0)))** | /Users/eroberts/Books/ProgrammingAbstractionsInC++/chapters/19-Inheritance/pictures/ExpressionTrees/ExpressionExercise1.png |
| **c.** **x - (y \* (x / y))** | /Users/eroberts/Books/ProgrammingAbstractionsInC++/chapters/19-Inheritance/pictures/ExpressionTrees/ExpressionExercise2.png |
| **e.** **x = (y = 2 \* x - 3 \* y)** | /Users/eroberts/Books/ProgrammingAbstractionsInC++/chapters/19-Inheritance/pictures/ExpressionTrees/ExpressionExercise3.png |
| **f.** **10 - 9 + 8 / 7 \* 6 - 5 + 4 \* 3 / 2 - 1** | /Users/eroberts/Books/ProgrammingAbstractionsInC++/chapters/19-Inheritance/pictures/ExpressionTrees/ExpressionExercise4.png |

21. Of the legal expressions in question 19, which ones are ambiguous with respect to the simple recursive definition of expressions?

**Expressions e and f.**

22. What are the differences between parse trees and expression trees?

**Parse trees show the derivation in the grammar and therefore include rules such as those used to group parentheses; these reductions are missing from expression trees.**

23. What are the three types of expressions that can occur in an expression tree?

**Constant expressions, identifier expressions, and compound expressions.**

24. True or false: The methods in the **exp.h** interface do not work with **Expression** objects directly but instead use pointers to **Expression** objects.

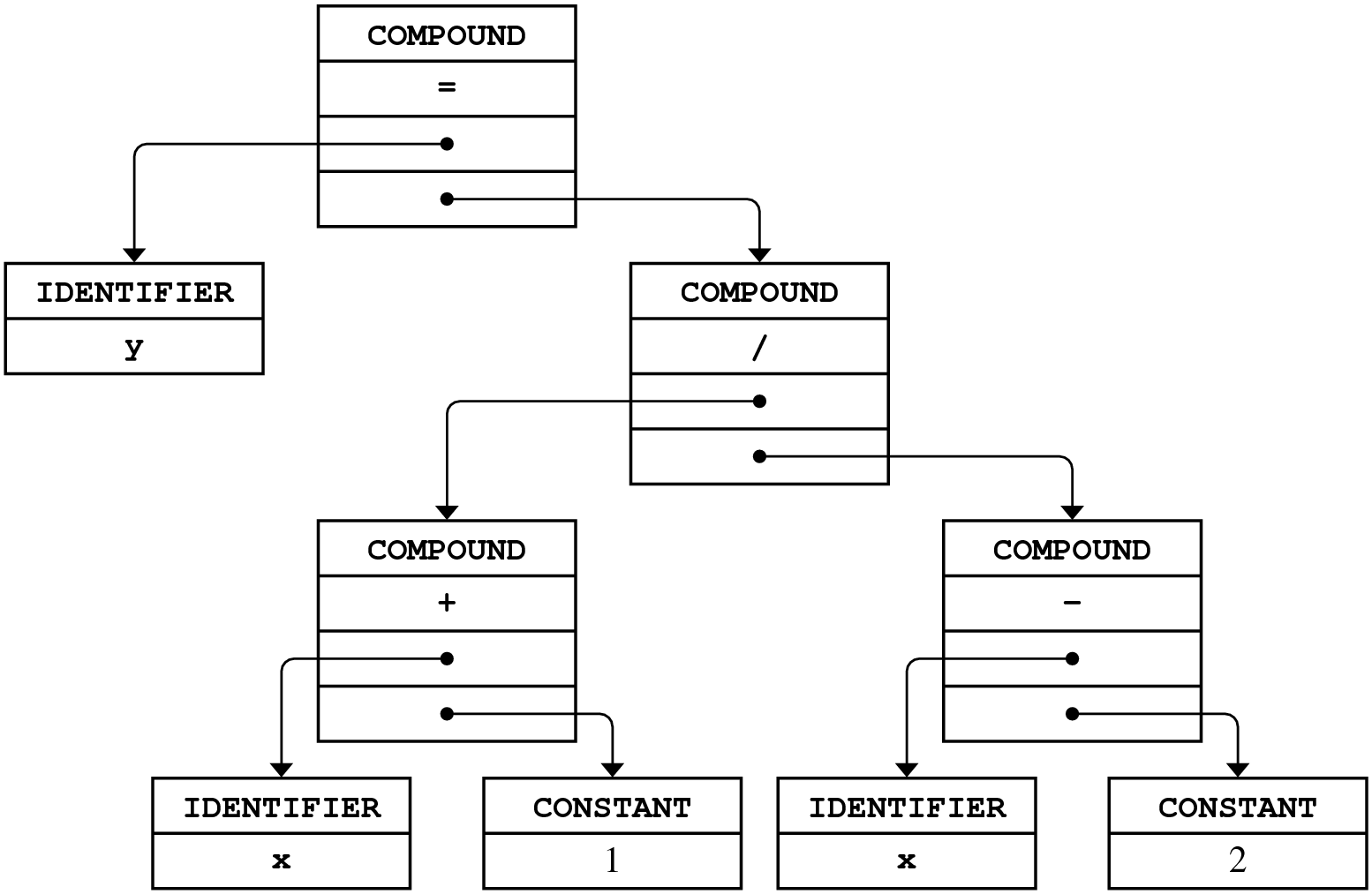
**True.**

25. What are the public methods of the **Expression** class?

**The general‑purpose methods are eval, toString, and getType. As a convenience, the abstract Expression class exports getters for the subtype fields. These methods are getConstantValue, getIdentifierName, getOperator, getLHS, and getRHS.**

26. Using Figure 19‑12 as a model, draw a complete structure diagram for the following expression:

y = (x + 1) / (x - 2)



27. Why are grammars useful in translating programming languages?

**Grammars provide a formal description of the syntax of a programming language that is compact and easy to understand. In most cases, it is possible to use automatic tools to generate a parser for a language when you have the grammar.**

28. What do the letters in *BNF* stand for?

**Backus‑Naur form, named after the inventors.**

29. In a grammar, what is the difference between a *terminal symbol* and a *nonterminal symbol?*

**A *terminal symbol* is one that appears in the input being parsed; a *nonterminal symbol* is a placeholder for a string of terminal symbols that represents one level in the structure of the grammar.**

30. What is a *recursive-descent parser*?

**A *recursive‑descent* *parser* uses recursion in its implementation to model the recursive rules in the grammar.**

31. What is the significance of the second argument to the **readE** function in the implementation of the parser?

**The second argument is the current precedence level, which is used to resolve the ambiguity in the grammar.**

32. If you look at the definition of **readT** in Figure 19‑14, you will see that the function body does not contain any calls to **readT**. Is **readT** a recursive function?

**The function is mutually recursive because readT calls** **readE, which in turn calls** **readT.**