**Chapter 18: Operator Overloading; String and Array Objects**

**Section 18.1 Introduction**

18.1 Q1: Which statement about operator overloading is *false*?

a. Operator overloading is the process of enabling C++'s operators to work with class objects.

b. C++ overloads the addition operator (+) and the subtraction operator (-) to perform differently, depending on their context in integer, floating-point and pointer arithmetic with data of fundamental types.

c. You can overload all C++ operators to be used with class objects.

d. When you overload operators to be used with class objects, the compiler generates the appropriate code based on the types of the operands.

ANS: c. You can overload all C++ operators to be used with class objects. [Most, not all, operators can be overloaded.]

**Section 18.2 Using the Overloaded Operators of Standard Library Class string**

18.2 Q1: Which of the following is *false*?

1. A string can be defined to store any data type.
2. Class string provides bounds checking in its member function at.
3. Class string’s overloaded [] operator returns a vector element as an *rvalue* or an *lvalue*, depending on the context.
4. An exception is thrown if the argument to string’s at member function is an invalid subscript.

**ANS: a. A string can be defined to store any data type.**

**Section 18.3 Fundamentals of Operator Overloading**

18.3 Q1: The correct function name for overloading the addition (+) operator is:

1. operator+
2. operator(+)
3. operator:+
4. operator\_+

**ANS a. operator+**

18.3 Q2: Which of the following operators *cannot* be overloaded?

1. The . operator.
2. The -> operator.
3. The & operator.
4. The [ ] operator.

**ANS a. The . operator.**

18.3 Q3: Which statement about operator overloading is *false*?

1. New operators can never be created.
2. Certain overloaded operators can change the number of arguments they take.
3. The precedence of an operator cannot be changed by overloading.
4. Overloading cannot change how an operator works on built-in types.

**ANS b. Certain overloaded operators can change the number of arguments they take.**

18.3 Q4: To implicitly overload the += operator:

1. Only the + operator needs to be overloaded.
2. Only the = operator needs to be overloaded.
3. Both the + and = operators need to be overloaded.
4. The += operator cannot be overloaded implicitly.

**ANS d. The += operator cannot be overloaded implicitly.**

**Section 18.4 Overloading Binary Operators**

18.4 Q1: Which of the following operators *can* be overloaded as a non-member function?

1. ()
2. []
3. +=
4. ==

**ANS: d. ==**

18.4 Q2: Which situation would *require* the operator to be overloaded as a non-member function?

1. The overloaded operator is =.
2. The left most operand must be a class object (or a reference to a class object).
3. The left operand is an int.
4. The operator returns a reference.

**ANS c. The left operand is an int.**

18.4 Q3: An overloaded + operator takes a class object and a double as operands. For it to be commutative (i.e., a + b and b + a both work):

1. operator+ must be a member function of the class from which the objects are instantiated.
2. operator+ must be a non-member function.
3. It must be overloaded twice; the operator+ function that takes the object as the left operand must be a member function, and the other operator+ function must be a global function.
4. The + operator cannot be overloaded to be commutative.

**ANS c. It must be overloaded twice; the operator+ function that takes the object as the left operand must be a member function, and the other operator+ function must be a global function.**

18.4 Q4: y and z are user-defined objects and the += operator is an overloaded member function. The operator is overloaded such that y += z adds z and y, then stores the result in y. Which of the following expressions is always equivalent to y += z?

1. y = y operator+= z
2. y.operator+=( z )
3. y = y + z
4. y operator+=( y + z )

**ANS: b. y.operator+=( z )**

18.4 Q5: For operators overloaded as non-static member functions:

1. Binary operators can have two arguments and unary operators can have one.
2. Both binary and unary operators take one argument.
3. Binary operators can have one argument, and unary operators cannot have any.
4. Neither binary nor unary operators can have arguments.

**ANS: c. Binary operators can have one argument, and unary operators cannot have any.**

**Section 18.5 Overloading the Binary Stream Insertion and Stream Extraction Operators**

18.5 Q1: Suppose you have a programmer-defined data type Data and want to overload the << operator to output your data type to the screen in the form cout << dataToPrint; and allow cascaded function calls. The first line of the function definition would be:

1. ostream &operator<<( ostream &output, const Data &dataToPrint )
2. ostream operator<<( ostream &output, const Data &dataToPrint )
3. ostream &operator<<( const Data &dataToPrint, ostream &output )
4. ostream operator<<( const Data &dataToPrint, ostream &output )

**ANS: a. ostream &operator<<( ostream &output, const Data &dataToPrint )**

**Section 18.6 Overloading Unary Operators**

18.6 Q1: Suppose the unary ! operator is an overloaded member function of class String. For a String object s, which function call is generated by the compiler when it finds the expression !s?

1. s.operator!()
2. s.operator!( *default\_value1*, *default\_value2*,…)
3. operator!( s )
4. A compiler error results because no arguments are given.

**ANS: a. s.operator!()**

**Section 18.7 Overloading the Unary Prefix and Postfix ++ and -- Operators**

18.7 Q1: The conventional way to distinguish between the overloaded preincrement and postincrement operators (++) is:

1. To assign a dummy value to preincrement.
2. To make the argument list of postincrement include an int.
3. To have the postincrement operator call the preincrement operator.
4. Implicitly done by the compiler.

**ANS: b. To make the argument list of postincrement include an int.**

18.7 Q2: Because the postfix increment operator returns objects by value and the prefix increment operator returns objects by reference:

1. Prefix increment has slightly more overhead than postfix increment.
2. The postfix increment operator returns the actual incremented object with its new value.
3. Objects returned by postfix increment cannot be used in larger expressions.
4. The postfix increment operator typically returns a temporary object that contains the original value of the object before the increment occurred.

**ANS: d. The postfix increment operator typically returns a temporary object that contains the original value of the object before the increment occurred.**

**Section 18.8 Case Study: A Date Class**

18.8 Q1: There exists a data type Date with member function Increment that increments the current Date object by one. The ++ operator is being overloaded to postincrement an object of type Date. Select the correct implementation:

1. Date Date::operator++( int )  
   {  
    Date temp = \*this;  
    Increment();  
    return \*temp;  
   }
2. Date Date::operator++( int )  
   {  
    Increment();  
    Date temp = \*this;  
    return temp;  
   }
3. Date Date::operator++( int )  
   {  
    Date temp = \*this;  
    return this;  
    temp.Increment();  
   }
4. Date Date::operator++( int )  
   {  
    Date temp = \*this;  
    Increment();  
    return temp;  
   }

**ANS: d. Date Date::operator++( int )  
 {  
 Date temp = \*this;  
 Increment();  
 return temp;  
 }**

**Section 18.9 Dynamic Memory Management**

18.9 Q1: Which of the following *is false* about the new operator and the object for which it allocates memory?

1. It calls the object’s constructor.
2. It returns a pointer.
3. It does *not* require the size of the object to be *explicitly* specified in the new expression.
4. It automatically destroys the object *after* mainis exited.

**ANS: d. It automatically destroys the object after main is exited.**

18.9 Q2: The delete [] operator:

1. Can terminate the program.
2. Must be told which destructor to call when destroying an object.
3. Can delete an entire array of objects declared using new.
4. Is called *implicitly* at the end of a program.

**ANS: c. Can delete an entire array of objects declared using new.**

18.9 Q3[C++11]: Which of the following statements about a unique\_ptr object is *true*?

1. A unique\_ptr is a “smart pointer” for managing dynamically allocated memory.
2. When a unique\_ptr goes out of scope, its destructor automatically returns the managed memory to the free store.
3. You must explicitly delete the memory that’s managed by a unique\_ptr before the object goes out of scope.
4. All of the above.

**ANS: c. You must explicitly delete the memory that’s managed by a unique\_ptr before the object goes out of scope.**

**Section 18.10 Case Study: Array Class**

18.10 Q1: Which of the following is *false*?

1. An entire non-char array cannot be input or output at once.
2. Two arrays cannot be meaningfully compared with equality or relational operators.
3. Arrays cannot be assigned to one another (i.e., array1 = array2;).
4. C++ ensures that you cannot “walk off” either end of an array.

**ANS: d. C++ ensures that you cannot “walk off” either end of an array.**

18.10 Q2: The array subscript operator [], when overloaded, *cannot*:

1. Be used with linked list classes.
2. Take a float as an operand.
3. Take multiple values inside (e.g., [4,8]).
4. Take user-defined objects as operands.

**ANS: c. Take multiple values inside (e.g., [4,8]).**

18.10 Q3: A copy constructor:

1. Is a constructor with only default arguments.
2. Is a constructor that initializes a newly declared object to the value of an existing object of the same class.
3. Is a constructor that takes no arguments.
4. None of the above.

**ANS: b. Is a constructor that initializes a newly declared object to the value of an existing object of the same class.**

18.10 Q4: A copy constructor *must* receive its argument by reference because:

1. Otherwise the constructor will only make a copy of a pointer to an object.
2. Otherwise infinite recursion occurs.
3. The copy of the argument passed by value has function scope.
4. The pointer needs to know the address of the original data, not a temporary copy of it.

**ANS: b. Otherwise infinite recursion occurs.**

18.10 Q5: To prevent class objects from being copied:

1. Make the overloaded assignment operator private.
2. Make the copy constructor private.
3. Both (a) and (b).
4. None of the above.

**ANS: c. Both (a) and (b).**

18.10 Q6: Which of the following is *not* a disadvantage of default memberwise copy with objects containing pointers?

1. Having the possibility of leaving a dangling pointer.
2. Allowing both objects to point to the same dynamically allocated storage.
3. Allowing the destructor of one object to be called while leaving the second pointer, to the same memory location, intact.
4. Requiring the explicit overloading of the assignment operator.

**ANS: d. Requiring the explicit overloading of the assignment operator.**

18.10 Q7[C++11]: To prevent class objects from being copied or assigned, you can:

1. Declare as private the class’s copy constructor and overloaded assignment operator.
2. Declare the class’s copy constructor and overloaded assignment operator with with = delete after the parameter list.
3. Simply do not declare a copy constructor or assignment operator in the class.
4. (a) or (b).

**ANS: d. (a) or (b).**

18.10 Q8[C++11]: Which of the following is *false*?

1. To receive a list initializer as an argument to a constructor, you can declare the constructor’s parameter as type list\_initialier<T> where T represents the type of the values in the list initializer.
2. To receive a list initializer as an argument to a constructor, you can declare the constructor’s parameter as type initializer\_list<T> where T represents the type of the values in the list initializer.
3. It’s not possible to pass a list initializer to a constructor.
4. (a) and (c).

**ANS: d. (a) and (c).**

**Section 18.11 Operators as Member Function vs. Non-Member Functions**

18.11 Q1: Which statement is *false*?

a. Based on whether an operator is implemented as a member function or as a non-member function, the operator is used differently in expressions.

b. When an operator function is implemented as a member function, the leftmost (or only) operand must be an object (or a reference to an object) of the operator's class.

c. Operator member functions of a specific class are called (implicitly by the compiler) only when the left operand of a binary operator is specifically an object of that class, or when the single operand of a unary operator is an object of that class.

d. Another reason why you might choose a non-member function to overload an operator is to enable the operator to be commutative.

ANS: a. Based on whether an operator is implemented as a member function or as a non-member function, the operator is used differently in expressions. [Whether an operator is implemented as a member function or as a non-member function, the operator is still used the *same* way in expressions.]

**Section 18.12 Converting between Types**

18.12 Q1: Conversion constructors:

1. Can have *multiple* arguments.
2. Can convert between user-defined types.
3. Are *implicitly* defined by the compiler if not *explicitly* written by the programmer.
4. Cannot convert built-in types to user defined types.

**ANS: b. Can convert between user-defined types.**

18.12 Q2: The prototypes of overloaded cast operator functions do *not*:

1. Specify the type they convert to.
2. Specify the type that is being converted.
3. Specify a return type.
4. Need to be defined inside the class whose objects are being converted.

**ANS: c. Specify a return type.**

18.12 Q3: Which of the following lines would be the prototype for an overloaded cast operator function that converts an object of user-defined type Time into a double?

1. Time::operator double() const;
2. Time::static\_cast double() const;
3. Time::operator\_cast(double) const;
4. Time::double() const;

**ANS: a. Time::operator double() const;**

**Section 18.13 explicit Constructors and Conversion Operator**

18.13 Q1: An explicit constructor:

1. *Cannot* be called outside of the class it is declared in.
2. Can be *implicitly* called by the compiler to perform a data type conversion.
3. Does *not* initialize its class’s data members.
4. Must take *exactly one* argument.

**ANS: d. Must take *exactly one* argument.**

**Section 18.14 Overloading the Function Call Operator ()**

18.14 Q3: Assume that the function call operator() is overloaded for data type String in the usual sense of selecting a substring from a larger string. For a String object string1 with the character string "ABCDEFGHI", what string does string1( 4 , 2 ) return?

1. "EF"
2. "EFGHI"
3. "CDEF"
4. "CD"

**ANS: a. "EF"**