Consider the following Java definition of a mutable string class.

1. class MutableString
2. {
3. private char[] chars = new char[200];
4. private int size = 0;
5. public boolean set(char aChar, int index)
6. {
7. if (index < 0 || index >= chars.length)
8. return false;
9. chars[index] = aChar;
10. return true;
11. }
12. public String toString()
13. {
14. String result = "";
15. for (int i = 0; i < size; i++)
16. result += chars[i];
17. return result;
18. }

}

Suppose this class was rewritten in C++ in two ways, the first with the array that represents the list as a stack-dynamic variable, and then with the list as a heap dynamic variable. Explain when constructors and destructors would be needed. Explain why no constructors or destructors are needed in the Java implementation.

Answer:

A constructor and destructor would be required for stack-dynamic, because stack-dynamic is when a programmer explicitly creates and destroys the variables. A heap-dynamic is the “new” isn’t required and tells it to create it in a heap, thus the compiler is taking care of allocating and deallocating (I believe at runtime).

Java is different because Java monitors objects on the heap to see if they are being referenced by anyone, once the unction exits, all the references to the dynamic memory are removed.

Consider the following C++ template class.

1. template <typename T, int length>
2. class Vector
3. {
4. public:
5. Vector(T values[length])
6. {
7. for (int i = 0; i < length; i++)
8. list[i] = values[i];
9. }
10. friend bool operator<(const Vector<T, length>& left, const Vector<T, length>& right)
11. {
12. bool result = true;
13. for (int i = 0; i < length; i++)
14. result &= left.list[i] < right.list[i];
15. return result;
16. }
17. private:
18. T list[length];
19. };
20. int main()
21. {
22. int first[] = {1, 2}, second[] = {2, 3};
23. Vector<int, 2> vector1(first), vector2(second);
24. cout << (vector1 < vector2) << endl;
25. return 0;

}

The class Vector cannot be instantiated for any arbitrary type. For example, consider the following instantiation for a wrapper integer class.

class Int

{

public:

Int(int i = 0) {this->i = i;}

private:

int i;

};

int main()

{

Int first[] = {Int(1), Int(2)}, second[] = {Int(2), Int(3)};

Vector<Int, 2> vector1(first), vector2(second);

cout << (vector1 < vector2) << endl;

return 0;

}

Explain why the second implementation fails. What must be added to that class so this program will compile? Suppose this program were written in Java. Explain how Java allows the constraints on a generic type parameter to be specified and how they would be specified in this case

**Answer:**  Error: Vector is not declared in this scope.

T list[length] needs a default Int() constructor for the array.

Int doesn’t define operator< for the comparison

As for Java: Type arguments can be forced to match given interfaces. For example, a generic SortedList could be limited to Number subclasses:

public class SortedList<T extends Number>

Java does have one limitation, however. Although wrapper classes can be used to instantiate generic type parameters, primitive types cannot. Explain why.

**Answer:** As I understand it Java generics were intended for operations on objects that reduced to Object class instances at compile time. i.e List.sort(Object)