Fall 2018 Test 2 Key w/ Explanations

1. **1**

Not on this test. May update later.

2. **3**

Not on this test. May update later.

3. **6**

Not on this test. May update later.

4. **7**

Not on this test. May update later.

5. **-1**

Not on this test. May update later.

6. **-2 -2**

Not on this test. May update later.

7. **1 0 1 0**

brv calls the constructor which takes an int which initializes v to 1. b1 calls the default constructor which initializes v to 0. b2 calls the constructor which takes an int which initializes v to 1. bP1 called the default constructor which initializes v to 0.

8. **1 1**

When we set two references equal it will call by default copy the information on the right into what is on the left side of the = sign. This code has defined its own assignment operator here however so that will be called. The issue here is that there is a problem in the assignment operator. How it should work when it is called is that the object on the left side is represented inside the function as 'this' and we set the variables 'this.v' for instance to be equal to that of the values in the reference that is passed in. However, all we do is here is return a copy of B(-b.v). This means if we were to (assuming we implemented a print function) do std::cout << b1 = b2 << std::endl that would print the new object but b1 would not be set.

9. **0 1 1**

Same thing here as above for b1 and b2, b1's value is not changed by the = operator. For bP1 though it doesn't use the assignment operator it just points at the bP2 object which does have a value of 1.

10. **7 1**

Even when we set references equal it does not change what the reference points to. Once a reference is assigned to an object it will not change what is points to.

11. **0 -1**

As xchange takes objects we use the copy constructor to copy those objects to be sent to the function. Normally we would want a copy constructor to take a const reference so it does not change what is passed in but here we do not have that. The copy constructor in fact modifies what it is making a copy of, changing its v value to the negative version of what it was. Whatever happens in the function that uses objects does not affect our original objects in the main but they were still made negative by the copy constructor.

12. **0 -1**

Here we have the same issue caused by the tricky assignment operator from before. Even though you might think that the values should be changed as they are passed by reference, due to the assignment operator not working properly each of the lines does nothing.

13. 7 1

Here also due to the faulty assignment operator nothing is changed and the values are as they were in question 10.

14. -100 -100

Here we have bP1 and bP2 being sent in, which if we remember from earlier were already pointing to the same object. Our assignment operator is ineffective as we already talked about in other problems but here we are not even using it as we are assigning pointers not objects. Not that it even matters however, as it just swaps pointers that were already pointing to the same thing. As these are pointers when a change is made to the object it does indeed affect the object residing in the main, causing the object that both pointers point to to be set to

15. 9

Here the + operator is called which functions as expected, returning a new object that is equal to the two values in n1 and n2 added together.

16. 3

Here we go into the - operator. It initializes a new object with a value of -3. notice that in the constructor (and again after in the function for some reason), the abs() function is called, which sets any negative value to go back to a positive value. For this reason the new object holds a value of 3.

17. 18

Here the function works as expected, returning a new object equal to the two object's n vals multiplied.

18. 2

Here there is a bit of trickiness going on. Note that whenever a member operator is called that what is on the left is the object whose function we are inside and what is on the right is passed in as the reference. This means that n1/n2 effectively does n2.val / n1.val, which is 6 / 3 and therefore 2.

19. 3

Here none of the operators have modified n1 so it still has its original value of 3.

20. Err

This is an error as we only have a constructor with C(int). Normally there is a default constructor if you do not declare any constructor yourself. However once you declare one that takes a parameter there is no longer a default constructor.

21. B::f1

Here Base has a virtual f1() but the C object it is pointing at has not overridden it, so B's f1() is called.

22. Err

While C does have a public f2() B does not have a public f2() only a private f2(), and therefore the B class pointer does not think it should be able to call this and errors out..

23. B::f3

For this C does have its own virtual f3() function, but in B f3() is not virtual and therefore does not use the virtual function table in its implementation. For this reason B's f3() is called as that is all the B pointer knows how to call.

24. C::f2

As this is a C pointer it knows that it is able to call its public f2() function and does so.

25. C::f3

Here even though f3() is not virtual in base, as this is a C pointer it knows about C's f3() and calls that.

26. B::f4

Here we are able to use a B pointer to call B's f4() function as C does not override it. C is derived from B and therefore we are able to pass c1 in to something that takes a B& reference as it has a B object inside of it..

27. Err

Here even though C has a f5() function, B does not have a f5() function and therefore has no idea what to call even though as it is a B pointer.

28. B::f1

Here we have a B reference which points to a C object. C does not have any version of f1() so we just call B's f1() here, but if it did C's f1() would indeed be called if we are referencing it this way.

29. Ok

This is Ok as C inherits from B.

30. B::f1

When we set objects equal all that we are doing is copying the values from one to another, it will still completely behave like a B object. Here B's f1() is called. Even if C did implement a f1() function though this still would have just called B's f1().

31. B D

When a new D is created the constructor first calls super() which calls the constructor of the class which it extends (B). Note that if the super() wasn't there explicitly it would still be there implicitly.

32. B::f1

Static functions are always tied to the class, never to the individual object. Therefore no virtual behavior will happen here, as it is a B reference we are using it will call B's static f1() function.

33. D::f2

All public functions in Java are by default virtual. As we are actually pointing to a D object due to how virtual behavior works, when B goes to call its f2() function it will instead call D's f2() function because it has overwritten B's f2() VFT entry.

34. B::f3 B::f4

Here we call B's f3() because even though D also implements f3() it has a different parameter and therefore it is overloading, not overriding. This will not cause virtual behavior. When we go to call f4() on the b object that is passed in, you may think that it would fall D's f4() as b is really a D object. In Java however, all private functions are non-virtual. Therefore as our B reference thinks that f4() is not virtual it will not perform virtual behavior and just call B's f4().

35. Err

f4() is a private function so the B reference is not able to call it like this as we are not inside of the class.

36. Err

Even though D has implemented f5() B has not implemented f5(), and therefore has no idea what to call. This results in an error.

37. D::f1

Here we have a D reference so it will always call the D f1() function as it is static and therefore just tied to the class itself.

38. D::f2

Here D has implemented its own version of f2() and therefore it will call it.

39. D::f3 D::f4

Here D has implemented its own f3() so that will be called. D will also call its own private version of f4() that it has made.

40. Err

f4() is a private function so we are not able to call it from outside of its class.

41. D::f5

Here even though f4() doesn't exist in B we have a D reference to a D object so we know about f5() and are able to call it.

42. B

This is Ok it just calls B's constructor and prints the appropriate message.

43. 2 2

Earlier in the program we set b.i which was originally 0 to 2. i is a static variable and is therefore shared between all instances of the class, therefore both the newly created object and the old object see the same static i which has a value of 2.

44. 2

Here can reference static variables like this as they belong to the B class.

45. Ok

This is ok it implements everything that the interface requires of it.

46. Err

This j could either be the j from I1 or the j from I2, as it is ambiguous this is an error.

47. E::f1

This function is able to run successfully as E only needs to implement one f1() and it will satisfy both interfaces.

48. E::f3

This is alright as E is allowed to have its own f3() function even if the interfaces don't specify it.

49. 2

D is not a valid class unless it implements everything that the interfaces told it to implement.

50. Err

Here out of the three options we can first eliminate f1(float, ddouble) and f1(float, int) as neither could take a double in their first argument. We then look at f(double, short) and the int could not fit into the short so none of this functions are able to match what we are passing in.

51. f1(f,d)

Here all three functions could take a float as their first argument. Looking at the double however, only f1(float, double) can fit that and the others have a second parameter that is too small.

52. Err

Here our first argument is a float which could fit into any of these so it is alright. When we look at the second argument it is a short which could also fit into any of the f1() implementations. Here we can see that f1(float, int) could have its second parameter widened to fit f1(float, double) and as it can be widened to fit in to f1(float, double), we can eliminate f1(float, double) from our potential candidates. Now we are left with f1(float, int) and f1(double short). f1(float, int) could widen its float into a double but not the int into a short. f1(double, short) could widen its short into an int for f1(float, int) but its double cannot go down to a float. Therefore we are left with two candidates who do not cancel each other out and therefore have an error.

53. f1(f,i)

Here f1(int,int) could not fit into f1(double, short) because a short lacks the space for an int. We are left with f1(float, double) and f1(float, int). As we stated in the previous answer f1(float, int) can widen into f1(float, double) and therefore cancels it, leaving us with f1(float, int) as our choice.