Fall 2019 Test 2 Key w/ Explanations

1. **Ok**

B derives from A so we may have A point to a B as any B object contains an A object inside of it.

2. **Err**

C derives from A, which means there is information in C which an A object doesn't have, so this gives a compile time error as this isn't something which should happen.

3. **Ok**

As long as the A pointer is casted as a C pointer this will allow the line to pass but this may lead to issue later if we actually try to use any functionality of a C object while we are really pointing at an A.

4. **Ok**

This line is allowed as even though it is an A pointer, this A pointer could be pointing to a C object and since a static cast only checks at compile time it doesn't know what the pointer points to so it allows this possibly legal cast.

5. **Ok**

We can not set a C pointer equal to an A pointer that points to an A object. The dynamic cast checks and sees that this is illegal. While it is illegal that doesn't mean it is an error, as it is a dynamic cast it will just set cP1 to null.

6. **Ok**

Ok I thought this was supposed to be Err and I'm not sure why it is Ok as I thought that cP1 should be null at this point. ¯_ (ツ)_/¯ maybe when we took the test he said aP2 should have been initialized to new C(), in which case the dynamic cast would have been allowed.

7. **(0)**

w0 has its val initialized to 0 and when we go to print it, it uses the overloaded << operator to print with parentheses around the number and it uses getValue() which is able to grab the val of the Weirdmath object which is 0.

8. **(1)**

The + operator actually returns an object which is initialized to a value of the val of the left object minus the val of the right object which is 2 - 1 = 1. Then (1) is printed.

9. **(2)**

Here we only change the value of w0 but actually ending up printing w2 which still has its original value of 2, so we print (2).

10. **#2#**

Here we have overloaded the + operator to work with the ostream class. Therefore whenever we have a Weirdmath object on the left and an ostream object on the right it will call our overloaded operation which prints out the value of the Weirdmath object with #s around it.

11. **No**

val is private and therefore if any function outside the class wants to access it, it must be a friend. If the function was no longer a friend it would get a compile error of trying to access a private variable where it didn't have permission.

12. **Yes**

This function would be alright as it does not try to directly access the private variable but instead uses a getter function.

13. 1 1

Vec v1(2) means that initially the values were [0, 1] inside of values[]. When we call v1.incValue(0) it increments the value at position 0 by 1, resulting in [1, 1]. As v1R is a reference to v1 and its values[0] is the same as v1's values[0], 1 1 is printed here.

14. 1

Here we pass v1 to a function and the copy constructor is used to make a copy of it for the function to use as one of its arguments. As this copy is properly done where it has its own new array, the changes made to the copy do not affect the original v1 and we keep the same values[0] value (not that it would have changed values[1] anyway.

15. 1

Here this negate takes a reference so the changes made do really affect the v1 in the main, but what has been changed is values[0] and we are printing values[1] which was not changed.

16. -10

v1R is a reference to v1 so any changes made to base object v1 will show when you try and use the reference.

17. -10 1

When we set v1 = v2 we go through the assignment operator. This sets the values inside of v1 to [2, 3] assuming that the array is initialized with zeros when it is created but these values don't really matter for any of the following outputs, all that matters is that it creates its own separate new array to use instead of referencing the same array as what it is being set equal to. Changing values[1] of v1 does not affect v2 as they do not share the same array so -10 1 is printed.

18. -10 -10 1

This is the same thing as above. v1R = v2 doesn't mean that the reference now points at V2, it means that it tries to assign the object that v1R references equal to v2 through the assignment operator just like what happened above.

19. Base Derived

The base must always be constructed before the derived class so Base is printed in Base's constructor then Derived's constructor is able to print Derived. Even if we don't explicitly call super() it will implicitly be the first line of our constructor.

20. Base(1) Derived(1)

Here we explicitly call our super's constructor with 1 in Derived's constructor so first Base(1) is printed then Derived's constructor continues and prints Derived(1).

21. 1 1

si is a static variable so there is only one copy that belongs to the class. Therefore even though Q19 sets it to 0, Q20 overwrites it and it is now 1.

22. Err

foo is private and therefore can not be called from this context.

23. bar(int) B: foo

bar is overloaded to take a int, Base, or long. There is no way for a short to become a Base so that is removed, leaving us with bar(int) or bar(long). As an int can be widened to a long, we can remove bar(long) and that leaves us with bar(int) being called. This then calls foo().

24. bar(Base b) B: foo

As we have a bar(Base) we can treat the Derived reference as a Base reference as each Derived contains a Base within it. Therefore we call bar(Base) which goes on to call foo().

25. Base:f1

Here we call Base's f1() and there are no issues.

26. Der:f1

public methods are by default virtual, so when our Base reference now refers to a Derived object which overwrites that f1(), we end up calling Derived's f1().

27. Base:f2

Our f2() is also virtual but is not overwritten in Derived so we still just call Base's f2().

28. Err

We are a Base reference that does not know what a f3() is so this is an error.

29. Err

This is an error as foo is private and can not be called like this, it would need to be public.

30. Der:f1

Here the Derived object is able to call its f1() method.

31. Base:f2

The Derived class does not have its own f2() so when an attempt to call f2() is made it will just call the f2() of the Base which it derives from.

32. Der:f3

Here the Derived object is able to call its f3() method.

33. Err

A Derived reference can not be set equal to a Base reference as Base is a parent of Derived and therefore does not have all the information necessary for a Derived object.

34. Err

Base is an abstract class and can therefore not be instantiated.

35. Err

I1 is an interface and interfaces can't be instantiated.

36. Ok

Derived implements both foo() and bar() so it can be instantiated.

37. foo

Here we can call the foo() function which Derived implemented.

38. Err

We have a Base reference here which does not know what f1() is so this is an error.

39. Ok

We can create a Derived object.

40. Err

Here it is ambiguous if we are referring to I1's constI or I2's constI so this is an error. If we wanted to print one of the values we could choose I1.constI or I2.constI.