1. **A**

This line creates a new A which goes and calls A's constructor therefore printing "A".

2. **A B**

This line creates a new B. When a derived class is to be constructed it must construct its parent object before it constructs itself, therefore it runs A's constructor to build its parent object which prints "A", and then is able to run its constructor which prints "B".

3. **a1 a6**

As the A pointer is pointing to an A object it will call A::a1() which is ok since it is a public function and this prints "a1". It then calls a6() which is allowed because even though it is a private function we are inside of the class it belongs to so we may access that function.

4. **a1a a7**

As the A pointer is pointing to an A object it will call A::a1(A*) as this is the definition of a1 which matches what we called the function with, this is called function overloading. This prints "a1a" in a1(A*) and then goes on to call a7 which it is allowed to do because even though it is private we are inside of the class that that private function belongs to.

5. **a4**

As the A pointer is pointing to an A object it will be able to call this a4() function and print "a4".

6. **Err**

Here we are not able to access A's a5 function because it is protected. This means that we could only access it from inside of class A or any class that derives from it and currently we are trying to access it from outside publicly.

7. **Err**

Here we are not able to access A's a6 function because it is private. This means that we could only access it from inside of class A and currently we are trying to access it from outside publicly.

8. **a1 B::a6**

Here we have an A pointer pointing at a B object. When we go to call a1() it is indeed virtual but B does not override the a1() function so our call will just call A's original a1() function and print "a1". We then go to call a6() with our A pointer pointing at a B object. Even though a6() is private it does not mean that virtual does not apply to it, any classes which derive from A may still override it. B has a definition but it says "void a6()" and does not say virtual in front of it. No matter whether it is explicitly said or not, for any derived class which implements their own version of a virtual function their function will be virtual even if they didn't say so. Therefore B has overwritten a6() and when the pointer goes to call a6() its VFT will end up calling the code for B's a6() and therefore it prints "B::a6"

9. **a1a a7**

Here we have an A pointer pointing to a B object. When we call a1(A*) it is a virtual function but B has not chosen to override it so we will end up calling A's original a1(A*) function and print "a1a". We then go to call A's a7() function which is not virtual so even if it had been defined in B, which it is not, we would have just called A's a7() and printed "a7".

10. **b::a2**

Here we have an A pointer pointing to a B object. When we go to try and call a2() we will end up calling the code for B's a2(). The reason for this is as stated earlier that even though in B it only says “void a2()” and does not say virtual in front of it, it must be virtual as the parent it derived from had that function listed as virtual. Therefore in its VFT it has overridden what code gets called when the A pointer tries to call a2() in the VFT and it will end up calling B’s a2() and printing “B::a2”.

11. **a3i**

Here when the A pointer goes to call a3(A*) even though it is virtual it will end up calling A’s original a3(A*) because B does not implement its own version of a3(A*).

12. **a4**

In this call while our A pointer is really pointing at a B object, we will not see it call B’s a4() function. The reason for this is that a4() is not virtual and therefore when it is trying to call a4() it does not do so through the VFT, so it will always end up calling A’s a4().

13. **Err**

Even though we are pointing to a B object, the A pointer has no idea what a b1() function is so this line results in an error because it doesn’t know what code to call.

14. **A B C**

As stated previously when a derived class wants to construct itself it must construct its parent first. This means that all parents need to be constructed before it can construct itself. Even if we had a long chain of inheritance like E isa D isa C isa B isa A it would end up constructing all of them and printing A B C D E when we try and construct an E class..

15. **a1 B::a6**

Here as the B class has not made its own definition of a1() it will call its parent’s definition and print as in #8.

16. **a1a a7**

Here as the B class has not made its own definition of a1(A*) it will call its parent’s definition and print as in #9.

17. **B::a2**

As we have a B class pointer here it will go to call B’s a2(). This a2() is virtual because its parent’s a2() is virtual but C has not chosen to implement its own version of a2() so B’s a2() will be called.

18. **B::a3d**

Here we exhibit what is known as hiding. Even though there may be a better fitting definition of a3() in A, when we call a3() we look for any acceptable in our own class before ever looking at the parent. This means any type of number we will put into a3() with a B pointer will use B’s a3() because we could change that number to fit a double and thus never consider looking at how A has defined a3().

19. **Ca4**

Here since we are a B pointer pointing at a C object we will end up calling C’s definition of a4(). a4() is a virtual function in B so even though C does not put the word virtual in front of where it says “void a4()” this function is still virtual. This means that it has overridden a4()’s VFT entry and so when B goes to call its VFT entry for a4() it will end up calling C’s a4 and printing “Ca4” instead”.

20. **b1 a5**

Here as we have a B pointer pointing to a C object and the virtual function b1() has not been overridden in C we will end up calling B’s b1() function and print “b1”. Even though a5() is virtual neither B not C implements their own version so we will end up calling A’s a5(). Here we are allowed to access it even though it is protected because we are inside of a class that derives from A so we are allowed to access the protected function.

21. **Err**

The B class pointer has no idea what a c1() is so it gives an error as it is not sure what code it should be calling.

22. **Err**

This is an error as a5() is a protected function. Here we are trying to access it from outside the class while protected means that it can only be accessed from inside of A or any of the classes which derive from it.

23. **uninitialized**

When we go to initialize A::A(1) we have to make sure we know in what order an initializer list is gone through. Here we have z(p), y(1), x(z), but this is not the order that we do those at all. The order of initializer lists is set based on the order they appear in the declaration. This means we will set x to z, then y to 1, then z to p. As we are setting x to z before z is ever initialized this means that x will have an uninitialized value.

24. **Err**

This access is illegal as class B inherits a protected A. this means that all members of A which were public are now protected and they can not be accessed here because we are not inside of B or a class that derives from it.

25. **Err**

Here we are not able to access y as y is a protected member of A and we are not inside of A or a class that derives from it.

26. **uninitialized 1 3 1**

Here C call’s B’s print which calls A’s print. Remember that we are allowed to access static members from inside any of A’s non-static functions but if we were in a static function we could not access any of A’s non-static members. Back when we discussed the A constructor in 23 we determined that it will “set x to z, then y to 1, then z to p” when p was 1. Therefore we print “uninitialized 1 3” and then it will print “1” as numPrints was initially zero and we had just incremented it to be 1.

27. **~B ~A**

Here we pass a C object to a function that expects a B object. This is ok because a C object contains a B object and therefore we can pass that part of our object to it. Due to this function taking an object as a parameter we will call the copy constructor and make a copy of the B portion of our object for it to use. When this function ends that object is no longer needed so the destructor is called. Classes must be destructed from the bottom up, and therefore we first call B’s destructor which prints “~B” and then A’s destructor which prints “~B”

28. **~C ~B ~A**

Here we pass a C object to a function that expects a C object. It takes an object as a parameter so we use the copy constructor to create a copy for that function to use. When that function is over it must call the destructor and the object’s classes must be destroyed from the bottom up.

Therefore C's destructor will be called and print "~C", then B's destructor will be called and print "~B", and finally A's destructor will be called and print "~A".

29. **Err**

Here A has "virtual void f( )=0;" and "virtual void g( )=0;". We can tell because of the "=0" at the end that these are pure virtual functions, making A an abstract class. What pure virtual means is that any class which wishes to be able to create an instance of itself must implement these functions before we allow them to do that. Note that you could implement f() or g() in A and be ok but then it would be quite weird to have chosen to make them pure virtual.

30. **Err**

The B class has implemented f() but has not implemented g(). It must implement all pure virtual functions before it is allowed to create an instance of itself.

31. **Ok**

The C class has defined g() and it has a definition of f() from it's parent class B, therefore we are allowed to create an instance of C.

32. **Ok**

An A* pointer is allowed to point to a C because the C class derives from A and therefore has an A inside of it.

33. **f**

Here because f() is virtual when A goes to call f() which it is allowed to do, B will have overwritten that spot in the VFT and A will end up calling B's implementation of f which prints "f". Note that even though it only says "void f()" in B and does not say virtual, because its parent is virtual it is virtual regardless of it explicitly saying so.

34. **g**

Here because g() is virtual when A goes to call g() which it is allowed to do, C will have overwritten that spot in the VFT and A will end up calling C's implementation of g which prints "g". Note that even though it only says "void g()" in C and does not say virtual, because its parent is virtual it is virtual regardless of it explicitly saying so.

35. **f**

Here C has not defined its own f() function but it has access to its parent B's f() function so it is able to call that and prints "f".

36. **0**

Here "i" was initialized to zero earlier and has no reason to have changed? I think I didn't notice that we forgot to call barI() as was possibly intended. Oops. Note that if we had called barI() i would not have changes as it would have sent a copy.

37. **4**

Here we call barO() with our C object. Note that the C object sent to the function is a copy of the object and not the object itself. This function sets x to 4 and then prints it so we will print "4" here.

38. **10**

When we constructed the C object it can be seen in the initializer list that we set x to 10. When we called barO() it sent a copy and not the original. This means that the change we made has not affected our original copy of the object and it still holds an x value of 10.

39. **4**

Here we send a pointer to the C object and set its x to 4, we then print the value of that x so of course it is “4”.

40. 4

Due to the fact that we sent a pointer, it was pointing to the same object that we have in our main. This means that any changes made to it changed our original object. Therefore when we print x here it is not 10 because it was changed to 4 in the function, and we print “4”.