1a)Pass by val so any changes in funcs wont effect outer val:
baz
bar
1
1b)Pass by ref so any changes in funcs will also effect the outer val:
baz
bar
4
1c)Pass by object ref so assignments within functions will change the object the parameter points to, but will not cha nge the val of the object the parameter originally
pointed to so:
baz
bar
1
1d)Pass by need, so since the bar function never needs b, it never calls on baz which was passed in as b. Also, a gets the expression tree computed by the functions
which is eventually called when print(a) is called
bar
4

2)
Optionals are best used when there is a single point of failure that can easily be recovered from. Exceptions are used for errors that are more rare but can also be
recovered from. Optional errors place their burden on the users of the function. Exceptions on the other hand place the burden on the person who coded the exception
as they must handle the thrown exception outside of any function(s) that may throw an exception. In this scenario, t here is only a single point of failure (not
finding the value) and thus should be handled by optionals.

3a)
bar(0) -> tries foo(0) -> switch throws range error -> inner try cannot handle it so exits to outer try -> can handle run time errors which super of range error:
catch 2
I'm done
that's what I say
Really done!
3b)
bar(1) -> tries foo(1) -> switch throws invalid arg err -> caught by inner try since can handle logic errors as super of invalid arg:
catch 1
hurray!
I'm done
that's what I say
Really done
3c)
bar(2) -> tries foo(2) -> switch throws logic error -> caught by inner try:
catch 1
hurray!
I'm done
that's what I say
Really done
3d)
bar(3) -> tries foo(3) -> switch throws bad exception error -> not caught by inner try -> not caught by outer try -> ba ck into bar func -> caught by base exception class:
catch 3

3e)
bar(4) -> tries foo(4) -> switch breaks -> no exceptions so no catches
hurray!
I'm done!
that's what I say
Really done!

4a)
We cannot use interface types to define concrete objects because they have no implementation to construct a concret e object off of. The reason we can use interface types
to to define object references, references, and pointers is because none of those create a concrete object. They all poi nt to or alias another actual concrete object.

4b)
Interfaces are not necessary in dyncamically typed languages as we do not care about the type of a variable, we just c are about the operations it supports. For this
reason dynamically typed languages rely on duck-typing where we can use two variables equivalently in code as lon g as they both support the functions required for our
operations.

4c)
We want to use interface inheritance over hybrid inheritance when two objects arent related, but can support the sam e group of behaviors. We may also want to use
interface inheritance to avoid the fragile base class scenarios where changes to the implementation in the base class may break subclass implementations.

5a)
I would expect both print statements to print 43 as shared_count is a class variable.

5b)
The class variable does not come into existence until an object of the class comes into existence which then created a nd initialized the class variable. Once it has
been created it no longer needs to be initialized so the creation of more objects will not call initial_value() again. Ie i t is only called once.

5c)
Instance functions are able to access members of an object since it is secretly passed in the object itself while class/st atic functions are not tied to a particular
object.

6)
When a function is called, a new lexical environment is formed with all the arguments passed in as formal parameter s. However when this is done for a instance function,
the object itself is secretly passed in as the formal param this/self.

7)
In this case composition with delegation is better since we only want to use the implementation of the vector class, a nd not inherit its interface. Since a stack is
essentially just a vector where you can only interact with the last element, we have no need for the interface of vecto rs beyond the functions necessary to
interact with the final element.

8a)
Classes that inherit interfaces dont recieve any implementations. This means there would not be any conflicting imp lementations when we inherit multiple interfaces.

Even if both interfaces we inherit from contain a function with the same name there are no conflicts as there is only one implementation, the one the class itself defines.

8b)
As stated above, there would be no problem as there would only be a single implementation of the function foo.

9a)
The main benefit to defaulting class methods to static is safety. It ensures that any function that is safe to be overrid den is allowed to do. The benefit to defaulting
class methods to virtual allows us to quickly overwrite the base class versions of the function so that the subclass ver sion may implement a version of the function that
is more useful to it.

9b)
The bark() function would be referenced in cpp's vtable as it is the only virtual function