Question 1

(a) Template approach:
Some benefits of the template approach is a reduction in type-errors since it will do compile-time type-checking. Additionally, since teplates are evaluated at compile-time, it allows for compiler optimizations. Some drawbacks include longer compilation times since everything is evaluated at compile time. Moreover, templates are prone to complex syntax.

(b) Generics approach:
Some benefits of the generics approach is that it will do basic type checking at compile time, catching errors earlier on. Generics have a shorter compile time since types are substituted during runtime. Some drawbacks of generics is that since types are substituted during runtime, there may be a performance hit during runtime due to type checking.

(c) Duck typing approach:
Some benefits of duck typing is that it allows for more flexibility, since all you need is the function signature to match. This allows for code reusability since the function signature just needs to have the correct parameters. Some drawbacks of duck typing may be a performance hit during runtime, since duck typing will dynamically check the function signature every time a function is called. Additionally, duck typing may be confusing since the wrong function may be called or the expected behavior of the function could be different from what actually happens.

Question 2

Dynamically typed languages cannot support parametric polymorphism since they do not have types assigned to variables, but rather have types assigned to values. Therefore, dynamically typed languages have no way of knowing (until runtime) what types the parameters of a function are, so it cannot determine during compile time which function to bind to.

Question 3

We can use template metaprogramming to simulate duck typing. This way, we can set specific parameters for certain functions. This is similar to duck typing since we rely more on the function signature rather than their explicit types. It is different to dynamically typed languages since it will still type check at compile-time. Some pros of using template metaprogramming over duck typing is the benefit of static typing (compile-time type checking). One drawback of this approach is that the code base can get bloated and may be confusing to read since it requires playing around the type system.

Question 4

(a) C++ templates are generally safer since using void pointers can be risky as it may or may not type check and even if it does, it may not error correctly or it may error when it's not supposed to. Some pros of the void pointer approach is that it allows for truly generic types since void pointers will point to anything. Additionally, it allows for backwards compatability. Additionally, void pointers may allow for some dynamic behavior wheras templates are static (for the most part). Some cons of the void pointer approach is that void pointers will have to perform type conversions every time, which can affect runtime performance.

(b) This approach is similar to generics in languages like Java since there is more type flexibility (types are assigned dynamically). Additionally, the void pointer approach allows for code reuse. Some differences to generics are that generics will perform type conversions automatically whereas the void pointer approach will not (you have to explicitly convert). Generics in Java also have garbage collection whereas the void pointer approach has manual memory management.

(c) Use a template.

Question 5

I would go about creating uniform sets of objects by first defining a constructor function that will define the attributes of the object. Then, factory functions will call the constructor and return a new instance of the "object", along with a common set of functions that each object should have.

Question 7

Only pointers. This is because IShape is an interface, and therefore has no function implementations itself, just function declarations. Hence, an IShape object cannot be instantiated. However, IShape pointers can be created since they can be pointed at any derived class of IShape that implements the virtual functions.

Question 8

Interfaces aren't used/needed in dynamically typed languages since they have duck typing and dynamic type checking. Therefore, since variables have no types assigned to them, when the values are evaluated, the types just have to be compatible in order for the function to work. Additionally, duck typing is essentially an implementation of a virtual function defintion. Hence, dynamically typed languages don't use/need interfaces.

Question 10

One example of when I would prefer interface inheritance is when I want to create a general template for a group of objects, not necessarily related, to share a common behavior. For example, the area of a shape varies by shape, but they all have an area. An example of when I would prefer subtype inheritance is when my subtype wants to extend my base class. An example being: A student inherits from a Person since all people have $x,\ y,\ z$ traits, but students also have $classes,\ GPA,\ etc.$