## **AnnotatedWrapper**

1 - Class Description)

Advanced AnnotatedWrapper class would be designed to provide a flexible tagging system for objects. It'll allow annotations in different formats beyond just strings. The goal is to support multiple tag types such as strings, numbers, key-value pairs, etc. This will allow for greater flexibility in how objects are categorized and stored, making it more data-base friendly and efficient for querying. Overall, the class will provide a template-based annotation system, efficient retrieval of annotations, and support filtering and searching annotations dynamically.

2 - Similar Classes)
std::unordered_map<K, V>: Used to store the unique tags and store annotations efficiently
std::variant<Types…>: To store multiple types safely
std::any: If a fully dynamic type system is needed
std::optional<T>: Help avoid exceptions when retrieving annotations

3 - Key Functions)
Advanced Functionality on top of current:
New member variables like font size, background color, text color, border type, etc.
Void listAnnotations() const; - Prints all annotations, handling different types correctly.
Void clearAnnotations(); - Clears all stored annotations

4 - Error Conditions)
Accessing a non-existent annotation
Type mismatch when retrieving annotation
Duplicate keys (overwriting)
Removing an annotation that doesn't exist
Empty key names

5 - Other Class Projects)
DataMap (Group 2 - Ritchie): This class maps names to arbitrary data types, similar to the planned key-value annotations. If integrated with AnnotatedWrapper, this could be used to store structured metadata across different entities (Like how a TagManager entry that stores associated numerical values).

## **AuditedPointer**

1 - Class Description)

Advanced AuditedPointer class would add an ID to every AuditedPointer that is active in the program that is outputted when error conditions are met to help track down where the error is attributed to. The mechanism for tracking memory leaks in the program of having a counter would be changed to having a set of active AuditedPointers that persists within every object. If there is a memory leak at the end of the program the ID of the leaked objects are shown to help improve tracking methods and ensure problems are located quickly.

2 - Similar Classes)

Shared_ptr, set or map for AuditedPointer storage, std::allocator for memory management

3 - Key Functions)

LeakChecker::CheckForLeaks() - Now checks over whole set instead of just counter variable
Aptr::Constructor and MakeAudited() - Now adds a casted void pointer to the set of active AuditedPointers
Aptr::Delete() - Ensures the right Aptr from the set is deleted (found using ID). Error conditions noted
Aptr::Reset() - New function that deletes and removes all active Aptr objects?
Aptr::GetActiveAptrs() - New function that returns a list of active Aptr objects and their IDs
Aptr::GetID() - Returns ID of that Aptr object

4 - Error Conditions)

- Invalid access into the set when searching for deletion (Programmer error)
- Out of bounds access to the set during insertion or searching (Programmer error)
- Resource depletion for the active set (Recoverable error)
- Invalid casting leading to a nullptr being added to the set (Programmer error)
- Double deletion from the set of active Aptrs (User error)

5 - Other Class Projects)

- OutputLog (Group 7) - Could be used in conjunction with my class to better learn exactly where memory issues are a problem and to provide a more detailed explanation of Aptrs and their location within the program.
- All other Audited classes (Vector, String) as they all work to check after coding whether functionality works as intended. The current implementation may have methods to track active instances that could be of use to learn.