Initial C++ Class Writeups

MemoryFactory

1. The name of this class is MemoryFactory. The purpose of this class is to provide fast, predictable memory allocation for many objects of the same type by using a fixed-size pool allocator. This is intended for games where objects are frequently created and destroyed such as agents, and where reducing allocation matters. The factory will manage memory in blocks and also maintains a free list of available slots. It will have functions that will pull a slot from the free list as well as a destructor that will destroy the object and return its slot to the free list.

2. std::allocator<T> is a similar class to the one I will be creating. The STL separates the two capabilities that my class will do, which is to allocate raw memory and construct objects inside that memory.

   std::vector<T> is also a similar class. Although it is not an allocator, it shows the idea of capacity growth that I will be using

   std::unique_ptr<T> and std::shared_ptr<T> is a similar class because it enforces the idea of deleting objects.

3. Constructor: creates a memory factory for a single type. Takes in how many objects should be stored in each memory block.

   Create: a function that requests memory for one object of the factory's type. Constructs the object in place and returns a pointer to that object.

   Destroy: a function that takes a pointer to a previously created object and calls the objects destructor. Returns the memory back to the factory to be used again later.

Allocate: a function that allocates a block of raw memory large enough to hold multiple objects.

Get block size: a function that returns how many objects each block can hold.

Get number of allocated blocks: reports how many memory blocks the factory has so far

Factory destructor: releases all memory blocks owned by the factory.

4. Destroy being called with a pointer not from this factory. This would be a programmer error.

   Destroy being called twice on the same pointer. This would be a programmer error.

   Passing a nullptr to destroy. This would be a programmer error.

   The system running out of memory during pool expansion. This would be a recoverable error.

5. An expected challenge in creating this is determining how much to expand the pool by. Another is being able to efficiently track which memory slots are available.

6. This class should coordinate with the classes of TagManager and AnnotationSet in Group 1, BehaviorTree, WorldPath, and PathGenerator in Group 2, and Scheduler and EventQueue in Group 14.

FeatureVector

1) The FeatureVector class represents a fixed vector that supports many different numeric features for an object, where each feature has an index and meaning throughout all FeatureVectors. It supports fast access and operations like similarity/distance, weighted scoring, and selecting objects that maximize/minimize feature combinations. The class emphasizes speed by keeping features in a contiguous structure and enforcing consistent indexing.

2) Some of the different STL types that I can use

   Std::vector<double> -- for contiguous numeric storage and indexing.

   std::span<const double> -- non-owning view

   Std::unordered_map <std::string, size_t> (name → index mapping, if needed elsewhere)

   <numeric> and <algorithm> for dot products and different reductions.

3) Constructor: Creates a FeatureVector using a predefined feature schema and a list of feature values.

   Size: Returns the number of features in the vector

   At: Provides safe, bounds-checked access to a feature value.

   Dot: Computes the dot product between two FeatureVectors.

   Normalize: Scales the vector so its magnitude becomes 1.

4) Values size does not match schema feature count is a programmer error.

Comparing/scoring with different schema/IDs is also a programmer error.

Index out of range in operator[] is a programmer error or a runtime error for the at method
Non-finite values on construction is a user/input error.

5) Challenges
   Deciding ownership model (own values vs view via span)

   Performance: caching norms, SIMD opportunities, avoiding repeated allocations

   Numerical stability (normalization, handling zeros)

6) This class should interact with WorldPath from Group 2, DataMap from Group 14, and WeightedSet from Group 15.

Annotation Set

1) The Annotation Set represents a collection of string-based tags, or annotations, that are attached to a single object. These objects could be an agent, item, place, etc. The goal is to let the AI system store and query small pieces of memory about an object.
   a. High level functionality includes storing unique tags
   b. Supporting fast operations of adding and removing tags, etc.

    c. Integrate with TagManager so that queries remain fast
        i. When a tag is added/removed, AnnotationSet notifies TagManager so TagManager can keep an index from tag to objects
2) Standard Library classes utilized
    a. Std::unordered_set<std::string>
        i. Utilizaed for unique tags
    b. Std::unordered_map / std::map
    c. Std::vector<std::string>
    d. Std::string, std::string_view
        i. Represents tag values
3) Key Functions
    a. Construction & Ownership
        i. AnnotationSet(Object owner, TagManager* manager = nullptr);
        ii. Void AttachManager(TagManager* manager);
    b. Tag Ops
        i. Bool AddTag(std:string_view tag);
            1. Return True if created, False if already there
        ii. Bool RemoveTag(std::string_view tag);
            1. Return True if removed, False if cant be found
        iii. Bool CheckTag(std::string_view tag);
            1. Check if tag is present

4) Error Conditions
    a. Empty Tag or Invalid Format Tag
        i. Type 1 Programmer error
        ii. Handling: assert(!tag.empty())
    b. Tag already exists
        i. Type 3 User condition error
        ii. Returns False from AddTag()
    c. Removing a tag that does not exist
        i. Type 3 User condition error
        ii. Returns False from RemoveTag()
    d. Memory Allocation Failure

        i. Type 2 Memory Allocation Error
        ii. Handling: std::bad_alloc
    e. Inconsistencies with TagManager
        i. Type 2 Memory Allocation Error (Most Likely?)
        ii. Handling: Design TagManager's update_tag() function to "not-throw" exceptions. Keeping it in sync with AnnotationSet

5) Expected Challenges
    a. Performance tradeoffs for tag storage
        i. Large sets benefit from unordered_sets, however regular vector sets work better for smaller sets
    b. String ownership
        i. Full string storage for object is expensive
        ii. Could modify TagManager to store strings anbd AnnotationSet can store integers for better storage utilization
    c. General Consistency with TagManager
        i. We need to keep both in sync, where AnnotationSet leads and TagManager follows, or vice versa

6) Compatibility with other groups
    a. Constant synchronization with other elements of the product.
        i. AnnotationSet needs to communicate its data and ensure everything is in line, and vice versa
    b. Group 1's TagManager
        i. This is a given and discussed earlier
    c. Group 14's DataMap
        i. Set regulations as to who records what to ensure there are no duplicated records that take up storage space
        ii. Ex: AnnotationSet could hold boolean and/or integer tags, DataMap can hold other type values
    d. Group 2's ActionMap

i. If we utilize runtime commands, we could add to the debug console:
    1. Tag add <id>
    2. Tag Remove <id>

TagManager

1) Description: A class that centralizes and manages tags that work with *AnnotationSet*. While *AnnotationSet* owns an object's local set of tags, *TagManager* owns the global inverted index of objects that belong to the tag in question. Ex: tag -> {Obj1, Obj2, Obj3}. This helps enable quick searches for objects in a specific tag or group of tags.

   High-level Functionality:
   - The class will maintain a global index of the tags.
   - It will support fast query searches when a tag or multiple tags are needed.
   - It will work closely with *AnnotationSet* whenever tags are added or removed, so the index can stay consistent.
   - It will use *ObjectID* (value-type identifier) to represent the object to avoid lifetime issues

2) Std Library classes:
   - std::string
     - Stores tag values.
   - std::string_view
     - Used for query inputs (Since storing query inputs is not necessary).
   - std::vector<std::string>
     - Query input lists.
   - std::unordered_map<std::string, T>
     - Maps each tag to the collection of objects that contain it.
   - std::unordered_set<ObjectID>
     - Stores the object IDs for a given tag

3) Key Functions
   - Constructor: Initializes an empty global tag index that will be populated incrementally as *AnnotationSet* instances attach to it and add or remove tags.
   - OnTagAdded: Update index so owner is included in the set of tag/tags. (Detailed tag adding will be handled by *AnnotationSet*)
   - OnTagRemoved: Update index so owner is removed from the tag/tags in question. (Detailed tag removing will be handled by *AnnotationSet*)
   - TagQuery/Find: Search for objects by using their tags (include, exclude, or all)

4) Error Conditions
   - Empty tag/Invalid tag format: When a tag is used incorrectly or there is no tag. The program will handle this through ex: *assert(!tag.empty)*. (Programmer Error)
   - Memory Allocation Failure: As the index grows larger, it might run out of memory. *std::badalloc* can handle this exception. But we must make sure this error handling is also in *AnnotationSet* or else tags mightbe inconsistent. (Recoverable error)
   - There should also be exceptions for adding and removing duplicate tags. But *AnnotationSet* already handles them.

5) Expected Challenges
   - Using *unordered_set* makes checking fast but uses a lot of memory overhead.
   - The use of *string_view* and *string* must be managed and handled properly.
   - Same challenge as *AnnotationSet*. Both *AnnotationSet* and *TagManager* need to be in sync and must handle exceptions together to avoid inconsistency
6) Outside Classes to Coordinate
   - Obviously *AnnotationSet* as stated earlier.
   - *BehaviorTree*: It will most likely model the agent's behavior depending on the tags of objects. Like find nearest enemy (tag = “enemy”) etc. The

BehaviorTree will call *TagManager* to get the objects with the corresponding tags.

- *DataFileManager*: If the *DataFileManager* logs world-state metrics, then it can use *TagManager* as a data source for logging tagged objects each time Update() is called.
- *ActionMap*: It can use tags to count the entities for debugging or other action purposes.
- *PathGenerator*: If it needs to calculate the shortest path needed for an object with a specific tag. Locations might be counted as entities with tags if made to do so. It will need access to *TagManager*. If that is the case, then *WorldPath* might need *TagManager* as well for the same reason.

Robhinhood map

Description: robinhood map is a high-performance container that stores key value pairs using open addressing with robin hood hashing. Its goal is to provide much faster lookups and insertions than std::unorderd_map. It works by keeping probe lengths short and memory access cache friendly. This class is indented for performance critical systems where fast access matters more than iterator stability.

High level functionality:

- Stores key value pairs in bucket array using open addressing
- Uses robin hood hashing to minimize variance in probe lengths during collisions
- Supports fast lookup insert and deletion operations
- Automatically resizes and rehashes when the load factor exceeds a threshold

Std library classes:

1. Std::hash<k> - used to generate hash values for keys
2. Std::vector<> used to store buckets in memory
3. Std:: pair<x,y> - represents stored key value pairs
4. Std::optional<> - used to represent empty or erased bucket states

Key functions:

Constructor: initializes the map with a bucket count and load factor

Insert: inserts or updates a key value pair using robin hood hashing

Find: searches for value associated with a key and returns a pointer or iterator

Erase: removes a key value pair and preserves probe invariants

Contains: returns whether a key exists in the map

Rehash: expands the bucket array and reinserts all elements

Size: returns the number of stores elements

Expected conditions:

1. Invalid or non-hash table key types: programmer error and should be caught at compile time
2. Memory allocation failure during rehashing: recoverable error and should throw a bad allocation std
3. Erasing or finding a key that does not exist user level condition and should return a special value like false
4. Using iterators after insertion: programmer error (document early)

Expected challenges:

1. Correctly implementing robinhood displacement and probe distance tracking

2. Handling deletion while maintaining correct probe order
3. Choosing appropriate load factor thresholds for performance

Outside classes to coordinate:

Tag manager, Annotiation set - > these classes rely on fast key-based lookups

Scheduler, Eventqueeue : use this map for efficient event or tasking indexing

Behavior tree – could use robinhoodmap to efficiently associate behaviors with keys or identifiers.

Vision for the Agent Module

The goal for the AI Agent Module is to act as a flexible decision-making machine through game rule behaviors, data of the game, and the obstacles it will need to face.

The agent will receive constrained perception of the world along with a set of current viable actions. The agent's 'perception' is expressed using the **FeatureVector's** numeric state information and the tags in the **AnnotationSet** for entity context. The **TagManager** will allow the agent/model to quickly find and generalize objects and the possible combination of tags. These will serve as the inputs for our learning models (TBD) and will be trained overtime for better decision making.

Overall, the AI agent is designed to be trained repeatedly in simulated Worlds and deployed into real game environments as flexible, adaptable, autonomous agents.