Write-up for Initial Class Development

Group 3: Hopper

FixedPoint

1) **Class Description**

- Provides a fixed-precision representation of decimal values using an integer-based underlying type. This allows precise arithmetic operations without the rounding errors associated with floating-point numbers (float and double). Also supports arithmetic operations, conversions, and comparisons while maintaining a fixed precision.
- Specific goals include ensuring predictable and deterministic behavior for computations, providing an interface similar to standard arithmetic by overloading operators, providing easy conversion, and optimizing performance compared to double when possible.
- High-level functionality includes storing numbers as scaled integers, supporting operators while preserving precision, and enabling conversion between arithmetic types.

2) A list of **similar classes** in the standard library that you should be familiar with or use to inform the functionality you will be developing.

- std::ratio — Represents rational numbers.
- std::chrono::duration — Maintains precision.
- std::numeric_limits<T> — Can define precision constraints for safe conversion.
- std::fixed — Can format output.
- std::optional<T> — Could handle invalid operations.

3) A list of **key functions** that you plan to implement. This does not need to be an exhaustive list, but it should give a strong indication of how the class should be used.

- Operators for arithmetic (+, -, *, /, +=, -=, *=, /=) and comparison (==, !=, <, >, <=, >=) that return the value with fixed point representation
- Conversion functions for int and double
- Functions to return the internal integer representation

4) A set of **error conditions** that you will be responsive to. For each, indicate if it's source was (1) *programmer error*, such as invalid arguments to a function, which should be caught by an assert, (2) a potentially *recoverable error*, such as a resource limitation, which should trigger an exception, or (3) a *user error*, such as invalid input, which needs to simply return a special

condition for the programmer to respond to (note: these can usually be handled with exceptions as well if you prefer.)

- Dividing by zero — Programmer error
- Arithmetic overflow or underflow — Potentially recoverable
- Precision loss when converting from double — User error
- Invalid scaling value for conversions — Programmer error
- Large double values exceeding integer range — User error

5) Any **expected challenges** that you will be facing, and especially any extra topics you need to learn about. This information will help me make sure to cover topics early in the course or else provide your group with extra guidance on the project.

- No particular challenges come to mind. I recall the concepts behind two's complement and IEEE 754 from CSE 320 decently, but I'll gladly take any tips or suggestions for things you think I should look out for.

6) A list of any **other class projects** from other groups that you think you may want to use, either in the development of your own classes or in your final application.

- BitVector — Group 4
- Assert — Group 4
- ErrorManager — Group 6
- StaticVector — Group 10

FunctionSet

1) A **class description**, including what it's goals are, and what it's high-level functionality should look like. This does not need to perfectly match the description you were given, but it should be in the same general spirit unless you confirm any changes with the instructors ahead of time.

This class is designed as a container to hold multiple callable objects and all share the same signature. The primary goal of this class is to let the program store and invoke a **set** of functions at once using a single call. For more advanced functionality, it should have something like collecting outputs (when the function return type is not void): The program might retrieve the outputs of all functions in a container (example: std::vector<ReturnType>)

2) A list of **similar classes** in the standard library that you should be familiar with or use to inform the functionality you will be developing.

There is no direct standard library that hold sets of functions in this manner, but there are several components that inform or facilitate this design:

- **std::function<Signature>**: A general-purpose wrapper for any callable (function pointer, lambda, functor). Likely to be the return type of this container
- **std::vector<T>**: A dynamic array that can store elements of type T; in this case, combine with std::function<Signature>, it could be something like std::vector<std::function<Signature>> for the internal storage.
- **std::bind/Lambdas**: Tools that let you adapt function calls to match a desired signature. Not a direct competitor, but relevant to this design.

3) A list of **key functions** that you plan to implement. This does not need to be an exhaustive list, but it should give a strong indication of how the class should be used.

At this moment:

- Constructor / Destructor
    - To initialize the empty sets of functions.
- callAll(Args … args)
    - To call all the stored functions at once.
- Optional:
    - addFuntion()
        - To stored the function in the set
    - removeFuntion()
        - To remove the function in the set
    - clean()
        - Clean all the functions in the set
    - countSet()
        - Return the number of functions stored in the set

- isEmpty()
    - Bool to see if the set is empty or not
- findFunction()
    - See if that function existed or not

4) A set of **error conditions** that you will be responsive to. For each, indicate if it's source was (1) *programmer error*, such as invalid arguments to a function, which should be caught by an assert, (2) a potentially *recoverable error*, such as a resource limitation, which should trigger an exception, or (3) a *user error*, such as invalid input, which needs to simply return a special condition for the programmer to respond to (note: these can usually be handled with exceptions as well if you prefer.)

1. Programmer error:
    a. Invalid Function Signature happens, it will usually be a compile-time error. Thus, compile-time error or static_assert, no need for runtime error.
    b. Delete a function that does not exist in the FunctionSet. In this case, using an assertion in debug mode could catch programmer mistakes.
2. Recoverable error:
    a. Resource Exhaustion, adding too much of functions and leads to out-of-memory. If std::bad_alloc is thrown by the internal container, it can be caught by the user. This is a standard exception scenario.
3. User error:
    a. Often belong inside the functions being called, which can return special codes or throw exceptions.

5) Any **expected challenges** that you will be facing, and especially any extra topics you need to learn about. This information will help me make sure to cover topics early in the course or else provide your group with extra guidance on the project.

1. **Template usage**, ensure the class is properly templated to handle different function signatures
2. **Memory Management**, which deciding how to store function objects
3. **Exception Handling**, when a single or few functions is throwing exceptions, how to handle that.
4. **Performance Considerations**:
    a. If calling many functions repeatedly
    b. Inlined function pointers or functors might be faster
    c. Possibly investigating ways to reduce overhead if performance is critical

6) A list of any **other class projects** from other groups that you think you may want to use, either in the development of your own classes or in your final application.

1. Random, WeightedSet >> Group1
2. ActionMap >> Group2
3. CSVfile / DataGrid >> Group5
4. ArgManager / CommandLine >> Group6

5. OutputLog (keep track of the program, in this case can be the output and the expectation of the functions in the sets) >> Group7
6. EventQueue, EventManager >> Group9

Circle

1) A **class description**, including what it's goals are, and what it's high-level functionality should look like. This does not need to perfectly match the description you were given, but it should be in the same general spirit unless you confirm any changes with the instructors ahead of time.

This class will hold the graphic information, position, radius of the circles in our program as well as identifying if the circle is overlapping with another circle. The primary focus is of the individual circle itself as well as its own boundaries, with minor inclusion of interaction between like circles.

2) A list of **similar classes** in the standard library that you should be familiar with or use to inform the functionality you will be developing.

The most important classes needed for this class are firstly any graphics libraries the final program may use. The circles will often need to be used by many other classes in the program, primarily the surface class as it will make up the shapes stored in the surface. As characteristics are added to the circles the fixed point as well as the datatracker class to hold important attributes the circle may need.

3) A list of **key functions** that you plan to implement. This does not need to be an exhaustive list, but it should give a strong indication of how the class should be used.

- Constructor/Destructor
- Function to check overlap of other circles
- Circle modifier function: radius, speed, etc..
- Misc characteristics the circle may have, ex circle is set to rock for rock paper scissors example

4) A set of **error conditions** that you will be responsive to. For each, indicate if it's source was (1) *programmer error*, such as invalid arguments to a function, which should be caught by an assert, (2) a potentially *recoverable error*, such as a resource limitation, which should trigger an exception, or (3) a *user error*, such as invalid input, which needs to simply return a special condition for the programmer to respond to (note: these can usually be handled with exceptions as well if you prefer.)

4. Programmer error:
    a. Circle position being out of range of surface
    b. Circle size being out of range of surface
5. Recoverable error:
    a. In general pushing the implemented graphics beyond what the program or computer can handle
6. User error:
    a. Assuming circles can be spawned in by the user, attempting to add circles outside the surface area
    b. Filling the entire surface with circles, this should be limited by a maximum circle count

c. Assuming circles could be modified, giving circles invalid characteristics: out of range position, too large diameter, etc..

5) Any **expected challenges** that you will be facing, and especially any extra topics you need to learn about. This information will help me make sure to cover topics early in the course or else provide your group with extra guidance on the project.

Graphic implementation: using a graphics library that can not only handle a sufficient amount of circles that are moving on screen but optimizing it in a way that will allow a bunch of moving circles on screen. Also storage of the characteristics of each circle, if each individual circle has bunch of different characteristics that need to be kept track of this could become difficult with 100+ circles on screen that are interacting with each other.

6) A list of any **other class projects** from other groups that you think you may want to use, either in the development of your own classes or in your final application.

- Productivity dungeon (graphics)
- Graph generator and analyzer (database)
- Ecology simulator (gameplay functionality)

Surface

1) **Class description.**

Surface will be a class that will act as an “arena” or container of shapes. The class will handle shapes moving inside of it, along with being able to detect when shapes bump into each

other/overlap. Sectors seem like the best choice for actually doing this and has good speed and efficiency.

2) **Similar classes**.

A few classes that will be used are vectors, it will hold all of the shapes that will be in the zone. Pairs or tuples will also be good for storing overlaps and positions. A queue could be useful for overlap detection as well. Vectors or shared pointers could also be useful for position tracking, we will see which is better after weighing the pros and cons of each. Sector will be good as well, as mentioned in the specs

3) **Key functions.**

Get a list of shapes in/on the surface (however you want to say/interpret it). Get a list of overlapping shapes, some sort of function for shape movement, and visualize will be useful later on too. There will need to be functions that divide up the surface into sectors, get sectors the shapes are in, and update the sectors. Remove and add shapes to the surface

4) **Error conditions.**

(1) Putting an invalid shape in, moving shapes that don't exist, duplicates, overlaps for shapes that don't exist.

(2) too much shapes/memory being used, unable to divide surface into sectors, overlapping not being detected.

(3) no shapes, adding shapes outside of the surface, removing non-existent shapes

5) **Expected challenges.**

My computer taking forever to run the program, memory management, updating, I will need to learn about sectors and how those functions work, will also have to do some refreshing on pointers

6) **Other class projects.**

BitVector, IndexSet, Assert, DataGrid, AuditedArray, Scheduler, Serializer (for web app part), EventQueue, EventManager, Image, WebLayout, MemoryFactor

DataTracker

1) **Class description.**

DataTracker will be a container that stores numeric values. This class will be able to return many different forms of statistical data from stored values and the ability to add or delete values as necessary at any given time.

2) **Similar classes**.

- Vector Class: Will be used to store the numerical values. This class will be similar to DataTracker.

- Statistics Class: has mean, median, stdev, variance, and mode functions. These functions will be useful in simplifying my class and can be used for the actual calculations.
- Collections class: has deque and counter functions. These could be useful for going through the values that have been collected.

3) **Key functions**.

- Mean: Returns the calculated mean of the values collected
- Median: Returns the calculated mean of the values collected
- Min: Returns the calculated minimum value of the values collected
- Max: Returns the calculated maximum value of the values collected
- Total: Returns the total amount of values collected
- Add_value: Adds a value to the class instance
- Delete_value: Deleted a value from the class instance

4) **Error conditions.**

- (1) If the instance of the class is empty/has no numerical values in it
- (1) If types that are not numerical try to get added to the class instance.
- (2) Running out of memory for the amount of data.
- (3) Calling functions like mean, median, min, and max when there is no data
- (3) Invalid input to future features like filtering values

5) **Expected challenges**

- Efficiency with Large Datasets, perhaps different data structures could be used to handle this
- Being able to update data as it changes in real time.
- Memory management issues with data sets too large

6) **Other class projects**

- DataGrid (group 5)
- IndexSet (group 4)
- Assert (group 4)
- ErrorManager (group 6)