Random

1. Person taking lead: Rachel Loren
2. Class Description
    a. Through the implementation of various functions, this class allows the user to generate a random number. The user can input a range of numbers to generate between, and a seed to use as the generator. If the user does not supply a range, the class will use a default range that depends on what specific function they are calling. If the user does not supply a seed, the program will use the current time off of the computer.
3. Similar Classes
    a. The standard library for c++ has a variety of pre-existing functions and classes to handle random number generation. The <random> class has a wide variety of different functions to be able to 'randomly' generate all sorts of things. The most common way to do this(and a method I have personally used several times before) is to create a std::random_device (this allows you to set the seed if you would like to), a std::mt19937 (this is the generator), and a std::uniform_int_distribution (this allows you to set a range to generate a number between).
4. Key Functions
    a. This class will allow the user to generate a random int, double, float, char or bool. As a result, there will be the following functions: int GetInt(int i_min = 0, int i_max = 100), double GetDouble(double d_min = 0.0, double d_max = 100.0), float GetFloat(float f_min = 0.00f, float f_max = 100.00f), char GetChar(char c_min = 'A', char c_max = 'Z'), and char GetBool(bool b_min = false, bool b_max = true). Additional classes can be added to handle other data types if other groups have a use for it. The class will also have the function bool P(double probability = 0.5), which will randomly generate whether it is true or not, and weight itself based on the given probability. In addition to all of these functions, the class will include the function void SetSeed(long long seed). This will allow the user to change and manually set the seed being used for the random generation. By default, the class will generate the seed based on the current time when the class is constructed. This function will allow the user to manually set and change this.
5. Error Conditions
    a. There are a variety of error conditions that the random class will account for. Each of the Get[Type] functions will have error handling to check and make sure that the user inputs the minimum and maximum values in the proper order. These are all considered programmer errors, as it is invalid input to put the maximum value before the minimum value. The called function will throw an exception to quickly alert the programmer that the input is invalid. Each of the Get[Type] functions also have the possibility to run into potentially recoverable errors if they attempt to generate a value that there is no space in memory for. In these cases, an exception will be thrown.
6. Expected Challenges

    a. There aren't any major challenges that I feel like I can definitely expect when creating this class. However, there might be challenges in learning how to apply something like the xoshiro shift algorithm to all of the different data types.
7. Other Group's C++ classes
    a. There aren't any other group's classes that the Random class will depend on. However, there are a few classes that might want to use the Random class, and I would want to hear from them on what data types/types of generation would be useful for them. I could see the following classes potentially making use of the Random class: TagManager from Group 1, BehaviorTree from Group 2, PathGenerator from Group 2, Scheduler from Group 14, DataLog from Group 16 and ActionLog from Group 16.

WeightedSet

1. Person taking lead: Abigail Franzmeier
2. Class Description
    a. This class provides a data structure for the user to store a collection of items that have a numerical weight. The user can add/delete weighted items to the collection. The collection tracks total weight and the user can index into the set to find an item by searching for any number between its weight range. The total weight of items before the current item is the beginning of the search range and the end is the total weight before plus the current item's individual weight.

3. Similar Classes
    a. Similar classes include containers like std::set, std::multiset, std::map, and std::multimap. These classes are made to have an underlying data structure that is a red-black tree, which is what I would plan to use to make this set.

4. Key Functions
    a. This class will have functions such as Insert(item, weight), remove(item), updateWeight(item, new weight), find_item(weight), get_total_weight(), size() (returns number of elements), and is_empty(). There will also be inner functions that the user won't need to interact with. The data structure will be a red-black tree so there will need to be rebalancing functions that run when user-functions run, such as update.

5. Error Conditions
    a. Programmer errors: Assertions can be used to check if red-black tree properties are violated after rebalancing occurs where there should no longer be any. Potentially Recoverable errors: Memory space could be limited which could cause errors when adding new nodes. User errors: invalid weights (user tries to enter item with weight < 0), indexing by weight out of range (total weight = 7.0, user searches for <0 or >7), updating/ deleting non-existent items.

6. Expected Challenges
    a. The main challenge will be speed, as a set of items will be large and we want to be able to return data quickly and accurately.

7. Other Group's C++ classes
    a. This class likely won't depend on another group's class. Within our group, random will be used to index into the set so that we can generate an items/rooms/enemies

DataFileManager

1. Person taking lead: Zhixiang Miao
2. Class Description
    a. Manage files and data, process files, information, data, and sanity check. Detecting bugs. Manage input/output files. Design and collect data such as values, abilities, levels of difficulty and player level requirement for specific area, bosses, and specify type and rarity of weapons.
3. Similar Classes
    a. <fstream> to stream files, to input and output files.
    b. <vector> to store values of dungeon levels, dungeon difficulty, weapon type, weapon rarity, type of enemy, level of enemy, ability of enemy, boss type, boss difficulty.
    c. <map> to map and connect the specific dungeon to its enemies together, and connect the chest reward to the difficulty of the dungeon.
4. Key Functions
    a. Test cases for programming files.
    b. openFile function.
    c. exitFile function.
    d. informationOfCharacter
        i. To show the specific information about the character.
    e. informationOfEnemy
        i. To show the specific information about the enemy. Design any types of enemies, bosses.
    f. informationOfWeapon
        i. To design the weapon type and its attributes and rarity.
    g. informationOfDungeon
        i. Level of the dungeon according to the player's level. Difficulty of the dungeon.
    h. informationOfChest
        i. Color/style of treasure chest to indicate the level of rarity of the chest (normal->white, rare->yellow, refined->blue, epic->purple, legendary->orange), which corresponds to the rarity of reward inside.
    i. Attributes

        i. Category of abilities in the game world, each with a specific value, the higher the stronger. The player's value of attributes can increase after leveling up or wearing armors, weapons.
5. Error Conditions
    a. Risk that the player might get the weapon with levels too high or too low (So I have to make sure I mapped them correctly).
    b. Risk that the player might enter the wrong dungeon (too high for their level or too low, not the difficulty level, since the difficulty level can be based on the dungeon's level and different difficulty levels can only increase the amount of enemies inside and variety of enemies, and traps inside the dungeon).
    c. Risk that the reward or chest reward might be mistakenly distributed. For example, we want the player to get the epic (higher percentage) or legendary (lower percentage) levels of weapons in a nightmare difficulty level dungeon, instead of normal level of weapons.
6. Expected Challenges
    a. Too much data to process, since I have to design all values (attributes, levels, powers of both the player and enemies) of the game world.
7. Other Group's C++ classes
    a. While this class won't require other classes to function, it will likely need collaboration to find out what format other groups want data stored in. Some examples may include(but are not limited to) TagManager, FeatureVector and MemoryFactory from Group 1; WorldPath, PathGenerator and MemoFunction from Group 2; DataMap and EventQueue from Group 14; and DataLog and ActionLog from Group 16.

StateGrid

1. Person taking lead: Paul Bui
2. Class Description
    a. A dynamic 2D grid where each position is one of a set of pre-defined states. Each state type should have a name, a symbol, and any other information that defines that state (perhaps as a DataMap). For example, if a StateGrid was used to represent a maze, you might give it two types: a "wall" with the symbol '#' and the property "Hardness" set to 20, as well as an "open" state with the symbol ' ' (space) and no properties.
3. Similar Classes
    a. <vector> for dynamically changing storage to hold the CellTypes(?), <cassert>/GTest for unit tests/debugging coordinate and tiles (catching errors in functions), <memory> for smart pointers in order to dynamically create/delete tiles when moving from room to room,
4. Key Functions
    - General Overview: Assuming we're going with a top-down dungeon crawler with the player exploring multiple procedurally generated rooms, this class will provide functionality to allow programmers to access/manage cell properties (height,

width, tile layout/properties) and modify the setup of the rooms. Although, this overview extends to other game types as well.

a. GetCellInfo()
    i. Definition: Adding this here for potential debugging purposes, group was discussing the idea of having multiple different tile types, from damaging tiles such as spike traps, environmental tiles (boulders, trees…) that player characters can't go through, or interactable items that give player items. Returns information of type of cell, properties of it, and any additional information discussing its interactability with the player/enemy character
b. IsActive()
    i. Definition: Whether or not tile map is active, a false state means that the map is redrawn
c. (Struct, not function) TileInfo()
    i. Definition: Properties of the Tile and its associated values
    ii. Example properties: wallState, isInteractable, tileType(enum), isInteractable
d. GetTileWidth()
    i. Getter for the Width of the grid
e. GetTileHeight()
    i. Getter for the Height of the grid
f. IsValid()
    i. Definition: Tests if specific coordinates/world coordinates are in range for the StateGrid, returns a boolean determining whether or not coords are within the world.
g. GenerateCellMap()
    i. Generates cell map, taking into consideration the different parameters of the cell map level. During group talks we discussed the idea of different tiers of difficulty as the player traverses, so this class would be responsible for taking in data info of the current tier and generating a tile-map based on that information. Parameters being the amount of different obstacles per map level, the amount of enemies/types of enemies, whether or not
h. AddCellType()
    i. Helper function that allows the configuration of a new type of cell that's generated within the cell grid. Parameters might include the name, description, and a symbol to represent that tile. For the example of the symbol parameter, if we were to represent the cell as an ASCII value, having a char type parameter to represent the char value would be used. Another example would be if we were to represent the tile as an image file, it'd be passed in as a string type (its file path directory) for the function to load the image.

5. Error Conditions

    a. The various listed functions above will have built-in checkers to ensure that the programmer does not misuse the function in any way that'll cause any unexpected behaviors or errors. For example, when generating a tile-set map, setting limits in place on certain parameters such as width and height in order to prevent egregious values from being put in and having an extremely large map beyond the scope of the standard map setting. In the case there's not enough memory for the tile-map to generate, an exception will be thrown. In the case that the certain agents are able to pass through tiles that should not be able too, an exception and a special message should be thrown.
6. Expected Challenges
    a. When approaching procedurally generated tile environments, some general challenges would be ensuring that neither player/enemy characters are able to walk through generating wall tile types. To prevent this, creating proper HitTest functionality and having built-in boolean checkers to ensure that the player agent can't cross certain tiles is necessary. In addition, creating unique and varying kinds of rooms for players to explore that have unique/changing shapes and environments. Map patterns in the way that not every room is going to be a rectangular block with maps being different shapes in addition to having environmental foliage to it. Creating a function that can generate these in a consistent way taking into consideration of tiles that might conflict with each other is something that would need focus on. For example, we'd want to avoid having obstacles that would block the way of the door to the next area the player character needs to head through.
7. Other Group's C++ classes
    a. This class likely won't rely on classes from other groups. Although it may need to use the Random class to generate information of the rooms such as the traps, environmental features, and room shape. It'll need some collaboration between groups, however, for what gui interface platform the project will be taking place on (wxWidgets, SDL, GTK) and how we want to represent tiles within the game.

StateGridPosition

1. Person taking lead: Joey Hyun
2. Class Description
    a. Tracks the position and orientation of a single agent (player, enemy, or other) within a StateGrid. It stores the coordinates and direction comparing positions with other agents. Allows for collision detection, pathfinding, and relationship checks between agents and objects in the game world.
3. Similar Classes
    a. Std::pair - can store two coordinate values
4. Key Functions
    a. StateGridPosition(int x, int y, orientation dir =)
        i. Constructor to initialize positions and optional facing direction
    b. Int GetX() and int GetY() - getters for current coordinates

    c. Set x and Set y - setters for coordinates
    d. Orientation - returns current facing direction
    e. Move() - Moves the position of the agent
    f. Bool operator() - compares if two positions occupy the same cell
5. Error Conditions
    a. Monster/NPC/Player agent moving out of bounds
    b. Tracks wrong coordinates or wrong direction
6. Expected Challenges
    a. Main challenge will be ensuring this class works well with StateGrid for validity checking. The class only tracks position data however, move, may need to verify if the destination is valid within the grid. Also making sure each agent in the current map is accurately tracked may cause some issues.
7. This class likely won't depend on another group's classes. However, within our group, DataFileManager may be used to save data.

Vision for the Main Module

Group 15 is the 'Generative World' group. We envision creating a rouge-like dungeon crawler to achieve this. The major part of the module will be the procedurally generated 'rooms'. Walking from one room to the next will generate the next 'room'. There will be different types of rooms, here is a list of the ones we are currently planning on: treasure rooms, monster rooms, NPC rooms(merchants), boss monster rooms, and generic/save rooms. Treasure rooms, and potentially other types of rooms, will be able to generate their own treasure. It could be weapons of different rarities, types and values of each category, money or trinkets that can be sold but take inventory space, and consumable items like health kits. Each room will also be able to generate its own environment. This will include traps, stuff blocking paths and secret doors(that take you to better rooms).

As you proceed through the rooms, you will proceed through 4 different 'areas' that increase in difficulty as you progress through the game. To signify the change, the background visuals, types of monsters and loot will change as you progress. For example, you might start in a forest, proceed into a cave, then into a dungeon, and finally into a castle. (very rough concept sketches are included at the end of this file).

Some different systems that will need to be implemented is an NPC merchant character that sells and buys items, an inventory UI (a very rough concept sketch is included at the end of this file), and a player character. We would like to use actual art for everything if possible - and have a team member with experience drawing for fun who is willing to draw it.

Overall Module C++ Classes to Collaborate with

While each individual class have their own classes to collaborate with other groups on, there are some things that the module as a whole will need to collaborate. Group 2's classic agents would be needed for the NPCs and Player character. Group 1's AI Agents module would be needed potentially for enemy monsters. Group 14 focuses on building different systems for the world. We'd want to collaborate with them on building an inventory system for the player agent to

interact with. We would want to collaborate with Groups 17 and 18 to have actual graphics and things appearing on the screen.

Level Concept Sketches

Inventory Concept Sketch