```cpp
void output_Minority_Game_Attendance_History(int NUM_STRATEGIES_PER_AGENT, int NUM_DAYS_AGENTS_PLAY, unsigned int AGENT_POPULATION, int
NUM_INDICES_IN_STRATEGY) {
    //initialization
    auto strategy_scores = TwoDimensionalVector(AGENT_POPULATION, NUM_STRATEGIES_PER_AGENT, 0);
    auto binary_history = MarketInitialization::binaryMarketHistoryGenerator(NUM_INDICES_IN_STRATEGY);
    auto market_history = MarketInitialization::marketHistoryGenerator(binary_history, AGENT_POPULATION);

    for (int i = 0; i < NUM_DAYS_AGENTS_PLAY; ++i) {
        market_history.push_back(
                StrategyManipulation::market_count(strategy_scores,
                                                   binary_history,
                                                   NUM_INDICES_IN_STRATEGY,
                                                   NUM_STRATEGIES_PER_AGENT));
        StrategyManipulation::binaryHistoryUpdate(binary_history,
                                                  market_history.back());
        strategy_scores = StrategyManipulation::strategyScoreUpdate(AGENT_POPULATION,
                                                                    NUM_STRATEGIES_PER_AGENT,
                                                                    NUM_INDICES_IN_STRATEGY,
                                                                    binary_history,
                                                                    strategy_scores);
    }
}
```

Above is the basic simulation's outline; to run through the basic minority game, without any evolutionary dynamics, with the procedural generation of strategies in C++. This is a tedious explanation aimed at myself for troubleshooting and anyone who doesn't know programming, or just wants a step-by-step tutorial on the minority game. Below we walk through what's happening.

1. Strategy Score Declaration: Here we create an empty two-dimensional vector of height equal to agent population and breadth equal to number of strategies per agent (usually 2), and call it strategy_scores.
2. Binary History Declaration: Creates a one-dimensional vector and fills it with random elements (+/-1), calls it binary history.
3. Market history declaration: needlessly creates a series of random values between +/-AgentPop that correspond to the binary history; i.e. for a binary history value of -1, it creates a value in the range between (-AgentPop, 0)
4. For-Loop: Loops through the number of iterations we would like the game to run (in this case, called NUM_DAYS_AGENTS_PLAY)
5. *Market Count* Evaluation: Here the overall goal is to find what happens when we aggregate all the choices the agents make based on the best preforming strategy available to them, and then add that number (the total sum over all agents individually best available choices) to the end of the market history vector. This therefore requires a description of what happens to evaluate all the strategies for every agent, and sum them;
    a. *Market count* is called, which runs through a for loop, which for every agent looks at all their strategy's values and chooses the highest value one) this is done by identifying the index of the highest number in the vector associated with that agent (i.e. column associated with the row that identifies the agent in the two-dimensional strategies vector).[1] Market count then evaluates the prediction of each strategy by calling *evaluate strategy* with the corresponding 'best preforming' index.
    b. *Evaluate strategy* uses the two unique values given it; the strategy index and the binary market history, and returns a random bit. No matter how many times this function is called, it will always deterministically give the same bit for the same input, and does this through pure computation.[2] First the binary history last memory digits are converted to its decimal equivalent (-1 -> 0 in the binary history interpretation)[3] and then used as a meta strategy index, which we combine with the strategy's strategy vector index[4] to give us a unique seed for a procedural bit generator. However, in order to ensure the randomness of the eventual seed, I could not simply add or multiply both raw values (as then there would be overlap in values for low values of meta-strategy indices and high values of index seed and vice-versa) and instead must combine them in a way that deterministically produces another, random value. Though I am certain that there is ample room for improvement here, I did this by simply creating two mt19937 rng generators with each unique seed, and then generating one value from each, which I then compare to the predefined uniform int gen size to generate the +/-1 that corresponds to that index combination.
    c. *Market Count* then adds each agent's best strategy's evaluation and returns that value, which will range between +/-AgentPop, as each agent adds +/-1 to the total. This value then constitutes the market history.
6. Binary History Update: This simply takes the sign of the last element of the market history, as just determined by evaluating all best strategies and adds it to the binary history vector. This ensures that the functions that rely on the memory of the agents will be using up to date, accurate memory information.
7. Strategy Score Update: This function runs through every element of the strategy score vector and evaluates it via the *Evaluate Strategy function* (explained in step 5b). If the evaluation would have lead that agent to be in the minority, then the strategy is given a point (+1) , and -1 if not, thus updating the internal scores of each agent, preparing the program to run through the simulation again from step 5 as many times as requested.

[1] Initially, as you will have of noted, all the values in the strategy vector will be zero, as it was initialized to zero. However, for each iteration, all the strategies are updated (rewarded or punished with a +/-1) according to the accuracy of their prediction, and so this step gains importance, choosing the best preforming strategy to pass onto the *evaluate strategy* function.

[2] No memory retrieval is used, and this is more generalizable when different agents with different memory lengths begin to appear in the evolutionary version.

[3] For example, For m = 3, and a binary history of […,-1, 1, 1, -1, -1], we'd take the last 3 digits [1, -1, -1] and convert them to [1, 0, 0], which is 8 in decimal, and use that 8 as our meta strategy index

[4] Which is found by simply running through the two-dimensional vector as if it were one-dimensional, and labeling all the values incrementally, thus assigning a single value to every element.