```cpp
/*
    B)
    For part 2 it's the scenario as before, however now we use a retrospective approach. First we get an aggregate rate by summing up
    all of the rates (bike arrival, class1/2/3 arrivals). We are able to do this due to superposition ( If we have two independent Poisson
    processes with rates a and b respectively, then the combined process of the arrivals from both processes is a Poisson process with rate
    a + b). Once we generate the poisson random number for a given time unit, we use a uniform number generator + the weights of the original
    events to determine the order of the events. To achieve this we use std::discrete_distribution which allows you to generate a value given
    a specific weight. The weight used is the rate of the original event. We ran the simulation 10000 times, and then averaged the totalMoney
    at the end of each run.
*/

#include <iostream>
#include <random>
#include <queue>
#include <time.h>

struct Client
{
    int type;
};

int main()
{
    const int T = 120;
    int X[121] = { 0 }; //There are T+1 events
    const double bikeArrivalRate = 6;
    //clients have rate r1 = 3, r2 = 1, r3 = 4
    const double clientRates[4] = { 0, 3.0, 1.0, 4.0 };

    //client class 1/2 pay annually (K1 = 0.5, k2 = .1), total amount is (K1*r1 + K2*r2)
    //class 3 pays per ride amount k3 = 1.25

    //when annual members (class 1/2) arrive at empty station, there is penalty c1 = 1.0, c2 = 0.25, c3 = 0
    const double clientPenalty[4] = { 0, -1.0, -0.25, 0 };

    //create and seed the generator
    std::default_random_engine generator;
    generator.seed(time(0));

    //aggregate poisson
    std::cout << "Aggregate Lambda is : " << bikeArrivalRate + clientRates[1] + clientRates[2] + clientRates[3] << std::endl;
    std::poisson_distribution<int> poissonRandomVariableGenerator(bikeArrivalRate + clientRates[1] + clientRates[2] + clientRates[3]);
```

```
/*  std::discrete_distribution produces random integers on the interval [0, n),
    where the probability of each individual integer i is defined as the weight of
    the ith integer divided by the sum of all n weights. */
    std::discrete_distribution<> weightedDistributionEventGenerator({ bikeArrivalRate, clientRates[1], clientRates[2], clientRates[3] });

    const int numberOfTrials = 10000;
    double averageMoneyAmount = 0;

    std::cout << "Starting the trials" << std::endl;

    for (int t = 0; t < numberOfTrials; t++)
    {
        //client queue
        std::queue<Client> line;

        //we can assume total money starts at 0 + the deterministic annual prorated charge of clients classes 1 and 2
        double totalMoney = (0.5 * clientRates[1]) + (0.1 * clientRates[2]);
        X[0] = 10; //we start with 10 bikes at X(0)

        //for every X[i] to X[T]
        for (int i = 1; i <= T; i++)
        {
            X[i] = X[i - 1]; //new time interval starts with bike amount from prev interval
            int generatedValue = poissonRandomVariableGenerator(generator);
            //std::cout << "Generated p.r.v : " << generatedValue << std::endl;
            for (int rEvent = 0; rEvent < generatedValue; rEvent++)
            {
                //we don't actually care about the actual time an event happened on the interval, we only care about
                //the order in which they happen, so generate a u.r.v. {0: Bike Arrival, 1: Class1, 2: Class2, 3: Class3)
                //I don't think it would make a diff if I generated the event times first, sorted them by arrival, and then classified
                //since the classification itself uses uniform generation + weights, so just generate the events
                int eventType = weightedDistributionEventGenerator(generator);

                if (eventType == 0) //a bike has arrived
                {
                    X[i]++; //increment bike amount
                }

                //distribute the bikes to any clients waiting
                while (!line.empty() && X[i] > 0)
                {
                    auto client = line.front();
```

```
                    line.pop(); //remove from queue

                    X[i]--; //decrement bike count
                }

                if(eventType != 0) //a client has arrived
                {
                    //if no more bikes, add to queue, else decrement bike count
                    if (X[i] == 0) line.emplace(Client{ eventType });
                    else X[i]--;

                    //if a client arrives and there are no bikes
                    if (X[i] == 0)
                    {
                        //add the client into the queue
                        line.emplace(Client{ eventType });
                        //we apply a penalty for waiting in line, for class3 penalty is 0
                        totalMoney += clientPenalty[eventType];
                    }
                    else
                    {
                        X[i]--; //otherise just give the client a bike
                    }

                    //pay per ride charge for class 3
                    if (eventType == 3)
                    {
                        totalMoney += 1.25;
                    }
                }
            }
        }

        std::cout << "Total Money at the end of experiment " << totalMoney << std::endl;
        averageMoneyAmount += totalMoney;
    }

    std::cout << "Average amount of money over " << numberOfTrials << " iterations" << " : "
        << (averageMoneyAmount / numberOfTrials) << std::endl;

    return 0;
}

/*
    C)
    For this problem the retrospective approach was much better than the tick based approach. The main reason for this is the speed of the
    simulation. When using bernoulli trials to approximate a poisson distribution we must select a sufficiently small interval, meaning the
```

```
    number of bernouli trials run per poisson time unit needs to be very large.
*/
```