```cpp
#include <iostream>
#include <random>
#include <algorithm>
#include <iomanip>
#include <vector>
#include <utility>

//Function Prototypes
std::vector <std::pair<double, int>> generateNormalDistributionRandomNumbers(double, double, int, int);
std::vector <std::pair<double, int>> generateUniformDistributionRandomNumbers(double, double, int, int);

std::vector<std::vector<std::vector<std::tuple<std::string, int , int>>>> classificationOfClass(const std::vector<std::tuple<std::string,int,int>> &students, int groupSize, int numSections);
std::vector<std::vector<std::vector<std::tuple<std::string, int , int>>>> groupingOfStudents(std::vector<std::vector<std::vector<std::tuple<std::string, int , int>>>> &groups, int groupSize);
void printClass(const std::vector<std::vector<std::vector<std::tuple<std::string, int , int>>>> &objectGroups);
double roundToTwo(double num);
void makeHistogram(std::vector <std::pair<double, int>> &points);

void testGenerateNormalDistributionRandomNumbers();
void testGenerateUniformDistributionRandomNumbers();
void testStudentGrouping();

int main(){



    std::vector <std::pair<double, int>> pointsNormalDistribution{};
    std::vector <std::pair<double, int>> pointsUniformDistribution{};
    std::vector<std::vector<std::vector<std::tuple<std::string, int , int>>>> groupings{};



    std::cout<<"Enter the mean of the normal distribution: ";
    double mean{0};
    std::cin>>mean;
    std::cout<<"Enter the standard deviation of the normal distribution: ";
    double standardDeviation{0};
    std::cin>>standardDeviation;
    std::cout<<"Enter the number of samples for Normal distribution: ";
    int numSamplesND{0};
    std::cin>>numSamplesND;
    std::cout<<"Enter the number of bins for Normal distribution: ";
    int numBinsND{0};
    std::cin>>numBinsND;

//    Generating Normal Distribution Random Numbers
    pointsNormalDistribution = generateNormalDistributionRandomNumbers(mean, standardDeviation, numSamplesND, numBinsND);
    makeHistogram(pointsNormalDistribution);

    std::vector<std::tuple<std::string,int,int>> classOfStudents{
            {"Khalid", 1, 1},
            {"Jaylene", 1, 1},
            {"Diya", 1, 1},
            {"Hilary", 1, 2},
            {"Khaliesi", 1, 2},
            {"Dat", 1, 3},
            {"Sam", 1, 4},
            {"Elena", 1, 4},
            {"Miguel", 1, 4},
            {"Leo", 1, 5},
            {"Zack", 1, 5},
            {"Rene", 2, 1},
            {"Emma", 2, 1},
            {"Oliver", 2, 1},
            {"Liam", 2, 2},
            {"Ava", 2, 2},
            {"Benjamin", 2, 3},
            {"Charlotte", 2, 4},
            {"Amelia", 2, 4},
            {"Elijah", 2, 5},
            {"Harper", 2, 5},
            {"James", 2, 5},
            {"Sara", 1, 2},
            {"Lucas", 1, 0},
            {"Marie", 1, 0},
            {"Mia", 1, 0},
            {"Sophia", 2, 3},
            {"Noah", 2, 0},
            {"Isabella", 2, 0},
            {"William", 2, 0},
            {"Alice",1,0},
            {"Bob",1,0},
            {"Charlie",1,0},
            {"Daisy",1,0},
            {"Edward",1,0},
            {"Fiona",2,0},
            {"George",2,0},
            {"Hannah",2,0}
    };


    std::cout<<std::endl<<std::endl;

    //Generating Uniform Distribution Random Numbers
    std::cout<<"Enter the lower bound of the uniform distribution: ";
    double a{0};
    std::cin>>a;
    std::cout<<"Enter the upper bound of the uniform distribution: ";
    double b{0};
    std::cin>>b;
    std::cout<<"Enter the number of samples for Uniform distribution: ";
    int numSamplesUD{0};
    std::cin>>numSamplesUD;
    std::cout<<"Enter the number of bins for Uniform distribution: ";
    int numBinsUD{0};
    std::cin>>numBinsUD;

    pointsUniformDistribution = generateUniformDistributionRandomNumbers(a, b, numSamplesUD, numBinsUD);
    makeHistogram(pointsUniformDistribution);


    std::cout<<"Enter number of Sections: "<<std::endl;
    int numSections;
    std::cin>>numSections;
    std::cout<<"Enter number of members per group: "<<std::endl;
    int numMembers;
    std::cin>>numMembers;

    //Classifying students into sections and groups
    groupings = classificationOfClass(classOfStudents, numMembers, numSections);


    std::cout<<"Students in Group 0 are not assigned to any group yet"<<std::endl<<std::endl;
    std::cout<<"Initial grouping of students: "<<std::endl<<std::endl;
    std::cout<<"---------------------------------"<<std::endl<<std::endl;
    printClass(groupings);
    //Further grouping students

    groupings = groupingOfStudents(groupings, numMembers);



    std::cout<<"Final grouping of students: "<<std::endl<<std::endl;
    std::cout<<"---------------------------------"<<std::endl<<std::endl;


    printClass(groupings);

    testGenerateNormalDistributionRandomNumbers();
    testGenerateUniformDistributionRandomNumbers();
    testStudentGrouping();





    return 0;

}


// Function to round a number to two decimal places
double roundToTwo(double num)
{
    return round(num*10)/10;
}

//Function for getting frequency Points for Normal Distribution
std::vector <std::pair<double, int>> generateNormalDistributionRandomNumbers(double mean, double standardDeviation, int numSamples, int numBins)
{
    std::vector <std::pair<double, int>> tempPoints;



    //Preparing the vector for bins according to user specified bins, mean and standard deviation

    int startingPoint{0};
    int endingPoint{0};
    if(numBins%2 == 0)
    {
        startingPoint = -1*(numBins/2) + 1;
        endingPoint = startingPoint + numBins-1;
    }
    else
    {
        startingPoint = -1*((numBins-1)/2);
        endingPoint = startingPoint + numBins -1;
    }

    for(int i = startingPoint; i<=endingPoint;i++)
    {
        tempPoints.push_back(std::make_pair(round(mean+(i*standardDeviation)),0));
    }

    //Generating random numbers and categorizing them into bins

    std::random_device rd{};
    std::mt19937 gen{rd()};

    std::normal_distribution d{mean, standardDeviation};
    int currRandom{0};
    for(int n = 0; n < numSamples; ++n) {

        currRandom = round(d(gen));
        auto p = std::find_if(tempPoints.begin(), tempPoints.end(), [currRandom](std::pair<double,int> a){return a.first == currRandom;});
        p->second++;
    }

    //Returning the vector of bins
    return tempPoints;

}

std::vector <std::pair<double, int>> generateUniformDistributionRandomNumbers(double a, double b, int numSamplesUD, int numBinsUD)
{


    //Preparing the vector for bins according to user specified bins and range
    std::vector <std::pair<double, int>> tempPoints;

    double binCenter = (b-a)*1.0/(numBinsUD-1);

    for(int i = 0; i < numBinsUD; i++)
    {
        tempPoints.push_back(std::make_pair(roundToTwo(a+ i*binCenter) , 0));
    }

    //Generating random numbers and categorizing them into bins
    std::random_device rd;  // Will be used to obtain a seed for the random number engine
    std::mt19937 gen(rd()); // Standard mersenne_twister_engine seeded with rd()
    std::uniform_real_distribution<> dis(a, b);
    double currRandom{0};
    for (int n = 0; n < numSamplesUD; ++n) {

        currRandom = roundToTwo(dis(gen));
        auto p = std::find_if(tempPoints.begin(), tempPoints.end(),[currRandom](std::pair<double, int> a) { return a.first == currRandom; });
        p->second++;
    }

    //Returning the vector of bins
    return tempPoints;

}

void makeHistogram(std::vector <std::pair<double, int>> &points)
{
    //Scaling the histogram to fit the screen
    int maxBarLength{0};
    for(auto a : points)
    {
        if(a.second > maxBarLength)
        {
            maxBarLength = a.second;
        }
    }


    int unitLength = maxBarLength/8;

    for(auto &a : points)
    {
        a.second = a.second/unitLength;
    }

    //Printing the histogram

    for(int i = 8; i >=0; i--)
    {
        for(auto a : points)
        {
            if(a.second > i)
            {
                std::cout << std::setw(8) << "*" ;
            }
            else
            {
                std::cout <<std::setw(8)<< " ";
            }
        }
        std::cout << std::endl;
    }

    //Printing the x-axis

    for(auto a : points)
    {
        std::cout << std::setw(8) << a.first;
    }


}

// Function to classify students into sections and groups
std::vector<std::vector<std::vector<std::tuple<std::string, int , int>>>> classificationOfClass(const std::vector<std::tuple<std::string,int,int>> &students, int groupSize = 3, int numSections = 2){

    std::vector<std::vector<std::vector<std::tuple<std::string, int , int>>>> tempGroups(numSections);
    for(auto &a : tempGroups){
        a.resize(10);
    }
    //Checking each student and assigning them to a section and group
    for(auto a : students){
        int temp1 = std::get<1>(a);
        int temp2 = std::get<2>(a);
        tempGroups[temp1-1][temp2].push_back(a);
    }

    //Removing empty groups
    for(auto &a : tempGroups){
        while(a.at(a.size()-1).size() ==0){
            a.pop_back();
        }
    }

    //returning the vector of groups
    return tempGroups;

}

// Function to further group students
// This function takes a vector of groups as input and further groups students
// based on the number of free spaces in each group

std::vector<std::vector<std::vector<std::tuple<std::string, int , int>>>> groupingOfStudents(std::vector<std::vector<std::vector<std::tuple<std::string, int , int>>>> &groups, int groupSize = 3)
{

    //Iterating through each section

    for(auto &a : groups){


        int x = a.at(0).size();                     //Number of students who are not in any group yet in the current section
        std::vector<std::pair<int,int>> freeGroups;
        int freeSpaces{0};

        for(int i = 1; i < a.size(); i++){
            if(groupSize - a.at(i).size() > 0){
                freeGroups.push_back({i, groupSize - a.at(i).size()});  //Pushing the index of the group and the number of free spaces in that group
                freeSpaces += groupSize - a.at(i).size();                       //Calculating the total number of free spaces
            }
        }

        //If there are enough free spaces to accommodate all students who are not in any group yet, then assign them to the free spaces
        if(freeSpaces >= x) {
            std::cout << "There are enough free spaces to accommodate all students" << std::endl;
            while(freeSpaces > 0 && !a.at(0).empty()) {

                //pick a random student from unassigned students of the section
                int randomStudentIndex = rand() % a.at(0).size();
                std::tuple<std::string, int, int> temp = a.at(0).at(randomStudentIndex);

                //pick a random group from the free groups
                int randomGroupIndex = rand() % freeGroups.size();
                a.at(freeGroups.at(randomGroupIndex).first).push_back(temp);

                //remove the student from unassigned students
                a.at(0).erase(a.at(0).begin() + randomStudentIndex);
                //reduce the number of free spaces in the group
                freeGroups.at(randomGroupIndex).second--;

                //if the group is full, remove it from the free groups
                if(freeGroups.at(randomGroupIndex).second == 0) {
                    freeGroups.erase(freeGroups.begin() + randomGroupIndex);
                }
                //reduce the total number of free spaces
                freeSpaces--;
                //reduce the number of students who are not in any group yet
                x--;
            }
        }
            //If there are not enough free spaces to accommodate all students who are not in any group yet, then create new groups
        else{
            //If there are no free groups, then create new groups
            while(x > freeSpaces) {
                std::vector<std::tuple<std::string, int, int>> newGroup;
                //creating groups of size groupSize of random students from unassigned students
                for(int i = 0; i < groupSize; i++) {
                    if(!a.at(0).empty()) {
                        int tempIndex = rand() % a.at(0).size();
                        std::tuple<std::string, int, int> tempStudent = a.at(0).at(tempIndex);
                        a.at(0).erase(a.at(0).begin() + tempIndex);
                        newGroup.push_back(tempStudent);
                        x--;
                    }
                }
                //adding the new group to the section
                a.push_back(newGroup);
            }

            //Assigning the remaining students to the free spaces
            while(freeSpaces > 0 && !a.at(0).empty()){
                std::tuple<std::string,int, int> temp = a.at(0).back(); // Access the last element directly with back()
                int temp1 = rand() % freeGroups.size();
                a.at(freeGroups.at(temp1).first).push_back(temp);
                a.at(0).pop_back();
                freeGroups.at(temp1).second--;
                if(freeGroups.at(temp1).second == 0){
                    freeGroups.erase(freeGroups.begin() + temp1);
                }
                freeSpaces--;
                x--;
            }



        } //else bracket


    }


    return groups;

}

// Function to print the class
void printClass(const std::vector<std::vector<std::vector<std::tuple<std::string, int , int>>>> &objectGroups)
{
    // Calculate maximum name length for formatting
    int max_width = 0;
    for (const auto& section : objectGroups) {
        for (const auto& group : section) {
            for (const auto& student : group) {
                int name_length = std::get<0>(student).length();
                max_width = std::max(max_width, name_length);
            }
        }
    }

    // Print the classOfStudents in a tabulated format
    for (int section = 0; section < objectGroups.size(); section++) {
        std::cout << "Section " << section + 1 << std::endl;
        for (int group = 0; group < objectGroups[section].size(); group++) {
            std::cout << "Group " << group  << std::endl;
            for (const auto& student : objectGroups[section][group]) {
                std::cout << std::left << std::setw(max_width + 2) << std::get<0>(student);
            }
            std::cout << std::endl;
        }
        std::cout << std::endl;
    }
}


//Unit tests for the functions
// Test for the function `generateNormalDistributionRandomNumbers`
void testGenerateNormalDistributionRandomNumbers() {
    auto points = generateNormalDistributionRandomNumbers(20,2,20000,9);

    if (points.empty()) {
        std::cout << "Test for generateNormalDistributionRandomNumbers FAILED: Empty points." << std::endl;
    } else {
        std::cout << "Test for generateNormalDistributionRandomNumbers PASSED." << std::endl;
    }

    for (const auto& point : points) {
        if (point.second < 0) {
            std::cout << "Test for generateNormalDistributionRandomNumbers FAILED: Negative frequencies detected." << std::endl;
            return;
        }
    }
}

// Test for the function `generateUniformDistributionRandomNumbers
void testGenerateUniformDistributionRandomNumbers() {
    auto points = generateUniformDistributionRandomNumbers(2,10,20000,21);

    if (points.empty()) {
        std::cout << "Test for generateUniformDistributionRandomNumbers FAILED: Empty points." << std::endl;
    } else {
        std::cout << "Test for generateUniformDistributionRandomNumbers PASSED." << std::endl;
    }

    for (const auto& point : points) {
        if (point.second < 0) {
            std::cout << "Test for generateUniformDistributionRandomNumbers FAILED: Negative frequencies detected." << std::endl;
            return;
        }
    }
}


// Test for the function `classifyStudents` and `groupingStudents` belonging to the grouping algorithm
void testStudentGrouping() {
    // Sample data for testing
    std::vector<std::tuple<std::string,int,int>> testStudents = {
            {"John", 1, 0},
            {"Jane", 1, 0},
            {"Doe", 1, 0},
            {"Alan", 2, 0},
            {"Amy", 2, 0},
            {"David", 2, 0},
    };


    // Given a group size of 3 and 2 sections, we process the student assignment
    auto groupedStudents = classificationOfClass(testStudents, 3, 2);
    groupedStudents = groupingOfStudents(groupedStudents, 3);


    //Ensuring that no group exceeds its size limit.
    for (const auto &section : groupedStudents) {
        for (const auto &group : section) {
            if (group.size() > 3) {
                std::cout << "Test FAILED: Group size exceeded the limit." << std::endl;
                return;
            }
        }
    }

    //Making sure that students from different sections aren't mixed up in the same group.
    for (const auto &section : groupedStudents) {
        if(section[0].empty()) continue;  // Skip empty groups
        int sectionNum = std::get<1>(section[0][0]);
        for (const auto &group : section) {
            for (const auto &student : group) {
                if (std::get<1>(student) != sectionNum) {
                    std::cout << "Test FAILED: Students from different sections are in the same group." << std::endl;
                    return;
                }
            }
        }
    }

    std::cout << "Test PASSED: Students correctly grouped." << std::endl;
}
```