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

// Function declarations
std::vector <std::pair<double, int>> generateNormalDistributionRandomNumbers();
std::vector <std::pair<double, int>> generateUniformDistributionRandomNumbers();
std::vector<std::vector<std::vector<std::tuple<std::string, int , int>>>> classifyStudents(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>>>> groupingStudents(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);

// Unit test declarations
void testGenerateNormalDistributionRandomNumbers();
void testGenerateUniformDistributionRandomNumbers();
void testStudentGrouping();



int main(){
    // Initial menu display
    std::cout << "CSC 340 Small Group Project | Group 2" << std::endl;
    int keepRunning{true};
    int userInput{0};
    // Main application loop
    while(keepRunning) {
        // Displaying the user options menu
        std::cout<< "1) Generate a histogram of randomly generated floating-point numbers according to a normal distribution with a user-specified mean and standard deviation." <<std::endl;
        std::cout<< "2) Generate a histogram of randomly generated floating-point numbers according to a uniform distribution with a user-specified minimum and maximum." <<std::endl;
        std::cout<< "3) There are x learners who haven't found a small group yet. Please design and implement an algorithm to randomly assign them to an existing group or a newly created group if all existing groups are full. Constraints: (1) Each group will be limited to 3 members; and (2) students in the same group must be enrolled in the same section." <<std::endl;

        std::cout<< "0) Exit" <<std::endl;

        std::cout<< "Please enter a number: ";
        std::cin >> userInput;


        // Switch-case to handle user input
        switch (userInput) {

            case 1:{
                // Generating a histogram for a normal distribution
                std::cout << "You chose Q1" << std::endl;
                auto points = generateNormalDistributionRandomNumbers();
                makeHistogram(points);
                std::cout<<std::endl;
                std::cout<<std::endl;

                break;
            }
            case 2:{
                // Generating a histogram for a uniform distribution
                std::cout << "You chose 2" << std::endl;
                auto points = generateUniformDistributionRandomNumbers();
```

```
59                  makeHistogram(points);
60                  std::cout<<std::endl;
61                  std::cout<<std::endl;
62                  break;
63              }
64              case 3:{
65                  // Assigning students to groups
66                  std::cout << "You chose 3" << std::endl;
67
68                  // Sample student data with name, section, and group info
69                  // Note: A group number of '0' indicates no group assignment
70                  std::vector<std::tuple<std::string,int,int>> classOfStudents{
71                          {"Khalid", 1, 1},
72                          {"Jaylene", 1, 1},
73                          {"Diya", 1, 1},
74                          {"Hilary", 1, 2},
75                          {"Khaliesi", 1, 2},
76                          {"Dat", 1, 3},
77                          {"Sam", 1, 4},
78                          {"Elena", 1, 4},
79                          {"Miguel", 1, 4},
80                          {"Leo", 1, 5},
81                          {"Zack", 1, 5},
82                          {"Rene", 2, 1},
83                          {"Emma", 2, 1},
84                          {"Oliver", 2, 1},
85                          {"Liam", 2, 2},
86                          {"Ava", 2, 2},
87                          {"Benjamin", 2, 3},
88                          {"Charlotte", 2, 4},
89                          {"Amelia", 2, 4},
90                          {"Elijah", 2, 5},
91                          {"Harper", 2, 5},
92                          {"James", 2, 5},
93                          {"Sara", 1, 2},
94                          {"Lucas", 1, 0},
95                          {"Marie", 1, 0},
96                          {"Mia", 1, 0},
97                          {"Sophia", 2, 3},
98                          {"Noah", 2, 0},
99                          {"Isabella", 2, 0},
100                         {"William", 2, 0},
101                         {"Alice",1,0},
102                         {"Bob",1,0},
103                         {"Charlie",1,0},
104                         {"Daisy",1,0},
105                         {"Edward",1,0},
106                         {"Fiona",2,0},
107                         {"George",2,0},
108                         {"Hannah",2,0}
109                 };
110
111                 // Calculate maximum name length for formatting
112                 int max_width = 0;
113                 for (const auto& student : classOfStudents) {
114                     int nameLength = std::get<0>(student).length();
115                     max_width = std::max(max_width, nameLength);
116                 }
117
118
119                 std::cout<< "Group number 0 means student is not in any group yet." << std::endl;
```

```
120                 std::cout<< std::left << std::setw(max_width + 2) << "Student" << "  " << "Sec" << "  " << "Group" << std::endl;
121
122                 // Print the classOfStudents in a tabulated format
123                 for (const auto& student : classOfStudents) {
124                     std::cout << std::left << std::setw(max_width + 2) << std::get<0>(student) << "  "
125                               << std::get<1>(student) << "      "
126                               << std::get<2>(student) << std::endl;
127                 }
128
129                 // Take user input for number of sections and group members
130                 std::cout<<"Enter number of Sections: "<<std::endl;
131                 int numSections;
132                 std::cin>>numSections;
133                 std::cout<<"Enter number of members per group: "<<std::endl;
134                 int numMembers;
135                 std::cin>>numMembers;
136
137                 // Classify students into sections and groups
138                 auto objectGroups = classifyStudents(classOfStudents, numMembers, numSections);
139                 std::cout<<"Initial grouping of students: "<<std::endl;
140                 printClass(objectGroups);
141
142                 // Further group students
143                 objectGroups = groupingStudents(objectGroups, numMembers);
144                 std::cout<<"Final grouping of students: "<<std::endl;
145                 printClass(objectGroups);
146
147                 break;
148             }
149             case 0:{
150                 // Exit the program
151                 keepRunning = false;
152                 std::cout << "Exiting..." << std::endl;
153                 break;
154             }
155
156             default:{
157                 // Invalid input
158                 keepRunning = false;
159                 userInput = 0;
160                 break;
161             }
162         }
163
164     }
165
166     testGenerateNormalDistributionRandomNumbers();
167     testGenerateUniformDistributionRandomNumbers();
168     testStudentGrouping();
169
170     return 0;
171
172
173 }
174
175 // Function to round a number to two decimal places
176 double roundToTwo(double num)
177 {
178     return round(num*10)/10;
179 }
180
```

```cpp

// Function to generate random numbers following a normal distribution.
// This function prompts the user for necessary parameters like mean, standard deviation,
// number of samples, and number of bins. It then generates random numbers based on the
// given parameters, categorizing them into specified bins.
std::vector <std::pair<double, int>> generateNormalDistributionRandomNumbers()
{
    std::vector <std::pair<double, int>> points;
    std::cout<< "Enter Mean" << std::endl;
    double mean;
    std::cin >> mean;
    std::cout<< "Enter Standard Deviation (Standard deviation should be greater than or equal to 1)" << std::endl;
    double standardDeviation;
    std::cin >> standardDeviation;
    std::cout<< "Enter Number of Samples" << std::endl;
    int numSamples;
    std::cin >> numSamples;
    std::cout<<"Enter Number of Bins" << std::endl;
    int numBins;
    std::cin >> numBins;


    //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++)
    {
        points.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(points.begin(), points.end(), [currRandom](std::pair<double,int> a){return a.first == currRandom;});
        p->second++;
    }

    //Returning the vector of bins
    return points;

}

```

```cpp
// Function to generate random numbers following a uniform distribution.
// This function prompts the user for necessary parameters like minimum, maximum,
// number of samples, and number of bins. It then generates random numbers based on the
// given parameters, categorizing them into specified bins.

std::vector <std::pair<double, int>> generateUniformDistributionRandomNumbers()
{
    std::cout<<"Enter range of numbers: "<<std::endl;
    double  a, b;
    std::cin>>a>>b;
    std::cout<< "Enter Number of Samples" << std::endl;
    int samples;
    std::cin >> samples;
    std::cout<<"Enter Number of Bins" << std::endl;
    int bins;
    std::cin >> bins;

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

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

    for(int i = 0; i < bins; i++)
    {
        points.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 < samples; ++n) {

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

    //Returning the vector of bins
    return points;

}

// Function to generate a histogram from a vector of bins
// This function takes a vector of bins as input and prints a histogram
// based on the number of samples in each bin, the functions scales the histogram to fit the screen
// and print the histogram upright

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;
           }
       }

```

```
303
304         int unitLength = maxBarLength/8;
305
306           for(auto &a : points)
307           {
308                a.second = a.second/unitLength;
309           }
310
311           //Printing the histogram
312
313       for(int i = 8; i >=0; i--)
314       {
315           for(auto a : points)
316           {
317               if(a.second > i)
318               {
319                   std::cout << std::setw(8) << "*" ;
320               }
321               else
322               {
323                   std::cout <<std::setw(8)<< " ";
324               }
325           }
326           std::cout << std::endl;
327       }
328
329       //Printing the x-axis
330
331       for(auto a : points)
332       {
333           std::cout << std::setw(8) << a.first;
334       }
335
336
337   }
338
339   // Function to classify students into sections and groups
340   std::vector<std::vector<std::vector<std::tuple<std::string, int , int>>>> classifyStudents(const std::vector<std::tuple<std::string,int,int>> &students, int groupSize = 3, int
      numSections = 2){
341
342       std::vector<std::vector<std::vector<std::tuple<std::string, int , int>>>> tempGroups(numSections);
343       for(auto &a : tempGroups){
344           a.resize(10);
345       }
346       //Checking each student and assigning them to a section and group
347       for(auto a : students){
348           int temp1 = std::get<1>(a);
349           int temp2 = std::get<2>(a);
350           tempGroups[temp1-1][temp2].push_back(a);
351       }
352
353       //Removing empty groups
354       for(auto &a : tempGroups){
355           while(a.at(a.size()-1).size() ==0){
356               a.pop_back();
357           }
358       }
359
360       //returning the vector of groups
361       return tempGroups;
362
```

```
}


// 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>>>> groupingStudents(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
```

```
423                 for(int i = 0; i < groupSize; i++) {
424                     if(!a.at(0).empty()) {
425                         int tempIndex = rand() % a.at(0).size();
426                         std::tuple<std::string, int, int> tempStudent = a.at(0).at(tempIndex);
427                         a.at(0).erase(a.at(0).begin() + tempIndex);
428                         newGroup.push_back(tempStudent);
429                         x--;
430                     }
431                 }
432                 //adding the new group to the section
433                 a.push_back(newGroup);
434             }
435 
436             //Assigning the remaining students to the free spaces
437             while(freeSpaces > 0 && !a.at(0).empty()){
438                 std::tuple<std::string,int, int> temp = a.at(0).back(); // Access the last element directly with back()
439                 int temp1 = rand() % freeGroups.size();
440                 a.at(freeGroups.at(temp1).first).push_back(temp);
441                 a.at(0).pop_back();
442                 freeGroups.at(temp1).second--;
443                 if(freeGroups.at(temp1).second == 0){
444                     freeGroups.erase(freeGroups.begin() + temp1);
445                 }
446                 freeSpaces--;
447                 x--;
448             }
449 
450 
451 
452         } //else bracket
453 
454 
455     }
456 
457 
458     return groups;
459 
460 }
461 
462 // Function to print the class
463 void printClass(const std::vector<std::vector<std::vector<std::tuple<std::string, int , int>>>> &objectGroups)
464 {
465     // Calculate maximum name length for formatting
466     int max_width = 0;
467     for (const auto& section : objectGroups) {
468         for (const auto& group : section) {
469             for (const auto& student : group) {
470                 int name_length = std::get<0>(student).length();
471                 max_width = std::max(max_width, name_length);
472             }
473         }
474     }
475 
476     // Print the classOfStudents in a tabulated format
477     for (int section = 0; section < objectGroups.size(); section++) {
478         std::cout << "Section " << section + 1 << std::endl;
479         for (int group = 0; group < objectGroups[section].size(); group++) {
480             std::cout << "Group " << group  << std::endl;
481             for (const auto& student : objectGroups[section][group]) {
482                 std::cout << std::left << std::setw(max_width + 2) << std::get<0>(student);
483             }
```

```
            std::cout << std::endl;
        }
        std::cout << std::endl;
    }
}


//Unit tests for the functions
// Test for the function `generateNormalDistributionRandomNumbers`
void testGenerateNormalDistributionRandomNumbers() {
    auto points = generateNormalDistributionRandomNumbers();

    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();

    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 = classifyStudents(testStudents, 3, 2);
    groupedStudents = groupingStudents(groupedStudents, 3);

```

```
545
546     //Ensuring that no group exceeds its size limit.
547     for (const auto &section : groupedStudents) {
548         for (const auto &group : section) {
549             if (group.size() > 3) {
550                 std::cout << "Test FAILED: Group size exceeded the limit." << std::endl;
551                 return;
552             }
553         }
554     }
555
556     //Making sure that students from different sections aren't mixed up in the same group.
557     for (const auto &section : groupedStudents) {
558         int sectionNum = std::get<1>(section[0][0]); // assuming at least one student in a group
559         for (const auto &group : section) {
560             for (const auto &student : group) {
561                 if (std::get<1>(student) != sectionNum) {
562                     std::cout << "Test FAILED: Students from different sections are in the same group." << std::endl;
563                     return;
564                 }
565             }
566         }
567     }
568
569     std::cout << "Test PASSED: Students correctly grouped." << std::endl;
570 }
571
572
```