

```

1 #include <iostream>
2 #include <random>
3 #include <algorithm>
4 #include <iomanip>
5 #include <vector>
6 #include <utility>
7
8 //Function Prototypes
9 std::vector<std::pair<double, int>> generateNormalDistributionRandomNumbers(double, double, int, int);
10 std::vector<std::pair<double, int>> generateUniformDistributionRandomNumbers(double, double, int, int);
11
12 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);
13 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);
14 void printClass(const std::vector<std::vector<std::vector<std::tuple<std::string, int, int>>> &objectGroups);
15 double roundToTwo(double num);
16 void makeHistogram(std::vector<std::pair<double, int>> &points);
17
18 void testGenerateNormalDistributionRandomNumbers();
19 void testGenerateUniformDistributionRandomNumbers();
20 void testStudentGrouping();
21
22 int main(){
23
24
25
26     std::vector<std::pair<double, int>> pointsNormalDistribution{};
27     std::vector<std::pair<double, int>> pointsUniformDistribution{};
28     std::vector<std::vector<std::vector<std::tuple<std::string, int, int>>> groupings{};
29
30
31
32     std::cout<<"Enter the mean of the normal distribution: ";
33     double mean{0};
34     std::cin>>mean;
35     std::cout<<"Enter the standard deviation of the normal distribution: ";
36     double standardDeviation{0};
37     std::cin>>standardDeviation;
38     std::cout<<"Enter the number of samples for Normal distribution: ";
39     int numSamplesND{0};
40     std::cin>>numSamplesND;
41     std::cout<<"Enter the number of bins for Normal distribution: ";
42     int numBinsND{0};
43     std::cin>>numBinsND;
44
45     //    Generating Normal Distribution Random Numbers
46     pointsNormalDistribution = generateNormalDistributionRandomNumbers(mean, standardDeviation, numSamplesND, numBinsND);
47     makeHistogram(pointsNormalDistribution);
48
49     std::vector<std::tuple<std::string, int, int>> classOfStudents{
50         {"Khalid", 1, 1},
51         {"Jaylene", 1, 1},
52         {"Diya", 1, 1},
53         {"Hilary", 1, 2},
54         {"Khaliesi", 1, 2},
55         {"Dat", 1, 3},
56         {"Sam", 1, 4},
57         {"Elena", 1, 4},
58         {"Miguel", 1, 4},
59         {"Leo", 1, 5},

```

```

60         {"Zack", 1, 5},
61         {"Rene", 2, 1},
62         {"Emma", 2, 1},
63         {"Oliver", 2, 1},
64         {"Liam", 2, 2},
65         {"Ava", 2, 2},
66         {"Benjamin", 2, 3},
67         {"Charlotte", 2, 4},
68         {"Amelia", 2, 4},
69         {"Elijah", 2, 5},
70         {"Harper", 2, 5},
71         {"James", 2, 5},
72         {"Sara", 1, 2},
73         {"Lucas", 1, 0},
74         {"Marie", 1, 0},
75         {"Mia", 1, 0},
76         {"Sophia", 2, 3},
77         {"Noah", 2, 0},
78         {"Isabella", 2, 0},
79         {"William", 2, 0},
80         {"Alice", 1, 0},
81         {"Bob", 1, 0},
82         {"Charlie", 1, 0},
83         {"Daisy", 1, 0},
84         {"Edward", 1, 0},
85         {"Fiona", 2, 0},
86         {"George", 2, 0},
87         {"Hannah", 2, 0}
88     };
89
90
91     std::cout<<std::endl<<std::endl;
92
93     //Generating Uniform Distribution Random Numbers
94     std::cout<<"Enter the lower bound of the uniform distribution: ";
95     double a{0};
96     std::cin>>a;
97     std::cout<<"Enter the upper bound of the uniform distribution: ";
98     double b{0};
99     std::cin>>b;
100    std::cout<<"Enter the number of samples for Uniform distribution: ";
101    int numSamplesUD{0};
102    std::cin>>numSamplesUD;
103    std::cout<<"Enter the number of bins for Uniform distribution: ";
104    int numBinsUD{0};
105    std::cin>>numBinsUD;
106
107    pointsUniformDistribution = generateUniformDistributionRandomNumbers(a, b, numSamplesUD, numBinsUD);
108    makeHistogram(pointsUniformDistribution);
109
110
111    std::cout<<"Enter number of Sections: "<<std::endl;
112    int numSections;
113    std::cin>>numSections;
114    std::cout<<"Enter number of members per group: "<<std::endl;
115    int numMembers;
116    std::cin>>numMembers;
117
118    //Classifying students into sections and groups
119    groupings = classificationOfClass(classOfStudents, numMembers, numSections);
120

```

```

121
122     std::cout<<"Students in Group 0 are not assigned to any group yet"<<std::endl<<std::endl;
123     std::cout<<"Initial grouping of students: "<<std::endl<<std::endl;
124     std::cout<<"-----"<<std::endl<<std::endl;
125     printClass(groupings);
126     //Further grouping students
127
128     groupings = groupingOfStudents(groupings, numMembers);
129
130
131
132     std::cout<<"Final grouping of students: "<<std::endl<<std::endl;
133     std::cout<<"-----"<<std::endl<<std::endl;
134
135
136     printClass(groupings);
137
138     testGenerateNormalDistributionRandomNumbers();
139     testGenerateUniformDistributionRandomNumbers();
140     testStudentGrouping();
141
142
143
144
145
146     return 0;
147 }
148 }
149
150
151 // Function to round a number to two decimal places
152 double roundToTwo(double num)
153 {
154     return round(num*10)/10;
155 }
156
157 //Function for getting frequency Points for Normal Distribution
158 std::vector <std::pair<double, int>> generateNormalDistributionRandomNumbers(double mean, double standardDeviation, int numSamples, int numBins)
159 {
160     std::vector <std::pair<double, int>> tempPoints;
161
162
163
164     //Preparing the vector for bins according to user specified bins, mean and standard deviation
165
166     int startingPoint{0};
167     int endingPoint{0};
168     if(numBins%2 == 0)
169     {
170         startingPoint = -1*(numBins/2) + 1;
171         endingPoint = startingPoint + numBins-1;
172     }
173     else
174     {
175         startingPoint = -1*((numBins-1)/2);
176         endingPoint = startingPoint + numBins -1;
177     }
178
179     for(int i = startingPoint; i<=endingPoint;i++)
180     {
181         tempPoints.push_back(std::make_pair(round(mean+(i*standardDeviation)),0));

```

```

182     }
183
184     //Generating random numbers and categorizing them into bins
185
186     std::random_device rd{};
187     std::mt19937 gen{rd()};
188
189     std::normal_distribution d{mean, standardDeviation};
190     int currRandom{0};
191     for(int n = 0; n < numSamples; ++n) {
192
193         currRandom = round(d(gen));
194         auto p = std::find_if(tempPoints.begin(), tempPoints.end(), [currRandom](std::pair<double, int> a){return a.first == currRandom;});
195         p->second++;
196     }
197
198     //Returning the vector of bins
199     return tempPoints;
200 }
201 }
202
203 std::vector<std::pair<double, int>> generateUniformDistributionRandomNumbers(double a, double b, int numSamplesUD, int numBinsUD)
204 {
205
206
207     //Preparing the vector for bins according to user specified bins and range
208     std::vector<std::pair<double, int>> tempPoints;
209
210     double binCenter = (b-a)*1.0/(numBinsUD-1);
211
212     for(int i = 0; i < numBinsUD; i++)
213     {
214         tempPoints.push_back(std::make_pair(roundToTwo(a+ i*binCenter) , 0));
215     }
216
217     //Generating random numbers and categorizing them into bins
218     std::random_device rd; // Will be used to obtain a seed for the random number engine
219     std::mt19937 gen(rd()); // Standard mersenne_twister_engine seeded with rd()
220     std::uniform_real_distribution<> dis(a, b);
221     double currRandom{0};
222     for (int n = 0; n < numSamplesUD; ++n) {
223
224         currRandom = roundToTwo(dis(gen));
225         auto p = std::find_if(tempPoints.begin(), tempPoints.end(), [currRandom](std::pair<double, int> a) { return a.first == currRandom; });
226         p->second++;
227     }
228
229     //Returning the vector of bins
230     return tempPoints;
231 }
232 }
233
234 void makeHistogram(std::vector<std::pair<double, int>> &points)
235 {
236     //Scaling the histogram to fit the screen
237     int maxBarLength{0};
238     for(auto a : points)
239     {
240         if(a.second > maxBarLength)
241         {
242             maxBarLength = a.second;

```

```

243     }
244 }
245
246
247 int unitLength = maxBarLength/8;
248
249 for(auto &a : points)
250 {
251     a.second = a.second/unitLength;
252 }
253
254 //Printing the histogram
255
256 for(int i = 8; i >=0; i--)
257 {
258     for(auto a : points)
259     {
260         if(a.second > i)
261         {
262             std::cout << std::setw(8) << "*" ;
263         }
264         else
265         {
266             std::cout <<std::setw(8)<< " ";
267         }
268     }
269     std::cout << std::endl;
270 }
271
272 //Printing the x-axis
273
274 for(auto a : points)
275 {
276     std::cout << std::setw(8) << a.first;
277 }
278
279
280 }
281
282 // Function to classify students into sections and groups
283 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){
284
285     std::vector<std::vector<std::vector<std::tuple<std::string, int , int>>>> tempGroups(numSections);
286     for(auto &a : tempGroups){
287         a.resize(10);
288     }
289     //Checking each student and assigning them to a section and group
290     for(auto a : students){
291         int temp1 = std::get<1>(a);
292         int temp2 = std::get<2>(a);
293         tempGroups[temp1-1][temp2].push_back(a);
294     }
295
296     //Removing empty groups
297     for(auto &a : tempGroups){
298         while(a.at(a.size()-1).size() ==0){
299             a.pop_back();
300         }
301     }
302

```



```

303     //returning the vector of groups
304     return tempGroups;
305
306 }
307
308 // Function to further group students
309 // This function takes a vector of groups as input and further groups students
310 // based on the number of free spaces in each group
311
312 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)
313 {
314
315     //Iterating through each section
316
317     for(auto &a : groups){
318
319
320         int x = a.at(0).size();                //Number of students who are not in any group yet in the current section
321         std::vector<std::pair<int,int>> freeGroups;
322         int freeSpaces{0};
323
324         for(int i = 1; i < a.size(); i++){
325             if(groupSize - a.at(i).size() > 0){
326                 freeGroups.push_back({i, groupSize - a.at(i).size()}); //Pushing the index of the group and the number of free spaces in that group
327                 freeSpaces += groupSize - a.at(i).size();                //Calculating the total number of free spaces
328             }
329         }
330
331         //If there are enough free spaces to accommodate all students who are not in any group yet, then assign them to the free spaces
332         if(freeSpaces >= x) {
333             std::cout << "There are enough free spaces to accommodate all students" << std::endl;
334             while(freeSpaces > 0 && !a.at(0).empty()) {
335
336                 //pick a random student from unassigned students of the section
337                 int randomStudentIndex = rand() % a.at(0).size();
338                 std::tuple<std::string, int, int> temp = a.at(0).at(randomStudentIndex);
339
340                 //pick a random group from the free groups
341                 int randomGroupIndex = rand() % freeGroups.size();
342                 a.at(freeGroups.at(randomGroupIndex).first).push_back(temp);
343
344                 //remove the student from unassigned students
345                 a.at(0).erase(a.at(0).begin() + randomStudentIndex);
346                 //reduce the number of free spaces in the group
347                 freeGroups.at(randomGroupIndex).second--;
348
349                 //if the group is full, remove it from the free groups
350                 if(freeGroups.at(randomGroupIndex).second == 0) {
351                     freeGroups.erase(freeGroups.begin() + randomGroupIndex);
352                 }
353                 //reduce the total number of free spaces
354                 freeSpaces--;
355                 //reduce the number of students who are not in any group yet
356                 x--;
357             }
358         }
359         //If there are not enough free spaces to accommodate all students who are not in any group yet, then create new groups
360         else{
361             //If there are no free groups, then create new groups
362             while(x > freeSpaces) {

```

```

363         std::vector<std::tuple<std::string, int, int>> newGroup;
364         //creating groups of size groupSize of random students from unassigned students
365         for(int i = 0; i < groupSize; i++) {
366             if(!a.at(0).empty()) {
367                 int tempIndex = rand() % a.at(0).size();
368                 std::tuple<std::string, int, int> tempStudent = a.at(0).at(tempIndex);
369                 a.at(0).erase(a.at(0).begin() + tempIndex);
370                 newGroup.push_back(tempStudent);
371                 x--;
372             }
373         }
374         //adding the new group to the section
375         a.push_back(newGroup);
376     }
377
378     //Assigning the remaining students to the free spaces
379     while(freeSpaces > 0 && !a.at(0).empty()){
380         std::tuple<std::string,int, int> temp = a.at(0).back(); // Access the last element directly with back()
381         int temp1 = rand() % freeGroups.size();
382         a.at(freeGroups.at(temp1).first).push_back(temp);
383         a.at(0).pop_back();
384         freeGroups.at(temp1).second--;
385         if(freeGroups.at(temp1).second == 0){
386             freeGroups.erase(freeGroups.begin() + temp1);
387         }
388         freeSpaces--;
389         x--;
390     }
391
392
393
394     } //else bracket
395
396
397 }
398
399
400 return groups;
401
402 }
403
404 // Function to print the class
405 void printClass(const std::vector<std::vector<std::vector<std::tuple<std::string, int , int>>>> &objectGroups)
406 {
407     // Calculate maximum name length for formatting
408     int max_width = 0;
409     for (const auto& section : objectGroups) {
410         for (const auto& group : section) {
411             for (const auto& student : group) {
412                 int name_length = std::get<0>(student).length();
413                 max_width = std::max(max_width, name_length);
414             }
415         }
416     }
417
418     // Print the classOfStudents in a tabulated format
419     for (int section = 0; section < objectGroups.size(); section++) {
420         std::cout << "Section " << section + 1 << std::endl;
421         for (int group = 0; group < objectGroups[section].size(); group++) {
422             std::cout << "Group " << group << std::endl;
423             for (const auto& student : objectGroups[section][group]) {

```

```

424         std::cout << std::left << std::setw(max_width + 2) << std::get<0>(student);
425     }
426     std::cout << std::endl;
427 }
428     std::cout << std::endl;
429 }
430 }
431
432
433 //Unit tests for the functions
434 // Test for the function `generateNormalDistributionRandomNumbers`
435 void testGenerateNormalDistributionRandomNumbers() {
436     auto points = generateNormalDistributionRandomNumbers(20,2,20000,9);
437
438     if (points.empty()) {
439         std::cout << "Test for generateNormalDistributionRandomNumbers FAILED: Empty points." << std::endl;
440     } else {
441         std::cout << "Test for generateNormalDistributionRandomNumbers PASSED." << std::endl;
442     }
443
444     for (const auto& point : points) {
445         if (point.second < 0) {
446             std::cout << "Test for generateNormalDistributionRandomNumbers FAILED: Negative frequencies detected." << std::endl;
447             return;
448         }
449     }
450 }
451
452 // Test for the function `generateUniformDistributionRandomNumbers`
453 void testGenerateUniformDistributionRandomNumbers() {
454     auto points = generateUniformDistributionRandomNumbers(2,10,20000,21);
455
456     if (points.empty()) {
457         std::cout << "Test for generateUniformDistributionRandomNumbers FAILED: Empty points." << std::endl;
458     } else {
459         std::cout << "Test for generateUniformDistributionRandomNumbers PASSED." << std::endl;
460     }
461
462     for (const auto& point : points) {
463         if (point.second < 0) {
464             std::cout << "Test for generateUniformDistributionRandomNumbers FAILED: Negative frequencies detected." << std::endl;
465             return;
466         }
467     }
468 }
469
470
471 // Test for the function `classifyStudents` and `groupingStudents` belonging to the grouping algorithm
472 void testStudentGrouping() {
473     // Sample data for testing
474     std::vector<std::tuple<std::string,int,int>> testStudents = {
475         {"John", 1, 0},
476         {"Jane", 1, 0},
477         {"Doe", 1, 0},
478         {"Alan", 2, 0},
479         {"Amy", 2, 0},
480         {"David", 2, 0},
481     };
482
483
484     // Given a group size of 3 and 2 sections, we process the student assignment

```



```
485     auto groupedStudents = classificationOfClass(testStudents, 3, 2);
486     groupedStudents = groupingOfStudents(groupedStudents, 3);
487
488     //Ensuring that no group exceeds its size limit.
489     for (const auto &section : groupedStudents) {
490         for (const auto &group : section) {
491             if (group.size() > 3) {
492                 std::cout << "Test FAILED: Group size exceeded the limit." << std::endl;
493                 return;
494             }
495         }
496     }
497 }
498
499 //Making sure that students from different sections aren't mixed up in the same group.
500 for (const auto &section : groupedStudents) {
501     if(section[0].empty()) continue; // Skip empty groups
502     int sectionNum = std::get<1>(section[0][0]);
503     for (const auto &group : section) {
504         for (const auto &student : group) {
505             if (std::get<1>(student) != sectionNum) {
506                 std::cout << "Test FAILED: Students from different sections are in the same group." << std::endl;
507                 return;
508             }
509         }
510     }
511 }
512
513 std::cout << "Test PASSED: Students correctly grouped." << std::endl;
514 }
```