```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <queue>
#include <vector>
#include <cmath>
#include <utility>
#include <algorithm>
#include <unordered_map>

using std::cout;
using std::cerr;

typedef std::vector<std::vector<int>> OrientedGraph; // It is a list of graph
nodes, each with it's own list of nodes it has edge to

void printProgress(const int percentage) {
	std::cout << "\r" << percentage << "%";
}

class Map {
protected:
	int *data;
	int width;
	int height;
	int offset;

	int countHotspots();
	void drawBorder();
	void drawWall(const int position, const int direction1, const int
direction2);
	int findHotspotPosition(const int hotspotID);
	void floodFill(const int position, const int value);
	void generateDoors(const bool onlyBareMinimum);
	void generateHotspots();
	void generateWalls(const int pillarCount);
	int markRooms();
	int moveInDirection(const int position, const int direction);
	void reduceGraphToSkelet(OrientedGraph &graph);
	void removeMarkers();
	void sanitizeHotspots();

public:
	void print();
	void generate(const int pillarCount, const bool minimumDoors);
	bool init(const int width, const int height, const int offset);
	void deinit();
	Map(){ data = NULL; width = 0; height = 0; }
	~Map() { deinit(); }
};

int main(int argc, char *argv[]) {
	srand(unsigned(time(NULL)));

	Map map;
	if (not map.init(45, 21, 3)) return 1;

	map.generate(4, false);
	map.print();

	return 0;
}
```

```
int Map::countHotspots() {
	int cnt = 0;

	for (int y = offset + 1; y < height - 1; y += (offset + 1)) {
		for (int x = offset + 1; x < width - 1; x += (offset + 1)) {
			if (data[y * width + x] == 2) cnt++;
		}
	}

	return cnt;
}

void Map::drawBorder() {
	for (int y = 0; y < height; y++) {
		data[y * width] = 1;
		data[y * width + width - 1] = 1;
	}

	for (int x = 0; x < width; x++) {
		data[x] = 1;
		data[(height - 1) * width + x] = 1;
	}
}

void Map::drawWall(const int position, const int direction1, const int direction2)
{
	int positions[2] = {
		moveInDirection(position, direction1),
		moveInDirection(position, direction2)
	};
	int directions[2] = { direction1, direction2 };

	data[position] = 1;
	for (int i = 0; i < 2; i++) {
		while (data[positions[i]] != 1) {
			data[positions[i]] = 1;
			positions[i] = moveInDirection(positions[i], directions[i]);
		}
	}
}

int Map::findHotspotPosition(const int hotspotID) {
	int cnt = 0;

	for (int y = offset + 1; y < height - 1; y += (offset + 1)) {
		for (int x = offset + 1; x < width - 1; x += (offset + 1)) {
			if (data[y * width + x] == 2) {
				if (cnt == hotspotID) return y * width + x;
				else cnt++;
			}
		}
	}

	return 0;
}
```

```
void Map::floodFill(const int position, const int value) {
	std::queue<int> positions;
	positions.push(position);

	int oldValue = data[position];

	while (!positions.empty()) {
		int pos = positions.front();
		positions.pop();

		data[pos] = value;

		// Look at neighbours at put them into queue
		if (data[pos - width] == oldValue) positions.push(pos - width);
		if (data[pos - 1] == oldValue) positions.push(pos - 1);
		if (data[pos + width] == oldValue) positions.push(pos + width);
		if (data[pos + 1] == oldValue) positions.push(pos + 1);
	}
}

void Map::generateDoors(const bool onlyBareMinimum) {
	printProgress(20);

	const int tests[2] = {1, width};
	const int roomCount = markRooms();

	OrientedGraph graph (roomCount);
	std::unordered_map<int, std::vector<int>> potentialDoors;
	std::vector<std::vector<bool>> checks(roomCount);
	for (int i = 0; i < roomCount; i++) {
		checks[i].resize(roomCount);
		for (int p = 0; p < roomCount; p++) {
			checks[i][p] = false;
		}
	}

	printProgress(40);

	// Build up graph and also list of data coordinates that are candidates for
doors between rooms
	for (int y = 1; y < height - 1; y++) {
		for (int x = 1; x < width - 1; x++) {
			int pos = y * width + x;
			if (data[pos] != 1) continue;

			for (int i = 0; i < 2; i++) {
				int t = tests[i];
				if (data[pos - t] < 0 && data[pos + t] < 0 && data[pos -
t] != data[pos + t]) {
					int fromIndex = std::max(data[pos - t], data[pos +
t]) * -1 - 1;
					int toIndex = std::min(data[pos - t], data[pos +
t]) * -1 - 1;

					if (not checks[fromIndex][toIndex]) {
						graph[fromIndex].push_back(toIndex);
						checks[fromIndex][toIndex] = true;
					}
					potentialDoors[fromIndex * roomCount +
toIndex].push_back(pos);
				}
			}
		}
	}
	checks.clear();

	printProgress(60);
```

```
	if (onlyBareMinimum) reduceGraphToSkelet(graph);

	printProgress(70);

	// Generate doors itselves
	for (unsigned n = 0; n < graph.size(); n++) {
		for (unsigned d = 0; d < graph[n].size(); d++) {
			unsigned range = potentialDoors[n * roomCount +
graph[n][d]].size();

			if (range == 0) continue;

			int index = rand() % range;
			int pos = potentialDoors[n * roomCount + graph[n][d]][index];
			data[pos] = 0;
		}
	}

	printProgress(90);

	removeMarkers();

	printProgress(100);
}

void Map::generateHotspots() {
	for (int y = 0; y < height; y += (offset + 1)) {
		for (int x = 0; x < width; x += (offset + 1)) {
			data[y * width + x] = 2;
		}
	}
}

void Map::generateWalls(const int pillarCount) {
	int hotspots = countHotspots();
	while (hotspots > pillarCount) {
		int randHotspotID = rand() % hotspots;
		int direction1 = rand() % 4;
		int direction2 = rand() % 4;
		int position = findHotspotPosition(randHotspotID);

		drawWall(position, direction1, direction2);
		hotspots = countHotspots();
	}
}

int Map::markRooms() {
	// Mark all rooms
	int cnt = -1;
	for (int y = 1; y < height; y += offset + 1) {
		for (int x = 1; x < width; x += offset + 1) {
			int pos = y * width + x;
			if (data[pos] == 0) {
				floodFill(pos, cnt);
				cnt--;
			}
		}
	}

	// Return number of marked rooms
	return cnt * -1 - 1;
}
```

```
int Map::moveInDirection(const int position, const int direction) {
	if (direction == 0) return position - width;
	else if (direction == 1) return position - 1;
	else if (direction == 2) return position + width;
	else if (direction == 3) return position + 1;
	return position;
}

void Map::reduceGraphToSkelet(OrientedGraph &graph) {
	std::vector<bool> visited;
	visited.resize(graph.size());
	for (unsigned i = 0; i < visited.size(); i++) visited[i] = false;

	std::queue<int> toProcess;
	toProcess.push(0);
	visited[0] = true;

	while(not toProcess.empty()) {
		int node = toProcess.front();
		toProcess.pop();

		for (unsigned i = 0; i < graph[node].size(); i++) {
			if (not visited[graph[node][i]]) {
				toProcess.push(graph[node][i]);
				visited[graph[node][i]] = true;
			}
			else {
				graph[node].erase(graph[node].begin() + i);
				i--;
			}
		}
	}
}

void Map::removeMarkers() {
	for (int y = 1; y < height; y += offset + 1) {
		for (int x = 1; x < width; x += offset + 1) {
			int pos = y * width + x;
			if (data[pos] < 0) {
				floodFill(pos, 0);
			}
		}
	}
}

void Map::sanitizeHotspots() {
	for (int y = 0; y < height; y += (offset + 1)) {
		for (int x = 0; x < width; x += (offset + 1)) {
			if (data[y * width + x] == 2) {
				data[y * width + x] = 1;
			}
		}
	}
}

void Map::print() {
	for (int y = 0; y < height; y++) {
		for (int x = 0; x < width; x++) {
			int value = data[y * width + x];
			if (value == 0) cout << " ";
			else if (value == 1) cout << "#";
			else if (value == 2) cout << ".";
			else if (value < 0) cout << (value * -1);
			else cout << "?";
		}
		cout << "\n";
	}
}
```

```
// *** PUBLIC METHODS ***

void Map::generate(const int pillarCount, const bool minimumDoors) {
	generateWalls(pillarCount);
	sanitizeHotspots();
	print();
	generateDoors(minimumDoors);
	std::cout << "\n";
}

bool Map::init(const int width, const int height, const int offset) {
	if (((width - 1) % (offset + 1)) != 0) {
		cerr << "ERROR: Bad width\n";
		return false;
	}
	if (((height - 1) % (offset + 1)) != 0) {
		cerr << "ERROR: Bad height\n";
		return false;
	}

	Map::width = width;
	Map::height = height;
	Map::offset = offset;

	data = new int[width * height];
	if (data == NULL) return false;

	for (int i = 0; i < width * height; i++) data[i] = 0;

	generateHotspots();
	drawBorder();

	return true;
}

void Map::deinit() {
	if (data != NULL) {
		delete[] data;
		data = NULL;
	}
}
```