You will need a vector class which lets you handle vectors in 2 dimensions. I have chosen to use SFML here.

Lets first define 9 directions e_i,

```
sf::Vector2f e[9];
e[0] = sf::Vector2f(0,0);
e[1] = sf::Vector2f(0,1);
e[2] = sf::Vector2f(0,-1);
e[3] = sf::Vector2f(1,0);
e[4] = sf::Vector2f(-1,0);
e[5] = sf::Vector2f(1,1);
e[6] = sf::Vector2f(1,-1);
e[7] = sf::Vector2f(-1,-1);
e[8] = sf::Vector2f(-1,1);
```

and a set of weights w_i,

```
float w[9];
w[0] = 16/36.0;
w[1] = 4/36.0;
w[2] = 4/36.0;
w[3] = 4/36.0;
w[4] = 4/36.0;
w[5] = 1/36.0;
w[6] = 1/36.0;
w[7] = 1/36.0;
w[8] = 1/36.0;
```

that we will use later

I also assume we are working on a square grid, with size given by n.

```
const int n = 128;  // just an example
```

Now lets discuss the data structures that you will need. I chose to use

```
float* f[9];
float* ft[9];
```

So the value of f_i at gridposition (x, y) will be located at

```
f[i][x+y*n]
```

The array ft will be used to create a copy of f in the calculation.

At each position in the grid you will also need the density

$$\rho = \sum_i f_i$$

and the velocity

$$u = \frac{1}{\rho} \sum_i f_i \cdot e_i.$$

You could store these values in arrays if you wish, but it is perhaps easier to just implement two functions

```
sf::Vector2f u(int x, int y);
float rho(int x, int y);
```

which computes the values from the array f. You will need to divide by ρ in the function u, which means that you have to be careful not to divide by zero. I suggest you check the size of ρ, and if it falls below a predefined threshhold, you write a warning to the console, and then clamp it to the threshold.

The algorithm has four steps

Initialisation:

You need to initialise the values in the array f, so that the density is not zero anywhere. Also, you may want to put higher density in an area of the grid to test your algorithm.

Then repeat the following three steps:

Transport:

If you are not at a wall, or at the boundary of the grid, transport is done with

```
ft[k][(y+(int)e[k].y)*n+x+(int)e[k].x] = f[k][y*n+x];
```

for each k, x and y. Note that we store these values in the temporary array.

Transport at a wall or the boundary must be handled separately.

If you are located just to the left of a wall, then the fluid bounces off the wall using

```
ft[4][y*n+x] = f[3][y*n+x];
ft[7][y*n+x] = f[6][y*n+x];
ft[8][y*n+x] = f[5][y*n+x];
```

Transport is as usual for the other k.

Similar equations are needed when you are right of a wall and over/under a wall. I leave the handling of corners to you.

You should think carefully about how to handle walls, but I leave this up to you. At the very least you should try to avoid if-statements in inner loops (we have 3 nested loops here), as this will make your program slow. Ask if you want some of my ideas.

Collision step:

This is the step where the physics happen.

You need to compute an approximation to something called the "equlibrium distribution".

```
sf::Vector2f feq[9];
sf::Vector2f u = u(x,y)  // introduced to make the next line shorter
feq[k] = w[k]*rho(x,y)*(1+3*dot(e[0],u)+
            4.5*dot(e[0],u)*dot(e[0],u)-1.5*dot(u,u));
```

Interpolation:

Finally the end result is an interpolation between the transport and the collision step.

```
const float v = 0.5;  // Experiment with this number.
f[k][x+n*y] = v*ft[k][x+y*n]+(1-v)*feq[k];
```

If you implement the above with high density in an area and low densitity in the rest and run the program you should reach equilibrium after a while. If you do you can assume that your program works and continue.

Now:

1. In the transport/collision-loop, update an area on the very left with constant density.
2. Open the right wall.

After having run the computation for a while, you can use the values of u to move particles through your fluid. The update function of a particle could be

```
void update(float dt)
{
pos += u(pos.x,pos.y)*dt;
```

```
4

}
```