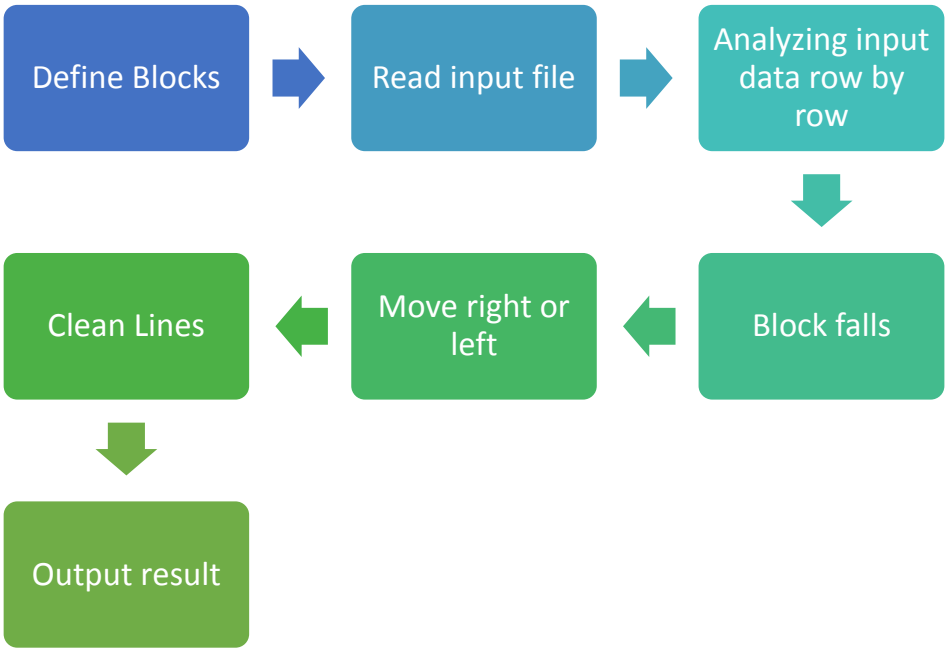
Project Description

Program Flow Chart:



Detailed Description:

1. Define Blocks:

```
1.  bool t1[4][4] = {{0, 0, 0, 0}, {0, 0, 0, 0}, {1, 1, 1, 0}, {0, 1, 0, 0}};
2.  bool t2[4][4] = {{0, 0, 0, 0}, {0, 1, 0, 0}, {1, 1, 0, 0}, {0, 1, 0, 0}};
3.  bool t3[4][4] = {{0, 0, 0, 0}, {0, 0, 0, 0}, {0, 1, 0, 0}, {1, 1, 1, 0}};
4.  bool t4[4][4] = {{0, 0, 0, 0}, {1, 0, 0, 0}, {1, 1, 0, 0}, {1, 0, 0, 0}};
5.
6.  bool l1[4][4] = {{0, 0, 0, 0}, {1, 0, 0, 0}, {1, 0, 0, 0}, {1, 1, 0, 0}};
7.  bool l2[4][4] = {{0, 0, 0, 0}, {0, 0, 0, 0}, {1, 1, 1, 0}, {1, 0, 0, 0}};
8.  bool l3[4][4] = {{0, 0, 0, 0}, {1, 1, 0, 0}, {0, 1, 0, 0}, {0, 1, 0, 0}};
9.  bool l4[4][4] = {{0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 1, 0}, {1, 1, 1, 0}};
10.
11. bool j1[4][4] = {{0, 0, 0, 0}, {0, 1, 0, 0}, {0, 1, 0, 0}, {1, 1, 0, 0}};
12. bool j2[4][4] = {{0, 0, 0, 0}, {0, 0, 0, 0}, {1, 0, 0, 0}, {1, 1, 1, 0}};
13. bool j3[4][4] = {{0, 0, 0, 0}, {1, 1, 0, 0}, {1, 0, 0, 0}, {1, 0, 0, 0}};
14. bool j4[4][4] = {{0, 0, 0, 0}, {0, 0, 0, 0}, {1, 1, 1, 0}, {0, 0, 1, 0}};
```

```
15.
16.  bool s1[4][4] = {{0, 0, 0, 0}, {0, 0, 0, 0}, {0, 1, 1, 0}, {1, 1, 0, 0}};
17.  bool s2[4][4] = {{0, 0, 0, 0}, {1, 0, 0, 0}, {1, 1, 0, 0}, {0, 1, 0, 0}};
18.
19.  bool z1[4][4] = {{0, 0, 0, 0}, {0, 0, 0, 0}, {1, 1, 0, 0}, {0, 1, 1, 0}};
20.  bool z2[4][4] = {{0, 0, 0, 0}, {0, 1, 0, 0}, {1, 1, 0, 0}, {1, 0, 0, 0}};
21.
22.  bool i1[4][4] = {{1, 0, 0, 0}, {1, 0, 0, 0}, {1, 0, 0, 0}, {1, 0, 0, 0}};
23.  bool i2[4][4] = {{0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}, {1, 1, 1, 1}};
24.
25.  bool o[4][4] = {{0, 0, 0, 0}, {0, 0, 0, 0}, {1, 1, 0, 0}, {1, 1, 0, 0}};
```

Because all blocks and the matrix are represented by 0 and 1. To reduce the memory I choose to use traditional array with the type of boolean. I think this could make the code more readable and effecient.

2. Read input file

```
1.  //open file
2.      ifstream Infile;
3.      Infile.open(argv[1], fstream::in);
4.      if(!Infile.is_open()) {
5.          cout<<"Error:Failed to open input file."<<endl;
6.          return 1;
7.      }
```

(Using the argv[1] to grab the file as TA mentioned)

3. Analyzing input data row by row

```
1.  while(Infile>>temp && strcmp(temp, "End") != 0){
2.          //init map
3.          if(row == 0 && col == 0){
4.              if(temp[1] >= 48 && temp[1] <= 58){
5.                  row = 10 * (temp[0] - 48) + temp[1] - 48;
6.              }
7.              else{
8.                  row = temp[0] - 48;
9.              }
```

```
            Infile>>col;
            puzzle = new bool *[row];
            for(int i = 0; i < row; i++){
                puzzle[i] = new bool [col];
            }
            for(int i = 0; i < row; i++){
                for(int j = 0; j < col; j++){
                    puzzle[i][j] = 0;
                }
            }
```

Read data from files, if row and column is the initial value 0, that means it is the first line. We need to build the Tetris matrix by the given row and column.
If the first string of the input file row is “End”, the program will directly jump out of the loop and prepare to write the output file. For the rest of the case, we need to place the blocks with given types, positions, and movements after the block hit the bottom.

```
//put the block
        else{
            Infile>>pos>>move;
            pos--;
            bool block[4][4];
            int rh = -1;
            int rm = 0;
            bool legal = true;
            //judge the block
            if(strcmp(temp, "T1") == 0){
                SetBlock(t1, block);
            }
            else if(strcmp(temp, "T2") == 0){
                SetBlock(t2, block);
            }
```

As we know we need to place the block, we start from getting the position and movement of the block. Moreover, we need to identify the block type for sure. (The SetBlock is just a simple function as

following instruction)

```
    void SetBlock(bool source[4][4], bool aim[4][4]);
```

4. Block falls

   As we get sufficient data about the block, we can start to place the block in right position.

```
int height = 0;
 while(legal && height < row){
    //continue to fall
    for(int i = 0; i < ((height >= 4) ? 4 : height + 1) ; i++){
        for(int j = 0; j < 4; j++){
            if(block[3 - i][j] == 1 &&
               puzzle[height - i][pos + j] == 1){
                legal = false;
            }
        }
    }
    if(legal){
        rh = height;
        height ++;
    }
}
```

We use rh to store the right height of the block. As we make sure the current position is valid, store the current right height. Subsequently, we test the next line. If the next line is invalid, we use the current height as right height, which means the block has hit the bottom.

5. Move right or left
   When the block has hit the bottom, we can deal with the final move of the block.

```
if(move > 0){
    for (int a = 1; a <= move; a++){
        for (int i = 0; i < 4; i++){
            for (int j = 0; j < 4; j++){
                //prevent hitting the wall
                if( !(block[3 - i][j] && puzzle[rh - i][pos + j + a])){
                    rm = a;
                }
            }
        }
    }
}
else if(move < 0){
    for (int a = -1; a >= move; a--){
        for (int i = 0; i < 4; i++){
            for (int j = 0; j < 4; j++){
                if( !(block[3 - i][j] && puzzle[rh - i][pos + j + a])){
                    rm = a;
                }
            }
        }
    }
}
else{
    rm = 0;
}
```

   The Process is similar to judge the right height of the block as mentioning above. After moving, the block may have the chance to fall again. Therefore, we use nearly the same code again as above.

```
//fall after move
legal = true;
pos += rm;
while(legal && height < row){
//continue to fall
    for (int i = 0; i < ((height >= 4) ? 4 : height + 1); i++)
    {
        for (int j = 0; j < 4; j++)
        {
            if (block[3 - i][j] == 1 &&
                puzzle[height - i][pos + j] == 1)
            {
                legal = false;
            }
        }
    }
    if(legal){
        rh = height;
        height ++;
    }
}
```

Now, we can finally put the block on the matrix.

```
//actually put the block
for(int a = 0; a < 4; a ++){
    for(int b = 0; b < 4; b++){
        if(block[3 - a][b]){
            puzzle[rh - a][pos + b] = 1;
        }
    }
}
```

6. Clean lines
   After putting the block correctly, we need to check if we need to clean a full line.

```
//check line0.
int line[row];
int num_line = 0;
for (int i = 0; i < row; i++)
    line[i] = -1;
//record remove lines
for (int i = 0; i < row; i++){
    for (int j = 0; j < col; j++){
        if(!puzzle[i][j]){
            break;
        }
            else if(j == col - 1 && puzzle[i][j]){
            line[num_line++] = i;
        }
    }
}
//blocks drop
for (int a = 0; line[a] != -1; a++){
    for (int i = line[a]; i >= 0; i--){
        for (int j = 0; j < col; j++){
            if(i - 1 < 0){
                puzzle[i][j] = 0;
            }
            else{
                puzzle[i][j] = puzzle[i - 1][j];
            }
        }
    }
}
```

I use an arrray to store lines we need to clean. Everytime we remove a line, all the blocks above should fall. By this, if a block cause multiple lines to be removed, it shouldn't have any mistake.

7. Output result
   After the while loop is finished, that means it is time to write the output file.

```cpp
//output file
fstream Outfile;
Outfile.open("108034017_proj1.final", fstream::out);
for (int i = 4; i < row; i++){
    for (int j = 0; j < col; j++){
        Outfile << puzzle[i][j] << " ";
    }
    Outfile << endl;
}
//close the file
Infile.close();
Outfile.close();
```

Test Case Design

108034017_proj1.data

```
10  3
L1  1   0
S2  1   0
I1  3   0
T4  1   1
J3  2  -1
L3  2   0
Z2  1   0
L2  1   0
O   1   0
I1  1   2
T1  1   0
T2  2   0
L4  1   0
J1  1   0
J4  1   0
I1  1   0
Z1  1   0
J2  1   0
T2  2   0
S1  1   0
T3  1   0
End
```

I just simply test all the blocks listed in the file that TA provides.