/\*

\* sudoku.h

\* Sudoku

\* Created by Prof. Ramavarapu Sreenivas

\* Inspired by: http://web.eecs.utk.edu/courses/spring2012/cs140/Notes/Sudoku/index.html

\*/

#ifndef sudoku

#define sudoku

#include <vector>

#include <fstream>

**using** std::vector;

**using** **namespace** std;

**class** Sudoku

{

// Private

**int** puzzle[9][9];

// Private member function that checks if the named row is valid

**bool** row\_valid(**int** row, **int** k)

{

// write code that checks if "row" is valid

**for** (**int** col=0; col<9; col++){

**if** (puzzle[row][col] == k)

**return** **false**;

}

**return** **true**;

}

// Private member function that checks if the named column is valid

**bool** col\_valid(**int** col, **int** k)

{

// check validity of "col"

**for**(**int** row=0; row<9;row++){

**if**(puzzle[row][col] == k)

**return** **false**;

}

**return** **true**;

}

// Private member function that checks if the named 3x3 block is valid

**bool** block\_valid(**int** row, **int** col, **int** k)

{

// check 3 x 3 block validity// for every k, find which block it belongs to, then start search every part of that block to check validity.

**int** StartRow = row - row % 3;

**int** StartCol = col - col % 3;

**for** (**int** i = 0; i < 3; i++)

**for** (**int** j = 0; j< 3; j++)

**if** (puzzle[StartRow+i][StartCol+j] == k)

**return** **false**;

**return** **true**;

}

**bool** find\_blank(**int**& row, **int**& col)

{

**for** ( row=0;row<9;row++)

**for**(col=0;col<9;col++)

**if** (puzzle[row][col] == 0)

**return** **true**;

**return** **false**;

}

**public**:

// Public member function that reads the incomplete puzzle

// we are not doing any checks on the input puzzle -- that is,

// we are assuming they are indeed valid

**void** read\_puzzle(**int** argc, **char** \* **const** argv[])

{

// write code that reads the input puzzle using the

// guidelines of figure 23 of the bootcamp material

// The input file name is "input1"

// It contains the numbers P\_1 P\_2 ... P\_M

ifstream input\_file(argv[1]);

**if** (input\_file.is\_open()) {// If "input1" exists in the local directory

// As long as your are not at the "end-of-file"

**for** (**int** i=0;i<9;i++)

**for** (**int** j=0;j<9;j++)

input\_file>>puzzle[i][j];

input\_file.close();

}

**else**

cout << "Input file does not exist in PWD" << endl;

}

// Public member function that prints the puzzle when called

**void** print\_puzzle()

{

std::cout << std::endl << "Board Position" << std::endl;

**for** (**int** i = 0; i < 9; i++)

{

**for** (**int** j = 0; j < 9; j++)

{

// check if we have a legitimate integer between 1 and 9

**if** ((puzzle[i][j] >= 1) && (puzzle[i][j] <= 9))

{

// printing initial value of the puzzle with some formatting

std::cout << puzzle[i][j] << " ";

}

**else** {

// printing initial value of the puzzle with some formatting

std::cout << "X ";

}

}

std::cout << std::endl;

}

}

// Public member function that (recursively) implements the brute-force

// search for possible solutions to the incomplete Sudoku puzzle

**bool** Solve(**int** row, **int** col)

{

// this part of the code identifies the row and col number of the

// first incomplete (i.e. 0) entry in the puzzle. If the puzzle has

// no zeros, the variable row will be 9 => the puzzle is done, as

// each entry is row-, col- and block-valid...

// use the pseudo code of figure 3 of the description

//find blank. try from k=1 to k=9; check validity, use recursion. if it's valid, put k there, else put 0 back.

if we finish 1-9 this loop and still cannot find any k which is valid, return false. If we cannot find any blank anymore, we are done.

**if** (find\_blank(row,col)) {

**for** (**int** k=1;k<=9; k++){

**if**(row\_valid(row, k)&&col\_valid(col, k)&&block\_valid(row, col, k)){

puzzle[row][col]=k;

**if** (Solve(row,col)){

**return** **true**;}

**else** {

puzzle[row][col]=0;

}

}

} **return** **false**;

}

**else**

**return** **true**;

}

};

#endif

**//all\_solution part**

**bool** Solve(**int** row, **int** col)

{

// this part of the code identifies the row and col number of the

// first incomplete (i.e. 0) entry in the puzzle. If the puzzle has

// no zeros, the variable row will be 9 => the puzzle is done, as

// each entry is row-, col- and block-valid...

// use the pseudo code of figure 3 of the description

//this time, we cannot return true, otherwise it will stop once find a solution, so we just do recursion. same as we find one solution, except here we just do recursion. If we find a blank, we try k=1 to k=9, if it's valid, put k into that blank, do recursion, and put 0 back so that part won't infect next-time search. If we cannot find a blank, we finish one solution, so we print it out.

**if** (!find\_blank(row,col)) {

cout<<"solution" << i <<":"<<endl;

print\_puzzle();

i ++;

**return** **false**;

}

**for** (**int** k=1;k<=9; k++){

**if**(row\_valid(row, k)&&col\_valid(col, k)&&block\_valid(row, col, k)){

puzzle[row][col]=k;

Solve(row,col);

puzzle[row][col]=0;

}

}

**return** **true**;

}

};

#endif