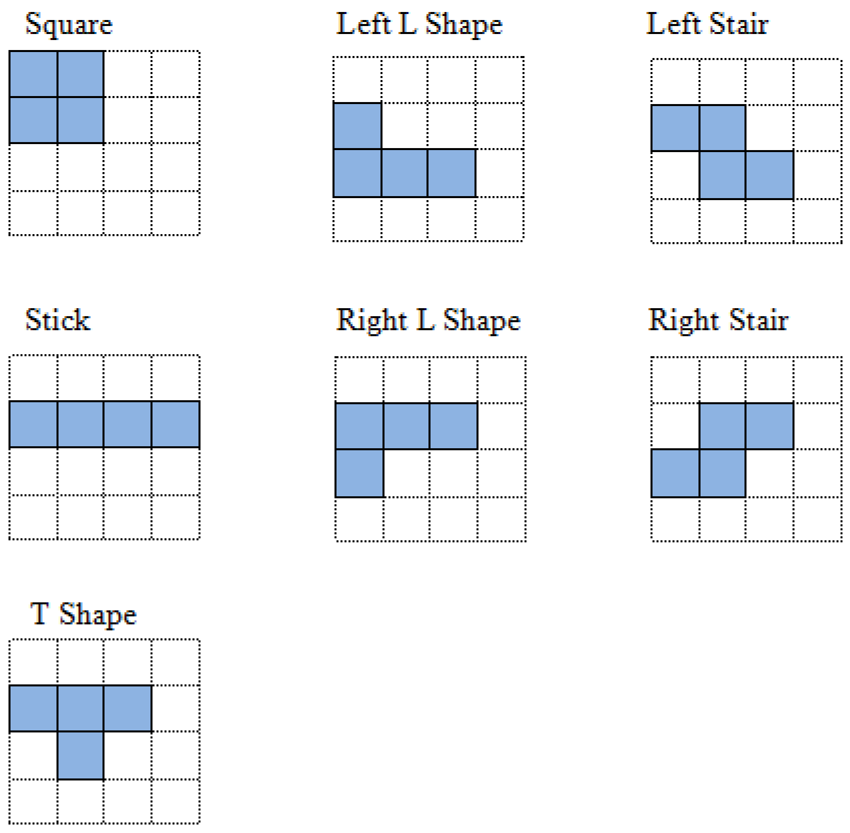
Programming Assignment # 5 Object Oriented Programming (Inheritance and MFC Document-View Architecture)	
Release Date	**03-Feb-2014**
Due Date	**14-Feb-2014 (In Lab Checking on your Laptops)**
Total Marks	**TBA**

Tetris

NOTE:

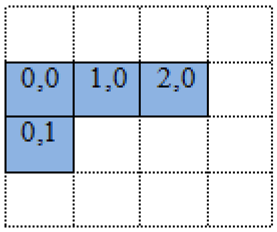
- In this assignment you can work in groups of two. Groups of 3 or more are not allowed!
- You've to register your groups by informing me in person in class, when asked.
- You have to make this game as MFC (Microsoft Foundation Class) Application.

Introduction:
Tetris is a puzzle game first introduced by Alexey Pajitnov in 1984, at Academy of Science of the USSR in Moscow. The game is played with special pieces each composed of four blocks.
Shown below are the 7 Tetris Pieces used in standard Tetris Game;



Description:
Your Tetris Piece is composed of four blocks. Each block is represented by two co-ordinates. Thus a TetrisPiece is represented by 4 sets of co-ordinates, one for each block. Shown below is the sample representation of a TetrisPiece. In above figure the Right L Shape is represented as [(0, 0), (1, 0), (2, 0), (0, 1)]. Shape is redrawn showing which co-ordinates represent which blocks.

Right L Shape with Coordinates



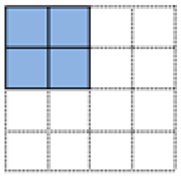
Also a Piece should have the value 'maxima' that is the maximum possible value of y, amongst all the blocks in the Piece. In above the maxima is 1, the 'y' coordinate of the block (0, 1). All the other blocks have y co-ordinate 0.

In the game, each of the Tetris Piece can be rotated anti-clock wise. Each Piece can be rotated into fixed number of distinct states then the original state is restored. For each Piece the maximum no of distinct rotations is different. For instance rotating the Stair Peice twice brings it back to original state. Shown below are distinct rotations for all pieces.

Distinct Rotations:

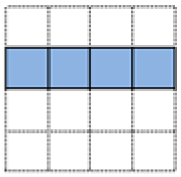
Square:
1 distinct rotation

State 0

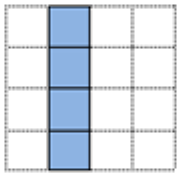


Stick:
2 distinct rotations

State 0

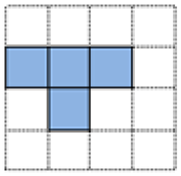


State 1

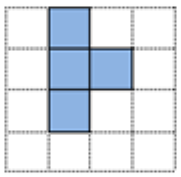


TShape:
4 distinct rotations

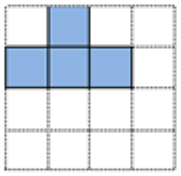
State 0



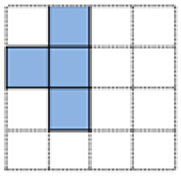
State 1



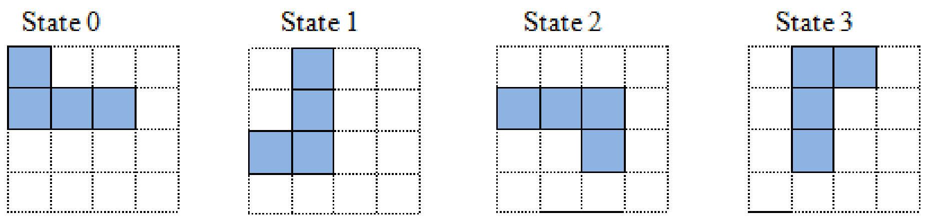
State 2



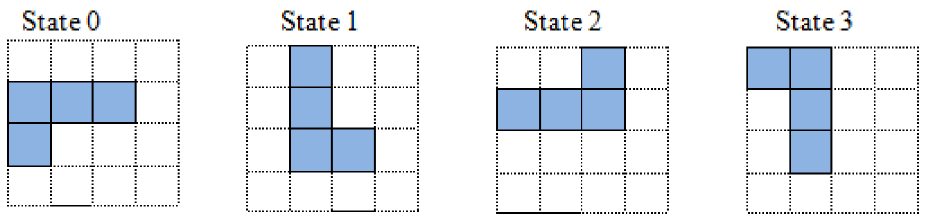
State 3



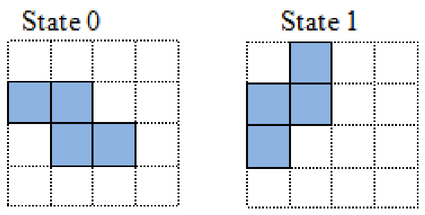
LeftLShape:
4 distinct rotations



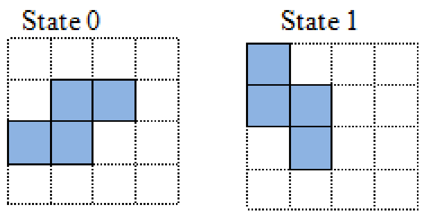
RightLShape:
4 distinct rotations



LeftStair:
2 distinct rotations

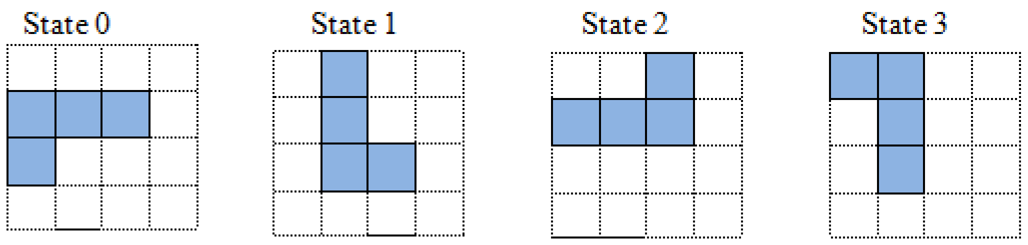


RightStair:
2 distinct rotations



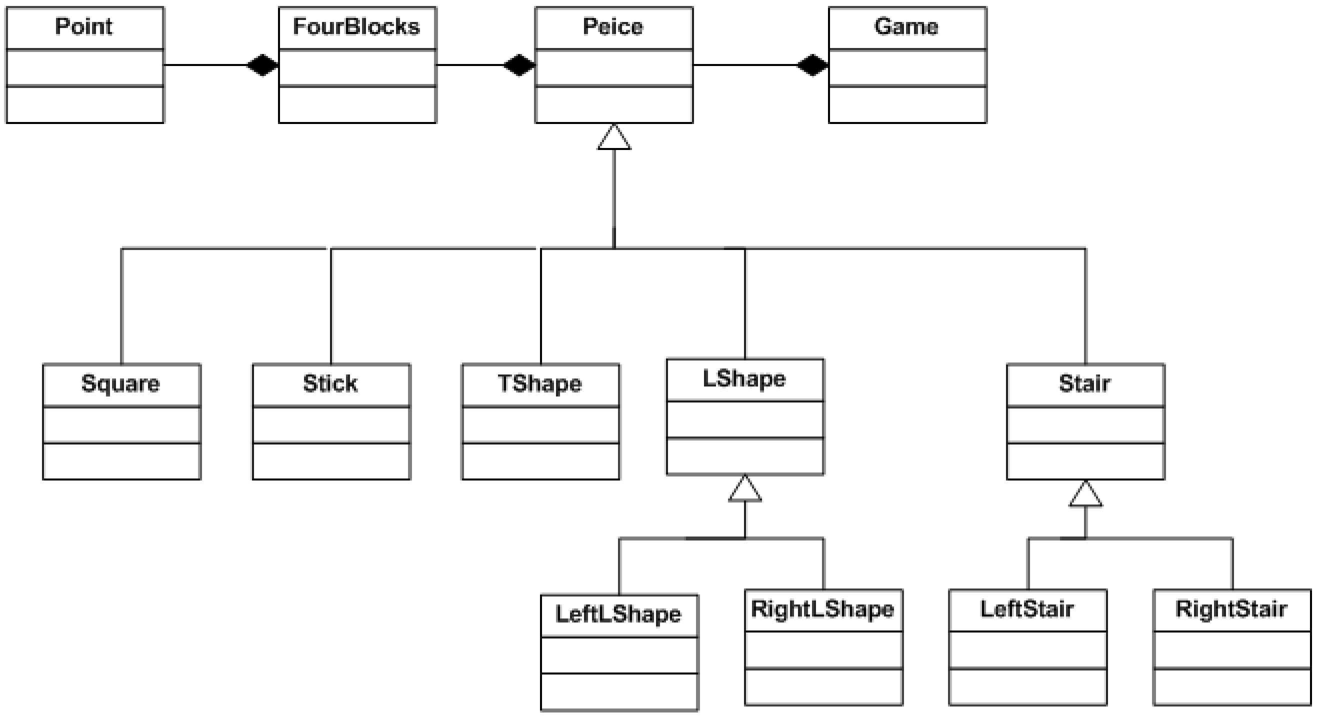
In our game we assume whenever piece is made it is always in its sate 0. The co-ordinates of each block in the piece are given values by assigning the co-ordinates (0, 0) to **top left block** of the piece at state 0. The remaining 3 blocks are numbered relative to that. In representation Co-ordinates are always saved starting from top-left block and reading each block at its right and then reading in same manner on the lower row and so on.

In rest of the states we assume the co-ordinates relative to the co-ordinates assigned in State 0. For instance, for RightLShape each of the rotation state is represented below:



State 0: [(0, 0), (1, 0), (2, 0), (0, 1)]
State 1: [(1, -1), (1, 0), (1, 1), (2, 1)]
State 2: [(2, -1), (0, 0), (1, 0), (2, 0)]
State 3: [(0, -1), (1, -1), (1, 0), (1, 1)]

Implementation:
You have to implement the following hierarchy of classes. They are discussed below.



Global Variables:
Declare following global variables.
```
const int BLOCK_HEIGHT = 30;
const int BLOCK_WIDTH = 30;
const COLORREF colorArray[5]={...};
```

`BLOCK_HEIGHT` and `BLOCK_WIDTH` specify size of a single block which determines the window size and height and width of blocks, of which our pieces are made.
In `colorArray` you must give white color (`RGB(255,255,255)`) at index 0. For the remaining four indices you can use any four colors of your choice.

Point:
`Point` is a struct having x and y co-ordinates. Also provide a setter void Point::set(int x, int y).

FourBlocks:
`FourBlocks` is a struct that represents four blocks of a piece. Since each block is represented by a point thus it has an array of four points `point blockPts[4]`. Also it has the `int maxima` (the maximum value of y coordinate in all blocks). Also provide constructors with following signatures:

```
FourBlocks();
FourBlocks(int x0,int y0, int x1, int y1, int x2, int y2, int x3, int y3, int maxima);
```

Second constructor will facilitate initializing co-ordinates of 4 blocks.

TetrisPiece:
`TetrisPiece` is a class having following data members:

```
COLORREF color;  //color of the TetrisPiece
int colorIndex;  //index to select value for 'color' from global const colorArray[5]
FourBlocks* fptr;  //fptr is a pointer that will be inherited in derived classes of
                   //TetrisPiece and will be made to point a fourBlocks structure
                   //as will be discussed later.

const int rotations;    //total distinct rotations possible for a peice
int currentState; //current state of the TetrisPiece. It will be used in setting fptr
```

Reason to keep `fptr`, `rotations` and `currentState`, and how to initialize them is still not clear. To understand their purpose fully let us assume the piece `RightLShape` has derived from class `TetrisPiece`. Since `RightLShape` can have 4 distinct rotations so the variable `rotations` must have value 4 for this class. Each state can be represented by 1 object of type `FourBlocks`. Thus to represent 4 states we need 4 objects of type `FourBlocks`. We will make an array `FourBlocks State[4]`. According to our representation scheme for `RightLShape` discussed above `State` will be initialized as follows.

```
State[0] = FourBlocks(0,0,1,0,2,0,0,1, 1);
State[1] = FourBlocks(1,-1,1,0,1,1,2,1, 1);
State[2] = FourBlocks(2,-1,0,0,1,0,2,0, 0);
State[3] = FourBlocks(0,-1,1,-1,1,0,1,1, 1);
```

Now `fptr` will point to any one of these declared states at any time. Let us say whenever a `RightLShape` is created it will be in state 0. Thus `currentState` is set to 0 and fptr wil point to `State[0]`. On an anti-clockwise rotation `currentState` will be updated to 1 and `fptr` will point to `State[1]`. On each rotation `fptr` points to next state in the array `State`. It starts over from State 0 once the rotation at last State is carried out.

Provide appropriate constructor. Color must set in the constructor. Our global `colorArray` has 4 colors from index 1 to 4 (excluding index 0, white color) so the color of `TetrisPiece` must be picked randomly from one of these four values. Set `colorIndex` to a random value from 1 to 4. Set `color` to the value `colorArray[colorIndex]`.

Provide getter setters for private/protected data members wherever necessary.

Provide a pure virtual function `rotateAntiClock()`.

Dearived Classes:
Your derived class hierarchy should be implemented according to UML diagram provided above.
Each derived class Square, Stick, TShape, LShape and Stair must initialize inherited member `rotations` equal to their respective distinct possible rotations.
Declare array of states as data member of each derived class as `FourBlocks State[rotations]`.
Also initialize inherited members `currentState` and `fptr` to appropriate values.

`LeftLShape` and `RightLShape` are further derived from class `LShape`. Each will initialize array State to different values.
Likewise `LeftStair` and `RightStair` are derived from class `Stair`.

Class Game:
Class `Game` has following structure;

```
class Game
{
private:
      int** board;
      int rows, cols;
      TetrisPiece* piecePtr;
      int currentRow;
      int currentCol;
      bool isGameOver;
public:
      Game();
      void selectNextPiece();
      void rotatePiece();
      void clearCompleteRows();
      COLORREF getCellColor(int col, int row);  //returns color of cell(c,r)
      void setCellColor(int c, int r, int val); //set index value of color at cell(c,r)
};
```

Game();
Provide a constructor initializing all data members to appropriate values. In constructor define a board of size `rows` by `cols`. Each cell of this grid holds an integer value from 0 to 4, which is used as an index to get appropriate color for each board cell to display on screen. Initially each cell is initialized to 0 that will result in white color. Board must have 20 rows and 15 columns.
`peicePtr` must be initialized to point to any one of the 7 different pieces on random.
`currentRow` represent the row number at which `TetrisPiece` must be displayed.
`currentCol` represent the col bumber at which `TetrisPiece` must be displayed.
`isGameOver` is initially set to false.

Provide appropriate getter setters.

void selectNextPiece();
It deletes previous value of `piecePtr` and points it to a new randomly selected piece.

void rotatePiece();
It rotates the piece that `piecePtr` is pointing at. A piece should not be rotated if resulting rotation causes it to go out of board boundaries.

Void clearCompleteRows();
It is called whenever a piece is placed on board and it clears the complete rows (i.e. their cell color is set to white).

MFC Application Document/View Architecture:
You will declare an object of `Game` class in Document header file. In View class in order to display you must understand how to display rectangles. In `OnDraw`( ) function draw the board. Also the `TetrisPiece` should be displayed. Write a function View Class `void DrawPiece(CDC* pDC)` that displays currently selected `TetrisPiece`. The point of display on screen, for each block in piece, can be computed by using 'representation co-ordinates' of the piece (saved in array of `TetrisPiece::fptr`), `currentCol`, `currentRow`, `BLOCK_HEIGHT` and `BLOCK_WIDTH`. Devise a formula to compute 'display co-ordinates' of a block.
CAUTION: Remember 'display co-ordinates' of a block are different from its 'representation co-ordinates'. `OnDraw()` function will call `DrawPiece(CDC* pDC)` and will pass `CDC` pointer to it.

To show movement of the piece the display must be redrawn again and again. You can use the function `RedrawWindow()` in following way;

```
//Redrawing the window display
CRect rcClient;
GetClientRect(&rcClient);
RedrawWindow(rcClient,0,RDW_ERASE|RDW_INVALIDATE);
```

You can use `Sleep(...)` function to give delays between redrawing.
Override event function `OnKeyDown(...)` in View class to enable left, right movements and rotations of Tetris Piece through keys.
In the game the completely filled rows should be eliminated and also the display should update. Game is over if the Tetris pieces are filled up to top. You are required to design logic for these functionalities of the game.