TETRIS-GAME

OBJECT ORIENTED PROGRAMMING PROJECT

made by: Mateusz Drygiel

Instructions:

left arrow key (<-) - move the piece left
right arrow key (->) - move the piece right
arrow down key (v) - move the piece down
“X” - rotate the piece once
“S” - rotate the piece once in an opposite direction
“B” - pause the game, make a ‘break’

Description of the program itself:

The program is a console application meant to imitate the famous game of Tetris. The goal of the game is to get as many points as possible by stacking the pieces and filling the lines. The player loses only when the field of the game when the blocks reach the top with no way for the player to continue playing. After the game is finished the player is presented with the achieved score and the “speed” he has achieved (the “speed” being the time interval of the falling blocks).

Description of methods and classes:

- tetromino.h:
  This file contains the class of Tetromino which in other words are just the pieces we use when playing a Tetris game. Besides attributes in it we can find the following getters:

  getActualBrickTemplate() - responsible for getting the template of our tetromino brick.

```
const char** Tetromino::getActualBrickTemplate() const
{
    return const_cast<const char**>( this->actualBrickTemplate );
}
```

  getFormat() - getting the format of each brick of the tetromino.

```
const char Tetromino::getFormat() const
{
    return this->brickform;
}
```

Methods:

print() - responsible for printing the tetromino,
using two loops inside the function and a pointer to the actualBrickTemplate.

```
void Tetromino::print() const
{
    for( size_t i=0; i<4; i++ )
    {
        for( size_t j=0; j<4; j++ )
        {
            cout << this->actualBrickTemplate[i][j];
        }
        cout << endl;
    }
}
```

rotate() - the rotation of our tetromino when the ‘X” key is pressed the program chooses the corresponding Tetromino template depending on how many there are (each piece’s rotation option vary, which will be more described in “actualBrickTemplate” below).

```
void Tetromino::rotate( int key )                                    //the rotation portio
{
    int index = this->actualBrickIndex;

    switch( key )
    {
    case 'x':
    case 'X':
        index = ( (index-1) + this->brickFormTemplate.size() ) % this->brickFormTemplate.size();
        this->actualBrickIndex = index;
        this->actualBrickTemplate = this->brickFormTemplate[index];
        break;


    }
}
```

*actualBrickTemplate - I would like to explain this portion as well to avoid any confusion, because at first glance it may look intimidating or a maybe unreadable, given the fact that the code is very long and looks repetitive. Each of the pieces is stored in a 4x4 matrix. In this matrix we have “empty” and “filled” blocks, where “filled” is one brick of the tetromino piece.
To resolve the problem with rotation each rotation of the piece has also been implemented in it, for example: if we have the “Square” shaped tetromino we need only 1 copy of this matrix because the piece will look the same every single time we perform the rotation. This action, while takes a little of space and in turn memory made it easier for resolving the rotation.

generateTemplate() - generating the template for each tetromino, this method among the others is necessary for the program to function properly. Given two ‘for’ loops to produce the template of the tetromino.

```
void Tetromino::generateTemplate( const char t[4][4] )
{
    this->brickFormTemplate.push_back( new char*[4] );

    for( size_t i=0; i<4; i++ )
    {
        this->brickFormTemplate[this->brickFormTemplate.size()-1][i] = new char[4];
    }

    for( size_t i=0; i<4; i++ )
    {
        for( size_t j=0; j<4; j++ )
        {
            this->brickFormTemplate[this->brickFormTemplate.size()-1][i][j] = t[i][j];
        }
    }
```

- field.h:
  File containing the methods and class of the Bucketfield for the Tetris game.
  In it we find attributes, and the following structure of getters and setters:

```
struct Point
{
    unsigned short x;
    unsigned short y;

    Point() : x(0), y(0){}

    unsigned short getX() const //getter
    {
        return this->x;
    }

    unsigned short getY() const     //ge
    {
        return this->y;
    }

    void setX( const unsigned short x )
    {
        this->x = x;
    }

    void setY( const unsigned short y )
    {
        this->y = y;
    }
};
```

  Each of the getters/setters is used for the horizontal and diagonal information.

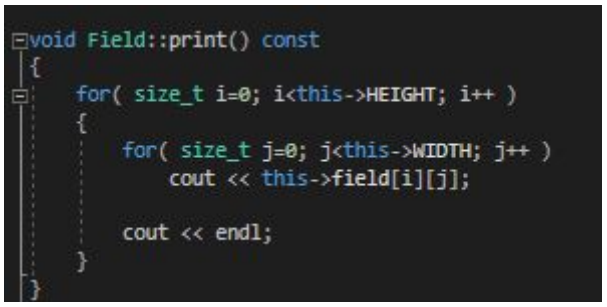
Outside of the struct we can also find the getters:

```
size_t getting_Hi() const; /
size_t getting_Wi() const;
const Point& getCursor() con
```

They are responsible for getting the height, width of the bucketfield, as well as the cursor which “guides” the tetropieces on the field.

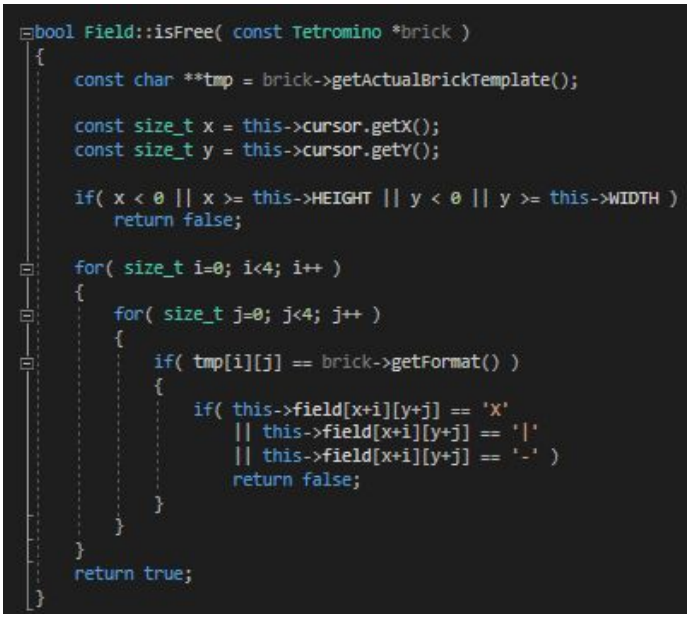
Methods:

print() - responsible for printing the bucketfield, using 2 loops for width and height.



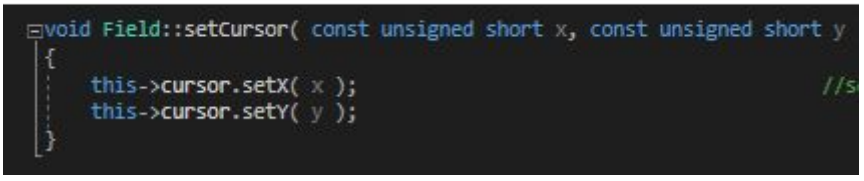
```
void Field::print() const
{
    for( size_t i=0; i<this->HEIGHT; i++ )
    {
        for( size_t j=0; j<this->WIDTH; j++ )
            cout << this->field[i][j];

        cout << endl;
    }
}
```

isFree() - checking if a position on the field is not occupied by any block or boundry, so that it is possible to place a tetromino piece.



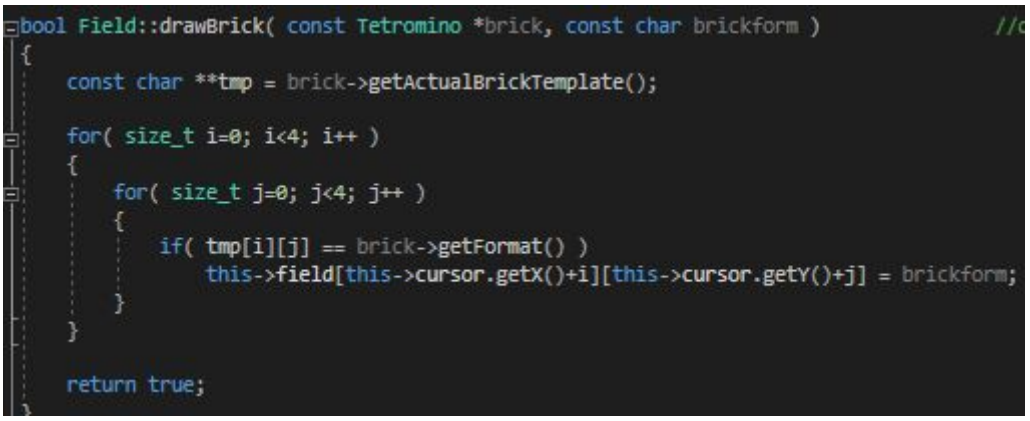
```
bool Field::isFree( const Tetromino *brick )
{
    const char **tmp = brick->getActualBrickTemplate();

    const size_t x = this->cursor.getX();
    const size_t y = this->cursor.getY();

    if( x < 0 || x >= this->HEIGHT || y < 0 || y >= this->WIDTH )
        return false;

    for( size_t i=0; i<4; i++ )
    {
        for( size_t j=0; j<4; j++ )
        {
            if( tmp[i][j] == brick->getFormat() )
            {
                if( this->field[x+i][y+j] == 'X'
                    || this->field[x+i][y+j] == '|'
                    || this->field[x+i][y+j] == '-' )
                    return false;
            }
        }
    }
    return true;
}
```

setCursor()- setting the cursor on the bucketfield.



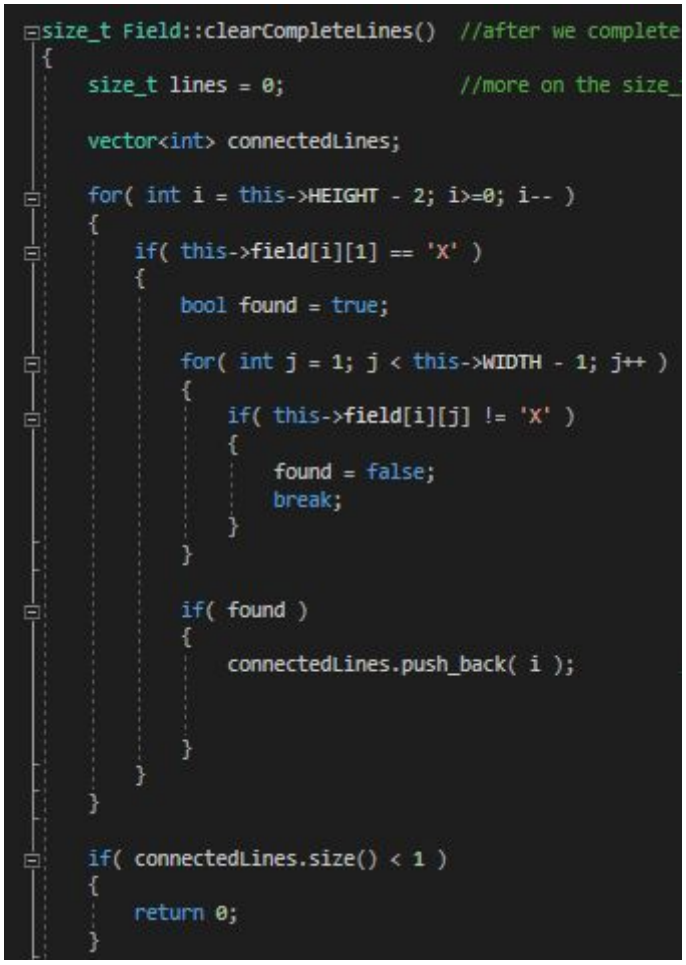
```
void Field::setCursor( const unsigned short x, const unsigned short y
{
    this->cursor.setX( x );                                    //s
    this->cursor.setY( y );
}
```

drawBrick()- drawing the tetromino piece on the bucketboard



```
bool Field::drawBrick( const Tetromino *brick, const char brickform )
{
    const char **tmp = brick->getActualBrickTemplate();

    for( size_t i=0; i<4; i++ )
    {
        for( size_t j=0; j<4; j++ )
        {
            if( tmp[i][j] == brick->getFormat() )
                this->field[this->cursor.getX()+i][this->cursor.getY()+j] = brickform;
        }
    }

    return true;
}
```

clearCompleteLines()- clearing the completed lines once the row is "filled". After this action is performed we dynamically allocate a new bucket field with the pieces moved downwards.



```
size_t Field::clearCompleteLines()  //after we complete
{
    size_t lines = 0;               //more on the size_

    vector<int> connectedLines;

    for( int i = this->HEIGHT - 2; i>=0; i-- )
    {
        if( this->field[i][1] == 'X' )
        {
            bool found = true;

            for( int j = 1; j < this->WIDTH - 1; j++ )
            {
                if( this->field[i][j] != 'X' )
                {
                    found = false;
                    break;
                }
            }

            if( found )
            {
                connectedLines.push_back( i );


            }
        }
    }

    if( connectedLines.size() < 1 )
    {
        return 0;
    }
```

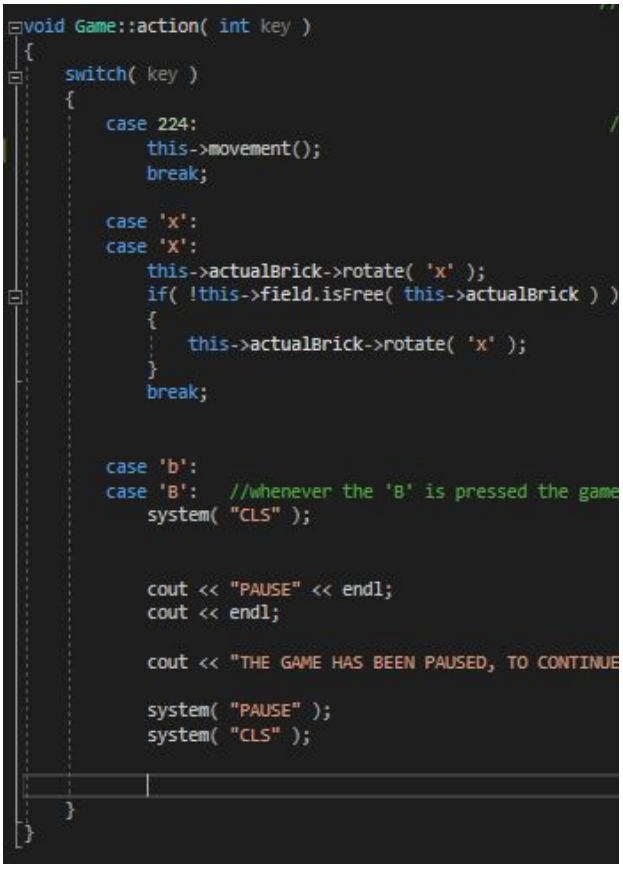
- game.h:
  This file contains the class, attributes and methods for the Game portion of the Tetris game. It contains the following getters:

```
Field& Game::getField()
{
    return this->field;
}

Tetromino* Game::getActualBrick() const
{
    return this->actualBrick;
}

const size_t& Game::getIntervall() const
{
    return this->intervall;
}

const size_t& Game::getHighscore() const
{
    return this->highscore;
}
```

getField() - getting the bucketfield where the game is held.
getActualBrick()- getting the tetromino piece for the game.
getIntervall() - getting the time interval for how fast a piece is falling inside the bucketfield.
getHighScore() - getting the current score of the player (the 'high' score is because in the first iteration it was indeed supposed to display the highest score achieved, this has been changed into the 'current' but the name remains the same).

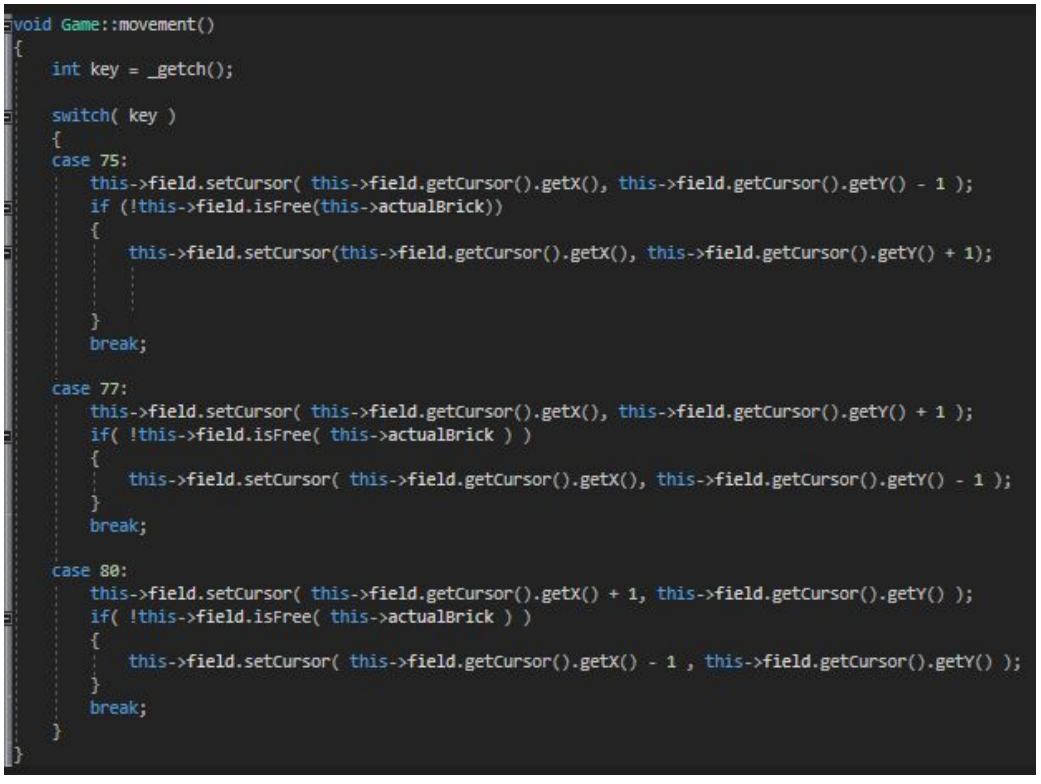
Methods:

action() - performing an action to happen on the screen with the use of the keyboard keys.



```
void Game::action( int key )
{
    switch( key )
    {
        case 224:
            this->movement();
            break;

        case 'x':
        case 'X':
            this->actualBrick->rotate( 'x' );
            if( !this->field.isFree( this->actualBrick ) )
            {
                this->actualBrick->rotate( 'x' );
            }
            break;


        case 'b':
        case 'B':   //whenever the 'B' is pressed the game
            system( "CLS" );


            cout << "PAUSE" << endl;
            cout << endl;

            cout << "THE GAME HAS BEEN PAUSED, TO CONTINUE

            system( "PAUSE" );
            system( "CLS" );

    }
}
```

movement() - the movement of the tetropiece using the keyboard keys.



```
void Game::movement()
{
    int key = _getch();

    switch( key )
    {
    case 75:
        this->field.setCursor( this->field.getCursor().getX(), this->field.getCursor().getY() - 1 );
        if (!this->field.isFree(this->actualBrick))
        {
            this->field.setCursor(this->field.getCursor().getX(), this->field.getCursor().getY() + 1);


        }
        break;

    case 77:
        this->field.setCursor( this->field.getCursor().getX(), this->field.getCursor().getY() + 1 );
        if( !this->field.isFree( this->actualBrick ) )
        {
            this->field.setCursor( this->field.getCursor().getX(), this->field.getCursor().getY() - 1 );
        }
        break;

    case 80:
        this->field.setCursor( this->field.getCursor().getX() + 1, this->field.getCursor().getY() );
        if( !this->field.isFree( this->actualBrick ) )
        {
            this->field.setCursor( this->field.getCursor().getX() - 1 , this->field.getCursor().getY() );
        }
        break;
    }
}
```

initializeNewBrick() - initialization of the new tetromino piece.
If the next brick is not nullptr then it gets initialized, else a RandomBrick() funtion is called giving us 1 of the 7 tetrominos at random (using srand), later the program is checking if there is space in the “spawn zone” for the brick, if there is then it gets initialized, if not then the game is over, hence the isGameOver=true.

```
bool Game::initializeNewBrick()
{
    if( this->actualBrick != nullptr )
        delete this->actualBrick;

    this->field.setCursor( 0, ( this->field.getting_Wi() / 2 ) - 1 );

    if( this->nextBrick != nullptr )
    {
        this->actualBrick = this->nextBrick;
    }
    else
    {
        this->actualBrick = getRandomBrick();
    }
    this->nextBrick = getRandomBrick();

    if( this->field.isFree( this->actualBrick ) )
    {
        return true;
    }
    else
    {
        this->isGameOver = true;                //when the gameover is
        return false;
    }
}
```

running() - the initial running of the program/ setting the game in motion/returning unless it hits GameOver.

```
bool Game::running()
{
    return !this->isGameOver;
}
```

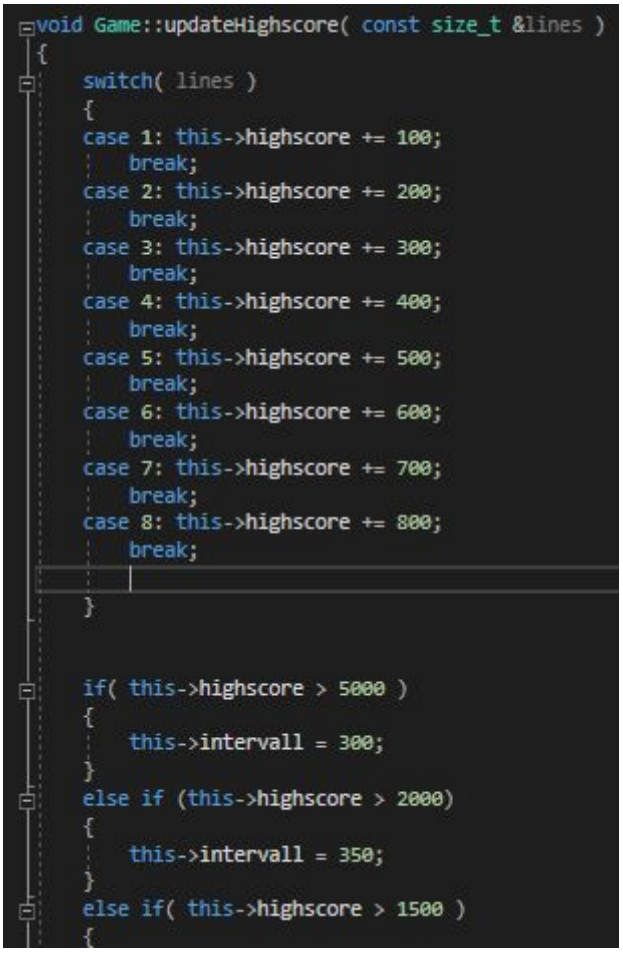
print() - function with pointer to print bucketfield.

```
void Game::print() const
{
    this->field.print();
}
```

printNextBrick() - printing the next Tetromino piece by setting a pointer.

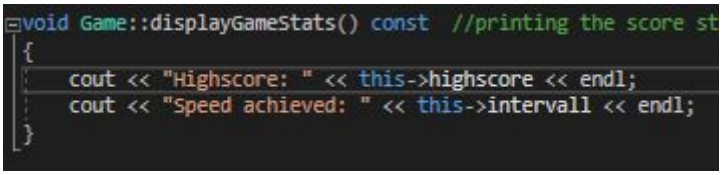
```
void Game::printNextBrick() const
{
    if( this->nextBrick != nullptr )
    {
        this->nextBrick->print();
    }
}
```

updateHighScore() - updating the current score of the player. Using switch the program is able to assign the correct amount of points, depending on how many lines have been cleared by the player (in this case I did only 8 as max, but it can be easily changed to even 100).
In the same function the speed interval for the falling pieces also is changing depending on the player's current score, making it harder for him to proceed further.



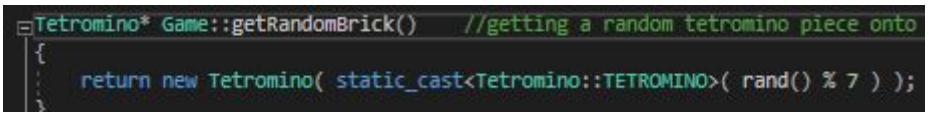
```
void Game::updateHighscore( const size_t &lines )
{
    switch( lines )
    {
    case 1: this->highscore += 100;
        break;
    case 2: this->highscore += 200;
        break;
    case 3: this->highscore += 300;
        break;
    case 4: this->highscore += 400;
        break;
    case 5: this->highscore += 500;
        break;
    case 6: this->highscore += 600;
        break;
    case 7: this->highscore += 700;
        break;
    case 8: this->highscore += 800;
        break;

    }


    if( this->highscore > 5000 )
    {
        this->intervall = 300;
    }
    else if (this->highscore > 2000)
    {
        this->intervall = 350;
    }
    else if( this->highscore > 1500 )
    {
```

displayGameStats() - printing the information to the player after the game is finished, describing what his final score was, the time played and also the speed interval he has achieved during the game.



```
void Game::displayGameStats() const  //printing the score st
{
    cout << "Highscore: " << this->highscore << endl;
    cout << "Speed achieved: " << this->intervall << endl;
}
```

getRandomBrick() - getting the random tetropiece using srand.



Sidenote:

The total time that the project took was somewhere around 2 weeks. During the time I separated the steps into a few pieces, getting to know how to create the pieces and the bucket for them, the game logic and how to make it perform well without many bugs as well as making sure that the whole thing is displayed in the CMD.