Greyson Beck

805010132

Project 3 Report

**Description of Member Functions**

1. **Class Game**

\*\*\*Note: For all functions in the Game class, none are marked virtual because Game is not a base class/has no derived classes. Thus, there is no need for the virtual keyword in front of any member functions.

~Game():

Deletes the Piece pointers, currPiece and nextPiece (which point to dynamic memory), because the compiler generated destructor would not free dynamically allocated memory correctly.

void displayStatus():

This function displays the game’s score, rows left, and level by moving the m\_screen data member (which is mainly why it must be a part of Game) to specific X,Y coordinates and then printing the score, rows left, and level at those points using the printString function. To deal with the right-justify and character alignment, it calls the ‘justify’ function also defined in Game. It is not const because it operates on multiple private data members. It also must be a part of Game because it is one of the overarching ‘parts’ of the whole Tetris game, and the Game class deals with the aesthetics when it comes to status, prompts, etc.

Bool checkCollisionDown(Piece\* piece) const:

This function accepts a Piece pointer as a parameter, and checks to see what PieceType it is. If the type is that of a Vapor Bomb or a Foam Bomb, then the ‘Well::isTaken1’ or ‘Well::isTaken2’ functions are called, depending on whether it is Vapor or Foam. Otherwise, the well’s ‘Well::isTaken’ function is called, because the piece is either normal or a crazy piece, all of which have 4 coordinate pairs as part of their piece. The isTaken function will be discussed in more detail later, but its implementation in this function is that it is called for each coordinate pair in the piece (or 2 if type = Vapor Bomb, or 1 if type = Foam Bomb). If it is false for every pair, then then this function returns false. Otherwise, it returns true, indicating there is a collision below. It is marked const because the function does not modify any variables including that which is passed in, and the functions which it calls inside (Well::isTaken) also does not modify any variables. This is in the Game class because only the game has access to both the well (via m\_well) and Piece through the currPiece and nextPiece pointers. Trying to access the Piece class through Well would require Well to have a Piece pointer to the dynamically allocated pieces, and that would create a lot of headache. Game is the centralized platform with fairly easy access to both.

Bool checkCollisionRight(Piece\* piece) const:

This function works almost identically to the checkCollisionDown function above, except that it calls the Well::isTaken function on points that are to the right of each coordinate pair. In other words, each x coordinate entered for the isTaken function has 1 added to it, such that the coordinates being checked are immediately to the right. Like checkCollisionDown, this function checks if the PieceType is a Foam Bomb or a Vapor Bomb, and calls the Well class’s function specifically for that special piece, otherwise the Well::isTaken function is called. For all control paths, the function returns true if at least one of the spaces to the right of the piece is taken, false otherwise. It is marked const because the function does not modify any variables including that which is passed in, and the functions which it calls inside (Well::isTaken) also does not modify any variables. . This is in the Game class because only the game has access to both the well (via m\_well) and Piece through the currPiece and nextPiece pointers. Trying to access the Piece class through Well would require Well to have a Piece pointer to the dynamically allocated pieces, and that would create a lot of headache. Game is the centralized platform with fairly easy access to both.

Bool checkCollisionLeft(Piece\* piece) const:

This function works almost identically to checkCollisionDown and checkCollisionRight, except that it checks the coordinates to the immediate left (surprise!) of each coordinate pair of the piece. Well::isTaken is called to do most of the work for all pieces that are not either Vapor Bomb or Foam Bomb. Should the piece argument passed in be a Foam Bomb or the Vapor Bomb, the Well’s ‘isTaken1’ or ‘isTaken2’ is called accordingly. For all control paths, the function returns true if at least one of the spaces to the left of the pieces is taken, false otherwise. It is marked const because the function does not modify any variables including that which is passed in, and the functions which it calls inside (Well::isTaken) also does not modify any variables. . This is in the Game class because only the game has access to both the well (via m\_well) and Piece through the currPiece and nextPiece pointers. Trying to access the Piece class through Well would require Well to have a Piece pointer to the dynamically allocated pieces, and that would create a lot of headache. Game is the centralized platform with fairly easy access to both.

Void updateGame(Piece\* piece):

This function takes a pointer to a piece as its parameter, and it checks to see if the type is Vapor Bomb, Foam Bomb, or any of the other piece types. It calls the respective ‘update’ function in well depending on piece type, and no there is no inheritance in this particular area of the code. When I say the respective function, I am simply referring to individual functions in the Well class that were each designed to update the wellGrid based on different piece types. After the wellGrid is updated, this function deletes the current piece and sets it equal to the next piece. At this point, a ‘new’ next piece is determined, with the chooseRandomPiecetype() function passed into a switch statement. The next piece is then dynamically allocated via the nextPiece pointer. The function then sets the coordinates for the *current piece* to fall from, and also checks to see how many rows, if any, have been vaporized. The rowsNeeded and score are changed accordingly. Finally, the well is displayed as well as the game’s status.

Bool playOneLevel():

This function is, of course, where all the magic happens. The function chooses (and sets the coordiantes of) the current piece AND next piece via a switch statement with the chooseRandomPieceType() function passed in, and it also instantiates the timer. These code segments are only run once per level, as only one timer is needed, and the current piece and next pieces are allocated JUST to start the game (Recall in ‘updateGame’ above, the transfer of next piece to current piece and allocating a new next piece is taken care of). Then, a ‘code’ int is declared to help with updating the game later on. At this point, the program enters a while loop that will keep that level running as long as the (rows needed to win) > 0 and the well’s top row has not exceeded the boundary of the well. Next, the current and next pieces are displayed in their respective positions, and the timer begins. A second while loop runs as long as the elapsed time is < (a ‘max’ function that takes into account the current level vs a constant, and returns the greater), and in this time the getCharIfAny function is called, and the user can input commands to the piece. This is implemented via switch statement…Very important: If a user tries to manually move left, right, or down, the function first checks if the piece is a CRAZY PIECE, and if it is, then it checks the reverse of either checkCollisionLeft or checkCollisionRight, (because of the adverse effects of this particular piece). If it is not a CRAZY PIECE, then normal checkCollisionLeft, CheckCollisionRight, or checkCollisionDown is called, and (only for the right and left collision checks) if that returns true then another check occurs to see if the collision is with the side of the well. Should it be with the side of the well, then the statement breaks, avoiding a problem where the user could try to move to the side without any room, but the program takes that as a collision and replaces the ‘#’s with ‘$’s in the well. This extra check solves that issue with the break statement. After the switch statement is done, the screen is cleared and everything is displayed again to avoid excessive lag with the piece’s displays. Next, once out of the switch statement (regardless of whether there was user input or not), checkCollisionDown call AND a check to see if ‘code’ = 0, which indicates that user input did not lead to a collision already. If there is no collision but ‘code’ is still = 0, then the screen clears and the piece is moved down, then the well is displayed again along with the status. This function must be in Game not only because that is where it was originally placed in the skeleton, but most importantly because Game has access to the Screen, Well, and Piece classes. playOneLevel() is called by another Game function, play(), so to better transition between levels and to gain access to all of Game’s member variables (ie. the level, score, rows left) this function must be in Game.

int max(int a, int b) const:

Returns the greater of the two integers entered, which is useful for determining the time elapsed of the timer’s ‘tick.’ It is called in playOneLevel to determine the greater of two possible times, so that the pieces do not fall too fast once higher levels are achieved. It is marked ‘const’ because the function only does a comparison between the two arguments and nothing else. This function is in Game so that it can be easily accessed during the one time it is needed in each level, and because one of the arguments passed in is the private member variable, m\_level.

void justify(std::string &one, std::string &two, std::string &three):

This function deals with the portion of the display() function where the score, rows left, and current level must be right justified in a seven character wide field. It uses the std::to\_string function from <string> to set the score, rows left, and level to separate string variables. It then subtracts the length of each of these strings from 7 to determine how many spaces should be printed out, and this is stored in the one, two, and three arguments passed by reference. The function is not const because it modifies variables that are passed into it. It is part of game because it needs access to the m\_score, m\_rowsNeeded, and m\_level private data members, and it needs to be called in the displayStatus() function. Thus, Game is both the necessary and ideal location for it.

int getLevel() const:

This function returns the m\_level variable, which just indicates the level. It is const because it just returns the m\_level member variable, and this is also why it must be in the Game class.

int getScore() const:

This function returns the m\_score variable, which just indicates the score of the game. It is const because it just returns the m\_score member variable, and this is also why it must be in the Game class.

void setScore(int rowsFilledatOnce):

This function implements a switch statement that takes in the rowsFilledatOnce, and increments the m\_score variable according to the scoring guidelines of the spec. More points are awarded for a greater number of rows filled at the same time. This is not const because it does in fact modify the m\_score member variable. In order to set the score, this function must be in Game so it can access the m\_score private member variable.

1. **Class Well**

Well():

Obviously not marked const because it is the class’s constructor, and it is not virtual because constructors cannot be marked virtual. This constructor loops thru wellGrid member array, initializing each cell to empty, except for the sides and bottom of the well, which are given the ‘@’ chars to symbolize the boundaries.

\*\*\*Note: For all member functions in Well, none are marked virtual because the class is not a base class/has no derived classes\*\*\*

void display(Screen& screen, int x, int y):

This function uses 2 loops to access each element of the wellGrid member array. At each point, it uses the Screen reference to print out the contents of said cell. It is not marked const because it modifies the screen variable, which must be passes in by reference for any changes made to take effect outside this function. The two int parameters are utilized as the starting points for each dimensions for the loops. This function is in the Well class because there must be a way for the Well to be displayed, since you cannot just return the array which holds the well (at least for my implementation I took that route).

Void updateWell(int x1, int y1, int x2, int y2, int x3, int y3, int x4, int y4):

This function takes in 4 sets of coordinate pairs representing those of a Piece + some integer (depending on type of collision). It is used to update the contents of the well after a collision has occurred with any piece type except the Foam Bomb or Vapor Bomb, and it does so by setting each of the four points on the grid equal to ‘$’ character. This function is not marked const because it modifies the wellGrid member array. It is in the Well class because in order to update the wellGrid array, it needs access to the private data member itself…accessing and mutating from the outside would be very tedious.

Void updateWellVapor(Piece\* vaporPiece):

This function utilizes two loops to check if that element is not equal to a ‘@’ char, which would mean the vapor bomb cannot blow that portion of the wellGrid up. If the element is in fact not equal to ‘@’ then it loops thru the cells above the original piece (if available) and moves down until each cell not equal to ‘@’ is filled in with a blank (‘ ‘) (if available). It is not marked const because the wellGrid member array is almost always modified in the process. It is in the Well class because in order to update the wellGrid array, it needs access to the private data member itself…accessing and mutating from the outside would be very tedious.

Void updateWellFoam(Piece\* foamPiece):

This function calls its helper function, updateWellFoamHelp, to do all of the work. However, before it does that, this function sets the original cell where the Foam Bomb landed equal to ‘\*’.

It is not marked const because it always modifies the member array, wellGrid. It is in the Well class because in order to update the wellGrid array, it needs access to the private data member itself…accessing and mutating from the outside would be very tedious.

Bool updateWellFoamHelp(int sr, int sc, Piece\* foamPiece):

This function does most of the work for updateWellFoam, as mentioned above. It recursively checks each possible path (like a maze) that an object could take within a 5x5 bounding box. It does this recursively, with four base cases…If the cell’s position is not equal to a blank (‘ ‘), if the starting row (sr) or starting column (sc) is less than zero, if the current cell being accessed is vertically out of bounds (within cells of original position), or if the current cell being access is horizontally out of bounds. Following the base cases, the current cell is filled in with ‘\*’ char, and then the original function is called for above, right, down, and left of the current position. This function is not marked const because it always modifies the wellGrid member array. This function as well needs access to the wellGrid member array, so putting it in Well simplifies the problem. It can then be called by the updateWellFoam function, otherwise a pointer to the Well class would be needed, along with a function to mess with the grid, which would be a nightmare.

int vaporizeRows():

Because this function calls the vaporizeRowHelp function, which does modify the member array, this function cannot be marked const. This function initializes both a ‘count’ int and ‘numRowsGone’ int to 0. It then loops thru the wellGrid and increments count for each cell that equals ‘$’ or ‘\*’. Every time it breaks out of the inner loop, there is a check to see if count equals (width of well – 2), as this means the entire row is filled. If this condition is met, the vaporizeRowHelp function is called on the current row, and numRowsGone is incremented. Then, count is set back to 0 to start the check all over again. At the end of the entire looping process, numRowsGone is returned.

Void vaporizeRowHelp( int row ):

This function is called by the vaporizeRows() function to actually implement the vaporizing and shifting process. It first loops through each element of the row that needs to be vaporized, assigning each cell’s value to blank (‘ ‘). Then, it executes a ‘for’ loop that starts with the row argument passed in, and goes across that row, setting the previously vaporized cells equal to the character in the cell directly above them. This function is not marked const because it modifies the member array, wellGrid.

int topRow() const:

This function loops thru each cell of the wellGrid, but stops and returns its current row once the first ‘$’ is found. Otherwise, the well is empty and the function returns the depth of the well. Because no changes to data members occurs in this function, just ‘getting,’ it is marked const. This function also needs access to the wellGrid member array, so that is why it exists in Well.

Char getCell(int x, int y) const:

This function simply returns the char that is at position x,y in wellGrid (wellGrid[y][x]), because you must swap to deal with system used in game), which is why it is marked const. No changing of any members, simply a retrieval of a value, which is also why it must be in the Well class, so that it can retrieve the cell in wellGrid.

bool isTaken(int x, int y) const:

This function utilizes the getCell function described above to access the cell at the given x,y position in wellGrid. If that value is a ‘$’, ‘@’, or a ‘\*’, then it returns true, otherwise false. The function is marked const because there is no setting of values, it just accesses a cell and returns a Boolean, nothing more. This function does not necessarily have to be in the Well class because it uses the getCell function to access wellGrid, but for the sake of organization it is logical to place it amongst other functions relating to the wellGrid and its values.

1. **Class Piece**

Piece() : orientation(0), xCoord(16), yCoord(4) :

The general Piece constructor sets the initial orientation to 0, the xCoord to 16, and the yCoord to 4 via the member initialization list. This is done because every time a new Piece is dynamically allocated and set to next Piece, it will need to be displayed in its default orientation at (16,4). The only exception to this is at the start of each level, when the very first piece’s xCoord and yCoord is respectively (3,0). Of course, the constructor for a class must be in the class itself.

virtual ~Piece():

Because I deal with dynamically allocated objects using the Piece class as a base pointer, I need to declare the destructor virtual, even though the body of the destructor does nothing. The destruction of the currPiece and nextPiece pointers is handled in Game’s destructor, but it is still necessary to make one here and mark it virtual so when those pointers do get destructed, the process does not skip over the derived Piece’s destructors. Obviously, the destructor for the class must be in the class itself.

Void moveDown():

This function moves a Piece down by incrementing its yCoord by one. This function is not marked virtual because all Pieces have a common method of moving down. It is not const because a data member, yCoord is modified, as mentioned. This function needs access to the yCoord data member, so it must be in Piece.

virtual void moveLeft():

This function decrements the xCoord variable by one, moving the object to the left by unit of one. Because of this, the function cannot be marked const. However, it is virtual because every Piece can move left, the crazyPiece’s move to the right when this function is called (in order to implement its special action of reverse movement simpler). This function needs access to the xCoord data member, so it must be in Piece.

virtual void moveRight():

This function increments the xCoord variable by one, moving the object to the right by unit of one. Because of this, the function cannot be marked const. It is, however, marked virtual because every Piece can move right, but the crazyPiece’s move to the left when this function is called (in order to implement its special action of reverse movement simpler). This function needs access to the xCoord data member, so it must be in Piece.

Void rotate():

This function rotates a Piece via a switch statement that takes in the object’s orientation, and just sets the new orientation with the setOrientation function (it could be done with the orientation member variable, but both accomplish the same task). It is not marked const because it changes the orientation member variable, and it is not virtual because all Pieces have the same 4 rotation options, and rotate in the same way. They don’t look the same when they rotate, but the process is the same. This function deals with Piece operations and movement, so it should be in the Piece class.

virtual void displayPiece(Screen& screen) const = 0:

This function is a pure virtual function because a general piece will never be displayed, only *specific* types of Pieces will be displayed and created. Thus, it become an Abstract Base Class and fits the description perfectly. It is marked const because the display functions in the derived Piece classes do not modify any base class member variables, and they do not have any of their own. To display a piece, it is helpful to have access to the xCoord and yCoord variables, so that is why, if not just for organization’s sake, it is in the Piece class.

int getOrientation() const:

This function simply returns the object’s orientation member variable, so it can be marked const. It is not marked virtual because every Piece deals with orientation in the same manner. The orientation data member is in Piece, so this function must be in Piece in order to access it.

int getxCoord() const:

This function returns the xCoord member variable, and since each type of Piece has the same coordinate system, there is no need to mark it virtual and implement it differently based on class. It is const because it just returns a value. The xCoord data member is in Piece, so this function must be in Piece in order to access it.

int getyCoord() const:

This function returns the yCoord member variable, and since each type of Piece has the same coordinate system, there is no need to mark it virtual and implement it differently based on class. It is const because it just returns a value. The yCoord data member is in Piece, so this function must be in Piece in order to access it.

void int setxCoord(int n):

This function takes in an integer argument and sets the xCoord member variable equal to that value. Because this member variable is altered, the function cannot be const. It is not virtual because every type of Piece sets the x coordinate in the same way. In order to modify the xCoord variable, this function must be in xCoord’s class, so that is why it is in Piece.

void int setyCoord(int n):

This function takes in an integer argument and sets the yCoord member variable equal to that value. Because this member variable is altered, the function cannot be const. It is not virtual because every type of Piece sets the y coordinate in the same way. In order to modify the yCoord variable, this function must be in yCoord’s class, so that is why it is in Piece.

PieceType getPieceType() const:

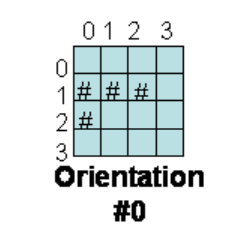
This function only returns the type member variable, so it can and is marked const. Because every piece has a type, this function is not marked virtual. The PieceType enum declaration is in the Piece class, thus to return a value of that type, this function must be in its scope. It is not virtual because every Piece has a type and it can be returned by this function.

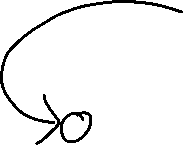
void setPieceType(PieceType piece\_type):

This function sets the object’s type to piece\_type, so it cannot be marked const. It is not virtual because every Piece can use this exact functionality, as each Piece only has one type variable. It is in Piece because the PieceType enum declaration is in Piece.

\*\*\*Note: The following coord() functions implement almost identical code, the only difference is the specific spot in the Piece where it is applied. All of them are declared in Piece because they refer to specific spots in each Piece. All of them are marked const because they do not modify any data members, and they do not call any functions that do so either. They just return an int representing the exact x/y position (depending on xCoord vs yCoord) of a specific part of a Piece, and they are Piece specific (so it takes into account the Piece’s xCoord PLUS where that piece ACTUALLY is displayed in the grid). The general pattern I followed was the ‘first’/0th xCoord (xCoord0()) was farthest left and farthest up, and went from left to right, top to bottom). Yes, I hard coded it to make other parts of the project easier, namely the collision process. I do apologize for the explanation, but it is challenging to illustrate what I did exactly. Basically, in each function I ran a series of if statements to determine the Piece’s type, then a switch on that Piece’s orientation, and then the function returns the xCoord/yCoord I am looking for.

Example:





xCoord for that specific part of piece = 0

virtual int xCoord0() const:

This function returns the first xCoord in the specific Piece.

int xCoord1() const:

This function returns the second xCoord in the specific Piece.

int xCoord2() const:

This function returns the third xCoord in the specific Piece.

int xCoord3() const:

This function returns the fourth and last xCoord in the specific Piece.

virtual int yCoord0() const:

This function returns the first yCoord in the specific Piece.

virtual int yCoord1() const:

This function returns the second yCoord in the specific Piece.

int yCoord2() const:

This function returns the third yCoord in the specific Piece.

int yCoord3() const:

This function returns the fourth and last yCoord in the specific Piece.

\*\*\*Note: Each of these derived classes has the function:

virtual void displayPiece(Screen& screen) const

To avoid excessive repetition in the following documentation, I am describing here because the implementation for each class follows the exact same procedure, it just prints in different areas of the 4x4 bounding box for each piece. Basically, the function calls a switch statement on the Piece’s orientation, and depending on which orientation it is in, it goes to where each ‘#’ should be printed for the piece to form (using the screen reference that is passed in) and then uses printChar to print ‘#’ at that spot on the screen. This function is virtual because each piece has a different layout, so to avoid one preposterously large base class function that checks for the Piece Type, I broke it down by class. That explains why it is virtual, as well as why it is in each derived class. But more to that point, it must be part of the general Piece class because it deals with the display of a game Piece, and as aforementioned, each derived class must have its own specialized version for the sake of organization and efficiency. Finally, the function is const due to the fact that it does not modify any member variables, it purely prints to the screen.

**I\_Piece : public Piece**

I\_Piece():

This constructor calls the setPieceType function to initialize its type to PIECE\_I. Because it is a constructor, it is not marked virtual, and it is definitely not const.

virtual ~I\_Piece():

The destructor has no body.

**L\_Piece : public Piece**

L\_Piece():

This constructor calls the setPieceType function to initialize its type to PIECE\_L. Because it is a constructor, it is not marked virtual, and it is definitely not const.

virtual ~L\_Piece():

The destructor has no body.

**J\_Piece : public Piece**

J\_Piece():

This constructor calls the setPieceType function to initialize its type to PIECE\_J. Because it is a constructor, it is not marked virtual, and it is definitely not const.

virtual ~J\_Piece():

The destructor has no body

**T\_Piece : public Piece**

T\_Piece():

This constructor calls the setPieceType function to initialize its type to PIECE\_T. Because it is a constructor, it is not marked virtual, and it is definitely not const.

virtual ~T\_Piece():

The destructor has no body

**O\_Piece : public Piece**

O\_Piece():

This constructor calls the setPieceType function to initialize its type to PIECE\_O. Because it is a constructor, it is not marked virtual, and it is definitely not const.

virtual ~O\_Piece():

The destructor has no body.

**S\_Piece : public Piece**

S\_Piece():

This constructor calls the setPieceType function to initialize its type to PIECE\_S. Because it is a constructor, it is not marked virtual, and it is definitely not const.

virtual ~S\_Piece():

The destructor has no body.

**Z\_Piece : public Piece**

Z\_Piece():

This constructor calls the setPieceType function to initialize its type to PIECE\_Z. Because it is a constructor, it is not marked virtual, and it is definitely not const.

virtual ~Z\_Piece():

The destructor has no body.

**VAPORBOMB : public Piece**

VAPORBOMB():

This constructor calls the setPieceType function to initialize its type to PIECE\_VAPOR. Because it is a constructor, it is not marked virtual, and it is definitely not const.

virtual ~VAPORBOMB():

The destructor has no body.

**FOAMBOMB : public Piece**

FOAMBOMB():

This constructor calls the setPieceType function to initialize its type to PIECE\_FOAM. Because it is a constructor, it is not marked virtual, and it is definitely not const.

virtual ~FOAMBOMB():

The destructor has no body.

**CRAZYSHAPE : public Piece**

CRAZYSHAPE():

This constructor calls the setPieceType function to initialize its type to PIECE\_CRAZY. Because it is a constructor, it is not marked virtual, and it is definitely not const.

virtual ~CRAZYSHAPE():

The destructor has no body.

virtual void moveLeft():

This function is virtual because this Piece type behaves differently when the user tries to move left: it moves right. Thus, the implementation for Crazy Shape’s moveLeft is different in that it increments the xCoord by 1, instead of decrementing like the base class’s function. Because it modifies the xCoord data member, it cannot be marked const. It is in CRAZYSHAPE because it is something the CRAZYSHAPE does, and while it cannot access the xCoord private member variable directly, it still uses the public setter functions provided in the base Piece class.

virtual void moveRight():

This function behaves just like moveLeft(), except that it decrements xCoord by 1 when the user tries to go right, because that again is the effect of this Piece Type. Because it modifies the xCoord data member, it cannot be marked const. It is in CRAZYSHAPE because like moveLeft(), it is something the CRAZYSHAPE does, and while it cannot access the xCoord private member variable directly, it still uses the public setter functions provided in the base Piece class.

**Unfinished Functionalities/Issues:**

One issue that I have currently found in my program is that when running on the windows console, when the tick is done, the moving piece/cursor can appear rather glitchy. However, this problem is almost nonexistent when running on the linux server.

**Design Decisions and Implementations:**

While the spec itself was very specific, I found it very challenging as to how I should communicate the collision between a falling piece and the well/already fallen pieces. I first thought it best to represent each piece with a private member array in each derived Piece class, but that became a challenge accessing outside the class. Thus, I decided to make my Pieces not actual parts of the well grid until they had collided. This left me with the issue I mentioned: how do I detect these collisions, when each piece has a different format/appearance and the x and y coordinates only refer to the top left of the bounding box? I made the tough decision to create another set of x and y coordinate functions in my base Piece class, and these would be hardcoded with a location relative to the bounding box’s x and y coordinate. Therefore, when the time came to check if a part of the well was taken by ‘$’s (ie. right before the time elapsed is over), I just had to call a function that took in the bounding box’s x and y coordinate, each added to either the extra x coordinate or y coordinate + some int. This way, I was checking if the cell just below, to the right, or to the left was occupied:

This is xCoord4(), and to check to its right, I check (getxCoord() + xCoord4() + 1), so I’m checking this:

