CS32 Project 3 Report

Benjamin He

UID: 804962948

**Member Function Descriptions**

**Well.h/Well.cpp:**

NOTE: None of these functions are virtual because I don’t have any derived classes of the Well class.

Well();

This is the constructor of my Well class, which initializes the m\_well member array to be an empty 10x18 space surrounded by @ signs to represent the Well walls. This is in the Well class because it’s the constructor.

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

This function displays the entire Well with whatever contents it holds (settled pieces, foam, and the current moving piece) with the top left corner of the Well in the location specified by the x and y coordinates passed into the function. I put this in the Well class because it was easiest to access the contents of the m\_well private member 2D array, which holds what’s the data for what’s in the Well at any given point. Also, I felt like displaying what was in the Well should be delegated to the Well class.

void dispPiece(Screen& screen, int x, int y);

This function displays only the bounding box of the piece to update the display when the Piece moves because it takes too long to display the entire Well each time the Piece changes (it has a lag when I try to display the whole Well each time). This function knows where to find the bounding box of the current Piece because the top left corner of the Piece’s bounding box is passed into the x and y parameters in the function. I put this in the Well class again because it’s easiest to access the m\_well contents and because it makes sense to me for the Well to be displaying what’s in it.

void clearWell();

This function clears the Well after each level by setting everything contained within the Well to an empty character. This is in the Well class because this only affects the Well, it makes it easier to access the Well’s m\_well contents, and I feel like there’s no reason to put it in any other class.

int clearRows();

This function checks for and removes all the completed rows every time a Piece settles in the Well by looping through the entire Well. It also returns the number of rows that were cleared by the function so my program can keep track of how many rows are left until the level was completed. This is in the Well class again because it only affects the Well, and it’s easier for this function to access the Well’s m\_well contents and modify them.

char wellVal(int row, int col) const;

This function returns the value of the element in the Well indicated by the row and col (column) numbers passed into the function. This is the getter function for the Well’s m\_well. I made this function because I needed to access the contents of the m\_well in some of my Piece class functions. It’s in the Well class because there’s no other place it could be and still maintain its purpose.

void setVal(char val, int row, int col);

This function is the setter function that goes hand-in hand with the getter function wellVal where it allows non-member functions to modify the values in the Well’s m\_well member array. I created this because some of the Piece class functions needed to modify the Well’s m\_well. This function’s in the Well class because there’s no other place it can be.

void imprint();

This function “imprints” a settled Piece into the well by changing the settled Piece in the Well from ‘#’ signs to ‘$’ signs, making the settled Piece part of the Well in a sense so that the next Piece can appear and treat the previous settled Pieces as if they’re part of the Well. This function’s in the Well class because it only applies to the Well and it makes sense for the Well to be the one that modifies itself.

**Piece.h/Piece.cpp:**

**class Piece**

Piece(Well& pwell, char id);

This is the constructor of the Piece base class so it just initializes all the values of a Piece, setting the m\_orientation to 0, m\_well pointer to the Well that the Piece is going to be in (which is passed in as a parameter), the m\_x and m\_y to the coordinates in the well where the Piece’s bounding box’s top left corner is located, and the m\_id to specific Piece’s ID (which is also passed in as a parameter). It’s in the Piece base class because it’s the Piece’s constructor. It’s not virtual again because it’s just a constructor for the class.

virtual ~Piece()

This virtual destructor is necessary because I create instances of this Piece class in dynamic heap memory. This function doesn’t actually execute anything. It’s virtual because I use polymorphism with this Piece.

void statusBlock(Screen& screen) const;

This function displays the next Piece that is going to appear in the Well after the current one is settled on the right side under the “Next Piece: ” text in the status on the Screen that is passed into the function. It’s in the Piece base class, and it’s not virtual because all the derived classes of Piece are displayed in the exact same way and there’s no need to override the function within the derived classes.

void makeBlock(int x, int y);

This function puts whatever Piece is in the Piece base class’s m\_block member 2D array into the Well that is pointed to by the m\_well member pointer at the specified x and y location (which are passed into the function). This uses the getter and setter functions in the Well class to modify the Well’s m\_well array. This function DOES NOT actually display the Piece. The dispPiece/display function in the Well class does that. I put this function in the Piece class rather than the Well class because I felt like it fit better because this function makes the Piece in the Well the Piece belongs to. It’s not virtual again because the way the Piece is put into the Well is the same regardless of what kind if Piece it is.

void clearBlock(int x, int y);

This function clears a 4x4 block where the top left corner of the block is at the x and y coordinates passed into the function inside the Piece’s Well. I use this with the makeBlock function to simulate a Piece’s movement by clearing the Piece then remaking it one row lower each time the Piece “moves.” This function also DOES NOT display the cleared area—that job is still left to the dispPiece/display function in the Well class. I put this function in the Piece also because it made sense to me that any Piece-related modifications should be in the Piece class even though it actually modifies the Well that the Piece belongs to. This function is not virtual because every Piece is cleared the same way.

bool conflict(int x, int y) const;

This function detects whether there will be an overlap between a Piece and some component of the Well (either a settled Piece, foam, or the Well’s wall) if the Piece were to be constructed with its bounding block’s top left corner being at the x/y location passed into the function. The function returns true if there is such an overlap, otherwise it returns false. I put this in the Piece class for the same reasons as the above two functions. This function’s also not virtual because every Piece is tested for overlap in the same way.

virtual void specialMove();

This function executes the special ability of a certain Piece if it has one after it settles. The base class implementation of this function does nothing because most of the Pieces don’t have any special moves after they settle, but the VaporBomb does, so this Piece overrides this function in its derived class (which is why this function is virtual but not pure virtual). This function is in the Piece class because this function is unique to objects of the Piece class, and it needs to be overridden by the Piece\_VAPOR derived class.

virtual void specialMove(int sr, int sc, int minR, int maxR, int minC, int maxC);

This function executes the special ability of a certain Piece if it has one after it settles with some given parameters. The base class implementation of this function does nothing because most of the Pieces don’t have any special moves after they settle, but the FoamBomb does, so this Piece overrides this function in its derived class (which is why this function is virtual but not pure virtual). I made this second specialMove function because this one needed to use recursion for the FoamBomb, and I would not be able to use recursion if there were no parameters passed into the function. This function is in the Piece class because this function is unique to objects of the Piece class, and it needs to be overridden by the Piece\_FOAM derived class.

virtual void rotatePiece() = 0;

This function just rotates the Piece as specified in the spec and updates the Piece’s orientation accordingly. Within this function’s implementations, this function will only rotate the Piece if rotating the Piece will not result in an overlap with something already in the Well. This function is pure virtual because every derived class can be rotated, but they all have their own variations in how they are rotated.

void setBlock(

char v00, char v10, char v20, char v30,

char v01, char v11, char v21, char v31,

char v02, char v12, char v22, char v32,

char v03, char v13, char v23, char v33

);

This function sets the m\_block member array in the Piece base class to have the values that are passed into the functions’s parameters. I made this function to allow the derived classes of Piece to modify the private m\_block member array because it’s the only way for them to access m\_block. This function is in the Piece base class because it is the Piece’s setter for its m\_block data member, so there’s no other place to have it. It’s not virtual because the way the m\_block is set does not vary between the different derived Piece classes.

void setOrient(int orientation);

This function is the setter for the Piece base class’s m\_orientation member variable so that the derived classes of Piece have a way to modify this private member. This function is in the Piece base class because its purpose is to provide access to a private member of Piece, so there’s no other place to put it. It’s not virtual because it won’t be overridden by any derived Piece classes.

int getOrient() const;

This function is the getter for the Piece base class’s m\_orientation member variable so that the derived classes of Piece have a way to see the value of this private member. It returns the value of m\_orientation. This function is in the Piece base class because its purpose is to provide access to a private member of Piece, so there’s no other place to put it. It’s not virtual because it won’t be overridden by any derived Piece classes.

int getBaseX() const;

This function is the getter for the Piece base class’s m\_x member variable so that the derived classes of Piece have a way to see the value of this private member. It returns the value of m\_x. This function is in the Piece base class because its purpose is to provide access to a private member of Piece, so there’s no other place to put it. It’s not virtual because it won’t be overridden by any derived Piece classes.

int getBaseY() const;

This function is the getter for the Piece base class’s m\_y member variable so that the derived classes of Piece have a way to view this private member. It returns the value of m\_y. This function is in the Piece base class because its purpose is to provide access to a private member of Piece, so there’s no other place to put it. It’s not virtual because it won’t be overridden by any derived Piece classes.

Well\* getWell() const;

This function is the getter for the Piece base class’s m\_well member variable so that the derived classes of Piece have a way to get the value of this private member. It returns the address of m\_well. This function is in the Piece base class because its purpose is to provide access to a private member of Piece, so there’s no other place to put it. It’s not virtual because it won’t be overridden by any derived Piece classes.

char getID() const;

This function is the getter for the Piece base class’s m\_id member variable so that the derived classes of Piece have a way to see this private member. It returns the address of m\_id. This function is in the Piece base class because its purpose is to provide access to a private member of Piece, so there’s no other place to put it. It’s not virtual because it won’t be overridden by any derived Piece classes.

void setID(char id);

This function is the setter for the Piece base class’s m\_id member variable so that the derived classes of Piece have a way to modify this private member. This function is in the Piece base class because its purpose is to provide access to a private member of Piece, so there’s no other place to put it. It’s not virtual because it won’t be overridden by any derived Piece classes.

**class Piece\_I : public Piece**

Piece\_I(Well& pwell);

This function is the constructor of the Piece\_I derived class. It calls the Piece base class constructor and passes in the well that’s passed into this constructor as well as ‘I’ for the ID (the Piece base class’s second parameter) to differentiate this specific kind of Piece from others. It also sets the m\_block of the Piece to the I-Piece’s Orientation #0 as specified in the project spec using the setBlock function. This function’s in the Piece\_I class because it’s this class’s constructor, which is also why it’s not virtual.

~Piece\_I()

This destructor is necessary because I (potentially) dynamically allocate an object of Piece\_I in my pickPiece function, so this is there to prevent any undefined behavior from occurring when the Piece\_I is destructing. Nothing happens in the statement block. It’s not virtual because there are no derived classes from this class.

virtual void rotatePiece();

This function changes the orientation of the Piece to the next orientation (which it does by setting the m\_block of the Piece to the layout of that next orientation by using the setBlock function) while also updating the value of the Piece’s m\_orientation variable using the setOrient function. The reason this function is implemented in each derived class is because the layouts of the orientations and how they rotate is different for each class, so this function is just overridden in each derived class to have the specific orientations of that derived class. This function is virtual because of syntax since it implements a pure virtual function in the base class.

**class Piece\_L : public Piece**

Piece\_L(Well& pwell);

This function is the constructor of the Piece\_L derived class. It calls the Piece base class constructor and passes in the Well that’s passed into this constructor as well as ‘L’ for the ID (the Piece base class’s second parameter) to differentiate this specific kind of Piece from others. It also sets the m\_block of the Piece to the L-Piece’s Orientation #0 as specified in the project spec using the setBlock function. This function’s in the Piece\_L class because it’s this class’s constructor, which is also why it’s not virtual.

~Piece\_L()

(Same reasoning for everything as the destructor in Piece\_I class.)

virtual void rotatePiece();

(See the description for Piece\_I’s rotatePiece function. These functions are all implemented basically identically in each derived class except the specific orientations/rotations are unique for each piece and are set accordingly.)

**class Piece\_J : public Piece**

Piece\_J(Well& pwell);

This function is the constructor of the Piece\_J derived class. It calls the Piece base class constructor and passes in the Well that’s passed into this constructor as well as ‘J’ for the ID (the Piece base class’s second parameter) to differentiate this specific kind of Piece from others. It also sets the m\_block of the Piece to the J-Piece’s Orientation #0 as specified in the project spec using the setBlock function. This function’s in the Piece\_J class because it’s this class’s constructor, which is also why it’s not virtual.

~Piece\_J()

(Same reasoning for everything as the destructor in Piece\_I class.)

virtual void rotatePiece();

(See the description for Piece\_I’s rotatePiece function. These functions are all implemented basically identically in each derived class except the specific orientations/rotations are unique for each piece and are set accordingly.)

**class Piece\_T : public Piece**

Piece\_T(Well& pwell);

This function is the constructor of the Piece\_T derived class. It calls the Piece base class constructor and passes in the Well that’s passed into this constructor as well as ‘T’ for the ID (the Piece base class’s second parameter) to differentiate this specific kind of Piece from others. It also sets the m\_block of the Piece to the T-Piece’s Orientation #0 as specified in the project spec using the setBlock function. This function’s in the Piece\_T class because it’s this class’s constructor, which is also why it’s not virtual.

~Piece\_T()

(Same reasoning for everything as the destructor in Piece\_I class.)

virtual void rotatePiece();

(See the description for Piece\_I’s rotatePiece function. These functions are all implemented basically identically in each derived class except the specific orientations/rotations are unique for each piece and are set accordingly.)

**class Piece\_O : public Piece**

Piece\_O(Well& pwell);

This function is the constructor of the Piece\_O derived class. It calls the Piece base class constructor and passes in the Well that’s passed into this constructor as well as ‘O’ for the ID (the Piece base class’s second parameter) to differentiate this specific kind of Piece from others. It also sets the m\_block of the Piece to the O-Piece’s Orientation #0 as specified in the project spec using the setBlock function. This function’s in the Piece\_O class because it’s this class’s constructor, which is also why it’s not virtual.

~Piece\_O()

(Same reasoning for everything as the destructor in Piece\_I class.)

virtual void rotatePiece();

(See the description for Piece\_I’s rotatePiece function. These functions are all implemented basically identically in each derived class except the specific orientations/rotations are unique for each piece and are set accordingly.)

**class Piece\_S : public Piece**

Piece\_S(Well& pwell);

This function is the constructor of the Piece\_S derived class. It calls the Piece base class constructor and passes in the Well that’s passed into this constructor as well as ‘S’ for the ID (the Piece base class’s second parameter) to differentiate this specific kind of Piece from others. It also sets the m\_block of the Piece to the S-Piece’s Orientation #0 as specified in the project spec using the setBlock function. This function’s in the Piece\_S class because it’s this class’s constructor, which is also why it’s not virtual.

~Piece\_S()

(Same reasoning for everything as the destructor in Piece\_I class.)

virtual void rotatePiece();

(See the description for Piece\_I’s rotatePiece function. These functions are all implemented basically identically in each derived class except the specific orientations/rotations are unique for each piece and are set accordingly.)

**class Piece\_Z : public Piece**

Piece\_Z(Well& pwell);

This function is the constructor of the Piece\_Z derived class. It calls the Piece base class constructor and passes in the Well that’s passed into this constructor as well as ‘Z’ for the ID (the Piece base class’s second parameter) to differentiate this specific kind of Piece from others. It also sets the m\_block of the Piece to the Z-Piece’s Orientation #0 as specified in the project spec using the setBlock function. This function’s in the Piece\_Z class because it’s this class’s constructor, which is also why it’s not virtual.

~Piece\_Z()

(Same reasoning for everything as the destructor in Piece\_I class.)

virtual void rotatePiece();

(See the description for Piece\_I’s rotatePiece function. These functions are all implemented basically identically in each derived class except the specific orientations/rotations are unique for each piece and are set accordingly.)

**class Piece\_VAPOR : public Piece**

Piece\_VAPOR(Well& pwell);

This function is the constructor of the Piece\_VAPOR derived class. It calls the Piece base class constructor and passes in the Well that’s passed into this constructor as well as ‘V’ for the ID (the Piece base class’s second parameter) to differentiate this specific kind of Piece from others. It also sets the m\_block of the Piece to the VaporBomb’s Orientation #0 as specified in the project spec using the setBlock function. This function’s in the Piece\_VAPOR class because it’s this class’s constructor, which is also why it’s not virtual.

~Piece\_VAPOR()

(Same reasoning for everything as the destructor in Piece\_I class.)

virtual void rotatePiece();

(See the description for Piece\_I’s rotatePiece function. These functions are all implemented basically identically in each derived class except the specific orientations/rotations are unique for each piece and are set accordingly.)

virtual void specialMove();

This function is specifically overridden in the Piece\_VAPOR derived class and it just clears four spaces directly above, below, and it clears the piece itself by setting all those spaces to the ‘ ‘ character, unless any of the four spaces below the piece passes/overlaps with the bottom of the Well, in which case it doesn’t clear those spaces. This function is overridden in this derived class because of the VaporBomb’s special move, and it’s virtual because of syntax since it overrides a virtual function in the base class.

**class Piece\_FOAM : public Piece**

Piece\_FOAM(Well& pwell);

This function is the constructor of the Piece\_FOAM derived class. It calls the Piece base class constructor and passes in the Well that’s passed into this constructor as well as ‘F’ for the ID (the Piece base class’s second parameter) to differentiate this specific kind of Piece from others. It also sets the m\_block of the Piece to the Foambomb’s Orientation #0 as specified in the project spec using the setBlock function. This function’s in the Piece\_FOAM class because it’s this class’s constructor, which is also why it’s not virtual.

~Piece\_FOAM()

(Same reasoning for everything as the destructor in Piece\_I class.)

virtual void rotatePiece();

(See the description for Piece\_I’s rotatePiece function. These functions are all implemented basically identically in each derived class except the specific orientations/rotations are unique for each piece and are set accordingly.)

virtual void specialMove(int sr, int sc, int minR, int maxR, int minC, int maxC);

This function is specifically overridden in the Piece\_FOAM derived class and it uses recursion to search through and set any open space that can be reached starting from the point where the FoamBomb settles in the Well to the ‘\*’ character within a 5x5 box around that starting point. It does so by setting the current space that is specified by the sr and sc (row and column) values to ‘\*’ then by looking in four directions above and below this starting point and if any of those directions are also open, then this function is called again on those open spaces if the spaces’s row and column locations don’t exceed the dimensions of 5x5 box of the very first starting point, which are given by the minR, maxR, minC, and maxC parameters. The reason why there’s this second specialMove function that has parameters is because I wouldn’t be able to use recursion if there are no parameters to pass into the function like in the other specialMove function that has no parameters. This function is virtual because of syntax since it’s overriding a virtual function in the base Piece class.

**class Piece\_CRAZY : public Piece**

Piece\_CRAZY(Well& pwell);

This function is the constructor of the Piece\_CRAZY derived class. It calls the Piece base class constructor and passes in the Well that’s passed into this constructor as well as ‘C’ for the ID (the Piece base class’s second parameter) to differentiate this specific kind of Piece from others. It also sets the m\_block of the Piece to the CrazyPiece’s Orientation #0 as specified in the project spec using the setBlock function. This function’s in the Piece\_CRAZY class because it’s this class’s constructor, which is also why it’s not virtual.

~Piece\_CRAZY()

(Same reasoning for everything as the destructor in Piece\_I class.)

virtual void rotatePiece();

(See the description for Piece\_I’s rotatePiece function. These functions are all implemented basically identically in each derived class except the specific orientations/rotations are unique for each piece and are set accordingly.)

**Game.h/Game.cpp:**

NOTE: There are no virtual functions in this class because there are no derived classes from this class.

Game(int width, int height);

This function is the constructor for the Game, but this was already provided in the skeleton code so I didn’t do anything else with it. It’s in this class because it’s this class’s constructor.

void play();

This function was also already provided in the skeleton code and I did not change it at all. It’s in this class because it manages what happens when the player plays a Game.

bool playOneLevel();

This function controls all the logic that occurs in each level when a Game is played. First, the number of rows that need to be cleared is calculated based on what level the Game is, then the Well is displayed using the Well class’s display function and the first Piece is created using Game’s pickPiece member function, then the code enters a while loop that repeats itself as long as the number of rows cleared have not yet met the required amount for the level. Within this while loop, the current Piece is continually displayed with the Well class’s dispPiece function, cleared with the Piece class’s clearBlock function, rebuilt on a lower row with the Piece class’s makeBlock function, then displayed again with the Well class’s dispPiece function to simulate the movement of the piece. However, before moving the piece down each time, there’s another while loop that checks for user input and modifies the Piece accordingly based on the key the user hits. If the user hits ‘q’ or ‘Q’, the game quits by returning false. If the user hits ‘ ‘, the current Piece is automatically dropped to its settled position. If the user taps the left or right arrow, the Piece shifts one space in that direction using the shiftRight or shiftLeft Game member functions depending on the direction of the arrow key hit as well as whether the current Piece is a CrazyPiece, which can be determined by calling the Piece’s getID function. If the user taps the up arrow, the Piece rotates to the next orientation. Finally, if the user taps the down arrow, the Piece is immediately shifted down a row and the timer for making moves is reset. When the Piece finally settles in its position, the Well’s clearRows function checks for and clears any completed rows in the Well, the number of rows left is subtracted from the rows cleared, and the Well is redisplayed. If a new Piece overlaps with something in the Well the moment the Piece is put in the Well by the makeBlock function, the playOneLevel function returns false and the game ends, however, if the number of rows left (m\_rLeft) reaches zero before this happens, the function return true and the m\_level of the Game is incremented and this function is called again in the Game’s play function. This function is in the Game class because it is exclusively used by the Game and is called by the Game’s play function.

void displayPrompt(std::string s);

This function was already provided in the skeleton code given and I made no changes to it. It’s in the Game class because it is used in the play function, so there’s no reason why it should be a function outside of this class.

void displayStatus();

This function displays the statistics of the current Game being played on the side of the screen next to the Well. Those statistics are: the number of rows left (m\_rLeft), the current level (m\_level), and the current score (m\_score). This function’s in this Game class for the same reasons as the two functions above.

void pieceRotate(Piece\* currPiece, int& x, int& y);

This function calls the currPiece’s rotatePiece() function and redisplays the new orientation for the user to see at the x and y location passed into the function. This function’s in the Game class because it’s only used by the playOneLevel function that’s in the Game class, so there’s no reason to put it anywhere else.

void shiftRight(Piece\* currPiece, int& x, int& y);

This function shifts the Piece one space to the right by clearing the currPiece at the location specified by the x and y location passed into this function using the Piece’s clearBlock function, then increasing x by 1 and rebuilding the Piece with its makeBlock function and redisplaying the Piece for the user to see. This function’s in the Game class because it’s only used by the playOneLevel function that’s in the Game class, so there’s no reason to put it anywhere else.

void shiftLeft(Piece\* currPiece, int& x, int& y);

This function shifts the Piece one space to the left by clearing the currPiece at the location specified by the x and y location passed into this function using the Piece’s clearBlock function, then decreasing x by 1 and rebuilding the Piece with its makeBlock function and redisplaying the Piece for the user to see. This function’s in the Game class because it’s only used by the playOneLevel function that’s in the Game class, so there’s no reason to put it anywhere else.

Piece\* pickPiece(Well& well);

This function creates a new randomly chosen Piece in the heap (because otherwise the Piece would destruct once the function finished) and the address of that Piece is returned so that it can be used in the playOneLevel Game function. This function’s in the Game class because it’s only used by the playOneLevel function that’s in the Game class to pick the next Piece, so there’s no reason to put it anywhere else.

**Failed Functionality**

As far as I know, there are no bugs in my classes, and all the functionality that is supposed to be in the complete game is present. I ran and played my game numerous times to verify this, though there may still be errors that I did not catch/notice.

**Other Design Decisions and Assumptions**

* The provided enum PieceType in the has an additional enum called NUM\_PIECE\_TYPES which I believe has to do with the chooseRandomPieceType function’s implementation, so I didn’t write a specific case where the chooseRandomPieceType would have chosen the NUM\_PIECE\_TYPES enum. Instead, my pickPiece function would default to creating an I-Piece if the randomly chosen enum doesn’t match with any of the other PieceType enums.