Students:
Section 14.7 is a part of 1 assignment:
Programming Project 07 - Outlast the Baddies & Avoid the Abyss

Includes: 🟩 zyLab

Due: 12/05/2024, 11:59 PM CST

Instructor created

14.7 P07 - Outlast the Baddies & Avoid the Abyss

Game Boards

Games are commonly played on two-dimensional, rectangular boards, composed of a regular, rectangular grid of cells. At any given instant, each cell of the game board may be occupied in a variety of different ways, which determines how that cell behaves as the round is played out.

In some basic games, such as Tic-Tac-Toe or Connect-Four, the cells are marked with a player's symbol and can never be changed. In that case, the cell behavior never varies throughout the gameplay (instead, only the characteristics of a cell may change, not their behaviors). However, more complicated games, such as Chess or Pac-Man, involve pieces or characters that behave differently and move throughout the board as the game is played. In these situations, defining the game board as a two-dimensional grid of polymorphic objects is advantageous.

Figure 14.7.1: Tic-Tac-Toe game board and sample game play

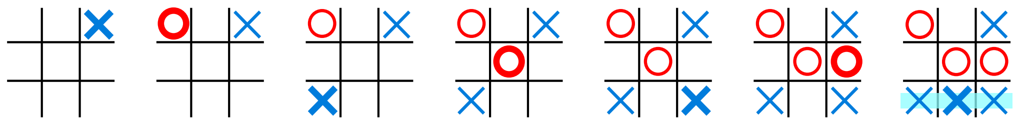


Figure 14.7.2: Connect-Four game board and sample game play

Figure 14.7.3: Pac-Man game board and sample game play

Figure 14.7.4: Chess game board and sample game play

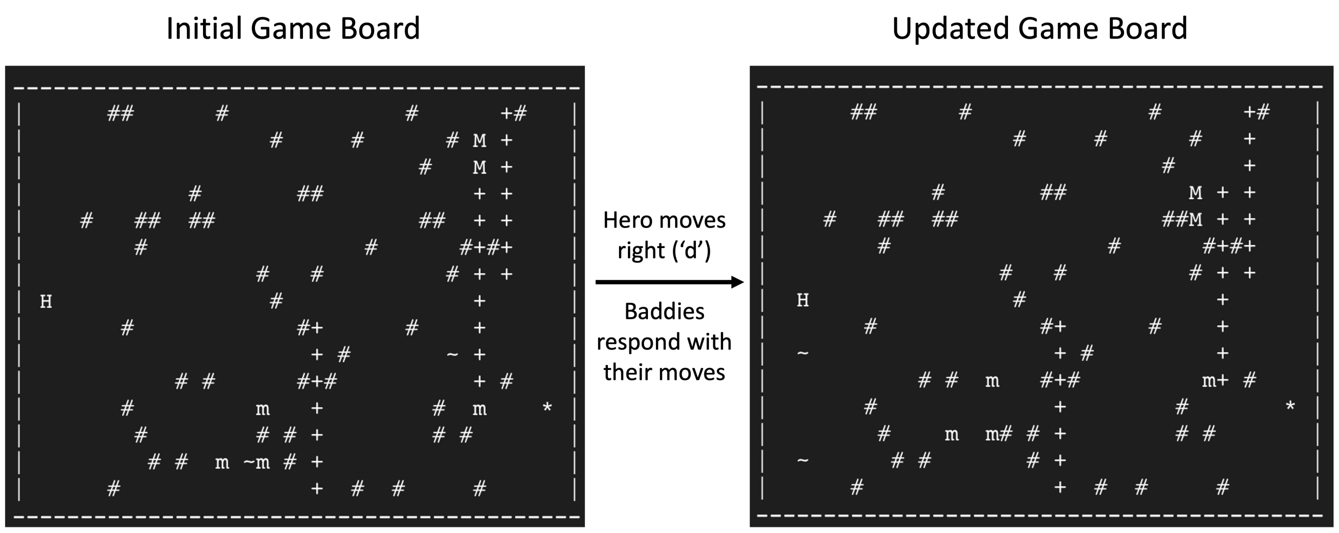
The Game - Outlasting the Baddies & Avoiding the Abyss

Our game is played on an arbitrarily sized rectangular grid, where each cell can contain a variety of different occupants:

- The user plays as the **Hero (H)**, who is trying to exit the board by navigating to
- the **Escape Ladder (*)**, but there are many obstacles in the way:
- **Wall (+)** cells are barriers that can not be landed on,
- **Abyss (#)** cells are holes in the floor that lead to the demise of any object that lands on it,
- **Regular Monsters (m)** continually seek to capture the hero, one step at a time either/both vertically or/and horizontally,
- **Super Monsters (M)** continually seek to capture the hero, but must take two steps at a time either/both vertically or/and horizontally, and
- **Bats (~)** continually seek to occupy the hero's column, but cannot change rows.

Here is a randomly generated 15x40 (i.e. 15 rows, 40 columns) sample game board, with 3 vertical walls, 50 abyss cells, 4 regular monsters, 2 super monsters, and 2 bats:

Figure 14.7.5: Sample game board for one round of play



```
------------------------------------------
|      ##      #             #      +#   |
|                  #     #      # M +    |
|                             #   M +    |
|            #       ##           + +    |
|    #   ##  ##               ##  + +    |
|        #                #      #+#+    |
|                 #   #         # + +    |
| H                #              +      |
|       #            #+      #    +      |
|                     + #       ~ +      |
|           # #      #+#          + #    |
|       #         m   +        #  m    * |
|        #        # # +        # #       |
|         # #  m ~m # +                  |
|      #              +  #  #     #      |
------------------------------------------
```

```
------------------------------------------
|      ##      #             #      +#   |
|                  #     #      #   +    |
|                             #     +    |
|            #       ##         M + +    |
|    #   ##  ##               ##M + +    |
|        #                #      #+#+    |
|                 #   #         # + +    |
|  H               #              +      |
|       #            #+      #    +      |
|  ~                  + #         +      |
|           # #  m   #+#         m+ #    |
|       #             +        #       * |
|        #    m  m# # +        # #       |
|  ~      # #       # +                  |
|      #              +  #  #     #      |
------------------------------------------
```

Once the initial board is set up, the cells occupied by Walls, Abysses, and the exit EscapeLadder will never change (i.e. they are static). However, the Hero and Baddies (regular Monsters, super Monsters, and Bats) will move throughout the board. **Note that a Baddie is defined as a villain or antagonist in a story or game.**

Whereas the Hero (i.e. the user) can see the entire board to build a strategy for navigating an escape path while avoiding Abyss cells and Baddies, a misstep by the Hero, where they move into an Abyss cell or a cell occupied by a Baddie, results in their demise with the Hero being removed from the board and the end of the game; this is a losing result. However, if the Hero reaches the EscapeLadder successfully, then they have escaped, which also results in the Hero being removed from the board and the end of the game; in this case, we have a winning result!

The Baddies, on the other hand, have a single focus of capturing the Hero. Each type of Baddie makes moves following their specific movement rules that only depend on the Hero's position, with no regard for avoiding Abyss cells:

- Regular Monsters compare their current position (row and column) on the board to the Hero's position (row and column) and take 1-step toward the Hero horiztonally and/or 1-step toward the Hero vertically.
- Super Monsters do the same comparison of their position to the Hero's position, but are much bigger than regular Monsters and, therefore, MUST take 2-steps toward the Hero either/both horiztonally or/and vertically. Super Monsters are unable to only take 1-step movements in either direction.
- Finally, Bats immediately fly to the Hero's column, but can never leave their initial row.

The Hero and all Baddies are not allowed to end their move on a Wall cell OR go beyond the edge of the board. Attempted out-of-bounds moves are dealt with by separately checking the horizontal and vertical components of the movement, and **ignoring** either OR both components that put the Hero or Baddie out-of-bounds. Attempted movements onto a Wall need a little bit more care since Wall cells can be anywhere: attempts to move onto a Wall are dealt with by completely ignoring the horizontal portion of the attempted move and only moving vertically, unless that is still a Wall, in which case the attempted move is completely ignored and they stay put for that round. Here is what that means for the various types of attempted movements:

- if the Hero or a Baddie attempts to move perfectly horiztonally to end on a Wall, then their attempted move is ignored and they stay put for that round;
- if the Hero or a Baddie attempts to move perfectly vertically to end on a Wall, then their attempted move is ignored and they stay put for that round.
- if the Hero or a Baddie attempts to move diagonally to end on a Wall, then the horizontal portion of their movement is ignored and their attempted move is updated with the vertical portion only; then, their attempted move is checked again for a Wall.

One special case is that the exit EscapeLadder acts a Wall cell for the Baddies, but it is the (ONLY) exit cell for the Hero.

Since the movement of the Baddies is predictable, a successful Hero's strategy is to move in such a way that will cause each Baddie to fall into an Abyss cell. This can be challenging and/or impossible depending on the random arrangement of the initial board (quantity and location of Abyss cells, quantity and location of Baddies, etc.).

Polymorphic Game Board Cells - Classes for the Game and the Game Application

The BoardCell abstract class and its polymorphic derived classes (boardcell.h)

The BoardCell abstract class is used to define all cell types that can exist on the game board. The class is already well-developed in boardcell.h with the following private data members:

```
        size_t myRow; // current row for this board cell in a 2D grid
        size_t myCol; // current column for this board cell in a 2D grid
        bool moved;   // true = this board cell already moved in the current round
```

There is also a plethora of member functions. First,

```
   virtual char display( ) = 0; // pure virtual function; this is an abstract base class
```

is a pure virtual function that makes BoardCell an abstract class. The derived classes require a display() function definition to return the character associated with that cell type. There are also many member functions that can be used to classify the various cell types. For example, isStatic() returns true by default, so the Wall, Abyss, and EscapeLadder subclasses do not need their own definitions for isStatic(); but, Hero, Monster, and Bat do have their own definitions for isStatic(), which simply returns false since these cell types are not static and move about the board. There are many other provided member functions. You are strongly encouraged to read through the class definitions in boardcell.h at this point to get a full understanding of the various classes and their inheritance/polymorphic relationships.

The derived classes are Hero (H), Monster (m or M), Bat (~), Wall (+), Abyss (#), EscapeLadder (*), and Nothing ( ). It is important to note that the distinction between a regular Monster (m) and a super Monster (M) is the data member **power**, where regular Monsters have power = 1 and super Monsters have power = 2. Conveniently, the value of **power** sets the step-sizes that the two forms of Monsters take whenever they move.

The only member function of BoardCell (and its derived classes) that needs development is attemptMoveTo(), which applies the movement rule for the specific cell type to produce a *proposed new position* for that object's *attempted next move*. Potential collisions and/or illegal moves are not handled at this stage. This is simply an application of the movement rule to propose the next move. For example, suppose a small board currently has the following configuration:

```
--------
|      |
| +    |
| +M   |
|H+m   |
--------
```

The Hero (H) is currently in row 3 and column 0, so it's position is (3,0). Interactive user-input is used to set the Hero's next move as follows: the user enters 's' to have the Hero "stay put" with no change to position. Then, the adjacent keys to 's' on a standard QWERTY keyboard are used to assign directionality, as follows:

```
      'q' = up and left       'w' = up        'e' = up and right
      'a' = left              's' = stay      'd' = right
      'z' = down and left     'x' = down      'c' = down and right
   interpret ANY other input as 's' = stay
```

If the Hero attempts any move except to stay put ('s') or vertically up ('w'), they will be attempting an illegal move. However, attemptMoveTo() should not fix the illegal attempted moves. Instead, here are some possible attempted moves and function results:

- attempted move is **stay** ('s')... attemptMoveTo position (3, 0)
- attempted move is **up** ('w')... attemptMoveTo position (2, 0)
- attempted move is **right** ('d')... attemptMoveTo position (3, 1)
- attempted move is **down-left** ('z')... attemptMoveTo position (4, -1)*

- attempted move is **up-right** ('e')... attemptMoveTo position (4, 1)

There are also two Baddies, namely a regular Monster (m) at (3, 2) and a super Monster (M) at (2, 2). Assuming the Hero stayed put for its move, here are the attempted moves for the Baddies:

- regular Monster (m) attemptMoveTo position (3, 1)
- super Monster (M) attemptMoveTo position (4, 0)

Both of these attempted moves are illegal, but we do not handle it inside the BoardCell derived classes.

Note that the BoardCell version of attemptMoveTo() should keep the position unchanged as a default, which works for all static cell types. Thus, the Wall, Abyss, and EscapeLadder derived classes do not need their own definitions for attemptMoveTo().

> **The passed-by-reference parameters (newR and newC) of attemptMoveTo for the new position have size_t type, which is an unsigned integer. Thus, they do not support negative values. However, the C++ standard automatically wraps negative values to the largest possible value for the size_t type, such that -1 is actually stored as 18446744073709551615, which will still be an illegal position for any reasonably sized game board.*

The GameBoard class and its BoardCell* Grid object (gameboard.h)

The GameBoard class defined in gameboard.h is used to construct and manage the game board throughout the gameplay. There are many private data members:

```
Grid<BoardCell*> board;
size_t numRows;
size_t numCols;
size_t HeroRow; // Hero's position row
size_t HeroCol; // Hero's position column
int numMonsters;
int numSuperMonsters;
int numAbysses;
int numBats;
bool wonGame; // false, unless the Hero reached the exit successfully
```

Most importantly, **board** is a Grid object of pointers to BoardCell objects, which is the game board that contains the polymorphic cell occupants described in the previous section. The Grid class used here is identical to grid.h that was developed in Project 06, and thus does not require further description (refer to the Project 06 description for details). Recall that the Grid class supports arbitrarily sized two-dimensional arrays. Thus, **numRows** and **numCols** represents the dimensionality of board. **HeroRow** and **HeroCol** represent the position of the Hero at any point during game play. When the Hero leaves the game board, HeroRow and HeroCol are both explicitly set to -1 to signify the absence of the Hero. However, due the type being **size_t**, the value of -1 is automatically wrapped to the highest possible value for the unsigned integer, which is actually what is stored in **HeroRow** and **HeroCol** when the Hero leaves the board. The next four data members, namely **numMonsters**, **numSuperMonsters**, **numAbysses**, and **numBats**, are used to set up the initial game board with a random arrangement of obstacles and Baddies. These parameters are set by the user in the game application (main.cpp). Finally, **wonGame** is a boolean that is set to false when the game board is constructed and should only be set to true if and when the Hero has reached the EscapeLadder and exited to victory.

There are many member functions fully developed and provided in the GameBoard class. You are strongly encouraged to explore gameboard.h at this point to get a full understanding of the various provided methods. The Programming Tasks for this project leave some flexibility for utilizing the provided methods or writing your own (whether in the GameBoard class or the BoardCell class and its derivatives).

Of particular importance is the incomplete makeMoves() member function, which is responsible for the operation of a single round of gameplay. This essentially boils down to four big steps:

1. the Hero proposes an attempted move to a new location,
2. the Hero makes a move after any collisions or illegal position exceptions are handled,
3. all Baddies propose an attempted move to a new location, and
4. all Baddies make a move after any collisions or illegal position exceptions are handled.

Further details for this process are provided in the starter code comments and in the Programming Tasks descriptions below.

The Game Application (main.cpp)

The game is played by running the primary application main.cpp. You are strongly encouraged to explore main.cpp at this point to get a full understanding of the overall steps of how the game is setup and run, which includes the following:

- **interactive user-input to set game parameters**
    - user-input to set the number of rows for the board
    - user-input to set the number of columns for the board
    - user-input to set the number of Abyss cells for the board
    - user-input to set the number of Monsters for the initial board (~1/3 of this value are super Monsters, the rest are regular Monsters)

    - user-input to set the number of Bats for the initial board
    - user-input to set a seed for the pseudorandom number generator (entering -1 will use system time as the seed, which has the effect of producing a *unique* board configuration for each run of the program); reusing the same seed is a good idea to help test and debug program components
- **initial game board set up**: all boards randomly set locations for the following:
    - the Hero in the left-most three columns,
    - the EscapeLadder in the right-most three columns,
    - three vertical walls that stretch over half of the board height somewhere between the Hero and EscapeLadder,
    - Abyss cells, regular Monsters, super Monsters, and Bats (quantity determined by user-input in the previous step) somewhere between the Hero and EscapeLadder
- **display the initial game board**
- **game play**: repeatedly do the following until makeMoves() returns false, signifying the Hero has left the board and the game is over:
    - get the next move for the Hero using interactive user input
    - call makeMoves(), which operates a single round of game play
    - display the updated game board at the end of the round
- **report the result of the game**: either *"Hero Escaped!"* or *"Hero did not escape..."*

Programming Tasks

Navigate to the IDE below to find the starter code. It is strongly recommended that you complete, independently test, and test against the autograder the following tasks in the order they are presented here, since later tasks rely on the full functionality of the previous tasks.

Task I - Grid class (grid.h) - 24 points

Open grid.h. Copy in your fully functioning Grid class from the Project 06 grid.h file. We will use it here in its identical form. If you do not have a fully functioning Grid class, then now is the time to get it fully functioning. Refer to the Project 06 description for details on the Grid class. The autograded test cases for the grid.h class in this project are the same as in Project 06, so you can develop and test in either IDE.

Task II - BoardCell attemptMoveTo( ) (boardcell.h) - 20 points

Open boardcell.h. Find the TODO statements, which all involve writing code for the various instances of the attemptMoveTo() function. Follow the descriptions above and in the code comments to implement the appropriate movement rule for each BoardCell derived class type. Remember: this function produces an *attempted* move; collisions and illegal attempted moves are NOT resolved here.

Task III - GameBoard setters/getter for Hero position (gameboard.h) - 10 points

Open gameboard.h. This is really just a warm-up to help prepare for the next task, which is where the complexity ramps up. Search for the TODO statements, which involve writing the following functions:

- **getHeroPosition()** - basic getter for two data members (utilizes passed-by-reference parameters to return both)
- **setHeroPosition()** - basic setter for two data members
- **findHero()** - an updater function that requires searching for the Hero on the board and updating its current position (with a call to the setter above???)
- **makeMoves()** - all operations required to complete a round of game play; this is broken up into the remaining tasks below

For now, write the first three functions. Check the code comments for more details. Only once they are fully tested for functionality should you move on to the next task.

Task IV - GameBoard makeMoves() for Hero movement ONLY (gameboard.h) - 16 points

If you made it to this task, all that remains is to write the makeMoves() function. However, there is a lot that this function should do. So, we are breaking it down into three big tasks (i.e. Tasks IV, V, and VI), each of which have multiple sub-tasks that can be implemented and tested separately, using independent testing, comparison to the solution executable, and submission to the autograder (only after you are convinced of the functionality from your other testing strategies).

For now, let's just focus on getting the Hero movement to work. That is, the Baddies will not move for now; we will worry about their movement in the next Task.

Recall, in main(), each round of game play begins with the user entering a character to represent the next move for the Hero. That character is passed into the makeMoves() function. Here are the steps to make the Hero move:

- Get the attempted move position for the Hero using the polymorphic BoardCell function attemptMoveTo()
- Move the Hero, resolving any potential collisions or illegal move attempts (try/catch statements should be used here; an example is provided in the starter code; you are required to use at least three additional try/catch statements throughout the code you develop), as follows:
    - if the attempted move puts the Hero out-of-bounds, then change the row and/or column to stay on the board
    - if the attempted move puts the Hero on a Wall cell, then change the row and/or column to stay off the Wall (recall: diagonal attempted moves that end on a Wall cell are resolved by first ignoring the horizontal component of the attempted move and checking for a valid move, only ignoring the vertical component of the attempted move if also necessary)
    - if the attempted move puts the Hero on the EscapeLadder, then remove the Hero from the board; the Hero escaped and this is a winning result!
    - if the attempted move puts the Hero on an Abyss cell, then remove the Hero from the board; the Hero did not escape and this is a losing result.
    - if the attempted move puts the Hero on a cell occupied by a Baddie, then remove the Hero from the board; the Hero did not escape and this is a losing result.
    - if the attempted move puts the Hero on an empty cell with Nothing, then the actual move to make is the attempted move

Note: the myRow and myCol data members for the Hero object (inherited from the BoardCell class) must be updated whenever a move is made. Additionally, the HeroRow and HeroCol data members for the GameBoard object must be updated whenever the Hero has moved, which can be done easily with a call to findHero().

Note: many of the moves made by the Hero involve constructing and/or destructing objects. Task VI is dedicated to carefully checking your memory management, specifically to make sure you free (i.e. delete) the memory for BoardCell derived class objects when you are done with them, e.g. the Hero has "left the board." If you are careful with your memory management now, then Task VI is simply a check that you have freed all memory.

The function should return true if the Hero is still on the board at the end of the round of game play. Otherwise, if the Hero is no longer on the board, then return false. Note, a return of false may be due to a winning result (i.e. Hero reached the EscapeLadder) or a losing results (e.g. Hero falls into Abyss, Hero moves into a cell occupied by a Monster, etc.)

Task V - GameBoard makeMoves() for Baddie movement ALSO (gameboard.h) - 24 points

With makeMoves() fully functional for Hero movement ONLY, it is now time to extend it to also move the Baddies, i.e. regular Monsters, super Monsters, and Bats. Here are the steps to make the Baddies move:

- Visit each cell, looping through each row, then each column within each row
    - if the cell is not a Baddie, then there is nothing to do
    - if the cell is a Baddie, but the Baddie has already moved during this round of game play, then there is nothing to do
    - if the cell is a Baddie, AND the Baddie has not moved during this round of game play, then...
- Get the attempted move position for the Baddie using the polymorphic BoardCell function attemptMoveTo()
- Move the Baddie, resolving any potential collisions or illegal move attempts (once again, try/catch statements should be used here; an example is provided in the starter code; you are required to use at least three additional try/catch statements throughout the code you develop), as follows:
    - if the attempted move puts the Baddie out-of-bounds, then change the row and/or column to stay on the board
    - if the attempted move puts the Baddie on a Wall cell or the EscapeLadder, then change the row and/or column to stay off the Wall (recall: diagonal attempted moves that end on a Barrier are resolved by first ignoring the horizontal component of the attempted move and checking for a valid move, only ignoring the vertical component of the attempted move if also necessary)
    - if the attempted move puts the Baddie on an Abyss cell, then remove the Baddie from the board.
    - if the attempted move puts the Baddie on a cell occupied by a Hero, then remove the Hero from the board; the Hero did not escape and this is a losing result; the actual move for the Baddie to make is the attempted move (i.e. the Baddie replaces the Hero)
    - if the attempted move puts the Baddie on a cell occupied by another Baddie, then ignore the attempted move and keep the Baddie where it was at the start of this round (potential collisions between two Baddies means that the gameplay may be affected by the order in which cells are visited; it is recommended that you loop through rows first, then loop through columns nested in the row loop to visit each cell)
    - if the attempted move puts the Baddie on an empty cell with Nothing, then the actual move to make is the attempted move

Note: all Baddies (just like the Hero) fall into the Abyss if their attempted move ends on an Abyss cell. So, even Bats are drawn into the Abyss if their attempted move ends there. However, some Baddies are able to travel over Abyss cells, as long as their attempted move does not END on an Abyss. This only applies, in practice, to super Monsters (MUST take 2-steps) and bats, since their step sizes can be more than 1, while regular Monsters and the

Hero are limited to steps of size 1 and will never jump over an Abyss.

Note: similar to jumping over Abyss cells, super Monsters and Bats can easily travel over Walls, as long as the END of their attempted move is not on a Wall cell.

Note: to prevent a single Baddie from making multiple moves during a single round of the game, whenever a Baddie has moved, it must be marked as "already moved" in some fashion. There are many ways to do this, but the suggested process is to use the **moved** data member of the BoardCell base class, which has provided setter and getter functions. The **moved** data members must be reset for all BoardCells at the beginning of each round.

Note: similar to the Hero, the myRow and myCol data members for Baddies (inherited from the BoardCell class) must be updated whenever a move is made.

Note: similar to the Hero, many of the moves made by Baddies involve constructing and/or destructing objects. Task VI is dedicated to carefully checking your memory management, specifically to make sure you free (i.e. delete) the memory for BoardCell derived class objects when you are done with them, e.g. a Monster has fallen into an Abyss. If you are careful with your memory management now, then Task VI is simply a check that you have freed all memory.

Task VI - Full gameplay functionality with no memory leaks or errors (main.cpp, gameboard.h, boardcell.h) - 6 points

The final task is to carefully check your memory management to ensure all allocated memory is freed whenever the program terminates. Note that the Grid class should have a fully functioning destructor AND the Gameboard class includes a provided destructor, which means that you do not need to explicitly free the board object, as the destructor will automatically get called when board goes out of scope. Instead, careful attention must be taken inside of makeMoves() to ensure appropriate memory is freed whenever a derived BoardCell object is no longer needed, e.g. when the Hero reaches the EscapeLadder, when a Bat is drawn into the Abyss, when a Monster captures the Hero, etc. Your regular testing strategy should include checking for memory leaks and errors using valgrind.

Playing the Game / Running the Program / Solution Executable

Instead of providing a bunch of sample game play screenshots, a fully functioning solution executable is provided for comparison as you develop code OR simply to enjoy playing the game before you get your version fully working. It is a surprisingly entertaining game for certain parameter sets; i.e. a balance between number of Baddies and number of Abyss cells must be sought. For example, the following inputs provide a nice medium sized game with a good balance where winning and/or losing are both feasible results:

```
numRows = 15
numCols = 40
numAbysses = 50
numMonsters = 6
numBats = 2
seed = -1
```

You can run the solution executable using **make run_solution.**

The makefile also includes targets for compiling (**make build**), running (**make run**), and valgrind-ing (**make valgrind**) your program.

Optional Extensions (Extra Credit)

There are endless ways to extend the game play to make the experience more entertaining for the user. The majority of these enhancements involve creating additional subtypes for the cells of the game board, i.e. additional derived classes of BoardCell that exhibit new and different behavior from the set already included in this project. Here are some ideas:

- A teleport, to either a specific location or a randomly chosen one. If there is something there, then a collision occurs, or if it is a barrier, then the player loses. (Can Baddies teleport too?)
- Some sort of treasure or award. May require a more advanced points system.
- Magic items that give the holder powers, such as two steps per turn, teleportation, collision resistance, enhanced vision, etc.
- A new form of Baddie with a different/enhanced sort of AI than the monsters that mindlessly pursue the player.
- Waldo, a hidden character that the player doesn't see, or who only shows his location occasionally and then hides again.
- "Foggy" cells, where the player does not get to see the whole board, but only parts that have already been visited, (and possibly not moving Baddies that have changed positions since the player last saw them.)

Another idea involves providing a means of playing again when the game is over, without rerunning the program. This could include increasing levels of difficulty or complexity.

Dynamically adjustable difficulty level would make the next game harder if the player won the previous game, or make the next game easier if the player lost. You could keep track of and report number of wins and losses, such that the difficulty could be a function of these values.

The possibilities are really limitless here. You are free to add to the boardcell.h, gameboard.h, and main.cpp files, without changing the core functionality of the game. For example, there must be a way to turn off the added subtypes so that the game can be played as designed before the extension was added. For example, if the user inputs 0 for the number of bats, then there are not bats in the game. A good idea is to add a command-line argument to main() such that the enhanced game is only run with a **-x** flag.

This component is **optional** and only for **extra credit**. To receive the extra credit you MUST include a file titled **extension.pdf** with your submission, which fully explains how you enhanced the game play and instructions for running the enhanced version.

Requirements

- Use the starter code as provided. Data members and member functions that are provided in the starter code are required for passing test cases. Thus, you cannot modify their names or definitions. However, you are free to add additional data members and/or member functions to any of the classes that you need to complete the tasks and/or as a part of any game enhancements.
- Solve each task and the program at large as intended. For example, polymorphism must be utilized as an integral part in making moves for the non-static cell types.
- The grid class (grid.h) is carried over from Project 06, so it must achieve all requirement laid out in the Project 06 requirements.
- The checking for and handling of illegal moves and collision exceptions must utilize at least three try/catch statements, in addition to the one provided in the starter code, in makeMoves().
- All dynamic heap-allocated memory must be freed to prevent possible memory leaks. This issue is checked by the autograder but may also receive a manually graded deduction.
- Coding style issues are manually graded using deductions, worth up to 25% of the total project score. Style points are graded for following the course standards and best practices laid out in the syllabus, including a header comments, meaningful identifier names, effective comments, code layout, functional decomposition, and appropriate data and control structures.

- Programming projects must be completed and submitted individually. Sharing of code between students in any fashion is not allowed. Use of any support outside of course-approved resources is not allowed, and is considered academic misconduct. Examples of what is allowed: referencing the zyBook, getting support from a TA, general discussions about concepts on piazza, asking questions during lecture, etc. Examples of what is NOT allowed: asking a friend or family member for help, using a "tutor" or posting/checking "tutoring" websites (e.g. Chegg), copy/pasting portions of the project description to an AI chatbot, etc. Check the syllabus for Academic Integrity policies. Violations of this requirement will receive a manually graded deduction, and may be reported to the Dean of Students office.

Tips for Success on this Programming Project

- **Entertaining Gameplay** - The end product of this project is a surprisingly entertaining game, despite its relative simplicity. A solution executable is provided so that you can play it right now!
- **NOT all tasks/functions are created equal** - As described above, the Programming Tasks for this project are diverse in their nature. There is a great deal of starter code provided. The level of scaffolding is designed to lead you through the tasks that leaves room for creatively applying course concepts and tools. A natural consequence is that..
    - *some Tasks are straightforward and some Tasks are challenging;*
    - *some Tasks ask you to do one thing specifically and some Tasks are purposefully open-end with multiple components;*
    - *some Tasks test your ability to follow instructions verbatim and some Tasks require you to practice problem-solving and critical-thinking skills;*
    - *some Tasks may only take you a few minutes and some Tasks may require an initial attempt followed by a break to do something else only to come back multiple times to tackle the challenge;*

Submission & Grading

Develop your code in the IDE below. The provided starter code is substantial. Thus, make sure to read all instructions carefully and only make changes to the specific files referenced in the project description.

Do NOT use the "Run" button; instead, compile your code using the make utility (e.g. "**make build**") and run the resulting executable (e.g. "**make run**") in the terminal to test your code interactively as you develop your program. There is also a valgrind target in the makefile (e.g. "**make valgrind**") to check for memory leaks or errors. Use the "Submit for grading" button to test your code against the suite of autograded test cases, which also submits your program for functionality grading.

When you are ready to formally submit, download the following files from the zyLab IDE file tree: **main.cpp, grid.h, GameBoard.h, BoardCell.h,** and **makefile**. Additionally, if you extended the game with enhanced gameplay for extra credit, then you will also upload **extension.pdf**, which should thoroughly describe your extension.

Then, **upload these files to the Project 07 Gradescope submission form**; look for *Project 07*. If you fail to submit to Gradescope, you have not formally submitted anything, and the resulting score is a zero for the project.

The standard deadline for the project is **Thursday, December, 5th**. As detailed in the syllabus, early submissions receive +1 extra credit point/day early, up to a maximum of +3 points. Also, students are allocated a certain number of late days throughout the term, where no late penalty is applied. Students can use their late days whenever they choose, simply by submitting their work to Gradescope after the deadline. However, no more than two late days are allowed for each project.

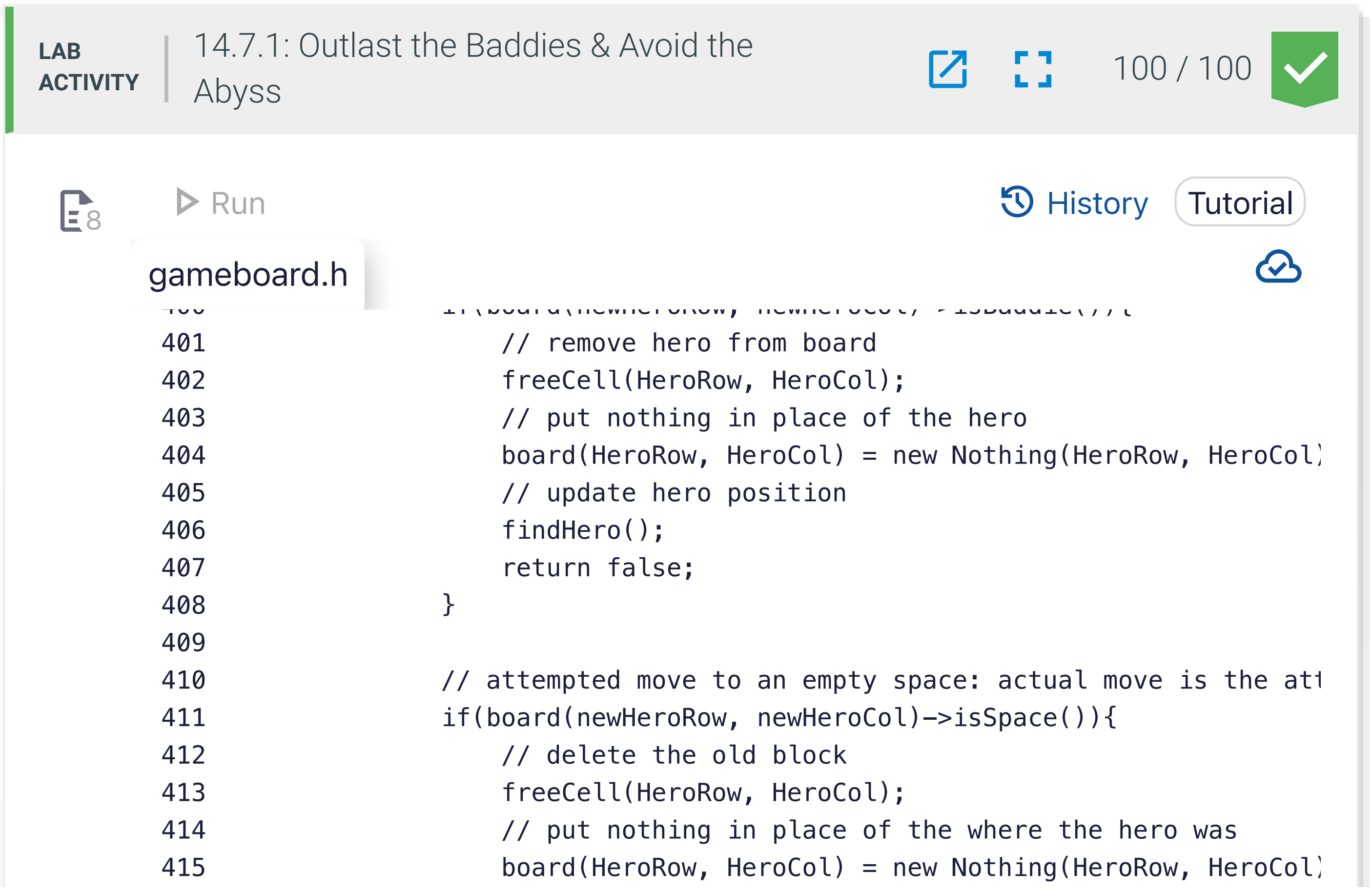
The grading breakdown is as follows:

- **100 points for functionality** - autograder points with manual deductions for any style issues (check syllabus for how style is graded) and program requirements not adhered to. Style is checked for the code you write in **GameBoard.h** and **BoardCell.h**.
    - **24 points** for **grid.h** implementation (simply copied from **Project 06**)
    - **20 points** for **attemptToMove()** in **BoardCell** class
    - **10 points** for **basic member functions** of **GameBoard** class; i.e. hero position setter, getter, and findHero()
    - **16 points** for **hero movement ONLY** in **makeMoves()** for **GameBoard** class
    - **24 points** for **Baddie movements ALSO** in **makeMoves()** for **GameBoard** class
    - **6 points** for **valgrind memory leak and error checks** for **full game play**
- **extension extra credit** - up to +3 points for a creative game play extension to make the experience more entertaining for the user; extra credit requires fully functional implementation AND a full description/instructions of the extension.
- **early submission extra credit** - +1 point/day early with a maximum of +3 extra credit points.

Citation/Inspiration

The idea for this programming project is inspired by John Bell at the University of Illinois Chicago.

Copyright Statement



```
        if(board(newHeroRow, newHeroCol)->isBaddie()){
            // remove hero from board
            freeCell(HeroRow, HeroCol);
            // put nothing in place of the hero
            board(HeroRow, HeroCol) = new Nothing(HeroRow, HeroCol
            // update hero position
            findHero();
            return false;
        }

        // attempted move to an empty space: actual move is the at
        if(board(newHeroRow, newHeroCol)->isSpace()){
            // delete the old block
            freeCell(HeroRow, HeroCol);
            // put nothing in place of the where the hero was
            board(HeroRow, HeroCol) = new Nothing(HeroRow, HeroCol
                // remove the new block
                freeCell(newHeroRow, newHeroCol);
                // move hero to cell
                board(newHeroRow, newHeroCol) = new Hero(newHeroRow, ne
                // reset hero position
                findHero();
            }

        bool monsterCatchesHero = false;

```

DESKTOP CONSOLE

Submit for grading

Coding trail of your work [What is this?]

```
11/22 F29,33,33 M40,24,44,48,52 W24,57,44,56 R71,71
,64,64,71,63,63,71,73 F44,44,73 S44,73,65,44,73,73,73
,73,73,73,65,73 M75,44,75,75,75,73,44,75,75,75,44,78
,73,84,100,44,65,44,65,66,66,44,44,65,44,65,100,100,100
,67,64,44,100,100,44,94,100,88,100,91,100,97,97,70,44
,44,100,100,100,100,100,100 min:245
```

Latest submission - 5:18 PM CST on 12/02/24

Submission passed all tests ✓

Total score: 100 / 100

☑ Only show failing tests *(40 tests hidden)*

Open submission's code

5 previous submissions

5:14 PM on 12/2/24	100 / 100	**View**
5:12 PM on 12/2/24	100 / 100	**View**

5:11 PM on 12/2/24	100 / 100	View
5:10 PM on 12/2/24	100 / 100	View
5:09 PM on 12/2/24	100 / 100	View

Trouble with lab?

Feedback?

Activity summary for assignment: Programming Project 07 - Outlast the Baddies & Avoid the Abyss

100 / 100 points

Due: 12/05/2024, 11:59 PM CST

Completion details