Assignment 2 – MIS and MID

Designed by struct by_lightning{};
Group 6
Kuir Aguer, Brendan Duke, Jean Ferreira,
Zachariah Levine and Pranesh Satish

Module Decomposition

The application has been decomposed into three broad conceptual modules: hardware hiding, behaviour hiding and software decision.

Hardware hiding

The application relies on the operating system to handle the interface to low-level machine details.

Software Decisions

The application uses the graphics and sdl2_connect4 modules to implement key software functionality. Specifically, the graphics module encapsulates the software functionality that processes graphics for the game board. Similarly, the sdl2_connect4 module represents the set of software functionality that process events triggered on the board such as mouse clicks. In addition to software hiding, these two modules encapsulate some of the application's behaviour. For example, the sdl2_connect4 module contains the main game loop that runs the application.

Behaviour hiding

The board and gameLogic modules handle the rest of the application's behaviour. In particular, the gameLogic module handles the application's logic including token processing and determining the game status. Similarly, the board module is a data abstraction for the board and the set of operations for interacting with it. The conceptual and concrete modules that comprise the application are summarized in the following diagram. Additional details can also be found in the design documents.

Design Verification and Review

March 16, 2015

Tasks Required and Description of Observed Completeness (Revision 1)

1. Program enables the user to make moves in turn on existing frame
   (a) This was observed as complete with no errors or changes required
2. Order of play is determined randomly
   (a) This was accomplished in the previous version and maintained
3. Application verifies validity of move as well as current state of the game
   (a) In the previous version of this application, internal game states hadalready been implemented and as such, these were simply ouput. Game state was clearly observed.
4. The user shall be able to start a new game, store an existing game, orresume an unfinished game
   (a) The addition of save/load was newly observed and the previous implementation of new game maintained

Architecture and Design Review (Revision 1)

Modules:

The tasks for the application have thus far been completed through applying 5 modules revolving around the concept of game and board states. This implementation was determined to have identified the unique portions and isolated them so as to emulate a properly designed application while completing the tasks identified.

- The modules were maintained and any implementation added simply expanded upon existing modules. The main changes to design/architecture involved the save/load game implementation as everything else was observed following the previous module implementation.

1. sdl2_connect4
   - sdl2_connect4 is the module that handles user interface decisions made in various designated states and contains the main game loop
2. gameLogic

- gameLogic maintains the state and a “current model” of the game and updates physics/logic accordingly
- Added save/load implementation to game logic as a temporary game state change. No issues were identified in this architecture/design decision although consideration was given to having independant methods being called form the main game loop as opposed to a chain from game logic.

3. graphics

   - graphics allows for graphical manipulation of the user interface with regards to the game thorugh use of game information through board and linkedList
   - Added implementation of game functionality resulted in an observed addition to button graphics. No issues were identified in this architecture/design decision

4. board

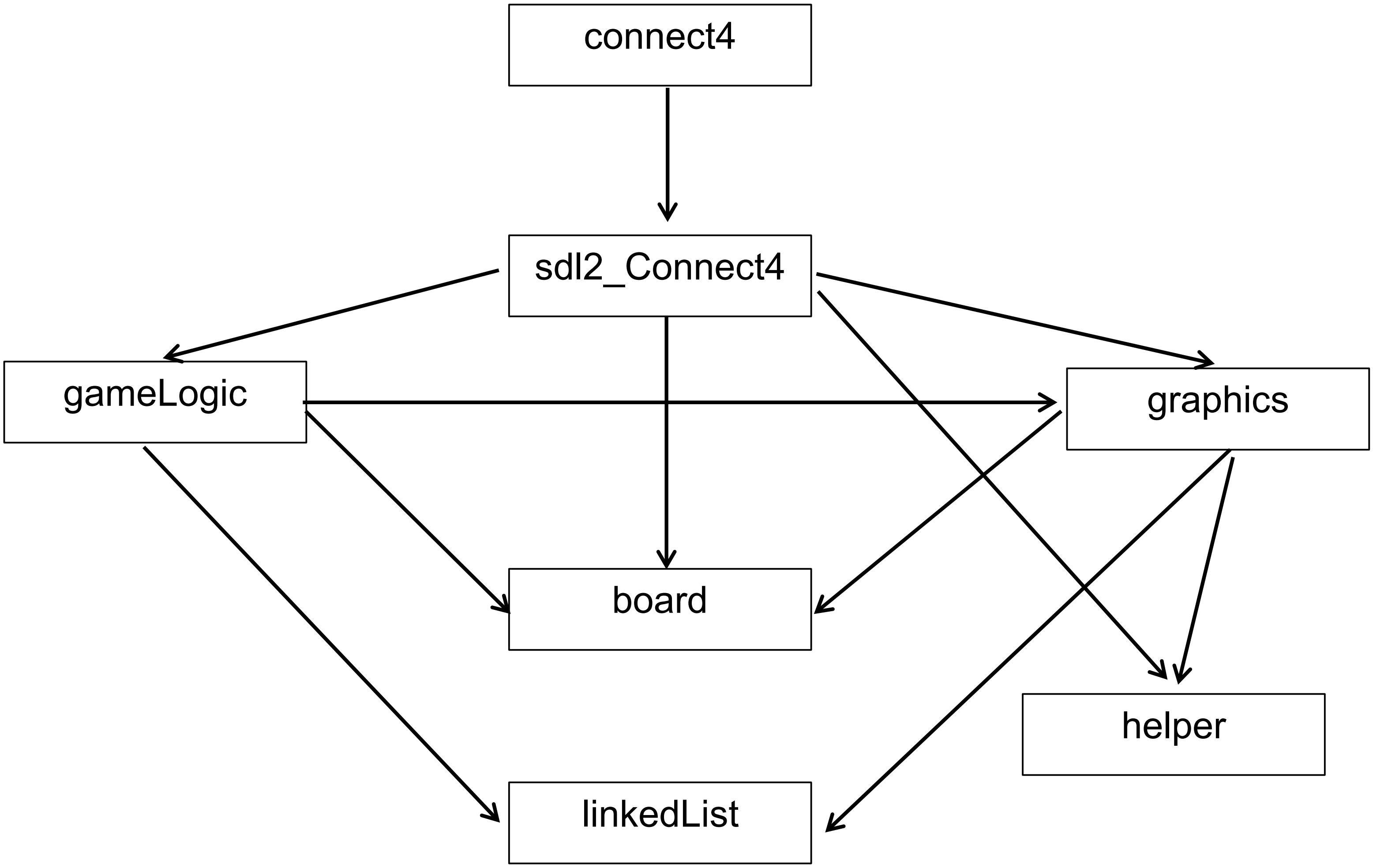
   - board is an API for the current state

5. linkedList

   - linkedList is an API for the falling token drop implementation

Note: Initially, both the game loop implementation as well as graphical update implementation were contained within the same module. However, it was identified that this could hamper design if the graphical implementation needed to be changed. So, the graphics was abstracted out into its own module with the game loop remaining with the sdl2_connect4

Uses Hierarchy



Revision Notation:

To track and document the versions of our functions we have developed the following revision notation:

Notation:	Meaning:
--rev a	The function was created in revision a
--rev b.c	The function was originally created in revision b and modified in revision c. Note: c>b
--rev b.c.d	The function was originally created in revision b, was modified in revision c, and was modified again in revision d. Note: d > c > b

Module: board

Defines a 6 row by 7 column Connect Four board, as an abstract data type (ADT) Board. Contains access programs to interact with the board. When compiling the production application, use the compiler flag -DNDEBUG.

--- Interface ---

Uses:

None

Defined Macros/Constants:

NUM_ROWS: integer
The number of rows in the Connect Four board

NUM_COLS: integer
The number of columns in the Connect Four board

Defined Types:

Token: an enumeration with elements: EMPTY, RED, BLUE
The type of the elements that are contained in, inserted to, and returned from the ADT Board

Board: a pointer or reference to the board “object”
Variables of this type are instances of the ADT Board

Access programs:

board_create:

Return Type: Board

Parameters: None

Creates and returns a Board object, with all cells initialized to EMPTY. Terminates the program if the board cannot be created.

board_destroy:

Return Type: void

Parameters: Board b

Frees the memory allocated to the Board b.

board_checkCell:

Return Type: Token

Parameters: Board b, integer row, integer col

Returns the value of the board element in (row, col).

board_dropToken:

Return Type: integer

Parameters: Board b, Token token, integer col

Inserts the token, into the board b, in the column col. The inserted position is determined by where the token would fall in a physical Connect Four board, by gravity. Specifically, the token will be inserted into the position (row, col) such that the board b has the value EMPTY at (row, col) and row is as large as possible. The function returns 0 if the token was successfully inserted into the board and -1 if it wasn't successful (the column is already full or it was passed a NULL pointer for b).

board_dropPosition:

Return Type: integer

Parameters: Board b, integer col

Returns the row where a token be inserted by the board_dropToken (see board_dropToken). Returns -1 if the column is already fullor it was passed a NULL pointer for b.

board_empty:

Return Type: void

Parameters: Board b

Sets all cells of the board b to EMPTY.

board_load: --rev 1

Return Type: void
Parameters: Board b, FILE *in_file
Loads the board b from a previously saved game in in_file.

board_save: --rev 1

Return Type: void
Parameters: Board b, FILE *out_file
Saves the game configuration (in board b) to file in_file.

--- Implementation ---

Uses:

board.h
stdio.h
stdlib.h
assert.h

Type Definitions/Structure,Union,Enumeration Declarations:

board_type: The internal representation of the board ADT; a structure of a two dimensional array of type Token. The top left of the board is defined to be (0,0).

```
struct board_type {
        Token board[NUM_ROWS][NUM_COLS];
};
```

Variables:

None

Access Programs:

Board board_create(void)

Creates an empty board by allocating memory of size the structure board_type.

```
Board b = (Board) malloc(sizeof(struct board_type))
board_empty(b)
return b
```

void board_destroy(Board b)

Deallocates the memory associated with b.

```
free(b)
```

Token board_checkCell(Board b, int row, int col)

Return the value of the board element at (row, col) by accessing the two-dimensional array in the structure that b points to.

```
if b != NULL
    return b.board[row][col]
else
    return EMPTY
```

int board_dropToken(Board b, Token token, int col)

Inserts the token, into the board b, in the column col. The inserted position is determined by where the token would fall in a physical Connect Four board, by gravity. Specifically, the token will be inserted into the position (row, col) such that the board b has the value EMPTY at (row, col) and row is as large as possible. The function returns 0 if the token was successfully inserted into the board and -1 if it wasn't successful (the column is already full or it was passed a NULL pointer for b).

```
Assert col >= 0 AND col < NUM_COLS

if b != NULL
    let int row = board_dropPosition(b, col)
    if row != -1
        b.board[row][col] = token
        return 0

return -1
```

int board_dropPosition(Board b, int col)

Returns the row where a token be inserted by the board_dropToken (see board_dropToken). Returns -1 if the column is already full or it was passed a NULL pointer for b. This was done by looping down the column of the two-dimensional array as long as the next element in the column was empty.

```
Assert col >= 0 AND col < NUM_COLS

if b != NULL
    let int row = 0
    if b.board[0][col] != EMPTY
        return -1
    while row + 1 < NUM_ROWS AND b.board[row + 1][col] == EMPTY
        row++
return row
```

void board_empty(Board b)

Sets all cells of the board b to EMPTY by looping through every element in the two-dimensional array and setting its value to EMPTY.

```
if b != NULL
    let int row = 0
    let int col = 0
    while row < NUM_ROWS
        while col < NUM_COLS
            b.board[row][col] = EMPTY
```

void board_load(Board b, FILE *in_file) --rev 1

Load the board from a previously saved game

```
if in_file != 0
    fread(b, sizeof(*b), 1, in_file)
else
    print invalid file pointer in board_load
```

void board_save(Board b, FILE *out_file) --rev 1

Save the board configuration

```
if in_file != 0
    fwrite(b, sizeof(*b), 1, out_file)
else
    print invalid file pointer in board_save
```

Module: gameLogic

Responsible for implementing the game logic including processing token positioning and game status.

Interface

Uses:

board.h
graphics.h
linkedList.h
C runtime library: stdio.h, stdlib.h and time.h

Defined Macros/Constants:

NUMBER_OF_STATES: integer – Rev 0
Number of columns on the Connect 4 board.

Global Declarations:

None

Defined Types:

Player: – Rev 0
An enumeration with elements: PLAYERONE, PLAYERTWO, and RANDOMPLAYER
Game modes representing a one- or two-player game, or a RANDOMPLAYER, meaning that the player who goes first should be selected randomly.

MenuState: – Rev 0
An enumeration with the elements: MAINMENU, ONEPLAYER, TWOPLAYER, SETUP, CREDITS, QUIT, DONOTHING.
These represent the current state of the game, meaning that the event handling, rendering and logic functions that are called from the game loop are controlled by which MenuState is currently set in the GameState. E.g. if the current MenuState in GameState is set to MAINMENU, then event handling, rendering and logic functions corresponding to the main menu will be called for each iteration of the game loop.

GameProgress: – Rev 1
An enum representing whether the current game is in progress, a draw, or whether red has won or blue has won.

GameState: – Rev 0.1
A struct type representing the current state (MenuState), current progress (GameProgress), current token (Token), current player (Player), the board (Board), the graphics state (GraphicsState), and two booleans loadGame and saveGame that indicate whether the game should be loaded or saved or not the next time the logic function is called.

Access Programs:

mainMenuLogic: – Rev 1
Return Type: void
Parameters: GameState *gameState
Executes any logic and updating that needs to be done in the MAINMENU state, to make the menu do what it is supposed to do. Logic enables loading and saving of games.

SetupLogic: – Rev 0.1
Return Type: void
Parameters: GameState *gameState
Executes any logic that needs to be done in the SETUP state. SetupLogic enables dropping of tokens and checking whether the game has been won in order to enable switching to two player, as well as printing error messages.

ReadyToTransitionSetupTwoPlayer: – Rev 1
Return Type: bool
Parameters: GameState *gameState

```
if (>= four red in a row) OR (>= four blue in a row) or (|red tokens – blue tokens| > 1) OR (board is full)
    return false
else
    return true
```

Implementation:

Access Programs:

mainMenuLogic: – Rev 1
Return Type: void
Parameters: GameState *gameState
Revision 1 changes: added loadGame functionality.
If the load game flag is set in gameState, then call loadGame, and set the load game flag to false.

```
if (gameState.loadGame == true)
    loadGame(gameState)
    gameState.loadGame = false
```

setupLogic: – Rev 0.1
Return Type: void
Parameters: GameState *gameState
Revision 1 changes: added saveGame functionality.

If the save game flag is set in gameState, then call saveGame, and set the save game flag to false. Then traverse the gFallingTokens data structure by calling traverseList, and call updateFallingToken on each node of gFallingTokens

```
if (gameState.saveGame == true)
        gameState.saveGame = false
        saveGame(gameState)
traverseList(updateFallingToken, 0.5, gFallingTokens)
```

twoPlayerLogic: – Rev 0.1
Return Type: void
Parameters: GameState *gameState
Revision 1 changes: added renderIndicatorToken flag changes to control showing where a token will drop. Added the renderStatus flags to decide whether to display that the game has been won, drawn or is in progress.

If the save game flag is set in gameState, then call saveGame, and set the save game flag to false. Reset all graphics state flags to zero by calling resetGraphicsState. Then traverse the gFallingTokens data structure by calling traverseList, and call updateFallingToken on each node of gFallingTokens. Set renderIndicatorToken flag in gameState.graphicsState to false if the game is in progress, so that we don't render the indicator token when the game is over. Check the game state to see if red won, blue won or if there is a draw. If red won or blue won, highlight the winning tokens set gameState.currentProgress to REDWON or BLUEWON respectively, and set the graphics state renderIndicatorToken flag to false, the renderStatusInProgress flag to false, and the renderStatusRedWon or renderStatusBlueWon to true respectively. If the game is a draw, then set the same flags except set renderStatusDrawGame to true instead of renderStatusBlueWon or renderStatusRedWon.

```
If (gameState.saveGame == true)
        gameState.saveGame = false
        saveGame(gameState)

resetGraphicsState(gameState.graphicsState)
traverseList(updateFallingToken, 0.5, gFallingTokens)

if (gameState.currentProgress != INPROGRESS)
        gameState.graphicsState.renderIndicatorToken = false

let didRedWin = didColourWin(gameState.board, RED)
let didBlueWin = didColourWin(gameState.board, BLUE)
let isDraw = checkDraw(gameState.board)

if (didRedWin OR didBlueWin)
    setHighlightedTokenList(getSequentialTokens(gameState.board),
        gameState.graphicsState)
if (didRedWin OR didBlueWin OR isDraw)
    gameState.graphicsState.renderIndicatorToken = false
    gameState.graphicsState.renderStatusInProgres = false
if (didRedWin)
    gameState.currentProgress = REDWON
    gameState.renderStatusRedWon = true
else if (didBlueWin)
    gameState.currentProgress = BLUEWON
    gameState.renderStatusBlueWon = true
else if (isDraw)
    gameState.currentProgress = DRAW
    gameState.renderStatusDrawGame = true
```

Local Programs:

countTokens: – Rev 0
Return Type: int
Parameters: Board board, Token colour

Check how many cells in the board are the colour passed as a parameter by iterating over the board and incrementing a counter.

```
Let numberOfTokens= 0
for all row in {0, …, NUM_ROWS – 1}
    for all col in {0, …, NUM_COLS – 1}
        if (board_checkCell(board, row, col) == colour)
            numberOfTokens = numberOfTokens + 1
return numberOfTokens
```

didColourWin: – Rev 0
Return Type: bool
Parameters: Board board, Token colour

Iterates over each cell in the board and checks if four in a row of the colour passed can be found going left, down, diagonally decreasing left, or diagonally increasing left starting at the current cell.

```
For all row in {0, …, NUM_ROWS – 1}
```

```
    for all col in {0, …, NUM_COLS – 1}
        if (board_checkCell(board, row, col) == colour)
            currentCol = col, currentRow = row
            while ((currentCol >= 0) AND
              board_checkCell(board, currentRow, currentCol) == colour))
                if ((col – currentCol) == 3)
                    return true
                currentCol = currentCol – 1

            currentCol = col, currentRow = row
            while ((currentRow >= 0) AND
              board_checkCell(board, currentRow, currentCol) == colour))
                if ((row – currentRow) == 3)
                    return true
                currentRow = currentRow – 1

            currentCol = col, currentRow = row
            while ((currentRow >= 0) AND (currentCol >= 0) AND
              board_checkCell(board, currentRow, currentCol) == colour))
                if ((row – currentRow) == 3)
                    return true
                currentRow = currentRow – 1
                currentCol = currentCol – 1

            currentCol = col, currentRow = row
            while ((currentRow >= 0) AND (currentCol < NUM_COLS) AND
              board_checkCell(board, currentRow, currentCol) == colour))
                if ((row – currentRow) == 3)
                    return true
                currentRow = currentRow – 1
                currentCol = currentCol + 1

        else
            return false
```

checkDraw: – Rev 0
Return Type: bool
Parameters: Board board

Checks if the game is drawn by checking if the board is full .

```
numberOfRedToken = countTokens(board, RED)
numberOfBlueToken = countTokens(board, BLUE)
if ((numberOfRedToken + numberOfBlueToken) == (NUM_ROWS *
NUM_COLS))
    return TRUE
else
    return FALSE
```

checkInvalidBoard: – Rev 0
Return Type: bool
Parameters: Board board
Checks whether the |# of red tokens - # of blue tokens| > 1; if so returns true for invalid board, else returns false.

```
numberOfRedToken = countTokens(board, RED)
numberOfBlueToken = countTokens(board, BLUE)
if square (numberOfRedToken - numberOfBlueToken) > 1
    return TRUE
else
    return FALSE
```

equals: – Rev 0
Return Type: bool
Parameters: TokenLocation *tokenA, TokenLocation *tokenB
Checks if tokenA and tokenB are equal in the sense that their row, column and colour are all equal.

```
return (tokenA->row == tokenB->row) &&
    (tokenA->colomn == tokenB->column) &&
    (tokenA->colour == tokenB->colour)
```

addNewTokenLocation: – Rev 0
Return Type: List<TokenLocation> *
Parameters: List<TokenLocation> tokenList, int row, int col, Token colour
Adds a tokenLocation(row, column, colour) to tokenList by first checking if an equal token is in tokenList already by calling reduceList; only add it to the list

```
Let newHighlightedToken = TokenLocation(row, column, colour)
matchingToken = reduceList(equals, newHighlightedToken, tokenList)
if (matchingToken == NULL)
    return addToList(newHighLightedToken, tokenList)
else
    free(newHighlightedToken)
    return tokenList
```

getSequentialTokens: Rev 0
Return Type: List<TokenLocation> *
Parameters: Board board
Returns a list of all tokens of the given colour that are part of >= 4 in a row sequences

```
Let sequentialTokens = NULL
For all row in {0, …, NUM_ROWS – 1}
    for all col in {0, …, NUM_COLS – 1}
        let colour =  board_CheckCell(board, row, col)
        if (colour != EMPTY)
            currentCol = col, currentRow = row
            while ((currentCol >= 0) AND
              board_checkCell(board, currentRow, currentCol) ==
colour))
                if ((col – currentCol) == 3)
                    while (currentCol <= col)
                        sequentialTokens =
    addNewTokenLocation(sequentialTokens, row, currentCol, colour)
                        currentCol = currentCol + 1

            currentCol = col, currentRow = row
            while ((currentRow >= 0) AND
              board_checkCell(board, currentRow, currentCol) ==
colour))
                if ((row – currentRow) == 3)
                    while (currentRow <= row)
                        sequentialTokens =
    addNewTokenLocation(sequentialTokens, currentRow, col, colour)
                        currentRow = currentRow + 1

            currentCol = col, currentRow = row
            while ((currentRow >= 0) AND (currentCol >= 0) AND
              board_checkCell(board, currentRow, currentCol) ==
colour))
                if ((row – currentRow) == 3)
                    while (currentRow <= row)
                        sequentialTokens =
    addNewTokenLocation(sequentialTokens, currentRow, currentCol, colour)
                        currentRow = currentRow + 1
                        currentCol = currentCol + 1

            currentCol = col, currentRow = row
            while ((currentRow >= 0) AND (currentCol < NUM_COLS)
AND
```

```
             board_checkCell(board, currentRow, currentCol) ==
colour))
                if ((row – currentRow) == 3)
                    while (currentRow <= row)
                        sequentialTokens =
    addNewTokenLocation(sequentialTokens, currentRow, currentCol, colour)
                        currentRow = currentRow + 1
                        currentCol = currentCol – 1

return sequentialTokens
```

setCurrentToken: – Rev 1
Return Type: void
Parameters: GameState *gameState
Sets the current token to red if there are more blue than red tokens, or vice versa. If the counts are equal, sets the current token to RANDOMTOKEN.

ReadyToTransitionSetupTwoPlayer: – Rev 1
Return Type: bool
Parameters: GameState *gameState

```
if (>= four red in a row) OR (>= four blue in a row) or (|red tokens – blue tokens|
> 1) OR (board is full)
    return false
else
    return true
```

Deleted in Revision	Declarations Deleted
1	transitionSetupMainMenu --rev0
1	square --rev0

Module: graphics

Responsible for all graphical representation on the screen, such as loading media files, setting up coordinates for the game layout, and positioning images throughout the screen accordingly.

--- Interface ---

Uses:

SDL2/SDL.h	-- rev 0
string	-- rev 0
board.h	-- rev 0
linkedList.h	-- rev 0
helper.h	-- rev 1

Constants:

CONNECT4_WINDOW_OFFSET_Y: int -- rev 0
Prevents the application window to open beyond the boundaries of the client's screen

SCALE: float -- rev 0
Allows the game screen to be scaled

SCREEN_WIDTH: int -- rev 0
Window width size for the game

SCREEN_HEIGHT: int -- rev 0
Window height size for the game

TOKEN_WIDTH: int -- rev 0
Width of the tokens used for the game

TOKEN_HEIGHT: int -- rev 0
Height of the tokens used for the game

GRID_OFFSET_Y: int -- rev 0
Offset of column lines for the game playing board

GRID_OFFSET_X: int -- rev 0
Offset of row lines for the game playing board

GRID_WIDTH: int -- rev 0
Width of the game board

GRID_HEIGHT: int -- rev 0
Height of the game board

MAINMENU_TWOPLAYER_BUTTON_RECT: Rectangle -- rev 1
Creates a rectangular button for two player mode in main menu

MAINMENU_SETUP_BUTTON_RECT: Rectangle -- rev 1
Creates a rectangular button for setup mode in main menu

MAINMENU_LOADGAME_BUTTON_RECT: Rectangle -- rev 1
Creates a rectangular button to load a previous game in main menu

MAINMENU_CREDIT_BUTTON_RECT: Rectangle -- rev 1
Creates a rectangular button to open the credits menu in main menu

CREDITS_QUIT_BUTTON_RECT: Rectangle -- rev 1
Creates a rectangular button to quit the credits menu

SETUP_1PLAYER_BUTTON_RECT: Rectangle -- rev 1
Creates a rectangular button to start one player mode from setup mode

SETUP_2PLAYER_BUTTON_RECT: Rectangle -- rev 1
Creates a rectangular button to start two player mode from setup mode

SETUP_MENU_BUTTON_RECT: Rectangle -- rev 1
Creates a rectangular button to go back to main menu from setup mode

REFRESH_BUTTON_RECT: Rectangle -- rev 1
Creates a rectangular button that allows the board to be cleared

SAVE_BUTTON_RECT: Rectangle -- rev 1
Creates a rectangular button that allows the current game to be saved

SETUP_CLICKY_TOKENS_OFFSET: integer -- rev 0
Calculates the position of the button that changes the current token colour in play(ie. Going from blue's turn to red's turn and vice versa)

TOKEN_RADIUS: integer -- rev 1
Represents the radius of the a token

SETUP_RED_CLICKY_TOKENS_CIRCLE: Circle -- rev 1
Creates a circular button to represent the position of the red token in the setup mode

SETUP_BLUE_CLICKY_TOKENS_CIRCLE: Circle -- rev 1
Creates a circular button to represent the position of the blue token in the setup mode

TWOPLAYER_MENU_BUTTON_RECT: Rectangle -- rev 1
Creates a rectangular button to go back to main menu from two player mode

SETUP_BOTTOM_BUTTONS_OFFSET: integer -- rev 0
Calculates the bottom position of all the buttons in the bottom of the setup game mode screen

INVALID_MESSAGE_WIDTH: integer -- rev 1
Represents the width of a message in the setup mode

INVALID_MESSAGE_HEIGHT: integer -- rev 1
Represents the height of a message in the setup mode

INVALID_MESSAGE_X: integer -- rev 0
Represents the x-axis position of a message in the setup mode

INVALID_MESSAGE_Y: integer -- rev 0
Represents the y-axis of a message in the setup mode

INVALID_TOKEN_MESSAGE_X: integer -- rev 0
Represents the x-axis of a message in the setup mode

INVALID_TOKEN_MESSAGE_Y: integer -- rev 0
Represents the y-axis of a message in the setup mode

STATUS_MESSAGE_Y: integer -- rev 1
Represents the y-axis position of the current game progress message

STATUS_MESSAGE_X: integer -- rev 1
Represents the x-axis position of the current game progress message

Global Declarations:

extern List<FallingToken> *gFallingTokens -- rev 0
A list that represents all the falling tokens during gameplay

Defined Functions:

initScaledRectange: Rectangle -- rev 1

Creates a rectangular object with two set of (x,y) points, and it scaled appropriately by the given value of the SCALE defined

```
initScaledRectanlge(args: : integer x1, integer y1, integer x2, integer y2, scale)
    define new rectangle, newRect
    newRect_topRight_X := x1 * scale
    newRect_topRight_Y := y1 * scale
    newRect_botLeft_X := x2 * scale
    newRect_botLeft_Y := y2 * scale
    return newRect
```

initCircle: Circle -- rev 1

Creates a circular object with a set of (x,y) point and a radius

```
initCircle(args: integer x, integer y, integer radius)
    define new Circle, newCirc
    newCirc_X = x1
    newCirc_Y = y
    newCirc_R = radius
    return newCirc
```

Defined Structures:

FallingToken -- rev 0

Holds information about each token, which will help gravity to be simulated on each token individually.

TextureWrapper -- rev 0

Provides a way to store dimensions to a texture.

TokenLocation (incomplete type) -- rev 0

Allows a tokens to be defined by its color and position within the game board.
(the structure definition for this struct is visible in the interface)

```
struct TokenLocation
  integer row
  integer column
  Token colour
```

Access programs:

drawFallingToken: -- rev 0

Return Type: void

Parameters: FallingToken *token

Given a token, determine the position to drop it.

clearFallingToken: -- rev 0

Return Type: void

Parameters: FallingToken *fallingToken

Finds the position of the falling token.

updateFallingToken: -- rev 0

Return Type: void

Parameters: FallingToken *fallingToken, float dt

*Updates the position of a falling token depending on the time (*dt*) it has been airborne.*

mainMenuRender: -- rev 0

Return Type: void

Parameters: none

Rendering for the main menu

creditsRender: -- rev 1

Return Type: void

Parameters: none

Rendering for the credits screen

setupRender: -- rev 0

Return Type: void

Parameters: none

Rendering for the setup game mode

twoPlayerRender: -- rev 1

Return Type: void

Parameters: none

Rendering for the two player screen

init: -- rev 0

Return Type: boolean value

Parameters: none

Returns true if the program has been initialized and window has been created successfully.

loadMedia: -- rev 0.1

Return Type: boolean value

Parameters: none

Returns true if all media (images) required for the game has been successfully accessed.

close_sdl: -- rev 0

Return Type: void

Parameters: none

Ends the program properly and closes the window

dropToken: -- rev 0

Return Type: boolean

Parameters: Board b, Token tokenColour, integer col

Returns true if a token and been successfully dropped onto the given board, at the specified column.

setHighlightedTokenList: -- rev 0

Return Type: void

Parameters: List<TokenLocation> *highlightedTokenList

List of token location that are highlighted in the setup game mode.

resetGraphicsState: -- rev 1

Return Type: void

Parameters: GraphicState *graphicState

Allows the current graphic state to be reset

renderIndicatorToken: -- rev 1

Return Type: void

Parameters: TokenLocation *indicatorToken

Allows rendering of the indicator tokens, these tokens are the ones that appear before clicking a position to drop.

loadGraphics: -- rev 1

Return Type: void

Parameters: GraphicsState *graphicsState, FILE *out_file

Loads the graphic state saved in the out_file

saveGraphics: -- rev 1

Return Type: void

Parameters: GraphicsState *graphicsState, FILE *in_file

Saves the graphic state saved in the in_file

Deleted in Revision	Declarations Deleted
1	MAINMENU_SETUP_BUTTON_LEFT --rev0
1	MAINMENU_SETUP_BUTTON_RIGHT --rev0
1	MAINMENU_SETUP_BUTTON_TOP --rev0
1	MAINMENU_SETUP_BUTTON_BOTTOM --rev0
1	MAINMENU_QUIT_BUTTON_LEFT --rev0
1	MAINMENU_QUIT_BUTTON_RIGHT --rev0
1	MAINMENU_QUIT_BUTTON_TOP --rev0
1	MAINMENU_QUIT_BUTTON_BOTTOM --rev0
1	SETUP_2PLAYER_BUTTON_WIDTH --rev0
1	SETUP_2PLAYER_BUTTON_HEIGHT --rev0
1	MAINMENU_QUIT_BUTTON_LEFT --rev0
1	SETUP_1PLAYER_BUTTON_WIDTH --rev0
1	SETUP_1PLAYER_BUTTON_HEIGHT --rev0
1	SETUP_MENU_BUTTON_WIDTH --rev0
1	SETUP_MENU_BUTTON_HEIGHT --rev0
1	transitionSetupRender -- rev 0
1	deleteStillToken -- rev 0

--- Implementation ---

Uses:

SDL2/SDL.h
string
graphics.h
stdlib.h

Type Declarations:

TextureWrapper: -- rev 0

Any object of this struct will be able to use SDL's texture function, and make it easier to calculate positioning for these textures by the use of the width and height.

```
struct TextureWrapper
   define new SDL texture
   integer width
   integer height
```

FallingToken: -- rev 0

Allows manipulation of multiple falling tokens at the same time, independently of each other.

```
struct FallingTone

   Define new Token enumeration , token // token is either  {BLUE, RED, EMPTY}
   integer x          //distance from left to right
   integer y          //distance from top to bottom
   integer v          // velocity
   integer yFinal     // final position
   boolean isFalling
```

Global Variables

*gFallingTokens: List<FallingToken> --rev 0
This list keeps track of all the tokens that are falling in the game.

Variables

(Any variables beginning with a g (ie gVarName) is a global variable without external linkage (meaning it's internal to module))

*gWindow: SDL_Window --rev 0
Represents the window to be used for the game, any change with the window itself (height, width, border, and full screen) will be made through this.

*gConnect4Board: TextureWrapper --rev 0
Represents the image of the game board.

*gRedToken: TextureWrapper --rev 0
Represents the image of the red token.

*gBlueToken: TextureWrapper --rev 0
Represents the image of the blue token.

*gMainMenu: TextureWrapper --rev 0
Represents the image of the main menu.

*gGlow: TextureWrapper --rev 0
Token highlighting texture.

*gInvalidMessage: TextureWrapper --rev 0.1
Invalid board message display

*gInvalidTokenMessage: TextureWrapper --rev 0.1
Invalid game setup message display.

*gRenderer: SDL_Rendered --rev 0
Responsible for rendering any image onto an SDL_Window.

*gCreditScreen: TextureWrapper --rev 1
Represents the image for the credits menu

*gSetupScreen: TextureWrapper --rev 1
Represents the image for the setup game mode screen

*gTwoPlayerScreen: TextureWrapper --rev 1

Represents the image for the two player mode screen

*gStatusBlueWon: TextureWrapper --rev 1
Message to be displayed when blue wins in two player mode

*gStatusRedWon: TextureWrapper --rev 1
Message to be displayed when red wins in two player mode

*gStatusDraw: TextureWrapper --rev 1
Message to be displayed when draw game happens in two player mode

*gStatusInProgress: TextureWrapper --rev 1
Message to be displayed while a two player game is in progress

*gRefresh: TextureWrapper --rev 1
Represents the refresh button image in any game mode

*gHighlightedTokens: List<TokenLocation> --rev 0
This list contains all currently highlighted tokens in game.

Local Programs:

static void highlightToken(TokenLocation *tokenToHighlight)

--rev 0

Given a tokenLocation pointer as argument, this function finds the row and column of the respective token and highlights it. The TextureWrapper gGlow is used to highlight the tokens.

```
highlightToken(args: pointer to TokenLocation):

    SDL_Rect fillRect := position of TokenLocation


    call SDL_SetRenderDrawColor(args: gRenderer, 0xFF, 0xFF, 0xFF, 0x66)
    call SDL_SetRenderDrawBlendMode(args: gRenderer,
                                          SDL_BLENDMODE_BLEND)
    call SDL_RenderFillRect(args:  gRenderer, &fillRect)
    call SDL_SetRenderDrawColor(args:  gRenderer, 128, 128, 128, 0xFF )
    call SDL_SetRenderDrawBlendMode(args: gRenderer,
                                          SDL_BLENDMODE_NONE);
```

static void freeTexture(TextureWrapper *myTexture) --rev 0

Responsible for de-allocating any Texture Wrapper. Given the argument of a pointer to TextureWrapper, destroy the texture with SDL_DestroyTexture, and set it to NULL.

```
freeTexture(args: pointer to TextureWrapper myTexture)
        if (myTexture != null)
                if (texture of myTexture != null)
                        call SDL_DestroyTexture(args: texture of myTexture)
                        texture of myTexture := null
                endif

                call free(args: myTexture)
        endif
```

bool *loadAllFiles(std::string fileNames[], TextureWrapper **textureNames[], int size) --rev 1

A function to load all the media files; will return true if all loaded successfully

```
for i = 0; I < textureNames.length; i = i + 1
        textureNames[i] = loadTexture(fileNames[i])
        if (textureNames[i] == NULL)
                print("Failed to load fileNames[i]")
                return false
return true
```

bool compareXPosition(FallingToken *listItem, FallingToken *item) --rev 0

Returns true if the two tokens are in the same column, false otherwise

```
return listItem.x == item.x
```

static void freeTokenLocation(TokenLocation *tokenLocation) –rev 0

Calls the standard library free() function, while removes the pointer of TokenLocation from memory.

```
freeTokenLocation(args: tokenLocation)
        call free(args: tokenLocation)
```

TextureWrapper *loadTexture(std::string path) --rev 0

Loads bitmaps stored at path and returns them in a TextureWrapper: a hardware texture that known its width and height. Also, scales the bitmap by the SCALE constant defined at the top of this module. Goes through a series of SDL function calls checking for failure points at each call: SDL_LoadBMP, SDL_CreateRGBSurface, SDL_BlitScaled, SDL_SetColorKey, SDL_CreateTextureFromSurface and finally SDL_FreeSurface.

```
LoadedSurface = SDL_LoadBMP(path)
if (loadedSurface == NULL)
        print(“Unable to load image at path! SDL Error: SDL_GetError())
else
        scaleRect = SDL_Rect(0, 0, loadedSurface.width*SCALE,
                                loadedSurface.height*SCALE)
if (scaleSurface == NULL)
        print(“Couldn't create surface scaledSurface”)
else
             SDL_BlitScaled(loadedSurface, NULL, scaledSurface, &scaleRect);

     // Color key image
     SDL_SetColorKey( scaledSurface, SDL_TRUE,
        SDL_MapRGB( scaledSurface->format, 0xFF, 0xFF, 0xFF));

     //Create texture from surface pixels
     newTexture = SDL_CreateTextureFromSurface(gRenderer, scaledSurface);
     if (newTexture == NULL)
       printf("Unable to create texture from path! SDL Error: SDL_GetError()”)
     else
       loadedTexture = TextureWrapper()
       if (loadedTexture == NULL)
         printf("Unable to allocate the TextureWrapper structure for path!")
       else
         loadedTexture.texture = newTexture
         loadedTexture.width = scaledSurface.width
         loadedTexture.height = scaledSurface.height

           SDL_FreeSurface(scaledSurface)
  return loadedTexture;
```

Access Programs:

void mainMenuRender()   --rev 0

Renders the main menu background texture on the screen by calling local function displayMainMenu(). DisplayMainMenu() just copies a global texture gMainMenu onto gRenderer. MainMenuRender() also presents the renderer (causing the texture to be displayed on the screen).

```
mainMenuRender(args: none)
   call displayMainMenu()
   call SDL_RenderPresent(arguments: gRenderer)
```

bool loadMedia()   --rev 0

Loads all the media. The file names to load are all encoded in an array called filesToLoad[] and the global textures are in an array called textureNames[] (both local to the function). We then call load all files with these arrays as arguments.

```
return loadAllFiles(filesToLoad, textureNames, sizeof(filesToLoad))
```

void close_sdl()   --rev 0

Frees all the global textures by calling freeTexture on them, sets those global textures to NULL., destroys gWindow by calling SDL_DestroyWindow(gWindow), and calls SDL_Quit().

void drawFallingToken(FallingToken *fallingToken)   --rev 0

Draws a falling token.

```
If (fallingToken.token == RED)
        tokenTexture = gRedToken
else
        tokenTexture = gBlueToken
tokenRect = SDL_Rect(fallingToken.x, fallingToken.y, TOKEN_WIDTH,
                     TOKEN_HEIGHT)
SDL_RenderCopy(gRenderer, tokenTexture.texture, NULL, tokenRect)
```

void updateFallingToken(FallingToken *fallingToken, float dt) -- rev 0

*Updates position/velocity of fallingToken.token. Increments velocity by ACCEL (local constant) * dt. Increments y position by velocity * dt. Damps the finally velocity by a factor of 1/3 if the fallingToken hit its final position and stops it if its velocity is less than 5 (magnitude).*

```
If (NOT fallingToken.isFalling)
        return
else
        fallingToken.y = fallingToken.y + fallingToken.v * dt
        fallingToken.v = fallingToken.v + ACCEL * dt
        if (fallingToken.y >= fallingToken.yFinal AND fallingToken.v > 0)
                fallingToken.v = -fallingToken.v / 3
                fallingToken.y = fallingToken.yFinal
        if (|v| <= 5 AND fallingToken.y >= fallingToken.yFinal)
                   fallingToken.y = fallingToken.yFinal
                   fallingToken.v = 0
                   fallingToken.isFalling = false
```

void loadGraphics(GraphicsState *graphicsState, FILE *in_file) -- rev 1

Loads the graphics state from a file handle in_file.

```
If (in_file == 0)
        print("Invalid file pointer in loadGraphics function")
else
        gFallingTokens = readListFromFile(gFallingTokens, in_file)
```

void saveGraphics(GraphicsState *graphicsState, FILE *in_file) -- rev 1

Saves the graphics state to a file handle in_file.

```
If (out_file == 0)
        print("Invalid file pointer in saveGraphics function")
else
        writeListToFile(gFallingTokens, out_file)
```

void resetGraphicsState(GraphicsState *graphicsState) --rev 1

This function allows variables of graphicStates (in-game messages or images) to be reset to their initial values. As a game progress, messages and images will be displayed and changed on the screen. By calling this function we can go back to the initial graphics state of the game.

```
resetGraphicsState(args: pointer to GraphicsState)
        set renderIndicatorToken of GraphicState to true
        set renderStatusInProgress of GraphicState to true

        set renderInvalidMessage of GraphicState to false
        set renderInvalidTokenMessage of GraphicState to false
        set renderHighlighted of GraphicState to false
        set renderStatusDrawGame of GraphicState to false
        set renderStatusBlueWon of GraphicState to false
        set renderStatusRedWon of GraphicState to false
```

void creditsRender(GraphicsState *graphicsState) --rev 1

Responsible for rendering the texture for the credits menu, gCreditScreen, and also to display the texture to the screen

```
creditsRender(args: pointer to GraphicsState)
        call SDL_RenderCopy(arguments: gRenderer, texture of gCreditScreen,
                            null, null)

        call SDL_RenderPresent(arguments: gRenderer)
```

bool init() --rev 0

Initializes all of SDL's graphical functions. Also creates a new window (gWindow) and renderer for the window (gRenderer). If everything is initialized successfully, it will return true.
If any initialization is unsuccessful the program will not start and will prompt the user with a message of what has gone wrong.

```
init(args: none)
boolean successFlag := true
if  (call SLD_Init(arguments: SDL_INIT_VIDEO) < 0)
    print error message
    call SDL_GetError()
endif
else
    gWindow := call SDL_CreateWindow(arguments: title,
                        windowCenter, windowOffset, width, height, 0)
    if (gWindow == null)
         display error message and call SDL_GetError
         successFlag := false
    endif
    else
         gRenderer := call SDL_SetRenderDrawColor(arguments: gWindow, SDL_RENDERER_ACCELERATED |DL_RENDERER_PRESENTVSYNC)
return successFlag
```

renderIndicatorToken(TokenLocation *indicatorToken) -- rev 1

When a mouse hovers over the board, a token of a darker color will appear indicating where the token will be dropped when the mouse is clicked.

```
renderIndicatorToken(args: pointer to TokenLocation):
    TextureWrapper pointer token
    if (indicatorToken position is out-of-bounds)
        return
    if (color of indicatorToken == blue)
        token := gBlueToken
    else
        token := gRedToken

    SDL_Rect destRect := position of indicatorToken

    call SDL_SetTextureColorMod(args: texture of token 127,
                                127, 127)
    call SDL_RenderCopy(args: gRenderer, texture of token,
                        null, address of destRect)
    call SDL_SetTextureColorMod(args: texture of token,
                                255, 255, 255)
```

void displaySetupTokens() --rev 0

Determine the position of the blue token and red tokens that are used to switch the colors in game, and render the texture wrapper of each token (gRedToken & gBlueToken) in the back buffer of the window with SDL_RenderCopy.

```
displaySetupTokens(args: none)
    SDL_Rect tokenRect

    tokenRect := location and size of gRedToken
    call SDL_RenderCopy(args: gRenderer, texture of gRedToken,
                        null, address of tokenRect)

    tokenRect := location and size of gBlueToken
    call SDL_RenderCopy(args: gRenderer, texture of gBlueToken,
                        null, address of tokenRect)
```

void setupRender() --rev 0

This function does the rendering for the SETUP state. It displays the board, then renders highlighted tokens if they have not been rendered since the last press of the "Two Player" button, which is determined by checking a boolean value " gRenderHighlighted" that is local to the graphics module. SetupRender also presents draws the texture stored in gRenderer onto the screen (it draws the state of the game in SETUP).

```
setupRender(args: pointer of GraphicsState)

        call SDL_RenderClear(args: gRenderer)
        call displaySetupTokens(args: none)
        call placeImage(args: texture of gSetupScreen, 0,0,
                              screen_width, screen_height)

        if (renderHighlighted)
             call traverseList(args: hightlightTokens, gHighlightedTokens)
        endif
        if (renderInvalidMessage)
              call placeImage(args: texture of gInvalidMessage, xPos,
                                    yPos, width of gInvalidMessage, height
                                    of gInvalidMessage);
        endif

        call SDL_RenderPresent(args: gRenderer)
```

void displayMainMenu() --rev 0

Render the main menu texture wrapper (gMainMenu) to the back buffer of the window with SDL_RenderCopy.

```
displayMainMenu(args: none)
        call SDL_RenderCopy(args: gRenderer, texture of gMainMenu,
                                  null, null)
```

void displayBoard() --rev 0

Determine position of where the board should be located, and render the texture wrapper of the board (gConnect4Board) in the back buffer of the window with SDL_RenderCopy.

```
displayBoard(args: none)
        SLD_Rect DestR

        x-coordinate of DestR := GRID_OFFSET_X - 1
        y-coordinate of DestR := GRID_OFFSET_Y - 1
        width of DestR := width of gConnect4Board
        height of DestR := height of gConnect4Board

        call SDL_RenderCopy( args: gRenderer, texture of
                                  gConnect4Board, NULL, address of DestR )
```

**void placeImage(SDL_Texture *image, int x, int y, int width,
int height)**

--rev 1

A function will render a texture, given the width and height of the image, and the coordinate points (x,y) of where the image should be placed.

```
placeImage(args: pointer of SDL_Texture, integer x, integer y,
                integer width, integer height)

        SDL_Rect  destRect

        x position of destRect := x
        y position of destRect := y
        width of destRect := width
        height of destRect := height

        call SDL_RenderCopy(args: gRenderer, image, null, address of
                                      destRect)
```

void twoPlayerRender(GraphicsState *graphicState) --rev 1

A function will render a texture, given the width and height of the image, and the coordinate points (x,y) of where the image should be placed.

```
twoPlayerRender(args: pointer of GraphicsState)
    call SDL_RenderClear(args: gRenderer)

    call placeImage(args: texture of gTwoPlayerScreen, 0, 0,
                    screen width, screen height)

    call traverseList(args: drawFallingToken, gFalingToken)

    // this process repeats for every possible graphicState instance
    if ( instance of graphicsState is true)
        call placeImage(args:  texture of image, x,y, width,height)
    endif

    call displayBoard(args: none)
    call SDL_RenderPresent(args: gRenderer)
```

bool dropToken(Board b, Token tokenColour, int col) --rev 0

Before dropping the token in a column, we must first check that the column is not full. If the drop is allowed to be made, create a new FallingToken pointer, and declare all the values required by the FallingToken struct. Then the falling token can then be inserted in the list gFallingTokens, where it can be accessed for gravity simulation.

```
dropToken(args: Board b, Token tokenColour, integer col)
    integer row := call board_dropPosition(args: b, col)

    if (row == -1)
        return false

    declare new FallingToken pointer: newToken
    x position of newToken = GRID_OFFSET_X + TOKEN_WIDTH * col
    y position of newToken = GRID_OFFSET_Y
    FallingToken currentHighert := call reduceList(args: compareXPosition,
                                                   newToken, gFallingTokens)
    gFallingTokens := call addToList(newToken, gFallingTokens)
    return true
```

void clearFallingToken(FallingToken *fallingToken) --rev 0

This function just overwrites a fallingToken's previous position with the background. This is to save rendering the whole background each frame (we just erase where the token WAS before it dropped a frame's distance further).

```
clearFallingToken(args: fallingToken)
    SDL_Rect tokenRect

    x position tokenRect = x position of fallingToken
    y position tokenRect = y position of fallingToken
    width of tokenRect = TOKEN_WIDTH
    height of tokenRect = TOKEN_HEIGHT

    call SDL_RenderFillRect(args: gRenderer, address of
                                  tokenRect)
```

void setHighlightedTokenList(List<TokenLocation> *highlightedTokenList, GraphicsState *graphicsState) -- rev 0

This function takes a List of TokenLocations (row, column and colour) and turns the data structure gHighlightedTokens into that List by first free'ing the old list, then setting gHighlightedTokens to the head of highlightedTokenList. This function also sets the global variable gRenderHighlighted to indicate to setupRender that the highlighted tokens must be rendered on the next frame flip.

```
setHighlightedTokenList(args: list highlightedTokenList, graphicState)
    call traverseList(args: freeTokenLocation, gHighlightedTokens)
    gHighlightedTokens := highlightedTokenList
    renderHighlighted := true
```

Deleted in Revision	Declarations Deleted
1	gRenderHighlighted --rev0
1	*gOnePlayerButton --rev0
1	*gTwoPlayerButton --rev0
1	*gMenuButton --rev0
1	transitionSetupRender -- rev 0
1	deleteStillToken -- rev 0

Module: sdl2_connect4

Contains the main game loop to run the Connect 4 program. All event handling from SDL_Events are done in the sdl2_connect4 module (e.g. mouse clicks/motion). Additionally, all transitions states, which perform some sort of setup or clean up between "official" states are here.

--- Interface ---

Uses:

None

Defined Macros/Constants:

None

Defined Types:

None

Access programs:

connect4: --rev 0

Return Type: integer

Parameters: none

Returns 0 for success. Plans to potentially return values other than 0 for different error codes. Runs the Connect 4 game.

--- Implementation ---

Uses:

board.h --rev 0

graphics.h --rev 0

gameLogic.h --rev 0

stdio.h (C runtime library) --rev 0

SDL.h (secret hidden by this module – event handling/graphics library) --rev 0

Type Definitions/Structure,Union,Enumeration Declarations:

None

Variables:

The three arrays: --rev 0.1

```
static void (*handleEvents[NUMBER_OF_STATES])(GameState *gameState) =
{mainMenuHandleEvents, handleEventsStub, twoPlayerHandleEvents,
setupHandleEvents, creditsHandleEvents, handleEventsStub,
handleEventsStub}
```

```
static void (*logic[NUMBER_OF_STATES])(GameState *gameState) =
{mainMenuLogic, logicStub, twoPlayerLogic, setupLogic, logicStub, logicStub,
logicStub}
```

```
static void (*render[NUMBER_OF_STATES])(GraphicsState *graphicsState) =
{mainMenuRender, renderStub, twoPlayerRender, setupRender, creditsRender,
renderStub, renderStub};
```

are array variables hidden in the sdl2_connect4 implementation, which contain function pointers returning type void and taking one argument of type GameState. These arrays contain the different functionality that should happen when the game is in a different state. For example, if the current state were MAINMENU, then the render[] array would be indexed to a function that renders the main menu background.

Local Programs:

static void logicStub(GameState *gameState) {} --rev 0.1
static void handleEventsStub(GameState *gameState) {} --rev 0
static void renderStub(GameState *gameState) {} --rev0.1

These three function are empty stub functions, used in the handleEvents[], logic[] and render[] arrays when we are in a state that doesn't, for example, do any logic. E.g. the MAINMENU state indexes the logic[] array to logicStub().

bool pointInsideRect(int x, int y, Rectangle rect) -- rev 1

This function checks if the position (x,y) is in the rectangle rect and returns true if it is, false otherwise.

```
if ((x >= rect.topLeft.x) && (y >= rect.topLeft.y) &&
    (x <= rect.bottomRight.x) && (y <= rect.bottomRight.y)) {
  return true;
} else {
  return false;
}
```

int square(int x) --rev 1

This is a helper function that returns the square of its input

```
        return x*x
```

```
// NOTE(brendan): checks if a click landed in the circle
```

bool pointInsideCircle(int x, int y, Circle circle) --rev 1

This function checks if the position (x,y) is in the circle, circle, and returns true if it is, false otherwise.

```
        int fromCircleCenterX = x - circle.center.x;
        int fromCircleCenterY = y - circle.center.y;
        if (square(circle.radius) >= square(fromCircleCenterX) +
square(fromCircleCenterY)) {                        return true;
        }
                return false;
```

Player choosePlayer() --rev 1

This function returns PLAYERTWO or PLAYERONE randomly

```
        return (rand() % 2) ? PLAYERONE : PLAYERTWO;
```

Token chooseToken() --rev 1

This function returns RED or BLUE randomly

```
        return (rand() % 2) ? RED : BLUE;
```

void transitionSetupTwoPlayer(GameState *gameState) --rev 1

This is the transition "state" from setup to twoplayer . It randomly chooses a player, player one or player two; if the currentToken state is RANDOMTOKEN then a token will be chosen randomly too; next the colour of the indicator token, which displays were the next token will be dropped, is set to the value of the currentToken state; finally, the logic function pointer, transitionSetupTwoPlayer, which currently corresponds to the TWOPLAYER state is replaced by the function pointer twoPlayerLogic.

```
        gameState->currentPlayer = choosePlayer();
        if(gameState->currentToken == RANDOMTOKEN) {
                gameState->currentToken = chooseToken();
        }
        gameState->graphicsState.indicatorToken.colour = gameState->currentToken;
        logic[TWOPLAYER] = twoPlayerLogic;
```

void transitionSetupMainMenu(GameState *gameState) --rev 1

This is the transition "state" from setup to mainmenu . Empty the board in the gameState, empty the list of falling tokens, reset the graphics state, set the currentProgress of the game in gameState to INPROGRESS, the logic function pointer, transitionSetupMainMenu, which currently corresponds to the MAINMENU state is replaced by the function pointer mainMenuLogic.

```
board_empty(gameState->board);
List<FallingToken>::emptyList(&gFallingTokens);
resetGraphicsState(&gameState->graphicsState);
gameState->currentProgress = INPROGRESS;
logic[MAINMENU] = mainMenuLogic;
```

void transitionMainMenuSetup(GameState *gameState) --rev 1

This is the transition “state” from MAINMENU to SETUP. Set the row and column of the indicator token to -1 (an invalid state), reset the graphics state. The logic function pointer, transitionMainMenuSetup, which currently corresponds to the SETUP state is replaced by the function pointer setupLogic.

```
gameState->graphicsState.indicatorToken.row = -1;
gameState->graphicsState.indicatorToken.column = -1;
resetGraphicsState(&gameState->graphicsState);
logic[SETUP] = setupLogic;
```

void transitionMainMenuTwoPlayer(GameState *gameState) --rev 1

This is the transition “state” from MAINMENU to TWOPLAYER. The currentPlayer is randomly chosen, the currentToken is randomly chosen, the row and column of the indicator token are set to -1 (an invalid state), the graphics are reset. The logic function pointer, transitionMainMenuTwoPlayer, which currently corresponds to the TWOPLAYER state is replaced by the function pointer twoPlayerLogic.

```
gameState->currentPlayer = choosePlayer();
gameState->currentToken = chooseToken();
gameState->graphicsState.indicatorToken.row = -1;
gameState->graphicsState.indicatorToken.column = -1;
resetGraphicsState(&gameState->graphicsState);
gameState->graphicsState.indicatorToken.colour = gameState->currentToken;
logic[TWOPLAYER] = twoPlayerLogic;
```

void transitionTwoPlayerMainMenu(GameState *gameState) --rev 1

This is the transition “state” from TWOPLAYER to MAINMENU. Empty the board in the gameState, empty the list of falling tokens, reset the graphics state, set the currentProgress of the game in gameState to INPROGRESS, the logic function pointer, transitionTwoPlayerMainMenu, which currently corresponds to the MAINMENU state is replaced by the function pointer mainMenuLogic.

```
board_empty(gameState->board);
List<FallingToken>::emptyList(&gFallingTokens);
```

```
        resetGraphicsState(&gameState->graphicsState);
        gameState->currentProgress = INPROGRESS;
        logic[MAINMENU] = mainMenuLogic;
```

static MenuState handleMainMenuMouseClick(int x, int y, GameState *gameState) -- rev 0.1

This function takes a point (x, y) and the gameState and checks if any of the buttons on the main menu were clicked on and returns the state accordingly. Additionally, the logic function pointer will be replaced by the appropriate transition function.

```
        if (pointInsideRect(x, y, MAINMENU_TWOPLAYER_BUTTON_RECT)) {
                logic[TWOPLAYER] = transitionMainMenuTwoPlayer;
                return TWOPLAYER;
        }
        if (pointInsideRect(x, y, MAINMENU_SETUP_BUTTON_RECT)) {
                logic[SETUP] = transitionMainMenuSetup;
                return SETUP;
        }
        if (pointInsideRect(x, y, MAINMENU_QUIT_BUTTON_RECT)) {
                return QUIT;
        }
        if (pointInsideRect(x, y, MAINMENU_CREDIT_BUTTON_RECT)) {
                return CREDITS;
        }
        if (pointInsideRect(x, y, MAINMENU_LOADGAME_BUTTON_RECT)) {
                gameState->loadGame = true;
        }
        return MAINMENU;
```

static MenuState handleCreditsMenuMouseClick(int x, int y) --rev 1

This function takes a point (x, y) and checks to see if the menu button was clicked on, if it was clicked on return the MAINMENU state, otherwise return the CREDITS state.

```
        if (pointInsideRect(x, y, CREDITS_QUIT_BUTTON_RECT)) {
                return MAINMENU;
        }
        return CREDITS;
```

static MenuState handleSetupMouseClick(int x, int y, GameState *gameState) --rev 0.1

This function takes a point (x, y) and the gameState and checks if any of the buttons outside the board in setup are clicked on and performs the desired action. The given options and actions are explicitly given below; the buttons are: refresh, two player, save, the red selector token, the blue selector token, and menu.

```
    if (pointInsideRect(x, y, REFRESH_BUTTON_RECT)) {
        List<FallingToken>::emptyList(&gFallingTokens);
        resetGraphicsState(&gameState->graphicsState);
        gameState->currentState = SETUP;
        gameState->currentProgress = INPROGRESS;
        board_empty(gameState->board);
    }
    if (pointInsideRect(x, y, SETUP_2PLAYER_BUTTON_RECT)) {
        if (readyToTransitionSetupTwoPlayer(gameState)) {
            logic[TWOPLAYER] = transitionSetupTwoPlayer;
            return TWOPLAYER;
        }
    }
    if (pointInsideRect(x, y, SAVE_BUTTON_RECT)) {
        gameState->saveGame = true;
    }
    if (pointInsideCircle(x, y, SETUP_RED_CLICKY_TOKENS_CIRCLE)) {
        gameState->currentToken = RED;
    }
    else if (pointInsideCircle(x, y, SETUP_BLUE_CLICKY_TOKENS_CIRCLE)) {
        gameState->currentToken = BLUE;
    }
    else if (pointInsideRect(x, y, SETUP_MENU_BUTTON_RECT)) {
        logic[MAINMENU] = transitionSetupMainMenu;
        return MAINMENU;
    }
        return SETUP;
    }
```

static void mainMenuHandleEvents(GameState *gameState) --rev 0.1

This function processes all user input that occurs during the main menu state. Each call to this fuction processes all events in the event queue, explicitly determining whether the player quit the game in anyway or clicked on any of the buttons.

```
    SDL_Event e;

    int x, y;

    while(SDL_PollEvent(&e) != 0) {
        if (e.type == SDL_QUIT) {
            gameState->currentState = QUIT;
        }
        else if(e.type == SDL_MOUSEBUTTONDOWN &&
        e.button.button == SDL_BUTTON_LEFT) {

            x = e.button.x;
            y = e.button.y;

            gameState->currentState = handleMainMenuMouseClick(x, y,
gameState);
        }
        else {
            //handleMainMenuMouseMotion();
        }
    }

    if (gameState->currentState == SETUP) {
        gameState->currentToken = RED;
    }
```

static void creditsHandleEvents(GameState *gameState) --rev 1

This function processes all user input that occurs during the credits state. Each call to this function processes all events in the event queue determining if the user quit the game in any way or if they clicked on a button.

```
    SDL_Event e;

    int x, y;

    while(SDL_PollEvent(&e) != 0) {
        if (e.type == SDL_QUIT) {
            gameState->currentState = QUIT;
        } else if (e.type == SDL_MOUSEBUTTONDOWN &&
            e.button.button == SDL_BUTTON_LEFT) {

            x = e.button.x;
            y = e.button.y;
            gameState->currentState = handleCreditsMenuMouseClick(x, y);
        } else {
            //handleCreditsMenuMouseMotion();
        }
    }
```

static void handleIndicatorMouseMotion(GameState *gameState) --rev 1

This function will determine the current position (x, y) of the mouse and calculate the row and column that a token would fall to if the user made a left click at the mouses current position; it also sets the colour of the indicator token to the colour of the currentToken.

```
    int x, y;
    int row, col;
    SDL_GetMouseState( &x, &y );
    // NOTE(Zach): If the click was outside the GRID
    if (x <= GRID_OFFSET_X || x >= GRID_OFFSET_X + GRID_WIDTH) return;
    if (y <= GRID_OFFSET_Y || y >= GRID_OFFSET_Y + GRID_HEIGHT) return;
    col = (x - GRID_OFFSET_X)/TOKEN_WIDTH;
    row = board_dropPosition(gameState->board, col);
    if (row != -1) {
        gameState->graphicsState.indicatorToken.row = row;
        gameState->graphicsState.indicatorToken.column = col;
        gameState->graphicsState.indicatorToken.colour = gameState-
>currentToken;
    }
```

static void setupHandleEvents(GameState *gameState) --rev 0.1

This function processes all user input that occurs during the setup state. Each call to this function processes all events in the event queue determining if the user quit the game in any way or if they clicked on a button. Additionally, after checking if any of the buttons were clicked on, any other clicks that are outside the game board area are disregarded. If the board is clicked on the column cliken on is calculated and dropping a token in that column is attempted, if the column was dropped successfullly then it is added to board in the gameState in the appropriate position. Finally, any mouse motion is processed and an indicator token is displayed where the token would fall if the user left-clicked at the current location.

```
SDL_Event e;

while( SDL_PollEvent( &e ) != 0 ) {
  if (e.type == SDL_QUIT) {
    gameState->currentState = QUIT;
  }
  else if (e.type == SDL_MOUSEBUTTONDOWN &&
      e.button.button == SDL_BUTTON_LEFT) {
    int x, y;
    x = e.button.x;
    y = e.button.y;

    gameState->currentState = handleSetupMouseClick(x, y, gameState);

    if (x <= GRID_OFFSET_X || x >= GRID_OFFSET_X + GRID_WIDTH) continue;
    if (y <= GRID_OFFSET_Y || y >= GRID_OFFSET_Y + GRID_HEIGHT) continue;

    int dropColumn = (x - GRID_OFFSET_X)/TOKEN_WIDTH;
    if(dropToken(gameState->board, gameState->currentToken, dropColumn)) {
      board_dropToken(gameState->board, gameState->currentToken, dropColumn);
    }
  } else {
    handleIndicatorMouseMotion(gameState);
  }
}
```

static void switchPlayer(Player *player) --rev 1

This function takes in a pointer to a Player and will switch PLAYERONE to PLAYERTWO and PLAYERTWO to PLAYERONE.

```
if (*player == PLAYERONE) {
  *player = PLAYERTWO;
} else {
  *player = PLAYERONE;
```

static void switchToken(Token *token) -- rev 1

This function take a pointer to a Token and will switch RED to BLUE and BLUE to RED.

```
 if (*token == RED) {
   *token = BLUE;
 } else {
   *token = RED;
 }
```

static MenuState twoPlayerHandleMouseClick(int x, int y, GameState *gameState) --rev 1

This function takes a point (x, y) and the gameState and checks if any of the buttons outside the board in twoPlayer are clicked on and performs the desired action. The given options and actions are explicitly given below; the buttons are: refresh, save, menu.

```
 if (pointInsideRect(x,y,REFRESH_BUTTON_RECT)) {
   List<FallingToken>::emptyList(&gFallingTokens);
   resetGraphicsState(&gameState->graphicsState);
   gameState->currentState = TWOPLAYER;
   gameState->currentProgress = INPROGRESS;
   board_empty(gameState->board);
 }

 // NOTE(brendan): register save game event
 if (pointInsideRect(x, y, SAVE_BUTTON_RECT)) {
   gameState->saveGame = true;
 }

 if(pointInsideRect(x, y, SETUP_MENU_BUTTON_RECT)) {
          logic[MAINMENU] = transitionTwoPlayerMainMenu;
   return MAINMENU;
 }
 return TWOPLAYER;
```

static void twoPlayerHandleEvents(GameState *gameState) --rev 1

This function processes all user input that occurs during the two player state. Each call to this function processes all events in the event queue determining if the user quit the game in any way or if they clicked on a button. Additionally, after checking if any of the buttons were clicked on, any other clicks that are outside the game board area are disregarded. Additionally, if the game is no longer inprogress (someone won or it was a draw) disregard and input to the board area. If the board is clicked on the column cliken on is calculated and dropping a token in that column is attempted, if the column was dropped successfullly then it is added to board in the gameState in the appropriate position and the player and token are switched. Finally, any mouse motion is processed and an indicator token is displayed where the token would fall if the user left-clicked at the current location.

```
  SDL_Event e;

  while( SDL_PollEvent( &e ) != 0 ) {
    if (e.type == SDL_QUIT) {
      gameState->currentState = QUIT;
    } else if (e.type == SDL_MOUSEBUTTONDOWN &&
        e.button.button == SDL_BUTTON_LEFT) {
      int x, y;
      x = e.button.x;
      y = e.button.y;

      gameState->currentState = twoPlayerHandleMouseClick(x, y, gameState);

      if (gameState->currentProgress != INPROGRESS) continue;

      // NOTE(brendan): if current state changed, click was outside grid area
      // NOTE(Zach): If the click was outside the GRID
      if (x <= GRID_OFFSET_X || x >= GRID_OFFSET_X + GRID_WIDTH) continue;
      if (y <= GRID_OFFSET_Y || y >= GRID_OFFSET_Y + GRID_HEIGHT) continue;

      int dropColumn = (x - GRID_OFFSET_X)/TOKEN_WIDTH;
      // NOTE(brendan): add token to list of falling tokens if valid drop
      if(dropToken(gameState->board, gameState->currentToken, dropColumn)) {
        // NOTE(Zach): Insert the token into the board
        board_dropToken(gameState->board, gameState->currentToken, dropColumn);
        switchPlayer(&gameState->currentPlayer);
        switchToken(&gameState->currentToken);
      }
    } else {
                                 handleIndicatorMouseMotion(gameState);
            }
  }
}
```

int connect4() --rev 0

This function contains the game loop, which actually runs the entire game, controlling timing and calling all event handling functions, as well as physics-updating (logic) functions and rendering functions (graphics) in other modules. First the graphics are initialized by calling init() and loadMedia() from the graphics module. Then a GameState object, which contains the Board, current token colour, and MenuState is created, which lives on the stack (global to the game loop, essentially). The game loop is as follows:

```
// NOTE(brendan): game loop: event handling -> logic -> rendering
while(gameState.currentState != QUIT) {
        currentTime = SDL_GetTicks();
        elapsedTime = currentTime - previousTime;
        previousTime = currentTime;
        // NOTE(Zach): lag is how much the game's time is behind
        // the real world's time
        lag += elapsedTime;

        // NOTE(Zach): handle events that occur in gameState.currentState
        handleEvents[gameState.currentState](&gameState);

        // NOTE(Zach): loop until the game time is up-to-date with
        // the real time
        while (lag >= MS_PER_UPDATE) {
        // NOTE(Zach): update the game logic of gameState.currentState
                logic[gameState.currentState]();
                lag -= MS_PER_UPDATE;
        }

        // NOTE(Zach): render images that occur in gameState.currentState
        render[gameState.currentState]();
}
```

This loop make three important function calls every frame (frame refreshes are sync'ed to the client's monitor refresh rate):

handleEvents[gameState.currentState](&gameState),
logic[gameState.currentState](&gameState)
and render[gameState.currentState](&gameState)

The functions executed by these calls correspond to the game state. The event handling functions are called in this module to handle events, such as mouse clicks. The logic functions are called in the gameLogic module and handle physics updates, such as moving falling tokens along the screen. The rendering functions do the rendering and presenting of textures onto the screen based on the current game state (stored in gameState). Physics updates are done at an interval independent of the frame update by keeping track of the elapsed time since the last physics update, and granularly updating the physics by calling logic[gameState.currentState](&gameState); once for N MS_PER_UPDATE intervals where N*MS_PER_UPDATE <= lag < (N+1)*MS_PER_UPDATE.

After the game loop exits (the gameState.currentState became equal to QUIT), the memory allocated for the board is freed using board_destroy(gameState.board), and the memory allocated for the graphics is freed using close_sdl(). 0 is returned for success after the close_sdl() call.

Module: linked list

A module that implements a generic linked list data structure.

--- Interface ---

Uses:

stdlib.h --rev 0
stdio.h --rev 1

Defined Macros/Constants:

None

Defined Types:

List<T>: A variable with only private members.
A variable of this type is a linked list.

Access programs:

addToList: --rev 0
Return type: List<T> *
Parameters: T *newitem, List<T> *list
Add item T to the list to the front of the list.

deleteFromList: --rev 0
Return Type: List<T> *
Parameters: T *toDeleteItem, List<T> *list
Delete the first occurrence of item T from the list.

traverseList: --rev 0
Return Type: None
Parameters: void (*f)(T *item), List<T> *list
Parameters: void (*f)(T *item), float dt, List<T> *list
Iterate over the list and execute function f on each node of the list.
Note that this is a overloaded function which can use either set of parameters

reduceList: --rev 0
Return type: T *
Parameters: bool (*f)(T *listItem, T *item), T *newest, List<T> *list
Iterate over the list and return the first item T that satisfies the function f

readListFromFile: --rev 1

Return type: List<T> *
Parameters: List<T> *list, FILE *fp
Returns a list from a file referenced by fp.

writeListToFile: --rev 1

Return type: None
Parameters: List<T> *list, FILE *fp
Writes the list to a file referenced by fp.

--- Implementation ---

Uses:

stdlib.h --rev 0
stdio.h   --rev 1

Type Definitions/Structure,Union,Enumeration Declarations:

List –rev 0: The internal representation of a linked list; a class representing our linked list node and containing the functions that can modify the list.

```
class List {
        T *item;
        List<T> *next;

        // private constructor so that clients can't make a LIst
        // object except by using addToList(item, null);
        List() {}
}
```

Variables:

None

Access Programs:

template<typename T> List<T> * List<T>::addToList(T *newItem, List<T> *list)   –rev 0

Adds item T to the front of the linked list, list

```
if(newItem != NULL) {
    List<T> *resultList = (List<T> *)malloc(sizeof(List<T>));
    resultList->item = newItem;
    resultList->next = list;
    return resultList;
  }
  else {
    return list;
  }
```

template<typename T> List<T> * List<T>::deleteFromList(T *toDeleteItem, List<T> *list) --rev 0

Delete the first occurrence of item T from the list, list.

```
  if(list != NULL) {
    List<T> *current;
    List<T> *previous;
    for(current = list, previous = NULL;
        current != NULL && current->item != toDeleteItem;
        previous = current, current = current->next);
    if(current != NULL) {
      if(previous != NULL) {
        previous->next = current->next;
      }
      else {
        list = list->next;
      }
      free(current);
    }
    return list;
  }
  else {
    return NULL;
  }
```

template<typename T> void List<T>::traverseList(void (*f)(T *item), List<T> *list)

template<typename T> void List<T>::traverseList(void (*f)(T *item, float dt), float dt, List<T> *list) –rev 0

Iterate over the list and execute function f on each node of the list.
Note that this is a overloaded function which can use either set of parameters

```
for(List<T> *current = list; current != NULL; current = current->next)
     (*f)(current->item)
```

template<typename T> T * List<T>::reduceList(bool (*f)(T *listItem, T *item), T *newest, List<T> *list) –rev 0

Iterate over the list and return the first item T that satisfies the function f

```
for(; list != NULL; list = list->next) {
   if((*f)(list->item, newest) == true ) {
   return list->item
 }
}
return NULL
```

template<typename T> List<T> * List<T>::readListFromFile(List<T> *list, FILE *fp) --rev 1

Empty the list, list, then read the number of nodes that will be in the list as a system integer to the variable count. Iterate count times and allocate a block of memory that is the size of the item T; read a block of memory that is the size of the item T and store it in the newly allocated block of memory; add the new item to the list and update the head of the list. Finally, return the list.

```
List<T>::emptyList(&list)
fread(&count, sizeof(int), 1, fp)
for(i = 0; i < count; ++i) {
  T *item = (T *)malloc(sizeof(T))
  fread(item, sizeof(T), 1, fp)
  list = List<T>::addToList(item, list)
}
return list
```

template<typename T> void List<T>::writeListToFile(List<T> *list, FILE *fp) –rev 1

Move forward in the file the size of a system integer, initialize a count to zero. Iterate through the list, list, and for each node in the list write a block of memory, that is the size of the item T starting at the address of the item in the node, to the file; increment the count by 1. Move backwards through the file a size of (count*(size of item T) + (size of system integer)). Write the count to a block of memory, the size of a system integer, to the file. Move forward through the file a size of (count*(size of item T)).

```
  fseek(fp, sizeof(int), SEEK_CUR)
  int count = 0
   for(; list != NULL; list = list->next) {
    fwrite(list->item, sizeof(T), 1, fp)
    ++count
   }
  fseek(fp, -(count*sizeof(T) + sizeof(int)), SEEK_CUR)
  fwrite(&count, sizeof(int), 1, fp)
  fseek(fp, count*sizeof(T), SEEK_CUR)
```