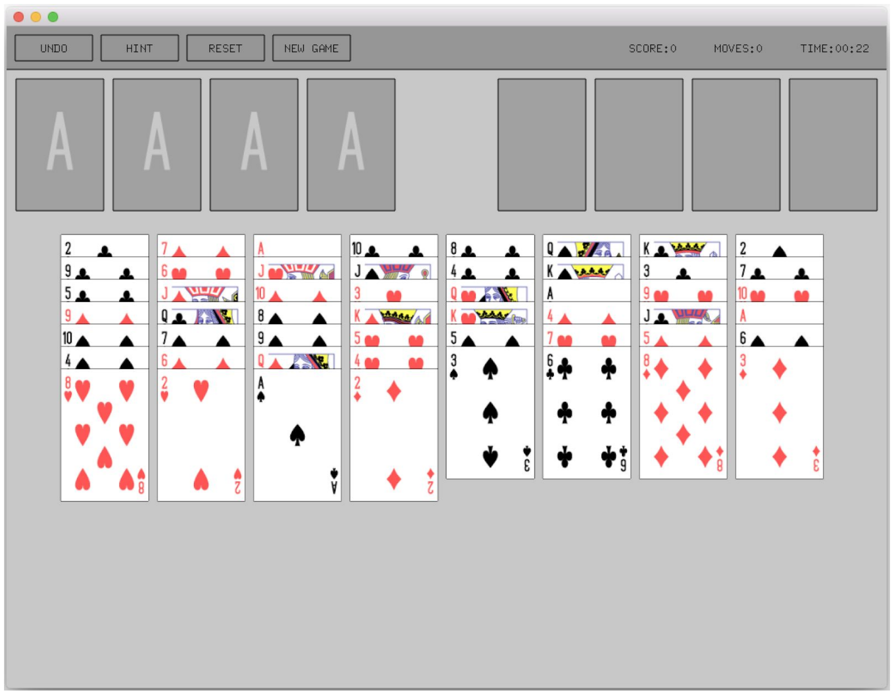
Project report - FreeCell game

By Maria Loks-Thompson



FreeCell game - initial setup

Project's description

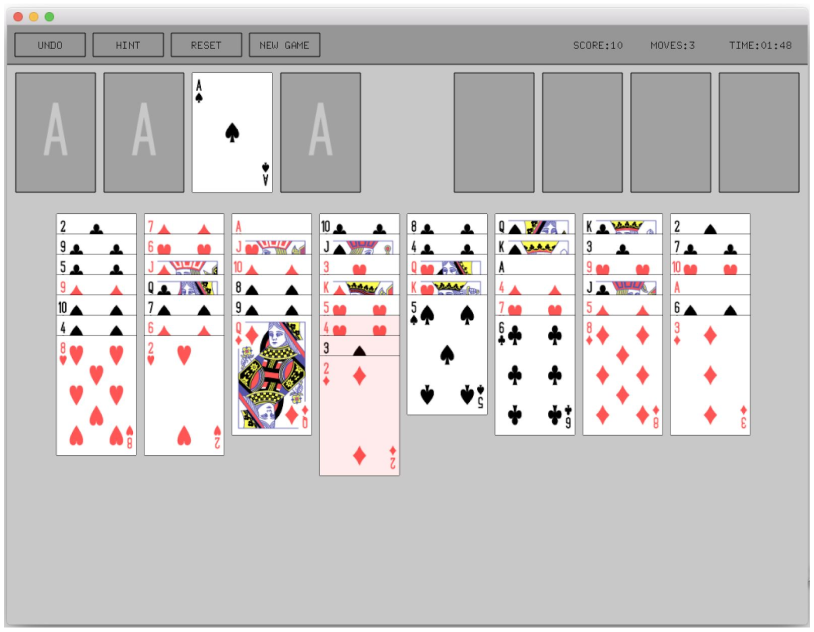
My final project is a version of the classic FreeCell game. The game follows a set of rules that are described in detail at this link: http://www.solitairecity.com/Help/FreeCell.shtml. In addition to implementing basic game logic, I added a few additional functionalities: undo tracks the movements of the cards within the game, so the player can go back one move at the time, hint shows the next possible move, reset lets the user start the game over and new game deals a new random deal. What is more, the game is scored, with the statistics displayed in the top-right corner of the window. Once the game is finished player's score is saved in the separate xml file (done using xmlSettings add-on for openframeworks), and subsequently compared to the best score retrieved from the same file. Another addition is the autocomplete function which keeps checking throughout the game if the game have been solved (that is, are the cards in each column set from the highest to the smallest rank). Once that happens a window pops up asking if the game should be finished using the autocomplete functionality - when the user presses "yes", all the cards will be placed within the home cells and the game will be completed.

It is important to note that this project is different to the one described in the original proposal - this is due to the fact that my earlier idea (and interactive model of a 3d hand) would make little sense executed in openframeworks without the use of shader programming which is beyond the scope of this module.

List of program functionality objectives

Basic Setup

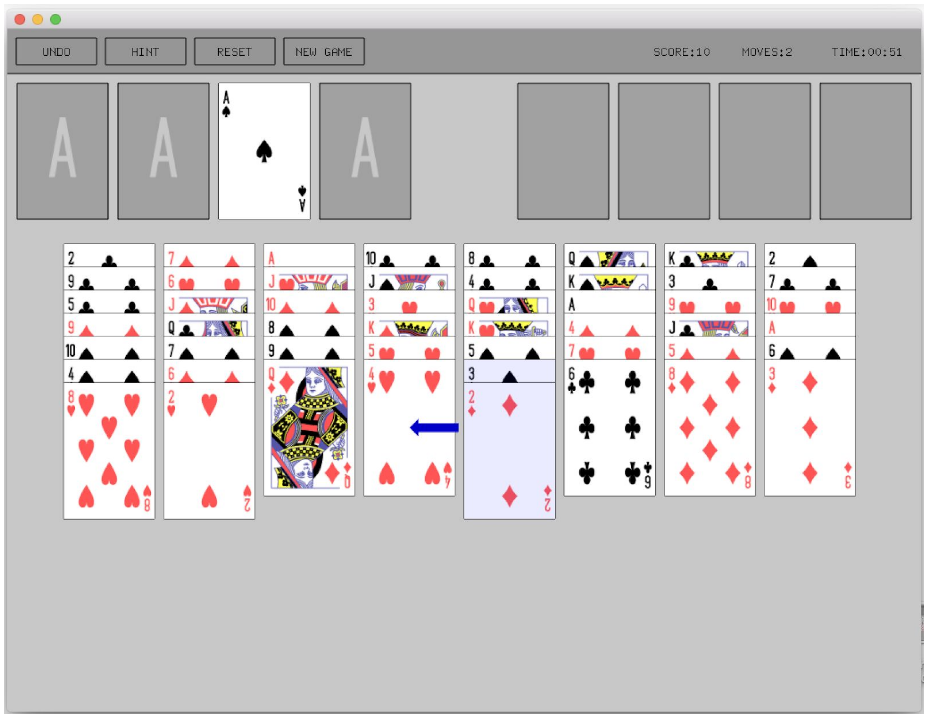
My first objective was to set up a fully functional game, where the player can move the cards accordingly to the rules of FreeCell. The initial layout of the deck of cards was implemented by the use of the algorithm found on the RosettaCode website: https://rosettacode.org/wiki/Deal_cards_for_FreeCell#OOP_version and adapting it in the way that would allow to display graphics. This deal is based on using a single vector to accommodate all card objects. What made the game setup most challenging was the fact that cards needed to be displayed on top of one another, which meant that as soon as the player moved a card, its position within the vector would have to be updated, or the cards within the deck wouldn't be stacked properly. This required a lot of consideration concerning the card's new position within the game (functions updateLocation(), updatePosition(), assessSituation(), findHighestIndex() and findUpdatedIndex() are all a part of the solution for this particular problem). Even though the objective was finally achieved next time I would definitely treat each column within a deck as a separate vector - in this way the cards would be able to move around, simply by popping out of the end of one column and being pushing into the end of another.



FreeCell game - cards are active and ready to be moved

Undo Functionality

The second objective was to implement the undo function, which would allow the user to go one move back each time the designated button was pressed. This proved to be the most challenging problem in the whole program. Each time the player made a move, I decided to store only the information about that move (index of the moving card and the index and type of location the card will be coming back to) instead of the whole state of the game - which would have been significantly easier to do, but required a lot more memory. What made this functionality especially complicated was that it was not only the index of the card to move back that would change (depending on the amount of cards on top of the card to move as well as the destination of the move), but also the index of the location underneath the card's initial position (place where the card would be coming back to) that could be affected. What is more, in case of cards placed within the free cells they could be in there several times with different indices, so those had to be stored as well in a specially assigned vector as a property of each card. Here again separating the tableau into eight distinct vector columns would greatly simplify the task, as it would be enough to store the index of both the columns the cards are moving between and the index of the card to move. Another thing that would make the program easier to write would be to assign separate index number to each free cell (at the moment every time the card moves to a free cell, its index changes to the amount of cards in the deck minus the amount of cards already in home cells). If the deck of cards was treated as separate vectors at each location there would be no need for the vector storing previous free cell indices, as the index of the card in a free cell would always be the same.



FreeCell game - hint functionality

Hint functionality

My final objective was to implement the hint functionality that would allow the user to highlight the next possible move. This is done by checking possible moves for cards placed either within the tableau or free cells. The program runs a check for each case (tableau to home cells, free cells to home cells, free cells to tableau and finally tableau to free cells). If any of the cases are true for any of the cards the checking stops and the possible move gets highlighted, otherwise the program displays “no more possible moves” message. For the cards moving to the free or home cells the program considers only the top cards, and for the cards moving within the tableau the program considers the lowest “clickable” card in each column. This functionality was relatively straightforward to implement (in comparison to aforementioned undo option) and I’m happy with its implementation - I honestly don’t know what would i do differently next time.

Description and documentation of the behaviour of the program

The interaction with the game is based solely on the mousePressed() function. User can either activate or deactivate cards, move activated cards around and finally take advantage of the undo, hint, reset and new game buttons. For more details concerning the program’s structure please consult the pdf files included in the submission folder. Each file contains a diagram breaking down the purpose of each function. Each child function is connected to its parent by the arrow, and children and sibling functions can be distinguished by the level of indentation. If the function is present in the code more than once, its child function would be listed in the diagram only at the moment when it’s present for the first time. Regular arrows indicate functions and tasks that run every time and arrows with a question mark indicate functions that run only if a condition is true. Those conditions are written in cursive in front of the functions. The hope of the diagrams is to make the structure of the program easier to understand.

Description of each class

Deck

The main class within the program is a Deck class which holds the code that allows for the game to run. Here all the calculations regarding card movements are happening as well as the undo and hint functionalities. The public functions from this class are newGame() (connected to the button allows the user to set up a new random game), refresh() (connected to the button allows user to start the same game from the beginning), draw() (draws cards, cells and hint indicator onto the screen), getEnough() (boolean triggering the “not enough space to make a move” announcement), getNoMore() (boolean triggering the “no more possible moves” announcement), getAutocomplete() (boolean triggering the “autocomplete?” dialog window), mousePressed() (triggering all user the interaction within the deck), undo() (connected to the button allows user to undo one move at the time), hint() (connected to the button triggers the hint functionality), skipAutocomplete() (connected to the “yes” button within the “autocomplete?”

dialog window allows user to skip the autocomplete process), doAutocomplete() (connected to the “no” button within the “autocomplete?” dialog window allows user to trigger the autocomplete process), getFinished() (boolean triggering the “congratulations” dialog window which displays all of the game scores), and finally getScore(), getMoves(), getTime() and getSeconds() (which are used to display both the games statistics in the top right corner during the game play and to calculate and save final score).

Button
Another class used in the program is a Button class. It’s a simple class that allows for creating an interface between the player and the functionality within the deck. This class has a draw() (draws button onto the screen), mousePressed() (takes a function from the deck class and triggers it when the button is pressed), and getHover() (determines if the mouse is over the button) functions.

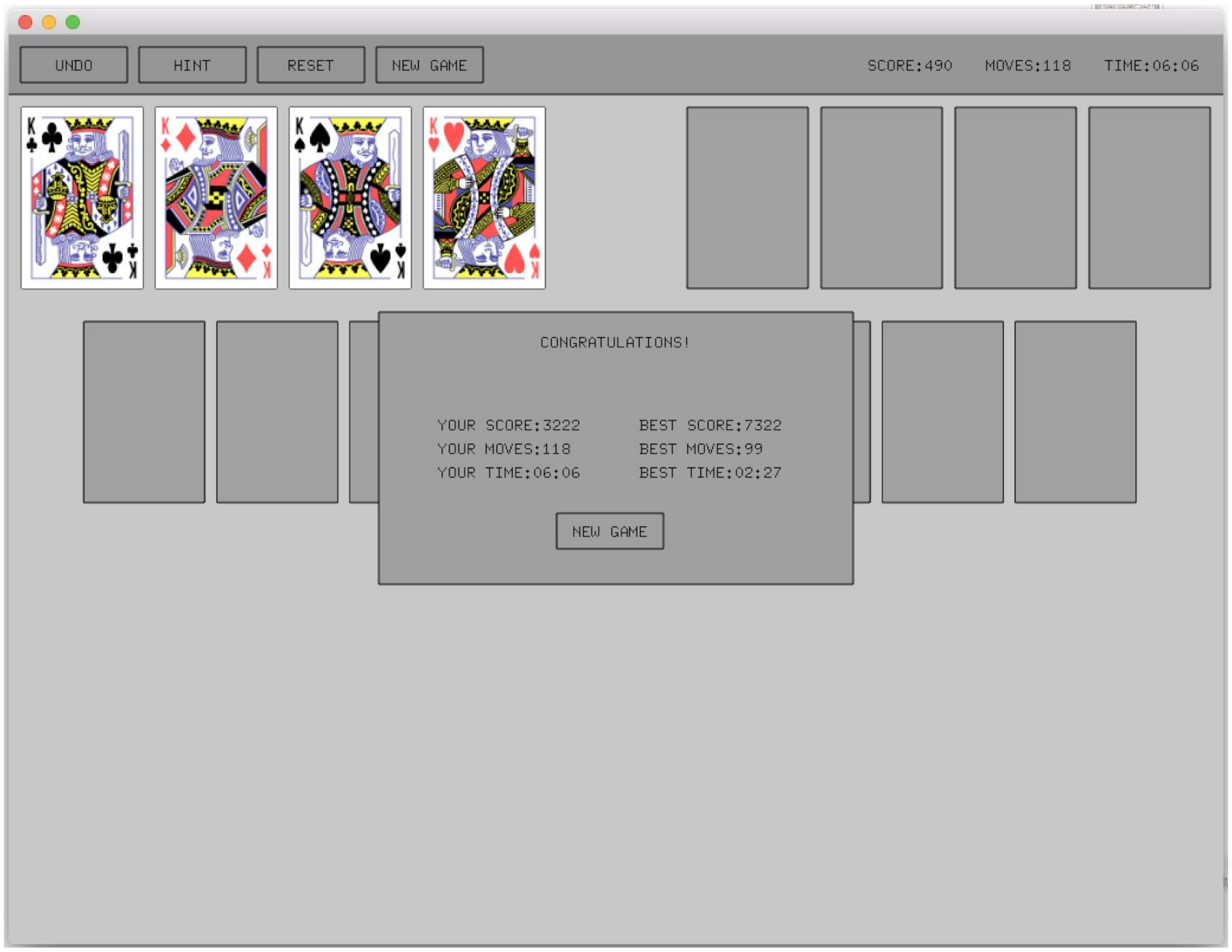
Card
A Card class holds all the information about the card object. Apart from the draw() function (draws a card on the screen at given position, possibly with a blue overlay if the card is suggested as a one to be moved by the hint functionality or a red one if the card is chosen as the card to move (active) or any of the cards on top of it (in case the user attempts to move several cards at once)). The Card class has a serious of set and get functions: set and getHint() (interface for hint property - creates a blue overlay if the hint is true), set and getRow() (sets the row value for each card), set and getOnTop() (determines if a card is on top of the active card), set and getPosition() (used to move cards around the screen), set and getActive() (used to activate card - mark it as the card that is about to be moved), set and getIneractive() (determines if the card can be activated), and set and getTop() (determines if the card is on top of the column). Apart from those there are four functions allowing for the interaction with PrevFcs vector: pushPrevFcs() (pushes card’s index into the vector when a card enters free cell),, getPrevFcs(returns an index stored in a vector when the card gets back to freecell during the undo process) popPrevFcs() (deletes last index from the vector when the card gets back to freecell during the undo process), and getPrevFcsSize() (returns the size of a vector). Finally there are four get functions returning values specific to the card that stay constant throughout the game: getSize (returns width and height of each card), getColour() (returns a boolean determining if the card is red or black), getRank() (returns card’s rank) and getSuit() (returns cards suit).

Pile
The Pile class is a parent class for all the remaining home and regular classes. It combines regular functions (common to the child classes), and virtual ones (either when the implementation differs between the child classes or if the function is specific to one of them). The regular functions are: set and getPosition() (set is used in the constructor of each pile and get is used to move a card onto the pile), set and getSize() (set is used in the constructor and get is used to draw the pile) and set and getOnTop() (used both is the constructor and to determine if the pile has any cards on top of it). The virtual functions are: draw() (draws each

pile onto the screen) and specifically for the Home class: function set and getSuit() (determines which cards can be placed within the home cells - helps to make sure that only cards of one suit will be placed within that particular cell) and set and getCRank() (sets a home cell to expect a card of a particular rank and gets updated to the higher rank once the card is placed within the home cell).

Regular and Home

The Regular class is responsible for the regular piles at the bottom of each column and well as the FreeCell piles. The Home class is specific to the home cells - they differ in both the appearance and behaviour and their public interface was described above.



CONGRATULATIONS!

YOUR SCORE:3222 BEST SCORE:7322
YOUR MOVES:118 BEST MOVES:99
YOUR TIME:06:06 BEST TIME:02:27

FreeCell game - finished dialog window

Code

From main.cpp

```
#include "ofMain.h"
#include "ofApp.h"

//========================================================================
int main( ){
      ofSetupOpenGL(1024,768,OF_WINDOW);              // <-------- setup the
GL context

      // this kicks off the running of my app
      // can be OF_WINDOW or OF_FULLSCREEN
      // pass in width and height too:
      ofRunApp(new ofApp());

}
```

From ofApp.h

```
#pragma once

#include "ofMain.h"
#include "deck.hpp"
#include "button.hpp"
#include "ofxXmlSettings.h"

class ofApp : public ofBaseApp {

      public:
            void setup();
            void draw();
        void drawTopBar();
        void drawDialog(string t, float x, float y, int w, int h);
        void drawAutocomplete();
        void drawFinished();
        void drawArrow(const float & x, const float & y);
        void drawScore(bool c, const string s1, const string s2, const
float x1, const float x2, const float y);
            void mousePressed(int x, int y, int button);
        void saveScore();
        void getBScore();
        void emptyScores();
        Deck d; // game
        //buttons
        unique_ptr<Button> un = make_unique<Button>("UNDO", 10, 10);
        unique_ptr<Button> hi = make_unique<Button>("HINT", 110, 10);
        unique_ptr<Button> re = make_unique<Button>("RESET", 210, 10);
        unique_ptr<Button> ng = make_unique<Button>("NEW GAME", 310, 10);
        unique_ptr<Button> yes = make_unique<Button>("YES",
(ofGetWidth()/2) - 150, (ofGetHeight()/2) - 100);
        unique_ptr<Button> no = make_unique<Button>("NO", (ofGetWidth()/2)
+ 50, (ofGetHeight()/2) - 100);
        unique_ptr<Button> newGame = make_unique<Button>("NEW GAME",
(ofGetWidth()/2) - 50, (ofGetHeight()/2) + 20);
        int timer; // fix for autocomplete
        // scoring
        ofxXmlSettings XML;
        int score;
        bool bScore;
        bool bTime;
        bool bMoves;
        int bestScore;
        int bestSeconds;
        string bestTime;
        int bestMoves;

};
```

From ofApp.cpp

```
#include "ofApp.h"
#include "deck.hpp"

//--------------------------------------------------------------
void ofApp::setup() {
    d.newGame(); // set up a new random game
    timer = 0;
    score = 0;
```

```
}

//--------------------------------------------------------------
void ofApp::draw(){
    d.draw(); // draw game
    ofPushStyle();
    drawTopBar(); // draw top bar
    if(!d.getEnough()) drawDialog("NOT ENOUGH SPACE TO MOVE",
ofGetWidth()/2-150, ofGetHeight()-60, 300, 50); // draw not enough
    if(d.getNoMore()) drawDialog("NO MORE POSSIBLE MOVES",
ofGetWidth()/2-150, ofGetHeight()-60, 300, 50); // draw no more moves
    if(d.getAutocomplete()) drawAutocomplete(); // draw autocomplete dialog
window
    if(d.getFinished()) drawFinished(); // draw game complete dialog window
    ofPopStyle();
}

//--------------------------------------------------------------
void ofApp::drawTopBar(){
    ofSetColor(150);
    ofDrawRectangle(0, 0, ofGetWidth(), 50); // background
    ofSetColor(0);
    ofDrawLine(0, 50, ofGetWidth(), 50); // border
    un->draw(); // undo
    hi->draw(); // hint
    re->draw(); // reset
    ng->draw(); // new game
    ofDrawBitmapString("SCORE:" + ofToString(d.getScore()), ofGetWidth() -
300, 30); // current score
    ofDrawBitmapString("MOVES:" + ofToString(d.getMoves()), ofGetWidth() -
200, 30); // current moves
    ofDrawBitmapString(d.getTime(), ofGetWidth() - 100, 30); // current
time
}

//--------------------------------------------------------------
void ofApp::drawDialog(string t, float x, float y, int w, int h){
    ofPushStyle();
    ofSetColor(160);
    ofFill();
    ofDrawRectangle(x, y, w, h); // background
    ofSetColor(0);
    ofNoFill();
    ofDrawRectangle(x, y, w, h); // border
    int width = t.length() * 8; // text width
    int align = (w - width) / 2; // center align
    ofDrawBitmapString(t, x + align, y + 30); // draw text
    ofPopStyle();
}

//--------------------------------------------------------------
void ofApp::drawAutocomplete(){
    timer++; // prevents accidental clicks
    drawDialog("AUTOCOMPLETE ?", (ofGetWidth()/2) - 200, (ofGetHeight()/2)
- 150, 400, 100); // draw basic dialog window
    yes->draw(); // draw yes button
    no->draw(); // draw no button
}

//--------------------------------------------------------------
void ofApp::drawFinished(){
    drawDialog("CONGRATULATIONS!", (ofGetWidth()/2) - 200,
(ofGetHeight()/2) - 150, 400, 230); // draw basic dialog window
    float yourX = (ofGetWidth()/2) - 150; // your score x position
    float bestX = (ofGetWidth()/2) + 20; // best score x position
    float scoreY = (ofGetHeight()/2) - 50; // first score y position
    ofFill();
    drawScore(bScore, "YOUR SCORE:" + ofToString(score), "BEST SCORE:" +
ofToString(bestScore), yourX, bestX, scoreY);
    drawScore(bMoves, "YOUR MOVES:" + ofToString(d.getMoves()), "BEST
MOVES:" + ofToString(bestMoves), yourX, bestX, scoreY + 20);
    drawScore(bTime, "YOUR " + d.getTime(), "BEST " + bestTime, yourX,
bestX, scoreY + 40);
    newGame->draw(); // draw start a new game button
}

//--------------------------------------------------------------
void ofApp::drawArrow(const float & x, const float & y){
    ofDrawRectangle(x - 15, y - 2, 10, 4);
    ofDrawTriangle(x - 5, y - 5, x - 5, y + 5, x - 2, y);
}

//--------------------------------------------------------------
void ofApp::drawScore(bool c, const string s1, const string s2, const float
x1, const float x2, const float y){
    if(c) drawArrow(x1 - 5, y - 5); // draw arrow next to your score if
beaten best score
    ofDrawBitmapString(s1, x1, y); // draw your score
    ofDrawBitmapString(s2, x2, y); // draw best score
}

//--------------------------------------------------------------
void ofApp::mousePressed(int x, int y, int button){
    d.mousePressed(); // deals with the game [cards and piles]
    if(un->getHover()) un->mousePressed(&Deck::undo, d); // undo button
    if(hi->getHover()) hi->mousePressed(&Deck::hint, d); // hint button
    if(re->getHover()) re->mousePressed(&Deck::refresh, d); // restart
button
    if(ng->getHover()) ng->mousePressed(&Deck::newGame, d); // new game
button
    if(d.getAutocomplete() && timer >= 30) { // if game is solved for half
a second
        if(no->getHover()) no->mousePressed(&Deck::skipAutocomplete, d); //
don't autocomplete button
        if(yes->getHover()) yes->mousePressed(&Deck::doAutocomplete, d); //
autocomplete button
        timer = 0; // reset timer
    }
    if(d.getFinished()) { // if the game is finished
        score = d.getScore() + 1000000 / d.getSeconds();; // calculate
final score
        saveScore(); // save score to xml file
        getBScore(); // get best score from xml and compare it to this game
score
        if(newGame->getHover()) newGame->mousePressed(&Deck::newGame, d);
// new game button
    }
}

//--------------------------------------------------------------
void ofApp::saveScore(){
    XML.loadFile("scoreTable.xml"); // load file
    XML.addTag("entry"); // create new entry
    XML.pushTag("entry", XML.getNumTags("entry") - 1); // push into newly
created entry
    XML.addValue("score", score); // save score
    XML.addValue("seconds", d.getSeconds()); // save time in seconds
    XML.addValue("time", d.getTime()); // save time as string
    XML.addValue("moves", d.getMoves()); // save number of moves
    XML.popTag(); // pop out of the entry
    XML.saveFile("scoreTable.xml"); // save updated settings
}

//--------------------------------------------------------------
void ofApp::getBScore(){
    emptyScores(); // prepare variables
    XML.loadFile("scoreTable.xml"); // load file
    for(int i = 0; i<XML.getNumTags("entry"); i++){ // for all entries
        XML.pushTag("entry", i); // push into each entry
        if(XML.getValue("score", 0) > bestScore) bestScore =
XML.getValue("score", 0); // find best score
        if(XML.getValue("seconds", 0) < bestSeconds) {
            bestSeconds  = XML.getValue("seconds", 0); // find best time in
seconds
            bestTime  = XML.getValue("time", ""); // find best time as
string
        }
        if(XML.getValue("moves", 0) < bestMoves) bestMoves =
XML.getValue("moves", 0); // find best moves
        XML.popTag();
    }
    if(bestScore <= score) bScore = true; // was this the best score?
    if(bestSeconds >= d.getSeconds()) bTime = true; // was this the best
time?
    if(bestMoves >= d.getMoves()) bMoves = true; // were those the best
moves?

}

//--------------------------------------------------------------
void ofApp::emptyScores(){
    bScore = false;
    bTime = false;
    bMoves = false;
    bestScore = 0;
    bestSeconds = 100000;
    bestTime = "";
    bestMoves = 100000;
```

```
}
```

From deck.hpp

```
#include "card.hpp"
#include "pile.hpp"
#include "regular.hpp"
#include "home.hpp"
#include "ofMain.h"


#ifndef deck_hpp
#define deck_hpp


class Deck {
public:
    Deck();
    void newGame();
    void refresh();
    void draw();
    bool getEnough();
    bool getNoMore();
    bool getAutocomplete();
    void mousePressed();
    void undo();
    void hint();
    void skipAutocomplete();
    void doAutocomplete();
    bool getFinished();
    int getMoves();
    int getScore();
    string getTime();
    int getSeconds();
private:
    // typedefs
    typedef pair<int,int> pint;
    template<class Piles>
    using pil = vector<shared_ptr<Piles>>;
    // game state variables
    int deckID;
    bool un;
    bool act;
    int cAtHome;
    int moves;
    bool hin;
    ofVec2f hintPos;
    bool autocomplete;
    bool dontAutocomplete;
    bool finished;
    string time = "";
    int sec;
    int score;
    bool enough;
    bool noMore;
    // vectors
    pil<Card> cards; // cards
    pil<Pile> regs; // regular cells
    pil<Pile> fcells; // free cells
    pil<Pile> homes; // home cells
    vector<array<int,4>> history; // undo vector
    // setup
    void arrangeCards(int GI);
    int RNG(int seed);
    void makePretty();
    void setupInteractivity();
    void checkForInteractive(const int &idx);
    void setupPiles();
    void measureTime();
    void drawHint();
    void deactivateStates();
    bool canActivate();
    template <class Piles>
    int find(const pil<Piles> &vec, const int target);
    template <class Piles>
    bool clicked(const pil<Piles> &vec, const int &i);
    void activateCard(const int &idx);
    pint findActive();
    template <class Piles>
    bool check(const pil<Piles> &vec, const pint &ac, const int target);
    bool anotherCard(const int &np, const pint & ac);
    bool enoughSpace(const int &onTop, bool reg);
    bool checkHomes(const pint &ac, const int &newPos);
    void moveCard(const pint &ac, const pair <int,int> &np);
    void occupyHistory(const pint &ac, const pint &np);
    void occupyNp(const pint &np);
    void fcHistory(const pint &np, const int &active);
    template <class Piles>
    int freePile(const pil<Piles> &vec, const int &active);
    void newHistoryEntry(const int &np, const int type, const int adj);
    void regHistory(const pint &np, const pint &ac);
    template <class Piles>
    void historyBiggerRow(const pil<Piles> &vec, const int &idx, const pint
&np, const int &onTop, const int &row);
    void cardHistory(const pint &np, const pint &ac);
    int freeCard(const int &active);
    void historyCardToCard(const int &idx, const pint &np, const int
&onTop);
    template <class Piles>
    void historySmallerRow(const pil<Piles> &vec, const int &idx, const
pint &np, const int &onTop, const int &row);
    int updateLocation(const pint &ac, const pint &np);
    int updatePosition(const pint &ac, const pint &np);
    int assesSituation(const int &nr, const int &activeC);
    int findUpdatedIndex(const int &active, const int &row);
    int findHighestIndex(const int &row);
    void reaarange(const int &row, const int &indexToMove, const int
&newIndex);
    void newActiveHistory(const int &newA, const int &target);
    void moveCardsOnTop(const int & onTop, int & newA);
    void checkAutocomplete();
    bool autocompleteColumn(int idx);
    int autocompleteCard(const int & idx);
    void checkFinished();
    void deactivate(const int &ac);
    void deactivateAllCards();
    void undoHome(const int &idx);
    bool checkHint(const bool fc, const int target);
    int hintFindIdx(const bool & fc, const int & sourceIdx, const int &
target);
    template <class Piles>
    bool hintFindTarget(const pil<Piles> &vec, const int &idx, const int
&target);
    void setHint(const int &idx, const int &targetIdx, const int &target);
    int autoFindIdx(const int &i);
};

#endif /* deck_hpp */

```

From deck.cpp

```
#include "deck.hpp"

#define NUMBER_OF_CARDS 52
#define NUMBER_OF_COLUMNS 8
#define SPACING 26
#define TOP 50

//--------------------------------------------------------------
Deck::Deck() {}

//--------------------------------------------------------------
void Deck::newGame() {
    deckID = ofRandom(32000); // assign deck id
    refresh(); // set up the new game
}

//--------------------------------------------------------------
void Deck::refresh() {
    arrangeCards(deckID); // deal new deck
    makePretty(); // set up positions of each card
    setupInteractivity(); // set up interactive and top
    setupPiles(); // set up Free, Home and Regular cells
    act = false; // state is not active
    cAtHome = 0; // there is no cards at home
    un = false; // undo is not happening
    hin = false; // hint is not happening
    autocomplete = false; // autocomplete is not happening
    dontAutocomplete = true; // autocomplete is not happening
    finished = false; // game is not finished
    enough = true; // enough space anoucement is off
    noMore = false; // no more moves anoucement is off
    moves = 0; // no moves have been taken
    score = 0; // score is 0
    sec = 0; // game lasted 0 seconds
    ofResetElapsedTimeCounter(); // reset time
}

//--------------------------------------------------------------
void Deck::arrangeCards(int GI) { // partially from
https://rosettacode.org/wiki/Deal_cards_for_FreeCell#OOP_version
    if(cards.size() != 0) cards.clear(); // empty vector in case of
reseting or starting new the game
    for (int i = 0; i < NUMBER_OF_CARDS; i++) { // for each card
        shared_ptr<Card> c (new Card((NUMBER_OF_CARDS - 1) - i)); // create
card
        cards.push_back(move(c)); // push it into the vector
    }
    for (int i = 0; i < (cards.size() - 1); i++) { // for each card
        int j = (cards.size() - 1) - RNG(GI) % (cards.size() - i); //
choose card to swap with
        swap(cards[i], cards[j]); // swap cards
    }
}

//--------------------------------------------------------------
int Deck::RNG(int seed) { // from
https://rosettacode.org/wiki/Deal_cards_for_FreeCell#OOP_version
    return (seed = (seed * 214013+2531011) & (1U << 31) - 1) >> 16; //
generate random number based on the seed
}

//--------------------------------------------------------------
void Deck::makePretty() {
    int y = 0; // initialise y position of the card
    int row = -1; // initialise row position
    float cardSpace = cards[0]->getSize().x * 1.1; // card with horizontal
spacing
    float margins = (ofGetWidth() - NUMBER_OF_COLUMNS * cardSpace) / 2; //
calc deck's distance from the left
    for(int i = 0; i < cards.size(); i++) { // for each card
        if(i % NUMBER_OF_COLUMNS == 0) { // for each row
            if(i != 0) y += SPACING; // increase y position
            row += 1; // increase row index
        }
        float xPos = margins + (i % NUMBER_OF_COLUMNS * cardSpace); // cals
card's x position
        float yPos = y + ofGetHeight()/4; // cals card's y position
        cards[i]->setPosition(ofVec2f(xPos,TOP + yPos));  // set card's
position
        cards[i]->setRow(row); // set row index
    }
}

//--------------------------------------------------------------
void Deck::setupInteractivity() {
    for(int i = 0; i < cards.size(); i++) {
        if(i <= cards.size() - 1 && i > cards.size() - (NUMBER_OF_COLUMNS +
1)) { // cards on top of each column
            cards[i]->setInteractive(1); // are clickable
            cards[i]->setTop(1); // and are on top of each column
            checkForInteractive(i);
        } else { // have cards on top
            cards[i]->setInteractive(0); // are not clickable
            cards[i]->setTop(0); // and have cards on top
        }
    }
}

//--------------------------------------------------------------
void Deck::checkForInteractive(const int & idx) {
    for(int i = 0; i < cards.size(); i++) {
        if(cards[i]->getPosition().y == cards[idx]->getPosition().y -
SPACING && // if the card is directly above
           cards[i]->getPosition().x == cards[idx]->getPosition().x && //
and in the same column
           cards[i]->getRank() == cards[idx]->getRank() + 1 && // and has a
rank one higher
           cards[i]->getColour() != cards[idx]->getColour()) { // and
different colour
            cards[i]->setInteractive(1); // make it interactive
            checkForInteractive(i); // and check the card above
        }
    }
}

//--------------------------------------------------------------
```

```
void Deck::setupPiles() {
    if(regs.size() != 0) regs.clear(); // empty vector in case of reseting
or starting new the game
    if(homes.size() != 0) homes.clear(); // empty vector in case of
reseting or starting new the game
    if(fcells.size() != 0) fcells.clear(); // empty vector in case of
reseting or starting new the game
    float width = ofGetWidth()/9; // top pile width with horizontal spacing
    float spaceH = width - cards[0]->getSize().x; // gap between two piles
    float gap = ofGetWidth() - (spaceH + NUMBER_OF_COLUMNS/2 * width); //
starting point of the second pile
    float yPos = TOP + spaceH; // y position of piles
    for(int i = 0; i < 4; i++) {
        float xPos = spaceH + i * width; // x position of each pile
        shared_ptr<Pile> h (new Home(ofVec2f(xPos, yPos),
cards[i]->getSize(), 0)); // create home piles
        shared_ptr<Pile> f (new Regular(ofVec2f(gap + xPos, yPos),
cards[i]->getSize(), 0)); // create fc piles
        homes.push_back(move(h)); // create vector of homes
        fcells.push_back(move(f)); // create vector of fcs
    }
    for(int i = 0; i < NUMBER_OF_COLUMNS; i++) { // for the first row
        shared_ptr<Pile> r (new
Regular((cards[i]->getPosition()),cards[i]->getSize(), 1)); // create regs
        regs.push_back(move(r)); // create vector of regs
    }
}

//--------------------------------------------------------------
void Deck::draw() {
    for(int i = 0; i < homes.size(); i++) homes[i]->draw();
    for(int i = 0; i < fcells.size(); i++) fcells[i]->draw();
    for(int i = 0; i < regs.size(); i++) regs[i]->draw();
    for(int i = 0; i < cards.size(); i++) cards[i]->draw();
    if(!finished) measureTime();
    if(hin) drawHint(); // highlights a location where the card could be
moved to
}

//--------------------------------------------------------------
void Deck::measureTime() {
    int s = (ofGetElapsedTimeMillis()/1000) % 60; // calc seconds
    int m = (ofGetElapsedTimeMillis()/60000); // calc minutes
    string minutes = "";
    string seconds = "";
    if (s < 10) seconds = "0" + ofToString(s); // add 0 for more elegant
look
    else seconds = ofToString(s);
    if (m < 10) minutes = "0" + ofToString(m); // add 0 for more elegant
look
    else minutes = ofToString(m);
    time = "TIME:" + minutes + ":" + seconds; // time in string format
    sec = ofGetElapsedTimeMillis()/1000; // time in int (seconds) format
for easier comparison with best score
}

//----------------------------------------------------------------
bool Deck::getEnough() {
    return enough;
}

//----------------------------------------------------------------
bool Deck::getNoMore() {
    return noMore;
}

//----------------------------------------------------------------
bool Deck::getAutocomplete() {
    return autocomplete;
}

//----------------------------------------------------------------
void Deck::drawHint() {
    // draws a blue arrow
    ofPushStyle();
    ofSetColor(0, 0, 200);
    ofDrawTriangle(hintPos.x, hintPos.y, hintPos.x + 10, hintPos.y + 10,
hintPos.x + 10, hintPos.y - 10);
    ofDrawRectangle(hintPos.x + 5, hintPos.y - 5, 50, 10);
    ofPopStyle();
}

//----------------------------------------------------------------
void Deck::mousePressed() {
    deactivateStates(); // deactivate possible annoucements
    if(!autocomplete) {
        if(!act){ // if state passive
            if(canActivate()) act = true; // activate pressed card
        } else { // id state active
            pint active = findActive(); // find active card and the amount
of cards on top of it
            // check which category card is moving to and and if the click
was ok perform further actions
            if(check(cards, active, 0) || check(regs, active, 1) ||
check(fcells, active, 2) || check(homes, active, 3)) {
                moves++; // count the moves
                if(dontAutocomplete) checkAutocomplete();
                checkFinished();
            }
            else deactivate(active.first); // unfortunate click
            act = false; // finish and deactivate state
        }
    }
}

//----------------------------------------------------------------
void Deck::deactivateStates() {
    if(hin) { // deactivate hint
        for (int i = 0; i < cards.size(); i++) if (cards[i]->getHint())
cards[i]->setHint(0);
        hin = false;
    }
    if(noMore) noMore = false; // deactivate no more possible moves
    if(!enough) enough = true; // deactivate not enough space
}

//----------------------------------------------------------------
bool Deck::canActivate() {
    int idx = find(cards, 0); // find card's index
    if (idx != -1) { // if index found
        if(cards[idx]->getInteractive() && cards[idx]->getRow() != -11) {
// only allow for interactive cards that are not at home
            activateCard(idx);
            return true; // success
        } else {
            return false; // fail
        }
    }
}

//--------------------------------------------------------------
template <class Piles>
int Deck::find(const pil<Piles> &vec, const int target) {
    int idx = -1; // initialise index
    for(int i = 0; i < vec.size(); i++) { // find base pile
        if (clicked(vec, i) && i > idx)  { // if position was clicked
            switch(target) { // depending on the situation
                case 0: idx = i; break; // find a card to activate
                case 1: if (cards[i]->getTop() && cards[i]->getRow() >= 0)
idx = i; break; // find a top card to move to
                case 2: if (!vec[i]->getOnTop()) idx = i; break; // find a
pile to move to (can't have any cards already there)
            }
        }
    }
    return idx; // return its index
}

//--------------------------------------------------------------
template <class Piles>
bool Deck::clicked(const pil<Piles> &vec, const int &i) {
    return ofGetMouseX() >= vec[i]->getPosition().x &&
    ofGetMouseX() <= vec[i]->getPosition().x + cards[0]->getSize().x &&
    ofGetMouseY() >= vec[i]->getPosition().y &&
    ofGetMouseY() <= vec[i]->getPosition().y + cards[0]->getSize().y;
}

//--------------------------------------------------------------
void Deck::activateCard(const int & idx) {
    cards[idx]->setActive(1); // activate the card
    for(int i = 0; i<cards.size(); i++)
        if(cards[i]->getPosition().x == cards[idx]->getPosition().x && i >
idx)
            cards[i]->setOnTop(true); // activate all cards on top of it
}

//--------------------------------------------------------------
pair<int,int> Deck::findActive() {
    pint idx; // create an int, int pair
    for(int i = 0; i < cards.size(); i++){
        if(cards[i]->getActive()) idx.first = i; // find active card index
        if(cards[i]->getOnTop()) idx.second ++; // count cards on top
    }
    return idx;
}

//----------------------------------------------------------------
template <class Piles>
bool Deck::check(const pil<Piles> &vec, const pint & ac, const int target)
{
    pint newPos(-1, target); // set up new position pair
    newPos.first = (target == 0) ? find(vec, 1) : find(vec, 2); // moves to
card or to pile
    if (newPos.first != -1) { // if new position found and moving to
another card
        bool condition;
        switch(target) {
            case 0: condition = anotherCard(newPos.first, ac); break; //
new pos is a card chceck if possible to move
            case 1: condition = enoughSpace(ac.second,1); break; // new pos
is a regular check if there is enough space to move it
            case 2: condition = cards[ac.first]->getTop(); break; // new
pos is a free cell only allowed for top cards
            case 3: condition = checkHomes(ac, newPos.first); break; // new
pos is a home check if rank and suit are ok
        }
        if(condition) { // if the move meets condition specific to it's
destination
            moveCard(ac, newPos); // move the card
            return true; // return ok
        } else return false; // the move didn't meet it's condition
    } else {
        return false; // the click wasn't made on any location
    }
}

//----------------------------------------------------------------
bool Deck::anotherCard(const int &np, const pint & ac) {
    return ac.first != np && // check if didn't click on the active card
    cards[ac.first]->getColour() != cards[np]->getColour() && // chceck
```

```
colour
    cards[ac.first]->getRank() == cards[np]->getRank() - 1 && // check rank
    enoughSpace(ac.second,0); // chceck if there is enough space to move
}

//----------------------------------------------------------------
bool Deck::enoughSpace(const int& onTop, bool reg) {
    int emptyRegs = 0;
    int emptyFcells = 0;
    int move = 0;
    for (int i = 0; i< regs.size(); i++) if(!regs[i]->getOnTop())
emptyRegs++; // calc empty regular piles
    for (int i = 0; i< fcells.size(); i++) if(!fcells[i]->getOnTop())
emptyFcells++; // calc empty freeCells piles
    if (emptyRegs == 0) move = emptyFcells; // if no empty regulars we can
move by amount of freecells
    else { // if some empty regulars
        if(reg) emptyRegs--; // moving to the regular means we have one
less to multiply by
        if (emptyFcells != 0) move = emptyFcells * (emptyRegs + 1); // if
there are empty freecells multiply by number of regs
        else move = emptyRegs; // if no empty free cells we can move by
number of regulars
    }
    if(move >= onTop) return 1; // if we can move move or equal to cards we
try to move we can move
    else {
        enough = false;
        return 0; // not enough space
    }
}

//----------------------------------------------------------------
bool Deck::checkHomes(const pint & ac, const int & newPos) {
    if(homes[newPos]->getCRank() == cards[ac.first]->getRank()) { // check
rank
        if(homes[newPos]->getSuit() == -1) { // if home has no suit
            homes[newPos]->setSuit(cards[ac.first]->getSuit()); // assign
cards suit
            homes[newPos]->setCRank(cards[ac.first]->getRank() + 1); // set
new home rank
            score += 10;
```

```cpp
            return true; // success
        } else {
            if (homes[newPos]->getSuit() == cards[ac.first]->getSuit()) {
                homes[newPos]->setCRank(cards[ac.first]->getRank() + 1); //
set new home rank
                score += 10;
                return true; // check if the suit is correct - success
            } else return false; // fail
        }
    } else return false; // fail
}

//----------------------------------------------------------------
void Deck::moveCard(const pint & ac, const pint & np) {
    occupyHistory(ac, np); // manage occupied states of affected cards and
save information to history
    int newActive = updateLocation(ac, np); // update card's position and
index values
    newActiveHistory(newActive, np.second);
    if(un) checkForInteractive(newActive); // in case the card got
deactivated while undo was active and cards are aligned
    cards[newActive]->setActive(0); // deactivate card
    if (ac.second != 0) moveCardsOnTop(ac.second, newActive); // move cards
on top
}

//----------------------------------------------------------------
void Deck::occupyHistory(const pint & ac, const pint & np) {
    occupyNp(np); // set the state of the pile card is coming to as
occupied
    switch(cards[ac.first]->getRow()) { // free the place card is leaving
and save information to history
        case -10: fcHistory(np, ac.first); break; // set history when the
active card was in free cell
        case -11: break; // used to be home
        case 0: regHistory(np, ac); break; // used to be last one in the
column
        default: cardHistory(np, ac); break; // used to be on top of other
card
    }
}
```

```cpp
//-------------------------------------------------------------
void Deck::occupyNp(const pint & np) {
    switch(np.second) {
        case 0:
            cards[np.first]->setTop(0);
            if(un) cards[np.first]->setInteractive(0);
            break; // new pos is a card - the card we will place stuff on
won't be on top
        case 1: regs[np.first]->setOnTop(1); break; // new pos is a regular
- mark the pile as full
        case 2: fcells[np.first]->setOnTop(1); break; // new pos is a free
cell - mark the pile as full
        case 3: break; // new pos is a home
    }
}

//-------------------------------------------------------------
void Deck::fcHistory(const pint & np, const int & active) {
    int fcellindex = freePile(fcells, active); // find index of the
abadoned free cell - free this pile
    if(!un){ // history
        cards[active]->pushPrevFcs(active); // save index if card wasn't
yet in fc
        newHistoryEntry(fcellindex, 2, 0); // save the location and index
of that location for the destination if undo
    }
}

//-------------------------------------------------------------
template <class Piles>
int Deck::freePile(const pil<Piles> &vec, const int &active) {
    int idx = -1; // set empty index
    for(int i = 0; i < vec.size(); i++){
        if(vec[i]->getPosition().x == cards[active]->getPosition().x) { //
find a pile under active card
            vec[i]->setOnTop(0); // set it free
            idx = i; // save its index
        }
    }
    return idx; // return index
}
```

```
//------------------------------------------------------------
void Deck::newHistoryEntry(const int& np, const int type, const int adj) {
    array<int, 4> entry; // create new entry
    entry[0] = -1; // set index of the card to move as empty (this happens
later)
    entry[1] = np; // save index of the cards location
    entry[2] = type; // save a type of location
    entry[3] = adj; // sace how much the index of the card to move should
be adjusted by
    history.push_back(entry); // push the entry into the history
}

//------------------------------------------------------------
void Deck::regHistory(const pint & np, const pint & ac) {
    int regindex = freePile(regs, ac.first); // find reg index
    if(!un) { // history
        switch(np.second) {
            case 0: historyBiggerRow(regs, regindex, np, ac.second,  -1);
break; // last in the column to another card
            default: newHistoryEntry(regindex, 1, 0); break; // last in the
column to FC, HOME or REG
        }
    }
}

//------------------------------------------------------------
template <class Piles>
void Deck::historyBiggerRow(const pil<Piles> &vec, const int &idx, const
pint &np, const int &onTop, const int &row) {
    int shiftBy = 0; // how many cards should the index of active card be
adjusted by when undo
    if(vec[idx]->getPosition().x > cards[np.first]->getPosition().x) { //
cards going left
        // the differece btw the card under it's future location and the
row of a card underneath it's current location
        shiftBy = cards[np.first]->getRow() - row - 1;
        // if the differece same or bigger than total amount of cards to
move
        if(shiftBy >= onTop + 1) shiftBy = onTop; // shift by the number of
cards on top of the card to move
    } else { // cards going right
        // the differece btw the card under it's future location and the
```

```
row of a card's current location
        shiftBy = cards[np.first]->getRow() - row;
        // if the differece same or bigger than amount of cards on top of
the card to move
        if(shiftBy >= onTop) shiftBy = onTop; // shift by the number of
cards on top of the card to move
    }
    int type; // declare type
    if (typeid(vec) == typeid(regs)) type = 1; // if the card moves from
reg pile type is regular
    else type = 0; // card moves from another card so type is card
    newHistoryEntry(idx, type, shiftBy); // save the location and index of
that location for the destination if undo
}

//----------------------------------------------------------------
void Deck::cardHistory(const pint & np, const pint & ac) {
    int cardindex = freeCard(ac.first);
    if (!un) {
        switch(np.second) {
            case 0: historyCardToCard(cardindex, np, ac.second); break; //
from other card to card
            case 1: historySmallerRow(regs, cardindex, np, ac.second, -1);
break; // from other card to regular
            default: newHistoryEntry(cardindex, 0, 0); break; // from other
card to free cell or home
        }
    }
}

//----------------------------------------------------------------
int Deck::freeCard(const int &active) {
    int idx; // declare index
    for(int i = 0; i<cards.size(); i++){
        if(cards[i]->getPosition().x == cards[active]->getPosition().x &&
// if same column
           cards[i]->getRow() == cards[active]->getRow() - 1) { // find the
card underneath the active card
            cards[i]->setInteractive(1); // make it clickable
            cards[i]->setTop(1); // and mark as the card on top of the pile
            checkForInteractive(i); // chceck if there are any accidentally
aligned cards underneath
```

```
            idx = i; // save index
        }
    }
    return idx; // return index
}

//----------------------------------------------------------------
void Deck::historyCardToCard(const int &idx, const pint & np, const int
&onTop) {
    if(cards[idx]->getRow() < cards[np.first]->getRow()) { // if card moves
to the bigger row than currently
        historyBiggerRow(cards, idx, np, onTop, cards[idx]->getRow()); //
set history
    } else if(cards[idx]->getRow() > cards[np.first]->getRow()) { // Card
moves to the smaller row than currently
        historySmallerRow(cards, idx, np, onTop,
cards[np.first]->getRow()); // set history
    } else { // the same row
        newHistoryEntry(idx, 0, 0); // set history
    }
}

//----------------------------------------------------------------
template <class Piles>
void Deck::historySmallerRow(const pil<Piles> &vec, const int &idx, const
pint &np, const int &onTop, const int &row) {
    // adjust the index of the card the active card will be coming back to
by a differece btw cards row and and the new row
    int howBigger = cards[idx]->getRow() - row;
    if (cards[idx]->getPosition().x > vec[np.first]->getPosition().x) { //
if the card is going right
        // if the differecne is bigger then the total amount of cards to
move adjust by the total amount of cards to move
        if(howBigger > onTop + 1) howBigger = onTop + 1;
    } else { // if the cards are going left
        howBigger--; // subtract one from that difference
        // if the subtacted difference is bigget than the amount of cards
on top of the card to move
        if(howBigger > onTop) howBigger = onTop + 1; // adjust by the total
amount of cards to mmove
    }
    newHistoryEntry(idx + howBigger, 0, 0); // save index to history
```

```
(always moves to another card)
}

//---------------------------------------------------------------
int Deck::updateLocation(const pint & ac, const pint & np) {
    int newRow = updatePosition(ac,np); // update cards position, return
it's new row
    int updatedIndex; // will hold the index of the active card after
updating indicies
    if(np.second == 0 || np.second == 1) updatedIndex =
assesSituation(newRow, ac.first); // cards will be placed within the main
deck
    else if(np.second == 2){ // cards will be placed  fc
        updatedIndex = cards.size()-1 - cAtHome;
        if(un) { // comes back to fc during undo
            updatedIndex =
cards[ac.first]->getPrevFcs(cards[ac.first]->getPrevFcsSize()-1);
            cards[ac.first]->popPrevFcs();
        }
    } else { // cards will be placed at home
        cAtHome++;
        updatedIndex = cards.size()-1;
    }
    reaarange(newRow, ac.first, updatedIndex); // update the order of
indicies
    return updatedIndex; // return the new index of the previously active
card
}

//---------------------------------------------------------------
int Deck::updatePosition(const pint & ac, const pint & np) {
    int newRow;
    switch(np.second) {
        case 0: // new pos is a card
            newRow = cards[np.first]->getRow() + 1; // the row card will be
in
            // position of card underneath + spacing

cards[ac.first]->setPosition(ofVec2f(cards[np.first]->getPosition().x,
cards[np.first]->getPosition().y + SPACING));
            break;
        case 1: // new pos is a regular
```

```
            newRow = 0; // if a card is goint to the pile it's new row will
be 0
            cards[ac.first]->setPosition(regs[np.first]->getPosition()); //
pile's position
            break;
        case 2: // new pos is a free cell
            newRow = -10; // if a card is goint to the pile it's new row
will be 0
            cards[ac.first]->setPosition(fcells[np.first]->getPosition());
// pile's position
            break;
        case 3: // new pos is a home
            newRow = -11; // if a card is goint to the pile it's new row
will be 0
            cards[ac.first]->setPosition(homes[np.first]->getPosition());
// pile's position
            break;
    }
    return newRow; // return the value of the card's new row
}

//----------------------------------------------------------------
int Deck::assesSituation(const int &nr,const int & activeC) {
    int updatedIndex;
    tuple<bool,float, int> situation(false, ofGetWidth(), -1);
    for(int i=0; i< cards.size(); i++) {
        if(cards[i]->getRow() == nr) { // if the card will land in the row
with other cards
            get<0>(situation) = true; // card will be in the same row as it
previously was
            if ((cards[i]->getPosition().x >
cards[activeC]->getPosition().x) && // if there is card to the right
                (cards[i]->getPosition().x < get<1>(situation))) { // find
the card closest to the active card
                get<1>(situation) = cards[i]->getPosition().x; // save its
position
                get<2>(situation) = i; // and its index
            }
        }
    }
    if(!get<0>(situation)) updatedIndex = findUpdatedIndex(activeC, nr -
1); // the card won't be moving to the same row
```

```
    else { // there were some cards
        if (get<2>(situation) == -1) updatedIndex =
findUpdatedIndex(activeC, nr); // but only on the left
        else { // there were some cards on the right
            if (get<2>(situation) > activeC) updatedIndex =
get<2>(situation) - 1; // if card moves to the right - place before card to
the right
            else updatedIndex = get<2>(situation); // replace card on the
right
        }
    }
    return updatedIndex;
}

//----------------------------------------------------------------
int Deck::findUpdatedIndex(const int& active, const int& row) {
    int idx;
    int hi = findHighestIndex(row); // find highest index of the given row
    if(cards[active]->getRow() > row || // if active row is bigger than
previous row
       cards[active]->getRow() == -11 || // but it's not in the freecell
       cards[active]->getRow() == -10) idx = hi + 1; // and not in the home
place after the hi
    else idx = hi; // place at hi
    return idx;
}

//----------------------------------------------------------------
int Deck::findHighestIndex(const int& row) {
    int hi = -1; // initialise new index
    for(int i = 0; i < cards.size(); i++) if (cards[i]->getRow() == row) if
(hi < i) hi = i; // find the highest index
    return hi; // and return it
}

//----------------------------------------------------------------
void Deck::reaarange(const int & row, const int & indexToMove, const int &
newIndex) {
    shared_ptr<Card> tmp(cards[indexToMove]); // create temporary card
    cards.erase(cards.begin() + indexToMove); // delete from the vector
    tmp->setRow(row); // update card's row information
    cards.insert(cards.begin()+newIndex, tmp); // insert at new index
```

```
}

//------------------------------------------------------------
void Deck::newActiveHistory(const int & newA, const int & target) {
    if(!un){
        switch(target) {
                // new pos is a free cell - save index of the last card
that is not yet home
            case 2: history[history.size()-1][0] = cards.size() - 1 -
cAtHome; break;
                // new pos is a home - save last index to history
            case 3: history[history.size()-1][0] = cards.size() - 1; break;
                // new pos is a card or a regular - save the index of the
new active minus eventual adjustement
            default: history[history.size()-1][0] = newA -
history[history.size()-1][3]; break;
        }
    }
}

//------------------------------------------------------------
void Deck::moveCardsOnTop(const int & onTop, int & newA) {
    for(int i = 0; i < onTop; i++) { // for each of them
        pint active(52,0);
        pint newP(newA,0);
        for(int j = 0; j < cards.size(); j++) { // look for the cards on
top
            // find one with the smallest index and make it activeCard
            if(cards[j]->getOnTop() && j < active.first) active.first = j;
        }
        newA = updateLocation(active, newP); // find the new index of the
active card
        cards[newA]->setOnTop(false); // deactivate card (no longer active
as on top)
    }
}

//------------------------------------------------------------
void Deck::checkAutocomplete() {
    int check = 0;
    for(int i = 0; i < regs.size(); i++) {
        int idx = -1;
        for(int j = 0; j < cards.size(); j++) {
            if(cards[j]->getPosition().x == regs[i]->getPosition().x &&
               cards[j]->getPosition().y == regs[i]->getPosition().y) {
                idx = j; // find cards on top of regs
            }
        }
        if (idx == -1 || autocompleteColumn(idx)) check++; // if the reg
empty or all cards in the column are ok
    }
    // if the amount of good columns is same as amount of columns it's
reafy for autocomplete
    if (check == regs.size()) autocomplete = true;
}

//----------------------------------------------------------------
bool Deck::autocompleteColumn(int idx) {
    int onTop = 0;
    for(int j = 0; j < cards.size(); j++) {
        if(cards[j]->getPosition().x == cards[idx]->getPosition().x && j >
idx) onTop++; // count cards on top of it
    }
    for(int j = 0; j < onTop; j++) { // for all cards on top of the reg
        if(idx != -1) idx = autocompleteCard(idx); // check if the cards
are sorted from the highest rank to the smaller one
        else break; // if not stop the loop
    }
    if (idx!=-1) return true; // column is ok
    else return false; // column is not ok
}

//----------------------------------------------------------------
int Deck::autocompleteCard(const int & idx) {
    int index = -1;
    for(int j = 0; j < cards.size(); j++) {
        if(cards[j]->getPosition().x == cards[idx]->getPosition().x && //
find the card on top
           cards[j]->getPosition().y == cards[idx]->getPosition().y +
SPACING &&
           cards[j]->getRank() <= cards[idx]->getRank()) { // chack if it
has a smaller rank
            index = j;
        }
    }
    return index;
}

//-----------------------------------------------------------------
void Deck::checkFinished() {
    int idx = -1; // check if all the cards are home
    for(int i = 0; i<cards.size(); i++) {
        if(cards[i]->getRow() != -11) {
            idx = i;
            break;
        }
    }
    if (idx == -1) finished = true; // if so set finished
}

//-----------------------------------------------------------------
void Deck::deactivate(const int &ac) {
    cards[ac]->setActive(0); // deactivate active card
    for(int i = 0; i < cards.size(); i++) if(cards[i]->getOnTop()) {
        cards[i]->setOnTop(false); // deactivate cards on top of it
    }
}

//-----------------------------------------------------------------
void Deck::undo() {
    if(act) deactivateAllCards(); // cancel card's activation
    if(history.size() > 0){ // if there is anything to undo
        score -= 5; // udno penalty
        un = true; // undo is in action
        activateCard(history[history.size()-1][0]); // activate previously
moved card
        pint active = findActive(); // assign it's index and count cards on
top of it
        // mark the location active used to be on top of before the move we
want to undo as new pos, and assign it a type of location
        pint np(history[history.size()-1][1],
history[history.size()-1][2]);
        // if the card is coming back from home - undo home state
        if (cards[active.first]->getRow() == -11) undoHome(active.first);
// if undoing from home
        moveCard(active, np); // move the cards
        history.pop_back(); // delete this entry from history
        un = false; // undo is now done
    }
}

//----------------------------------------------------------------
void Deck::deactivateAllCards() {
    act = false; // deactivate state in case undo was pressed while there
was an active card
    for(int i = 0; i<cards.size(); i++) if(cards[i]->getActive())
cards[i]->setActive(0); // deactivate any cards just in case
    for(int i = 0; i<cards.size(); i++) if(cards[i]->getActive())
cards[i]->setActive(0); // deactivate any cards just in case
}

//----------------------------------------------------------------
void Deck::undoHome(const int & idx) {
    score -= 10; // undo score for home
    cAtHome--; // decrease the amount of cards at home
    for(int i = 0; i< homes.size(); i++) {
        if(homes[i]->getPosition().x == cards[idx]->getPosition().x){
            if(cards[idx]->getRank() == 0) homes[i]->setSuit(-1); // if the
card was ace unasign the suit fron the home cell
            homes[i]->setCRank(cards[idx]->getRank()); // make it expect a
lower rank
        }
    }
}

//----------------------------------------------------------------

void Deck::hint() {
    if(act) deactivateAllCards(); // cancel card's activation
    // check for hints in that order:
    bool ok = (checkHint(0,3) || // card to home
               checkHint(1,3) || // fc to home
               checkHint(1,0) || // fc to card
               checkHint(0,0) || // card to card
               checkHint(0,1) || // card to reg
               checkHint(1,1) || // fc to reg
               checkHint(0,2)); // card to fc
    if(!ok) noMore = true; // if none of those are true there is no more
```

```
possible moves
    hin = ok; // if any is true show the hint
}

//----------------------------------------------------------------
bool Deck::checkHint(const bool fc, const int target) {
    cout<<"CHECK"<<endl;
    bool match = false;
    int howManyTimes = (fc) ? fcells.size() : regs.size(); // check for up
to as many times as either columns or freecells
    for(int i = 0; i < howManyTimes; i++) {
        // find a card to find a possible move for
        int idx = (fc) ? hintFindIdx(1, i, target) : hintFindIdx(0, i,
target);
        if(idx != -1) { // if found
            bool condition;
            switch(target) { // find possible match
                case 0: condition = hintFindTarget(cards, idx, 0); break;
                case 1: condition = hintFindTarget(regs, idx, 1); break;
                case 2: condition = hintFindTarget(fcells, idx, 2); break;
                case 3: condition = hintFindTarget(homes, idx, 3); break;
            }
            if(condition) { // if the match was found
                match = true; // there is a possible move
                break; // stop the loop
            }
        }
    }
    return match; // return possible move or lack of thereof
}

//----------------------------------------------------------------
int Deck::hintFindIdx(const bool & fc, const int & sourceIdx, const int &
target) {
    int idx = -1;
    if (fc) { // going from fc
        for(int i = 0; i < cards.size(); i++) { // if card is on top of fc
            if(cards[i]->getPosition() == fcells[sourceIdx]->getPosition())
idx = i;
        }
    } else { // going from cards
        if(regs[sourceIdx]->getOnTop()) { // only for columns that have
```

```
cards in it
            if(target == 0 || target == 1) { // going to other card or reg
                idx = 52;
                for(int i = 0; i < cards.size(); i++) { // find lowest
interactive index
                    if(cards[i]->getInteractive() &&
                       cards[i]->getPosition().x ==
regs[sourceIdx]->getPosition().x &&
                       i < idx) idx = i;
                }
            } else { // going home or fc
                for(int i = 0; i < cards.size(); i++) { // only find top
cards
                    if(cards[i]->getTop() && cards[i]->getPosition().x ==
regs[sourceIdx]->getPosition().x) idx = i;
                }
            }
        }
    }
    return idx;
}

//----------------------------------------------------------------
template <class Piles>
bool Deck::hintFindTarget(const pil<Piles> & vec, const int & idx, const
int & target) {
    pint cidx(idx,0); // this is needed for another card and enough space
functions
    if (target == 0 || target == 1) { // if target is a card or a reg
        for(int j = 0; j < cards.size(); j++) {
            if(cards[j]->getPosition().x == cards[idx]->getPosition().x &&
j > idx) cidx.second++; // count cards on top
        }
    }
    bool ok = false;
    for(int j = 0; j < vec.size(); j++) { // for the length of target
vector
        bool condition;
        switch (target) {
                // moveing to the top card that is not home or freeCell
only if suits and colours match and enougch space
            case 0: condition = cards[j]->getTop() &&
                cards[j]->getRow() != -10 && cards[j]->getRow() != -11 &&
anotherCard(j, cidx); enough = true; break;
                // moveing to the reg if it's empty and enough space
            case 1: condition = !regs[j]->getOnTop() &&
enoughSpace(cidx.second, 1); enough = true; break;
                // moveing to the fcell if it's empty
            case 2: condition = !fcells[j]->getOnTop(); break;
                // moveing to the home cell if rhe rank is ok and suit is
either empty or ok
            case 3: condition = homes[j]->getCRank() ==
cards[idx]->getRank() &&
                (homes[j]->getSuit() == -1 || homes[j]->getSuit() ==
cards[idx]->getSuit()); break;
        }
        if(condition) {
            setHint(idx, j, target); // set hintcard and target
            ok = true; // match is found for the column
            break;
        }
    }
    return ok; // return match
}

//----------------------------------------------------------------
void Deck::setHint(const int & idx, const int & targetIdx, const int &
target) {
    cards[idx]->setHint(1); // set hint card
    hintPos = (cards[0]->getSize()/2); // center the hint arrow on the
target
    switch (target) { // choose target
        case 3: { // home
            // if home is empty set tarhet to home
            if(!homes[targetIdx]->getOnTop()) hintPos +=
homes[targetIdx]->getPosition();
            else {
                int i = -1;
                for(int j = 0; j < cards.size(); j++) // find the topmost
card on top of the home
                    if(cards[j]->getPosition() ==
homes[targetIdx]->getPosition() && i > j) i = j;
                hintPos += cards[i]->getPosition(); // set hint target to
that card
```

```
            }
            break;
        }
        case 2: { hintPos += fcells[targetIdx]->getPosition(); break; } //
set fcell as target
        default: { // cards and regs
            // if cards get card pos as taget if reg get regs pos as target
            hintPos += (target == 0) ? cards[targetIdx]->getPosition() :
regs[targetIdx]->getPosition();
            // set all cards on top of the hint as hint cards as well
            for(int j = 0; j < cards.size(); j++)
if(cards[j]->getPosition().x == cards[idx]->getPosition().x && j > idx)
cards[j]->setHint(1);
            break;
        }
    }
}

//----------------------------------------------------------------
void Deck::skipAutocomplete() {
    dontAutocomplete = false;
    autocomplete = false;
}

//----------------------------------------------------------------
void Deck::doAutocomplete() {
    int cardsInGame;
    for(int i = 0; i < cards.size(); i++) if(cards[i]->getRow() != -11)
cardsInGame++; // count cards still in the game
    for(int i = 0; i < cardsInGame; i++) {
        int idx = autoFindIdx(i); // find smallest card
            if(idx!= 52) {
                pint ac(idx, 0);
                for(int j = 0; j < homes.size(); j++) {
                if(checkHomes(ac,j)) { // find a home to move to and if
found
                    pint home(j, 3);
                    moveCard(ac, home); // move to that home
                    moves++; // count moves
                    break; // stop checking for other freecells we already
got what we needed
                }
            }
        }
    }
    dontAutocomplete = false; // stop checking for autocomplete after every
move
    autocomplete = false; // disable autocomplete
    finished = true; // enable finished tab
}

//--------------------------------------------------------------
int Deck::autoFindIdx(const int &i) {
    int idx = 52;
    int rank = 13;
    for(int j = 0; j < cards.size(); j++) {
        // card is not at home find the smallest card in the deck
        if(cards[j]->getRow() != -11 && rank > cards[j]->getRank()) {
            if(cards[j]->getTop() || cards[j]->getRow() == -10) { // if the
card is in free cell or on top of the column
                idx = j;
                rank = cards[j]->getRank();
            }
        }
    }
    return idx;
}

//--------------------------------------------------------------
bool Deck::getFinished() {
    return finished;
}

//--------------------------------------------------------------
int Deck::getMoves() {
    return moves;
}

//--------------------------------------------------------------
int Deck::getScore() {
    return score;
}

//--------------------------------------------------------------
```

```
string Deck::getTime() {
    return time;
}

//--------------------------------------------------------------
int Deck::getSeconds() {
    return sec;
}
```

From button.hpp

```
#ifndef button_hpp
#define button_hpp
#include "ofMain.h"
#include "deck.hpp"

//--------------------------------------------------------------------------
-----

class Button {
public:
    Button(string lab, float x, float y);
    void draw();
    void mousePressed(void (Deck::*action)(), Deck& obj);
    bool getHover();


private:
    string label;
    float xPos;
    float yPos;
    int xSize = 90;
    int ySize = 30;
};

#endif /* button_hpp */
```

From button.cpp

```
#include "button.hpp"


//--------------------------------------------------------------
Button::Button(string lab, float x, float y) : label(lab), xPos(x), yPos(y)
{}

//--------------------------------------------------------------
void Button::draw() {
    ofPushStyle();
    if(getHover()){ // lighter color when hovered over
        ofFill();
        ofSetColor(200);
        ofDrawRectangle(xPos, yPos, xSize, ySize);
    }
    ofSetColor(0);
    ofNoFill();
    ofDrawRectangle(xPos, yPos, xSize, ySize);
    int labelWidth = label.length() * 8;
    int labelX = (xSize - labelWidth) / 2; // center align the text within
the button
    ofDrawBitmapString(label, xPos + labelX, yPos + 20);
    ofPopStyle();

}

//--------------------------------------------------------------
void Button::mousePressed(void (Deck::*action)(), Deck& obj) {
    (obj.*action)(); // takes a pointer to the function from the deck
}

//--------------------------------------------------------------
bool Button::getHover() { // returns true if the mose is over the button
    return ofGetMouseX() > xPos &&
    ofGetMouseY() > yPos &&
    ofGetMouseX() < xPos + xSize &&
    ofGetMouseY() < yPos + ySize;
}
```

From card.hpp

```
#ifndef card_hpp
#define card_hpp

#include "ofMain.h"


//--------------------------------------------------------------------------
-----

class Card {
public:
    Card(int val);
    void draw();
    void setHint(bool h);
    bool getHint();
    void setRow(int r);
    int getRow();
    void setOnTop(bool t);
    bool getOnTop();
    void setPosition(ofVec2f p);
    ofVec2f getPosition();
    void setActive(bool a);
    bool getActive();
    void setInteractive(bool i);
    bool getInteractive();
    void setTop(bool t);
    bool getTop();
    void pushPrevFcs(const int & idx);
    void popPrevFcs();
    int getPrevFcs(const int & size);
    int getPrevFcsSize();
    const ofVec2f getSize();
    const bool getColour();
    const int getRank();
    const int getSuit();
private:
    // initialised before setup (used for calc suit and rank)
    int value; // card's unique id
    // setup (used for loading texture)
    const char* suits = "CDHS";
    const char* ranks = "A23456789TJQK";
    // whole program
    ofImage face; // card's texture
    bool color; // card's color
    int rank; // card's rank
    int suit; // card's suit
    ofVec2f position; // card's position
    int row; // card's row
    ofVec2f size; // card's size
    bool active; // is the card active
    bool interactive; // can the card be clicked
    bool top; // is the card on top of the column
    bool onTopOfActive; // is the card on top of the active card
    bool hint; // if hint is on
    vector<int> prevFcs; // used in undo collects information of index no
in the free cell
    void drawFace(); // draw card
    void drawOverlay(bool condition, ofColor col); // draw overlay
};

#endif /* card_hpp */
```

From card.cpp

```
#include "card.hpp"

//--------------------------------------------------------------
Card::Card(int val) : value(val) {
    // begin from
https://rosettacode.org/wiki/Deal_cards_for_FreeCell#OOP_version
    rank = value / 4; // calc rank value
    suit = value % 4; // calc suit value
    stringstream s;
    s << ranks[rank] << suits[suit]; // create file name
    // end from
https://rosettacode.org/wiki/Deal_cards_for_FreeCell#OOP_version
    face.load("ca/"+s.str()+".png"); // load image
    if(suit == 1 || suit == 2) color = 0; // asign color black
    else color = 1; // asign color red
    size.x = ofGetWidth()/10; // calc size x
    float scale = face.getWidth()/size.x; // calc scale
    size.y = face.getHeight()/scale; // calc size y and scale it
    // VARIABLE
    active = false; // card is passive
    hint = false; // hint is off
    onTopOfActive = false; // card is not on top of another active card
    active = false; // card is passive
    interactive = false; // card can't be clicked
    top = false; // card is not on top of the column
}


//--------------------------------------------------------------
void Card::draw() {
    drawFace(); // draw a card
    drawOverlay(active || onTopOfActive, ofColor(255, 0, 0, 20)); // mark
as active or on top of it
    drawOverlay(hint, ofColor(0, 0, 255, 20)); // mark as hint or on top of
it
}

//--------------------------------------------------------------
void Card::drawFace(){
    ofPushStyle();
    ofSetColor(0);
    ofNoFill();
    ofDrawRectangle(position.x, position.y, size.x, size.y); // draw a nice
border around the card
    ofPopStyle();
    face.draw(position.x,position.y,size.x,size.y); // draw the card
}

//--------------------------------------------------------------
void Card::drawOverlay(bool condition, ofColor col){
    ofPushStyle();
    if (condition) {
        ofSetColor(col);
        ofDrawRectangle(position.x, position.y, size.x, size.y); // draw
coloured overlay
    }
    ofPopStyle();
}

//--------------------------------------------------------------
void Card::setHint(bool h){
    hint = h;
}

//--------------------------------------------------------------
bool Card::getHint(){
    return hint;
}

//--------------------------------------------------------------
void Card::setRow(int r){
    row = r;
}

//--------------------------------------------------------------
int Card::getRow(){
    return row;
}

//--------------------------------------------------------------
void Card::setOnTop(bool t){
    onTopOfActive = t;
}

//--------------------------------------------------------------
bool Card::getOnTop(){
    return onTopOfActive;
}

//--------------------------------------------------------------
void Card::setPosition(ofVec2f p){
    position = p;
}

//--------------------------------------------------------------
ofVec2f Card::getPosition(){
    return position;
}
```

```
//--------------------------------------------------------------
void Card::setActive(bool a){
    active = a;
}

//--------------------------------------------------------------
bool Card::getActive(){
    return active;
}

//--------------------------------------------------------------
void Card::setInteractive(bool i){
    interactive = i;
}

//--------------------------------------------------------------
bool Card::getInteractive(){
    return interactive;
}

//--------------------------------------------------------------
void Card::setTop(bool t){
    top = t;
}

//--------------------------------------------------------------
bool Card::getTop(){
    return top;
}

//--------------------------------------------------------------
void Card::pushPrevFcs(const int & idx){
    prevFcs.push_back(idx);
}

//--------------------------------------------------------------
void Card::popPrevFcs(){
    prevFcs.pop_back();
}

//--------------------------------------------------------------
```

```
int Card::getPrevFcsSize(){
    return prevFcs.size();
}

//--------------------------------------------------------------
int Card::getPrevFcs(const int & size){
    return prevFcs[size];
}

//--------------------------------------------------------------
const ofVec2f Card::getSize(){
    return size;
}

//--------------------------------------------------------------
const bool Card::getColour(){
    return color;
}

//--------------------------------------------------------------
const int Card::getRank(){
    return rank;
}

//--------------------------------------------------------------
const int Card::getSuit(){
    return suit;
}
```

From pile.hpp

```
#ifndef pile_hpp
#define pile_hpp

#include "ofMain.h"

class Pile {
public:
    // common to all piles
    Pile(ofVec2f pos, ofVec2f s, bool top) {};
    void setPosition(ofVec2f p) { position = p; };
    const ofVec2f getPosition() { return position; };
    void setSize(ofVec2f s) { size = s; };
    const ofVec2f getSize() { return size; };
    void setOnTop(bool c) { onTop = c; };
    bool getOnTop() { return onTop; };
    // pile specific
    virtual  void draw() {};
    // only in Home
    virtual void setSuit(int s) {};
    virtual int getSuit() {};
    virtual void setCRank(int cr) {};
    virtual int getCRank() {};
private:
    // common to all piles
    ofVec2f position;
    ofVec2f size;
    bool onTop;
};

#endif /* pile_hpp */
```

From home.hpp

```
#ifndef home_hpp
#define home_hpp

#include "ofMain.h"
#include "pile.hpp"

class Home : public Pile {
public:
    Home(ofVec2f pos, ofVec2f s, bool top);
    void draw();
    void setSuit(int s);
    int getSuit();
    void setCRank(int cr);
    int getCRank();
private:
    int currentRank;
    int suit = -1;
    ofImage homeImage;

};

#endif /* home_hpp */
```

From Home.cpp

```
#include "home.hpp"
#include "pile.hpp"

//--------------------------------------------------------------
Home::Home(ofVec2f pos, ofVec2f s, bool top) : Pile(pos, s, top) {
    setPosition(pos);
    setSize(s);
    setOnTop(top);
    currentRank = 0;
    homeImage.load("home.png");
}


//--------------------------------------------------------------
void Home::draw() {
    ofPushStyle();
    ofSetColor(0); // freeCell
    ofNoFill();
    ofDrawRectangle(getPosition().x, getPosition().y, getSize().x,
getSize().y);
    ofFill();
    ofSetColor(0,50); // freeCell
    homeImage.draw(getPosition().x, getPosition().y, getSize().x,
getSize().y);
    ofPopStyle();
}
```

```
//--------------------------------------------------------------
void Home::setSuit(int s) {
    suit = s;
}

//--------------------------------------------------------------
int Home::getSuit() {
    return suit;
}

//--------------------------------------------------------------
void Home::setCRank(int cr) {
    currentRank = cr;
}

//--------------------------------------------------------------
int Home::getCRank() {
    return currentRank;
}
```

From regular.hpp

```
#ifndef regular_hpp
#define regular_hpp

#include "ofMain.h"
#include "pile.hpp"

class Regular : public Pile {
public:
    Regular(ofVec2f pos, ofVec2f s, bool top);
    void draw();
};

#endif /* regular_hpp */
```

From regular.cpp

```cpp
#include "regular.hpp"

//--------------------------------------------------------------
Regular::Regular(ofVec2f pos, ofVec2f s, bool top) : Pile(pos, s, top) {
    setPosition(pos);
    setSize(s);
    setOnTop(top);
}

//--------------------------------------------------------------
void Regular::draw() {
    ofPushStyle();
    ofSetColor(0);
    ofNoFill();
    ofDrawRectangle(getPosition().x, getPosition().y, getSize().x,
getSize().y);
    ofFill();
    ofSetColor(0, 50);
    ofDrawRectangle(getPosition().x, getPosition().y, getSize().x,
getSize().y);
    ofPopStyle();
}
```