Project 1

In this project you should end up writing software that enables its user to play a very simple form of solitaire we call patience310. However, the educational goals are (1) to learn many of the general development and object oriented design and C++ coding practices covered in Chapters 1-2 of DSO plus (2) some data types and their implementation by the partially filled array data structure, similar to what is presented in Chapter 3 of DSO.

The game is played with one Deck of up to 52 cards. Unlike more conventional playing cards with 4 suites and 13 values, these cards are numbered with the integers 1 to the number of cards, one card bearing each of those numbers.

You specify the (1) number of Piles and (2) the number of cards in the Deck when you start each new game. When the game starts each Pile is initially empty. However, an unusual feature is the Deck has a specified order. You can shuffle or rotate it if you want to, but that costs you penalty points!

The objective is to draw all the cards from the Deck and arrange them in the Piles so that the numbers decrease as you go from the bottom to the top of each Pile and to do this with the fewest “penalty points” (defined below) as you can.

(Notice that the one deck and the multiple piles of cards are basic “entities” involved in the game definition and implementation. Other ones are the individual cards. We will therefore include in our object oriented design a class named Deck and a class named Pile. If the cards were more complicated, for example, they had conventional suites and values, and would be displayed pictorially, it would make sense to include a class named Card for them as well. However, the purposes of this project indicate that, at least to me, that using the primitive C/C++ datatype of unsigned int for each card would be a wiser choice. We will, in Part 2, specify that a C++ class named Game will hold one Deck instance, a number of Pile instances, some card and scoring values, and provide functions that implement the playing logic implied by the rules.)

To eventually define all the game rules, we first define certain concepts of our game:

Definition of draw: To do a draw, you first remove the top card from the Deck and put it face up, so you see the number. Then, you can finish the draw by putting the drawn card on top of any Pile you choose, as long as you do not put a larger numbered card on top of a smaller one. Regardless of whether it is possible to finish the draw you can take any number of “penalty moves” (defined below) before you finish the draw. (Ask yourself: In what situation is it impossible to finish a draw? Good software designers must ask themselves questions like this.)

Definition of penalty move: To do a penalty move, you move the top card from one Pile to the top of another Pile, with the rule that a larger numbered card never goes on top of a smaller numbered card. Each penalty move counts for one penalty point.

It is a fact that, assuming there are at least two Piles, after the first draw, it is always possible to do a penalty move. (Try to figure out logically and explain completely why this is true, i.e., try to prove it.)
However, after some number of penalty moves you might end up with the same position you had earlier. Furthermore, it might sometimes be impossible for any number of penalty moves to leave an empty Pile or a Pile with a large enough top card so the drawn card can be put on top of a Pile to finish the a draw.

So, there is one more rule: At any time during a draw, you might put the drawn card on bottom of the Deck and draw the new top card and continue. Or you might replace the drawn card to the top, shuffle the Deck, draw the new top card and continue. However, each time you shuffle or rotate the deck this way, the cost is 10 penalty points.

Finally, unlike normal card games, this game is started with the cards in the fixed order: 1, 2, 3, etc. from the top to the bottom of the deck. So, the first card drawn is always the 1 (one).

Part 1 of the project is to develop two separate classes, one named Deck and the other named Pile. Each class must have the member functions specified below to support the game and to verify operations like shuffling.

Part 2 is to integrate the classes from Part 1 to create a class named Game, and to create the user interface module (The main() function and preferably other supporting functions) which constructs one Game and enables the user (a person) to play with it. The instance of Game will hold one instance of Deck and an array of Piles. That Game will implement the moves, rule checking, scoring, detecting when the game ends, etc. The Game class must also have an overloaded constant friend function operator<<(ostream&, Game&) to send a display of the Piles, drawn card, and score report to a given C++ ostream.

Detailed instructions for submitting your work, and our CSI310 project late project grading policy have been provided in a previous handout.

Sample executable implementations or input and outputs are supplied will be supplied next week.

To “specify a class” (for this course CSI310) means to write public member function prototypes with the required pre- and post-condition documentation, as DSO illustrated throughout Chapters 2 and 3; see DSO programming projects “8” and “13” for more examples.

Part 1 must provide two test driver programs fashioned after the program on pages 129-131 of DSO. The test driver program for Part 2 will be a simple text-terminal user interface for playing the game and running debugging and verification features. The executable files holding the driver programs must be named TestDeck and TestPile in Part 1 and patience310 in part2. You are encouraged to “be lazy”: download DSO’s code for the “Interactive Test Program for the Sequence Class” and use it as a “pattern” for your Part 1 test drivers and your Part 2 game playing user interface.

Each test driver program must print an introduction and an informative list of choices as illustrated on page 132.

0.1 Part 1, Deck class

Study and copy from ~acsi310/Proj1/Deck.h the file whose contents are below. Then write the test driver. Next begin the implementation file Deck.cxx with stubs for any function body you cannot code immediately. Write a build.sh script (see Lab1). Compile the programs, correct and
fix the almost inevitable syntax errors one at a time. (Rerun your build script after each single correction because compilers like g++ often don’t report accurately errors in the code that comes after the first error.)

Finally, fill in any stubs, and test/debug as necessary.

For the TestDeck program, the choice letters must be lower case and be r, s, t, d, e, p, b and q.

Here is a sample list of choices, which specifies what the above letters mean:

The following choices are available:
- r Call the reset function with a given number
- s Shuffle the Deck with shuffle function
- t Rotate the Deck with rotate function
- d Call the draw function and print the return value
- e Call the not_empty function and print the return value
- p Print the Deck using the overloaded friend <<
- b Put the given card back in the deck by calling replace
- q Quit this test program

Enter choice:

The prompt for the choice in all parts must be exactly:

Enter choice:

(note the colon). The prompt for entering a number must end with a colon. This is critical for getting full credit because the testing system for grading your work will use these prompts to tell when to provide the sample test inputs!

0.1.1 Header file Deck.h

#include <iostream>

class Deck {
public:
    static const std::size_t capacity = 52;
    Deck(unsigned int how_many_cards=1) { reset(how_many_cards); }
    //Pre: 1<=how_many_cards<=52
    //Post: Let s=how_many_cards
// cards[] was initialized to {s,s-1,...,2,1}

void reset(unsigned int how_many_cards);
//Pre: 1<=how_many_cards<=52
//Post: s==how_many_cards-1
// shuffle() has been called after
// cards[] was initialized to {s,s-1,...,2,1}

void rotate();
//Pre: Let s=used and
// cards[]=c(0)c(1)...c(s-2)c(s-1)
// (This precondition merely defines the abbreviated notation
// used to write the postcondition. It should NOT be checked
// by assert() because such a check makes no sense!
//Post: cards[]=c(s-1)c(0)...c(s-2)
// i.e., the top card has been moved to the bottom of the
// deck, so all the other cards have been moved one position towards
// the top. If s==0 or s==1, nothing changes.
//Academic Calibration: Unlike shuffle, this should be
// easy if not immediate.

void shuffle();
//Pre: Let s=used and
// cards[]=c(0)c(1)...c(h-1)c(h)...c(s-1)
// where h=s/2 rounded down.
//Post: If used is odd,
// cards[]=c(h)c(h+1)c(0)c(h+2)...c(s-2)c(h-2)c(s-1)c(h-1)
// If used is even,
// cards[]=c(h)c(0)c(h+1)c(1)...c(s-2)c(h-2)c(s-1)c(h-1)
// Remark: The "top of the deck" is the entry
// with the highest subscript. The deck is
// evenly split if the number of cards (used)
// is even. If used is odd, the top portion
// of the split has one more card than the bottom.
// The shuffling starts by taking the first (top) card
// of the bottom portion, which is c(h-1), and putting
// it at the top of the shuffled deck. It continues
// next with the top card of the other (i.e. top) portion,
// and alternates further as much as possible.
//Tip: If this is confusing, make or use a sample deck with
// 5 REAL PAPER CARDS. Try it out in the two cases:
// 5 cards and 4 cards. Split and shuffle the little
decks according to the specification above,
// except do it face up and watch what happens.

// Don't Dispair: Programming the split and shuffle is tricky.
// (I found it hard, at least).
// If you don't get it, skip implementing it
// or leave the cards unshuffled, and you can still
// get most (95% or so) of the credit.

unsigned int draw();
// Pre: The deck isn’t empty. used==s, used>0
// Post: The top card (card[used-1])
// is removed and returned. used==s-1.

bool not_empty();
// Pre: none
// Post: returns true iff deck is not empty,
// equivalently, used!=0

void replace(int card);
// Pre: card is not in the deck and 0<card<=original_size
// Post: card is inserted on top of the deck.

friend std::ostream& operator<<(std::ostream& outs, Deck& deck);
// Pre: outs is a "good" ostream. (You don’t have to check.)
// Post: The sequence of card (numbers) in the deck is sent to
// outs, formatted as a space separated sequence of decimal numerals,
// in order from the bottom (first) of the deck to the top of
// the deck (last). The sequence may or may not have a trailing
// space character. Reference parameter outs is returned.
// Remark: See pages 74-78 of DSO for explanations and examples of
// how to code the body of this function.
// Tip: This body does NOT CONTAIN "cout <<" but it does contain
// "outs <<".

private:
    unsigned int cards[capacity];
    std::size_t used;
    std::size_t original_size;
    //Representation Invariants:
Each part 1 test driver should simply call the indicated member functions with parameters supplied by the user. You (and the graders) will observe how the precondition checking assertions “fire” when preconditions are violated.

For simplicity, use “cin >> intvar;” to input an integer when an integer is expected: If the program tries to re-read the integer after the player types a character instead, it will loop infinitely.

THAT’S OK for this first project. Future projects might require more robust recovery from user errors.

0.2 Part 1, class Pile

For this module, it is your job to create the header file which defines class Pile and has documentation in the form of pre and post conditions for all the member functions of the class.

As with class Deck, your build.sh script must build a test driver program named testPile, to conform with interface details specified below.

Like Deck, a Pile should hold a sequence of “cards” represented by C/C++ unsigned ints in a partially filled array.

The public member functions must include the following member functions:

- A DEFAULT constructor that initializes the Pile to the empty state. (This default constructor is needed because each instance of the Game class, see below, will have an array of Piles. In C++, the only way to specify a constructor to be used for each element in an array is to specify the default constructor. See DSO.)

Q To quit the TestPile program.

r A function to re-initialize the pile to the empty state, so the Pile can be reused in a re-initialized Game.

p A function to be called to indicate the (human) player specified a card should be taken from the top of the Pile. If that is legal (because the Pile isn’t empty) the function should perform the action. If not the function should not change the Pile’s state. In both cases, the return value must indicate whether or not the action the player specified was legal. Suggestion: Name functions like this with the word “try”.


x A (constant) function to return top card (integer value), and 0 if the Pile is empty.

k A function analogous to the “try to take” function above but which tries to put the given card on top of the pile. Note the rules of the game restrict that card can be given.

n A constant boolean function whose return value is true if and only if the Pile is non-empty.

- An overloaded output friend function that prints the same kind of representation of the Pile as we specified for the Deck.

1 Part 2

Design a class named Game with private data members including one Deck and an array of 13 Piles (note 13 is the maximum number of piles a player can specify). The Game class should have a member function for each move the (human) player can specify, plus an overloaded friend output function for printing.

The move functions must determine if a given move is legal, and then perform it if so and ignore it if not. These functions must also return values that the user interface program can use to tell whether the move was performed. If it is not legal, the user interface must print a message to that effect. If the move is legal, the user interface must reprint the Game (using the overloaded friend output function), reprint the choice menu, and prompt for a new choice.

Here is a sample run. The sample choice menu below SPECIFICIES the letters you must use to indicate each choice.

~/CSI310/Pr1 $ patience310
Welcome to patience310. How many cards (1-52) or quit?: 3
How many piles (1-13) or quit?: 4
Drawn: 1

Pile1:
Pile2:
Pile3:
Pile4:

You placed 0 draws, 3 to go, and have 0 penalty points.

The following choices are available:
  f Specify the destination pile for the drawn card
  n Specify source and then destination pile for a move (-1 point)
  s Replace drawn card and reshuffle (-10 points)
  t Replace drawn card and rotate (-10 points)
  r Resign this game, start another
  Q Quit this silly program
Print the deck (for debugging, unusual for card games!)
Enter choice: f
Please enter a positive pile number number: 4
4 has been read.
Drawn: 2

Pile1:
Pile2:
Pile3:
Pile4: 1

You placed 1 draws, 2 to go, and have 0 penalty points.

The following choices are available:
  f  Specify the desti ....

The following choices are available:

and so forth.... Observe: Trying to finish the draw of card 2 by putting it on Pile 4 is illegal, because Pile 4 has card 1 on top, and card 2 is not smaller than card 1.

2 Project Rules and Guidelines

The use of the following practices will be checked during grading and will contribute to your grade:

1. The project must be done and then submitted as a single directory, possibly with subdirectories if you like. (Each project, lab exercise, etc., must be done under its own directory; so you never worry about which files are for which project.)

2. All linked executable files and object files under it MUST be removed before the directory is submitted. (Such files consume disk quota of the course account and so cause problems for all students if some submissions include them. Hence the harsh penalties!)

3. Every implementation file (files with extension .cxx or .template) and non-trivial header file must be accompanied with an RCS database file that records its entire development history. Those RCS database files will be accessed only by RCS commands such as “ci -l” (here “l” is the lower-case letter “ell”). They have names obtained by appending ,v (comma v) to the
name of the original file. The database file may exist in an RCS subdirectory of the directory holding the original file, or at the same level as the original file.

The main reasons for requiring version control in this course are:

(a) If (when!!!!) you accidentally delete, overwrite, irreversibly mess up, or otherwise lose the original file, you can get back old versions of it which you “checked in” (with the \texttt{ci -l} command) before the disaster.

(b) We introduce you to an important software development practice, especially for large or group projects. They use more complex network based revision control systems such as CVS, but RCS is a good place to start.

The quick-start way to use RCS is (after optionally making the RCS subdirectory) start writing \texttt{filename} with the text editor, and then, each time you save the file, give the command “\texttt{ci -l filename}”. That command will prompt you to write a comment about the version you are “checking-in”. You can then review the history of the versions with the command “\texttt{rlog filename}”.

If (when!) disaster strikes, read the man pages and/or consult a TA, knowledgable classmate, or the professor for instructions on using certain other RCS commands to retrieve an older version\footnote{OK, here’s how to do it. (1) \texttt{ci filename} to check in the bad version, to unlock it and to keep it just in case you will need it. (2) \texttt{co -r1.n filename} to retrieve an older version, say 1..n, read-only. (3) Is it the one you can use? If not, repeat step (2). (4) \texttt{rcs -l filename} to “lock it”. (5) \texttt{ci -l filename} to check the “reverted version” back in as the latest version. This makes \texttt{filename} writable again, and you can continue from before the disaster! For further info, begin with \texttt{man rcsintro}.}

There is a handy \texttt{emacs} command “\texttt{C-x v v}” that automates somewhat more sophisticated RCS usage. Type this command while in the buffer editing the file in order to save the file; and then type it again in order to begin modifying the file again. When you command “\texttt{C-x v v}” after editing, \texttt{emacs} will open another buffer for you to write a version comment: After typing a comment about this version, press Control-c twice (\texttt{C-c C-c}).

4. Header (class declaration) files, function implementation files, and template files (when we get to them) for each C++ class you write MUST all be separate, and be separate from files that implement test drivers. Many examples of separating parts of a program into multiple files, each with a small, well-understood purpose, are given in the DSO textbook. (The Lab 1 exercise provides a simple first example.)

5. A “build shell script” named \texttt{build.sh} must be provided at the top level of the submitted project directory. This build script must contain the commands that build the project software so we can test it. (The Lab 1 exercise covers how to do this.)

6. “ZERO GRADE IF IT DOESN’T COMPILE”

When we grade a submission, we will recreate the directory you submitted. First we will delete all object and executable files (and subtract a grade penalty if there were any). Next, we will
run the build.sh script in its directory. If the build script fails to make your submission compile and link to specified executable files we can run, the project (or part) will get ZERO points AUTOMATICALLY.

3 Elementary Software Development Practices for CSI310

Follow the advice of section (1.3)! It is wise to not “change suspicious code on the hope that the change might work better.” But here’s one addition: Sometimes, usually at the beginning of a project, the code you wrote might be so confusing to yourself that is impossible to tell what is causing failure, even when program execution is examined with a debugger. In that case, it might be a good idea to throw out your first or so attempts and start over again! It will not be a waste of time because the first or so attempt at a concrete design makes your mind really understand the problem.

Some students have been taught to write each programming assignment all at once, trying to get it correct before beginning to typing it. Usually it is not correct, so hours of syntax error correcting and then debugging ensue. This practice is immature and practically useless, except for extremely small problems, for assignments meant to teach the simplest things (and for CS course exams).

After some careful specification, design and perhaps analysis, “interesting” programs are best written (i.e., implemented) incrementally: A compilable early version that doesn’t do much is created very quickly. Then, parts are added little by little, in an order designed to enable testing and debugging of each new part as soon as it is added. Often, test driver main() functions are written for the sole purpose of testing the parts (i.e., classes, when you use C++ or Java) you are going to write next. After a part or class is tested, debugged, and tested again, we can begin implementing a new class that USES it.

Thus, part 1 of this project begins by writing a declaration, test driver and then an implementation of a class (named peg) whose instances simulate the single “pegs” in the Tower of Hanoi game. Part 2 will develop a class (named towers) for which one instance will be the core of of Tower of Hanoi game software to enable a user (“child”) to play this form of solitaire. The test driver for part 2 will interact with the user to play the game. The class towers will of course USE peg.

4 Sample Class Design and Implementation Plan Outline

Tip: Check the check boxes as you proceed through the first part of the first project, to get the idea of how to work this way. In some professional programming groups and for big enough programs, management will demand a report at each step!

1. □ Study the assignment (specification) and sketch (perhaps on paper):
   (a) □ The name of the class (e.g., throttle).
   (b) □ Names of public member functions, notes clarifying what each does, what arguments and return values it should have. Perhaps sketch diagrams.
(c) REVIEW your sketchwork and the assignment to correct, improve and clarify the sketch. Ask yourself if the member names suggest their purpose as clearly as possible. If not, change them. For example, the throttle’s shift function is pretty well-chosen; to name it set would be misleading (why?)

2. Type the header file (e.g., throttle.h) to contain formal documentation and the class definition.
   (a) Document each public member function with its name, prototype and (IMPORTANT!) comments in the form of pre and post-conditions to document what each public member function is supposed to do.
   (b) Write the class definition’s public member function declarations only (no private members yet!)

3. Create a skeleton implementation file for the class (e.g. throttle.cxx) in which the bodies of member functions are stubs.
   (a) A stub is trivial function body that might be empty, or perhaps more usefully, prints a message that it is called. It might also print its arguments.
   (b) Make it return some reasonable constant that’s trivial to express, if its return type is not void.
   (c) The skeleton implementation file MUST have directive to “include” the classes header file. For example:
      #include "throttle.h"
   (d) Try to compile this implementation file. Correct this implementation file and/or header file as necessary until it compiles without any error or warning messages.
   (e) The implementation file compiles without error or warning messages.

4. Write a test driver (like demo1.cxx or demo2.cxx).
   (a) The test driver uses the class you are developing, so it MUST have a #include preprocessor directive to “include” the classes header file. That directive is identical to the one in the header file! (Just copy it–copy and paste makes that easy and error-free).
   (b) Try to compile the test driver.
      Correct it until it compiles.
      This step might show you something was wrong with the declarations in the header file. If so, correct the header file; but, if you do, (IMPORTANT) try to compile the stub implementation file again! Make corrections there too if necessary.
   (c) Try to link the test driver and class implementation object files. E.g. with the command “g++ -o demo2 demo2.o throttle.o”
      i. Edit and recompile and relink all until it succeeds.
ii. See if the test driver runs as you would expect with the stub implementations of the tested class. In other words, begin by *testing the tester!* Debug, recomile, relink till it’s right.

(d) NOW (finally!) begin coding the private data member declarations according to your data structure design choices, and implement the member functions.

i. Code the data members first.

ii. Code one or more constructors: Recompile, relink, and debug until you verify the constructors work.

iii. Replace the stubs (one or a few at a time) with real implementations. Recompile, relink and use the test driver to test what you have implemented as soon as possible.

iv. Choose new stubs to replace in an order that enables you to use previously written and tested functions to test the new ones.