Assignment #7: Solitaire

Due: Wednesday, November 22 at 5:00 p.m.

In this assignment, your job is to write a program that plays the solitaire game called Klondike, which is almost certainly the best-known solitaire card game in the United States. As an assignment, Klondike has two purposes:

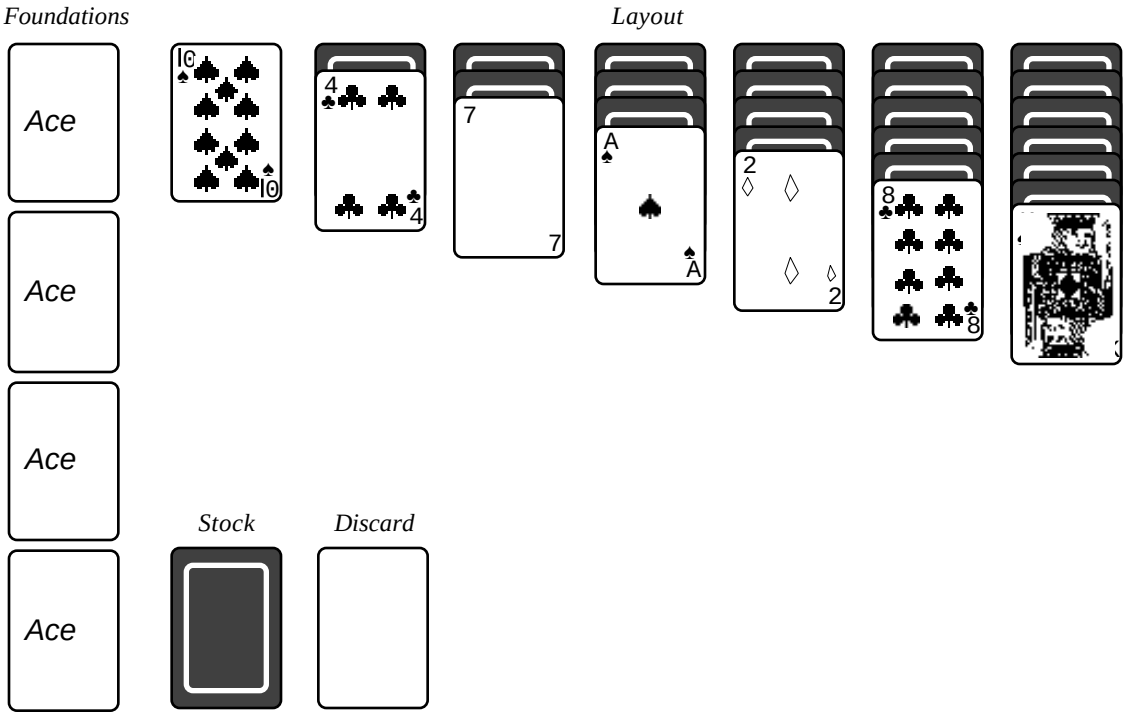
1. It gives you an opportunity to work with list structures in an application that is more interesting than the editor buffer.
2. It anticipates the topic of object-oriented programming, which is a paradigm popular in modern computer science and you will certainly encounter if you continue taking computer science courses. The assignment is designed so that the individual cards take center stage, and the game is organized as operations on cards. Cards, for example, can attach themselves to other cards and respond to being dragged with the mouse.

Approaching this problem from the perspective of the cards has two principal advantages. First, doing so reduces the complexity of the code. As you will see as you read the rest of the handout, the cards keep track of state information within their own structure, which means that there is little or no global state to maintain. Second, the overall structure becomes more flexible. The predefined operations on cards are independent of the specific rules of Klondike, which makes it much easier to implement other forms of solitaire.

The game of Klondike

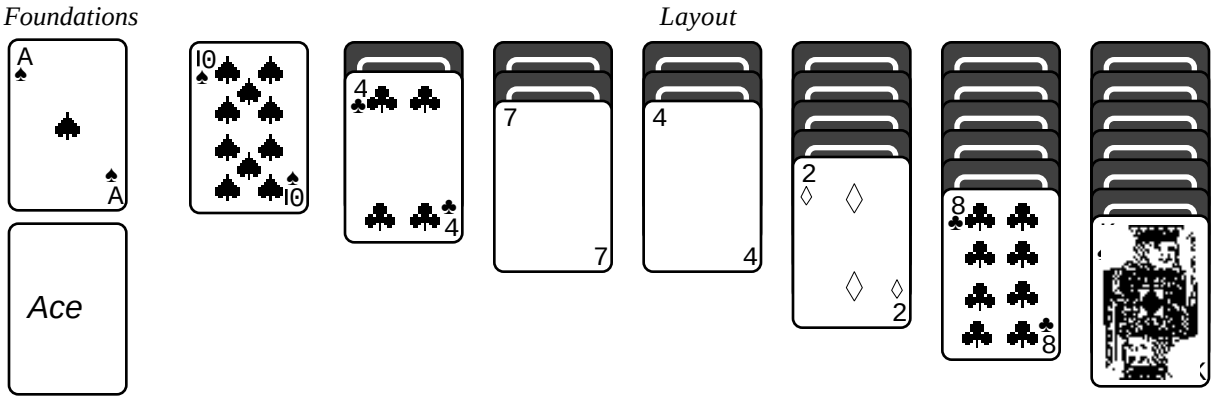
Klondike is played with a standard deck of 52 cards, which is dealt as shown in Figure 1.

Figure 1. Initial Klondike layout

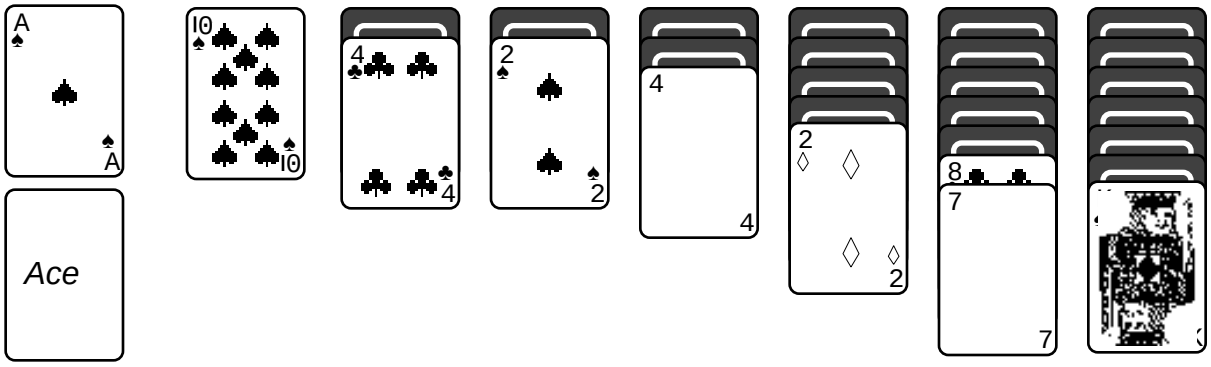


The first set of cards is dealt in a triangular pattern that forms the **layout.** The top row in the layout consists of seven cards, the next one contains six, and so forth, until only a single card is dealt in the last row. The first card in each row is dealt face up, but all of the others remain face down until they are exposed during play. Even though the layout is conventionally dealt in rows, it is more useful to think of it as a set of vertical piles. In Figure 1, for example, the first pile contains the ten of spades, the second contains the four of clubs on top of one face-down card, and so forth. After dealing the layout cards, the remainder of the deck is placed face down in front of you forming what Hoyle's *Rules of Games* calls the **stock.** Figure 1 also shows several other areas that are initially empty but which come into play later in the game: four **foundation** piles along the left side and a **discard pile** next to the stock.

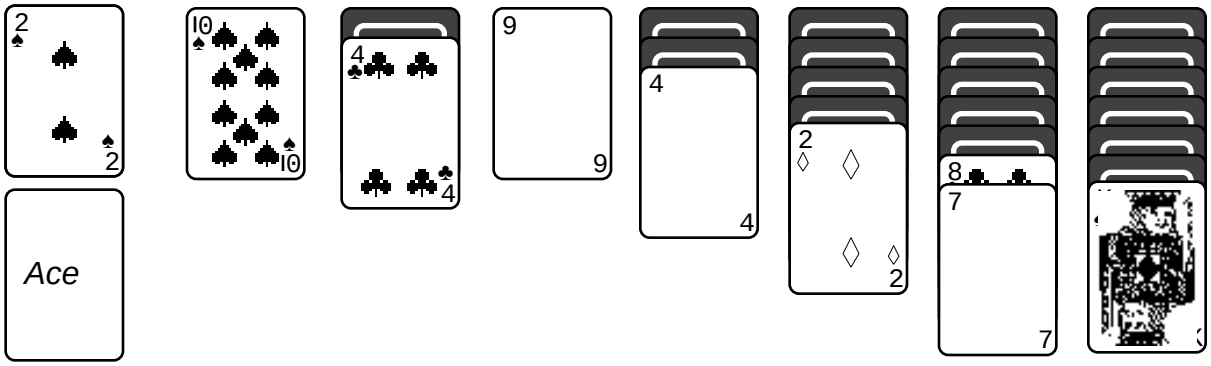
The object of the game is to move all 52 cards in the deck to the foundation piles. Each foundation must begin with an ace and then builds up from there in ascending order by suit. For example, given the initial configuration shown in Figure 1, the ace of spades can be moved immediately to a foundation pile. It doesn't matter which one initially, but after you place the ace of spades in one of the foundation piles, that pile can thereafter contain only spades. Whenever you move a card from the layout that exposes a face-down card, that card is then turned face up and is available for play. Thus, if you played the ace of spades and turned up the four of hearts, the layout and the top two foundation piles would look like this:



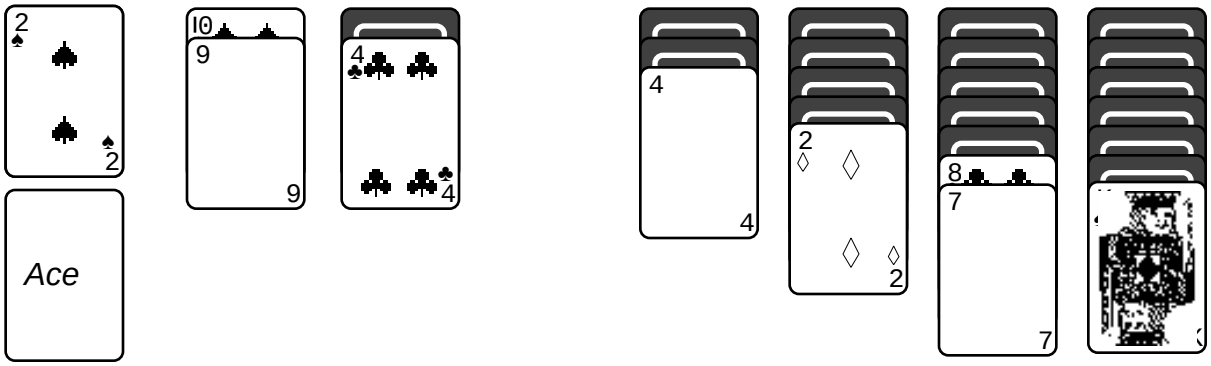
You can also make plays on the layout itself. Here, cards are played to form descending chains that alternate in color (the black-and-white pictures in the handout don't make things as clear as the sample application, but spades and clubs are black, while hearts and diamonds are red). In this layout, the seven of hearts can be played on the eight of clubs to reach this position:



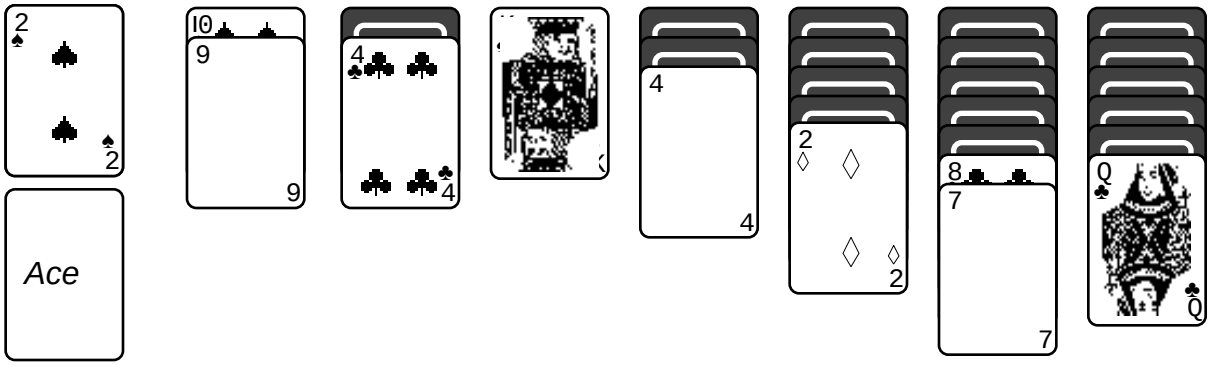
From here, you can play the recently exposed two of spades up to the foundation pile that already contains the ace of spades, thereby exposing the last card in pile number 2, as follows:



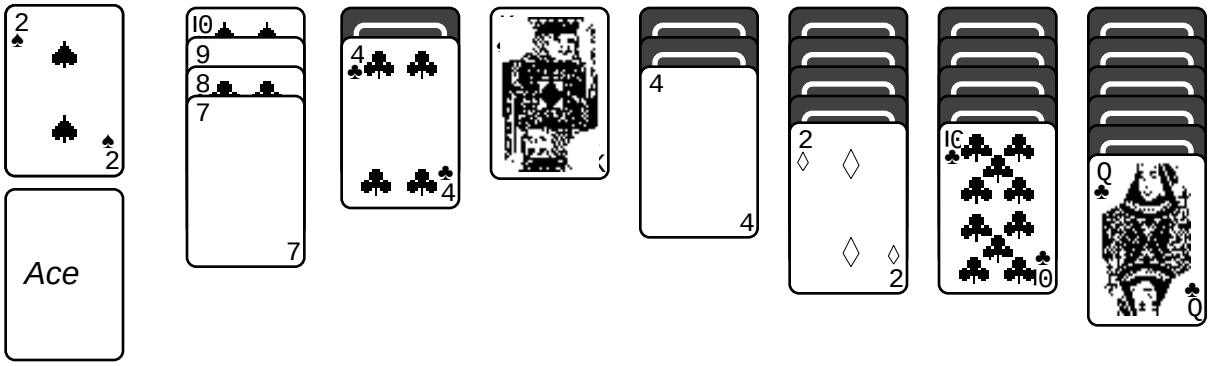
The nine of hearts plays on the ten of spades, which leaves an empty pile in the layout, because there is no face-down card underneath it:



Empty spaces in the layout may be filled by any king. Thus, the king of spades in pile number 6 can be moved into this space, as shown:



The last remaining move on the layout is that you can play the eight of clubs on the nine of hearts. This play is consistent with the rule that cards on the layout form descending chains in alternate colors. When you do so, however, you need to move the entire rest of the chain below the card you're moving. Thus, the seven of hearts comes along for the ride to yield the following configuration:



From here, there are no useful plays to make on the layout alone (you could move the chain beginning with the nine of hearts over to the ten of clubs, but it doesn't help anything). At this point, play continues by turning over cards, three at a time, from the stock onto the discard pile. The top card on the discard pile is then available for play and may be moved to the foundations or the layout in accordance with the rules already given for playing in those areas. For example, if you turned up the three of spades or any of the missing aces, you could play that card to the foundations. Similarly, you could play the queen of hearts on the king of spades. Playing the top discard reveals the card underneath, which then becomes playable.

Eventually, you will run out of cards in the stock. At that point, you are allowed to turn the entire discard pile over and make that a new stock. You can then continue turning three cards at a time from the stock as before. You can perform the operation of turning the discard pile back into the stock as many times as you want. Play ends when all 52 cards have been successfully played onto the foundation piles or when you decide to quit. Quitting is accomplished in the sample application by clicking in the console window and typing Command-period.

Overview of the module structure

Your job in this assignment is to write a program that plays Klondike, just like the sample application does. The finished project will consist of the following modules, all but two of which are given to you (the ones you have to write are underlined):

- **`solitaire.c`**—This is the main program module and contains nothing except for the following main program:

```
main()
{
    InitGraphics();
    InitCardDisplay("Light gray");
    Randomize();
    InitGame();
    while (TRUE) WaitForCardAction();
}
```

  Four of the statements in the implementation of **`main`** are used for initialization. The entire play of the game takes place in the last line, which repeatedly calls the function **`WaitForCardAction`**. This function monitors the motion of the mouse waiting for

the user to complete a legal action, which always consists of clicking on a card or dragging one card on top of a second one. When the action is complete, **WaitForCardAction** calls a function associated with the card being dragged which implements the operation, as described later in the handout. Object-oriented designs for interactive programs usually contain a loop like

```
while (TRUE) WaitForCardAction();
```

which is called an **event loop.**

- **klondike.c**—The **klondike** module is responsible for all aspects of the game that are specific to Klondike. The only function this module exports is the **InitGame** function, which is called by the main program to initialize the game. The task of initializing the game is discussed in more detail later in this handout, but the process consists primarily of arranging the cards in their initial positions on the display. As part of creating that initial configuration, one of the responsibilities of the **InitGame** function is to establish the behavior for each card. Because this is an object-oriented design, there is no game-level code that knows how to move cards around. In essence, the cards move themselves around. The internal data structure for each card contains a function that allows it to respond to being dragged from one place to another. Depending on whether the card is in the stock, the layout, the discard pile, or the foundations, the function that defines the behavior will be different. It is therefore up to the InitGame function to make sure that all cards start with functions that are appropriate to the location of the card.
- **cards.c**—This file implements the **cards.h** interface, which exports an abstract **cardADT** type along with a reasonably large collection of functions for manipulating values of that type. The detailed structure of this interface is discussed in the next section.
- **gcards.c**—This module is responsible for drawing cards on the display and monitoring the motion of the mouse. It exports three functions: **DrawCard**, **EraseCard**, and **WaitForCardAction**. The first two are called from the **cards.c** implementation when you need to change the position of a card on the screen. The **WaitForCardAction** function is quite simple in its operation. This module is supplied to you in source form, but you shouldn't need to look at it in any detail.
- **deck.c**—This module exports a **deckADT**, which is a collection of cards that can be shuffled and dealt one at a time. In addition to the **deckADT** type, the **deck.h** interface exports the functions **NewDeck**, **ShuffleDeck**, and **DealNextCard**. The usual pattern of use is illustrated by the following code:

  ```
  deck = NewDeck(52);
  ShuffleDeck(deck);
  while ((card = DealNextCard(deck)) != NULL) { . . . }
  ```

  The **deck.c** implementation is simple enough to write, but doesn't add much to the assignment. I decided to provide it to you mostly so that you don't have to agonize over how to shuffle a deck of cards.

Mac users - The only file whose purpose will not be immediately clear is the file **solitaire.π.rsrc**. This file contains the resources for the program, which are mostly the images of the playing cards. You must keep this file in the same folder as the project and its name must consist of the name of the project file with the additional extension **.rsrc**.

PC users - You will notice an extra directory named **Bitmaps** and a file named **SolitaireCards.rc** included with the .zip file. Be sure to have both the **Bitmaps** directory and **SolitaireCards.rc** inside your project folder, and add **SolitaireCards.rc** to your VC++ project just like any source file. Also included is

chord.wav, which the sample app uses to make the "beeping" noise caused by a bad card move.

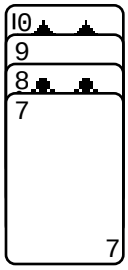
As in the Darwin assignment, we have created library files for the two modules you have to write, which means that you start with a working implementation. Your goal for the assignment is to replace the **klondike.lib** and **cards.lib** modules with implementations of your own.

The cards module

Before you can understand how to write the **klondike.c** file, you need to understand the structure of the **cards.h** interface. Given that it exports five types, eight constants, and 19 functions, the **cards.h** interface is fairly long. Even so, this module is likely to be easier to write than the **klondike** module, because the function implementations tend to be quite short. Most of them can be implemented with just a line or two of code. The complexity in this module is largely a matter of understanding the interface, although you may find one or two of the functions challenging on a conceptual level.

The principal type exported by the **cards.h** interface is the abstract type **cardADT**. Internally, a **cardADT** keeps track of the following properties:

- *Rank.* The rank of the card is represented by the type **rankT**, which is defined to be equivalent to **int**. To make it easier to refer to face cards, the interface exports constants for the ranks **Ace**, **Jack**, **Queen**, and **King**.
- *Suit.* The suit is represented by the enumeration type **suitT**, which consists of the constants **Clubs**, **Diamonds**, **Hearts**, and **Spades**.
- *Orientation.* Each card is in one of two orientations: **FaceUp** or **FaceDown**. The orientation determines how the card is drawn and whether it is available for play.
- *Position.* When a card is displayed on the screen, it is positioned at some *x/y* location in the graphics window. As with most graphics functions, the *x* and *y* values specify the coordinates of the lower left corner of the card. To help you set up the graphical arrangement, the **cards.h** interface exports constants **CardWidth** and **CardHeight**. If you want a card to exist, but not to be drawn on the screen, you can set its *x* and *y* coordinates to the constant **OffScreen**, which is also defined in this interface.
- *Attachments.* Cards can be attached to other cards to form linear chains. For example, each pile in the layout represents a chain of cards. So do the collections of cards on the stock, the discard pile, and each of the foundations. The internal data structure for the cards must maintain the attachment information, presumably in the form of a doubly linked list. The process of creating attachments is described in more detail in the section entitled "The **Attach** function" later in the handout. Given a card, you can determine the cards to which it is attached by calling **GetPredecessor** and **GetSuccessor**. The interface also exports **GetBottomCard** and **GetTopCard**, which find the card at the end of a chain. The "top" and "bottom" directions are determined by the stacking arrangement of the cards rather than by their *y* coordinate. Thus, in the chain



the seven of hearts is the top card.[1] Its predecessor is the eight of clubs, and it has no successor in the chain, which means that **GetSuccessor** would return **NULL**.

- *Action function.* The interactivity of the game is controlled by the event loop in the main program, which repeatedly calls the **WaitForCardAction** function in the **gcards** module. A legal action always consists of having the mouse go down on top of a *source* card and come up again on top of a *destination* card. If the cards are different, this operation represents dragging the source card to the destination card. If the cards are the same, the user just clicked on that card. In either case, the action taken by **WaitForCardAction** is to call an action function associated with the source card, passing both the source and destination cards as parameters. This action function is set individually for each card and is represented as a function pointer in the following class:

```
typedef void (*cardActionFnT)(cardADT src, cardADT
dst);
```

  The functions that are stored with each card implement the entire play of the game and are discussed in more detail in the section on the **klondike** module later in this handout.

- *Client data.* The client can associate an arbitrary data pointer with each card using the functions **SetCardData** and **GetCardData**. These functions make it possible to eliminate the need for global state in the program as a whole.

The majority of the functions in the interface simply get or set one of these properties. As an example, if you assume that the concrete structure for the card includes an **orientation** field, the implementations of the functions **GetCardOrientation** and **SetCardOrientation** look like this:

```
orientationT GetCardOrientation(cardADT card)
{
    return (card->orientation);
}

void SetCardOrientation(cardADT card, orientationT
orientation)
{
    card->orientation = orientation;
    DrawCard(card);
}
```

Note that the **SetCardOrientation** function must call **DrawCard** to update the display, because cards are drawn differently depending on their orientation.

The `Attach` function

Most of the functions in the **cards.c** implementation are no more complex than the **GetCardOrientation** and **SetCardOrientation** functions in the preceding section. The only function that is considerably more complicated is the **Attach** function, which makes it possible to cards to another to form linear chains. The **Attach** function has the following prototype:

```
void Attach(cardADT src, cardADT dst, double dx,
            double dy);
```

---

[1] As discussed in the section on "Special cards" later in this handout, the ten of spades is probably not the bottom card in this chain. In the suggested game design, there is an invisible card at the bottom of this pile that holds the space into which a king might be placed. The ten of spades is attached to this special card.

This function attaches the cards **src** and **dst** so that **src** becomes the successor of **dst** and **dst** becomes the predecessor of **src**. If these cards were previously part of other chains, those chains are broken at that point. No changes are made to the rest of the chains in which these cards appear. The **dx** and **dy** parameters indicate the new position of **src** relative to the coordinates of **dst**. For example, the call

```
Attach(src, dst, 0, 0);
```

draws the **src** card immediately on top of the **dst** one. You would use these values, for example, to put new cards on the foundations or the stock. To move a card onto an existing chain in the layout, however, the new card is offset in the *y* direction so that the cards partially overlap. In the sample application, the call is

```
Attach(src, dst, 0, -LayoutOffset);
```

where **LayoutOffset** has the following definition:

```
#define LayoutOffset 0.17
```

Note that **Attach** must move and redraw not only the **src** card, but every card in its successor chain as well. The new coordinates of these other cards should not change in relation to the coordinates of **src**, which means that the entire chain effectively moves as a unit. Moreover, must redraw any card to which **src** used to be attached, since that card is now exposed. Calling **Attach** therefore requires the following calls to the **gcards.h** interface:

1. Call **EraseCard** on **src** and every card in its successor chain to erase them from their old positions.
2. After repositioning the cards, call **DrawCard** on **src** and every card in its successor chain to redisplay them in their new positions.
3. Call **DrawCard** on the old predecessor of **src**, if any, to expose it to view.

Special cards

Moving to the object-oriented design creates an interesting wrinkle in the programming of the solitaire game. All actions in this model are defined to be interactions between two cards. For many of the actions that occur in the game, this interpretation is fine. For example, if you are moving a card from the discard pile to the layout or between two layout piles, there is an obvious source and destination card. What happens, however, if you want to move a king into an empty space or an ace onto an empty foundation? In such circumstances, what is the destination card?

There are many strategies that you could use to address this problem. One of the simplest—mostly because it maintains a symmetrical object-oriented structure—is to define a set of "special" cards that serve as placeholders on the screen. If you create, for example, a special card that acts as the base of a foundation, the operation of moving an ace to an empty foundation pile can use that special card as the destination.

The **cards.h** module includes an entry for creating a special card, which has the following prototype:

```
cardADT NewSpecialCard(string name);
```

The name of the card is used to define its display characteristics and must match an entry in the resource file for the project. The **solitaire.π.rsrc** file provided with the starter folder defines the special card names **StockBase**, **FoundationBase**, **LayoutBase**, and **DiscardBase**, each of which serves as the base of its respective pile.

The `klondike` module

The **`klondike.c`** module is responsible for everything having to do with the specific play of the game. In this module, you have to implement the function **`InitGame`**, which creates the initial state of the game, along with a set of action functions that define how cards behave when they are dragged on the display. The **`InitGame`** function has the following outline:

1. Use the **`deck.h`** interface to create and shuffle a standard deck of cards.
2. Create special cards to represent the base of each pile (four foundation piles, seven layout piles, the stock pile, and the discard pile) and position them at the appropriate places on the screen.
3. Deal the cards from the deck into their appropriate places in the layout and stock by attaching them to the proper chain. As part of this operation, you will also have to take care of setting the correct orientation for each card (**`FaceUp`** or **`FaceDown`**) and initializing the appropriate action function.[2]

The bulk of the code in **`klondike.c`** comes in the action functions, each of which has a prototype that looks like this:

```
void ActionName(cardADT src, cardADT dst);
```

By setting the action function, you can control the behavior of a card as it is dragged around the screen. Since cards behave differently depending upon what pile they're in, you need to set the action function depending on where the card is at any particular point in time. For example, cards in the discard pile might use a function called **`DiscardPileAction`**, which takes care of what happens if you drag the top card in the discard pile somewhere else on the screen. By contrast, cards that are moved onto the foundations are out of play and never move again. Thus, the action function for a card in the foundation pile might be as simple as this:

```
void FoundationPileAction(cardADT src, cardADT dst)
{
    printf("\a");
}
```

This function ignores the cards entirely and simply issues a "beep" sound, which is the effect of printing the special character **`'\a'`** that represents an audible alert.

[2] Although steps 2 and 3 are listed separately here, it makes more sense to interleave them. For example, to create a layout chain, the easiest approach is to create the base of the chain and then deal cards directly on top of it.

In the more general case, the action functions will typically have the following structure:

1. Determine the type of pile represented by the destination card by looking at the special card at the bottom of the chain.
2. Check whether the move is legal, which depends not only on the source and destination cards, but also on the pile types. If you are moving a card to a foundation, the two cards must be the same suit and ascending in sequence. If you are moving a card to the layout, the cards must be of opposite colors and descending in sequence. These rules must be coded as part of the action function.
3. If the move is illegal, the easiest thing to do is print the audible alert character so that the computer beeps.
4. If the move is legal, the usual response is to attach the source card to the destination card. For some plays, however, you may need to perform other operations such as turning over an exposed layout card.

The only special cases that need to be considered here are the process of turning over cards from the stock into the discard pile or flipping the entire discard pile to replenish the stock. Computationally, these operations are not at all hard and just involve attaching cards to the right pile. The only issue is how to get access to the source and destination information. In the sample application, clicking on the source pile is sufficient to move three cards to the discard pile; you don't need to drag them there. What this means is that the action function for the cards in the stock is called with both **`src`** and **`dst`** indicating the same card, which is the one on top of the stock. You want to take this card, and the two cards underneath it, and turn them over one at a time onto the discard pile.

Unfortunately, if all you have is the source card, it is not immediately clear how you can find the top of the discard pile to perform the necessary **`Attach`** operation. The easiest way to solve the problem is to use the **`SetCardData`** and **`GetCardData`** functions to store the special card forming the discard base as the data field of the special card forming the base of the stock. If you do so, you can get from the top card in the stock to the top card in the discard pile by calling the appropriate sequence of **`GetCardData`**, **`GetTopCard`**, and **`GetBottomCard`**. Using this facility, you should be able to implement both the **`cards.c`** and **`klondike.c`** module without using any global variables.

Summary of the rules of play

Your program should make sure that each of the following rules of play is observed during the play:

1. All play on the foundations is in increasing order by rank. Any ace may be moved into an empty foundation pile, but after that, only cards of that suit will be allowed. Only individual cards may be moved to the foundations.
2. Any card played on the layout must be opposite in color and one step less in rank than the card to which it is moved (red twos go on black threes, for example).
3. Kings may be moved only to blank spaces in the layout, and no other card may be moved into a blank space.
4. Cards can be moved only to exposed cards in the layout (or blank spaces in the case of kings). Thus, the destination card in a valid play to the layout must be the top card in its chain and not a card with other cards attached on top of it.
5. Chains of cards may be moved as a unit within the layout, but only if the top card represents a legal play according to rules 2 to 4. If a card is moved that has other cards attached beneath it, those cards all move as well.
6. If moving a card from the layout uncovers a face-down card, that card is immediately turned over as part of the play.

7. On any play, the user may turn three cards over from the stock by clicking on it (both the source and destination fields in the command specify the stock). The three cards are not reordered as they are played but are simply flipped over so that the third card in the stock becomes the top card on the discard pile. If the stock has fewer than three cards in it, the entire rest of the stock is flipped into the discard pile.
8. Only the top card in the discard pile is available for play. If it is played, the next card down in the discard pile becomes available until the pile is exhausted.
9. When the stock is exhausted, clicking on it has the effect of flipping over the entire discard pile to become a new stock. During the game, the user can go through the stock as many times as necessary.

Possible extensions

As usual, there are many extensions you could add to this program to make it more of a challenge, including the following:

- *Extend the command structure.* As it stands, user control over the game is quite limited. To quit, for example, it is necessary to use the menus because a quit button did not fit so symmetrically into the **`WaitForCardAction`** logic. More importantly, there are many shortcut operations that would be great for the user. For example, if you click on a card (as opposed to dragging it), you might want the program to play it on a foundation if it could. You can detect a click on a card by checking if the source and destination cards match. You can define operations by creating new special cards and drawing them somewhere on the screen.
- *Make it possible for the computer to play.* The main program shown in this handout just makes moves in response to user commands. Can you write code so that the computer itself chooses what plays to make?
- *Add a flashier ending.* When you win the PC version of this game, the cards jump out off the foundations one at a time and cascade down the screen, leaving a trail behind.
- *Implement an "undo" feature.* Provide some way for taking back the last move or, better yet, backing up through the whole game.
- *Implement other, more interesting solitaire games.* There are many solitaire games that are more fun than Klondike. I've provided two examples for games that I call "Sixteen" and "Big Ten." These games are implemented as the library files **`sixteen.lib`** and **`bigten.lib`**. To run one of these games, all you have to do is take out the **`klondike.lib`** file and substitute one of the others. Rules for these games are available by clicking on the **`Help`** button displayed on the screen.